Guile Reference Manual

Edition 3.0.4, revision 1, for use with Guile 3.0.4

The Guile Developers

This manual documents Guile version 3.0.4.

Table of Contents

Preface

This manual describes how to use Guile, GNU's Ubiquitous Intelligent Language for Extensions. It relates particularly to Guile version 3.0.4.

Contributors to this Manual

Like Guile itself, the Guile reference manual is a living entity, cared for by many people over a long period of time. As such, it is hard to identify individuals of whom to say "yes, this single person wrote the manual."

Still, among the many contributions, some caretakers stand out. First among them is Neil Jerram, who has worked on this document for over ten years. Neil's attention both to detail and to the big picture have made a real difference in the understanding of a generation of Guile hackers.

Next we should note Marius Vollmer's effect on this document. Marius maintained Guile during a period in which Guile's API was clarified—put to the fire, so to speak—and he had the good sense to effect the same change on the manual.

Martin Grabmueller made substantial contributions throughout the manual in preparation for the Guile 1.6 release, including filling out a lot of the documentation of Scheme data types, control mechanisms and procedures. In addition, he wrote the documentation for Guile's SRFI modules and modules associated with the Guile REPL.

Ludovic Courtès and Andy Wingo, who co-maintain Guile since 2010, along with Mark Weaver, have also made their dent in the manual, writing documentation for new modules and subsystems that arrived with Guile 2.0. Ludovic, Andy, and Mark are also responsible for ensuring that the existing text retains its relevance as Guile evolves. See Section 2.6 [Reporting Bugs], page 12, for more information on reporting problems in this manual.

The content for the first versions of this manual incorporated and was inspired by documents from Aubrey Jaffer, author of the SCM system on which Guile was based, and from Tom Lord, Guile's first maintainer. Although most of this text has been rewritten, all of it was important, and some of the structure remains.

The manual for the first versions of Guile were largely written, edited, and compiled by Mark Galassi and Jim Blandy. In particular, Jim wrote the original tutorial on Guile's data representation and the C API for accessing Guile objects.

Significant portions were also contributed by Thien-Thi Nguyen, Kevin Ryde, Mikael Djurfeldt, Christian Lynbech, Julian Graham, Gary Houston, Tim Pierce, and a few dozen more. You, reader, are most welcome to join their esteemed ranks. Visit Guile's web site at `http://www.gnu.org/software/guile/` to find out how to get involved.

The Guile License

Guile is Free Software. Guile is copyrighted, not public domain, and there are restrictions on its distribution or redistribution, but these restrictions are designed to permit everything a cooperating person would want to do.

- The Guile library (libguile) and supporting files are published under the terms of the GNU Lesser General Public License version 3 or later. See the files `COPYING.LESSER` and `COPYING`.

- The Guile readline module is published under the terms of the GNU General Public License version 3 or later. See the file `COPYING`.
- The manual you're now reading is published under the terms of the GNU Free Documentation License (see Appendix A [GNU Free Documentation License], page 877).

C code linking to the Guile library is subject to terms of that library. Basically such code may be published on any terms, provided users can re-link against a new or modified version of Guile.

C code linking to the Guile readline module is subject to the terms of that module. Basically such code must be published on Free terms.

Scheme level code written to be run by Guile (but not derived from Guile itself) is not restricted in any way, and may be published on any terms. We encourage authors to publish on Free terms.

You must be aware there is no warranty whatsoever for Guile. This is described in full in the licenses.

1 Introduction

Guile is an implementation of the Scheme programming language. Scheme (`http://schemers.org/`) is an elegant and conceptually simple dialect of Lisp, originated by Guy Steele and Gerald Sussman, and since evolved by the series of reports known as RnRS (the Revised^n Reports on Scheme).

Unlike, for example, Python or Perl, Scheme has no benevolent dictator. There are many Scheme implementations, with different characteristics and with communities and academic activities around them, and the language develops as a result of the interplay between these. Guile's particular characteristics are that

- it is easy to combine with other code written in C
- it has a historical and continuing connection with the GNU Project
- it emphasizes interactive and incremental programming
- it actually supports several languages, not just Scheme.

The next few sections explain what we mean by these points. The sections after that cover how you can obtain and install Guile, and the typographical conventions that we use in this manual.

1.1 Guile and Scheme

Guile implements Scheme as described in the Revised^5 Report on the Algorithmic Language Scheme (usually known as R5RS), providing clean and general data and control structures. Guile goes beyond the rather austere language presented in R5RS, extending it with a module system, full access to POSIX system calls, networking support, multiple threads, dynamic linking, a foreign function call interface, powerful string processing, and many other features needed for programming in the real world.

In 2007, the Scheme community agreed upon and published R6RS, a significant installment in the RnRS series. R6RS expands the core Scheme language, and standardises many non-core functions that implementations—including Guile—have previously done in different ways. Over time, Guile has been updated to incorporate almost all of the features of R6RS, and to adjust some existing features to conform to the R6RS specification. See Section 7.6 [R6RS Support], page 662, for full details.

In parallel to official standardization efforts, the SRFI process (`http://srfi.schemers.org/`) standardises interfaces for many practical needs, such as multithreaded programming and multidimensional arrays. Guile supports many SRFIs, as documented in detail in Section 7.5 [SRFI Support], page 582.

The process that led to the R6RS standard brought a split in the Scheme community to the surface. The implementors that wrote R6RS considered that it was impossible to write useful, portable programs in R5RS, and that only an ambitious standard could solve this problem. However, part of the Scheme world saw the R6RS effort as too broad, and as having included some components that would never be adopted by more minimalistic Scheme implementations. This second group succeeded in taking control of the official Scheme standardization track and in 2013 released a more limited R7RS, essentially consisting of R5RS, plus a module system. Guile supports R7RS also. See Section 7.7 [R7RS Support], page 704.

With R6RS and R7RS, the unified Scheme standardization process appears to have more or less run its course. There will continue to be more code written in terms of both systems, and modules defined using the SRFI process, and Guile will support both. However for future directions, Guile takes inspiration from other related language communities: Racket, Clojure, Concurrent ML, and so on.

In summary, Guile supports writing and running code written to the R5RS, R6RS, and R7RS Scheme standards, and also supports a number of SRFI modules. However for most users, until a need for cross-implementation portability has been identified, we recommend using the parts of Guile that are useful in solving the problem at hand, regardless of whether they proceed from a standard or whether they are Guile-specific.

1.2 Combining with C Code

Like a shell, Guile can run interactively—reading expressions from the user, evaluating them, and displaying the results—or as a script interpreter, reading and executing Scheme code from a file. Guile also provides an object library, *libguile*, that allows other applications to easily incorporate a complete Scheme interpreter. An application can then use Guile as an extension language, a clean and powerful configuration language, or as multi-purpose "glue", connecting primitives provided by the application. It is easy to call Scheme code from C code and vice versa, giving the application designer full control of how and when to invoke the interpreter. Applications can add new functions, data types, control structures, and even syntax to Guile, creating a domain-specific language tailored to the task at hand, but based on a robust language design.

This kind of combination is helped by four aspects of Guile's design and history. First is that Guile has always been targeted as an extension language. Hence its C API has always been of great importance, and has been developed accordingly. Second and third are rather technical points—that Guile uses conservative garbage collection, and that it implements the Scheme concept of continuations by copying and reinstating the C stack—but whose practical consequence is that most existing C code can be glued into Guile as is, without needing modifications to cope with strange Scheme execution flows. Last is the module system, which helps extensions to coexist without stepping on each others' toes.

Guile's module system allows one to break up a large program into manageable sections with well-defined interfaces between them. Modules may contain a mixture of interpreted and compiled code; Guile can use either static or dynamic linking to incorporate compiled code. Modules also encourage developers to package up useful collections of routines for general distribution; as of this writing, one can find Emacs interfaces, database access routines, compilers, GUI toolkit interfaces, and HTTP client functions, among others.

1.3 Guile and the GNU Project

Guile was conceived by the GNU Project following the fantastic success of Emacs Lisp as an extension language within Emacs. Just as Emacs Lisp allowed complete and unanticipated applications to be written within the Emacs environment, the idea was that Guile should do the same for other GNU Project applications. This remains true today.

The idea of extensibility is closely related to the GNU project's primary goal, that of promoting software freedom. Software freedom means that people receiving a software package can modify or enhance it to their own desires, including in ways that may not have

occurred at all to the software's original developers. For programs written in a compiled language like C, this freedom covers modifying and rebuilding the C code; but if the program also provides an extension language, that is usually a much friendlier and lower-barrier-of-entry way for the user to start making their own changes.

Guile is now used by GNU project applications such as AutoGen, Lilypond, Denemo, Mailutils, TeXmacs and Gnucash, and we hope that there will be many more in future.

1.4 Interactive Programming

Non-free software has no interest in its users being able to see how it works. They are supposed to just accept it, or to report problems and hope that the source code owners will choose to work on them.

Free software aims to work reliably just as much as non-free software does, but it should also empower its users by making its workings available. This is useful for many reasons, including education, auditing and enhancements, as well as for debugging problems.

The ideal free software system achieves this by making it easy for interested users to see the source code for a feature that they are using, and to follow through that source code step-by-step, as it runs. In Emacs, good examples of this are the source code hyperlinks in the help system, and `edebug`. Then, for bonus points and maximising the ability for the user to experiment quickly with code changes, the system should allow parts of the source code to be modified and reloaded into the running program, to take immediate effect.

Guile is designed for this kind of interactive programming, and this distinguishes it from many Scheme implementations that instead prioritise running a fixed Scheme program as fast as possible—because there are tradeoffs between performance and the ability to modify parts of an already running program. There are faster Schemes than Guile, but Guile is a GNU project and so prioritises the GNU vision of programming freedom and experimentation.

1.5 Supporting Multiple Languages

Since the 2.0 release, Guile's architecture supports compiling any language to its core virtual machine bytecode, and Scheme is just one of the supported languages. Other supported languages are Emacs Lisp, ECMAScript (commonly known as Javascript) and Brainfuck, and work is under discussion for Lua, Ruby and Python.

This means that users can program applications which use Guile in the language of their choice, rather than having the tastes of the application's author imposed on them.

1.6 Obtaining and Installing Guile

Guile can be obtained from the main GNU archive site `ftp://ftp.gnu.org` or any of its mirrors. The file will be named guile-*version*.tar.gz. The current version is 3.0.4, so the file you should grab is:

`ftp://ftp.gnu.org/gnu/guile/guile-3.0.4.tar.gz`

To unbundle Guile use the instruction

```
zcat guile-3.0.4.tar.gz | tar xvf -
```

which will create a directory called `guile-3.0.4` with all the sources. You can look at the file `INSTALL` for detailed instructions on how to build and install Guile, but you should be able to just do

```
cd guile-3.0.4
./configure
make
make install
```

This will install the Guile executable `guile`, the Guile library `libguile` and various associated header files and support libraries. It will also install the Guile reference manual.

Since this manual frequently refers to the Scheme "standard", also known as R5RS, or the "Revised[5] Report on the Algorithmic Language Scheme", we have included the report in the Guile distribution; see Section "Introduction" in *Revised(5) Report on the Algorithmic Language Scheme*. This will also be installed in your info directory.

1.7 Organisation of this Manual

The rest of this manual is organised into the following chapters.

Chapter 2: Hello Guile!
A whirlwind tour shows how Guile can be used interactively and as a script interpreter, how to link Guile into your own applications, and how to write modules of interpreted and compiled code for use with Guile. Everything introduced here is documented again and in full by the later parts of the manual.

Chapter 3: Hello Scheme!
For readers new to Scheme, this chapter provides an introduction to the basic ideas of the Scheme language. This material would apply to any Scheme implementation and so does not make reference to anything Guile-specific.

Chapter 4: Programming in Scheme
Provides an overview of programming in Scheme with Guile. It covers how to invoke the `guile` program from the command-line and how to write scripts in Scheme. It also introduces the extensions that Guile offers beyond standard Scheme.

Chapter 5: Programming in C
Provides an overview of how to use Guile in a C program. It discusses the fundamental concepts that you need to understand to access the features of Guile, such as dynamic types and the garbage collector. It explains in a tutorial like manner how to define new data types and functions for the use by Scheme programs.

Chapter 6: Guile API Reference
This part of the manual documents the Guile API in functionality-based groups with the Scheme and C interfaces presented side by side.

Chapter 7: Guile Modules
Describes some important modules, distributed as part of the Guile distribution, that extend the functionality provided by the Guile Scheme core.

Chapter 8: GOOPS
Describes GOOPS, an object oriented extension to Guile that provides classes, multiple inheritance and generic functions.

1.8 Typographical Conventions

In examples and procedure descriptions and all other places where the evaluation of Scheme expression is shown, we use some notation for denoting the output and evaluation results of expressions.

The symbol '⇒' is used to tell which value is returned by an evaluation:

```
(+ 1 2)
⇒ 3
```

Some procedures produce some output besides returning a value. This is denoted by the symbol '⊣'.

```
(begin (display 1) (newline) 'hooray)
⊣ 1
⇒ hooray
```

As you can see, this code prints '`1`' (denoted by '⊣'), and returns `hooray` (denoted by '⇒').

2 Hello Guile!

This chapter presents a quick tour of all the ways that Guile can be used. There are additional examples in the `examples/` directory in the Guile source distribution. It also explains how best to report any problems that you find.

The following examples assume that Guile has been installed in `/usr/local/`.

2.1 Running Guile Interactively

In its simplest form, Guile acts as an interactive interpreter for the Scheme programming language, reading and evaluating Scheme expressions the user enters from the terminal. Here is a sample interaction between Guile and a user; the user's input appears after the `$` and `scheme@(guile-user)>` prompts:

```
$ guile
scheme@(guile-user)> (+ 1 2 3)                ; add some numbers
$1 = 6
scheme@(guile-user)> (define (factorial n)    ; define a function
                       (if (zero? n) 1 (* n (factorial (- n 1)))))
scheme@(guile-user)> (factorial 20)
$2 = 2432902008176640000
scheme@(guile-user)> (getpwnam "root")        ; look in /etc/passwd
$3 = #("root" "x" 0 0 "root" "/root" "/bin/bash")
scheme@(guile-user)> C-d
$
```

2.2 Running Guile Scripts

Like AWK, Perl, or any shell, Guile can interpret script files. A Guile script is simply a file of Scheme code with some extra information at the beginning which tells the operating system how to invoke Guile, and then tells Guile how to handle the Scheme code.

Here is a trivial Guile script. See Section 4.3 [Guile Scripting], page 41, for more details.

```
#!/usr/local/bin/guile -s
!#
(display "Hello, world!")
(newline)
```

2.3 Linking Guile into Programs

The Guile interpreter is available as an object library, to be linked into applications using Scheme as a configuration or extension language.

Here is `simple-guile.c`, source code for a program that will produce a complete Guile interpreter. In addition to all usual functions provided by Guile, it will also offer the function `my-hostname`.

```
#include <stdlib.h>
#include <libguile.h>
```

```
static SCM
my_hostname (void)
{
  char *s = getenv ("HOSTNAME");
  if (s == NULL)
    return SCM_BOOL_F;
  else
    return scm_from_locale_string (s);
}

static void
inner_main (void *data, int argc, char **argv)
{
  scm_c_define_gsubr ("my-hostname", 0, 0, 0, my_hostname);
  scm_shell (argc, argv);
}

int
main (int argc, char **argv)
{
  scm_boot_guile (argc, argv, inner_main, 0);
  return 0; /* never reached */
}
```

When Guile is correctly installed on your system, the above program can be compiled and linked like this:

```
$ gcc -o simple-guile simple-guile.c \
    `pkg-config --cflags --libs guile-3.0`
```

When it is run, it behaves just like the `guile` program except that you can also call the new `my-hostname` function.

```
$ ./simple-guile
scheme@(guile-user)> (+ 1 2 3)
$1 = 6
scheme@(guile-user)> (my-hostname)
"burns"
```

2.4 Writing Guile Extensions

You can link Guile into your program and make Scheme available to the users of your program. You can also link your library into Guile and make its functionality available to all users of Guile.

A library that is linked into Guile is called an *extension*, but it really just is an ordinary object library.

The following example shows how to write a simple extension for Guile that makes the `j0` function available to Scheme code.

```
#include <math.h>
#include <libguile.h>
```

```
SCM
j0_wrapper (SCM x)
{
  return scm_from_double (j0 (scm_to_double (x)));
}

void
init_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

This C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux:

```
gcc ‘pkg-config --cflags guile-3.0‘ \
  -shared -o libguile-bessel.so -fPIC bessel.c
```

For creating shared libraries portably, we recommend the use of GNU Libtool (see Section “Introduction” in *GNU Libtool*).

A shared library can be loaded into a running Guile process with the function `load-extension`. The `j0` is then immediately available:

```
$ guile
scheme@(guile-user)> (load-extension "./libguile-bessel" "init_bessel")
scheme@(guile-user)> (j0 2)
$1 = 0.223890779141236
```

For more on how to install your extension, see Section 4.7 [Installing Site Packages], page 57.

2.5 Using the Guile Module System

Guile has support for dividing a program into *modules*. By using modules, you can group related code together and manage the composition of complete programs from largely independent parts.

For more details on the module system beyond this introductory material, See Section 6.20 [Modules], page 410.

2.5.1 Using Modules

Guile comes with a lot of useful modules, for example for string processing or command line parsing. Additionally, there exist many Guile modules written by other Guile hackers, but which have to be installed manually.

Here is a sample interactive session that shows how to use the `(ice-9 popen)` module which provides the means for communicating with other processes over pipes together with the `(ice-9 rdelim)` module that provides the function `read-line`.

```
$ guile
scheme@(guile-user)> (use-modules (ice-9 popen))
scheme@(guile-user)> (use-modules (ice-9 rdelim))
scheme@(guile-user)> (define p (open-input-pipe "ls -l"))
scheme@(guile-user)> (read-line p)
$1 = "total 30"
scheme@(guile-user)> (read-line p)
$2 = "drwxr-sr-x    2 mgrabmue mgrabmue     1024 Mar 29 19:57 CVS"
```

2.5.2 Writing new Modules

You can create new modules using the syntactic form `define-module`. All definitions following this form until the next `define-module` are placed into the new module.

One module is usually placed into one file, and that file is installed in a location where Guile can automatically find it. The following session shows a simple example.

```
$ cat /usr/local/share/guile/site/foo/bar.scm

(define-module (foo bar)
  #:export (frob))

(define (frob x) (* 2 x))

$ guile
scheme@(guile-user)> (use-modules (foo bar))
scheme@(guile-user)> (frob 12)
$1 = 24
```

For more on how to install your module, see Section 4.7 [Installing Site Packages], page 57.

2.5.3 Putting Extensions into Modules

In addition to Scheme code you can also put things that are defined in C into a module.

You do this by writing a small Scheme file that defines the module and call `load-extension` directly in the body of the module.

```
$ cat /usr/local/share/guile/site/math/bessel.scm

(define-module (math bessel)
  #:export (j0))

(load-extension "libguile-bessel" "init_bessel")

$ file /usr/local/lib/guile/3.0/extensions/libguile-bessel.so
... ELF 32-bit LSB shared object ...
$ guile
scheme@(guile-user)> (use-modules (math bessel))
scheme@(guile-user)> (j0 2)
$1 = 0.223890779141236
```

See Section 6.21.4 [Modules and Extensions], page 432, for more information.

2.6 Reporting Bugs

Any problems with the installation should be reported to `bug-guile@gnu.org`.

If you find a bug in Guile, please report it to the Guile developers, so they can fix it. They may also be able to suggest workarounds when it is not possible for you to apply the bug-fix or install a new version of Guile yourself.

Before sending in bug reports, please check with the following list that you really have found a bug.

- Whenever documentation and actual behavior differ, you have certainly found a bug, either in the documentation or in the program.
- When Guile crashes, it is a bug.

- When Guile hangs or takes forever to complete a task, it is a bug.
- When calculations produce wrong results, it is a bug.
- When Guile signals an error for valid Scheme programs, it is a bug.
- When Guile does not signal an error for invalid Scheme programs, it may be a bug, unless this is explicitly documented.
- When some part of the documentation is not clear and does not make sense to you even after re-reading the section, it is a bug.

Before reporting the bug, check whether any programs you have loaded into Guile, including your `.guile` file, set any variables that may affect the functioning of Guile. Also, see whether the problem happens in a freshly started Guile without loading your `.guile` file (start Guile with the `-q` switch to prevent loading the init file). If the problem does *not* occur then, you must report the precise contents of any programs that you must load into Guile in order to cause the problem to occur.

When you write a bug report, please make sure to include as much of the information described below in the report. If you can't figure out some of the items, it is not a problem, but the more information we get, the more likely we can diagnose and fix the bug.

- The version number of Guile. You can get this information from invoking '`guile --version`' at your shell, or calling `(version)` from within Guile.
- Your machine type, as determined by the `config.guess` shell script. If you have a Guile checkout, this file is located in `build-aux`; otherwise you can fetch the latest version from `http://git.savannah.gnu.org/gitweb/?p=config.git;a=blob_plain;f=config.guess;hb=HEAD`.

  ```
  $ build-aux/config.guess
  x86_64-unknown-linux-gnu
  ```

- If you installed Guile from a binary package, the version of that package. On systems that use RPM, use `rpm -qa | grep guile`. On systems that use DPKG, `dpkg -l | grep guile`.
- If you built Guile yourself, the build configuration that you used:

  ```
  $ ./config.status --config
  '--enable-error-on-warning' '--disable-deprecated'...
  ```

- A complete description of how to reproduce the bug.

  If you have a Scheme program that produces the bug, please include it in the bug report. If your program is too big to include, please try to reduce your code to a minimal test case.

  If you can reproduce your problem at the REPL, that is best. Give a transcript of the expressions you typed at the REPL.

- A description of the incorrect behavior. For example, "The Guile process gets a fatal signal," or, "The resulting output is as follows, which I think is wrong."

  If the manifestation of the bug is a Guile error message, it is important to report the precise text of the error message, and a backtrace showing how the Scheme program arrived at the error. This can be done using the `,backtrace` command in Guile's debugger.

If your bug causes Guile to crash, additional information from a low-level debugger such as GDB might be helpful. If you have built Guile yourself, you can run Guile under GDB via the `meta/gdb-uninstalled-guile` script. Instead of invoking Guile as usual, invoke the wrapper script, type `run` to start the process, then `backtrace` when the crash comes. Include that backtrace in your report.

3 Hello Scheme!

In this chapter, we introduce the basic concepts that underpin the elegance and power of the Scheme language.

Readers who already possess a background knowledge of Scheme may happily skip this chapter. For the reader who is new to the language, however, the following discussions on data, procedures, expressions and closure are designed to provide a minimum level of Scheme understanding that is more or less assumed by the chapters that follow.

The style of this introductory material aims about halfway between the terse precision of R5RS and the discursiveness of existing Scheme tutorials. For pointers to useful Scheme resources on the web, please see Section 3.5 [Further Reading], page 34.

3.1 Data Types, Values and Variables

This section discusses the representation of data types and values, what it means for Scheme to be a *latently typed* language, and the role of variables. We conclude by introducing the Scheme syntaxes for defining a new variable, and for changing the value of an existing variable.

3.1.1 Latent Typing

The term *latent typing* is used to describe a computer language, such as Scheme, for which you cannot, *in general*, simply look at a program's source code and determine what type of data will be associated with a particular variable, or with the result of a particular expression.

Sometimes, of course, you *can* tell from the code what the type of an expression will be. If you have a line in your program that sets the variable `x` to the numeric value 1, you can be certain that, immediately after that line has executed (and in the absence of multiple threads), `x` has the numeric value 1. Or if you write a procedure that is designed to concatenate two strings, it is likely that the rest of your application will always invoke this procedure with two string parameters, and quite probable that the procedure would go wrong in some way if it was ever invoked with parameters that were not both strings.

Nevertheless, the point is that there is nothing in Scheme which requires the procedure parameters always to be strings, or `x` always to hold a numeric value, and there is no way of declaring in your program that such constraints should always be obeyed. In the same vein, there is no way to declare the expected type of a procedure's return value.

Instead, the types of variables and expressions are only known – in general – at run time. If you *need* to check at some point that a value has the expected type, Scheme provides run time procedures that you can invoke to do so. But equally, it can be perfectly valid for two separate invocations of the same procedure to specify arguments with different types, and to return values with different types.

The next subsection explains what this means in practice, for the ways that Scheme programs use data types, values and variables.

3.1.2 Values and Variables

Scheme provides many data types that you can use to represent your data. Primitive types include characters, strings, numbers and procedures. Compound types, which allow a group

of primitive and compound values to be stored together, include lists, pairs, vectors and multi-dimensional arrays. In addition, Guile allows applications to define their own data types, with the same status as the built-in standard Scheme types.

As a Scheme program runs, values of all types pop in and out of existence. Sometimes values are stored in variables, but more commonly they pass seamlessly from being the result of one computation to being one of the parameters for the next.

Consider an example. A string value is created because the interpreter reads in a literal string from your program's source code. Then a numeric value is created as the result of calculating the length of the string. A second numeric value is created by doubling the calculated length. Finally the program creates a list with two elements – the doubled length and the original string itself – and stores this list in a program variable.

All of the values involved here – in fact, all values in Scheme – carry their type with them. In other words, every value "knows," at runtime, what kind of value it is. A number, a string, a list, whatever.

A variable, on the other hand, has no fixed type. A variable – `x`, say – is simply the name of a location – a box – in which you can store any kind of Scheme value. So the same variable in a program may hold a number at one moment, a list of procedures the next, and later a pair of strings. The "type" of a variable – insofar as the idea is meaningful at all – is simply the type of whatever value the variable happens to be storing at a particular moment.

3.1.3 Defining and Setting Variables

To define a new variable, you use Scheme's `define` syntax like this:

```
(define variable-name value)
```

This makes a new variable called *variable-name* and stores *value* in it as the variable's initial value. For example:

```
;; Make a variable 'x' with initial numeric value 1.
(define x 1)

;; Make a variable 'organization' with an initial string value.
(define organization "Free Software Foundation")
```

(In Scheme, a semicolon marks the beginning of a comment that continues until the end of the line. So the lines beginning `;;` are comments.)

Changing the value of an already existing variable is very similar, except that `define` is replaced by the Scheme syntax `set!`, like this:

```
(set! variable-name new-value)
```

Remember that variables do not have fixed types, so *new-value* may have a completely different type from whatever was previously stored in the location named by *variable-name*. Both of the following examples are therefore correct.

```
;; Change the value of 'x' to 5.
(set! x 5)

;; Change the value of 'organization' to the FSF's street number.
(set! organization 545)
```

In these examples, *value* and *new-value* are literal numeric or string values. In general, however, *value* and *new-value* can be any Scheme expression. Even though we have not yet covered the forms that Scheme expressions can take (see Section 3.3 [About Expressions], page 20), you can probably guess what the following `set!` example does. . .

```
(set! x (+ x 1))
```

(Note: this is not a complete description of `define` and `set!`, because we need to introduce some other aspects of Scheme before the missing pieces can be filled in. If, however, you are already familiar with the structure of Scheme, you may like to read about those missing pieces immediately by jumping ahead to the following references.

- Section 3.2.4 [Lambda Alternatives], page 20, to read about an alternative form of the `define` syntax that can be used when defining new procedures.
- Section 6.9.8 [Procedures with Setters], page 259, to read about an alternative form of the `set!` syntax that helps with changing a single value in the depths of a compound data structure.)
- See Section 6.12.3 [Internal Definitions], page 296, to read about using `define` other than at top level in a Scheme program, including a discussion of when it works to use `define` rather than `set!` to change the value of an existing variable.

3.2 The Representation and Use of Procedures

This section introduces the basics of using and creating Scheme procedures. It discusses the representation of procedures as just another kind of Scheme value, and shows how procedure invocation expressions are constructed. We then explain how `lambda` is used to create new procedures, and conclude by presenting the various shorthand forms of `define` that can be used instead of writing an explicit `lambda` expression.

3.2.1 Procedures as Values

One of the great simplifications of Scheme is that a procedure is just another type of value, and that procedure values can be passed around and stored in variables in exactly the same way as, for example, strings and lists. When we talk about a built-in standard Scheme procedure such as `open-input-file`, what we actually mean is that there is a pre-defined top level variable called `open-input-file`, whose value is a procedure that implements what R5RS says that `open-input-file` should do.

Note that this is quite different from many dialects of Lisp — including Emacs Lisp — in which a program can use the same name with two quite separate meanings: one meaning identifies a Lisp function, while the other meaning identifies a Lisp variable, whose value need have nothing to do with the function that is associated with the first meaning. In these dialects, functions and variables are said to live in different *namespaces*.

In Scheme, on the other hand, all names belong to a single unified namespace, and the variables that these names identify can hold any kind of Scheme value, including procedure values.

One consequence of the "procedures as values" idea is that, if you don't happen to like the standard name for a Scheme procedure, you can change it.

For example, `call-with-current-continuation` is a very important standard Scheme procedure, but it also has a very long name! So, many programmers use the following definition to assign the same procedure value to the more convenient name `call/cc`.

```
(define call/cc call-with-current-continuation)
```

Let's understand exactly how this works. The definition creates a new variable `call/cc`, and then sets its value to the value of the variable `call-with-current-continuation`; the latter value is a procedure that implements the behaviour that R5RS specifies under the name "call-with-current-continuation". So `call/cc` ends up holding this value as well.

Now that `call/cc` holds the required procedure value, you could choose to use `call-with-current-continuation` for a completely different purpose, or just change its value so that you will get an error if you accidentally use `call-with-current-continuation` as a procedure in your program rather than `call/cc`. For example:

```
(set! call-with-current-continuation "Not a procedure any more!")
```

Or you could just leave `call-with-current-continuation` as it was. It's perfectly fine for more than one variable to hold the same procedure value.

3.2.2 Simple Procedure Invocation

A procedure invocation in Scheme is written like this:

```
(procedure [arg1 [arg2 ...]])
```

In this expression, *procedure* can be any Scheme expression whose value is a procedure. Most commonly, however, *procedure* is simply the name of a variable whose value is a procedure.

For example, `string-append` is a standard Scheme procedure whose behaviour is to concatenate together all the arguments, which are expected to be strings, that it is given. So the expression

```
(string-append "/home" "/" "andrew")
```

is a procedure invocation whose result is the string value `"/home/andrew"`.

Similarly, `string-length` is a standard Scheme procedure that returns the length of a single string argument, so

```
(string-length "abc")
```

is a procedure invocation whose result is the numeric value 3.

Each of the parameters in a procedure invocation can itself be any Scheme expression. Since a procedure invocation is itself a type of expression, we can put these two examples together to get

```
(string-length (string-append "/home" "/" "andrew"))
```

— a procedure invocation whose result is the numeric value 12.

(You may be wondering what happens if the two examples are combined the other way round. If we do this, we can make a procedure invocation expression that is *syntactically* correct:

```
(string-append "/home" (string-length "abc"))
```

but when this expression is executed, it will cause an error, because the result of `(string-length "abc")` is a numeric value, and `string-append` is not designed to accept a numeric value as one of its arguments.)

3.2.3 Creating and Using a New Procedure

Scheme has lots of standard procedures, and Guile provides all of these via predefined top level variables. All of these standard procedures are documented in the later chapters of this reference manual.

Before very long, though, you will want to create new procedures that encapsulate aspects of your own applications' functionality. To do this, you can use the famous `lambda` syntax.

For example, the value of the following Scheme expression

```
(lambda (name address) body ...)
```

is a newly created procedure that takes two arguments: `name` and `address`. The behaviour of the new procedure is determined by the sequence of expressions and definitions in the *body* of the procedure definition. (Typically, *body* would use the arguments in some way, or else there wouldn't be any point in giving them to the procedure.) When invoked, the new procedure returns a value that is the value of the last expression in the *body*.

To make things more concrete, let's suppose that the two arguments are both strings, and that the purpose of this procedure is to form a combined string that includes these arguments. Then the full lambda expression might look like this:

```
(lambda (name address)
  (string-append "Name=" name ":Address=" address))
```

We noted in the previous subsection that the *procedure* part of a procedure invocation expression can be any Scheme expression whose value is a procedure. But that's exactly what a lambda expression is! So we can use a lambda expression directly in a procedure invocation, like this:

```
((lambda (name address)
   (string-append "Name=" name ":Address=" address))
 "FSF"
 "Cambridge")
```

This is a valid procedure invocation expression, and its result is the string:

```
"Name=FSF:Address=Cambridge"
```

It is more common, though, to store the procedure value in a variable —

```
(define make-combined-string
  (lambda (name address)
    (string-append "Name=" name ":Address=" address)))
```

— and then to use the variable name in the procedure invocation:

```
(make-combined-string "FSF" "Cambridge")
```

Which has exactly the same result.

It's important to note that procedures created using `lambda` have exactly the same status as the standard built in Scheme procedures, and can be invoked, passed around, and stored in variables in exactly the same ways.

3.2.4 Lambda Alternatives

Since it is so common in Scheme programs to want to create a procedure and then store it in a variable, there is an alternative form of the `define` syntax that allows you to do just that.

A `define` expression of the form

```
(define (name [arg1 [arg2 ...]])
  body ...)
```

is exactly equivalent to the longer form

```
(define name
  (lambda ([arg1 [arg2 ...]])
    body ...))
```

So, for example, the definition of `make-combined-string` in the previous subsection could equally be written:

```
(define (make-combined-string name address)
  (string-append "Name=" name ":Address=" address))
```

This kind of procedure definition creates a procedure that requires exactly the expected number of arguments. There are two further forms of the `lambda` expression, which create a procedure that can accept a variable number of arguments:

```
(lambda (arg1 ... . args) body ...)

(lambda args body ...)
```

The corresponding forms of the alternative `define` syntax are:

```
(define (name arg1 ... . args) body ...)

(define (name . args) body ...)
```

For details on how these forms work, see See Section 6.9.1 [Lambda], page 248.

Prior to Guile 2.0, Guile provided an extension to `define` syntax that allowed you to nest the previous extension up to an arbitrary depth. These are no longer provided by default, and instead have been moved to Section 7.19 [Curried Definitions], page 744.

(It could be argued that the alternative `define` forms are rather confusing, especially for newcomers to the Scheme language, as they hide both the role of `lambda` and the fact that procedures are values that are stored in variables in the same way as any other kind of value. On the other hand, they are very convenient, and they are also a good example of another of Scheme's powerful features: the ability to specify arbitrary syntactic transformations at run time, which can be applied to subsequently read input.)

3.3 Expressions and Evaluation

So far, we have met expressions that *do* things, such as the `define` expressions that create and initialize new variables, and we have also talked about expressions that have *values*, for example the value of the procedure invocation expression:

```
(string-append "/home" "/" "andrew")
```

but we haven't yet been precise about what causes an expression like this procedure invocation to be reduced to its "value", or how the processing of such expressions relates to the execution of a Scheme program as a whole.

This section clarifies what we mean by an expression's value, by introducing the idea of *evaluation*. It discusses the side effects that evaluation can have, explains how each of the various types of Scheme expression is evaluated, and describes the behaviour and use of the Guile REPL as a mechanism for exploring evaluation. The section concludes with a very brief summary of Scheme's common syntactic expressions.

3.3.1 Evaluating Expressions and Executing Programs

In Scheme, the process of executing an expression is known as *evaluation*. Evaluation has two kinds of result:

- the *value* of the evaluated expression
- the *side effects* of the evaluation, which consist of any effects of evaluating the expression that are not represented by the value.

Of the expressions that we have met so far, `define` and `set!` expressions have side effects — the creation or modification of a variable — but no value; `lambda` expressions have values — the newly constructed procedures — but no side effects; and procedure invocation expressions, in general, have either values, or side effects, or both.

It is tempting to try to define more intuitively what we mean by "value" and "side effects", and what the difference between them is. In general, though, this is extremely difficult. It is also unnecessary; instead, we can quite happily define the behaviour of a Scheme program by specifying how Scheme executes a program as a whole, and then by describing the value and side effects of evaluation for each type of expression individually.

So, some[1] definitions. . .

- A Scheme program consists of a sequence of expressions.
- A Scheme interpreter executes the program by evaluating these expressions in order, one by one.
- An expression can be
  - a piece of literal data, such as a number `2.3` or a string `"Hello world!"`
  - a variable name
  - a procedure invocation expression
  - one of Scheme's special syntactic expressions.

The following subsections describe how each of these types of expression is evaluated.

3.3.1.1 Evaluating Literal Data

When a literal data expression is evaluated, the value of the expression is simply the value that the expression describes. The evaluation of a literal data expression has no side effects.

So, for example,

- the value of the expression `"abc"` is the string value `"abc"`

[1] These definitions are approximate. For the whole and detailed truth, see Section "Formal syntax and semantics" in *The Revised(5) Report on the Algorithmic Language Scheme*.

- the value of the expression `3+4i` is the complex number 3 + 4i
- the value of the expression `#(1 2 3)` is a three-element vector containing the numeric values 1, 2 and 3.

For any data type which can be expressed literally like this, the syntax of the literal data expression for that data type — in other words, what you need to write in your code to indicate a literal value of that type — is known as the data type's *read syntax*. This manual specifies the read syntax for each such data type in the section that describes that data type.

Some data types do not have a read syntax. Procedures, for example, cannot be expressed as literal data; they must be created using a `lambda` expression (see Section 3.2.3 [Creating a Procedure], page 19) or implicitly using the shorthand form of `define` (see Section 3.2.4 [Lambda Alternatives], page 20).

3.3.1.2 Evaluating a Variable Reference

When an expression that consists simply of a variable name is evaluated, the value of the expression is the value of the named variable. The evaluation of a variable reference expression has no side effects.

So, after

```
(define key "Paul Evans")
```

the value of the expression `key` is the string value `"Paul Evans"`. If *key* is then modified by

```
(set! key 3.74)
```

the value of the expression `key` is the numeric value 3.74.

If there is no variable with the specified name, evaluation of the variable reference expression signals an error.

3.3.1.3 Evaluating a Procedure Invocation Expression

This is where evaluation starts getting interesting! As already noted, a procedure invocation expression has the form

```
(procedure [arg1 [arg2 ...]])
```

where *procedure* must be an expression whose value, when evaluated, is a procedure.

The evaluation of a procedure invocation expression like this proceeds by

- evaluating individually the expressions *procedure*, *arg1*, *arg2*, and so on
- calling the procedure that is the value of the *procedure* expression with the list of values obtained from the evaluations of *arg1*, *arg2* etc. as its parameters.

For a procedure defined in Scheme, "calling the procedure with the list of values as its parameters" means binding the values to the procedure's formal parameters and then evaluating the sequence of expressions that make up the body of the procedure definition. The value of the procedure invocation expression is the value of the last evaluated expression in the procedure body. The side effects of calling the procedure are the combination of the side effects of the sequence of evaluations of expressions in the procedure body.

For a built-in procedure, the value and side-effects of calling the procedure are best described by that procedure's documentation.

Note that the complete side effects of evaluating a procedure invocation expression consist not only of the side effects of the procedure call, but also of any side effects of the preceding evaluation of the expressions *procedure*, *arg1*, *arg2*, and so on.

To illustrate this, let's look again at the procedure invocation expression:

```
(string-length (string-append "/home" "/" "andrew"))
```

In the outermost expression, *procedure* is `string-length` and *arg1* is `(string-append "/home" "/" "andrew")`.

- Evaluation of `string-length`, which is a variable, gives a procedure value that implements the expected behaviour for "string-length".
- Evaluation of `(string-append "/home" "/" "andrew")`, which is another procedure invocation expression, means evaluating each of
  - `string-append`, which gives a procedure value that implements the expected behaviour for "string-append"
  - `"/home"`, which gives the string value `"/home"`
  - `"/"`, which gives the string value `"/"`
  - `"andrew"`, which gives the string value `"andrew"`

  and then invoking the procedure value with this list of string values as its arguments. The resulting value is a single string value that is the concatenation of all the arguments, namely `"/home/andrew"`.

In the evaluation of the outermost expression, the interpreter can now invoke the procedure value obtained from *procedure* with the value obtained from *arg1* as its arguments. The resulting value is a numeric value that is the length of the argument string, which is 12.

3.3.1.4 Evaluating Special Syntactic Expressions

When a procedure invocation expression is evaluated, the procedure and *all* the argument expressions must be evaluated before the procedure can be invoked. Special syntactic expressions are special because they are able to manipulate their arguments in an unevaluated form, and can choose whether to evaluate any or all of the argument expressions.

Why is this needed? Consider a program fragment that asks the user whether or not to delete a file, and then deletes the file if the user answers yes.

```
(if (string=? (read-answer "Should I delete this file?")
              "yes")
    (delete-file file))
```

If the outermost `(if ...)` expression here was a procedure invocation expression, the expression `(delete-file file)`, whose side effect is to actually delete a file, would already have been evaluated before the `if` procedure even got invoked! Clearly this is no use — the whole point of an `if` expression is that the *consequent* expression is only evaluated if the condition of the `if` expression is "true".

Therefore `if` must be special syntax, not a procedure. Other special syntaxes that we have already met are `define`, `set!` and `lambda`. `define` and `set!` are syntax because they need to know the variable *name* that is given as the first argument in a `define` or `set!` expression, not that variable's value. `lambda` is syntax because it does not immediately

evaluate the expressions that define the procedure body; instead it creates a procedure object that incorporates these expressions so that they can be evaluated in the future, when that procedure is invoked.

The rules for evaluating each special syntactic expression are specified individually for each special syntax. For a summary of standard special syntax, see See Section 3.3.4 [Syntax Summary], page 25.

3.3.2 Tail calls

Scheme is "properly tail recursive", meaning that tail calls or recursions from certain contexts do not consume stack space or other resources and can therefore be used on arbitrarily large data or for an arbitrarily long calculation. Consider for example,

```
(define (foo n)
  (display n)
  (newline)
  (foo (1+ n)))

(foo 1)
⊣
1
2
3
...
```

`foo` prints numbers infinitely, starting from the given n. It's implemented by printing n then recursing to itself to print $n + 1$ and so on. This recursion is a tail call, it's the last thing done, and in Scheme such tail calls can be made without limit.

Or consider a case where a value is returned, a version of the SRFI-1 `last` function (see Section 7.5.3.3 [SRFI-1 Selectors], page 586) returning the last element of a list,

```
(define (my-last lst)
  (if (null? (cdr lst))
      (car lst)
      (my-last (cdr lst))))

(my-last '(1 2 3)) ⇒ 3
```

If the list has more than one element, `my-last` applies itself to the `cdr`. This recursion is a tail call, there's no code after it, and the return value is the return value from that call. In Scheme this can be used on an arbitrarily long list argument.

A proper tail call is only available from certain contexts, namely the following special form positions,

- `and` — last expression
- `begin` — last expression
- `case` — last expression in each clause
- `cond` — last expression in each clause, and the call to a `=>` procedure is a tail call
- `do` — last result expression

- `if` — "true" and "false" leg expressions
- `lambda` — last expression in body
- `let`, `let*`, `letrec`, `let-syntax`, `letrec-syntax` — last expression in body
- `or` — last expression

The following core functions make tail calls,

- `apply` — tail call to given procedure
- `call-with-current-continuation` — tail call to the procedure receiving the new continuation
- `call-with-values` — tail call to the values-receiving procedure
- `eval` — tail call to evaluate the form
- `string-any`, `string-every` — tail call to predicate on the last character (if that point is reached)

The above are just core functions and special forms. Tail calls in other modules are described with the relevant documentation, for example SRFI-1 `any` and `every` (see Section 7.5.3.7 [SRFI-1 Searching], page 592).

It will be noted there are a lot of places which could potentially be tail calls, for instance the last call in a `for-each`, but only those explicitly described are guaranteed.

3.3.3 Using the Guile REPL

If you start Guile without specifying a particular program for it to execute, Guile enters its standard Read Evaluate Print Loop — or *REPL* for short. In this mode, Guile repeatedly reads in the next Scheme expression that the user types, evaluates it, and prints the resulting value.

The REPL is a useful mechanism for exploring the evaluation behaviour described in the previous subsection. If you type `string-append`, for example, the REPL replies `#<primitive-procedure string-append>`, illustrating the relationship between the variable `string-append` and the procedure value stored in that variable.

In this manual, the notation ⇒ is used to mean "evaluates to". Wherever you see an example of the form

expression
⇒
result

feel free to try it out yourself by typing *expression* into the REPL and checking that it gives the expected *result*.

3.3.4 Summary of Common Syntax

This subsection lists the most commonly used Scheme syntactic expressions, simply so that you will recognize common special syntax when you see it. For a full description of each of these syntaxes, follow the appropriate reference.

`lambda` (see Section 6.9.1 [Lambda], page 248) is used to construct procedure objects.

`define` (see Section 6.12.1 [Top Level], page 293) is used to create a new variable and set its initial value.

`set!` (see Section 6.12.1 [Top Level], page 293) is used to modify an existing variable's value.

`let`, `let*` and `letrec` (see Section 6.12.2 [Local Bindings], page 294) create an inner lexical environment for the evaluation of a sequence of expressions, in which a specified set of local variables is bound to the values of a corresponding set of expressions. For an introduction to environments, see See Section 3.4 [About Closure], page 26.

`begin` (see Section 6.13.1 [begin], page 298) executes a sequence of expressions in order and returns the value of the last expression. Note that this is not the same as a procedure which returns its last argument, because the evaluation of a procedure invocation expression does not guarantee to evaluate the arguments in order.

`if` and `cond` (see Section 6.13.2 [Conditionals], page 299) provide conditional evaluation of argument expressions depending on whether one or more conditions evaluate to "true" or "false".

`case` (see Section 6.13.2 [Conditionals], page 299) provides conditional evaluation of argument expressions depending on whether a variable has one of a specified group of values.

`and` (see Section 6.13.3 [and or], page 301) executes a sequence of expressions in order until either there are no expressions left, or one of them evaluates to "false".

`or` (see Section 6.13.3 [and or], page 301) executes a sequence of expressions in order until either there are no expressions left, or one of them evaluates to "true".

3.4 The Concept of Closure

The concept of *closure* is the idea that a lambda expression "captures" the variable bindings that are in lexical scope at the point where the lambda expression occurs. The procedure created by the lambda expression can refer to and mutate the captured bindings, and the values of those bindings persist between procedure calls.

This section explains and explores the various parts of this idea in more detail.

3.4.1 Names, Locations, Values and Environments

We said earlier that a variable name in a Scheme program is associated with a location in which any kind of Scheme value may be stored. (Incidentally, the term "vcell" is often used in Lisp and Scheme circles as an alternative to "location".) Thus part of what we mean when we talk about "creating a variable" is in fact establishing an association between a name, or identifier, that is used by the Scheme program code, and the variable location to which that name refers. Although the value that is stored in that location may change, the location to which a given name refers is always the same.

We can illustrate this by breaking down the operation of the `define` syntax into three parts: `define`

- creates a new location
- establishes an association between that location and the name specified as the first argument of the `define` expression
- stores in that location the value obtained by evaluating the second argument of the `define` expression.

A collection of associations between names and locations is called an *environment*. When you create a top level variable in a program using `define`, the name-location association for that variable is added to the "top level" environment. The "top level" environment also includes name-location associations for all the procedures that are supplied by standard Scheme.

It is also possible to create environments other than the top level one, and to create variable bindings, or name-location associations, in those environments. This ability is a key ingredient in the concept of closure; the next subsection shows how it is done.

3.4.2 Local Variables and Environments

We have seen how to create top level variables using the `define` syntax (see Section 3.1.3 [Definition], page 16). It is often useful to create variables that are more limited in their scope, typically as part of a procedure body. In Scheme, this is done using the `let` syntax, or one of its modified forms `let*` and `letrec`. These syntaxes are described in full later in the manual (see Section 6.12.2 [Local Bindings], page 294). Here our purpose is to illustrate their use just enough that we can see how local variables work.

For example, the following code uses a local variable `s` to simplify the computation of the area of a triangle given the lengths of its three sides.

```
(define a 5.3)
(define b 4.7)
(define c 2.8)

(define area
  (let ((s (/ (+ a b c) 2)))
    (sqrt (* s (- s a) (- s b) (- s c)))))
```

The effect of the `let` expression is to create a new environment and, within this environment, an association between the name `s` and a new location whose initial value is obtained by evaluating `(/ (+ a b c) 2)`. The expressions in the body of the `let`, namely `(sqrt (* s (- s a) (- s b) (- s c)))`, are then evaluated in the context of the new environment, and the value of the last expression evaluated becomes the value of the whole `let` expression, and therefore the value of the variable `area`.

3.4.3 Environment Chaining

In the example of the previous subsection, we glossed over an important point. The body of the `let` expression in that example refers not only to the local variable `s`, but also to the top level variables `a`, `b`, `c` and `sqrt`. (`sqrt` is the standard Scheme procedure for calculating a square root.) If the body of the `let` expression is evaluated in the context of the *local* `let` environment, how does the evaluation get at the values of these top level variables?

The answer is that the local environment created by a `let` expression automatically has a reference to its containing environment — in this case the top level environment — and that the Scheme interpreter automatically looks for a variable binding in the containing environment if it doesn't find one in the local environment. More generally, every environment except for the top level one has a reference to its containing environment, and the interpreter keeps searching back up the chain of environments — from most local to top level — until it either finds a variable binding for the required identifier or exhausts the chain.

This description also determines what happens when there is more than one variable binding with the same name. Suppose, continuing the example of the previous subsection, that there was also a pre-existing top level variable `s` created by the expression:

```
(define s "Some beans, my lord!")
```

Then both the top level environment and the local `let` environment would contain bindings for the name `s`. When evaluating code within the `let` body, the interpreter looks first in the local `let` environment, and so finds the binding for `s` created by the `let` syntax. Even though this environment has a reference to the top level environment, which also has a binding for `s`, the interpreter doesn't get as far as looking there. When evaluating code outside the `let` body, the interpreter looks up variable names in the top level environment, so the name `s` refers to the top level variable.

Within the `let` body, the binding for `s` in the local environment is said to *shadow* the binding for `s` in the top level environment.

3.4.4 Lexical Scope

The rules that we have just been describing are the details of how Scheme implements "lexical scoping". This subsection takes a brief diversion to explain what lexical scope means in general and to present an example of non-lexical scoping.

"Lexical scope" in general is the idea that

- an identifier at a particular place in a program always refers to the same variable location — where "always" means "every time that the containing expression is executed", and that
- the variable location to which it refers can be determined by static examination of the source code context in which that identifier appears, without having to consider the flow of execution through the program as a whole.

In practice, lexical scoping is the norm for most programming languages, and probably corresponds to what you would intuitively consider to be "normal". You may even be wondering how the situation could possibly — and usefully — be otherwise. To demonstrate that another kind of scoping is possible, therefore, and to compare it against lexical scoping, the following subsection presents an example of non-lexical scoping and examines in detail how its behavior differs from the corresponding lexically scoped code.

3.4.4.1 An Example of Non-Lexical Scoping

To demonstrate that non-lexical scoping does exist and can be useful, we present the following example from Emacs Lisp, which is a "dynamically scoped" language.

```
(defvar currency-abbreviation "USD")

(defun currency-string (units hundredths)
  (concat currency-abbreviation
          (number-to-string units)
          "."
          (number-to-string hundredths)))

(defun french-currency-string (units hundredths)
  (let ((currency-abbreviation "FRF"))
```

```
    (currency-string units hundredths)))
```

The question to focus on here is: what does the identifier `currency-abbreviation` refer to in the `currency-string` function? The answer, in Emacs Lisp, is that all variable bindings go onto a single stack, and that `currency-abbreviation` refers to the topmost binding from that stack which has the name "currency-abbreviation". The binding that is created by the `defvar` form, to the value `"USD"`, is only relevant if none of the code that calls `currency-string` rebinds the name "currency-abbreviation" in the meanwhile.

The second function `french-currency-string` works precisely by taking advantage of this behaviour. It creates a new binding for the name "currency-abbreviation" which overrides the one established by the `defvar` form.

```
;; Note!  This is Emacs Lisp evaluation, not Scheme!
(french-currency-string 33 44)
⇒
"FRF33.44"
```

Now let's look at the corresponding, *lexically scoped* Scheme code:

```
(define currency-abbreviation "USD")

(define (currency-string units hundredths)
  (string-append currency-abbreviation
                 (number->string units)
                 "."
                 (number->string hundredths)))

(define (french-currency-string units hundredths)
  (let ((currency-abbreviation "FRF"))
    (currency-string units hundredths)))
```

According to the rules of lexical scoping, the `currency-abbreviation` in `currency-string` refers to the variable location in the innermost environment at that point in the code which has a binding for `currency-abbreviation`, which is the variable location in the top level environment created by the preceding `(define currency-abbreviation ...)` expression.

In Scheme, therefore, the `french-currency-string` procedure does not work as intended. The variable binding that it creates for "currency-abbreviation" is purely local to the code that forms the body of the `let` expression. Since this code doesn't directly use the name "currency-abbreviation" at all, the binding is pointless.

```
(french-currency-string 33 44)
⇒
"USD33.44"
```

This begs the question of how the Emacs Lisp behaviour can be implemented in Scheme. In general, this is a design question whose answer depends upon the problem that is being addressed. In this case, the best answer may be that `currency-string` should be redesigned so that it can take an optional third argument. This third argument, if supplied, is interpreted as a currency abbreviation that overrides the default.

It is possible to change `french-currency-string` so that it mostly works without changing `currency-string`, but the fix is inelegant, and susceptible to interrupts that could leave the `currency-abbreviation` variable in the wrong state:

```
(define (french-currency-string units hundredths)
  (set! currency-abbreviation "FRF")
  (let ((result (currency-string units hundredths)))
    (set! currency-abbreviation "USD")
    result))
```

The key point here is that the code does not create any local binding for the identifier `currency-abbreviation`, so all occurrences of this identifier refer to the top level variable.

3.4.5 Closure

Consider a `let` expression that doesn't contain any `lambda`s:

```
(let ((s (/ (+ a b c) 2)))
  (sqrt (* s (- s a) (- s b) (- s c))))
```

When the Scheme interpreter evaluates this, it

- creates a new environment with a reference to the environment that was current when it encountered the `let`
- creates a variable binding for `s` in the new environment, with value given by `(/ (+ a b c) 2)`
- evaluates the expression in the body of the `let` in the context of the new local environment, and remembers the value `V`
- forgets the local environment
- continues evaluating the expression that contained the `let`, using the value `V` as the value of the `let` expression, in the context of the containing environment.

After the `let` expression has been evaluated, the local environment that was created is simply forgotten, and there is no longer any way to access the binding that was created in this environment. If the same code is evaluated again, it will follow the same steps again, creating a second new local environment that has no connection with the first, and then forgetting this one as well.

If the `let` body contains a `lambda` expression, however, the local environment is *not* forgotten. Instead, it becomes associated with the procedure that is created by the `lambda` expression, and is reinstated every time that that procedure is called. In detail, this works as follows.

- When the Scheme interpreter evaluates a `lambda` expression, to create a procedure object, it stores the current environment as part of the procedure definition.
- Then, whenever that procedure is called, the interpreter reinstates the environment that is stored in the procedure definition and evaluates the procedure body within the context of that environment.

The result is that the procedure body is always evaluated in the context of the environment that was current when the procedure was created.

This is what is meant by *closure*. The next few subsections present examples that explore the usefulness of this concept.

3.4.6 Example 1: A Serial Number Generator

This example uses closure to create a procedure with a variable binding that is private to the procedure, like a local variable, but whose value persists between procedure calls.

```
(define (make-serial-number-generator)
  (let ((current-serial-number 0))
    (lambda ()
      (set! current-serial-number (+ current-serial-number 1))
      current-serial-number)))

(define entry-sn-generator (make-serial-number-generator))

(entry-sn-generator)
⇒
1

(entry-sn-generator)
⇒
2
```

When `make-serial-number-generator` is called, it creates a local environment with a binding for `current-serial-number` whose initial value is 0, then, within this environment, creates a procedure. The local environment is stored within the created procedure object and so persists for the lifetime of the created procedure.

Every time the created procedure is invoked, it increments the value of the `current-serial-number` binding in the captured environment and then returns the current value.

Note that `make-serial-number-generator` can be called again to create a second serial number generator that is independent of the first. Every new invocation of `make-serial-number-generator` creates a new local `let` environment and returns a new procedure object with an association to this environment.

3.4.7 Example 2: A Shared Persistent Variable

This example uses closure to create two procedures, `get-balance` and `deposit`, that both refer to the same captured local environment so that they can both access the `balance` variable binding inside that environment. The value of this variable binding persists between calls to either procedure.

Note that the captured `balance` variable binding is private to these two procedures: it is not directly accessible to any other code. It can only be accessed indirectly via `get-balance` or `deposit`, as illustrated by the `withdraw` procedure.

```
(define get-balance #f)
(define deposit #f)

(let ((balance 0))
  (set! get-balance
        (lambda ()
          balance))
```

```
  (set! deposit
        (lambda (amount)
          (set! balance (+ balance amount))
          balance)))

(define (withdraw amount)
  (deposit (- amount)))

(get-balance)
⇒
0

(deposit 50)
⇒
50

(withdraw 75)
⇒
-25
```

An important detail here is that the `get-balance` and `deposit` variables must be set up by `define`ing them at top level and then `set!`ing their values inside the `let` body. Using `define` within the `let` body would not work: this would create variable bindings within the local `let` environment that would not be accessible at top level.

3.4.8 Example 3: The Callback Closure Problem

A frequently used programming model for library code is to allow an application to register a callback function for the library to call when some particular event occurs. It is often useful for the application to make several such registrations using the same callback function, for example if several similar library events can be handled using the same application code, but the need then arises to distinguish the callback function calls that are associated with one callback registration from those that are associated with different callback registrations.

In languages without the ability to create functions dynamically, this problem is usually solved by passing a `user_data` parameter on the registration call, and including the value of this parameter as one of the parameters on the callback function. Here is an example of declarations using this solution in C:

```
typedef void (event_handler_t) (int event_type,
                                void *user_data);

void register_callback (int event_type,
                        event_handler_t *handler,
                        void *user_data);
```

In Scheme, closure can be used to achieve the same functionality without requiring the library code to store a `user-data` for each callback registration.

```
;; In the library:

(define (register-callback event-type handler-proc)
```

```
  ...)

;; In the application:

(define (make-handler event-type user-data)
  (lambda ()
    ...
    <code referencing event-type and user-data>
    ...))

(register-callback event-type
                   (make-handler event-type ...))
```

As far as the library is concerned, `handler-proc` is a procedure with no arguments, and all the library has to do is call it when the appropriate event occurs. From the application's point of view, though, the handler procedure has used closure to capture an environment that includes all the context that the handler code needs — `event-type` and `user-data` — to handle the event correctly.

3.4.9 Example 4: Object Orientation

Closure is the capture of an environment, containing persistent variable bindings, within the definition of a procedure or a set of related procedures. This is rather similar to the idea in some object oriented languages of encapsulating a set of related data variables inside an "object", together with a set of "methods" that operate on the encapsulated data. The following example shows how closure can be used to emulate the ideas of objects, methods and encapsulation in Scheme.

```
(define (make-account)
  (let ((balance 0))
    (define (get-balance)
      balance)
    (define (deposit amount)
      (set! balance (+ balance amount))
      balance)
    (define (withdraw amount)
      (deposit (- amount)))

    (lambda args
      (apply
        (case (car args)
          ((get-balance) get-balance)
          ((deposit) deposit)
          ((withdraw) withdraw)
          (else (error "Invalid method!")))
        (cdr args)))))
```

Each call to `make-account` creates and returns a new procedure, created by the expression in the example code that begins "(lambda args".

```
(define my-account (make-account))
```

```
my-account
⇒
#<procedure args>
```

This procedure acts as an account object with methods `get-balance`, `deposit` and `withdraw`. To apply one of the methods to the account, you call the procedure with a symbol indicating the required method as the first parameter, followed by any other parameters that are required by that method.

```
(my-account 'get-balance)
⇒
0

(my-account 'withdraw 5)
⇒
-5

(my-account 'deposit 396)
⇒
391

(my-account 'get-balance)
⇒
391
```

Note how, in this example, both the current balance and the helper procedures `get-balance`, `deposit` and `withdraw`, used to implement the guts of the account object's methods, are all stored in variable bindings within the private local environment captured by the `lambda` expression that creates the account object procedure.

3.5 Further Reading

- The website `http://www.schemers.org/` is a good starting point for all things Scheme.
- Dorai Sitaram's online Scheme tutorial, *Teach Yourself Scheme in Fixnum Days*, at `http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html`. Includes a nice explanation of continuations.
- The complete text of *Structure and Interpretation of Computer Programs*, the classic introduction to computer science and Scheme by Hal Abelson, Jerry Sussman and Julie Sussman, is now available online at `http://mitpress.mit.edu/sicp/sicp.html`. This site also provides teaching materials related to the book, and all the source code used in the book, in a form suitable for loading and running.

4 Programming in Scheme

Guile's core language is Scheme, and a lot can be achieved simply by using Guile to write and run Scheme programs — as opposed to having to dive into C code. In this part of the manual, we explain how to use Guile in this mode, and describe the tools that Guile provides to help you with script writing, debugging, and packaging your programs for distribution.

For detailed reference information on the variables, functions, and so on that make up Guile's application programming interface (API), see Chapter 6 [API Reference], page 99.

4.1 Guile's Implementation of Scheme

Guile's core language is Scheme, which is specified and described in the series of reports known as *RnRS*. *RnRS* is shorthand for the *Revised^n Report on the Algorithmic Language Scheme*. Guile complies fully with R5RS (see Section "Introduction" in *R5RS*), and is largely compliant with R6RS and R7RS.

Guile also has many extensions that go beyond these reports. Some of the areas where Guile extends standard Scheme are:

- Guile's interactive documentation system
- Guile's support for POSIX-compliant network programming
- GOOPS – Guile's framework for object oriented programming.

4.2 Invoking Guile

Many features of Guile depend on and can be changed by information that the user provides either before or when Guile is started. Below is a description of what information to provide and how to provide it.

4.2.1 Command-line Options

Here we describe Guile's command-line processing in detail. Guile processes its arguments from left to right, recognizing the switches described below. For examples, see Section 4.3.4 [Scripting Examples], page 44.

`script arg...`
`-s` *`script arg...`*

By default, Guile will read a file named on the command line as a script. Any command-line arguments *arg...* following *script* become the script's arguments; the `command-line` function returns a list of strings of the form (*`script arg...`*).

It is possible to name a file using a leading hyphen, for example, `-myfile.scm`. In this case, the file name must be preceded by `-s` to tell Guile that a (script) file is being named.

Scripts are read and evaluated as Scheme source code just as the `load` function would. After loading *script*, Guile exits.

`-c` *`expr arg...`*

Evaluate *expr* as Scheme code, and then exit. Any command-line arguments *arg...* following *expr* become command-line arguments; the `command-line` func-

tion returns a list of strings of the form `(guile arg...)`, where *guile* is the path of the Guile executable.

`-- arg...` Run interactively, prompting the user for expressions and evaluating them. Any command-line arguments *arg...* following the `--` become command-line arguments for the interactive session; the `command-line` function returns a list of strings of the form `(guile arg...)`, where *guile* is the path of the Guile executable.

`-L directory`
Add *directory* to the front of Guile's module load path. The given directories are searched in the order given on the command line and before any directories in the `GUILE_LOAD_PATH` environment variable. Paths added here are *not* in effect during execution of the user's `.guile` file.

`-C directory`
Like `-L`, but adjusts the load path for *compiled* files.

`-x extension`
Add *extension* to the front of Guile's load extension list (see Section 6.18.7 [Load Paths], page 393). The specified extensions are tried in the order given on the command line, and before the default load extensions. Extensions added here are *not* in effect during execution of the user's `.guile` file.

`-l file` Load Scheme source code from *file*, and continue processing the command line.

`-e function`
Make *function* the *entry point* of the script. After loading the script file (with `-s`) or evaluating the expression (with `-c`), apply *function* to a list containing the program name and the command-line arguments—the list provided by the `command-line` function.

A `-e` switch can appear anywhere in the argument list, but Guile always invokes the *function* as the *last* action it performs. This is weird, but because of the way script invocation works under POSIX, the `-s` option must always come last in the list.

The *function* is most often a simple symbol that names a function that is defined in the script. It can also be of the form `(@ module-name symbol)`, and in that case, the symbol is looked up in the module named *module-name*.

As a shorthand you can use the form `(symbol ...)`, that is, a list of only symbols that doesn't start with `@`. It is equivalent to `(@ module-name main)`, where *module-name* is `(symbol ...)` form. See Section 6.20.2 [Using Guile Modules], page 411, and Section 4.3.4 [Scripting Examples], page 44.

`-ds` Treat a final `-s` option as if it occurred at this point in the command line; load the script here.

This switch is necessary because, although the POSIX script invocation mechanism effectively requires the `-s` option to appear last, the programmer may well want to run the script before other actions requested on the command line. For examples, see Section 4.3.4 [Scripting Examples], page 44.

`\` Read more command-line arguments, starting from the second line of the script file. See Section 4.3.2 [The Meta Switch], page 42.

`--use-srfi=`*`list`*

The option `--use-srfi` expects a comma-separated list of numbers, each representing a SRFI module to be loaded into the interpreter before evaluating a script file or starting the REPL. Additionally, the feature identifier for the loaded SRFIs is recognized by the procedure `cond-expand` when this option is used.

Here is an example that loads the modules SRFI-8 ('receive') and SRFI-13 ('string library') before the GUILE interpreter is started:

```
guile --use-srfi=8,13
```

`--r6rs` Adapt Guile's initial environment to better support R6RS. See Section 7.6.1 [R6RS Incompatibilities], page 662, for some caveats.

`--r7rs` Adapt Guile's initial environment to better support R7RS. See Section 7.7.1 [R7RS Incompatibilities], page 704, for some caveats.

`--debug` Start with the debugging virtual machine (VM) engine. Using the debugging VM will enable support for VM hooks, which are needed for tracing, breakpoints, and accurate call counts when profiling. The debugging VM is slower than the regular VM, though, by about ten percent. See Section 6.26.4.1 [VM Hooks], page 485, for more information.

By default, the debugging VM engine is only used when entering an interactive session. When executing a script with `-s` or `-c`, the normal, faster VM is used by default.

`--no-debug`

Do not use the debugging VM engine, even when entering an interactive session.

Note that, despite the name, Guile running with `--no-debug` *does* support the usual debugging facilities, such as printing a detailed backtrace upon error. The only difference with `--debug` is lack of support for VM hooks and the facilities that build upon it (see above).

`-q` Do not load the initialization file, `.guile`. This option only has an effect when running interactively; running scripts does not load the `.guile` file. See Section 4.4.1 [Init File], page 48.

`--listen[=`*`p`*`]`

While this program runs, listen on a local port or a path for REPL clients. If *p* starts with a number, it is assumed to be a local port on which to listen. If it starts with a forward slash, it is assumed to be the file name of a UNIX domain socket on which to listen.

If *p* is not given, the default is local port 37146. If you look at it upside down, it almost spells "Guile". If you have netcat installed, you should be able to *`nc localhost 37146`* and get a Guile prompt. Alternately you can fire up Emacs and connect to the process; see Section 4.5 [Using Guile in Emacs], page 56, for more details.

Note: Opening a port allows anyone who can connect to that port to do anything Guile can do, as the user that the Guile process is running as. Do not use `--listen` on multi-user machines. Of course, if you do not pass `--listen` to Guile, no port will be opened.

Guile protects against the *HTTP inter-protocol exploitation attack* (`https://en.wikipedia.org/wiki/Inter-protocol_exploitation`), a scenario whereby an attacker can, *via* an HTML page, cause a web browser to send data to TCP servers listening on a loopback interface or private network. Nevertheless, you are advised to use UNIX domain sockets, as in `--listen=/some/local/file`, whenever possible.

That said, `--listen` is great for interactive debugging and development.

`--auto-compile`
: Compile source files automatically (default behavior).

`--fresh-auto-compile`
: Treat the auto-compilation cache as invalid, forcing recompilation.

`--no-auto-compile`
: Disable automatic source file compilation.

`--language=`*lang*
: For the remainder of the command line arguments, assume that files mentioned with `-l` and expressions passed with `-c` are written in *lang*. *lang* must be the name of one of the languages supported by the compiler (see Section 9.4.1 [Compiler Tower], page 856). When run interactively, set the REPL's language to *lang* (see Section 4.4 [Using Guile Interactively], page 47).

  The default language is `scheme`; other interesting values include `elisp` (for Emacs Lisp), and `ecmascript`.

  The example below shows the evaluation of expressions in Scheme, Emacs Lisp, and ECMAScript:

```
guile -c "(apply + '(1 2))"
guile --language=elisp -c "(= (funcall (symbol-function '+) 1 2) 3)"
guile --language=ecmascript -c '(function (x) { return x * x; })(2);'
```

  To load a file written in Scheme and one written in Emacs Lisp, and then start a Scheme REPL, type:

```
guile -l foo.scm --language=elisp -l foo.el --language=scheme
```

`-h`, `--help`
: Display help on invoking Guile, and then exit.

`-v`, `--version`
: Display the current version of Guile, and then exit.

4.2.2 Environment Variables

The *environment* is a feature of the operating system; it consists of a collection of variables with names and values. Each variable is called an *environment variable* (or, sometimes, a "shell variable"); environment variable names are case-sensitive, and it is conventional to

use upper-case letters only. The values are all text strings, even those that are written as numerals. (Note that here we are referring to names and values that are defined in the operating system shell from which Guile is invoked. This is not the same as a Scheme environment that is defined within a running instance of Guile. For a description of Scheme environments, see Section 3.4.1 [About Environments], page 26.)

How to set environment variables before starting Guile depends on the operating system and, especially, the shell that you are using. For example, here is how to tell Guile to provide detailed warning messages about deprecated features by setting `GUILE_WARN_DEPRECATED` using Bash:

```
$ export GUILE_WARN_DEPRECATED="detailed"
$ guile
```

Or, detailed warnings can be turned on for a single invocation using:

```
$ env GUILE_WARN_DEPRECATED="detailed" guile
```

If you wish to retrieve or change the value of the shell environment variables that affect the run-time behavior of Guile from within a running instance of Guile, see Section 7.2.6 [Runtime Environment], page 517.

Here are the environment variables that affect the run-time behavior of Guile:

`GUILE_AUTO_COMPILE`

This is a flag that can be used to tell Guile whether or not to compile Scheme source files automatically. Starting with Guile 2.0, Scheme source files will be compiled automatically, by default.

If a compiled (`.go`) file corresponding to a `.scm` file is not found or is not newer than the `.scm` file, the `.scm` file will be compiled on the fly, and the resulting `.go` file stored away. An advisory note will be printed on the console.

Compiled files will be stored in the directory `$XDG_CACHE_HOME/guile/ccache`, where `XDG_CACHE_HOME` defaults to the directory `$HOME/.cache`. This directory will be created if it does not already exist.

Note that this mechanism depends on the timestamp of the `.go` file being newer than that of the `.scm` file; if the `.scm` or `.go` files are moved after installation, care should be taken to preserve their original timestamps.

Set `GUILE_AUTO_COMPILE` to zero (0), to prevent Scheme files from being compiled automatically. Set this variable to "fresh" to tell Guile to compile Scheme files whether they are newer than the compiled files or not.

See Section 6.18.5 [Compilation], page 389.

`GUILE_HISTORY`

This variable names the file that holds the Guile REPL command history. You can specify a different history file by setting this environment variable. By default, the history file is `$HOME/.guile_history`.

`GUILE_INSTALL_LOCALE`

This is a flag that can be used to tell Guile whether or not to install the current locale at startup, via a call to `(setlocale LC_ALL "")`[1]. See Section 7.2.13 [Locales], page 547, for more information on locales.

[1] The `GUILE_INSTALL_LOCALE` environment variable was ignored in Guile versions prior to 2.0.9.

You may explicitly indicate that you do not want to install the locale by setting `GUILE_INSTALL_LOCALE` to `0`, or explicitly enable it by setting the variable to `1`.

Usually, installing the current locale is the right thing to do. It allows Guile to correctly parse and print strings with non-ASCII characters. Therefore, this option is on by default.

`GUILE_LOAD_COMPILED_PATH`

This variable may be used to augment the path that is searched for compiled Scheme files (`.go` files) when loading. Its value should be a colon-separated list of directories. If it contains the special path component `...` (ellipsis), then the default path is put in place of the ellipsis, otherwise the default path is placed at the end. The result is stored in `%load-compiled-path` (see Section 6.18.7 [Load Paths], page 393).

Here is an example using the Bash shell that adds the current directory, `.`, and the relative directory `../my-library` to `%load-compiled-path`:

```
$ export GUILE_LOAD_COMPILED_PATH=".:../my-library"
$ guile -c '(display %load-compiled-path) (newline)'
(. ../my-library /usr/local/lib/guile/3.0/ccache)
```

`GUILE_LOAD_PATH`

This variable may be used to augment the path that is searched for Scheme files when loading. Its value should be a colon-separated list of directories. If it contains the special path component `...` (ellipsis), then the default path is put in place of the ellipsis, otherwise the default path is placed at the end. The result is stored in `%load-path` (see Section 6.18.7 [Load Paths], page 393).

Here is an example using the Bash shell that prepends the current directory to `%load-path`, and adds the relative directory `../srfi` to the end:

```
$ env GUILE_LOAD_PATH=".:...:../srfi" \
guile -c '(display %load-path) (newline)'
(. /usr/local/share/guile/3.0 \
/usr/local/share/guile/site/3.0 \
/usr/local/share/guile/site \
/usr/local/share/guile \
../srfi)
```

(Note: The line breaks, above, are for documentation purposes only, and not required in the actual example.)

`GUILE_WARN_DEPRECATED`

As Guile evolves, some features will be eliminated or replaced by newer features. To help users migrate their code as this evolution occurs, Guile will issue warning messages about code that uses features that have been marked for eventual elimination. `GUILE_WARN_DEPRECATED` can be set to "no" to tell Guile not to display these warning messages, or set to "detailed" to tell Guile to display more lengthy messages describing the warning. See Section 6.2 [Deprecation], page 100.

`HOME` Guile uses the environment variable `HOME`, the name of your home directory, to locate various files, such as `.guile` or `.guile_history`.

`GUILE_JIT_THRESHOLD`
Guile has a just-in-time (JIT) code generator that makes running Guile code fast. See Section 9.3.8 [Just-In-Time Native Code], page 854, for more. The unit of code generation is the function. Each function has its own counter that gets incremented when the function is called and at each loop iteration in the function. When the counter exceeds the `GUILE_JIT_THRESHOLD`, the function will get JIT-compiled. Set `GUILE_JIT_THRESHOLD` to `-1` to disable JIT compilation, or `0` to eagerly JIT-compile each function as it's first seen.

`GUILE_JIT_LOG`
Set to `1`, `2`, or `3` to give increasing amounts of logging for JIT compilation events. Used for debugging.

`GUILE_JIT_STOP_AFTER`
Though we have tested the JIT compiler as well as we can, it's possible that it has bugs. If you suspect that Guile's JIT compiler is causing your program to fail, set `GUILE_JIT_STOP_AFTER` to a positive integer indicating the maximum number of functions to JIT-compile. By bisecting over the value of `GUILE_JIT_STOP_AFTER`, you can pinpoint the precise function that is being miscompiled.

4.3 Guile Scripting

Like AWK, Perl, or any shell, Guile can interpret script files. A Guile script is simply a file of Scheme code with some extra information at the beginning which tells the operating system how to invoke Guile, and then tells Guile how to handle the Scheme code.

4.3.1 The Top of a Script File

The first line of a Guile script must tell the operating system to use Guile to evaluate the script, and then tell Guile how to go about doing that. Here is the simplest case:

- The first two characters of the file must be '`#!`'.

  The operating system interprets this to mean that the rest of the line is the name of an executable that can interpret the script. Guile, however, interprets these characters as the beginning of a multi-line comment, terminated by the characters '`!#`' on a line by themselves. (This is an extension to the syntax described in R5RS, added to support shell scripts.)
- Immediately after those two characters must come the full pathname to the Guile interpreter. On most systems, this would be '`/usr/local/bin/guile`'.
- Then must come a space, followed by a command-line argument to pass to Guile; this should be '`-s`'. This switch tells Guile to run a script, instead of soliciting the user for input from the terminal. There are more elaborate things one can do here; see Section 4.3.2 [The Meta Switch], page 42.
- Follow this with a newline.
- The second line of the script should contain only the characters '`!#`' — just like the top of the file, but reversed. The operating system never reads this far, but Guile treats this as the end of the comment begun on the first line by the '`#!`' characters.

- If this source code file is not ASCII or ISO-8859-1 encoded, a coding declaration such as `coding: utf-8` should appear in a comment somewhere in the first five lines of the file: see Section 6.18.8 [Character Encoding of Source Files], page 395.
- The rest of the file should be a Scheme program.

Guile reads the program, evaluating expressions in the order that they appear. Upon reaching the end of the file, Guile exits.

4.3.2 The Meta Switch

Guile's command-line switches allow the programmer to describe reasonably complicated actions in scripts. Unfortunately, the POSIX script invocation mechanism only allows one argument to appear on the '`#!`' line after the path to the Guile executable, and imposes arbitrary limits on that argument's length. Suppose you wrote a script starting like this:

```
#!/usr/local/bin/guile -e main -s
!#
(define (main args)
  (map (lambda (arg) (display arg) (display " "))
       (cdr args))
  (newline))
```

The intended meaning is clear: load the file, and then call `main` on the command-line arguments. However, the system will treat everything after the Guile path as a single argument — the string `"-e main -s"` — which is not what we want.

As a workaround, the meta switch `\` allows the Guile programmer to specify an arbitrary number of options without patching the kernel. If the first argument to Guile is `\`, Guile will open the script file whose name follows the `\`, parse arguments starting from the file's second line (according to rules described below), and substitute them for the `\` switch.

Working in concert with the meta switch, Guile treats the characters '`#!`' as the beginning of a comment which extends through the next line containing only the characters '`!#`'. This sort of comment may appear anywhere in a Guile program, but it is most useful at the top of a file, meshing magically with the POSIX script invocation mechanism.

Thus, consider a script named `/u/jimb/ekko` which starts like this:

```
#!/usr/local/bin/guile \
-e main -s
!#
(define (main args)
        (map (lambda (arg) (display arg) (display " "))
             (cdr args))
        (newline))
```

Suppose a user invokes this script as follows:

```
$ /u/jimb/ekko a b c
```

Here's what happens:

- the operating system recognizes the '`#!`' token at the top of the file, and rewrites the command line to:

  ```
  /usr/local/bin/guile \ /u/jimb/ekko a b c
  ```

  This is the usual behavior, prescribed by POSIX.

- When Guile sees the first two arguments, `\ /u/jimb/ekko`, it opens `/u/jimb/ekko`, parses the three arguments `-e`, `main`, and `-s` from it, and substitutes them for the `\` switch. Thus, Guile's command line now reads:

  ```
  /usr/local/bin/guile -e main -s /u/jimb/ekko a b c
  ```

- Guile then processes these switches: it loads `/u/jimb/ekko` as a file of Scheme code (treating the first three lines as a comment), and then performs the application `(main "/u/jimb/ekko" "a" "b" "c")`.

When Guile sees the meta switch `\`, it parses command-line argument from the script file according to the following rules:

- Each space character terminates an argument. This means that two spaces in a row introduce an argument `""`.
- The tab character is not permitted (unless you quote it with the backslash character, as described below), to avoid confusion.
- The newline character terminates the sequence of arguments, and will also terminate a final non-empty argument. (However, a newline following a space will not introduce a final empty-string argument; it only terminates the argument list.)
- The backslash character is the escape character. It escapes backslash, space, tab, and newline. The ANSI C escape sequences like `\n` and `\t` are also supported. These produce argument constituents; the two-character combination `\n` doesn't act like a terminating newline. The escape sequence `\`*NNN* for exactly three octal digits reads as the character whose ASCII code is *NNN*. As above, characters produced this way are argument constituents. Backslash followed by other characters is not allowed.

4.3.3 Command Line Handling

The ability to accept and handle command line arguments is very important when writing Guile scripts to solve particular problems, such as extracting information from text files or interfacing with existing command line applications. This chapter describes how Guile makes command line arguments available to a Guile script, and the utilities that Guile provides to help with the processing of command line arguments.

When a Guile script is invoked, Guile makes the command line arguments accessible via the procedure `command-line`, which returns the arguments as a list of strings.

For example, if the script

```
#! /usr/local/bin/guile -s
!#
(write (command-line))
(newline)
```

is saved in a file `cmdline-test.scm` and invoked using the command line `./cmdline-test.scm bar.txt -o foo -frumple grob`, the output is

```
("./cmdline-test.scm" "bar.txt" "-o" "foo" "-frumple" "grob")
```

If the script invocation includes a `-e` option, specifying a procedure to call after loading the script, Guile will call that procedure with `(command-line)` as its argument. So a script that uses `-e` doesn't need to refer explicitly to `command-line` in its code. For example, the script above would have identical behaviour if it was written instead like this:

```
#! /usr/local/bin/guile \
```

```
-e main -s
!#
(define (main args)
  (write args)
  (newline))
```

(Note the use of the meta switch \ so that the script invocation can include more than one Guile option: See Section 4.3.2 [The Meta Switch], page 42.)

These scripts use the `#!` POSIX convention so that they can be executed using their own file names directly, as in the example command line `./cmdline-test.scm bar.txt -o foo -frumple grob`. But they can also be executed by typing out the implied Guile command line in full, as in:

```
$ guile -s ./cmdline-test.scm bar.txt -o foo -frumple grob
```

or

```
$ guile -e main -s ./cmdline-test2.scm bar.txt -o foo -frumple grob
```

Even when a script is invoked using this longer form, the arguments that the script receives are the same as if it had been invoked using the short form. Guile ensures that the `(command-line)` or `-e` arguments are independent of how the script is invoked, by stripping off the arguments that Guile itself processes.

A script is free to parse and handle its command line arguments in any way that it chooses. Where the set of possible options and arguments is complex, however, it can get tricky to extract all the options, check the validity of given arguments, and so on. This task can be greatly simplified by taking advantage of the module `(ice-9 getopt-long)`, which is distributed with Guile, See Section 7.4 [getopt-long], page 577.

4.3.4 Scripting Examples

To start with, here are some examples of invoking Guile directly:

`guile -- a b c`
: Run Guile interactively; `(command-line)` will return `("/usr/local/bin/guile" "a" "b" "c")`.

`guile -s /u/jimb/ex2 a b c`
: Load the file `/u/jimb/ex2`; `(command-line)` will return `("/u/jimb/ex2" "a" "b" "c")`.

`guile -c '(write %load-path) (newline)'`
: Write the value of the variable `%load-path`, print a newline, and exit.

`guile -e main -s /u/jimb/ex4 foo`
: Load the file `/u/jimb/ex4`, and then call the function `main`, passing it the list `("/u/jimb/ex4" "foo")`.

`guile -e '(ex4)' -s /u/jimb/ex4.scm foo`
: Load the file `/u/jimb/ex4.scm`, and then call the function `main` from the module '(ex4)', passing it the list `("/u/jimb/ex4" "foo")`.

`guile -l first -ds -l last -s script`
: Load the files `first`, `script`, and `last`, in that order. The `-ds` switch says when to process the `-s` switch. For a more motivated example, see the scripts below.

Here is a very simple Guile script:

```
#!/usr/local/bin/guile -s
!#
(display "Hello, world!")
(newline)
```

The first line marks the file as a Guile script. When the user invokes it, the system runs `/usr/local/bin/guile` to interpret the script, passing `-s`, the script's filename, and any arguments given to the script as command-line arguments. When Guile sees `-s` *`script`*, it loads *script*. Thus, running this program produces the output:

```
Hello, world!
```

Here is a script which prints the factorial of its argument:

```
#!/usr/local/bin/guile -s
!#
(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(display (fact (string->number (cadr (command-line)))))
(newline)
```

In action:

```
$ ./fact 5
120
$
```

However, suppose we want to use the definition of `fact` in this file from another script. We can't simply `load` the script file, and then use `fact`'s definition, because the script will try to compute and display a factorial when we load it. To avoid this problem, we might write the script this way:

```
#!/usr/local/bin/guile \
-e main -s
!#
(define (fact n)
  (if (zero? n) 1
    (* n (fact (- n 1)))))

(define (main args)
  (display (fact (string->number (cadr args))))
  (newline))
```

This version packages the actions the script should perform in a function, `main`. This allows us to load the file purely for its definitions, without any extraneous computation taking place. Then we used the meta switch `\` and the entry point switch `-e` to tell Guile to call `main` after loading the script.

```
$ ./fact 50
30414093201713378043612608166064768844377641568960512000000000000
```

Suppose that we now want to write a script which computes the `choose` function: given a set of *m* distinct objects, (`choose` *n m*) is the number of distinct subsets containing *n* objects each. It's easy to write `choose` given `fact`, so we might write the script this way:

```
#!/usr/local/bin/guile \
-l fact -e main -s
!#
(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

The command-line arguments here tell Guile to first load the file `fact`, and then run the script, with `main` as the entry point. In other words, the `choose` script can use definitions made in the `fact` script. Here are some sample runs:

```
$ ./choose 0 4
1
$ ./choose 1 4
4
$ ./choose 2 4
6
$ ./choose 3 4
4
$ ./choose 4 4
1
$ ./choose 50 100
100891344545564193334812497256
```

To call a specific procedure from a given module, we can use the special form (`@ (`*module*`)` *procedure*):

```
#!/usr/local/bin/guile \
-l fact -e (@ (fac) main) -s
!#
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

We can use `@@` to invoke non-exported procedures. For exported procedures, we can simplify this call with the shorthand `(`*`module`*`)`:

```
#!/usr/local/bin/guile \
-l fact -e (fac) -s
!#
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

For maximum portability, we can instead use the shell to execute `guile` with specified command line arguments. Here we need to take care to quote the command arguments correctly:

```
#!/usr/bin/env sh
exec guile -l fact -e '(@ (fac) main)' -s "$0" "$@"
!#
(define-module (fac)
  #:export (main))

(define (choose n m)
  (/ (fact m) (* (fact (- m n)) (fact n))))

(define (main args)
  (let ((n (string->number (cadr args)))
        (m (string->number (caddr args))))
    (display (choose n m))
    (newline)))
```

Finally, seasoned scripters are probably missing a mention of subprocesses. In Bash, for example, most shell scripts run other programs like `sed` or the like to do the actual work.

In Guile it's often possible get everything done within Guile itself, so do give that a try first. But if you just need to run a program and wait for it to finish, use `system*`. If you need to run a sub-program and capture its output, or give it input, use `open-pipe`. See Section 7.2.7 [Processes], page 518, and See Section 7.2.10 [Pipes], page 529, for more information.

4.4 Using Guile Interactively

When you start up Guile by typing just `guile`, without a `-c` argument or the name of a script to execute, you get an interactive interpreter where you can enter Scheme expressions, and Guile will evaluate them and print the results for you. Here are some simple examples.

```
scheme@(guile-user)> (+ 3 4 5)
$1 = 12
scheme@(guile-user)> (display "Hello world!\n")
Hello world!
scheme@(guile-user)> (values 'a 'b)
$2 = a
$3 = b
```

This mode of use is called a *REPL*, which is short for "Read-Eval-Print Loop", because the Guile interpreter first reads the expression that you have typed, then evaluates it, and then prints the result.

The prompt shows you what language and module you are in. In this case, the current language is `scheme`, and the current module is `(guile-user)`. See Section 6.24 [Other Languages], page 461, for more information on Guile's support for languages other than Scheme.

4.4.1 The Init File, `~/.guile`

When run interactively, Guile will load a local initialization file from `~/.guile`. This file should contain Scheme expressions for evaluation.

This facility lets the user customize their interactive Guile environment, pulling in extra modules or parameterizing the REPL implementation.

To run Guile without loading the init file, use the `-q` command-line option.

4.4.2 Readline

To make it easier for you to repeat and vary previously entered expressions, or to edit the expression that you're typing in, Guile can use the GNU Readline library. This is not enabled by default because of licensing reasons, but all you need to activate Readline is the following pair of lines.

```
scheme@(guile-user)> (use-modules (ice-9 readline))
scheme@(guile-user)> (activate-readline)
```

It's a good idea to put these two lines (without the `scheme@(guile-user)>` prompts) in your `.guile` file. See Section 4.4.1 [Init File], page 48, for more on `.guile`.

4.4.3 Value History

Just as Readline helps you to reuse a previous input line, *value history* allows you to use the *result* of a previous evaluation in a new expression. When value history is enabled, each evaluation result is automatically assigned to the next in the sequence of variables `$1`, `$2`, ... You can then use these variables in subsequent expressions.

```
scheme@(guile-user)> (iota 10)
$1 = (0 1 2 3 4 5 6 7 8 9)
scheme@(guile-user)> (apply * (cdr $1))
$2 = 362880
scheme@(guile-user)> (sqrt $2)
$3 = 602.3952191045344
scheme@(guile-user)> (cons $2 $1)
$4 = (362880 0 1 2 3 4 5 6 7 8 9)
```

Value history is enabled by default, because Guile's REPL imports the `(ice-9 history)` module. Value history may be turned off or on within the repl, using the options interface:

```
scheme@(guile-user)> ,option value-history #f
scheme@(guile-user)> 'foo
foo
scheme@(guile-user)> ,option value-history #t
scheme@(guile-user)> 'bar
$5 = bar
```

Note that previously recorded values are still accessible, even if value history is off. In rare cases, these references to past computations can cause Guile to use too much memory. One may clear these values, possibly enabling garbage collection, via the `clear-value-history!` procedure, described below.

The programmatic interface to value history is in a module:

```
(use-modules (ice-9 history))
```

`value-history-enabled?` [Scheme Procedure]
Return true if value history is enabled, or false otherwise.

`enable-value-history!` [Scheme Procedure]
Turn on value history, if it was off.

`disable-value-history!` [Scheme Procedure]
Turn off value history, if it was on.

`clear-value-history!` [Scheme Procedure]
Clear the value history. If the stored values are not captured by some other data structure or closure, they may then be reclaimed by the garbage collector.

4.4.4 REPL Commands

The REPL exists to read expressions, evaluate them, and then print their results. But sometimes one wants to tell the REPL to evaluate an expression in a different way, or to do something else altogether. A user can affect the way the REPL works with a *REPL command*.

The previous section had an example of a command, in the form of `,option`.

```
scheme@(guile-user)> ,option value-history #t
```

Commands are distinguished from expressions by their initial comma ('`,`'). Since a comma cannot begin an expression in most languages, it is an effective indicator to the REPL that the following text forms a command, not an expression.

REPL commands are convenient because they are always there. Even if the current module doesn't have a binding for `pretty-print`, one can always `,pretty-print`.

The following sections document the various commands, grouped together by functionality. Many of the commands have abbreviations; see the online help (`,help`) for more information.

4.4.4.1 Help Commands

When Guile starts interactively, it notifies the user that help can be had by typing '`,help`'. Indeed, `help` is a command, and a particularly useful one, as it allows the user to discover the rest of the commands.

`help` [`all` | *group* | [`-c`] *command*] [REPL Command]
Show help.

With one argument, tries to look up the argument as a group name, giving help on that group if successful. Otherwise tries to look up the argument as a command, giving help on the command.

If there is a command whose name is also a group name, use the '`-c` *`command`*' form to give help on the command instead of the group.

Without any argument, a list of help commands and command groups are displayed.

`show` [*topic*] [REPL Command]
Gives information about Guile.

With one argument, tries to show a particular piece of information; currently supported topics are 'warranty' (or 'w'), 'copying' (or 'c'), and 'version' (or 'v').

Without any argument, a list of topics is displayed.

`apropos` *regexp* [REPL Command]
Find bindings/modules/packages.

`describe` *obj* [REPL Command]
Show description/documentation.

4.4.4.2 Module Commands

`module` [*module*] [REPL Command]
Change modules / Show current module.

`import` *module ...* [REPL Command]
Import modules / List those imported.

`load` *file* [REPL Command]
Load a file in the current module.

`reload` [*module*] [REPL Command]
Reload the given module, or the current module if none was given.

`binding` [REPL Command]
List current bindings.

`in` *module expression* [REPL Command]
`in` *module command arg ...* [REPL Command]
Evaluate an expression, or alternatively, execute another meta-command in the context of a module. For example, '`,in (foo bar) ,binding`' will show the bindings in the module `(foo bar)`.

4.4.4.3 Language Commands

`language` *language* [REPL Command]
Change languages.

4.4.4.4 Compile Commands

`compile` *exp* [REPL Command]
Generate compiled code.

`compile-file` *file* [REPL Command]
Compile a file.

`expand` *exp* [REPL Command]
Expand any macros in a form.

`optimize` *exp* [REPL Command]
Run the optimizer on a piece of code and print the result.

`disassemble` *exp* [REPL Command]
Disassemble a compiled procedure.

`disassemble-file` *file* [REPL Command]
Disassemble a file.

4.4.4.5 Profile Commands

`time` *exp* [REPL Command]
Time execution.

`profile` *exp* [*#:hz hz=100*] [*#:count-calls? count-calls?=#f*] [*#:display-style display-style=list*] [REPL Command]
Profile execution of an expression. This command compiled *exp* and then runs it within the statprof profiler, passing all keyword options to the `statprof` procedure. For more on statprof and on the the options available to this command, See Section 7.20 [Statprof], page 745.

`trace` *exp* [*#:width w*] [*#:max-indent i*] [REPL Command]
Trace execution.

By default, the trace will limit its width to the width of your terminal, or *width* if specified. Nested procedure invocations will be printed farther to the right, though if the width of the indentation passes the *max-indent*, the indentation is abbreviated.

These REPL commands can also be called as regular functions in scheme code on including the `(ice-9 time)` module.

4.4.4.6 Debug Commands

These debugging commands are only available within a recursive REPL; they do not work at the top level.

`backtrace` [*count*] [*#:width w*] [*#:full? f*] [REPL Command]
Print a backtrace.

Print a backtrace of all stack frames, or innermost *count* frames. If *count* is negative, the last *count* frames will be shown.

`up` [*count*] [REPL Command]
Select a calling stack frame.

Select and print stack frames that called this one. An argument says how many frames up to go.

`down` [*count*] [REPL Command]
Select a called stack frame.

Select and print stack frames called by this one. An argument says how many frames down to go.

`frame` [*idx*] [REPL Command]
Show a frame.

Show the selected frame. With an argument, select a frame by index, then show it.

`locals` [REPL Command]
Show local variables.

Show locally-bound variables in the selected frame.

`error-message` [REPL Command]
`error` [REPL Command]
Show error message.

Display the message associated with the error that started the current debugging REPL.

`registers` [REPL Command]
Show the VM registers associated with the current frame.

See Section 9.3.3 [Stack Layout], page 829, for more information on VM stack frames.

`width` [*cols*] [REPL Command]
Sets the number of display columns in the output of `,backtrace` and `,locals` to *cols*. If *cols* is not given, the width of the terminal is used.

The next 3 commands work at any REPL.

`break` *proc* [REPL Command]
Set a breakpoint at *proc*.

`break-at-source` *file line* [REPL Command]
Set a breakpoint at the given source location.

`tracepoint` *proc* [REPL Command]
: Set a tracepoint on the given procedure. This will cause all calls to the procedure to print out a tracing message. See Section 6.26.4.4 [Tracing Traps], page 489, for more information.

The rest of the commands in this subsection all apply only when the stack is *continuable* — in other words when it makes sense for the program that the stack comes from to continue running. Usually this means that the program stopped because of a trap or a breakpoint.

`step` [REPL Command]
: Tell the debugged program to step to the next source location.

`next` [REPL Command]
: Tell the debugged program to step to the next source location in the same frame. (See Section 6.26.4 [Traps], page 484, for the details of how this works.)

`finish` [REPL Command]
: Tell the program being debugged to continue running until the completion of the current stack frame, and at that time to print the result and reenter the REPL.

4.4.4.7 Inspect Commands

`inspect` *exp* [REPL Command]
: Inspect the result(s) of evaluating *exp*.

`pretty-print` *exp* [REPL Command]
: Pretty-print the result(s) of evaluating *exp*.

4.4.4.8 System Commands

`gc` [REPL Command]
: Garbage collection.

`statistics` [REPL Command]
: Display statistics.

`option` [*name*] [*exp*] [REPL Command]
: With no arguments, lists all options. With one argument, shows the current value of the *name* option. With two arguments, sets the *name* option to the result of evaluating the Scheme expression *exp*.

`quit` [REPL Command]
: Quit this session.

Current REPL options include:

`compile-options`
: The options used when compiling expressions entered at the REPL. See Section 6.18.5 [Compilation], page 389, for more on compilation options.

`interp`
: Whether to interpret or compile expressions given at the REPL, if such a choice is available. Off by default (indicating compilation).

`prompt` A customized REPL prompt. `#f` by default, indicating the default prompt.

`print` A procedure of two arguments used to print the result of evaluating each expression. The arguments are the current REPL and the value to print. By default, `#f`, to use the default procedure.

`value-history`
Whether value history is on or not. See Section 4.4.3 [Value History], page 48.

`on-error` What to do when an error happens. By default, `debug`, meaning to enter the debugger. Other values include `backtrace`, to show a backtrace without entering the debugger, or `report`, to simply show a short error printout.

Default values for REPL options may be set using `repl-default-option-set!` from `(system repl common)`:

`repl-default-option-set!` *key value* [Scheme Procedure]
Set the default value of a REPL option. This function is particularly useful in a user's init file. See Section 4.4.1 [Init File], page 48.

4.4.5 Error Handling

When code being evaluated from the REPL hits an error, Guile enters a new prompt, allowing you to inspect the context of the error.

```
scheme@(guile-user)> (map string-append '("a" "b") '("c" #\d))
ERROR: In procedure string-append:
ERROR: Wrong type (expecting string): #\d
Entering a new prompt.  Type ',bt' for a backtrace or ',q' to continue.
scheme@(guile-user) [1]>
```

The new prompt runs inside the old one, in the dynamic context of the error. It is a recursive REPL, augmented with a reified representation of the stack, ready for debugging.

`,backtrace` (abbreviated `,bt`) displays the Scheme call stack at the point where the error occurred:

```
scheme@(guile-user) [1]> ,bt
           1 (map #<procedure string-append _> ("a" "b") ("c" #\d))
           0 (string-append "b" #\d)
```

In the above example, the backtrace doesn't have much source information, as `map` and `string-append` are both primitives. But in the general case, the space on the left of the backtrace indicates the line and column in which a given procedure calls another.

You can exit a recursive REPL in the same way that you exit any REPL: via '`(quit)`', '`,quit`' (abbreviated '`,q`'), or *`C-d`*, among other options.

4.4.6 Interactive Debugging

A recursive debugging REPL exposes a number of other meta-commands that inspect the state of the computation at the time of the error. These commands allow you to

- display the Scheme call stack at the point where the error occurred;
- move up and down the call stack, to see in detail the expression being evaluated, or the procedure being applied, in each *frame*; and

- examine the values of variables and expressions in the context of each frame.

See Section 4.4.4.6 [Debug Commands], page 52, for documentation of the individual commands. This section aims to give more of a walkthrough of a typical debugging session.

First, we're going to need a good error. Let's try to macroexpand the expression `(unquote foo)`, outside of a `quasiquote` form, and see how the macroexpander reports this error.

```
scheme@(guile-user)> (macroexpand '(unquote foo))
ERROR: In procedure macroexpand:
ERROR: unquote: expression not valid outside of quasiquote in (unquote foo)
Entering a new prompt.  Type ',bt' for a backtrace or ',q' to continue.
scheme@(guile-user) [1]>
```

The `backtrace` command, which can also be invoked as `bt`, displays the call stack (aka backtrace) at the point where the debugger was entered:

```
scheme@(guile-user) [1]> ,bt
In ice-9/psyntax.scm:
  1130:21  3 (chi-top (unquote foo) () ((top)) e (eval) (hygiene #))
  1071:30  2 (syntax-type (unquote foo) () ((top)) #f #f (# #) #f)
  1368:28  1 (chi-macro #<procedure de9360 at ice-9/psyntax.scm...> ...)
In unknown file:
           0 (scm-error syntax-error macroexpand "~a: ~a in ~a" # #f)
```

A call stack consists of a sequence of stack *frames*, with each frame describing one procedure which is waiting to do something with the values returned by another. Here we see that there are four frames on the stack.

Note that `macroexpand` is not on the stack – it must have made a tail call to `chi-top`, as indeed we would find if we searched `ice-9/psyntax.scm` for its definition.

When you enter the debugger, the innermost frame is selected, which means that the commands for getting information about the "current" frame, or for evaluating expressions in the context of the current frame, will do so by default with respect to the innermost frame. To select a different frame, so that these operations will apply to it instead, use the `up`, `down` and `frame` commands like this:

```
scheme@(guile-user) [1]> ,up
In ice-9/psyntax.scm:
  1368:28  1 (chi-macro #<procedure de9360 at ice-9/psyntax.scm...> ...)
scheme@(guile-user) [1]> ,frame 3
In ice-9/psyntax.scm:
  1130:21  3 (chi-top (unquote foo) () ((top)) e (eval) (hygiene #))
scheme@(guile-user) [1]> ,down
In ice-9/psyntax.scm:
  1071:30  2 (syntax-type (unquote foo) () ((top)) #f #f (# #) #f)
```

Perhaps we're interested in what's going on in frame 2, so we take a look at its local variables:

```
scheme@(guile-user) [1]> ,locals
  Local variables:
  $1 = e = (unquote foo)
```

```
$2 = r = ()
$3 = w = ((top))
$4 = s = #f
$5 = rib = #f
$6 = mod = (hygiene guile-user)
$7 = for-car? = #f
$8 = first = unquote
$9 = ftype = macro
$10 = fval = #<procedure de9360 at ice-9/psyntax.scm:2817:2 (x)>
$11 = fe = unquote
$12 = fw = ((top))
$13 = fs = #f
$14 = fmod = (hygiene guile-user)
```

All of the values are accessible by their value-history names (`$n`):

```
scheme@(guile-user) [1]> $10
$15 = #<procedure de9360 at ice-9/psyntax.scm:2817:2 (x)>
```

We can even invoke the procedure at the REPL directly:

```
scheme@(guile-user) [1]> ($10 'not-going-to-work)
ERROR: In procedure macroexpand:
ERROR: source expression failed to match any pattern in not-going-to-work
Entering a new prompt.  Type ',bt' for a backtrace or ',q' to continue.
```

Well at this point we've caused an error within an error. Let's just quit back to the top level:

```
scheme@(guile-user) [2]> ,q
scheme@(guile-user) [1]> ,q
scheme@(guile-user)>
```

Finally, as a word to the wise: hackers close their REPL prompts with `C-d`.

4.5 Using Guile in Emacs

Any text editor can edit Scheme, but some are better than others. Emacs is the best, of course, and not just because it is a fine text editor. Emacs has good support for Scheme out of the box, with sensible indentation rules, parenthesis-matching, syntax highlighting, and even a set of keybindings for structural editing, allowing navigation, cut-and-paste, and transposition operations that work on balanced S-expressions.

As good as it is, though, two things will vastly improve your experience with Emacs and Guile.

The first is Taylor Campbell's Paredit (`http://www.emacswiki.org/emacs/ParEdit`). You should not code in any dialect of Lisp without Paredit. (They say that unopinionated writing is boring—hence this tone—but it's the truth, regardless.) Paredit is the bee's knees.

The second is José Antonio Ortega Ruiz's Geiser (`http://www.nongnu.org/geiser/`). Geiser complements Emacs' `scheme-mode` with tight integration to running Guile processes via a `comint-mode` REPL buffer.

Of course there are keybindings to switch to the REPL, and a good REPL environment, but Geiser goes beyond that, providing:

- Form evaluation in the context of the current file's module.
- Macro expansion.
- File/module loading and/or compilation.
- Namespace-aware identifier completion (including local bindings, names visible in the current module, and module names).
- Autodoc: the echo area shows information about the signature of the procedure/macro around point automatically.
- Jump to definition of identifier at point.
- Access to documentation (including docstrings when the implementation provides it).
- Listings of identifiers exported by a given module.
- Listings of callers/callees of procedures.
- Rudimentary support for debugging and error navigation.
- Support for multiple, simultaneous REPLs.

See Geiser's web page at `http://www.nongnu.org/geiser/`, for more information.

4.6 Using Guile Tools

Guile also comes with a growing number of command-line utilities: a compiler, a disassembler, some module inspectors, and in the future, a system to install Guile packages from the internet. These tools may be invoked using the `guild` program.

```
$ guild compile -o foo.go foo.scm
wrote ‘foo.go’
```

This program used to be called `guile-tools` up to Guile version 2.0.1, and for backward compatibility it still may be called as such. However we changed the name to `guild`, not only because it is pleasantly shorter and easier to read, but also because this tool will serve to bind Guile wizards together, by allowing hackers to share code with each other using a CPAN-like system.

See Section 6.18.5 [Compilation], page 389, for more on `guild compile`.

A complete list of guild scripts can be had by invoking `guild list`, or simply `guild`.

4.7 Installing Site Packages

At some point, you will probably want to share your code with other people. To do so effectively, it is important to follow a set of common conventions, to make it easy for the user to install and use your package.

The first thing to do is to install your Scheme files where Guile can find them. When Guile goes to find a Scheme file, it will search a *load path* to find the file: first in Guile's own path, then in paths for *site packages*. A site package is any Scheme code that is installed and not part of Guile itself. See Section 6.18.7 [Load Paths], page 393, for more on load paths.

There are several site paths, for historical reasons, but the one that should generally be used can be obtained by invoking the `%site-dir` procedure. See Section 6.23.1 [Build

Config], page 456. If Guile 3.0 is installed on your system in `/usr/`, then `(%site-dir)` will be `/usr/share/guile/site/3.0`. Scheme files should be installed there.

If you do not install compiled `.go` files, Guile will compile your modules and programs when they are first used, and cache them in the user's home directory. See Section 6.18.5 [Compilation], page 389, for more on auto-compilation. However, it is better to compile the files before they are installed, and to just copy the files to a place that Guile can find them.

As with Scheme files, Guile searches a path to find compiled `.go` files, the `%load-compiled-path`. By default, this path has two entries: a path for Guile's files, and a path for site packages. You should install your `.go` files into the latter directory, whose value is returned by invoking the `%site-ccache-dir` procedure. As in the previous example, if Guile 3.0 is installed on your system in `/usr/`, then `(%site-ccache-dir)` site packages will be `/usr/lib/guile/3.0/site-ccache`.

Note that a `.go` file will only be loaded in preference to a `.scm` file if it is newer. For that reason, you should install your Scheme files first, and your compiled files second. See Section 6.18.7 [Load Paths], page 393, for more on the loading process.

Finally, although this section is only about Scheme, sometimes you need to install C extensions too. Shared libraries should be installed in the *extensions dir*. This value can be had from the build config (see Section 6.23.1 [Build Config], page 456). Again, if Guile 3.0 is installed on your system in `/usr/`, then the extensions dir will be `/usr/lib/guile/3.0/extensions`.

4.8 Distributing Guile Code

There's a tool that doesn't come bundled with Guile and yet can be very useful in your day to day experience with it. This tool is Hall (`https://gitlab.com/a-sassmannshausen/guile-hall`).

Hall helps you create, manage, and package your Guile projects through a simple command-line interface. When you start a new project, Hall creates a folder containing a scaffold of your new project. It contains a directory for your tests, for your libraries, for your scripts and for your documentation. This means you immediately know where to put the files you are hacking on.

In addition, the scaffold will include your basic "Autotools" setup, so you don't have to take care of that yourself (see Section "The GNU Build System" in *Autoconf: Creating Automatic Configuration Scripts*, for more information on the GNU "Autotools"). Having Autotools set up with your project means you can immediately start hacking on your project without worrying about whether your code will work on other people's computers. Hall can also generate package definitions for the GNU Guix package manager, making it easy for Guix users to install it.

5 Programming in C

This part of the manual explains the general concepts that you need to understand when interfacing to Guile from C. You will learn about how the latent typing of Scheme is embedded into the static typing of C, how the garbage collection of Guile is made available to C code, and how continuations influence the control flow in a C program.

This knowledge should make it straightforward to add new functions to Guile that can be called from Scheme. Adding new data types is also possible and is done by defining *foreign objects*.

The Section 5.7 [Programming Overview], page 83, section of this part contains general musings and guidelines about programming with Guile. It explores different ways to design a program around Guile, or how to embed Guile into existing programs.

For a pedagogical yet detailed explanation of how the data representation of Guile is implemented, See Section 9.2 [Data Representation], page 818. You don't need to know the details given there to use Guile from C, but they are useful when you want to modify Guile itself or when you are just curious about how it is all done.

For detailed reference information on the variables, functions etc. that make up Guile's application programming interface (API), See Chapter 6 [API Reference], page 99.

5.1 Parallel Installations

Guile provides strong API and ABI stability guarantees during stable series, so that if a user writes a program against Guile version 2.2.3, it will be compatible with some future version 2.2.7. We say in this case that 2.2 is the *effective version*, composed of the major and minor versions, in this case 2 and 2.

Users may install multiple effective versions of Guile, with each version's headers, libraries, and Scheme files under their own directories. This provides the necessary stability guarantee for users, while also allowing Guile developers to evolve the language and its implementation.

However, parallel installability does have a down-side, in that users need to know which version of Guile to ask for, when they build against Guile. Guile solves this problem by installing a file to be read by the `pkg-config` utility, a tool to query installed packages by name. Guile encodes the version into its pkg-config name, so that users can ask for `guile-2.2` or `guile-3.0`, as appropriate.

For effective version 3.0, for example, you would invoke `pkg-config --cflags --libs guile-3.0` to get the compilation and linking flags necessary to link to version 3.0 of Guile. You would typically run `pkg-config` during the configuration phase of your program and use the obtained information in the Makefile.

Guile's `pkg-config` file, `guile-3.0.pc`, defines additional useful variables:

`sitedir` The default directory where Guile looks for Scheme source and compiled files (see Section 4.7 [Installing Site Packages], page 57). Run `pkg-config guile-3.0 --variable=sitedir` to see its value. See Section 5.8.2 [Autoconf Macros], page 94, for more on how to use it from Autoconf.

`extensiondir`
: The default directory where Guile looks for extensions—i.e., shared libraries providing additional features (see Section 6.21.4 [Modules and Extensions], page 432). Run `pkg-config guile-3.0 --variable=extensiondir` to see its value.

`guile`
`guild`
: The absolute file name of the `guile` and `guild` commands[1]. Run `pkg-config guile-3.0 --variable=guile` or `--variable=guild` to see their value.

  These variables allow users to deal with program name transformations that may be specified when configuring Guile with `--program-transform-name`, `--program-suffix`, or `--program-prefix` (see Section "Transformation Options" in *GNU Autoconf Manual*).

See the `pkg-config` man page, for more information, or its web site, `http://pkg-config.freedesktop.org/`. See Section 5.8 [Autoconf Support], page 94, for more on checking for Guile from within a `configure.ac` file.

5.2 Linking Programs With Guile

This section covers the mechanics of linking your program with Guile on a typical POSIX system.

The header file `<libguile.h>` provides declarations for all of Guile's functions and constants. You should `#include` it at the head of any C source file that uses identifiers described in this manual. Once you've compiled your source files, you need to link them against the Guile object code library, `libguile`.

As noted in the previous section, `<libguile.h>` is not in the default search path for headers. The following command lines give respectively the C compilation and link flags needed to build programs using Guile 3.0:

```
pkg-config guile-3.0 --cflags
pkg-config guile-3.0 --libs
```

5.2.1 Guile Initialization Functions

To initialize Guile, you can use one of several functions. The first, `scm_with_guile`, is the most portable way to initialize Guile. It will initialize Guile when necessary and then call a function that you can specify. Multiple threads can call `scm_with_guile` concurrently and it can also be called more than once in a given thread. The global state of Guile will survive from one call of `scm_with_guile` to the next. Your function is called from within `scm_with_guile` since the garbage collector of Guile needs to know where the stack of each thread is.

A second function, `scm_init_guile`, initializes Guile for the current thread. When it returns, you can use the Guile API in the current thread. This function employs some non-portable magic to learn about stack bounds and might thus not be available on all platforms.

One common way to use Guile is to write a set of C functions which perform some useful task, make them callable from Scheme, and then link the program with Guile. This yields

[1] The `guile` and `guild` variables defined starting from Guile version 2.0.12.

a Scheme interpreter just like `guile`, but augmented with extra functions for some specific application — a special-purpose scripting language.

In this situation, the application should probably process its command-line arguments in the same manner as the stock Guile interpreter. To make that straightforward, Guile provides the `scm_boot_guile` and `scm_shell` function.

For more about these functions, see Section 6.4 [Initialization], page 101.

5.2.2 A Sample Guile Main Program

Here is `simple-guile.c`, source code for a `main` and an `inner_main` function that will produce a complete Guile interpreter.

```
/* simple-guile.c --- Start Guile from C.  */

#include <libguile.h>

static void
inner_main (void *closure, int argc, char **argv)
{
  /* preparation */
  scm_shell (argc, argv);
  /* after exit */
}

int
main (int argc, char **argv)
{
  scm_boot_guile (argc, argv, inner_main, 0);
  return 0; /* never reached, see inner_main */
}
```

The `main` function calls `scm_boot_guile` to initialize Guile, passing it `inner_main`. Once `scm_boot_guile` is ready, it invokes `inner_main`, which calls `scm_shell` to process the command-line arguments in the usual way.

5.2.3 Building the Example with Make

Here is a Makefile which you can use to compile the example program. It uses `pkg-config` to learn about the necessary compiler and linker flags.

```
# Use GCC, if you have it installed.
CC=gcc

# Tell the C compiler where to find <libguile.h>
CFLAGS=‘pkg-config --cflags guile-3.0‘

# Tell the linker what libraries to use and where to find them.
LIBS=‘pkg-config --libs guile-3.0‘

simple-guile: simple-guile.o
```

```
        ${CC} simple-guile.o ${LIBS} -o simple-guile

simple-guile.o: simple-guile.c
        ${CC} -c ${CFLAGS} simple-guile.c
```

5.2.4 Building the Example with Autoconf

If you are using the GNU Autoconf package to make your application more portable, Autoconf will settle many of the details in the Makefile automatically, making it much simpler and more portable; we recommend using Autoconf with Guile. Here is a `configure.ac` file for `simple-guile` that uses the standard `PKG_CHECK_MODULES` macro to check for Guile. Autoconf will process this file into a `configure` script. We recommend invoking Autoconf via the `autoreconf` utility.

```
AC_INIT(simple-guile.c)

# Find a C compiler.
AC_PROG_CC

# Check for Guile
PKG_CHECK_MODULES([GUILE], [guile-3.0])

# Generate a Makefile, based on the results.
AC_OUTPUT(Makefile)
```

Run `autoreconf -vif` to generate `configure`.

Here is a `Makefile.in` template, from which the `configure` script produces a Makefile customized for the host system:

```
# The configure script fills in these values.
CC=@CC@
CFLAGS=@GUILE_CFLAGS@
LIBS=@GUILE_LIBS@

simple-guile: simple-guile.o
        ${CC} simple-guile.o ${LIBS} -o simple-guile
simple-guile.o: simple-guile.c
        ${CC} -c ${CFLAGS} simple-guile.c
```

The developer should use Autoconf to generate the `configure` script from the `configure.ac` template, and distribute `configure` with the application. Here's how a user might go about building the application:

```
$ ls
Makefile.in     configure*      configure.ac    simple-guile.c
$ ./configure
checking for gcc... ccache gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
```

```
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether ccache gcc accepts -g... yes
checking for ccache gcc option to accept ISO C89... none needed
checking for pkg-config... /usr/bin/pkg-config
checking pkg-config is at least version 0.9.0... yes
checking for GUILE... yes
configure: creating ./config.status
config.status: creating Makefile
$ make
[...]
$ ./simple-guile
guile> (+ 1 2 3)
6
guile> (getpwnam "jimb")
#("jimb" "83Z7d75W2tyJQ" 4008 10 "Jim Blandy" "/u/jimb"
  "/usr/local/bin/bash")
guile> (exit)
$
```

5.3 Linking Guile with Libraries

The previous section has briefly explained how to write programs that make use of an embedded Guile interpreter. But sometimes, all you want to do is make new primitive procedures and data types available to the Scheme programmer. Writing a new version of `guile` is inconvenient in this case and it would in fact make the life of the users of your new features needlessly hard.

For example, suppose that there is a program `guile-db` that is a version of Guile with additional features for accessing a database. People who want to write Scheme programs that use these features would have to use `guile-db` instead of the usual `guile` program. Now suppose that there is also a program `guile-gtk` that extends Guile with access to the popular Gtk+ toolkit for graphical user interfaces. People who want to write GUIs in Scheme would have to use `guile-gtk`. Now, what happens when you want to write a Scheme application that uses a GUI to let the user access a database? You would have to write a *third* program that incorporates both the database stuff and the GUI stuff. This might not be easy (because `guile-gtk` might be a quite obscure program, say) and taking this example further makes it easy to see that this approach can not work in practice.

It would have been much better if both the database features and the GUI feature had been provided as libraries that can just be linked with `guile`. Guile makes it easy to do just this, and we encourage you to make your extensions to Guile available as libraries whenever possible.

You write the new primitive procedures and data types in the normal fashion, and link them into a shared library instead of into a stand-alone program. The shared library can then be loaded dynamically by Guile.

5.3.1 A Sample Guile Extension

This section explains how to make the Bessel functions of the C library available to Scheme. First we need to write the appropriate glue code to convert the arguments and return values of the functions from Scheme to C and back. Additionally, we need a function that will add them to the set of Guile primitives. Because this is just an example, we will only implement this for the `j0` function.

Consider the following file `bessel.c`.

```
#include <math.h>
#include <libguile.h>

SCM
j0_wrapper (SCM x)
{
  return scm_from_double (j0 (scm_to_double (x)));
}

void
init_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

This C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux:

```
gcc ‘pkg-config --cflags guile-3.0‘ \
  -shared -o libguile-bessel.so -fPIC bessel.c
```

For creating shared libraries portably, we recommend the use of GNU Libtool (see Section “Introduction” in *GNU Libtool*).

A shared library can be loaded into a running Guile process with the function `load-extension`. In addition to the name of the library to load, this function also expects the name of a function from that library that will be called to initialize it. For our example, we are going to call the function `init_bessel` which will make `j0_wrapper` available to Scheme programs with the name `j0`. Note that we do not specify a filename extension such as `.so` when invoking `load-extension`. The right extension for the host platform will be provided automatically.

```
(load-extension "libguile-bessel" "init_bessel")
(j0 2)
⇒ 0.223890779141236
```

For this to work, `load-extension` must be able to find `libguile-bessel`, of course. It will look in the places that are usual for your operating system, and it will additionally look into the directories listed in the `LTDL_LIBRARY_PATH` environment variable.

To see how these Guile extensions via shared libraries relate to the module system, See Section 2.5.3 [Putting Extensions into Modules], page 12.

5.4 General concepts for using libguile

When you want to embed the Guile Scheme interpreter into your program or library, you need to link it against the `libguile` library (see Section 5.2 [Linking Programs With Guile], page 60). Once you have done this, your C code has access to a number of data types and

functions that can be used to invoke the interpreter, or make new functions that you have written in C available to be called from Scheme code, among other things.

Scheme is different from C in a number of significant ways, and Guile tries to make the advantages of Scheme available to C as well. Thus, in addition to a Scheme interpreter, libguile also offers dynamic types, garbage collection, continuations, arithmetic on arbitrary sized numbers, and other things.

The two fundamental concepts are dynamic types and garbage collection. You need to understand how libguile offers them to C programs in order to use the rest of libguile. Also, the more general control flow of Scheme caused by continuations needs to be dealt with.

Running asynchronous signal handlers and multi-threading is known to C code already, but there are of course a few additional rules when using them together with libguile.

5.4.1 Dynamic Types

Scheme is a dynamically-typed language; this means that the system cannot, in general, determine the type of a given expression at compile time. Types only become apparent at run time. Variables do not have fixed types; a variable may hold a pair at one point, an integer at the next, and a thousand-element vector later. Instead, values, not variables, have fixed types.

In order to implement standard Scheme functions like `pair?` and `string?` and provide garbage collection, the representation of every value must contain enough information to accurately determine its type at run time. Often, Scheme systems also use this information to determine whether a program has attempted to apply an operation to an inappropriately typed value (such as taking the `car` of a string).

Because variables, pairs, and vectors may hold values of any type, Scheme implementations use a uniform representation for values — a single type large enough to hold either a complete value or a pointer to a complete value, along with the necessary typing information.

In Guile, this uniform representation of all Scheme values is the C type `SCM`. This is an opaque type and its size is typically equivalent to that of a pointer to `void`. Thus, `SCM` values can be passed around efficiently and they take up reasonably little storage on their own.

The most important rule is: You never access a `SCM` value directly; you only pass it to functions or macros defined in libguile.

As an obvious example, although a `SCM` variable can contain integers, you can of course not compute the sum of two `SCM` values by adding them with the C `+` operator. You must use the libguile function `scm_sum`.

Less obvious and therefore more important to keep in mind is that you also cannot directly test `SCM` values for trueness. In Scheme, the value `#f` is considered false and of course a `SCM` variable can represent that value. But there is no guarantee that the `SCM` representation of `#f` looks false to C code as well. You need to use `scm_is_true` or `scm_is_false` to test a `SCM` value for trueness or falseness, respectively.

You also can not directly compare two `SCM` values to find out whether they are identical (that is, whether they are `eq?` in Scheme terms). You need to use `scm_is_eq` for this.

The one exception is that you can directly assign a `SCM` value to a `SCM` variable by using the C `=` operator.

The following (contrived) example shows how to do it right. It implements a function of two arguments (*a* and *flag*) that returns *a*+1 if *flag* is true, else it returns *a* unchanged.

```
SCM
my_incrementing_function (SCM a, SCM flag)
{
  SCM result;

  if (scm_is_true (flag))
    result = scm_sum (a, scm_from_int (1));
  else
    result = a;

  return result;
}
```

Often, you need to convert between `SCM` values and appropriate C values. For example, we needed to convert the integer `1` to its `SCM` representation in order to add it to *a*. Libguile provides many function to do these conversions, both from C to `SCM` and from `SCM` to C.

The conversion functions follow a common naming pattern: those that make a `SCM` value from a C value have names of the form `scm_from_`*`type`*` (...)` and those that convert a `SCM` value to a C value use the form `scm_to_`*`type`*` (...)`.

However, it is best to avoid converting values when you can. When you must combine C values and `SCM` values in a computation, it is often better to convert the C values to `SCM` values and do the computation by using libguile functions than to the other way around (converting `SCM` to C and doing the computation some other way).

As a simple example, consider this version of `my_incrementing_function` from above:

```
SCM
my_other_incrementing_function (SCM a, SCM flag)
{
  int result;

  if (scm_is_true (flag))
    result = scm_to_int (a) + 1;
  else
    result = scm_to_int (a);

  return scm_from_int (result);
}
```

This version is much less general than the original one: it will only work for values *A* that can fit into a `int`. The original function will work for all values that Guile can represent and that `scm_sum` can understand, including integers bigger than `long long`, floating point numbers, complex numbers, and new numerical types that have been added to Guile by third-party libraries.

Also, computing with `SCM` is not necessarily inefficient. Small integers will be encoded directly in the `SCM` value, for example, and do not need any additional memory on the heap. See Section 9.2 [Data Representation], page 818, to find out the details.

Some special `SCM` values are available to C code without needing to convert them from C values:

Scheme value	C representation
`#f`	`SCM_BOOL_F`
`#t`	`SCM_BOOL_T`
`()`	`SCM_EOL`

In addition to `SCM`, Guile also defines the related type `scm_t_bits`. This is an unsigned integral type of sufficient size to hold all information that is directly contained in a `SCM` value. The `scm_t_bits` type is used internally by Guile to do all the bit twiddling explained in Section 9.2 [Data Representation], page 818, but you will encounter it occasionally in low-level user code as well.

5.4.2 Garbage Collection

As explained above, the `SCM` type can represent all Scheme values. Some values fit entirely into a `SCM` value (such as small integers), but other values require additional storage in the heap (such as strings and vectors). This additional storage is managed automatically by Guile. You don't need to explicitly deallocate it when a `SCM` value is no longer used.

Two things must be guaranteed so that Guile is able to manage the storage automatically: it must know about all blocks of memory that have ever been allocated for Scheme values, and it must know about all Scheme values that are still being used. Given this knowledge, Guile can periodically free all blocks that have been allocated but are not used by any active Scheme values. This activity is called *garbage collection*.

Guile's garbage collector will automatically discover references to `SCM` objects that originate in global variables, static data sections, function arguments or local variables on the C and Scheme stacks, and values in machine registers. Other references to `SCM` objects, such as those in other random data structures in the C heap that contain fields of type `SCM`, can be made visible to the garbage collector by calling the functions `scm_gc_protect_object` or `scm_permanent_object`. Collectively, these values form the "root set" of garbage collection; any value on the heap that is referenced directly or indirectly by a member of the root set is preserved, and all other objects are eligible for reclamation.

In Guile, garbage collection has two logical phases: the *mark phase*, in which the collector discovers the set of all live objects, and the *sweep phase*, in which the collector reclaims the resources associated with dead objects. The mark phase pauses the program and traces all `SCM` object references, starting with the root set. The sweep phase actually runs concurrently with the main program, incrementally reclaiming memory as needed by allocation.

In the mark phase, the garbage collector traces the Scheme stack and heap *precisely*. Because the Scheme stack and heap are managed by Guile, Guile can know precisely where in those data structures it might find references to other heap objects. This is not the case, unfortunately, for pointers on the C stack and static data segment. Instead of requiring the user to inform Guile about all variables in C that might point to heap objects, Guile traces the C stack and static data segment *conservatively*. That is to say, Guile just treats every word on the C stack and every C global variable as a potential reference in to the

Scheme heap[2]. Any value that looks like a pointer to a GC-managed object is treated as such, whether it actually is a reference or not. Thus, scanning the C stack and static data segment is guaranteed to find all actual references, but it might also find words that only accidentally look like references. These "false positives" might keep `SCM` objects alive that would otherwise be considered dead. While this might waste memory, keeping an object around longer than it strictly needs to is harmless. This is why this technique is called "conservative garbage collection". In practice, the wasted memory seems to be no problem, as the static C root set is almost always finite and small, given that the Scheme stack is separate from the C stack.

The stack of every thread is scanned in this way and the registers of the CPU and all other memory locations where local variables or function parameters might show up are included in this scan as well.

The consequence of the conservative scanning is that you can just declare local variables and function parameters of type `SCM` and be sure that the garbage collector will not free the corresponding objects.

However, a local variable or function parameter is only protected as long as it is really on the stack (or in some register). As an optimization, the C compiler might reuse its location for some other value and the `SCM` object would no longer be protected. Normally, this leads to exactly the right behavior: the compiler will only overwrite a reference when it is no longer needed and thus the object becomes unprotected precisely when the reference disappears, just as wanted.

There are situations, however, where a `SCM` object needs to be around longer than its reference from a local variable or function parameter. This happens, for example, when you retrieve some pointer from a foreign object and work with that pointer directly. The reference to the `SCM` foreign object might be dead after the pointer has been retrieved, but the pointer itself (and the memory pointed to) is still in use and thus the foreign object must be protected. The compiler does not know about this connection and might overwrite the `SCM` reference too early.

To get around this problem, you can use `scm_remember_upto_here_1` and its cousins. It will keep the compiler from overwriting the reference. See Section 5.5.4 [Foreign Object Memory Management], page 77.

5.4.3 Control Flow

Scheme has a more general view of program flow than C, both locally and non-locally.

Controlling the local flow of control involves things like gotos, loops, calling functions and returning from them. Non-local control flow refers to situations where the program jumps across one or more levels of function activations without using the normal call or return operations.

The primitive means of C for local control flow is the `goto` statement, together with `if`. Loops done with `for`, `while` or `do` could in principle be rewritten with just `goto` and `if`. In Scheme, the primitive means for local control flow is the *function call* (together with `if`).

[2] Note that Guile does not scan the C heap for references, so a reference to a `SCM` object from a memory segment allocated with `malloc` will have to use some other means to keep the `SCM` object alive. See Section 6.19.1 [Garbage Collection Functions], page 404.

Thus, the repetition of some computation in a loop is ultimately implemented by a function that calls itself, that is, by recursion.

This approach is theoretically very powerful since it is easier to reason formally about recursion than about gotos. In C, using recursion exclusively would not be practical, though, since it would eat up the stack very quickly. In Scheme, however, it is practical: function calls that appear in a *tail position* do not use any additional stack space (see Section 3.3.2 [Tail Calls], page 24).

A function call is in a tail position when it is the last thing the calling function does. The value returned by the called function is immediately returned from the calling function. In the following example, the call to `bar-1` is in a tail position, while the call to `bar-2` is not. (The call to `1-` in `foo-2` is in a tail position, though.)

```
(define (foo-1 x)
  (bar-1 (1- x)))

(define (foo-2 x)
  (1- (bar-2 x)))
```

Thus, when you take care to recurse only in tail positions, the recursion will only use constant stack space and will be as good as a loop constructed from gotos.

Scheme offers a few syntactic abstractions (`do` and *named* `let`) that make writing loops slightly easier.

But only Scheme functions can call other functions in a tail position: C functions can not. This matters when you have, say, two functions that call each other recursively to form a common loop. The following (unrealistic) example shows how one might go about determining whether a non-negative integer *n* is even or odd.

```
(define (my-even? n)
  (cond ((zero? n) #t)
        (else (my-odd? (1- n)))))

(define (my-odd? n)
  (cond ((zero? n) #f)
        (else (my-even? (1- n)))))
```

Because the calls to `my-even?` and `my-odd?` are in tail positions, these two procedures can be applied to arbitrary large integers without overflowing the stack. (They will still take a lot of time, of course.)

However, when one or both of the two procedures would be rewritten in C, it could no longer call its companion in a tail position (since C does not have this concept). You might need to take this consideration into account when deciding which parts of your program to write in Scheme and which in C.

In addition to calling functions and returning from them, a Scheme program can also exit non-locally from a function so that the control flow returns directly to an outer level. This means that some functions might not return at all.

Even more, it is not only possible to jump to some outer level of control, a Scheme program can also jump back into the middle of a function that has already exited. This might cause some functions to return more than once.

In general, these non-local jumps are done by invoking *continuations* that have previously been captured using `call-with-current-continuation`. Guile also offers a slightly restricted set of functions, `catch` and `throw`, that can only be used for non-local exits. This restriction makes them more efficient. Error reporting (with the function `error`) is implemented by invoking `throw`, for example. The functions `catch` and `throw` belong to the topic of *exceptions*.

Since Scheme functions can call C functions and vice versa, C code can experience the more general control flow of Scheme as well. It is possible that a C function will not return at all, or will return more than once. While C does offer `setjmp` and `longjmp` for non-local exits, it is still an unusual thing for C code. In contrast, non-local exits are very common in Scheme, mostly to report errors.

You need to be prepared for the non-local jumps in the control flow whenever you use a function from `libguile`: it is best to assume that any `libguile` function might signal an error or run a pending signal handler (which in turn can do arbitrary things).

It is often necessary to take cleanup actions when the control leaves a function non-locally. Also, when the control returns non-locally, some setup actions might be called for. For example, the Scheme function `with-output-to-port` needs to modify the global state so that `current-output-port` returns the port passed to `with-output-to-port`. The global output port needs to be reset to its previous value when `with-output-to-port` returns normally or when it is exited non-locally. Likewise, the port needs to be set again when control enters non-locally.

Scheme code can use the `dynamic-wind` function to arrange for the setting and resetting of the global state. C code can use the corresponding `scm_internal_dynamic_wind` function, or a `scm_dynwind_begin`/`scm_dynwind_end` pair together with suitable 'dynwind actions' (see Section 6.13.10 [Dynamic Wind], page 320).

Instead of coping with non-local control flow, you can also prevent it by erecting a *continuation barrier*, See Section 6.13.14 [Continuation Barriers], page 331. The function `scm_c_with_continuation_barrier`, for example, is guaranteed to return exactly once.

5.4.4 Asynchronous Signals

You can not call libguile functions from handlers for POSIX signals, but you can register Scheme handlers for POSIX signals such as `SIGINT`. These handlers do not run during the actual signal delivery. Instead, they are run when the program (more precisely, the thread that the handler has been registered for) reaches the next *safe point*.

The libguile functions themselves have many such safe points. Consequently, you must be prepared for arbitrary actions anytime you call a libguile function. For example, even `scm_cons` can contain a safe point and when a signal handler is pending for your thread, calling `scm_cons` will run this handler and anything might happen, including a non-local exit although `scm_cons` would not ordinarily do such a thing on its own.

If you do not want to allow the running of asynchronous signal handlers, you can block them temporarily with `scm_dynwind_block_asyncs`, for example. See Section 6.22.3 [Asyncs], page 445.

Since signal handling in Guile relies on safe points, you need to make sure that your functions do offer enough of them. Normally, calling libguile functions in the normal course of action is all that is needed. But when a thread might spent a long time in a code section

that calls no libguile function, it is good to include explicit safe points. This can allow the user to interrupt your code with `C-c`, for example.

You can do this with the macro `SCM_TICK`. This macro is syntactically a statement. That is, you could use it like this:

```
while (1)
  {
    SCM_TICK;
    do_some_work ();
  }
```

Frequent execution of a safe point is even more important in multi threaded programs, See Section 5.4.5 [Multi-Threading], page 71.

5.4.5 Multi-Threading

Guile can be used in multi-threaded programs just as well as in single-threaded ones.

Each thread that wants to use functions from libguile must put itself into *guile mode* and must then follow a few rules. If it doesn't want to honor these rules in certain situations, a thread can temporarily leave guile mode (but can no longer use libguile functions during that time, of course).

Threads enter guile mode by calling `scm_with_guile`, `scm_boot_guile`, or `scm_init_guile`. As explained in the reference documentation for these functions, Guile will then learn about the stack bounds of the thread and can protect the `SCM` values that are stored in local variables. When a thread puts itself into guile mode for the first time, it gets a Scheme representation and is listed by `all-threads`, for example.

Threads in guile mode can block (e.g., do blocking I/O) without causing any problems[3]; temporarily leaving guile mode with `scm_without_guile` before blocking slightly improves GC performance, though. For some common blocking operations, Guile provides convenience functions. For example, if you want to lock a pthread mutex while in guile mode, you might want to use `scm_pthread_mutex_lock` which is just like `pthread_mutex_lock` except that it leaves guile mode while blocking.

All libguile functions are (intended to be) robust in the face of multiple threads using them concurrently. This means that there is no risk of the internal data structures of libguile becoming corrupted in such a way that the process crashes.

A program might still produce nonsensical results, though. Taking hashtables as an example, Guile guarantees that you can use them from multiple threads concurrently and a hashtable will always remain a valid hashtable and Guile will not crash when you access it. It does not guarantee, however, that inserting into it concurrently from two threads will give useful results: only one insertion might actually happen, none might happen, or the table might in general be modified in a totally arbitrary manner. (It will still be a valid hashtable, but not the one that you might have expected.) Guile might also signal an error when it detects a harmful race condition.

Thus, you need to put in additional synchronizations when multiple threads want to use a single hashtable, or any other mutable Scheme object.

[3] In Guile 1.8, a thread blocking in guile mode would prevent garbage collection to occur. Thus, threads had to leave guile mode whenever they could block. This is no longer needed with Guile 2.*x*.

When writing C code for use with libguile, you should try to make it robust as well. An example that converts a list into a vector will help to illustrate. Here is a correct version:

```
SCM
my_list_to_vector (SCM list)
{
  SCM vector = scm_make_vector (scm_length (list), SCM_UNDEFINED);
  size_t len, i;

  len = scm_c_vector_length (vector);
  i = 0;
  while (i < len && scm_is_pair (list))
    {
      scm_c_vector_set_x (vector, i, scm_car (list));
      list = scm_cdr (list);
      i++;
    }

  return vector;
}
```

The first thing to note is that storing into a `SCM` location concurrently from multiple threads is guaranteed to be robust: you don't know which value wins but it will in any case be a valid `SCM` value.

But there is no guarantee that the list referenced by *list* is not modified in another thread while the loop iterates over it. Thus, while copying its elements into the vector, the list might get longer or shorter. For this reason, the loop must check both that it doesn't overrun the vector and that it doesn't overrun the list. Otherwise, `scm_c_vector_set_x` would raise an error if the index is out of range, and `scm_car` and `scm_cdr` would raise an error if the value is not a pair.

It is safe to use `scm_car` and `scm_cdr` on the local variable *list* once it is known that the variable contains a pair. The contents of the pair might change spontaneously, but it will always stay a valid pair (and a local variable will of course not spontaneously point to a different Scheme object).

Likewise, a vector such as the one returned by `scm_make_vector` is guaranteed to always stay the same length so that it is safe to only use scm_c_vector_length once and store the result. (In the example, *vector* is safe anyway since it is a fresh object that no other thread can possibly know about until it is returned from `my_list_to_vector`.)

Of course the behavior of `my_list_to_vector` is suboptimal when *list* does indeed get asynchronously lengthened or shortened in another thread. But it is robust: it will always return a valid vector. That vector might be shorter than expected, or its last elements might be unspecified, but it is a valid vector and if a program wants to rule out these cases, it must avoid modifying the list asynchronously.

Here is another version that is also correct:

```
SCM
my_pedantic_list_to_vector (SCM list)
{
```

```
  SCM vector = scm_make_vector (scm_length (list), SCM_UNDEFINED);
  size_t len, i;

  len = scm_c_vector_length (vector);
  i = 0;
  while (i < len)
    {
      scm_c_vector_set_x (vector, i, scm_car (list));
      list = scm_cdr (list);
      i++;
    }

  return vector;
}
```

This version relies on the error-checking behavior of `scm_car` and `scm_cdr`. When the list is shortened (that is, when *list* holds a non-pair), `scm_car` will throw an error. This might be preferable to just returning a half-initialized vector.

The API for accessing vectors and arrays of various kinds from C takes a slightly different approach to thread-robustness. In order to get at the raw memory that stores the elements of an array, you need to *reserve* that array as long as you need the raw memory. During the time an array is reserved, its elements can still spontaneously change their values, but the memory itself and other things like the size of the array are guaranteed to stay fixed. Any operation that would change these parameters of an array that is currently reserved will signal an error. In order to avoid these errors, a program should of course put suitable synchronization mechanisms in place. As you can see, Guile itself is again only concerned about robustness, not about correctness: without proper synchronization, your program will likely not be correct, but the worst consequence is an error message.

Real thread-safety often requires that a critical section of code is executed in a certain restricted manner. A common requirement is that the code section is not entered a second time when it is already being executed. Locking a mutex while in that section ensures that no other thread will start executing it, blocking asyncs ensures that no asynchronous code enters the section again from the current thread, and the error checking of Guile mutexes guarantees that an error is signalled when the current thread accidentally reenters the critical section via recursive function calls.

Guile provides two mechanisms to support critical sections as outlined above. You can either use the macros `SCM_CRITICAL_SECTION_START` and `SCM_CRITICAL_SECTION_END` for very simple sections; or use a dynwind context together with a call to `scm_dynwind_critical_section`.

The macros only work reliably for critical sections that are guaranteed to not cause a non-local exit. They also do not detect an accidental reentry by the current thread. Thus, you should probably only use them to delimit critical sections that do not contain calls to libguile functions or to other external functions that might do complicated things.

The function `scm_dynwind_critical_section`, on the other hand, will correctly deal with non-local exits because it requires a dynwind context. Also, by using a separate mutex for each critical section, it can detect accidental reentries.

5.5 Defining New Foreign Object Types

The *foreign object type* facility is Guile's mechanism for importing object and types from C or other languages into Guile's system. If you have a C `struct foo` type, for example, you can define a corresponding Guile foreign object type that allows Scheme code to handle `struct foo *` objects.

To define a new foreign object type, the programmer provides Guile with some essential information about the type — what its name is, how many fields it has, and its finalizer (if any) — and Guile allocates a fresh type for it. Foreign objects can be accessed from Scheme or from C.

5.5.1 Defining Foreign Object Types

To create a new foreign object type from C, call `scm_make_foreign_object_type`. It returns a value of type `SCM` which identifies the new type.

Here is how one might declare a new type representing eight-bit gray-scale images:

```
#include <libguile.h>

struct image {
  int width, height;
  char *pixels;

  /* The name of this image */
  SCM name;

  /* A function to call when this image is
     modified, e.g., to update the screen,
     or SCM_BOOL_F if no action necessary */
  SCM update_func;
};

static SCM image_type;

void
init_image_type (void)
{
  SCM name, slots;
  scm_t_struct_finalize finalizer;

  name = scm_from_utf8_symbol ("image");
  slots = scm_list_1 (scm_from_utf8_symbol ("data"));
  finalizer = NULL;

  image_type =
    scm_make_foreign_object_type (name, slots, finalizer);
}
```

The result is an initialized `image_type` value that identifies the new foreign object type. The next section describes how to create foreign objects and how to access their slots.

5.5.2 Creating Foreign Objects

Foreign objects contain zero or more "slots" of data. A slot can hold a pointer, an integer that fits into a `size_t` or `ssize_t`, or a `SCM` value.

All objects of a given foreign type have the same number of slots. In the example from the previous section, the `image` type has one slot, because the slots list passed to `scm_make_foreign_object_type` is of length one. (The actual names given to slots are unimportant for most users of the C interface, but can be used on the Scheme side to introspect on the foreign object.)

To construct a foreign object and initialize its first slot, call `scm_make_foreign_object_1 (`*`type`*`, `*`first_slot_value`*`)`. There are similarly named constructors for initializing 0, 1, 2, or 3 slots, or initializing *n* slots via an array. See Section 6.7 [Foreign Objects], page 243, for full details. Any fields that are not explicitly initialized are set to 0.

To get or set the value of a slot by index, you can use the `scm_foreign_object_ref` and `scm_foreign_object_set_x` functions. These functions take and return values as `void *` pointers; there are corresponding convenience procedures like `_signed_ref`, `_unsigned_set_x` and so on for dealing with slots as signed or unsigned integers.

Foreign objects fields that are pointers can be tricky to manage. If possible, it is best that all memory that is referenced by a foreign object be managed by the garbage collector. That way, the GC can automatically ensure that memory is accessible when it is needed, and freed when it becomes inaccessible. If this is not the case for your program – for example, if you are exposing an object to Scheme that was allocated by some other, Guile-unaware part of your program – then you will probably need to implement a finalizer. See Section 5.5.4 [Foreign Object Memory Management], page 77, for more.

Continuing the example from the previous section, if the global variable `image_type` contains the type returned by `scm_make_foreign_object_type`, here is how we could construct a foreign object whose "data" field contains a pointer to a freshly allocated `struct image`:

```
SCM
make_image (SCM name, SCM s_width, SCM s_height)
{
  struct image *image;
  int width = scm_to_int (s_width);
  int height = scm_to_int (s_height);

  /* Allocate the 'struct image'.  Because we
     use scm_gc_malloc, this memory block will
     be automatically reclaimed when it becomes
     inaccessible, and its members will be traced
     by the garbage collector.  */
  image = (struct image *)
    scm_gc_malloc (sizeof (struct image), "image");
```

```
  image->width = width;
  image->height = height;

  /* Allocating the pixels with
     scm_gc_malloc_pointerless means that the
     pixels data is collectable by GC, but
     that GC shouldn't spend time tracing its
     contents for nested pointers because there
     aren't any.  */
  image->pixels =
    scm_gc_malloc_pointerless (width * height, "image pixels");

  image->name = name;
  image->update_func = SCM_BOOL_F;

  /* Now wrap the struct image* in a new foreign
     object, and return that object.  */
  return scm_make_foreign_object_1 (image_type, image);
}
```

We use `scm_gc_malloc_pointerless` for the pixel buffer to tell the garbage collector not to scan it for pointers. Calls to `scm_gc_malloc`, `scm_make_foreign_object_1`, and `scm_gc_malloc_pointerless` raise an exception in out-of-memory conditions; the garbage collector is able to reclaim previously allocated memory if that happens.

5.5.3 Type Checking of Foreign Objects

Functions that operate on foreign objects should check that the passed `SCM` value indeed is of the correct type before accessing its data. They can do this with `scm_assert_foreign_object_type`.

For example, here is a simple function that operates on an image object, and checks the type of its argument.

```
SCM
clear_image (SCM image_obj)
{
  int area;
  struct image *image;

  scm_assert_foreign_object_type (image_type, image_obj);

  image = scm_foreign_object_ref (image_obj, 0);
  area = image->width * image->height;
  memset (image->pixels, 0, area);

  /* Invoke the image's update function.  */
  if (scm_is_true (image->update_func))
    scm_call_0 (image->update_func);
```

```
  return SCM_UNSPECIFIED;
}
```

5.5.4 Foreign Object Memory Management

Once a foreign object has been released to the tender mercies of the Scheme system, it must be prepared to survive garbage collection. In the example above, all the memory associated with the foreign object is managed by the garbage collector because we used the `scm_gc_` allocation functions. Thus, no special care must be taken: the garbage collector automatically scans them and reclaims any unused memory.

However, when data associated with a foreign object is managed in some other way—e.g., `malloc`'d memory or file descriptors—it is possible to specify a *finalizer* function to release those resources when the foreign object is reclaimed.

As discussed in see Section 5.4.2 [Garbage Collection], page 67, Guile's garbage collector will reclaim inaccessible memory as needed. This reclamation process runs concurrently with the main program. When Guile analyzes the heap and determines that an object's memory can be reclaimed, that memory is put on a "free list" of objects that can be reclaimed. Usually that's the end of it—the object is available for immediate re-use. However some objects can have "finalizers" associated with them—functions that are called on reclaimable objects to effect any external cleanup actions.

Finalizers are tricky business and it is best to avoid them. They can be invoked at unexpected times, or not at all—for example, they are not invoked on process exit. They don't help the garbage collector do its job; in fact, they are a hindrance. Furthermore, they perturb the garbage collector's internal accounting. The GC decides to scan the heap when it thinks that it is necessary, after some amount of allocation. Finalizable objects almost always represent an amount of allocation that is invisible to the garbage collector. The effect can be that the actual resource usage of a system with finalizable objects is higher than what the GC thinks it should be.

All those caveats aside, some foreign object types will need finalizers. For example, if we had a foreign object type that wrapped file descriptors—and we aren't suggesting this, as Guile already has ports —then you might define the type like this:

```
static SCM file_type;

static void
finalize_file (SCM file)
{
  int fd = scm_foreign_object_signed_ref (file, 0);
  if (fd >= 0)
    {
      scm_foreign_object_signed_set_x (file, 0, -1);
      close (fd);
    }
}

static void
init_file_type (void)
{
```

```
  SCM name, slots;
  scm_t_struct_finalize finalizer;

  name = scm_from_utf8_symbol ("file");
  slots = scm_list_1 (scm_from_utf8_symbol ("fd"));
  finalizer = finalize_file;

  image_type =
    scm_make_foreign_object_type (name, slots, finalizer);
}

static SCM
make_file (int fd)
{
  return scm_make_foreign_object_1 (file_type, (void *) fd);
}
```

Note that the finalizer may be invoked in ways and at times you might not expect. In a Guile built without threading support, finalizers are invoked via "asyncs", which interleaves them with running Scheme code; see Section 6.22.3 [Asyncs], page 445. If the user's Guile is built with support for threads, the finalizer will probably be called by a dedicated finalization thread, unless the user invokes `scm_run_finalizers ()` explicitly.

In either case, finalizers run concurrently with the main program, and so they need to be async-safe and thread-safe. If for some reason this is impossible, perhaps because you are embedding Guile in some application that is not itself thread-safe, you have a few options. One is to use guardians instead of finalizers, and arrange to pump the guardians for finalizable objects. See Section 6.19.4 [Guardians], page 409, for more information. The other option is to disable automatic finalization entirely, and arrange to call `scm_run_finalizers ()` at appropriate points. See Section 6.7 [Foreign Objects], page 243, for more on these interfaces.

Finalizers are allowed to allocate memory, access GC-managed memory, and in general can do anything any Guile user code can do. This was not the case in Guile 1.8, where finalizers were much more restricted. In particular, in Guile 2.0, finalizers can resuscitate objects. We do not recommend that users avail themselves of this possibility, however, as a resuscitated object can re-expose other finalizable objects that have been already finalized back to Scheme. These objects will not be finalized again, but they could cause use-after-free problems to code that handles objects of that particular foreign object type. To guard against this possibility, robust finalization routines should clear state from the foreign object, as in the above `free_file` example.

One final caveat. Foreign object finalizers are associated with the lifetime of a foreign object, not of its fields. If you access a field of a finalizable foreign object, and do not arrange to keep a reference on the foreign object itself, it could be that the outer foreign object gets finalized while you are working with its field.

For example, consider a procedure to read some data from a file, from our example above.

```
SCM
```

```
read_bytes (SCM file, SCM n)
{
  int fd;
  SCM buf;
  size_t len, pos;

  scm_assert_foreign_object_type (file_type, file);

  fd = scm_foreign_object_signed_ref (file, 0);
  if (fd < 0)
    scm_wrong_type_arg_msg ("read-bytes", SCM_ARG1,
                            file, "open file");

  len = scm_to_size_t (n);
  SCM buf = scm_c_make_bytevector (scm_to_size_t (n));

  pos = 0;
  while (pos < len)
    {
      char *bytes = SCM_BYTEVECTOR_CONTENTS (buf);
      ssize_t count = read (fd, bytes + pos, len - pos);
      if (count < 0)
        scm_syserror ("read-bytes");
      if (count == 0)
        break;
      pos += count;
    }

  scm_remember_upto_here_1 (file);

  return scm_values (scm_list_2 (buf, scm_from_size_t (pos)));
}
```

After the prelude, only the `fd` value is used and the C compiler has no reason to keep the `file` object around. If `scm_c_make_bytevector` results in a garbage collection, `file` might not be on the stack or anywhere else and could be finalized, leaving `read` to read a closed (or, in a multi-threaded program, possibly re-used) file descriptor. The use of `scm_remember_upto_here_1` prevents this, by creating a reference to `file` after all data accesses. See Section 6.19.1 [Garbage Collection Functions], page 404.

`scm_remember_upto_here_1` is only needed on finalizable objects, because garbage collection of other values is invisible to the program – it happens when needed, and is not observable. But if you can, save yourself the headache and build your program in such a way that it doesn't need finalization.

5.5.5 Foreign Objects and Scheme

It is also possible to create foreign objects and object types from Scheme, and to access fields of foreign objects from Scheme. For example, the file example from the last section could be equivalently expressed as:

```
(define-module (my-file)
  #:use-module (system foreign-object)
  #:use-module ((oop goops) #:select (make))
  #:export (make-file))

(define (finalize-file file)
  (let ((fd (struct-ref file 0)))
    (unless (< fd 0)
      (struct-set! file 0 -1)
      (close-fdes fd))))

(define <file>
  (make-foreign-object-type '<file> '(fd)
                            #:finalizer finalize-file))

(define (make-file fd)
  (make <file> #:fd fd))
```

Here we see that the result of `make-foreign-object-type`, which is the equivalent of `scm_make_foreign_object_type`, is a struct vtable. See Section 6.6.18.1 [Vtables], page 223, for more information. To instantiate the foreign object, which is really a Guile struct, we use `make`. (We could have used `make-struct/no-tail`, but as an implementation detail, finalizers are attached in the `initialize` method called by `make`). To access the fields, we use `struct-ref` and `struct-set!`. See Section 6.6.18.2 [Structure Basics], page 224.

There is a convenience syntax, `define-foreign-object-type`, that defines a type along with a constructor, and getters for the fields. An appropriate invocation of `define-foreign-object-type` for the file object type could look like this:

```
(use-modules (system foreign-object))

(define-foreign-object-type <file>
  make-file
  (fd)
  #:finalizer finalize-file)
```

This defines the `<file>` type with one field, a `make-file` constructor, and a getter for the `fd` field, bound to `fd`.

Foreign object types are not only vtables but are actually GOOPS classes, as hinted at above. See Chapter 8 [GOOPS], page 773, for more on Guile's object-oriented programming system. Thus one can define print and equality methods using GOOPS:

```
(use-modules (oop goops))

(define-method (write (file <file>) port)
```

```
  ;; Assuming existence of the 'fd' getter
  (format port "#<<file> ~a>" (fd file)))

(define-method (equal? (a <file>) (b <file>))
  (eqv? (fd a) (fd b)))
```

One can even sub-class foreign types.

```
(define-class <named-file> (<file>)
  (name #:init-keyword #:name #:init-value #f #:accessor name))
```

The question arises of how to construct these values, given that `make-file` returns a plain old `<file>` object. It turns out that you can use the GOOPS construction interface, where every field of the foreign object has an associated initialization keyword argument.

```
(define* (my-open-file name #:optional (flags O_RDONLY))
  (make <named-file> #:fd (open-fdes name flags) #:name name))

(define-method (write (file <named-file>) port)
  (format port "#<<file> ~s ~a>" (name file) (fd file)))
```

See Section 6.7 [Foreign Objects], page 243, for full documentation on the Scheme interface to foreign objects. See Chapter 8 [GOOPS], page 773, for more on GOOPS.

As a final note, you might wonder how this system supports encapsulation of sensitive values. First, we have to recognize that some facilities are essentially unsafe and have global scope. For example, in C, the integrity and confidentiality of a part of a program is at the mercy of every other part of that program – because any part of the program can read and write anything in its address space. At the same time, principled access to structured data is organized in C on lexical boundaries; if you don't expose accessors for your object, you trust other parts of the program not to work around that barrier.

The situation is not dissimilar in Scheme. Although Scheme's unsafe constructs are fewer in number than in C, they do exist. The `(system foreign)` module can be used to violate confidentiality and integrity, and shouldn't be exposed to untrusted code. Although `struct-ref` and `struct-set!` are less unsafe, they still have a cross-cutting capability of drilling through abstractions. Performing a `struct-set!` on a foreign object slot could cause unsafe foreign code to crash. Ultimately, structures in Scheme are capabilities for abstraction, and not abstractions themselves.

That leaves us with the lexical capabilities, like constructors and accessors. Here is where encapsulation lies: the practical degree to which the innards of your foreign objects are exposed is the degree to which their accessors are lexically available in user code. If you want to allow users to reference fields of your foreign object, provide them with a getter. Otherwise you should assume that the only access to your object may come from your code, which has the relevant authority, or via code with access to cross-cutting `struct-ref` and such, which also has the cross-cutting authority.

5.6 Function Snarfing

When writing C code for use with Guile, you typically define a set of C functions, and then make some of them visible to the Scheme world by calling `scm_c_define_gsubr` or related functions. If you have many functions to publish, it can sometimes be annoying to keep the list of calls to `scm_c_define_gsubr` in sync with the list of function definitions.

Guile provides the `guile-snarf` program to manage this problem. Using this tool, you can keep all the information needed to define the function alongside the function definition itself; `guile-snarf` will extract this information from your source code, and automatically generate a file of calls to `scm_c_define_gsubr` which you can `#include` into an initialization function.

The snarfing mechanism works for many kind of initialization actions, not just for collecting calls to `scm_c_define_gsubr`. For a full list of what can be done, See Section 6.5 [Snarfing Macros], page 102.

The `guile-snarf` program is invoked like this:

```
guile-snarf [-o outfile] [cpp-args ...]
```

This command will extract initialization actions to *outfile*. When no *outfile* has been specified or when *outfile* is `-`, standard output will be used. The C preprocessor is called with *cpp-args* (which usually include an input file) and the output is filtered to extract the initialization actions.

If there are errors during processing, *outfile* is deleted and the program exits with non-zero status.

During snarfing, the pre-processor macro `SCM_MAGIC_SNARFER` is defined. You could use this to avoid including snarfer output files that don't yet exist by writing code like this:

```
#ifndef SCM_MAGIC_SNARFER
#include "foo.x"
#endif
```

Here is how you might define the Scheme function `clear-image`, implemented by the C function `clear_image`:

```
#include <libguile.h>

SCM_DEFINE (clear_image, "clear-image", 1, 0, 0,
            (SCM image),
            "Clear the image.")
{
  /* C code to clear the image in image... */
}

void
init_image_type ()
{
#include "image-type.x"
}
```

The `SCM_DEFINE` declaration says that the C function `clear_image` implements a Scheme function called `clear-image`, which takes one required argument (of type `SCM` and named `image`), no optional arguments, and no rest argument. The string `"Clear the image."` provides a short help text for the function, it is called a *docstring*.

`SCM_DEFINE` macro also defines a static array of characters initialized to the Scheme name of the function. In this case, `s_clear_image` is set to the C string, `"clear-image"`. You might want to use this symbol when generating error messages.

Assuming the text above lives in a file named `image-type.c`, you will need to execute the following command to prepare this file for compilation:

```
guile-snarf -o image-type.x image-type.c
```

This scans `image-type.c` for `SCM_DEFINE` declarations, and writes to `image-type.x` the output:

```
scm_c_define_gsubr ("clear-image", 1, 0, 0, (SCM (*)() ) clear_image);
```

When compiled normally, `SCM_DEFINE` is a macro which expands to the function header for `clear_image`.

Note that the output file name matches the `#include` from the input file. Also, you still need to provide all the same information you would if you were using `scm_c_define_gsubr` yourself, but you can place the information near the function definition itself, so it is less likely to become incorrect or out-of-date.

If you have many files that `guile-snarf` must process, you should consider using a fragment like the following in your Makefile:

```
snarfcppopts = $(DEFS) $(INCLUDES) $(CPPFLAGS) $(CFLAGS)
.SUFFIXES: .x
.c.x:
guile-snarf -o $@ $< $(snarfcppopts)
```

This tells make to run `guile-snarf` to produce each needed `.x` file from the corresponding `.c` file.

The program `guile-snarf` passes its command-line arguments directly to the C preprocessor, which it uses to extract the information it needs from the source code. this means you can pass normal compilation flags to `guile-snarf` to define preprocessor symbols, add header file directories, and so on.

5.7 An Overview of Guile Programming

Guile is designed as an extension language interpreter that is straightforward to integrate with applications written in C (and C++). The big win here for the application developer is that Guile integration, as the Guile web page says, "lowers your project's hacktivation energy." Lowering the hacktivation energy means that you, as the application developer, *and your users*, reap the benefits that flow from being able to extend the application in a high level extension language rather than in plain old C.

In abstract terms, it's difficult to explain what this really means and what the integration process involves, so instead let's begin by jumping straight into an example of how you might integrate Guile into an existing program, and what you could expect to gain by so doing. With that example under our belts, we'll then return to a more general analysis of the arguments involved and the range of programming options available.

5.7.1 How One Might Extend Dia Using Guile

Dia is a free software program for drawing schematic diagrams like flow charts and floor plans (`http://www.gnome.org/projects/dia/`). This section conducts the thought experiment of adding Guile to Dia. In so doing, it aims to illustrate several of the steps and considerations involved in adding Guile to applications in general.

5.7.1.1 Deciding Why You Want to Add Guile

First off, you should understand why you want to add Guile to Dia at all, and that means forming a picture of what Dia does and how it does it. So, what are the constituents of the Dia application?

- Most importantly, the *application domain objects* — in other words, the concepts that differentiate Dia from another application such as a word processor or spreadsheet: shapes, templates, connectors, pages, plus the properties of all these things.
- The code that manages the graphical face of the application, including the layout and display of the objects above.
- The code that handles input events, which indicate that the application user is wanting to do something.

(In other words, a textbook example of the *model - view - controller* paradigm.)

Next question: how will Dia benefit once the Guile integration is complete? Several (positive!) answers are possible here, and the choice is obviously up to the application developers. Still, one answer is that the main benefit will be the ability to manipulate Dia's application domain objects from Scheme.

Suppose that Dia made a set of procedures available in Scheme, representing the most basic operations on objects such as shapes, connectors, and so on. Using Scheme, the application user could then write code that builds upon these basic operations to create more complex procedures. For example, given basic procedures to enumerate the objects on a page, to determine whether an object is a square, and to change the fill pattern of a single shape, the user can write a Scheme procedure to change the fill pattern of all squares on the current page:

```
(define (change-squares'-fill-pattern new-pattern)
  (for-each-shape current-page
    (lambda (shape)
      (if (square? shape)
          (change-fill-pattern shape new-pattern)))))
```

5.7.1.2 Four Steps Required to Add Guile

Assuming this objective, four steps are needed to achieve it.

First, you need a way of representing your application-specific objects — such as `shape` in the previous example — when they are passed into the Scheme world. Unless your objects are so simple that they map naturally into builtin Scheme data types like numbers and strings, you will probably want to use Guile's *foreign object* interface to create a new Scheme data type for your objects.

Second, you need to write code for the basic operations like `for-each-shape` and `square?` such that they access and manipulate your existing data structures correctly, and then make these operations available as *primitives* on the Scheme level.

Third, you need to provide some mechanism within the Dia application that a user can hook into to cause arbitrary Scheme code to be evaluated.

Finally, you need to restructure your top-level application C code a little so that it initializes the Guile interpreter correctly and declares your *foreign objects* and *primitives* to the Scheme world.

The following subsections expand on these four points in turn.

5.7.1.3 How to Represent Dia Data in Scheme

For all but the most trivial applications, you will probably want to allow some representation of your domain objects to exist on the Scheme level. This is where foreign objects come in, and with them issues of lifetime management and garbage collection.

To get more concrete about this, let's look again at the example we gave earlier of how application users can use Guile to build higher-level functions from the primitives that Dia itself provides.

```
(define (change-squares'-fill-pattern new-pattern)
  (for-each-shape current-page
    (lambda (shape)
      (if (square? shape)
          (change-fill-pattern shape new-pattern)))))
```

Consider what is stored here in the variable `shape`. For each shape on the current page, the `for-each-shape` primitive calls `(lambda (shape) ...)` with an argument representing that shape. Question is: how is that argument represented on the Scheme level? The issues are as follows.

- Whatever the representation, it has to be decodable again by the C code for the `square?` and `change-fill-pattern` primitives. In other words, a primitive like `square?` has somehow to be able to turn the value that it receives back into something that points to the underlying C structure describing a shape.
- The representation must also cope with Scheme code holding on to the value for later use. What happens if the Scheme code stores `shape` in a global variable, but then that shape is deleted (in a way that the Scheme code is not aware of), and later on some other Scheme code uses that global variable again in a call to, say, `square?`?
- The lifetime and memory allocation of objects that exist *only* in the Scheme world is managed automatically by Guile's garbage collector using one simple rule: when there are no remaining references to an object, the object is considered dead and so its memory is freed. But for objects that exist in both C and Scheme, the picture is more complicated; in the case of Dia, where the `shape` argument passes transiently in and out of the Scheme world, it would be quite wrong the **delete** the underlying C shape just because the Scheme code has finished evaluation. How do we avoid this happening?

One resolution of these issues is for the Scheme-level representation of a shape to be a new, Scheme-specific C structure wrapped up as a foreign object. The foreign object is what is passed into and out of Scheme code, and the Scheme-specific C structure inside the foreign object points to Dia's underlying C structure so that the code for primitives like `square?` can get at it.

To cope with an underlying shape being deleted while Scheme code is still holding onto a Scheme shape value, the underlying C structure should have a new field that points to the Scheme-specific foreign object. When a shape is deleted, the relevant code chains through to the Scheme-specific structure and sets its pointer back to the underlying structure to NULL. Thus the foreign object value for the shape continues to exist, but any primitive

code that tries to use it will detect that the underlying shape has been deleted because the underlying structure pointer is NULL.

So, to summarize the steps involved in this resolution of the problem (and assuming that the underlying C structure for a shape is `struct dia_shape`):

- Define a new Scheme-specific structure that *points* to the underlying C structure:

```
struct dia_guile_shape
{
  struct dia_shape * c_shape;   /* NULL => deleted */
}
```

- Add a field to `struct dia_shape` that points to its `struct dia_guile_shape` if it has one —

```
struct dia_shape
{
  ...
  struct dia_guile_shape * guile_shape;
}
```

  — so that C code can set `guile_shape->c_shape` to NULL when the underlying shape is deleted.

- Wrap `struct dia_guile_shape` as a foreign object type.
- Whenever you need to represent a C shape onto the Scheme level, create a foreign object instance for it, and pass that.
- In primitive code that receives a shape foreign object instance, check the `c_shape` field when decoding it, to find out whether the underlying C shape is still there.

As far as memory management is concerned, the foreign object values and their Scheme-specific structures are under the control of the garbage collector, whereas the underlying C structures are explicitly managed in exactly the same way that Dia managed them before we thought of adding Guile.

When the garbage collector decides to free a shape foreign object value, it calls the *finalizer* function that was specified when defining the shape foreign object type. To maintain the correctness of the `guile_shape` field in the underlying C structure, this function should chain through to the underlying C structure (if it still exists) and set its `guile_shape` field to NULL.

For full documentation on defining and using foreign object types, see Section 5.5 [Defining New Foreign Object Types], page 74.

5.7.1.4 Writing Guile Primitives for Dia

Once the details of object representation are decided, writing the primitive function code that you need is usually straightforward.

A primitive is simply a C function whose arguments and return value are all of type `SCM`, and whose body does whatever you want it to do. As an example, here is a possible implementation of the `square?` primitive:

```
static SCM square_p (SCM shape)
{
```

```
  struct dia_guile_shape * guile_shape;

  /* Check that arg is really a shape object. */
  scm_assert_foreign_object_type (shape_type, shape);

  /* Access Scheme-specific shape structure. */
  guile_shape = scm_foreign_object_ref (shape, 0);

  /* Find out if underlying shape exists and is a
     square; return answer as a Scheme boolean. */
  return scm_from_bool (guile_shape->c_shape &&
                        (guile_shape->c_shape->type == DIA_SQUARE));
}
```

Notice how easy it is to chain through from the `SCM` `shape` parameter that `square_p` receives — which is a foreign object — to the Scheme-specific structure inside the foreign object, and thence to the underlying C structure for the shape.

In this code, `scm_assert_foreign_object_type`, `scm_foreign_object_ref`, and `scm_from_bool` are from the standard Guile API. We assume that `shape_type` was given to us when we made the shape foreign object type, using `scm_make_foreign_object_type`. The call to `scm_assert_foreign_object_type` ensures that *shape* is indeed a shape. This is needed to guard against Scheme code using the `square?` procedure incorrectly, as in `(square? "hello")`; Scheme's latent typing means that usage errors like this must be caught at run time.

Having written the C code for your primitives, you need to make them available as Scheme procedures by calling the `scm_c_define_gsubr` function. `scm_c_define_gsubr` (see Section 6.9.2 [Primitive Procedures], page 249) takes arguments that specify the Scheme-level name for the primitive and how many required, optional and rest arguments it can accept. The `square?` primitive always requires exactly one argument, so the call to make it available in Scheme reads like this:

```
scm_c_define_gsubr ("square?", 1, 0, 0, square_p);
```

For where to put this call, see the subsection after next on the structure of Guile-enabled code (see Section 5.7.1.6 [Dia Structure], page 88).

5.7.1.5 Providing a Hook for the Evaluation of Scheme Code

To make the Guile integration useful, you have to design some kind of hook into your application that application users can use to cause their Scheme code to be evaluated.

Technically, this is straightforward; you just have to decide on a mechanism that is appropriate for your application. Think of Emacs, for example: when you type *`ESC :`*, you get a prompt where you can type in any Elisp code, which Emacs will then evaluate. Or, again like Emacs, you could provide a mechanism (such as an init file) to allow Scheme code to be associated with a particular key sequence, and evaluate the code when that key sequence is entered.

In either case, once you have the Scheme code that you want to evaluate, as a null terminated string, you can tell Guile to evaluate it by calling the `scm_c_eval_string` function.

5.7.1.6 Top-level Structure of Guile-enabled Dia

Let's assume that the pre-Guile Dia code looks structurally like this:

- `main ()`
  - do lots of initialization and setup stuff
  - enter Gtk main loop

When you add Guile to a program, one (rather technical) requirement is that Guile's garbage collector needs to know where the bottom of the C stack is. The easiest way to ensure this is to use `scm_boot_guile` like this:

- `main ()`
  - do lots of initialization and setup stuff
  - `scm_boot_guile (argc, argv, inner_main, NULL)`
- `inner_main ()`
  - define all foreign object types
  - export primitives to Scheme using `scm_c_define_gsubr`
  - enter Gtk main loop

In other words, you move the guts of what was previously in your `main` function into a new function called `inner_main`, and then add a `scm_boot_guile` call, with `inner_main` as a parameter, to the end of `main`.

Assuming that you are using foreign objects and have written primitive code as described in the preceding subsections, you also need to insert calls to declare your new foreign objects and export the primitives to Scheme. These declarations must happen *inside* the dynamic scope of the `scm_boot_guile` call, but also *before* any code is run that could possibly use them — the beginning of `inner_main` is an ideal place for this.

5.7.1.7 Going Further with Dia and Guile

The steps described so far implement an initial Guile integration that already gives a lot of additional power to Dia application users. But there are further steps that you could take, and it's interesting to consider a few of these.

In general, you could progressively move more of Dia's source code from C into Scheme. This might make the code more maintainable and extensible, and it could open the door to new programming paradigms that are tricky to effect in C but straightforward in Scheme.

A specific example of this is that you could use the guile-gtk package, which provides Scheme-level procedures for most of the Gtk+ library, to move the code that lays out and displays Dia objects from C to Scheme.

As you follow this path, it naturally becomes less useful to maintain a distinction between Dia's original non-Guile-related source code, and its later code implementing foreign objects and primitives for the Scheme world.

For example, suppose that the original source code had a `dia_change_fill_pattern` function:

```
void dia_change_fill_pattern (struct dia_shape * shape,
                              struct dia_pattern * pattern)
{
```

```
  /* real pattern change work */
}
```

During initial Guile integration, you add a `change_fill_pattern` primitive for Scheme purposes, which accesses the underlying structures from its foreign object values and uses `dia_change_fill_pattern` to do the real work:

```
SCM change_fill_pattern (SCM shape, SCM pattern)
{
  struct dia_shape * d_shape;
  struct dia_pattern * d_pattern;

  ...

  dia_change_fill_pattern (d_shape, d_pattern);

  return SCM_UNSPECIFIED;
}
```

At this point, it makes sense to keep `dia_change_fill_pattern` and `change_fill_pattern` separate, because `dia_change_fill_pattern` can also be called without going through Scheme at all, say because the user clicks a button which causes a C-registered Gtk+ callback to be called.

But, if the code for creating buttons and registering their callbacks is moved into Scheme (using guile-gtk), it may become true that `dia_change_fill_pattern` can no longer be called other than through Scheme. In which case, it makes sense to abolish it and move its contents directly into `change_fill_pattern`, like this:

```
SCM change_fill_pattern (SCM shape, SCM pattern)
{
  struct dia_shape * d_shape;
  struct dia_pattern * d_pattern;

  ...

  /* real pattern change work */

  return SCM_UNSPECIFIED;
}
```

So further Guile integration progressively *reduces* the amount of functional C code that you have to maintain over the long term.

A similar argument applies to data representation. In the discussion of foreign objects earlier, issues arose because of the different memory management and lifetime models that normally apply to data structures in C and in Scheme. However, with further Guile integration, you can resolve this issue in a more radical way by allowing all your data structures to be under the control of the garbage collector, and kept alive by references from the Scheme world. Instead of maintaining an array or linked list of shapes in C, you would instead maintain a list in Scheme.

Rather like the coalescing of `dia_change_fill_pattern` and `change_fill_pattern`, the practical upshot of such a change is that you would no longer have to keep the `dia_shape` and `dia_guile_shape` structures separate, and so wouldn't need to worry about the pointers between them. Instead, you could change the foreign object definition to wrap the `dia_shape` structure directly, and send `dia_guile_shape` off to the scrap yard. Cut out the middle man!

Finally, we come to the holy grail of Guile's free software / extension language approach. Once you have a Scheme representation for interesting Dia data types like shapes, and a handy bunch of primitives for manipulating them, it suddenly becomes clear that you have a bundle of functionality that could have far-ranging use beyond Dia itself. In other words, the data types and primitives could now become a library, and Dia becomes just one of the many possible applications using that library — albeit, at this early stage, a rather important one!

In this model, Guile becomes just the glue that binds everything together. Imagine an application that usefully combined functionality from Dia, Gnumeric and GnuCash — it's tricky right now, because no such application yet exists; but it'll happen some day ...

5.7.2 Why Scheme is More Hackable Than C

Underlying Guile's value proposition is the assumption that programming in a high level language, specifically Guile's implementation of Scheme, is necessarily better in some way than programming in C. What do we mean by this claim, and how can we be so sure?

One class of advantages applies not only to Scheme, but more generally to any interpretable, high level, scripting language, such as Emacs Lisp, Python, Ruby, or TEX's macro language. Common features of all such languages, when compared to C, are that:

- They lend themselves to rapid and experimental development cycles, owing usually to a combination of their interpretability and the integrated development environment in which they are used.
- They free developers from some of the low level bookkeeping tasks associated with C programming, notably memory management.
- They provide high level features such as container objects and exception handling that make common programming tasks easier.

In the case of Scheme, particular features that make programming easier — and more fun! — are its powerful mechanisms for abstracting parts of programs (closures — see Section 3.4 [About Closure], page 26) and for iteration (see Section 6.13.4 [while do], page 301).

The evidence in support of this argument is empirical: the huge amount of code that has been written in extension languages for applications that support this mechanism. Most notable are extensions written in Emacs Lisp for GNU Emacs, in TEX's macro language for TEX, and in Script-Fu for the Gimp, but there is increasingly now a significant code eco-system for Guile-based applications as well, such as Lilypond and GnuCash. It is close to inconceivable that similar amounts of functionality could have been added to these applications just by writing new code in their base implementation languages.

5.7.3 Example: Using Guile for an Application Testbed

As an example of what this means in practice, imagine writing a testbed for an application that is tested by submitting various requests (via a C interface) and validating the

output received. Suppose further that the application keeps an idea of its current state, and that the "correct" output for a given request may depend on the current application state. A complete "white box"[4] test plan for this application would aim to submit all possible requests in each distinguishable state, and validate the output for all request/state combinations.

To write all this test code in C would be very tedious. Suppose instead that the testbed code adds a single new C function, to submit an arbitrary request and return the response, and then uses Guile to export this function as a Scheme procedure. The rest of the testbed can then be written in Scheme, and so benefits from all the advantages of programming in Scheme that were described in the previous section.

(In this particular example, there is an additional benefit of writing most of the testbed in Scheme. A common problem for white box testing is that mistakes and mistaken assumptions in the application under test can easily be reproduced in the testbed code. It is more difficult to copy mistakes like this when the testbed is written in a different language from the application.)

5.7.4 A Choice of Programming Options

The preceding arguments and example point to a model of Guile programming that is applicable in many cases. According to this model, Guile programming involves a balance between C and Scheme programming, with the aim being to extract the greatest possible Scheme level benefit from the least amount of C level work.

The C level work required in this model usually consists of packaging and exporting functions and application objects such that they can be seen and manipulated on the Scheme level. To help with this, Guile's C language interface includes utility features that aim to make this kind of integration very easy for the application developer. These features are documented later in this part of the manual: see REFFIXME.

This model, though, is really just one of a range of possible programming options. If all of the functionality that you need is available from Scheme, you could choose instead to write your whole application in Scheme (or one of the other high level languages that Guile supports through translation), and simply use Guile as an interpreter for Scheme. (In the future, we hope that Guile will also be able to compile Scheme code, so lessening the performance gap between C and Scheme code.) Or, at the other end of the C–Scheme scale, you could write the majority of your application in C, and only call out to Guile occasionally for specific actions such as reading a configuration file or executing a user-specified extension. The choices boil down to two basic questions:

- Which parts of the application do you write in C, and which in Scheme (or another high level translated language)?
- How do you design the interface between the C and Scheme parts of your application?

These are of course design questions, and the right design for any given application will always depend upon the particular requirements that you are trying to meet. In the context of Guile, however, there are some generally applicable considerations that can help you when designing your answers.

[4] A *white box* test plan is one that incorporates knowledge of the internal design of the application under test.

5.7.4.1 What Functionality is Already Available?

Suppose, for the sake of argument, that you would prefer to write your whole application in Scheme. Then the API available to you consists of:

- standard Scheme
- plus the extensions to standard Scheme provided by Guile in its core distribution
- plus any additional functionality that you or others have packaged so that it can be loaded as a Guile Scheme module.

A module in the last category can either be a pure Scheme module — in other words a collection of utility procedures coded in Scheme — or a module that provides a Scheme interface to an extension library coded in C — in other words a nice package where someone else has done the work of wrapping up some useful C code for you. The set of available modules is growing quickly and already includes such useful examples as `(gtk gtk)`, which makes Gtk+ drawing functions available in Scheme, and `(database postgres)`, which provides SQL access to a Postgres database.

Given the growing collection of pre-existing modules, it is quite feasible that your application could be implemented by combining a selection of these modules together with new application code written in Scheme.

If this approach is not enough, because the functionality that your application needs is not already available in this form, and it is impossible to write the new functionality in Scheme, you will need to write some C code. If the required function is already available in C (e.g. in a library), all you need is a little glue to connect it to the world of Guile. If not, you need both to write the basic code and to plumb it into Guile.

In either case, two general considerations are important. Firstly, what is the interface by which the functionality is presented to the Scheme world? Does the interface consist only of function calls (for example, a simple drawing interface), or does it need to include *objects* of some kind that can be passed between C and Scheme and manipulated by both worlds. Secondly, how does the lifetime and memory management of objects in the C code relate to the garbage collection governed approach of Scheme objects? In the case where the basic C code is not already written, most of the difficulties of memory management can be avoided by using Guile's C interface features from the start.

For the full documentation on writing C code for Guile and connecting existing C code to the Guile world, see REFFIXME.

5.7.4.2 Functional and Performance Constraints

5.7.4.3 Your Preferred Programming Style

5.7.4.4 What Controls Program Execution?

5.7.5 How About Application Users?

So far we have considered what Guile programming means for an application developer. But what if you are instead *using* an existing Guile-based application, and want to know what your options are for programming and extending this application?

The answer to this question varies from one application to another, because the options available depend inevitably on whether the application developer has provided any hooks

for you to hang your own code on and, if there are such hooks, what they allow you to do.[5] For example. . .

- If the application permits you to load and execute any Guile code, the world is your oyster. You can extend the application in any way that you choose.
- A more cautious application might allow you to load and execute Guile code, but only in a *safe* environment, where the interface available is restricted by the application from the standard Guile API.
- Or a really fearful application might not provide a hook to really execute user code at all, but just use Scheme syntax as a convenient way for users to specify application data or configuration options.

In the last two cases, what you can do is, by definition, restricted by the application, and you should refer to the application's own manual to find out your options.

The most well known example of the first case is Emacs, with its extension language Emacs Lisp: as well as being a text editor, Emacs supports the loading and execution of arbitrary Emacs Lisp code. The result of such openness has been dramatic: Emacs now benefits from user-contributed Emacs Lisp libraries that extend the basic editing function to do everything from reading news to psychoanalysis and playing adventure games. The only limitation is that extensions are restricted to the functionality provided by Emacs's built-in set of primitive operations. For example, you can interact and display data by manipulating the contents of an Emacs buffer, but you can't pop-up and draw a window with a layout that is totally different to the Emacs standard.

This situation with a Guile application that supports the loading of arbitrary user code is similar, except perhaps even more so, because Guile also supports the loading of extension libraries written in C. This last point enables user code to add new primitive operations to Guile, and so to bypass the limitation present in Emacs Lisp.

At this point, the distinction between an application developer and an application user becomes rather blurred. Instead of seeing yourself as a user extending an application, you could equally well say that you are developing a new application of your own using some of the primitive functionality provided by the original application. As such, all the discussions of the preceding sections of this chapter are relevant to how you can proceed with developing your extension.

[5] Of course, in the world of free software, you always have the freedom to modify the application's source code to your own requirements. Here we are concerned with the extension options that the application has provided for without your needing to modify its source code.

5.8 Autoconf Support

Autoconf, a part of the GNU build system, makes it easy for users to build your package. This section documents Guile's Autoconf support.

5.8.1 Autoconf Background

As explained in the *GNU Autoconf Manual*, any package needs configuration at build-time (see Section "Introduction" in *The GNU Autoconf Manual*). If your package uses Guile (or uses a package that in turn uses Guile), you probably need to know what specific Guile features are available and details about them.

The way to do this is to write feature tests and arrange for their execution by the `configure` script, typically by adding the tests to `configure.ac`, and running `autoconf` to create `configure`. Users of your package then run `configure` in the normal way.

Macros are a way to make common feature tests easy to express. Autoconf provides a wide range of macros (see Section "Existing Tests" in *The GNU Autoconf Manual*), and Guile installation provides Guile-specific tests in the areas of: program detection, compilation flags reporting, and Scheme module checks.

5.8.2 Autoconf Macros

As mentioned earlier in this chapter, Guile supports parallel installation, and uses `pkg-config` to let the user choose which version of Guile they are interested in. `pkg-config` has its own set of Autoconf macros that are probably installed on most every development system. The most useful of these macros is `PKG_CHECK_MODULES`.

```
PKG_CHECK_MODULES([GUILE], [guile-3.0])
```

This example looks for Guile and sets the `GUILE_CFLAGS` and `GUILE_LIBS` variables accordingly, or prints an error and exits if Guile was not found.

Guile comes with additional Autoconf macros providing more information, installed as *prefix*`/share/aclocal/guile.m4`. Their names all begin with `GUILE_`.

`GUILE_PKG` [*VERSIONS*] [Autoconf Macro]

This macro runs the `pkg-config` tool to find development files for an available version of Guile.

By default, this macro will search for the latest stable version of Guile (e.g. 3.0), falling back to the previous stable version (e.g. 2.2) if it is available. If no guile-*VERSION*.pc file is found, an error is signalled. The found version is stored in *GUILE_EFFECTIVE_VERSION*.

If `GUILE_PROGS` was already invoked, this macro ensures that the development files have the same effective version as the Guile program.

GUILE_EFFECTIVE_VERSION is marked for substitution, as by `AC_SUBST`.

`GUILE_FLAGS` [Autoconf Macro]

This macro runs the `pkg-config` tool to find out how to compile and link programs against Guile. It sets four variables: *GUILE_CFLAGS*, *GUILE_LDFLAGS*, *GUILE_LIBS*, and *GUILE_LTLIBS*.

GUILE_CFLAGS: flags to pass to a C or C++ compiler to build code that uses Guile header files. This is almost always just one or more `-I` flags.

GUILE_LDFLAGS: flags to pass to the compiler to link a program against Guile. This includes `-lguile-`*`VERSION`* for the Guile library itself, and may also include one or more `-L` flag to tell the compiler where to find the libraries. But it does not include flags that influence the program's runtime search path for libraries, and will therefore lead to a program that fails to start, unless all necessary libraries are installed in a standard location such as `/usr/lib`.

GUILE_LIBS and *GUILE_LTLIBS*: flags to pass to the compiler or to libtool, respectively, to link a program against Guile. It includes flags that augment the program's runtime search path for libraries, so that shared libraries will be found at the location where they were during linking, even in non-standard locations. *GUILE_LIBS* is to be used when linking the program directly with the compiler, whereas *GUILE_LTLIBS* is to be used when linking the program is done through libtool.

The variables are marked for substitution, as by `AC_SUBST`.

`GUILE_SITE_DIR` [Autoconf Macro]

This looks for Guile's "site" directories. The variable *GUILE_SITE* will be set to Guile's "site" directory for Scheme source files (usually something like PREFIX/share/guile/site). *GUILE_SITE_CCACHE* will be set to the directory for compiled Scheme files also known as `.go` files (usually something like PREFIX/lib/guile/*GUILE_EFFECTIVE_VERSION*/site-ccache). *GUILE_EXTENSION* will be set to the directory for compiled C extensions (usually something like PREFIX/lib/guile/*GUILE_EFFECTIVE_VERSION*/extensions). The latter two are set to blank if the particular version of Guile does not support them. Note that this macro will run the macros `GUILE_PKG` and `GUILE_PROGS` if they have not already been run.

The variables are marked for substitution, as by `AC_SUBST`.

`GUILE_PROGS` [*VERSION*] [Autoconf Macro]

This macro looks for programs `guile` and `guild`, setting variables *GUILE* and *GUILD* to their paths, respectively. The macro will attempt to find `guile` with the suffix of `-X.Y`, followed by looking for it with the suffix `X.Y`, and then fall back to looking for `guile` with no suffix. If `guile` is still not found, signal an error. The suffix, if any, that was required to find `guile` will be used for `guild` as well.

By default, this macro will search for the latest stable version of Guile (e.g. 3.0). x.y or x.y.z versions can be specified. If an older version is found, the macro will signal an error.

The effective version of the found `guile` is set to *GUILE_EFFECTIVE_VERSION*. This macro ensures that the effective version is compatible with the result of a previous invocation of `GUILE_FLAGS`, if any.

As a legacy interface, it also looks for `guile-config` and `guile-tools`, setting *GUILE_CONFIG* and *GUILE_TOOLS*.

The variables are marked for substitution, as by `AC_SUBST`.

`GUILE_CHECK_RETVAL` *var check* [Autoconf Macro]

var is a shell variable name to be set to the return value. *check* is a Guile Scheme expression, evaluated with "$GUILE -c", and returning either 0 or non-#f to indicate

the check passed. Non-0 number or #f indicates failure. Avoid using the character "#" since that confuses autoconf.

`GUILE_MODULE_CHECK` *var module featuretest description* [Autoconf Macro]
var is a shell variable name to be set to "yes" or "no". *module* is a list of symbols, like: (ice-9 common-list). *featuretest* is an expression acceptable to GUILE_CHECK, q.v. *description* is a present-tense verb phrase (passed to AC_MSG_CHECKING).

`GUILE_MODULE_AVAILABLE` *var module* [Autoconf Macro]
var is a shell variable name to be set to "yes" or "no". *module* is a list of symbols, like: (ice-9 common-list).

`GUILE_MODULE_REQUIRED` *symlist* [Autoconf Macro]
symlist is a list of symbols, WITHOUT surrounding parens, like: ice-9 common-list.

`GUILE_MODULE_EXPORTS` *var module modvar* [Autoconf Macro]
var is a shell variable to be set to "yes" or "no". *module* is a list of symbols, like: (ice-9 common-list). *modvar* is the Guile Scheme variable to check.

`GUILE_MODULE_REQUIRED_EXPORT` *module modvar* [Autoconf Macro]
module is a list of symbols, like: (ice-9 common-list). *modvar* is the Guile Scheme variable to check.

5.8.3 Using Autoconf Macros

Using the autoconf macros is straightforward: Add the macro "calls" (actually instantiations) to `configure.ac`, run `aclocal`, and finally, run `autoconf`. If your system doesn't have guile.m4 installed, place the desired macro definitions (`AC_DEFUN` forms) in `acinclude.m4`, and `aclocal` will do the right thing.

Some of the macros can be used inside normal shell constructs: `if foo ; then GUILE_BAZ ; fi`, but this is not guaranteed. It's probably a good idea to instantiate macros at top-level.

We now include two examples, one simple and one complicated.

The first example is for a package that uses libguile, and thus needs to know how to compile and link against it. So we use `PKG_CHECK_MODULES` to set the vars `GUILE_CFLAGS` and `GUILE_LIBS`, which are automatically substituted in the Makefile.

```
In configure.ac:

  PKG_CHECK_MODULES([GUILE], [guile-3.0])

In Makefile.in:

  GUILE_CFLAGS  = @GUILE_CFLAGS@
  GUILE_LIBS = @GUILE_LIBS@

  myprog.o: myprog.c
          $(CC) -o $ $(GUILE_CFLAGS) $<
  myprog: myprog.o
```

```
        $(CC) -o $ $< $(GUILE_LIBS)
```

The second example is for a package of Guile Scheme modules that uses an external program and other Guile Scheme modules (some might call this a "pure scheme" package). So we use the `GUILE_SITE_DIR` macro, a regular `AC_PATH_PROG` macro, and the `GUILE_MODULE_AVAILABLE` macro.

In configure.ac:

```
GUILE_SITE_DIR

probably_wont_work=""

# pgtype pgtable
GUILE_MODULE_AVAILABLE(have_guile_pg, (database postgres))
test $have_guile_pg = no &&
    probably_wont_work="(my pgtype) (my pgtable) $probably_wont_work"

# gpgutils
AC_PATH_PROG(GNUPG,gpg)
test x"$GNUPG" = x &&
    probably_wont_work="(my gpgutils) $probably_wont_work"

if test ! "$probably_wont_work" = "" ; then
    p="         ***"
    echo
    echo "$p"
    echo "$p NOTE:"
    echo "$p The following modules probably won't work:"
    echo "$p   $probably_wont_work"
    echo "$p They can be installed anyway, and will work if their"
    echo "$p dependencies are installed later.  Please see README."
    echo "$p"
    echo
fi
```

In Makefile.in:

```
instdir = @GUILE_SITE@/my

install:
      $(INSTALL) my/*.scm $(instdir)
```

6 API Reference

Guile provides an application programming interface (*API*) to developers in two core languages: Scheme and C. This part of the manual contains reference documentation for all of the functionality that is available through both Scheme and C interfaces.

6.1 Overview of the Guile API

Guile's application programming interface (*API*) makes functionality available that an application developer can use in either C or Scheme programming. The interface consists of *elements* that may be macros, functions or variables in C, and procedures, variables, syntax or other types of object in Scheme.

Many elements are available to both Scheme and C, in a form that is appropriate. For example, the `assq` Scheme procedure is also available as `scm_assq` to C code. These elements are documented only once, addressing both the Scheme and C aspects of them.

The Scheme name of an element is related to its C name in a regular way. Also, a C function takes its parameters in a systematic way.

Normally, the name of a C function can be derived given its Scheme name, using some simple textual transformations:

- Replace `-` (hyphen) with `_` (underscore).
- Replace `?` (question mark) with `_p`.
- Replace `!` (exclamation point) with `_x`.
- Replace internal `->` with `_to_`.
- Replace `<=` (less than or equal) with `_leq`.
- Replace `>=` (greater than or equal) with `_geq`.
- Replace `<` (less than) with `_less`.
- Replace `>` (greater than) with `_gr`.
- Prefix with `scm_`.

A C function always takes a fixed number of arguments of type `SCM`, even when the corresponding Scheme function takes a variable number.

For some Scheme functions, some last arguments are optional; the corresponding C function must always be invoked with all optional arguments specified. To get the effect as if an argument has not been specified, pass `SCM_UNDEFINED` as its value. You can not do this for an argument in the middle; when one argument is `SCM_UNDEFINED` all the ones following it must be `SCM_UNDEFINED` as well.

Some Scheme functions take an arbitrary number of *rest* arguments; the corresponding C function must be invoked with a list of all these arguments. This list is always the last argument of the C function.

These two variants can also be combined.

The type of the return value of a C function that corresponds to a Scheme function is always `SCM`. In the descriptions below, types are therefore often omitted but for the return value and for the arguments.

6.2 Deprecation

From time to time functions and other features of Guile become obsolete. Guile's *deprecation* is a mechanism that can help you cope with this.

When you use a feature that is deprecated, you will likely get a warning message at run-time. Also, if you have a new enough toolchain, using a deprecated function from `libguile` will cause a link-time warning.

The primary source for information about just what interfaces are deprecated in a given release is the file `NEWS`. That file also documents what you should use instead of the obsoleted things.

The file `README` contains instructions on how to control the inclusion or removal of the deprecated features from the public API of Guile, and how to control the deprecation warning messages.

The idea behind this mechanism is that normally all deprecated interfaces are available, but you get feedback when compiling and running code that uses them, so that you can migrate to the newer APIs at your leisure.

6.3 The SCM Type

Guile represents all Scheme values with the single C type `SCM`. For an introduction to this topic, See Section 5.4.1 [Dynamic Types], page 65.

`SCM` [C Type]

`SCM` is the user level abstract C type that is used to represent all of Guile's Scheme objects, no matter what the Scheme object type is. No C operation except assignment is guaranteed to work with variables of type `SCM`, so you should only use macros and functions to work with `SCM` values. Values are converted between C data types and the `SCM` type with utility functions and macros.

`scm_t_bits` [C Type]

`scm_t_bits` is an unsigned integral data type that is guaranteed to be large enough to hold all information that is required to represent any Scheme object. While this data type is mostly used to implement Guile's internals, the use of this type is also necessary to write certain kinds of extensions to Guile.

`scm_t_signed_bits` [C Type]

This is a signed integral type of the same size as `scm_t_bits`.

`scm_t_bits SCM_UNPACK` (*SCM x*) [C Macro]

Transforms the `SCM` value *x* into its representation as an integral type. Only after applying `SCM_UNPACK` it is possible to access the bits and contents of the `SCM` value.

`SCM SCM_PACK` (*scm_t_bits x*) [C Macro]

Takes a valid integral representation of a Scheme object and transforms it into its representation as a `SCM` value.

6.4 Initializing Guile

Each thread that wants to use functions from the Guile API needs to put itself into guile mode with either `scm_with_guile` or `scm_init_guile`. The global state of Guile is initialized automatically when the first thread enters guile mode.

When a thread wants to block outside of a Guile API function, it should leave guile mode temporarily with `scm_without_guile`, See Section 6.22.6 [Blocking], page 452.

Threads that are created by `call-with-new-thread` or `scm_spawn_thread` start out in guile mode so you don't need to initialize them.

`void * scm_with_guile (`*void *(*func)(void *), void *data*`)` [C Function]

Call *func*, passing it *data* and return what *func* returns. While *func* is running, the current thread is in guile mode and can thus use the Guile API.

When `scm_with_guile` is called from guile mode, the thread remains in guile mode when `scm_with_guile` returns.

Otherwise, it puts the current thread into guile mode and, if needed, gives it a Scheme representation that is contained in the list returned by `all-threads`, for example. This Scheme representation is not removed when `scm_with_guile` returns so that a given thread is always represented by the same Scheme value during its lifetime, if at all.

When this is the first thread that enters guile mode, the global state of Guile is initialized before calling `func`.

The function *func* is called via `scm_with_continuation_barrier`; thus, `scm_with_guile` returns exactly once.

When `scm_with_guile` returns, the thread is no longer in guile mode (except when `scm_with_guile` was called from guile mode, see above). Thus, only `func` can store `SCM` variables on the stack and be sure that they are protected from the garbage collector. See `scm_init_guile` for another approach at initializing Guile that does not have this restriction.

It is OK to call `scm_with_guile` while a thread has temporarily left guile mode via `scm_without_guile`. It will then simply temporarily enter guile mode again.

`void scm_init_guile ()` [C Function]

Arrange things so that all of the code in the current thread executes as if from within a call to `scm_with_guile`. That is, all functions called by the current thread can assume that `SCM` values on their stack frames are protected from the garbage collector (except when the thread has explicitly left guile mode, of course).

When `scm_init_guile` is called from a thread that already has been in guile mode once, nothing happens. This behavior matters when you call `scm_init_guile` while the thread has only temporarily left guile mode: in that case the thread will not be in guile mode after `scm_init_guile` returns. Thus, you should not use `scm_init_guile` in such a scenario.

When a uncaught throw happens in a thread that has been put into guile mode via `scm_init_guile`, a short message is printed to the current error port and the thread is exited via `scm_pthread_exit (NULL)`. No restrictions are placed on continuations.

The function `scm_init_guile` might not be available on all platforms since it requires some stack-bounds-finding magic that might not have been ported to all platforms that Guile runs on. Thus, if you can, it is better to use `scm_with_guile` or its variation `scm_boot_guile` instead of this function.

`void scm_boot_guile (`*int* `argc`*, char **`*`argv`*, void (**`*`main_func`*`)` [C Function]
`(`*void **`*`data`*, int* `argc`*, char **`*`argv`*), void **`*`data`*`)`

Enter guile mode as with `scm_with_guile` and call *main_func*, passing it *data*, *argc*, and *argv* as indicated. When *main_func* returns, `scm_boot_guile` calls `exit (0)`; `scm_boot_guile` never returns. If you want some other exit value, have *main_func* call `exit` itself. If you don't want to exit at all, use `scm_with_guile` instead of `scm_boot_guile`.

The function `scm_boot_guile` arranges for the Scheme `command-line` function to return the strings given by *argc* and *argv*. If *main_func* modifies *argc* or *argv*, it should call `scm_set_program_arguments` with the final list, so Scheme code will know which arguments have been processed (see Section 7.2.6 [Runtime Environment], page 517).

`void scm_shell (`*int* `argc`*, char **`*`argv`*`)` [C Function]

Process command-line arguments in the manner of the `guile` executable. This includes loading the normal Guile initialization files, interacting with the user or running any scripts or expressions specified by `-s` or `-e` options, and then exiting. See Section 4.2 [Invoking Guile], page 35, for more details.

Since this function does not return, you must do all application-specific initialization before calling this function.

6.5 Snarfing Macros

The following macros do two different things: when compiled normally, they expand in one way; when processed during snarfing, they cause the `guile-snarf` program to pick up some initialization code, See Section 5.6 [Function Snarfing], page 81.

The descriptions below use the term 'normally' to refer to the case when the code is compiled normally, and 'while snarfing' when the code is processed by `guile-snarf`.

`SCM_SNARF_INIT (`*code*`)` [C Macro]

Normally, `SCM_SNARF_INIT` expands to nothing; while snarfing, it causes *code* to be included in the initialization action file, followed by a semicolon.

This is the fundamental macro for snarfing initialization actions. The more specialized macros below use it internally.

`SCM_DEFINE (`*c_name, scheme_name, req, opt, var, arglist, docstring*`)` [C Macro]

Normally, this macro expands into

```
static const char s_c_name[] = scheme_name;
SCM
c_name arglist
```

While snarfing, it causes

```
scm_c_define_gsubr (s_c_name, req, opt, var,
                    c_name);
```

to be added to the initialization actions. Thus, you can use it to declare a C function named *c_name* that will be made available to Scheme with the name *scheme_name*.

Note that the *arglist* argument must have parentheses around it.

`SCM_SYMBOL` (*c_name*, *scheme_name*) [C Macro]
`SCM_GLOBAL_SYMBOL` (*c_name*, *scheme_name*) [C Macro]

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_from_locale_symbol (scheme_name));
```

Thus, you can use them declare a static or global variable of type `SCM` that will be initialized to the symbol named *scheme_name*.

`SCM_KEYWORD` (*c_name*, *scheme_name*) [C Macro]
`SCM_GLOBAL_KEYWORD` (*c_name*, *scheme_name*) [C Macro]

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_c_make_keyword (scheme_name));
```

Thus, you can use them declare a static or global variable of type `SCM` that will be initialized to the keyword named *scheme_name*.

`SCM_VARIABLE` (*c_name*, *scheme_name*) [C Macro]
`SCM_GLOBAL_VARIABLE` (*c_name*, *scheme_name*) [C Macro]

These macros are equivalent to `SCM_VARIABLE_INIT` and `SCM_GLOBAL_VARIABLE_INIT`, respectively, with a *value* of `SCM_BOOL_F`.

`SCM_VARIABLE_INIT` (*c_name*, *scheme_name*, *value*) [C Macro]
`SCM_GLOBAL_VARIABLE_INIT` (*c_name*, *scheme_name*, *value*) [C Macro]

Normally, these macros expand into

```
static SCM c_name
```

or

```
SCM c_name
```

respectively. While snarfing, they both expand into the initialization code

```
c_name = scm_permanent_object (scm_c_define (scheme_name, value));
```

Thus, you can use them declare a static or global C variable of type `SCM` that will be initialized to the object representing the Scheme variable named *scheme_name* in the current module. The variable will be defined when it doesn't already exist. It is always set to *value*.

6.6 Data Types

Guile's data types form a powerful built-in library of representations and functionality that you can apply to your problem domain. This chapter surveys the data types built-in to Guile, from the simple to the complex.

6.6.1 Booleans

The two boolean values are `#t` for true and `#f` for false. They can also be written as `#true` and `#false`, as per R7RS.

Boolean values are returned by predicate procedures, such as the general equality predicates `eq?`, `eqv?` and `equal?` (see Section 6.11.1 [Equality], page 283) and numerical and string comparison operators like `string=?` (see Section 6.6.5.7 [String Comparison], page 148) and `<=` (see Section 6.6.2.8 [Comparison], page 117).

```
(<= 3 8)
⇒ #t

(<= 3 -3)
⇒ #f

(equal? "house" "houses")
⇒ #f

(eq? #f #f)
⇒
#t
```

In test condition contexts like `if` and `cond` (see Section 6.13.2 [Conditionals], page 299), where a group of subexpressions will be evaluated only if a *condition* expression evaluates to "true", "true" means any value at all except `#f`.

```
(if #t "yes" "no")
⇒ "yes"

(if 0 "yes" "no")
⇒ "yes"

(if #f "yes" "no")
⇒ "no"
```

A result of this asymmetry is that typical Scheme source code more often uses `#f` explicitly than `#t`: `#f` is necessary to represent an `if` or `cond` false value, whereas `#t` is not necessary to represent an `if` or `cond` true value.

It is important to note that `#f` is **not** equivalent to any other Scheme value. In particular, `#f` is not the same as the number 0 (like in C and C++), and not the same as the "empty list" (like in some Lisp dialects).

In C, the two Scheme boolean values are available as the two constants `SCM_BOOL_T` for `#t` and `SCM_BOOL_F` for `#f`. Care must be taken with the false value `SCM_BOOL_F`: it is not false when used in C conditionals. In order to test for it, use `scm_is_false` or `scm_is_true`.

`not` *x* [Scheme Procedure]
`scm_not` (*x*) [C Function]
Return `#t` if *x* is `#f`, else return `#f`.

`boolean?` *obj* [Scheme Procedure]
`scm_boolean_p` (*obj*) [C Function]
Return `#t` if *obj* is either `#t` or `#f`, else return `#f`.

`SCM SCM_BOOL_T` [C Macro]
The `SCM` representation of the Scheme object `#t`.

`SCM SCM_BOOL_F` [C Macro]
The `SCM` representation of the Scheme object `#f`.

`int scm_is_true` (*SCM obj*) [C Function]
Return `0` if *obj* is `#f`, else return `1`.

`int scm_is_false` (*SCM obj*) [C Function]
Return `1` if *obj* is `#f`, else return `0`.

`int scm_is_bool` (*SCM obj*) [C Function]
Return `1` if *obj* is either `#t` or `#f`, else return `0`.

`SCM scm_from_bool` (*int val*) [C Function]
Return `#f` if *val* is `0`, else return `#t`.

`int scm_to_bool` (*SCM val*) [C Function]
Return `1` if *val* is `SCM_BOOL_T`, return `0` when *val* is `SCM_BOOL_F`, else signal a 'wrong type' error.

You should probably use `scm_is_true` instead of this function when you just want to test a `SCM` value for trueness.

6.6.2 Numerical data types

Guile supports a rich "tower" of numerical types — integer, rational, real and complex — and provides an extensive set of mathematical and scientific functions for operating on numerical data. This section of the manual documents those types and functions.

You may also find it illuminating to read R5RS's presentation of numbers in Scheme, which is particularly clear and accessible: see Section "Numbers" in *R5RS*.

6.6.2.1 Scheme's Numerical "Tower"

Scheme's numerical "tower" consists of the following categories of numbers:

integers Whole numbers, positive or negative; e.g. –5, 0, 18.

rationals The set of numbers that can be expressed as p/q where p and q are integers; e.g. 9/16 works, but pi (an irrational number) doesn't. These include integers ($n/1$).

real numbers
The set of numbers that describes all possible positions along a one-dimensional line. This includes rationals as well as irrational numbers.

complex numbers
: The set of numbers that describes all possible positions in a two dimensional space. This includes real as well as imaginary numbers ($a + bi$, where a is the *real part*, b is the *imaginary part*, and i is the square root of -1.)

It is called a tower because each category "sits on" the one that follows it, in the sense that every integer is also a rational, every rational is also real, and every real number is also a complex number (but with zero imaginary part).

In addition to the classification into integers, rationals, reals and complex numbers, Scheme also distinguishes between whether a number is represented exactly or not. For example, the result of $2\sin(\pi/4)$ is exactly $\sqrt{2}$, but Guile can represent neither $\pi/4$ nor $\sqrt{2}$ exactly. Instead, it stores an inexact approximation, using the C type `double`.

Guile can represent exact rationals of any magnitude, inexact rationals that fit into a C `double`, and inexact complex numbers with `double` real and imaginary parts.

The `number?` predicate may be applied to any Scheme value to discover whether the value is any of the supported numerical types.

`number?` *obj* [Scheme Procedure]
`scm_number_p` (*obj*) [C Function]
Return `#t` if *obj* is any kind of number, else `#f`.

For example:

```
(number? 3)
⇒ #t

(number? "hello there!")
⇒ #f

(define pi 3.141592654)
(number? pi)
⇒ #t
```

`int scm_is_number` (*SCM obj*) [C Function]
This is equivalent to `scm_is_true (scm_number_p (obj))`.

The next few subsections document each of Guile's numerical data types in detail.

6.6.2.2 Integers

Integers are whole numbers, that is numbers with no fractional part, such as 2, 83, and -3789.

Integers in Guile can be arbitrarily big, as shown by the following example.

```
(define (factorial n)
  (let loop ((n n) (product 1))
    (if (= n 0)
        product
        (loop (- n 1) (* product n)))))
```

```
(factorial 3)
⇒ 6

(factorial 20)
⇒ 2432902008176640000

(- (factorial 45))
⇒ -119622220865480194561963161495657715064383733760000000000
```

Readers whose background is in programming languages where integers are limited by the need to fit into just 4 or 8 bytes of memory may find this surprising, or suspect that Guile's representation of integers is inefficient. In fact, Guile achieves a near optimal balance of convenience and efficiency by using the host computer's native representation of integers where possible, and a more general representation where the required number does not fit in the native form. Conversion between these two representations is automatic and completely invisible to the Scheme level programmer.

C has a host of different integer types, and Guile offers a host of functions to convert between them and the `SCM` representation. For example, a C `int` can be handled with `scm_to_int` and `scm_from_int`. Guile also defines a few C integer types of its own, to help with differences between systems.

C integer types that are not covered can be handled with the generic `scm_to_signed_integer` and `scm_from_signed_integer` for signed types, or with `scm_to_unsigned_integer` and `scm_from_unsigned_integer` for unsigned types.

Scheme integers can be exact and inexact. For example, a number written as `3.0` with an explicit decimal-point is inexact, but it is also an integer. The functions `integer?` and `scm_is_integer` report true for such a number, but the functions `exact-integer?`, `scm_is_exact_integer`, `scm_is_signed_integer`, and `scm_is_unsigned_integer` only allow exact integers and thus report false. Likewise, the conversion functions like `scm_to_signed_integer` only accept exact integers.

The motivation for this behavior is that the inexactness of a number should not be lost silently. If you want to allow inexact integers, you can explicitly insert a call to `inexact->exact` or to its C equivalent `scm_inexact_to_exact`. (Only inexact integers will be converted by this call into exact integers; inexact non-integers will become exact fractions.)

`integer?` *x* [Scheme Procedure]
`scm_integer_p` (*x*) [C Function]

Return `#t` if *x* is an exact or inexact integer number, else return `#f`.

```
(integer? 487)
⇒ #t

(integer? 3.0)
⇒ #t

(integer? -3.4)
⇒ #f
```

```
(integer? +inf.0)
⇒ #f
```

`int scm_is_integer (`*SCM x*`)` [C Function]
This is equivalent to `scm_is_true (scm_integer_p (x))`.

`exact-integer?` *x* [Scheme Procedure]
`scm_exact_integer_p (`*x*`)` [C Function]
Return `#t` if *x* is an exact integer number, else return `#f`.

```
(exact-integer? 37)
⇒ #t

(exact-integer? 3.0)
⇒ #f
```

`int scm_is_exact_integer (`*SCM x*`)` [C Function]
This is equivalent to `scm_is_true (scm_exact_integer_p (x))`.

`scm_t_int8` [C Type]
`scm_t_uint8` [C Type]
`scm_t_int16` [C Type]
`scm_t_uint16` [C Type]
`scm_t_int32` [C Type]
`scm_t_uint32` [C Type]
`scm_t_int64` [C Type]
`scm_t_uint64` [C Type]
`scm_t_intmax` [C Type]
`scm_t_uintmax` [C Type]
The C types are equivalent to the corresponding ISO C types but are defined on all platforms, with the exception of `scm_t_int64` and `scm_t_uint64`, which are only defined when a 64-bit type is available. For example, `scm_t_int8` is equivalent to `int8_t`.

You can regard these definitions as a stop-gap measure until all platforms provide these types. If you know that all the platforms that you are interested in already provide these types, it is better to use them directly instead of the types provided by Guile.

`int scm_is_signed_integer (`*SCM x, scm_t_intmax min, scm_t_intmax max*`)` [C Function]
`int scm_is_unsigned_integer (`*SCM x, scm_t_uintmax min, scm_t_uintmax max*`)` [C Function]
Return `1` when *x* represents an exact integer that is between *min* and *max*, inclusive.

These functions can be used to check whether a `SCM` value will fit into a given range, such as the range of a given C integer type. If you just want to convert a `SCM` value to a given C integer type, use one of the conversion functions directly.

`scm_t_intmax scm_to_signed_integer` (*SCM x, scm_t_intmax min, scm_t_intmax max*) [C Function]
`scm_t_uintmax scm_to_unsigned_integer` (*SCM x, scm_t_uintmax min, scm_t_uintmax max*) [C Function]

When *x* represents an exact integer that is between *min* and *max* inclusive, return that integer. Else signal an error, either a 'wrong-type' error when *x* is not an exact integer, or an 'out-of-range' error when it doesn't fit the given range.

`SCM scm_from_signed_integer` (*scm_t_intmax x*) [C Function]
`SCM scm_from_unsigned_integer` (*scm_t_uintmax x*) [C Function]

Return the `SCM` value that represents the integer *x*. This function will always succeed and will always return an exact number.

`char scm_to_char` (*SCM x*) [C Function]
`signed char scm_to_schar` (*SCM x*) [C Function]
`unsigned char scm_to_uchar` (*SCM x*) [C Function]
`short scm_to_short` (*SCM x*) [C Function]
`unsigned short scm_to_ushort` (*SCM x*) [C Function]
`int scm_to_int` (*SCM x*) [C Function]
`unsigned int scm_to_uint` (*SCM x*) [C Function]
`long scm_to_long` (*SCM x*) [C Function]
`unsigned long scm_to_ulong` (*SCM x*) [C Function]
`long long scm_to_long_long` (*SCM x*) [C Function]
`unsigned long long scm_to_ulong_long` (*SCM x*) [C Function]
`size_t scm_to_size_t` (*SCM x*) [C Function]
`ssize_t scm_to_ssize_t` (*SCM x*) [C Function]
`scm_t_uintptr scm_to_uintptr_t` (*SCM x*) [C Function]
`scm_t_ptrdiff scm_to_ptrdiff_t` (*SCM x*) [C Function]
`scm_t_int8 scm_to_int8` (*SCM x*) [C Function]
`scm_t_uint8 scm_to_uint8` (*SCM x*) [C Function]
`scm_t_int16 scm_to_int16` (*SCM x*) [C Function]
`scm_t_uint16 scm_to_uint16` (*SCM x*) [C Function]
`scm_t_int32 scm_to_int32` (*SCM x*) [C Function]
`scm_t_uint32 scm_to_uint32` (*SCM x*) [C Function]
`scm_t_int64 scm_to_int64` (*SCM x*) [C Function]
`scm_t_uint64 scm_to_uint64` (*SCM x*) [C Function]
`scm_t_intmax scm_to_intmax` (*SCM x*) [C Function]
`scm_t_uintmax scm_to_uintmax` (*SCM x*) [C Function]
`scm_t_intptr scm_to_intptr_t` (*SCM x*) [C Function]
`scm_t_uintptr scm_to_uintptr_t` (*SCM x*) [C Function]

When *x* represents an exact integer that fits into the indicated C type, return that integer. Else signal an error, either a 'wrong-type' error when *x* is not an exact integer, or an 'out-of-range' error when it doesn't fit the given range.

The functions `scm_to_long_long`, `scm_to_ulong_long`, `scm_to_int64`, and `scm_to_uint64` are only available when the corresponding types are.

`SCM scm_from_char` (*char x*) [C Function]
`SCM scm_from_schar` (*signed char x*) [C Function]

`SCM scm_from_uchar (unsigned char x)` [C Function]
`SCM scm_from_short (short x)` [C Function]
`SCM scm_from_ushort (unsigned short x)` [C Function]
`SCM scm_from_int (int x)` [C Function]
`SCM scm_from_uint (unsigned int x)` [C Function]
`SCM scm_from_long (long x)` [C Function]
`SCM scm_from_ulong (unsigned long x)` [C Function]
`SCM scm_from_long_long (long long x)` [C Function]
`SCM scm_from_ulong_long (unsigned long long x)` [C Function]
`SCM scm_from_size_t (size_t x)` [C Function]
`SCM scm_from_ssize_t (ssize_t x)` [C Function]
`SCM scm_from_uintptr_t (uintptr_t x)` [C Function]
`SCM scm_from_ptrdiff_t (scm_t_ptrdiff x)` [C Function]
`SCM scm_from_int8 (scm_t_int8 x)` [C Function]
`SCM scm_from_uint8 (scm_t_uint8 x)` [C Function]
`SCM scm_from_int16 (scm_t_int16 x)` [C Function]
`SCM scm_from_uint16 (scm_t_uint16 x)` [C Function]
`SCM scm_from_int32 (scm_t_int32 x)` [C Function]
`SCM scm_from_uint32 (scm_t_uint32 x)` [C Function]
`SCM scm_from_int64 (scm_t_int64 x)` [C Function]
`SCM scm_from_uint64 (scm_t_uint64 x)` [C Function]
`SCM scm_from_intmax (scm_t_intmax x)` [C Function]
`SCM scm_from_uintmax (scm_t_uintmax x)` [C Function]
`SCM scm_from_intptr_t (scm_t_intptr x)` [C Function]
`SCM scm_from_uintptr_t (scm_t_uintptr x)` [C Function]

Return the `SCM` value that represents the integer *x*. These functions will always succeed and will always return an exact number.

`void scm_to_mpz (SCM val, mpz_t rop)` [C Function]

Assign *val* to the multiple precision integer *rop*. *val* must be an exact integer, otherwise an error will be signalled. *rop* must have been initialized with `mpz_init` before this function is called. When *rop* is no longer needed the occupied space must be freed with `mpz_clear`. See Section "Initializing Integers" in *GNU MP Manual*, for details.

`SCM scm_from_mpz (mpz_t val)` [C Function]

Return the `SCM` value that represents *val*.

6.6.2.3 Real and Rational Numbers

Mathematically, the real numbers are the set of numbers that describe all possible points along a continuous, infinite, one-dimensional line. The rational numbers are the set of all numbers that can be written as fractions p/q, where p and q are integers. All rational numbers are also real, but there are real numbers that are not rational, for example $\sqrt{2}$, and π.

Guile can represent both exact and inexact rational numbers, but it cannot represent precise finite irrational numbers. Exact rationals are represented by storing the numera-

tor and denominator as two exact integers. Inexact rationals are stored as floating point numbers using the C type `double`.

Exact rationals are written as a fraction of integers. There must be no whitespace around the slash:

```
1/2
-22/7
```

Even though the actual encoding of inexact rationals is in binary, it may be helpful to think of it as a decimal number with a limited number of significant figures and a decimal point somewhere, since this corresponds to the standard notation for non-whole numbers. For example:

```
0.34
-0.00000142857931198
-5648394822220000000000.0
4.0
```

The limited precision of Guile's encoding means that any finite "real" number in Guile can be written in a rational form, by multiplying and then dividing by sufficient powers of 10 (or in fact, 2). For example, '`-0.00000142857931198`' is the same as −142857931198 divided by 100000000000000000. In Guile's current incarnation, therefore, the `rational?` and `real?` predicates are equivalent for finite numbers.

Dividing by an exact zero leads to a error message, as one might expect. However, dividing by an inexact zero does not produce an error. Instead, the result of the division is either plus or minus infinity, depending on the sign of the divided number and the sign of the zero divisor (some platforms support signed zeroes '`-0.0`' and '`+0.0`'; '`0.0`' is the same as '`+0.0`').

Dividing zero by an inexact zero yields a NaN ('not a number') value, although they are actually considered numbers by Scheme. Attempts to compare a NaN value with any number (including itself) using `=`, `<`, `>`, `<=` or `>=` always returns `#f`. Although a NaN value is not `=` to itself, it is both `eqv?` and `equal?` to itself and other NaN values. However, the preferred way to test for them is by using `nan?`.

The real NaN values and infinities are written '`+nan.0`', '`+inf.0`' and '`-inf.0`'. This syntax is also recognized by `read` as an extension to the usual Scheme syntax. These special values are considered by Scheme to be inexact real numbers but not rational. Note that non-real complex numbers may also contain infinities or NaN values in their real or imaginary parts. To test a real number to see if it is infinite, a NaN value, or neither, use `inf?`, `nan?`, or `finite?`, respectively. Every real number in Scheme belongs to precisely one of those three classes.

On platforms that follow IEEE 754 for their floating point arithmetic, the '`+inf.0`', '`-inf.0`', and '`+nan.0`' values are implemented using the corresponding IEEE 754 values. They behave in arithmetic operations like IEEE 754 describes it, i.e., `(= +nan.0 +nan.0)` ⇒ `#f`.

`real?` *obj* [Scheme Procedure]
`scm_real_p` (*obj*) [C Function]

Return `#t` if *obj* is a real number, else `#f`. Note that the sets of integer and rational values form subsets of the set of real numbers, so the predicate will also be fulfilled if *obj* is an integer number or a rational number.

`rational?` *x* [Scheme Procedure]
`scm_rational_p` (*x*) [C Function]
Return `#t` if *x* is a rational number, `#f` otherwise. Note that the set of integer values forms a subset of the set of rational numbers, i.e. the predicate will also be fulfilled if *x* is an integer number.

`rationalize` *x eps* [Scheme Procedure]
`scm_rationalize` (*x*, *eps*) [C Function]
Returns the *simplest* rational number differing from *x* by no more than *eps*.

As required by R5RS, `rationalize` only returns an exact result when both its arguments are exact. Thus, you might need to use `inexact->exact` on the arguments.

```
(rationalize (inexact->exact 1.2) 1/100)
⇒ 6/5
```

`inf?` *x* [Scheme Procedure]
`scm_inf_p` (*x*) [C Function]
Return `#t` if the real number *x* is '`+inf.0`' or '`-inf.0`'. Otherwise return `#f`.

`nan?` *x* [Scheme Procedure]
`scm_nan_p` (*x*) [C Function]
Return `#t` if the real number *x* is '`+nan.0`', or `#f` otherwise.

`finite?` *x* [Scheme Procedure]
`scm_finite_p` (*x*) [C Function]
Return `#t` if the real number *x* is neither infinite nor a NaN, `#f` otherwise.

`nan` [Scheme Procedure]
`scm_nan` () [C Function]
Return '`+nan.0`', a NaN value.

`inf` [Scheme Procedure]
`scm_inf` () [C Function]
Return '`+inf.0`', positive infinity.

`numerator` *x* [Scheme Procedure]
`scm_numerator` (*x*) [C Function]
Return the numerator of the rational number *x*.

`denominator` *x* [Scheme Procedure]
`scm_denominator` (*x*) [C Function]
Return the denominator of the rational number *x*.

`int scm_is_real` (*SCM val*) [C Function]
`int scm_is_rational` (*SCM val*) [C Function]
Equivalent to `scm_is_true (scm_real_p (val))` and `scm_is_true (scm_rational_p (val))`, respectively.

`double scm_to_double` (*SCM val*) [C Function]
Returns the number closest to *val* that is representable as a `double`. Returns infinity for a *val* that is too large in magnitude. The argument *val* must be a real number.

`SCM scm_from_double` (*double val*) [C Function]
Return the `SCM` value that represents *val*. The returned value is inexact according to the predicate `inexact?`, but it will be exactly equal to *val*.

6.6.2.4 Complex Numbers

Complex numbers are the set of numbers that describe all possible points in a two-dimensional space. The two coordinates of a particular point in this space are known as the *real* and *imaginary* parts of the complex number that describes that point.

In Guile, complex numbers are written in rectangular form as the sum of their real and imaginary parts, using the symbol `i` to indicate the imaginary part.

```
3+4i
⇒
3.0+4.0i

(* 3-8i 2.3+0.3i)
⇒
9.3-17.5i
```

Polar form can also be used, with an '`@`' between magnitude and angle,

```
1@3.141592 ⇒ -1.0      (approx)
-1@1.57079 ⇒ 0.0-1.0i  (approx)
```

Guile represents a complex number as a pair of inexact reals, so the real and imaginary parts of a complex number have the same properties of inexactness and limited precision as single inexact real numbers.

Note that each part of a complex number may contain any inexact real value, including the special values '`+nan.0`', '`+inf.0`' and '`-inf.0`', as well as either of the signed zeroes '`0.0`' or '`-0.0`'.

`complex?` *z* [Scheme Procedure]
`scm_complex_p` (*z*) [C Function]
Return `#t` if *z* is a complex number, `#f` otherwise. Note that the sets of real, rational and integer values form subsets of the set of complex numbers, i.e. the predicate will also be fulfilled if *z* is a real, rational or integer number.

`int scm_is_complex` (*SCM val*) [C Function]
Equivalent to `scm_is_true (scm_complex_p (val))`.

6.6.2.5 Exact and Inexact Numbers

R5RS requires that, with few exceptions, a calculation involving inexact numbers always produces an inexact result. To meet this requirement, Guile distinguishes between an exact integer value such as '`5`' and the corresponding inexact integer value which, to the limited precision available, has no fractional part, and is printed as '`5.0`'. Guile will only convert the latter value to the former when forced to do so by an invocation of the `inexact->exact` procedure.

The only exception to the above requirement is when the values of the inexact numbers do not affect the result. For example `(expt n 0)` is '`1`' for any value of `n`, therefore `(expt 5.0 0)` is permitted to return an exact '`1`'.

`exact?` *z* [Scheme Procedure]
`scm_exact_p` (*z*) [C Function]
Return `#t` if the number *z* is exact, `#f` otherwise.

```
(exact? 2)
⇒ #t

(exact? 0.5)
⇒ #f

(exact? (/ 2))
⇒ #t
```

`int scm_is_exact` (*SCM z*) [C Function]
Return a `1` if the number *z* is exact, and `0` otherwise. This is equivalent to `scm_is_true (scm_exact_p (z))`.

An alternate approch to testing the exactness of a number is to use `scm_is_signed_integer` or `scm_is_unsigned_integer`.

`inexact?` *z* [Scheme Procedure]
`scm_inexact_p` (*z*) [C Function]
Return `#t` if the number *z* is inexact, `#f` else.

`int scm_is_inexact` (*SCM z*) [C Function]
Return a `1` if the number *z* is inexact, and `0` otherwise. This is equivalent to `scm_is_true (scm_inexact_p (z))`.

`inexact->exact` *z* [Scheme Procedure]
`scm_inexact_to_exact` (*z*) [C Function]
Return an exact number that is numerically closest to *z*, when there is one. For inexact rationals, Guile returns the exact rational that is numerically equal to the inexact rational. Inexact complex numbers with a non-zero imaginary part can not be made exact.

```
(inexact->exact 0.5)
⇒ 1/2
```

The following happens because 12/10 is not exactly representable as a `double` (on most platforms). However, when reading a decimal number that has been marked exact with the "#e" prefix, Guile is able to represent it correctly.

```
(inexact->exact 1.2)
⇒ 5404319552844595/4503599627370496

#e1.2
⇒ 6/5
```

`exact->inexact` *z* [Scheme Procedure]
`scm_exact_to_inexact` (*z*) [C Function]
Convert the number *z* to its inexact representation.

6.6.2.6 Read Syntax for Numerical Data

The read syntax for integers is a string of digits, optionally preceded by a minus or plus character, a code indicating the base in which the integer is encoded, and a code indicating whether the number is exact or inexact. The supported base codes are:

`#b`
`#B` the integer is written in binary (base 2)

`#o`
`#O` the integer is written in octal (base 8)

`#d`
`#D` the integer is written in decimal (base 10)

`#x`
`#X` the integer is written in hexadecimal (base 16)

If the base code is omitted, the integer is assumed to be decimal. The following examples show how these base codes are used.

```
-13
⇒ -13

#d-13
⇒ -13

#x-13
⇒ -19

#b+1101
⇒ 13

#o377
⇒ 255
```

The codes for indicating exactness (which can, incidentally, be applied to all numerical values) are:

`#e`
`#E` the number is exact

`#i`
`#I` the number is inexact.

If the exactness indicator is omitted, the number is exact unless it contains a radix point. Since Guile can not represent exact complex numbers, an error is signalled when asking for them.

```
(exact? 1.2)
⇒ #f

(exact? #e1.2)
⇒ #t
```

```
(exact? #e+1i)
ERROR: Wrong type argument
```

Guile also understands the syntax '`+inf.0`' and '`-inf.0`' for plus and minus infinity, respectively. The value must be written exactly as shown, that is, they always must have a sign and exactly one zero digit after the decimal point. It also understands '`+nan.0`' and '`-nan.0`' for the special 'not-a-number' value. The sign is ignored for 'not-a-number' and the value is always printed as '`+nan.0`'.

6.6.2.7 Operations on Integer Values

`odd?` *n* [Scheme Procedure]
`scm_odd_p` (*n*) [C Function]

Return `#t` if *n* is an odd number, `#f` otherwise.

`even?` *n* [Scheme Procedure]
`scm_even_p` (*n*) [C Function]

Return `#t` if *n* is an even number, `#f` otherwise.

`quotient` *n d* [Scheme Procedure]
`remainder` *n d* [Scheme Procedure]
`scm_quotient` (*n*, *d*) [C Function]
`scm_remainder` (*n*, *d*) [C Function]

Return the quotient or remainder from *n* divided by *d*. The quotient is rounded towards zero, and the remainder will have the same sign as *n*. In all cases quotient and remainder satisfy $n = q * d + r$.

```
(remainder 13 4) ⇒ 1
(remainder -13 4) ⇒ -1
```

See also `truncate-quotient`, `truncate-remainder` and related operations in Section 6.6.2.11 [Arithmetic], page 119.

`modulo` *n d* [Scheme Procedure]
`scm_modulo` (*n*, *d*) [C Function]

Return the remainder from *n* divided by *d*, with the same sign as *d*.

```
(modulo 13 4) ⇒ 1
(modulo -13 4) ⇒ 3
(modulo 13 -4) ⇒ -3
(modulo -13 -4) ⇒ -1
```

See also `floor-quotient`, `floor-remainder` and related operations in Section 6.6.2.11 [Arithmetic], page 119.

`gcd` *x*... [Scheme Procedure]
`scm_gcd` (*x*, *y*) [C Function]

Return the greatest common divisor of all arguments. If called without arguments, 0 is returned.

The C function `scm_gcd` always takes two arguments, while the Scheme function can take an arbitrary number.

`lcm` *x*... [Scheme Procedure]
`scm_lcm` (*x*, *y*) [C Function]
Return the least common multiple of the arguments. If called without arguments, 1 is returned.

The C function `scm_lcm` always takes two arguments, while the Scheme function can take an arbitrary number.

`modulo-expt` *n k m* [Scheme Procedure]
`scm_modulo_expt` (*n*, *k*, *m*) [C Function]
Return *n* raised to the integer exponent *k*, modulo *m*.

```
(modulo-expt 2 3 5)
   ⇒ 3
```

`exact-integer-sqrt` *k* [Scheme Procedure]
`void scm_exact_integer_sqrt` (*SCM k*, *SCM *s*, *SCM *r*) [C Function]
Return two exact non-negative integers *s* and *r* such that $k = s^2 + r$ and $s^2 <= k < (s+1)^2$. An error is raised if *k* is not an exact non-negative integer.

```
(exact-integer-sqrt 10) ⇒ 3 and 1
```

6.6.2.8 Comparison Predicates

The C comparison functions below always takes two arguments, while the Scheme functions can take an arbitrary number. Also keep in mind that the C functions return one of the Scheme boolean values `SCM_BOOL_T` or `SCM_BOOL_F` which are both true as far as C is concerned. Thus, always write `scm_is_true (scm_num_eq_p (x, y))` when testing the two Scheme numbers `x` and `y` for equality, for example.

`=` [Scheme Procedure]
`scm_num_eq_p` (*x*, *y*) [C Function]
Return `#t` if all parameters are numerically equal.

`<` [Scheme Procedure]
`scm_less_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically increasing.

`>` [Scheme Procedure]
`scm_gr_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically decreasing.

`<=` [Scheme Procedure]
`scm_leq_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically non-decreasing.

`>=` [Scheme Procedure]
`scm_geq_p` (*x*, *y*) [C Function]
Return `#t` if the list of parameters is monotonically non-increasing.

`zero?` *z* [Scheme Procedure]
`scm_zero_p` (*z*) [C Function]
Return `#t` if *z* is an exact or inexact number equal to zero.

`positive?` *x* [Scheme Procedure]
`scm_positive_p` (*x*) [C Function]
Return `#t` if *x* is an exact or inexact number greater than zero.

`negative?` *x* [Scheme Procedure]
`scm_negative_p` (*x*) [C Function]
Return `#t` if *x* is an exact or inexact number less than zero.

6.6.2.9 Converting Numbers To and From Strings

The following procedures read and write numbers according to their external representation as defined by R5RS (see Section "Lexical structure" in *The Revised^5 Report on the Algorithmic Language Scheme*). See Section 6.25.4 [Number Input and Output], page 468, for locale-dependent number parsing.

`number->string` *n* [*radix*] [Scheme Procedure]
`scm_number_to_string` (*n*, *radix*) [C Function]
Return a string holding the external representation of the number *n* in the given *radix*. If *n* is inexact, a radix of 10 will be used.

`string->number` *string* [*radix*] [Scheme Procedure]
`scm_string_to_number` (*string*, *radix*) [C Function]
Return a number of the maximally precise representation expressed by the given *string*. *radix* must be an exact integer, either 2, 8, 10, or 16. If supplied, *radix* is a default radix that may be overridden by an explicit radix prefix in *string* (e.g. `"#o177"`). If *radix* is not supplied, then the default radix is 10. If string is not a syntactically valid notation for a number, then `string->number` returns `#f`.

`SCM scm_c_locale_stringn_to_number` (*const char *string*, *size_t len*, *unsigned radix*) [C Function]
As per `string->number` above, but taking a C string, as pointer and length. The string characters should be in the current locale encoding (`locale` in the name refers only to that, there's no locale-dependent parsing).

6.6.2.10 Complex Number Operations

`make-rectangular` *real_part imaginary_part* [Scheme Procedure]
`scm_make_rectangular` (*real_part*, *imaginary_part*) [C Function]
Return a complex number constructed of the given *real-part* and *imaginary-part* parts.

`make-polar` *mag ang* [Scheme Procedure]
`scm_make_polar` (*mag*, *ang*) [C Function]
Return the complex number *mag* * e^(i * *ang*).

`real-part` *z* [Scheme Procedure]
`scm_real_part` (*z*) [C Function]
Return the real part of the number *z*.

`imag-part` *z* [Scheme Procedure]
`scm_imag_part` (*z*) [C Function]
Return the imaginary part of the number *z*.

`magnitude` *z* [Scheme Procedure]
`scm_magnitude` (*z*) [C Function]
Return the magnitude of the number *z*. This is the same as `abs` for real arguments, but also allows complex numbers.

`angle` *z* [Scheme Procedure]
`scm_angle` (*z*) [C Function]
Return the angle of the complex number *z*.

`SCM scm_c_make_rectangular` (*double re, double im*) [C Function]
`SCM scm_c_make_polar` (*double x, double y*) [C Function]
Like `scm_make_rectangular` or `scm_make_polar`, respectively, but these functions take `double`s as their arguments.

`double scm_c_real_part` (*z*) [C Function]
`double scm_c_imag_part` (*z*) [C Function]
Returns the real or imaginary part of *z* as a `double`.

`double scm_c_magnitude` (*z*) [C Function]
`double scm_c_angle` (*z*) [C Function]
Returns the magnitude or angle of *z* as a `double`.

6.6.2.11 Arithmetic Functions

The C arithmetic functions below always takes two arguments, while the Scheme functions can take an arbitrary number. When you need to invoke them with just one argument, for example to compute the equivalent of `(- x)`, pass `SCM_UNDEFINED` as the second one: `scm_difference (x, SCM_UNDEFINED)`.

`+` *z1 ...* [Scheme Procedure]
`scm_sum` (*z1, z2*) [C Function]
Return the sum of all parameter values. Return 0 if called without any parameters.

`-` *z1 z2 ...* [Scheme Procedure]
`scm_difference` (*z1, z2*) [C Function]
If called with one argument *z1*, -*z1* is returned. Otherwise the sum of all but the first argument are subtracted from the first argument.

`*` *z1 ...* [Scheme Procedure]
`scm_product` (*z1, z2*) [C Function]
Return the product of all arguments. If called without arguments, 1 is returned.

`/` *z1 z2 ...* [Scheme Procedure]
`scm_divide` (*z1, z2*) [C Function]
Divide the first argument by the product of the remaining arguments. If called with one argument *z1*, 1/*z1* is returned.

`1+` *z* [Scheme Procedure]
`scm_oneplus` (*z*) [C Function]
Return $z + 1$.

`1-` *z* [Scheme Procedure]
`scm_oneminus` (*z*) [C function]
Return $z - 1$.

`abs` *x* [Scheme Procedure]
`scm_abs` (*x*) [C Function]
Return the absolute value of *x*.

x must be a number with zero imaginary part. To calculate the magnitude of a complex number, use `magnitude` instead.

`max` *x1 x2 ...* [Scheme Procedure]
`scm_max` (*x1*, *x2*) [C Function]
Return the maximum of all parameter values.

`min` *x1 x2 ...* [Scheme Procedure]
`scm_min` (*x1*, *x2*) [C Function]
Return the minimum of all parameter values.

`truncate` *x* [Scheme Procedure]
`scm_truncate_number` (*x*) [C Function]
Round the inexact number *x* towards zero.

`round` *x* [Scheme Procedure]
`scm_round_number` (*x*) [C Function]
Round the inexact number *x* to the nearest integer. When exactly halfway between two integers, round to the even one.

`floor` *x* [Scheme Procedure]
`scm_floor` (*x*) [C Function]
Round the number *x* towards minus infinity.

`ceiling` *x* [Scheme Procedure]
`scm_ceiling` (*x*) [C Function]
Round the number *x* towards infinity.

`double scm_c_truncate` (*double x*) [C Function]
`double scm_c_round` (*double x*) [C Function]
Like `scm_truncate_number` or `scm_round_number`, respectively, but these functions take and return `double` values.

`euclidean/` *x y* [Scheme Procedure]
`euclidean-quotient` *x y* [Scheme Procedure]
`euclidean-remainder` *x y* [Scheme Procedure]
`void scm_euclidean_divide` (*SCM x, SCM y, SCM *q, SCM *r*) [C Function]
`SCM scm_euclidean_quotient` (*SCM x, SCM y*) [C Function]
`SCM scm_euclidean_remainder` (*SCM x, SCM y*) [C Function]
These procedures accept two real numbers *x* and *y*, where the divisor *y* must be non-zero. `euclidean-quotient` returns the integer *q* and `euclidean-remainder` returns the real number *r* such that $x = q * y + r$ and $0 <= r < |y|$. `euclidean/` returns

both q and r, and is more efficient than computing each separately. Note that when $y > 0$, `euclidean-quotient` returns $floor(x/y)$, otherwise it returns $ceiling(x/y)$.

Note that these operators are equivalent to the R6RS operators `div`, `mod`, and `div-and-mod`.

```
(euclidean-quotient 123 10) ⇒ 12
(euclidean-remainder 123 10) ⇒ 3
(euclidean/ 123 10) ⇒ 12 and 3
(euclidean/ 123 -10) ⇒ -12 and 3
(euclidean/ -123 10) ⇒ -13 and 7
(euclidean/ -123 -10) ⇒ 13 and 7
(euclidean/ -123.2 -63.5) ⇒ 2.0 and 3.8
(euclidean/ 16/3 -10/7) ⇒ -3 and 22/21
```

`floor/` *x y* [Scheme Procedure]
`floor-quotient` *x y* [Scheme Procedure]
`floor-remainder` *x y* [Scheme Procedure]
`void scm_floor_divide` (*SCM x, SCM y, SCM *q, SCM *r*) [C Function]
`SCM scm_floor_quotient` (*x, y*) [C Function]
`SCM scm_floor_remainder` (*x, y*) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `floor-quotient` returns the integer q and `floor-remainder` returns the real number r such that $q = floor(x/y)$ and $x = q * y + r$. `floor/` returns both q and r, and is more efficient than computing each separately. Note that r, if non-zero, will have the same sign as y.

When x and y are integers, `floor-remainder` is equivalent to the R5RS integer-only operator `modulo`.

```
(floor-quotient 123 10) ⇒ 12
(floor-remainder 123 10) ⇒ 3
(floor/ 123 10) ⇒ 12 and 3
(floor/ 123 -10) ⇒ -13 and -7
(floor/ -123 10) ⇒ -13 and 7
(floor/ -123 -10) ⇒ 12 and -3
(floor/ -123.2 -63.5) ⇒ 1.0 and -59.7
(floor/ 16/3 -10/7) ⇒ -4 and -8/21
```

`ceiling/` *x y* [Scheme Procedure]
`ceiling-quotient` *x y* [Scheme Procedure]
`ceiling-remainder` *x y* [Scheme Procedure]
`void scm_ceiling_divide` (*SCM x, SCM y, SCM *q, SCM *r*) [C Function]
`SCM scm_ceiling_quotient` (*x, y*) [C Function]
`SCM scm_ceiling_remainder` (*x, y*) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `ceiling-quotient` returns the integer q and `ceiling-remainder` returns the real number r such that $q = ceiling(x/y)$ and $x = q * y + r$. `ceiling/` returns both q and r, and is more efficient than computing each separately. Note that r, if non-zero, will have the opposite sign of y.

```
(ceiling-quotient 123 10) ⇒ 13
```

```
(ceiling-remainder 123 10) ⇒ -7
(ceiling/ 123 10) ⇒ 13 and -7
(ceiling/ 123 -10) ⇒ -12 and 3
(ceiling/ -123 10) ⇒ -12 and -3
(ceiling/ -123 -10) ⇒ 13 and 7
(ceiling/ -123.2 -63.5) ⇒ 2.0 and 3.8
(ceiling/ 16/3 -10/7) ⇒ -3 and 22/21
```

`truncate/` *x y* [Scheme Procedure]
`truncate-quotient` *x y* [Scheme Procedure]
`truncate-remainder` *x y* [Scheme Procedure]
`void scm_truncate_divide (`*SCM x, SCM y, SCM *q, SCM *r*`)` [C Function]
`SCM scm_truncate_quotient (`*x, y*`)` [C Function]
`SCM scm_truncate_remainder (`*x, y*`)` [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `truncate-quotient` returns the integer q and `truncate-remainder` returns the real number r such that q is x/y rounded toward zero, and $x = q * y + r$. `truncate/` returns both q and r, and is more efficient than computing each separately. Note that r, if non-zero, will have the same sign as x.

When x and y are integers, these operators are equivalent to the R5RS integer-only operators `quotient` and `remainder`.

```
(truncate-quotient 123 10) ⇒ 12
(truncate-remainder 123 10) ⇒ 3
(truncate/ 123 10) ⇒ 12 and 3
(truncate/ 123 -10) ⇒ -12 and 3
(truncate/ -123 10) ⇒ -12 and -3
(truncate/ -123 -10) ⇒ 12 and -3
(truncate/ -123.2 -63.5) ⇒ 1.0 and -59.7
(truncate/ 16/3 -10/7) ⇒ -3 and 22/21
```

`centered/` *x y* [Scheme Procedure]
`centered-quotient` *x y* [Scheme Procedure]
`centered-remainder` *x y* [Scheme Procedure]
`void scm_centered_divide (`*SCM x, SCM y, SCM *q, SCM *r*`)` [C Function]
`SCM scm_centered_quotient (`*SCM x, SCM y*`)` [C Function]
`SCM scm_centered_remainder (`*SCM x, SCM y*`)` [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `centered-quotient` returns the integer q and `centered-remainder` returns the real number r such that $x = q * y + r$ and $-|y/2| <= r < |y/2|$. `centered/` returns both q and r, and is more efficient than computing each separately.

Note that `centered-quotient` returns x/y rounded to the nearest integer. When x/y lies exactly half-way between two integers, the tie is broken according to the sign of y. If $y > 0$, ties are rounded toward positive infinity, otherwise they are rounded toward negative infinity. This is a consequence of the requirement that $-|y/2| <= r < |y/2|$.

Note that these operators are equivalent to the R6RS operators `div0`, `mod0`, and `div0-and-mod0`.

```
(centered-quotient 123 10) ⇒ 12
```

```
(centered-remainder 123 10) ⇒ 3
(centered/ 123 10) ⇒ 12 and 3
(centered/ 123 -10) ⇒ -12 and 3
(centered/ -123 10) ⇒ -12 and -3
(centered/ -123 -10) ⇒ 12 and -3
(centered/ 125 10) ⇒ 13 and -5
(centered/ 127 10) ⇒ 13 and -3
(centered/ 135 10) ⇒ 14 and -5
(centered/ -123.2 -63.5) ⇒ 2.0 and 3.8
(centered/ 16/3 -10/7) ⇒ -4 and -8/21
```

`round/` *x y* [Scheme Procedure]
`round-quotient` *x y* [Scheme Procedure]
`round-remainder` *x y* [Scheme Procedure]
`void scm_round_divide` (*SCM x, SCM y, SCM *q, SCM *r*) [C Function]
`SCM scm_round_quotient` (*x*, *y*) [C Function]
`SCM scm_round_remainder` (*x*, *y*) [C Function]

These procedures accept two real numbers x and y, where the divisor y must be non-zero. `round-quotient` returns the integer q and `round-remainder` returns the real number r such that $x = q * y + r$ and q is x/y rounded to the nearest integer, with ties going to the nearest even integer. `round/` returns both q and r, and is more efficient than computing each separately.

Note that `round/` and `centered/` are almost equivalent, but their behavior differs when x/y lies exactly half-way between two integers. In this case, `round/` chooses the nearest even integer, whereas `centered/` chooses in such a way to satisfy the constraint $-|y/2| <= r < |y/2|$, which is stronger than the corresponding constraint for `round/`, $-|y/2| <= r <= |y/2|$. In particular, when x and y are integers, the number of possible remainders returned by `centered/` is $|y|$, whereas the number of possible remainders returned by `round/` is $|y| + 1$ when y is even.

```
(round-quotient 123 10) ⇒ 12
(round-remainder 123 10) ⇒ 3
(round/ 123 10) ⇒ 12 and 3
(round/ 123 -10) ⇒ -12 and 3
(round/ -123 10) ⇒ -12 and -3
(round/ -123 -10) ⇒ 12 and -3
(round/ 125 10) ⇒ 12 and 5
(round/ 127 10) ⇒ 13 and -3
(round/ 135 10) ⇒ 14 and -5
(round/ -123.2 -63.5) ⇒ 2.0 and 3.8
(round/ 16/3 -10/7) ⇒ -4 and -8/21
```

6.6.2.12 Scientific Functions

The following procedures accept any kind of number as arguments, including complex numbers.

`sqrt` *z* [Scheme Procedure]
Return the square root of *z*. Of the two possible roots (positive and negative), the one with a positive real part is returned, or if that's zero then a positive imaginary part. Thus,

```
(sqrt 9.0)       ⇒ 3.0
(sqrt -9.0)      ⇒ 0.0+3.0i
(sqrt 1.0+1.0i)  ⇒ 1.09868411346781+0.455089860562227i
(sqrt -1.0-1.0i) ⇒ 0.455089860562227-1.09868411346781i
```

`expt` *z1 z2* [Scheme Procedure]
Return *z1* raised to the power of *z2*.

`sin` *z* [Scheme Procedure]
Return the sine of *z*.

`cos` *z* [Scheme Procedure]
Return the cosine of *z*.

`tan` *z* [Scheme Procedure]
Return the tangent of *z*.

`asin` *z* [Scheme Procedure]
Return the arcsine of *z*.

`acos` *z* [Scheme Procedure]
Return the arccosine of *z*.

`atan` *z* [Scheme Procedure]
`atan` *y x* [Scheme Procedure]
Return the arctangent of *z*, or of y/x.

`exp` *z* [Scheme Procedure]
Return e to the power of *z*, where e is the base of natural logarithms (2.71828...).

`log` *z* [Scheme Procedure]
Return the natural logarithm of *z*.

`log10` *z* [Scheme Procedure]
Return the base 10 logarithm of *z*.

`sinh` *z* [Scheme Procedure]
Return the hyperbolic sine of *z*.

`cosh` *z* [Scheme Procedure]
Return the hyperbolic cosine of *z*.

`tanh` *z* [Scheme Procedure]
Return the hyperbolic tangent of *z*.

`asinh` *z* [Scheme Procedure]
Return the hyperbolic arcsine of *z*.

`acosh` *z* [Scheme Procedure]
Return the hyperbolic arccosine of *z*.

`atanh` *z* [Scheme Procedure]
Return the hyperbolic arctangent of *z*.

6.6.2.13 Bitwise Operations

For the following bitwise functions, negative numbers are treated as infinite precision twos-complements. For instance −6 is bits ...111010, with infinitely many ones on the left. It can be seen that adding 6 (binary 110) to such a bit pattern gives all zeros.

`logand` *n1 n2 ...* [Scheme Procedure]
`scm_logand` (*n1*, *n2*) [C Function]
Return the bitwise AND of the integer arguments.

```
(logand) ⇒ -1
(logand 7) ⇒ 7
(logand #b111 #b011 #b001) ⇒ 1
```

`logior` *n1 n2 ...* [Scheme Procedure]
`scm_logior` (*n1*, *n2*) [C Function]
Return the bitwise OR of the integer arguments.

```
(logior) ⇒ 0
(logior 7) ⇒ 7
(logior #b000 #b001 #b011) ⇒ 3
```

`logxor` *n1 n2 ...* [Scheme Procedure]
`scm_loxor` (*n1*, *n2*) [C Function]
Return the bitwise XOR of the integer arguments. A bit is set in the result if it is set in an odd number of arguments.

```
(logxor) ⇒ 0
(logxor 7) ⇒ 7
(logxor #b000 #b001 #b011) ⇒ 2
(logxor #b000 #b001 #b011 #b011) ⇒ 1
```

`lognot` *n* [Scheme Procedure]
`scm_lognot` (*n*) [C Function]
Return the integer which is the ones-complement of the integer argument, ie. each 0 bit is changed to 1 and each 1 bit to 0.

```
(number->string (lognot #b10000000) 2)
   ⇒ "-10000001"
(number->string (lognot #b0) 2)
   ⇒ "-1"
```

`logtest` *j k* [Scheme Procedure]
`scm_logtest` (*j*, *k*) [C Function]
Test whether *j* and *k* have any 1 bits in common. This is equivalent to `(not (zero? (logand j k)))`, but without actually calculating the `logand`, just testing for non-zero.

```
(logtest #b0100 #b1011) ⇒ #f
```

```
(logtest #b0100 #b0111) ⇒ #t
```

`logbit?` *index j* [Scheme Procedure]
`scm_logbit_p` (*index*, *j*) [C Function]

Test whether bit number *index* in *j* is set. *index* starts from 0 for the least significant bit.

```
(logbit? 0 #b1101) ⇒ #t
(logbit? 1 #b1101) ⇒ #f
(logbit? 2 #b1101) ⇒ #t
(logbit? 3 #b1101) ⇒ #t
(logbit? 4 #b1101) ⇒ #f
```

`ash` *n count* [Scheme Procedure]
`scm_ash` (*n*, *count*) [C Function]

Return $floor(n * 2^{count})$. *n* and *count* must be exact integers.

With *n* viewed as an infinite-precision twos-complement integer, `ash` means a left shift introducing zero bits when *count* is positive, or a right shift dropping bits when *count* is negative. This is an "arithmetic" shift.

```
(number->string (ash #b1 3) 2)     ⇒ "1000"
(number->string (ash #b1010 -1) 2) ⇒ "101"

;; -23 is bits ...11101001, -6 is bits ...111010
(ash -23 -2) ⇒ -6
```

`round-ash` *n count* [Scheme Procedure]
`scm_round_ash` (*n*, *count*) [C Function]

Return $round(n * 2^{c}ount)$. *n* and *count* must be exact integers.

With *n* viewed as an infinite-precision twos-complement integer, `round-ash` means a left shift introducing zero bits when *count* is positive, or a right shift rounding to the nearest integer (with ties going to the nearest even integer) when *count* is negative. This is a rounded "arithmetic" shift.

```
(number->string (round-ash #b1 3) 2)     ⇒ \"1000\"
(number->string (round-ash #b1010 -1) 2) ⇒ \"101\"
(number->string (round-ash #b1010 -2) 2) ⇒ \"10\"
(number->string (round-ash #b1011 -2) 2) ⇒ \"11\"
(number->string (round-ash #b1101 -2) 2) ⇒ \"11\"
(number->string (round-ash #b1110 -2) 2) ⇒ \"100\"
```

`logcount` *n* [Scheme Procedure]
`scm_logcount` (*n*) [C Function]

Return the number of bits in integer *n*. If *n* is positive, the 1-bits in its binary representation are counted. If negative, the 0-bits in its two's-complement binary representation are counted. If zero, 0 is returned.

```
(logcount #b10101010)
   ⇒ 4
(logcount 0)
   ⇒ 0
```

```
(logcount -2)
   ⇒ 1
```

`integer-length` *n* [Scheme Procedure]
`scm_integer_length` (*n*) [C Function]

Return the number of bits necessary to represent *n*.

For positive *n* this is how many bits to the most significant one bit. For negative *n* it's how many bits to the most significant zero bit in twos complement form.

```
(integer-length #b10101010) ⇒ 8
(integer-length #b1111)     ⇒ 4
(integer-length 0)          ⇒ 0
(integer-length -1)         ⇒ 0
(integer-length -256)       ⇒ 8
(integer-length -257)       ⇒ 9
```

`integer-expt` *n k* [Scheme Procedure]
`scm_integer_expt` (*n*, *k*) [C Function]

Return *n* raised to the power *k*. *k* must be an exact integer, *n* can be any number.

Negative *k* is supported, and results in $1/n^{|k|}$ in the usual way. n^0 is 1, as usual, and that includes 0^0 is 1.

```
(integer-expt 2 5)   ⇒ 32
(integer-expt -3 3)  ⇒ -27
(integer-expt 5 -3)  ⇒ 1/125
(integer-expt 0 0)   ⇒ 1
```

`bit-extract` *n start end* [Scheme Procedure]
`scm_bit_extract` (*n*, *start*, *end*) [C Function]

Return the integer composed of the *start* (inclusive) through *end* (exclusive) bits of *n*. The *start*th bit becomes the 0-th bit in the result.

```
(number->string (bit-extract #b1101101010 0 4) 2)
   ⇒ "1010"
(number->string (bit-extract #b1101101010 4 9) 2)
   ⇒ "10110"
```

6.6.2.14 Random Number Generation

Pseudo-random numbers are generated from a random state object, which can be created with `seed->random-state` or `datum->random-state`. An external representation (i.e. one which can written with `write` and read with `read`) of a random state object can be obtained via `random-state->datum`. The *state* parameter to the various functions below is optional, it defaults to the state object in the `*random-state*` variable.

`copy-random-state` [*state*] [Scheme Procedure]
`scm_copy_random_state` (*state*) [C Function]

Return a copy of the random state *state*.

`random` *n* [*state*] [Scheme Procedure]
`scm_random` (*n*, *state*) [C Function]

Return a number in [0, *n*).

Accepts a positive integer or real n and returns a number of the same type between zero (inclusive) and *n* (exclusive). The values returned have a uniform distribution.

`random:exp` [*state*] [Scheme Procedure]
`scm_random_exp` (*state*) [C Function]
Return an inexact real in an exponential distribution with mean 1. For an exponential distribution with mean *u* use `(* u (random:exp))`.

`random:hollow-sphere!` *vect* [*state*] [Scheme Procedure]
`scm_random_hollow_sphere_x` (*vect, state*) [C Function]
Fills *vect* with inexact real random numbers the sum of whose squares is equal to 1.0. Thinking of *vect* as coordinates in space of dimension *n* = `(vector-length vect)`, the coordinates are uniformly distributed over the surface of the unit n-sphere.

`random:normal` [*state*] [Scheme Procedure]
`scm_random_normal` (*state*) [C Function]
Return an inexact real in a normal distribution. The distribution used has mean 0 and standard deviation 1. For a normal distribution with mean *m* and standard deviation *d* use `(+ m (* d (random:normal)))`.

`random:normal-vector!` *vect* [*state*] [Scheme Procedure]
`scm_random_normal_vector_x` (*vect, state*) [C Function]
Fills *vect* with inexact real random numbers that are independent and standard normally distributed (i.e., with mean 0 and variance 1).

`random:solid-sphere!` *vect* [*state*] [Scheme Procedure]
`scm_random_solid_sphere_x` (*vect, state*) [C Function]
Fills *vect* with inexact real random numbers the sum of whose squares is less than 1.0. Thinking of *vect* as coordinates in space of dimension *n* = `(vector-length vect)`, the coordinates are uniformly distributed within the unit *n*-sphere.

`random:uniform` [*state*] [Scheme Procedure]
`scm_random_uniform` (*state*) [C Function]
Return a uniformly distributed inexact real random number in [0,1).

`seed->random-state` *seed* [Scheme Procedure]
`scm_seed_to_random_state` (*seed*) [C Function]
Return a new random state using *seed*.

`datum->random-state` *datum* [Scheme Procedure]
`scm_datum_to_random_state` (*datum*) [C Function]
Return a new random state from *datum*, which should have been obtained by `random-state->datum`.

`random-state->datum` *state* [Scheme Procedure]
`scm_random_state_to_datum` (*state*) [C Function]
Return a datum representation of *state* that may be written out and read back with the Scheme reader.

`random-state-from-platform` [Scheme Procedure]
`scm_random_state_from_platform ()` [C Function]

Construct a new random state seeded from a platform-specific source of entropy, appropriate for use in non-security-critical applications. Currently `/dev/urandom` is tried first, or else the seed is based on the time, date, process ID, an address from a freshly allocated heap cell, an address from the local stack frame, and a high-resolution timer if available.

`*random-state*` [Variable]

The global random state used by the above functions when the *state* parameter is not given.

Note that the initial value of `*random-state*` is the same every time Guile starts up. Therefore, if you don't pass a *state* parameter to the above procedures, and you don't set `*random-state*` to `(seed->random-state your-seed)`, where `your-seed` is something that *isn't* the same every time, you'll get the same sequence of "random" numbers on every run.

For example, unless the relevant source code has changed, `(map random (cdr (iota 30)))`, if the first use of random numbers since Guile started up, will always give:

```
(map random (cdr (iota 19)))
⇒
(0 1 1 2 2 2 1 2 6 7 10 0 5 3 12 5 5 12)
```

To seed the random state in a sensible way for non-security-critical applications, do this during initialization of your program:

```
(set! *random-state* (random-state-from-platform))
```

6.6.3 Characters

In Scheme, there is a data type to describe a single character.

Defining what exactly a character *is* can be more complicated than it seems. Guile follows the advice of R6RS and uses The Unicode Standard to help define what a character is. So, for Guile, a character is anything in the Unicode Character Database.

The Unicode Character Database is basically a table of characters indexed using integers called 'code points'. Valid code points are in the ranges 0 to `#xD7FF` inclusive or `#xE000` to `#x10FFFF` inclusive, which is about 1.1 million code points.

Any code point that has been assigned to a character or that has otherwise been given a meaning by Unicode is called a 'designated code point'. Most of the designated code points, about 200,000 of them, indicate characters, accents or other combining marks that modify other characters, symbols, whitespace, and control characters. Some are not characters but indicators that suggest how to format or display neighboring characters.

If a code point is not a designated code point – if it has not been assigned to a character by The Unicode Standard – it is a 'reserved code point', meaning that they are reserved for future use. Most of the code points, about 800,000, are 'reserved code points'.

By convention, a Unicode code point is written as "U+XXXX" where "XXXX" is a hexadecimal number. Please note that this convenient notation is not valid code. Guile does not interpret "U+XXXX" as a character.

In Scheme, a character literal is written as `#\name` where *name* is the name of the character that you want. Printable characters have their usual single character name; for example, `#\a` is a lower case `a`.

Some of the code points are 'combining characters' that are not meant to be printed by themselves but are instead meant to modify the appearance of the previous character. For combining characters, an alternate form of the character literal is `#\` followed by U+25CC (a small, dotted circle), followed by the combining character. This allows the combining character to be drawn on the circle, not on the backslash of `#\`.

Many of the non-printing characters, such as whitespace characters and control characters, also have names.

The most commonly used non-printing characters have long character names, described in the table below.

Character Name	Codepoint
`#\nul`	U+0000
`#\alarm`	U+0007
`#\backspace`	U+0008
`#\tab`	U+0009
`#\linefeed`	U+000A
`#\newline`	U+000A
`#\vtab`	U+000B
`#\page`	U+000C
`#\return`	U+000D
`#\esc`	U+001B
`#\space`	U+0020
`#\delete`	U+007F

There are also short names for all of the "C0 control characters" (those with code points below 32). The following table lists the short name for each character.

0 = `#\nul`	1 = `#\soh`	2 = `#\stx`	3 = `#\etx`
4 = `#\eot`	5 = `#\enq`	6 = `#\ack`	7 = `#\bel`
8 = `#\bs`	9 = `#\ht`	10 = `#\lf`	11 = `#\vt`
12 = `#\ff`	13 = `#\cr`	14 = `#\so`	15 = `#\si`
16 = `#\dle`	17 = `#\dc1`	18 = `#\dc2`	19 = `#\dc3`
20 = `#\dc4`	21 = `#\nak`	22 = `#\syn`	23 = `#\etb`
24 = `#\can`	25 = `#\em`	26 = `#\sub`	27 = `#\esc`
28 = `#\fs`	29 = `#\gs`	30 = `#\rs`	31 = `#\us`
32 = `#\sp`			

The short name for the "delete" character (code point U+007F) is `#\del`.

The R7RS name for the "escape" character (code point U+001B) is `#\escape`.

There are also a few alternative names left over for compatibility with previous versions of Guile.

Alternate	Standard
`#\nl`	`#\newline`
`#\np`	`#\page`

`#\null` `#\nul`

Characters may also be written using their code point values. They can be written with as an octal number, such as `#\10` for `#\bs` or `#\177` for `#\del`.

If one prefers hex to octal, there is an additional syntax for character escapes: `#\xHHHH` – the letter 'x' followed by a hexadecimal number of one to eight digits.

`char?` *x* [Scheme Procedure]
`scm_char_p` (*x*) [C Function]
Return `#t` if *x* is a character, else `#f`.

Fundamentally, the character comparison operations below are numeric comparisons of the character's code points.

`char=?` *x y* [Scheme Procedure]
Return `#t` if code point of *x* is equal to the code point of *y*, else `#f`.

`char<?` *x y* [Scheme Procedure]
Return `#t` if the code point of *x* is less than the code point of *y*, else `#f`.

`char<=?` *x y* [Scheme Procedure]
Return `#t` if the code point of *x* is less than or equal to the code point of *y*, else `#f`.

`char>?` *x y* [Scheme Procedure]
Return `#t` if the code point of *x* is greater than the code point of *y*, else `#f`.

`char>=?` *x y* [Scheme Procedure]
Return `#t` if the code point of *x* is greater than or equal to the code point of *y*, else `#f`.

Case-insensitive character comparisons use *Unicode case folding*. In case folding comparisons, if a character is lowercase and has an uppercase form that can be expressed as a single character, it is converted to uppercase before comparison. All other characters undergo no conversion before the comparison occurs. This includes the German sharp S (Eszett) which is not uppercased before conversion because its uppercase form has two characters. Unicode case folding is language independent: it uses rules that are generally true, but, it cannot cover all cases for all languages.

`char-ci=?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is the same as the case-folded code point of *y*, else `#f`.

`char-ci<?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is less than the case-folded code point of *y*, else `#f`.

`char-ci<=?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is less than or equal to the case-folded code point of *y*, else `#f`.

`char-ci>?` *x y* [Scheme Procedure]
Return `#t` if the case-folded code point of *x* is greater than the case-folded code point of *y*, else `#f`.

`char-ci>=?` *x y* [Scheme Procedure]

Return `#t` if the case-folded code point of *x* is greater than or equal to the case-folded code point of *y*, else `#f`.

`char-alphabetic?` *chr* [Scheme Procedure]
`scm_char_alphabetic_p` (*chr*) [C Function]

Return `#t` if *chr* is alphabetic, else `#f`.

`char-numeric?` *chr* [Scheme Procedure]
`scm_char_numeric_p` (*chr*) [C Function]

Return `#t` if *chr* is numeric, else `#f`.

`char-whitespace?` *chr* [Scheme Procedure]
`scm_char_whitespace_p` (*chr*) [C Function]

Return `#t` if *chr* is whitespace, else `#f`.

`char-upper-case?` *chr* [Scheme Procedure]
`scm_char_upper_case_p` (*chr*) [C Function]

Return `#t` if *chr* is uppercase, else `#f`.

`char-lower-case?` *chr* [Scheme Procedure]
`scm_char_lower_case_p` (*chr*) [C Function]

Return `#t` if *chr* is lowercase, else `#f`.

`char-is-both?` *chr* [Scheme Procedure]
`scm_char_is_both_p` (*chr*) [C Function]

Return `#t` if *chr* is either uppercase or lowercase, else `#f`.

`char-general-category` *chr* [Scheme Procedure]
`scm_char_general_category` (*chr*) [C Function]

Return a symbol giving the two-letter name of the Unicode general category assigned to *chr* or `#f` if no named category is assigned. The following table provides a list of category names along with their meanings.

Lu	Uppercase letter	Pf	Final quote punctuation
Ll	Lowercase letter	Po	Other punctuation
Lt	Titlecase letter	Sm	Math symbol
Lm	Modifier letter	Sc	Currency symbol
Lo	Other letter	Sk	Modifier symbol
Mn	Non-spacing mark	So	Other symbol
Mc	Combining spacing mark	Zs	Space separator
Me	Enclosing mark	Zl	Line separator
Nd	Decimal digit number	Zp	Paragraph separator
Nl	Letter number	Cc	Control
No	Other number	Cf	Format
Pc	Connector punctuation	Cs	Surrogate
Pd	Dash punctuation	Co	Private use
Ps	Open punctuation	Cn	Unassigned
Pe	Close punctuation		
Pi	Initial quote punctuation		

`char->integer` *chr* [Scheme Procedure]
`scm_char_to_integer` (*chr*) [C Function]
Return the code point of *chr*.

`integer->char` *n* [Scheme Procedure]
`scm_integer_to_char` (*n*) [C Function]
Return the character that has code point *n*. The integer *n* must be a valid code point. Valid code points are in the ranges 0 to `#xD7FF` inclusive or `#xE000` to `#x10FFFF` inclusive.

`char-upcase` *chr* [Scheme Procedure]
`scm_char_upcase` (*chr*) [C Function]
Return the uppercase character version of *chr*.

`char-downcase` *chr* [Scheme Procedure]
`scm_char_downcase` (*chr*) [C Function]
Return the lowercase character version of *chr*.

`char-titlecase` *chr* [Scheme Procedure]
`scm_char_titlecase` (*chr*) [C Function]
Return the titlecase character version of *chr* if one exists; otherwise return the uppercase version.

For most characters these will be the same, but the Unicode Standard includes certain digraph compatibility characters, such as `U+01F3` "dz", for which the uppercase and titlecase characters are different (`U+01F1` "DZ" and `U+01F2` "Dz" in this case, respectively).

`scm_t_wchar scm_c_upcase` (*scm_t_wchar c*) [C Function]
`scm_t_wchar scm_c_downcase` (*scm_t_wchar c*) [C Function]
`scm_t_wchar scm_c_titlecase` (*scm_t_wchar c*) [C Function]
These C functions take an integer representation of a Unicode codepoint and return the codepoint corresponding to its uppercase, lowercase, and titlecase forms respectively. The type `scm_t_wchar` is a signed, 32-bit integer.

Characters also have "formal names", which are defined by Unicode. These names can be accessed in Guile from the `(ice-9 unicode)` module:

```
(use-modules (ice-9 unicode))
```

`char->formal-name` *chr* [Scheme Procedure]
Return the formal all-upper-case Unicode name of *ch*, as a string, or `#f` if the character has no name.

`formal-name->char` *name* [Scheme Procedure]
Return the character whose formal all-upper-case Unicode name is *name*, or `#f` if no such character is known.

6.6.4 Character Sets

The features described in this section correspond directly to SRFI-14.

The data type *charset* implements sets of characters (see Section 6.6.3 [Characters], page 129). Because the internal representation of character sets is not visible to the user, a lot of procedures for handling them are provided.

Character sets can be created, extended, tested for the membership of a characters and be compared to other character sets.

6.6.4.1 Character Set Predicates/Comparison

Use these procedures for testing whether an object is a character set, or whether several character sets are equal or subsets of each other. `char-set-hash` can be used for calculating a hash value, maybe for usage in fast lookup procedures.

`char-set?` *obj* [Scheme Procedure]
`scm_char_set_p` (*obj*) [C Function]
Return `#t` if *obj* is a character set, `#f` otherwise.

`char-set=` *char_set ...* [Scheme Procedure]
`scm_char_set_eq` (*char_sets*) [C Function]
Return `#t` if all given character sets are equal.

`char-set<=` *char_set ...* [Scheme Procedure]
`scm_char_set_leq` (*char_sets*) [C Function]
Return `#t` if every character set *char_set*i is a subset of character set *char_set*i+1.

`char-set-hash` *cs* [*bound*] [Scheme Procedure]
`scm_char_set_hash` (*cs*, *bound*) [C Function]
Compute a hash value for the character set *cs*. If *bound* is given and non-zero, it restricts the returned value to the range 0 ... *bound* - 1.

6.6.4.2 Iterating Over Character Sets

Character set cursors are a means for iterating over the members of a character sets. After creating a character set cursor with `char-set-cursor`, a cursor can be dereferenced with `char-set-ref`, advanced to the next member with `char-set-cursor-next`. Whether a cursor has passed past the last element of the set can be checked with `end-of-char-set?`.

Additionally, mapping and (un-)folding procedures for character sets are provided.

`char-set-cursor` *cs* [Scheme Procedure]
`scm_char_set_cursor` (*cs*) [C Function]
Return a cursor into the character set *cs*.

`char-set-ref` *cs cursor* [Scheme Procedure]
`scm_char_set_ref` (*cs*, *cursor*) [C Function]
Return the character at the current cursor position *cursor* in the character set *cs*. It is an error to pass a cursor for which `end-of-char-set?` returns true.

`char-set-cursor-next` *cs cursor* [Scheme Procedure]
`scm_char_set_cursor_next` (*cs*, *cursor*) [C Function]
Advance the character set cursor *cursor* to the next character in the character set *cs*. It is an error if the cursor given satisfies `end-of-char-set?`.

`end-of-char-set?` *cursor* [Scheme Procedure]
`scm_end_of_char_set_p` (*cursor*) [C Function]
Return `#t` if *cursor* has reached the end of a character set, `#f` otherwise.

`char-set-fold` *kons knil cs* [Scheme Procedure]
`scm_char_set_fold` (*kons, knil, cs*) [C Function]
Fold the procedure *kons* over the character set *cs*, initializing it with *knil*.

`char-set-unfold` *p f g seed* [*base_cs*] [Scheme Procedure]
`scm_char_set_unfold` (*p, f, g, seed, base_cs*) [C Function]
This is a fundamental constructor for character sets.

- *g* is used to generate a series of "seed" values from the initial seed: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of the seed values.
- *f* maps each seed value to a character. These characters are added to the base character set *base_cs* to form the result; *base_cs* defaults to the empty set.

`char-set-unfold!` *p f g seed base_cs* [Scheme Procedure]
`scm_char_set_unfold_x` (*p, f, g, seed, base_cs*) [C Function]
This is a fundamental constructor for character sets.

- *g* is used to generate a series of "seed" values from the initial seed: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of the seed values.
- *f* maps each seed value to a character. These characters are added to the base character set *base_cs* to form the result; *base_cs* defaults to the empty set.

`char-set-for-each` *proc cs* [Scheme Procedure]
`scm_char_set_for_each` (*proc, cs*) [C Function]
Apply *proc* to every character in the character set *cs*. The return value is not specified.

`char-set-map` *proc cs* [Scheme Procedure]
`scm_char_set_map` (*proc, cs*) [C Function]
Map the procedure *proc* over every character in *cs*. *proc* must be a character -> character procedure.

6.6.4.3 Creating Character Sets

New character sets are produced with these procedures.

`char-set-copy` *cs* [Scheme Procedure]
`scm_char_set_copy` (*cs*) [C Function]
Return a newly allocated character set containing all characters in *cs*.

`char-set` *chr ...* [Scheme Procedure]
`scm_char_set` (*chrs*) [C Function]
Return a character set containing all given characters.

`list->char-set` *list* [*base_cs*] [Scheme Procedure]
`scm_list_to_char_set` (*list*, *base_cs*) [C Function]
Convert the character list *list* to a character set. If the character set *base_cs* is given, the character in this set are also included in the result.

`list->char-set!` *list base_cs* [Scheme Procedure]
`scm_list_to_char_set_x` (*list*, *base_cs*) [C Function]
Convert the character list *list* to a character set. The characters are added to *base_cs* and *base_cs* is returned.

`string->char-set` *str* [*base_cs*] [Scheme Procedure]
`scm_string_to_char_set` (*str*, *base_cs*) [C Function]
Convert the string *str* to a character set. If the character set *base_cs* is given, the characters in this set are also included in the result.

`string->char-set!` *str base_cs* [Scheme Procedure]
`scm_string_to_char_set_x` (*str*, *base_cs*) [C Function]
Convert the string *str* to a character set. The characters from the string are added to *base_cs*, and *base_cs* is returned.

`char-set-filter` *pred cs* [*base_cs*] [Scheme Procedure]
`scm_char_set_filter` (*pred*, *cs*, *base_cs*) [C Function]
Return a character set containing every character from *cs* so that it satisfies *pred*. If provided, the characters from *base_cs* are added to the result.

`char-set-filter!` *pred cs base_cs* [Scheme Procedure]
`scm_char_set_filter_x` (*pred*, *cs*, *base_cs*) [C Function]
Return a character set containing every character from *cs* so that it satisfies *pred*. The characters are added to *base_cs* and *base_cs* is returned.

`ucs-range->char-set` *lower upper* [*error* [*base_cs*]] [Scheme Procedure]
`scm_ucs_range_to_char_set` (*lower*, *upper*, *error*, *base_cs*) [C Function]
Return a character set containing all characters whose character codes lie in the half-open range [*lower*,*upper*).

If *error* is a true value, an error is signalled if the specified range contains characters which are not contained in the implemented character range. If *error* is `#f`, these characters are silently left out of the resulting character set.

The characters in *base_cs* are added to the result, if given.

`ucs-range->char-set!` *lower upper error base_cs* [Scheme Procedure]
`scm_ucs_range_to_char_set_x` (*lower*, *upper*, *error*, *base_cs*) [C Function]
Return a character set containing all characters whose character codes lie in the half-open range [*lower*,*upper*).

If *error* is a true value, an error is signalled if the specified range contains characters which are not contained in the implemented character range. If *error* is `#f`, these characters are silently left out of the resulting character set.

The characters are added to *base_cs* and *base_cs* is returned.

`->char-set` *x* [Scheme Procedure]
`scm_to_char_set` (*x*) [C Function]
Coerces x into a char-set. *x* may be a string, character or char-set. A string is converted to the set of its constituent characters; a character is converted to a singleton set; a char-set is returned as-is.

6.6.4.4 Querying Character Sets

Access the elements and other information of a character set with these procedures.

`%char-set-dump` *cs* [Scheme Procedure]
Returns an association list containing debugging information for *cs*. The association list has the following entries.

`char-set` The char-set itself

`len` The number of groups of contiguous code points the char-set contains

`ranges` A list of lists where each sublist is a range of code points and their associated characters

The return value of this function cannot be relied upon to be consistent between versions of Guile and should not be used in code.

`char-set-size` *cs* [Scheme Procedure]
`scm_char_set_size` (*cs*) [C Function]
Return the number of elements in character set *cs*.

`char-set-count` *pred cs* [Scheme Procedure]
`scm_char_set_count` (*pred*, *cs*) [C Function]
Return the number of the elements int the character set *cs* which satisfy the predicate *pred*.

`char-set->list` *cs* [Scheme Procedure]
`scm_char_set_to_list` (*cs*) [C Function]
Return a list containing the elements of the character set *cs*.

`char-set->string` *cs* [Scheme Procedure]
`scm_char_set_to_string` (*cs*) [C Function]
Return a string containing the elements of the character set *cs*. The order in which the characters are placed in the string is not defined.

`char-set-contains?` *cs ch* [Scheme Procedure]
`scm_char_set_contains_p` (*cs*, *ch*) [C Function]
Return `#t` if the character *ch* is contained in the character set *cs*, or `#f` otherwise.

`char-set-every` *pred cs* [Scheme Procedure]
`scm_char_set_every` (*pred*, *cs*) [C Function]
Return a true value if every character in the character set *cs* satisfies the predicate *pred*.

`char-set-any` *pred cs* [Scheme Procedure]
`scm_char_set_any` (*pred, cs*) [C Function]
Return a true value if any character in the character set *cs* satisfies the predicate *pred*.

6.6.4.5 Character-Set Algebra

Character sets can be manipulated with the common set algebra operation, such as union, complement, intersection etc. All of these procedures provide side-effecting variants, which modify their character set argument(s).

`char-set-adjoin` *cs chr ...* [Scheme Procedure]
`scm_char_set_adjoin` (*cs, chrs*) [C Function]
Add all character arguments to the first argument, which must be a character set.

`char-set-delete` *cs chr ...* [Scheme Procedure]
`scm_char_set_delete` (*cs, chrs*) [C Function]
Delete all character arguments from the first argument, which must be a character set.

`char-set-adjoin!` *cs chr ...* [Scheme Procedure]
`scm_char_set_adjoin_x` (*cs, chrs*) [C Function]
Add all character arguments to the first argument, which must be a character set.

`char-set-delete!` *cs chr ...* [Scheme Procedure]
`scm_char_set_delete_x` (*cs, chrs*) [C Function]
Delete all character arguments from the first argument, which must be a character set.

`char-set-complement` *cs* [Scheme Procedure]
`scm_char_set_complement` (*cs*) [C Function]
Return the complement of the character set *cs*.

Note that the complement of a character set is likely to contain many reserved code points (code points that are not associated with characters). It may be helpful to modify the output of `char-set-complement` by computing its intersection with the set of designated code points, `char-set:designated`.

`char-set-union` *cs ...* [Scheme Procedure]
`scm_char_set_union` (*char_sets*) [C Function]
Return the union of all argument character sets.

`char-set-intersection` *cs ...* [Scheme Procedure]
`scm_char_set_intersection` (*char_sets*) [C Function]
Return the intersection of all argument character sets.

`char-set-difference` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_difference` (*cs1, char_sets*) [C Function]
Return the difference of all argument character sets.

`char-set-xor` *cs ...* [Scheme Procedure]
`scm_char_set_xor` (*char_sets*) [C Function]
Return the exclusive-or of all argument character sets.

`char-set-diff+intersection` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_diff_plus_intersection` (*cs1*, *char_sets*) [C Function]
Return the difference and the intersection of all argument character sets.

`char-set-complement!` *cs* [Scheme Procedure]
`scm_char_set_complement_x` (*cs*) [C Function]
Return the complement of the character set *cs*.

`char-set-union!` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_union_x` (*cs1*, *char_sets*) [C Function]
Return the union of all argument character sets.

`char-set-intersection!` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_intersection_x` (*cs1*, *char_sets*) [C Function]
Return the intersection of all argument character sets.

`char-set-difference!` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_difference_x` (*cs1*, *char_sets*) [C Function]
Return the difference of all argument character sets.

`char-set-xor!` *cs1 cs ...* [Scheme Procedure]
`scm_char_set_xor_x` (*cs1*, *char_sets*) [C Function]
Return the exclusive-or of all argument character sets.

`char-set-diff+intersection!` *cs1 cs2 cs ...* [Scheme Procedure]
`scm_char_set_diff_plus_intersection_x` (*cs1*, *cs2*, *char_sets*) [C Function]
Return the difference and the intersection of all argument character sets.

6.6.4.6 Standard Character Sets

In order to make the use of the character set data type and procedures useful, several predefined character set variables exist.

These character sets are locale independent and are not recomputed upon a `setlocale` call. They contain characters from the whole range of Unicode code points. For instance, `char-set:letter` contains about 100,000 characters.

`char-set:lower-case` [Scheme Variable]
`scm_char_set_lower_case` [C Variable]
All lower-case characters.

`char-set:upper-case` [Scheme Variable]
`scm_char_set_upper_case` [C Variable]
All upper-case characters.

`char-set:title-case` [Scheme Variable]
`scm_char_set_title_case` [C Variable]
All single characters that function as if they were an upper-case letter followed by a lower-case letter.

`char-set:letter` [Scheme Variable]
`scm_char_set_letter` [C Variable]
All letters. This includes `char-set:lower-case`, `char-set:upper-case`, `char-set:title-case`, and many letters that have no case at all. For example, Chinese and Japanese characters typically have no concept of case.

`char-set:digit` [Scheme Variable]
`scm_char_set_digit` [C Variable]
All digits.

`char-set:letter+digit` [Scheme Variable]
`scm_char_set_letter_and_digit` [C Variable]
The union of `char-set:letter` and `char-set:digit`.

`char-set:graphic` [Scheme Variable]
`scm_char_set_graphic` [C Variable]
All characters which would put ink on the paper.

`char-set:printing` [Scheme Variable]
`scm_char_set_printing` [C Variable]
The union of `char-set:graphic` and `char-set:whitespace`.

`char-set:whitespace` [Scheme Variable]
`scm_char_set_whitespace` [C Variable]
All whitespace characters.

`char-set:blank` [Scheme Variable]
`scm_char_set_blank` [C Variable]
All horizontal whitespace characters, which notably includes `#\space` and `#\tab`.

`char-set:iso-control` [Scheme Variable]
`scm_char_set_iso_control` [C Variable]
The ISO control characters are the C0 control characters (U+0000 to U+001F), delete (U+007F), and the C1 control characters (U+0080 to U+009F).

`char-set:punctuation` [Scheme Variable]
`scm_char_set_punctuation` [C Variable]
All punctuation characters, such as the characters `!"#%&'()*,-./:;?@[\\]_{}`

`char-set:symbol` [Scheme Variable]
`scm_char_set_symbol` [C Variable]
All symbol characters, such as the characters `$+<=>^`|~`.

`char-set:hex-digit` [Scheme Variable]
`scm_char_set_hex_digit` [C Variable]
The hexadecimal digits `0123456789abcdefABCDEF`.

`char-set:ascii` [Scheme Variable]
`scm_char_set_ascii` [C Variable]
All ASCII characters.

`char-set:empty` [Scheme Variable]
`scm_char_set_empty` [C Variable]
The empty character set.

`char-set:designated` [Scheme Variable]
`scm_char_set_designated` [C Variable]
This character set contains all designated code points. This includes all the code points to which Unicode has assigned a character or other meaning.

`char-set:full` [Scheme Variable]
`scm_char_set_full` [C Variable]
This character set contains all possible code points. This includes both designated and reserved code points.

6.6.5 Strings

Strings are fixed-length sequences of characters. They can be created by calling constructor procedures, but they can also literally get entered at the REPL or in Scheme source files.

Strings always carry the information about how many characters they are composed of with them, so there is no special end-of-string character, like in C. That means that Scheme strings can contain any character, even the '`#\nul`' character '`\0`'.

To use strings efficiently, you need to know a bit about how Guile implements them. In Guile, a string consists of two parts, a head and the actual memory where the characters are stored. When a string (or a substring of it) is copied, only a new head gets created, the memory is usually not copied. The two heads start out pointing to the same memory.

When one of these two strings is modified, as with `string-set!`, their common memory does get copied so that each string has its own memory and modifying one does not accidentally modify the other as well. Thus, Guile's strings are 'copy on write'; the actual copying of their memory is delayed until one string is written to.

This implementation makes functions like `substring` very efficient in the common case that no modifications are done to the involved strings.

If you do know that your strings are getting modified right away, you can use `substring/copy` instead of `substring`. This function performs the copy immediately at the time of creation. This is more efficient, especially in a multi-threaded program. Also, `substring/copy` can avoid the problem that a short substring holds on to the memory of a very large original string that could otherwise be recycled.

If you want to avoid the copy altogether, so that modifications of one string show up in the other, you can use `substring/shared`. The strings created by this procedure are called *mutation sharing substrings* since the substring and the original string share modifications to each other.

If you want to prevent modifications, use `substring/read-only`.

Guile provides all procedures of SRFI-13 and a few more.

6.6.5.1 String Read Syntax

The read syntax for strings is an arbitrarily long sequence of characters enclosed in double quotes (`"`).

Backslash is an escape character and can be used to insert the following special characters. `\"` and `\\` are R5RS standard, `\|` is R7RS standard, the next seven are R6RS standard — notice they follow C syntax — and the remaining four are Guile extensions.

`\\` Backslash character.

`\"` Double quote character (an unescaped `"` is otherwise the end of the string).

`\|` Vertical bar character.

`\a` Bell character (ASCII 7).

`\f` Formfeed character (ASCII 12).

`\n` Newline character (ASCII 10).

`\r` Carriage return character (ASCII 13).

`\t` Tab character (ASCII 9).

`\v` Vertical tab character (ASCII 11).

`\b` Backspace character (ASCII 8).

`\0` NUL character (ASCII 0).

`\(` Open parenthesis. This is intended for use at the beginning of lines in multiline strings to avoid confusing Emacs lisp modes.

`\` followed by newline (ASCII 10)
Nothing. This way if `\` is the last character in a line, the string will continue with the first character from the next line, without a line break.

If the `hungry-eol-escapes` reader option is enabled, which is not the case by default, leading whitespace on the next line is discarded.

```
"foo\
  bar"
⇒ "foo  bar"
(read-enable 'hungry-eol-escapes)
"foo\
  bar"
⇒ "foobar"
```

`\xHH` Character code given by two hexadecimal digits. For example `\x7f` for an ASCII DEL (127).

`\uHHHH` Character code given by four hexadecimal digits. For example `\u0100` for a capital A with macron (U+0100).

`\UHHHHHH` Character code given by six hexadecimal digits. For example `\U010402`.

The following are examples of string literals:

```
"foo"
"bar plonk"
"Hello World"
"\"Hi\", he said."
```

The three escape sequences `\xHH`, `\uHHHH` and `\UHHHHHH` were chosen to not break compatibility with code written for previous versions of Guile. The R6RS specification suggests a different, incompatible syntax for hex escapes: `\xHHHH;` – a character code followed by one to eight hexadecimal digits terminated with a semicolon. If this escape format is desired instead, it can be enabled with the reader option `r6rs-hex-escapes`.

```
(read-enable 'r6rs-hex-escapes)
```

For more on reader options, See Section 6.18.2 [Scheme Read], page 385.

6.6.5.2 String Predicates

The following procedures can be used to check whether a given string fulfills some specified property.

`string?` *obj* [Scheme Procedure]
`scm_string_p` (*obj*) [C Function]
Return `#t` if *obj* is a string, else `#f`.

`int scm_is_string` (*SCM obj*) [C Function]
Returns `1` if *obj* is a string, `0` otherwise.

`string-null?` *str* [Scheme Procedure]
`scm_string_null_p` (*str*) [C Function]
Return `#t` if *str*'s length is zero, and `#f` otherwise.

```
(string-null? "")  ⇒ #t
y                  ⇒ "foo"
(string-null? y)   ⇒ #f
```

`string-any` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_any` (*char_pred, s, start, end*) [C Function]
Check if *char_pred* is true for any character in string *s*.

char_pred can be a character to check for any equal to that, or a character set (see Section 6.6.4 [Character Sets], page 134) to check for any in that set, or a predicate procedure to call.

For a procedure, calls (*char_pred* `c`) are made successively on the characters from *start* to *end*. If *char_pred* returns true (ie. non-`#f`), `string-any` stops and that return value is the return from `string-any`. The call on the last character (ie. at *end* − 1), if that point is reached, is a tail call.

If there are no characters in *s* (ie. *start* equals *end*) then the return is `#f`.

`string-every` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_every` (*char_pred, s, start, end*) [C Function]
Check if *char_pred* is true for every character in string *s*.

char_pred can be a character to check for every character equal to that, or a character set (see Section 6.6.4 [Character Sets], page 134) to check for every character being in that set, or a predicate procedure to call.

For a procedure, calls (*char_pred* `c`) are made successively on the characters from *start* to *end*. If *char_pred* returns `#f`, `string-every` stops and returns `#f`. The call

on the last character (ie. at *end* − 1), if that point is reached, is a tail call and the return from that call is the return from `string-every`.

If there are no characters in *s* (ie. *start* equals *end*) then the return is `#t`.

6.6.5.3 String Constructors

The string constructor procedures create new string objects, possibly initializing them with some specified character data. See also See Section 6.6.5.5 [String Selection], page 145, for ways to create strings from existing strings.

`string` *char...* [Scheme Procedure]

Return a newly allocated string made from the given character arguments.

```
(string #\x #\y #\z) ⇒ "xyz"
(string)             ⇒ ""
```

`list->string` *lst* [Scheme Procedure]
`scm_string` (*lst*) [C Function]

Return a newly allocated string made from a list of characters.

```
(list->string '(#\a #\b #\c)) ⇒ "abc"
```

`reverse-list->string` *lst* [Scheme Procedure]
`scm_reverse_list_to_string` (*lst*) [C Function]

Return a newly allocated string made from a list of characters, in reverse order.

```
(reverse-list->string '(#\a #\B #\c)) ⇒ "cBa"
```

`make-string` *k* [*chr*] [Scheme Procedure]
`scm_make_string` (*k*, *chr*) [C Function]

Return a newly allocated string of length *k*. If *chr* is given, then all elements of the string are initialized to *chr*, otherwise the contents of the string are unspecified.

`SCM scm_c_make_string` (*size_t len*, *SCM chr*) [C Function]

Like `scm_make_string`, but expects the length as a `size_t`.

`string-tabulate` *proc len* [Scheme Procedure]
`scm_string_tabulate` (*proc*, *len*) [C Function]

proc is an integer->char procedure. Construct a string of size *len* by applying *proc* to each index to produce the corresponding string element. The order in which *proc* is applied to the indices is not specified.

`string-join` *ls* [*delimiter* [*grammar*]] [Scheme Procedure]
`scm_string_join` (*ls*, *delimiter*, *grammar*) [C Function]

Append the string in the string list *ls*, using the string *delimiter* as a delimiter between the elements of *ls*. *delimiter* defaults to ' ', that is, strings in *ls* are appended with the space character in between them. *grammar* is a symbol which specifies how the delimiter is placed between the strings, and defaults to the symbol `infix`.

`infix` Insert the separator between list elements. An empty string will produce an empty list.

`strict-infix`
Like `infix`, but will raise an error if given the empty list.

`suffix` Insert the separator after every list element.

`prefix` Insert the separator before each list element.

6.6.5.4 List/String conversion

When processing strings, it is often convenient to first convert them into a list representation by using the procedure `string->list`, work with the resulting list, and then convert it back into a string. These procedures are useful for similar tasks.

`string->list` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_to_list` (*str*, *start*, *end*) [C Function]
`scm_string_to_list` (*str*) [C Function]
Convert the string *str* into a list of characters.

`string-split` *str char_pred* [Scheme Procedure]
`scm_string_split` (*str*, *char_pred*) [C Function]
Split the string *str* into a list of substrings delimited by appearances of characters that

- equal *char_pred*, if it is a character,
- satisfy the predicate *char_pred*, if it is a procedure,
- are in the set *char_pred*, if it is a character set.

Note that an empty substring between separator characters will result in an empty string in the result list.

```
(string-split "root:x:0:0:root:/root:/bin/bash" #\:)
⇒
("root" "x" "0" "0" "root" "/root" "/bin/bash")

(string-split "::" #\:)
⇒
("" "" "")

(string-split "" #\:)
⇒
("")
```

6.6.5.5 String Selection

Portions of strings can be extracted by these procedures. `string-ref` delivers individual characters whereas `substring` can be used to extract substrings from longer strings.

`string-length` *string* [Scheme Procedure]
`scm_string_length` (*string*) [C Function]
Return the number of characters in *string*.

`size_t scm_c_string_length` (*SCM str*) [C Function]
Return the number of characters in *str* as a `size_t`.

`string-ref` *str k* [Scheme Procedure]
`scm_string_ref` (*str*, *k*) [C Function]
Return character *k* of *str* using zero-origin indexing. *k* must be a valid index of *str*.

`SCM scm_c_string_ref` (*SCM str, size_t k*) [C Function]
Return character *k* of *str* using zero-origin indexing. *k* must be a valid index of *str*.

`string-copy` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_copy` (*str*, *start*, *end*) [C Function]
`scm_string_copy` (*str*) [C Function]
Return a copy of the given string *str*.

The returned string shares storage with *str* initially, but it is copied as soon as one of the two strings is modified.

`substring` *str start* [*end*] [Scheme Procedure]
`scm_substring` (*str*, *start*, *end*) [C Function]
Return a new string formed from the characters of *str* beginning with index *start* (inclusive) and ending with index *end* (exclusive). *str* must be a string, *start* and *end* must be exact integers satisfying:

0 <= *start* <= *end* <= `(string-length str)`.

The returned string shares storage with *str* initially, but it is copied as soon as one of the two strings is modified.

`substring/shared` *str start* [*end*] [Scheme Procedure]
`scm_substring_shared` (*str*, *start*, *end*) [C Function]
Like `substring`, but the strings continue to share their storage even if they are modified. Thus, modifications to *str* show up in the new string, and vice versa.

`substring/copy` *str start* [*end*] [Scheme Procedure]
`scm_substring_copy` (*str*, *start*, *end*) [C Function]
Like `substring`, but the storage for the new string is copied immediately.

`substring/read-only` *str start* [*end*] [Scheme Procedure]
`scm_substring_read_only` (*str*, *start*, *end*) [C Function]
Like `substring`, but the resulting string can not be modified.

`SCM scm_c_substring` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_shared` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_copy` (*SCM str, size_t start, size_t end*) [C Function]
`SCM scm_c_substring_read_only` (*SCM str, size_t start, size_t end*) [C Function]
Like `scm_substring`, etc. but the bounds are given as a `size_t`.

`string-take` *s n* [Scheme Procedure]
`scm_string_take` (*s*, *n*) [C Function]
Return the *n* first characters of *s*.

`string-drop` *s n* [Scheme Procedure]
`scm_string_drop` (*s*, *n*) [C Function]
Return all but the first *n* characters of *s*.

`string-take-right` *s n* [Scheme Procedure]
`scm_string_take_right` (*s*, *n*) [C Function]
Return the *n* last characters of *s*.

`string-drop-right` *s n* [Scheme Procedure]
`scm_string_drop_right` (*s*, *n*) [C Function]
Return all but the last *n* characters of *s*.

`string-pad` *s len* [*chr* [*start* [*end*]]] [Scheme Procedure]
`string-pad-right` *s len* [*chr* [*start* [*end*]]] [Scheme Procedure]
`scm_string_pad` (*s*, *len*, *chr*, *start*, *end*) [C Function]
`scm_string_pad_right` (*s*, *len*, *chr*, *start*, *end*) [C Function]
Take characters *start* to *end* from the string *s* and either pad with *chr* or truncate them to give *len* characters.

`string-pad` pads or truncates on the left, so for example

```
(string-pad "x" 3)     ⇒ "  x"
(string-pad "abcde" 3) ⇒ "cde"
```

`string-pad-right` pads or truncates on the right, so for example

```
(string-pad-right "x" 3)     ⇒ "x  "
(string-pad-right "abcde" 3) ⇒ "abc"
```

`string-trim` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`string-trim-right` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`string-trim-both` *s* [*char_pred* [*start* [*end*]]] [Scheme Procedure]
`scm_string_trim` (*s*, *char_pred*, *start*, *end*) [C Function]
`scm_string_trim_right` (*s*, *char_pred*, *start*, *end*) [C Function]
`scm_string_trim_both` (*s*, *char_pred*, *start*, *end*) [C Function]
Trim occurrences of *char_pred* from the ends of *s*.

`string-trim` trims *char_pred* characters from the left (start) of the string, `string-trim-right` trims them from the right (end) of the string, `string-trim-both` trims from both ends.

char_pred can be a character, a character set, or a predicate procedure to call on each character. If *char_pred* is not given the default is whitespace as per `char-set:whitespace` (see Section 6.6.4.6 [Standard Character Sets], page 139).

```
(string-trim " x ")              ⇒ "x "
(string-trim-right "banana" #\a) ⇒ "banan"
(string-trim-both ".,xy:;" char-set:punctuation)
                  ⇒ "xy"
(string-trim-both "xyzzy" (lambda (c)
                            (or (eqv? c #\x)
                                (eqv? c #\y))))
                  ⇒ "zz"
```

6.6.5.6 String Modification

These procedures are for modifying strings in-place. This means that the result of the operation is not a new string; instead, the original string's memory representation is modified.

`string-set!` *str k chr* [Scheme Procedure]
`scm_string_set_x` (*str*, *k*, *chr*) [C Function]
Store *chr* in element *k* of *str* and return an unspecified value. *k* must be a valid index of *str*.

`void scm_c_string_set_x` (*SCM str, size_t k, SCM chr*) [C Function]
Like `scm_string_set_x`, but the index is given as a `size_t`.

`string-fill!` *str chr* [*start* [*end*]] [Scheme Procedure]
`scm_substring_fill_x` (*str, chr, start, end*) [C Function]
`scm_string_fill_x` (*str, chr*) [C Function]
Stores *chr* in every element of the given *str* and returns an unspecified value.

`substring-fill!` *str start end fill* [Scheme Procedure]
`scm_substring_fill_x` (*str, start, end, fill*) [C Function]
Change every character in *str* between *start* and *end* to *fill*.

```
(define y (string-copy "abcdefg"))
(substring-fill! y 1 3 #\r)
y
⇒ "arrdefg"
```

`substring-move!` *str1 start1 end1 str2 start2* [Scheme Procedure]
`scm_substring_move_x` (*str1, start1, end1, str2, start2*) [C Function]
Copy the substring of *str1* bounded by *start1* and *end1* into *str2* beginning at position *start2*. *str1* and *str2* can be the same string.

`string-copy!` *target tstart s* [*start* [*end*]] [Scheme Procedure]
`scm_string_copy_x` (*target, tstart, s, start, end*) [C Function]
Copy the sequence of characters from index range [*start*, *end*) in string *s* to string *target*, beginning at index *tstart*. The characters are copied left-to-right or right-to-left as needed – the copy is guaranteed to work, even if *target* and *s* are the same string. It is an error if the copy operation runs off the end of the target string.

6.6.5.7 String Comparison

The procedures in this section are similar to the character ordering predicates (see Section 6.6.3 [Characters], page 129), but are defined on character sequences.

The first set is specified in R5RS and has names that end in `?`. The second set is specified in SRFI-13 and the names have not ending `?`.

The predicates ending in `-ci` ignore the character case when comparing strings. For now, case-insensitive comparison is done using the R5RS rules, where every lower-case character that has a single character upper-case form is converted to uppercase before comparison. See See Section 6.25.2 [Text Collation], page 466, for locale-dependent string comparison.

`string=?` *s1 s2 s3 ...* [Scheme Procedure]
Lexicographic equality predicate; return `#t` if all strings are the same length and contain the same characters in the same positions, otherwise return `#f`.

The procedure `string-ci=?` treats upper and lower case letters as though they were the same character, but `string=?` treats upper and lower case as distinct characters.

`string<?` *s1 s2 s3 ...* [Scheme Procedure]
Lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically less than *str_i+1*.

`string<=?` *s1 s2 s3 . . .* [Scheme Procedure]

Lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically less than or equal to *str_i+1*.

`string>?` *s1 s2 s3 . . .* [Scheme Procedure]

Lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically greater than *str_i+1*.

`string>=?` *s1 s2 s3 . . .* [Scheme Procedure]

Lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically greater than or equal to *str_i+1*.

`string-ci=?` *s1 s2 s3 . . .* [Scheme Procedure]

Case-insensitive string equality predicate; return `#t` if all strings are the same length and their component characters match (ignoring case) at each position; otherwise return `#f`.

`string-ci<?` *s1 s2 s3 . . .* [Scheme Procedure]

Case insensitive lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically less than *str_i+1* regardless of case.

`string-ci<=?` *s1 s2 s3 . . .* [Scheme Procedure]

Case insensitive lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically less than or equal to *str_i+1* regardless of case.

`string-ci>?` *s1 s2 s3 . . .* [Scheme Procedure]

Case insensitive lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically greater than *str_i+1* regardless of case.

`string-ci>=?` *s1 s2 s3 . . .* [Scheme Procedure]

Case insensitive lexicographic ordering predicate; return `#t` if, for every pair of consecutive string arguments *str_i* and *str_i+1*, *str_i* is lexicographically greater than or equal to *str_i+1* regardless of case.

`string-compare` *s1 s2 proc_lt proc_eq proc_gt* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]

`scm_string_compare` (*s1, s2, proc_lt, proc_eq, proc_gt, start1, end1, start2, end2*) [C Function]

Apply *proc_lt*, *proc_eq*, *proc_gt* to the mismatch index, depending upon whether *s1* is less than, equal to, or greater than *s2*. The mismatch index is the largest index *i* such that for every 0 <= *j* < *i*, *s1*[*j*] = *s2*[*j*] – that is, *i* is the first position that does not match.

`string-compare-ci` *s1 s2 proc_lt proc_eq proc_gt* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_compare_ci` (*s1, s2, proc_lt, proc_eq, proc_gt, start1, end1, start2, end2*) [C Function]

Apply *proc_lt*, *proc_eq*, *proc_gt* to the mismatch index, depending upon whether *s1* is less than, equal to, or greater than *s2*. The mismatch index is the largest index *i* such that for every 0 <= *j* < *i*, *s1*[*j*] = *s2*[*j*] – that is, *i* is the first position where the lowercased letters do not match.

`string=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_eq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are not equal, a true value otherwise.

`string<>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_neq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are equal, a true value otherwise.

`string<` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_lt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater or equal to *s2*, a true value otherwise.

`string>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_gt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less or equal to *s2*, a true value otherwise.

`string<=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_le` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater to *s2*, a true value otherwise.

`string>=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ge` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less to *s2*, a true value otherwise.

`string-ci=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_eq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are not equal, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_neq` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* and *s2* are equal, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_lt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is greater or equal to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci>` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_gt` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return `#f` if *s1* is less or equal to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci<=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_le` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return `#f` if *s1* is greater to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-ci>=` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_ci_ge` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return `#f` if *s1* is less to *s2*, a true value otherwise. The character comparison is done case-insensitively.

`string-hash` *s* [*bound* [*start* [*end*]]] [Scheme Procedure]
`scm_substring_hash` (*s, bound, start, end*) [C Function]
Compute a hash value for *s*. The optional argument *bound* is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

`string-hash-ci` *s* [*bound* [*start* [*end*]]] [Scheme Procedure]
`scm_substring_hash_ci` (*s, bound, start, end*) [C Function]
Compute a hash value for *s*. The optional argument *bound* is a non-negative exact integer specifying the range of the hash function. A positive value restricts the return value to the range [0,bound).

Because the same visual appearance of an abstract Unicode character can be obtained via multiple sequences of Unicode characters, even the case-insensitive string comparison functions described above may return `#f` when presented with strings containing different representations of the same character. For example, the Unicode character "LATIN SMALL LETTER S WITH DOT BELOW AND DOT ABOVE" can be represented with a single character (U+1E69) or by the character "LATIN SMALL LETTER S" (U+0073) followed by the combining marks "COMBINING DOT BELOW" (U+0323) and "COMBINING DOT ABOVE" (U+0307).

For this reason, it is often desirable to ensure that the strings to be compared are using a mutually consistent representation for every character. The Unicode standard defines two methods of normalizing the contents of strings: Decomposition, which breaks composite characters into a set of constituent characters with an ordering defined by the Unicode Standard; and composition, which performs the converse.

There are two decomposition operations. "Canonical decomposition" produces character sequences that share the same visual appearance as the original characters, while "compatibility decomposition" produces ones whose visual appearances may differ from the originals but which represent the same abstract character.

These operations are encapsulated in the following set of normalization forms:

NFD Characters are decomposed to their canonical forms.

NFKD Characters are decomposed to their compatibility forms.

NFC Characters are decomposed to their canonical forms, then composed.

NFKC Characters are decomposed to their compatibility forms, then composed.

The functions below put their arguments into one of the forms described above.

`string-normalize-nfd` *s* [Scheme Procedure]
`scm_string_normalize_nfd` (*s*) [C Function]
Return the `NFD` normalized form of *s*.

`string-normalize-nfkd` *s* [Scheme Procedure]
`scm_string_normalize_nfkd` (*s*) [C Function]
Return the `NFKD` normalized form of *s*.

`string-normalize-nfc` *s* [Scheme Procedure]
`scm_string_normalize_nfc` (*s*) [C Function]
Return the `NFC` normalized form of *s*.

`string-normalize-nfkc` *s* [Scheme Procedure]
`scm_string_normalize_nfkc` (*s*) [C Function]
Return the `NFKC` normalized form of *s*.

6.6.5.8 String Searching

`string-index` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_index` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from left to right, returning the index of the first occurrence of a character which

- equals *char_pred*, if it is character,
- satisfies the predicate *char_pred*, if it is a procedure,
- is in the set *char_pred*, if it is a character set.

Return `#f` if no match is found.

`string-rindex` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_rindex` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from right to left, returning the index of the last occurrence of a character which

- equals *char_pred*, if it is character,
- satisfies the predicate *char_pred*, if it is a procedure,
- is in the set if *char_pred* is a character set.

Return `#f` if no match is found.

`string-prefix-length` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_length` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common prefix of the two strings.

`string-prefix-length-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_length_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common prefix of the two strings, ignoring character case.

`string-suffix-length` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_length` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common suffix of the two strings.

`string-suffix-length-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_length_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]
Return the length of the longest common suffix of the two strings, ignoring character case.

`string-prefix?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_p` (*s1, s2, start1, end1, start2, end2*) [C Function]
Is *s1* a prefix of *s2*?

`string-prefix-ci?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_prefix_ci_p` (*s1, s2, start1, end1, start2, end2*) [C Function]
Is *s1* a prefix of *s2*, ignoring character case?

`string-suffix?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_p` (*s1, s2, start1, end1, start2, end2*) [C Function]
Is *s1* a suffix of *s2*?

`string-suffix-ci?` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_suffix_ci_p` (*s1, s2, start1, end1, start2, end2*) [C Function]
Is *s1* a suffix of *s2*, ignoring character case?

`string-index-right` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_index_right` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from right to left, returning the index of the last occurrence of a character which

- equals *char_pred*, if it is character,
- satisfies the predicate *char_pred*, if it is a procedure,
- is in the set if *char_pred* is a character set.

Return `#f` if no match is found.

`string-skip` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_skip` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from left to right, returning the index of the first occurrence of a character which

- does not equal *char_pred*, if it is character,
- does not satisfy the predicate *char_pred*, if it is a procedure,
- is not in the set if *char_pred* is a character set.

`string-skip-right` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_skip_right` (*s, char_pred, start, end*) [C Function]
Search through the string *s* from right to left, returning the index of the last occurrence of a character which

- does not equal *char_pred*, if it is character,
- does not satisfy the predicate *char_pred*, if it is a procedure,
- is not in the set if *char_pred* is a character set.

`string-count` *s char_pred* [*start* [*end*]] [Scheme Procedure]
`scm_string_count` (*s, char_pred, start, end*) [C Function]
Return the count of the number of characters in the string *s* which

- equals *char_pred*, if it is character,
- satisfies the predicate *char_pred*, if it is a procedure.
- is in the set *char_pred*, if it is a character set.

`string-contains` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_contains` (*s1, s2, start1, end1, start2, end2*) [C Function]
Does string *s1* contain string *s2*? Return the index in *s1* where *s2* occurs as a substring, or false. The optional start/end indices restrict the operation to the indicated substrings.

`string-contains-ci` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_contains_ci` (*s1, s2, start1, end1, start2, end2*) [C Function]
Does string *s1* contain string *s2*? Return the index in *s1* where *s2* occurs as a substring, or false. The optional start/end indices restrict the operation to the indicated substrings. Character comparison is done case-insensitively.

6.6.5.9 Alphabetic Case Mapping

These are procedures for mapping strings to their upper- or lower-case equivalents, respectively, or for capitalizing strings.

They use the basic case mapping rules for Unicode characters. No special language or context rules are considered. The resulting strings are guaranteed to be the same length as the input strings.

See Section 6.25.3 [Character Case Mapping], page 467, for locale-dependent case conversions.

`string-upcase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_upcase` (*str, start, end*) [C Function]
`scm_string_upcase` (*str*) [C Function]
Upcase every character in `str`.

`string-upcase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_upcase_x` (*str, start, end*) [C Function]
`scm_string_upcase_x` (*str*) [C Function]
Destructively upcase every character in `str`.

```
(string-upcase! y)
⇒ "ARRDEFG"
y
⇒ "ARRDEFG"
```

`string-downcase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_downcase` (*str, start, end*) [C Function]
`scm_string_downcase` (*str*) [C Function]
Downcase every character in *str*.

`string-downcase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_substring_downcase_x` (*str*, *start*, *end*) [C Function]
`scm_string_downcase_x` (*str*) [C Function]
Destructively downcase every character in *str*.

```
y
⇒ "ARRDEFG"
(string-downcase! y)
⇒ "arrdefg"
y
⇒ "arrdefg"
```

`string-capitalize` *str* [Scheme Procedure]
`scm_string_capitalize` (*str*) [C Function]
Return a freshly allocated string with the characters in *str*, where the first character of every word is capitalized.

`string-capitalize!` *str* [Scheme Procedure]
`scm_string_capitalize_x` (*str*) [C Function]
Upcase the first character of every word in *str* destructively and return *str*.

```
y                      ⇒ "hello world"
(string-capitalize! y) ⇒ "Hello World"
y                      ⇒ "Hello World"
```

`string-titlecase` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_titlecase` (*str*, *start*, *end*) [C Function]
Titlecase every first character in a word in *str*.

`string-titlecase!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_titlecase_x` (*str*, *start*, *end*) [C Function]
Destructively titlecase every first character in a word in *str*.

6.6.5.10 Reversing and Appending Strings

`string-reverse` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_reverse` (*str*, *start*, *end*) [C Function]
Reverse the string *str*. The optional arguments *start* and *end* delimit the region of *str* to operate on.

`string-reverse!` *str* [*start* [*end*]] [Scheme Procedure]
`scm_string_reverse_x` (*str*, *start*, *end*) [C Function]
Reverse the string *str* in-place. The optional arguments *start* and *end* delimit the region of *str* to operate on. The return value is unspecified.

`string-append` *arg* ... [Scheme Procedure]
`scm_string_append` (*args*) [C Function]
Return a newly allocated string whose characters form the concatenation of the given strings, *arg* ....

```
(let ((h "hello "))
  (string-append h "world"))
⇒ "hello world"
```

`string-append/shared` *arg ...* [Scheme Procedure]
`scm_string_append_shared` (*args*) [C Function]
Like `string-append`, but the result may share memory with the argument strings.

`string-concatenate` *ls* [Scheme Procedure]
`scm_string_concatenate` (*ls*) [C Function]
Append the elements (which must be strings) of *ls* together into a single string. Guaranteed to return a freshly allocated string.

`string-concatenate-reverse` *ls* [*final_string* [*end*]] [Scheme Procedure]
`scm_string_concatenate_reverse` (*ls, final_string, end*) [C Function]
Without optional arguments, this procedure is equivalent to

```
(string-concatenate (reverse ls))
```

If the optional argument *final_string* is specified, it is consed onto the beginning to *ls* before performing the list-reverse and string-concatenate operations. If *end* is given, only the characters of *final_string* up to index *end* are used.

Guaranteed to return a freshly allocated string.

`string-concatenate/shared` *ls* [Scheme Procedure]
`scm_string_concatenate_shared` (*ls*) [C Function]
Like `string-concatenate`, but the result may share memory with the strings in the list *ls*.

`string-concatenate-reverse/shared` *ls* [*final_string* [*end*]] [Scheme Procedure]
`scm_string_concatenate_reverse_shared` (*ls, final_string, end*) [C Function]
Like `string-concatenate-reverse`, but the result may share memory with the strings in the *ls* arguments.

6.6.5.11 Mapping, Folding, and Unfolding

`string-map` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_map` (*proc, s, start, end*) [C Function]
proc is a char->char procedure, it is mapped over *s*. The order in which the procedure is applied to the string elements is not specified.

`string-map!` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_map_x` (*proc, s, start, end*) [C Function]
proc is a char->char procedure, it is mapped over *s*. The order in which the procedure is applied to the string elements is not specified. The string *s* is modified in-place, the return value is not specified.

`string-for-each` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_for_each` (*proc, s, start, end*) [C Function]
proc is mapped over *s* in left-to-right order. The return value is not specified.

`string-for-each-index` *proc s* [*start* [*end*]] [Scheme Procedure]
`scm_string_for_each_index` (*proc, s, start, end*) [C Function]
Call (*proc* `i`) for each index i in *s*, from left to right.

For example, to change characters to alternately upper and lower case,

```
(define str (string-copy "studly"))
(string-for-each-index
    (lambda (i)
      (string-set! str i
        ((if (even? i) char-upcase char-downcase)
         (string-ref str i))))
    str)
str ⇒ "StUdLy"
```

`string-fold` *kons knil s* [*start* [*end*]] [Scheme Procedure]
`scm_string_fold` (*kons*, *knil*, *s*, *start*, *end*) [C Function]

Fold *kons* over the characters of *s*, with *knil* as the terminating element, from left to right. *kons* must expect two arguments: The actual character and the last result of *kons*' application.

`string-fold-right` *kons knil s* [*start* [*end*]] [Scheme Procedure]
`scm_string_fold_right` (*kons*, *knil*, *s*, *start*, *end*) [C Function]

Fold *kons* over the characters of *s*, with *knil* as the terminating element, from right to left. *kons* must expect two arguments: The actual character and the last result of *kons*' application.

`string-unfold` *p f g seed* [*base* [*make_final*]] [Scheme Procedure]
`scm_string_unfold` (*p*, *f*, *g*, *seed*, *base*, *make_final*) [C Function]

- *g* is used to generate a series of *seed* values from the initial *seed*: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of these seed values.
- *f* maps each seed value to the corresponding character in the result string. These chars are assembled into the string in a left-to-right order.
- *base* is the optional initial/leftmost portion of the constructed string; it default to the empty string.
- *make_final* is applied to the terminal seed value (on which *p* returns true) to produce the final/rightmost portion of the constructed string. The default is nothing extra.

`string-unfold-right` *p f g seed* [*base* [*make_final*]] [Scheme Procedure]
`scm_string_unfold_right` (*p*, *f*, *g*, *seed*, *base*, *make_final*) [C Function]

- *g* is used to generate a series of *seed* values from the initial *seed*: *seed*, (*g seed*), (*g*^2 *seed*), (*g*^3 *seed*), ...
- *p* tells us when to stop – when it returns true when applied to one of these seed values.
- *f* maps each seed value to the corresponding character in the result string. These chars are assembled into the string in a right-to-left order.
- *base* is the optional initial/rightmost portion of the constructed string; it default to the empty string.

- *make_final* is applied to the terminal seed value (on which *p* returns true) to produce the final/leftmost portion of the constructed string. It defaults to `(lambda (x) )`.

6.6.5.12 Miscellaneous String Operations

`xsubstring` *s from* [*to* [*start* [*end*]]] [Scheme Procedure]
`scm_xsubstring` (*s, from, to, start, end*) [C Function]

This is the *extended substring* procedure that implements replicated copying of a substring of some string.

s is a string, *start* and *end* are optional arguments that demarcate a substring of *s*, defaulting to 0 and the length of *s*. Replicate this substring up and down index space, in both the positive and negative directions. `xsubstring` returns the substring of this string beginning at index *from*, and ending at *to*, which defaults to *from* + (*end* - *start*).

`string-xcopy!` *target tstart s sfrom* [*sto* [*start* [*end*]]] [Scheme Procedure]
`scm_string_xcopy_x` (*target, tstart, s, sfrom, sto, start, end*) [C Function]

Exactly the same as `xsubstring`, but the extracted text is written into the string *target* starting at index *tstart*. The operation is not defined if `(eq? target s)` or these arguments share storage – you cannot copy a string on top of itself.

`string-replace` *s1 s2* [*start1* [*end1* [*start2* [*end2*]]]] [Scheme Procedure]
`scm_string_replace` (*s1, s2, start1, end1, start2, end2*) [C Function]

Return the string *s1*, but with the characters *start1* ... *end1* replaced by the characters *start2* ... *end2* from *s2*.

`string-tokenize` *s* [*token_set* [*start* [*end*]]] [Scheme Procedure]
`scm_string_tokenize` (*s, token_set, start, end*) [C Function]

Split the string *s* into a list of substrings, where each substring is a maximal non-empty contiguous sequence of characters from the character set *token_set*, which defaults to `char-set:graphic`. If *start* or *end* indices are provided, they restrict `string-tokenize` to operating on the indicated substring of *s*.

`string-filter` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_filter` (*char_pred, s, start, end*) [C Function]

Filter the string *s*, retaining only those characters which satisfy *char_pred*.

If *char_pred* is a procedure, it is applied to each character as a predicate, if it is a character, it is tested for equality and if it is a character set, it is tested for membership.

`string-delete` *char_pred s* [*start* [*end*]] [Scheme Procedure]
`scm_string_delete` (*char_pred, s, start, end*) [C Function]

Delete characters satisfying *char_pred* from *s*.

If *char_pred* is a procedure, it is applied to each character as a predicate, if it is a character, it is tested for equality and if it is a character set, it is tested for membership.

The following additional functions are available in the module `(ice-9 string-fun)`. They can be used with:

```
(use-modules (ice-9 string-fun))
```

`string-replace-substring` *str substring replacement* [Scheme Procedure]
Return a new string where every instance of *substring* in string *str* has been replaced by *replacement*. For example:

```
(string-replace-substring "a ring of strings" "ring" "rut")
⇒ "a rut of struts"
```

6.6.5.13 Representing Strings as Bytes

Out in the cold world outside of Guile, not all strings are treated in the same way. Out there there are only bytes, and there are many ways of representing a strings (sequences of characters) as binary data (sequences of bytes).

As a user, usually you don't have to think about this very much. When you type on your keyboard, your system encodes your keystrokes as bytes according to the locale that you have configured on your computer. Guile uses the locale to decode those bytes back into characters – hopefully the same characters that you typed in.

All is not so clear when dealing with a system with multiple users, such as a web server. Your web server might get a request from one user for data encoded in the ISO-8859-1 character set, and then another request from a different user for UTF-8 data.

Guile provides an *iconv* module for converting between strings and sequences of bytes. See Section 6.6.12 [Bytevectors], page 193, for more on how Guile represents raw byte sequences. This module gets its name from the common UNIX command of the same name.

Note that often it is sufficient to just read and write strings from ports instead of using these functions. To do this, specify the port encoding using `set-port-encoding!`. See Section 6.14.1 [Ports], page 331, for more on ports and character encodings.

Unlike the rest of the procedures in this section, you have to load the `iconv` module before having access to these procedures:

```
(use-modules (ice-9 iconv))
```

`string->bytevector` *string encoding* [*conversion-strategy*] [Scheme Procedure]
Encode *string* as a sequence of bytes.

The string will be encoded in the character set specified by the *encoding* string. If the string has characters that cannot be represented in the encoding, by default this procedure raises an `encoding-error`. Pass a *conversion-strategy* argument to specify other behaviors.

The return value is a bytevector. See Section 6.6.12 [Bytevectors], page 193, for more on bytevectors. See Section 6.14.1 [Ports], page 331, for more on character encodings and conversion strategies.

`bytevector->string` *bytevector encoding* [*conversion-strategy*] [Scheme Procedure]
Decode *bytevector* into a string.

The bytes will be decoded from the character set by the *encoding* string. If the bytes do not form a valid encoding, by default this procedure raises an `decoding-error`.

As with `string->bytevector`, pass the optional *conversion-strategy* argument to modify this behavior. See Section 6.14.1 [Ports], page 331, for more on character encodings and conversion strategies.

`call-with-output-encoded-string` *encoding proc* [*conversion-strategy*] [Scheme Procedure]

Like `call-with-output-string`, but instead of returning a string, returns a encoding of the string according to *encoding*, as a bytevector. This procedure can be more efficient than collecting a string and then converting it via `string->bytevector`.

6.6.5.14 Conversion to/from C

When creating a Scheme string from a C string or when converting a Scheme string to a C string, the concept of character encoding becomes important.

In C, a string is just a sequence of bytes, and the character encoding describes the relation between these bytes and the actual characters that make up the string. For Scheme strings, character encoding is not an issue (most of the time), since in Scheme you usually treat strings as character sequences, not byte sequences.

Converting to C and converting from C each have their own challenges.

When converting from C to Scheme, it is important that the sequence of bytes in the C string be valid with respect to its encoding. ASCII strings, for example, can't have any bytes greater than 127. An ASCII byte greater than 127 is considered *ill-formed* and cannot be converted into a Scheme character.

Problems can occur in the reverse operation as well. Not all character encodings can hold all possible Scheme characters. Some encodings, like ASCII for example, can only describe a small subset of all possible characters. So, when converting to C, one must first decide what to do with Scheme characters that can't be represented in the C string.

Converting a Scheme string to a C string will often allocate fresh memory to hold the result. You must take care that this memory is properly freed eventually. In many cases, this can be achieved by using `scm_dynwind_free` inside an appropriate dynwind context, See Section 6.13.10 [Dynamic Wind], page 320.

`SCM scm_from_locale_string` (*const char *str*) [C Function]
`SCM scm_from_locale_stringn` (*const char *str, size_t len*) [C Function]

Creates a new Scheme string that has the same contents as *str* when interpreted in the character encoding of the current locale.

For `scm_from_locale_string`, *str* must be null-terminated.

For `scm_from_locale_stringn`, *len* specifies the length of *str* in bytes, and *str* does not need to be null-terminated. If *len* is `(size_t)-1`, then *str* does need to be null-terminated and the real length will be found with `strlen`.

If the C string is ill-formed, an error will be raised.

Note that these functions should *not* be used to convert C string constants, because there is no guarantee that the current locale will match that of the execution character set, used for string and character constants. Most modern C compilers use UTF-8 by default, so to convert C string constants we recommend `scm_from_utf8_string`.

`SCM scm_take_locale_string` (*char *str*) [C Function]
`SCM scm_take_locale_stringn` (*char *str, size_t len*) [C Function]
Like `scm_from_locale_string` and `scm_from_locale_stringn`, respectively, but also frees *str* with `free` eventually. Thus, you can use this function when you would free *str* anyway immediately after creating the Scheme string. In certain cases, Guile can then use *str* directly as its internal representation.

`char * scm_to_locale_string` (*SCM str*) [C Function]
`char * scm_to_locale_stringn` (*SCM str, size_t *lenp*) [C Function]
Returns a C string with the same contents as *str* in the character encoding of the current locale. The C string must be freed with `free` eventually, maybe by using `scm_dynwind_free`, See Section 6.13.10 [Dynamic Wind], page 320.

For `scm_to_locale_string`, the returned string is null-terminated and an error is signalled when *str* contains `#\nul` characters.

For `scm_to_locale_stringn` and *lenp* not `NULL`, *str* might contain `#\nul` characters and the length of the returned string in bytes is stored in `*lenp`. The returned string will not be null-terminated in this case. If *lenp* is `NULL`, `scm_to_locale_stringn` behaves like `scm_to_locale_string`.

If a character in *str* cannot be represented in the character encoding of the current locale, the default port conversion strategy is used. See Section 6.14.1 [Ports], page 331, for more on conversion strategies.

If the conversion strategy is `error`, an error will be raised. If it is `substitute`, a replacement character, such as a question mark, will be inserted in its place. If it is `escape`, a hex escape will be inserted in its place.

`size_t scm_to_locale_stringbuf` (*SCM str, char *buf, size_t max_len*) [C Function]
Puts *str* as a C string in the current locale encoding into the memory pointed to by *buf*. The buffer at *buf* has room for *max_len* bytes and `scm_to_local_stringbuf` will never store more than that. No terminating `'\0'` will be stored.

The return value of `scm_to_locale_stringbuf` is the number of bytes that are needed for all of *str*, regardless of whether *buf* was large enough to hold them. Thus, when the return value is larger than *max_len*, only *max_len* bytes have been stored and you probably need to try again with a larger buffer.

For most situations, string conversion should occur using the current locale, such as with the functions above. But there may be cases where one wants to convert strings from a character encoding other than the locale's character encoding. For these cases, the lower-level functions `scm_to_stringn` and `scm_from_stringn` are provided. These functions should seldom be necessary if one is properly using locales.

`scm_t_string_failed_conversion_handler` [C Type]
This is an enumerated type that can take one of three values: `SCM_FAILED_CONVERSION_ERROR`, `SCM_FAILED_CONVERSION_QUESTION_MARK`, and `SCM_FAILED_CONVERSION_ESCAPE_SEQUENCE`. They are used to indicate a strategy for handling characters that cannot be converted to or from a given character encoding. `SCM_FAILED_CONVERSION_ERROR` indicates that a conversion should

throw an error if some characters cannot be converted. `SCM_FAILED_CONVERSION_QUESTION_MARK` indicates that a conversion should replace unconvertable characters with the question mark character. And, `SCM_FAILED_CONVERSION_ESCAPE_SEQUENCE` requests that a conversion should replace an unconvertable character with an escape sequence.

While all three strategies apply when converting Scheme strings to C, only `SCM_FAILED_CONVERSION_ERROR` and `SCM_FAILED_CONVERSION_QUESTION_MARK` can be used when converting C strings to Scheme.

`char *scm_to_stringn` (*SCM str, size_t *lenp, const char *encoding, scm_t_string_failed_conversion_handler handler*) [C Function]

This function returns a newly allocated C string from the Guile string *str*. The length of the returned string in bytes will be returned in *lenp*. The character encoding of the C string is passed as the ASCII, null-terminated C string *encoding*. The *handler* parameter gives a strategy for dealing with characters that cannot be converted into *encoding*.

If *lenp* is `NULL`, this function will return a null-terminated C string. It will throw an error if the string contains a null character.

The Scheme interface to this function is `string->bytevector`, from the `ice-9 iconv` module. See Section 6.6.5.13 [Representing Strings as Bytes], page 159.

`SCM scm_from_stringn` (*const char *str, size_t len, const char *encoding, scm_t_string_failed_conversion_handler handler*) [C Function]

This function returns a scheme string from the C string *str*. The length in bytes of the C string is input as *len*. The encoding of the C string is passed as the ASCII, null-terminated C string `encoding`. The *handler* parameters suggests a strategy for dealing with unconvertable characters.

The Scheme interface to this function is `bytevector->string`. See Section 6.6.5.13 [Representing Strings as Bytes], page 159.

The following conversion functions are provided as a convenience for the most commonly used encodings.

`SCM scm_from_latin1_string` (*const char *str*) [C Function]
`SCM scm_from_utf8_string` (*const char *str*) [C Function]
`SCM scm_from_utf32_string` (*const scm_t_wchar *str*) [C Function]

Return a scheme string from the null-terminated C string *str*, which is ISO-8859-1-, UTF-8-, or UTF-32-encoded. These functions should be used to convert hard-coded C string constants into Scheme strings.

`SCM scm_from_latin1_stringn` (*const char *str, size_t len*) [C Function]
`SCM scm_from_utf8_stringn` (*const char *str, size_t len*) [C Function]
`SCM scm_from_utf32_stringn` (*const scm_t_wchar *str, size_t len*) [C Function]

Return a scheme string from C string *str*, which is ISO-8859-1-, UTF-8-, or UTF-32-encoded, of length *len*. *len* is the number of bytes pointed to by *str* for `scm_from_latin1_stringn` and `scm_from_utf8_stringn`; it is the number of elements (code points) in *str* in the case of `scm_from_utf32_stringn`.

`char *scm_to_latin1_stringn` (*SCM str, size_t *lenp*) [C function]
`char *scm_to_utf8_stringn` (*SCM str, size_t *lenp*) [C function]
`scm_t_wchar *scm_to_utf32_stringn` (*SCM str, size_t *lenp*) [C function]
Return a newly allocated, ISO-8859-1-, UTF-8-, or UTF-32-encoded C string from Scheme string *str*. An error is thrown when *str* cannot be converted to the specified encoding. If *lenp* is `NULL`, the returned C string will be null terminated, and an error will be thrown if the C string would otherwise contain null characters. If *lenp* is not `NULL`, the string is not null terminated, and the length of the returned string is returned in *lenp*. The length returned is the number of bytes for `scm_to_latin1_stringn` and `scm_to_utf8_stringn`; it is the number of elements (code points) for `scm_to_utf32_stringn`.

It is not often the case, but sometimes when you are dealing with the implementation details of a port, you need to encode and decode strings according to the encoding and conversion strategy of the port. There are some convenience functions for that purpose as well.

`SCM scm_from_port_string` (*const char *str, SCM port*) [C Function]
`SCM scm_from_port_stringn` (*const char *str, size_t len, SCM port*) [C Function]
`char* scm_to_port_string` (*SCM str, SCM port*) [C Function]
`char* scm_to_port_stringn` (*SCM str, size_t *lenp, SCM port*) [C Function]
Like `scm_from_stringn` and friends, except they take their encoding and conversion strategy from a given port object.

6.6.5.15 String Internals

Guile stores each string in memory as a contiguous array of Unicode code points along with an associated set of attributes. If all of the code points of a string have an integer range between 0 and 255 inclusive, the code point array is stored as one byte per code point: it is stored as an ISO-8859-1 (aka Latin-1) string. If any of the code points of the string has an integer value greater that 255, the code point array is stored as four bytes per code point: it is stored as a UTF-32 string.

Conversion between the one-byte-per-code-point and four-bytes-per-code-point representations happens automatically as necessary.

No API is provided to set the internal representation of strings; however, there are pair of procedures available to query it. These are debugging procedures. Using them in production code is discouraged, since the details of Guile's internal representation of strings may change from release to release.

`string-bytes-per-char` *str* [Scheme Procedure]
`scm_string_bytes_per_char` (*str*) [C Function]
Return the number of bytes used to encode a Unicode code point in string *str*. The result is one or four.

`%string-dump` *str* [Scheme Procedure]
`scm_sys_string_dump` (*str*) [C Function]
Returns an association list containing debugging information for *str*. The association list has the following entries.

`string` The string itself.

`start` The start index of the string into its stringbuf

`length` The length of the string

`shared` If this string is a substring, it returns its parent string. Otherwise, it returns `#f`

`read-only`
`#t` if the string is read-only

`stringbuf-chars`
A new string containing this string's stringbuf's characters

`stringbuf-length`
The number of characters in this stringbuf

`stringbuf-shared`
`#t` if this stringbuf is shared

`stringbuf-wide`
`#t` if this stringbuf's characters are stored in a 32-bit buffer, or `#f` if they are stored in an 8-bit buffer

6.6.6 Symbols

Symbols in Scheme are widely used in three ways: as items of discrete data, as lookup keys for alists and hash tables, and to denote variable references.

A *symbol* is similar to a string in that it is defined by a sequence of characters. The sequence of characters is known as the symbol's *name*. In the usual case — that is, where the symbol's name doesn't include any characters that could be confused with other elements of Scheme syntax — a symbol is written in a Scheme program by writing the sequence of characters that make up the name, *without* any quotation marks or other special syntax. For example, the symbol whose name is "multiply-by-2" is written, simply:

```
multiply-by-2
```

Notice how this differs from a *string* with contents "multiply-by-2", which is written with double quotation marks, like this:

```
"multiply-by-2"
```

Looking beyond how they are written, symbols are different from strings in two important respects.

The first important difference is uniqueness. If the same-looking string is read twice from two different places in a program, the result is two *different* string objects whose contents just happen to be the same. If, on the other hand, the same-looking symbol is read twice from two different places in a program, the result is the *same* symbol object both times.

Given two read symbols, you can use `eq?` to test whether they are the same (that is, have the same name). `eq?` is the most efficient comparison operator in Scheme, and comparing two symbols like this is as fast as comparing, for example, two numbers. Given two strings, on the other hand, you must use `equal?` or `string=?`, which are much slower comparison operators, to determine whether the strings have the same contents.

```
(define sym1 (quote hello))
```

```
(define sym2 (quote hello))
(eq? sym1 sym2) ⇒ #t

(define str1 "hello")
(define str2 "hello")
(eq? str1 str2) ⇒ #f
(equal? str1 str2) ⇒ #t
```

The second important difference is that symbols, unlike strings, are not self-evaluating. This is why we need the `(quote ...)`s in the example above: `(quote hello)` evaluates to the symbol named `"hello"` itself, whereas an unquoted `hello` is *read* as the symbol named `"hello"` and evaluated as a variable reference . . . about which more below (see Section 6.6.6.3 [Symbol Variables], page 167).

6.6.6.1 Symbols as Discrete Data

Numbers and symbols are similar to the extent that they both lend themselves to `eq?` comparison. But symbols are more descriptive than numbers, because a symbol's name can be used directly to describe the concept for which that symbol stands.

For example, imagine that you need to represent some colours in a computer program. Using numbers, you would have to choose arbitrarily some mapping between numbers and colours, and then take care to use that mapping consistently:

```
;; 1=red, 2=green, 3=purple

(if (eq? (colour-of vehicle) 1)
    ...)
```

You can make the mapping more explicit and the code more readable by defining constants:

```
(define red 1)
(define green 2)
(define purple 3)

(if (eq? (colour-of vehicle) red)
    ...)
```

But the simplest and clearest approach is not to use numbers at all, but symbols whose names specify the colours that they refer to:

```
(if (eq? (colour-of vehicle) 'red)
    ...)
```

The descriptive advantages of symbols over numbers increase as the set of concepts that you want to describe grows. Suppose that a car object can have other properties as well, such as whether it has or uses:

- automatic or manual transmission
- leaded or unleaded fuel
- power steering (or not).

Then a car's combined property set could be naturally represented and manipulated as a list of symbols:

```
(properties-of vehicle1)
```

```
⇒
(red manual unleaded power-steering)

(if (memq 'power-steering (properties-of vehicle1))
    (display "Unfit people can drive this vehicle.\n")
    (display "You'll need strong arms to drive this vehicle!\n"))
⊣
Unfit people can drive this vehicle.
```

Remember, the fundamental property of symbols that we are relying on here is that an occurrence of `'red` in one part of a program is an *indistinguishable* symbol from an occurrence of `'red` in another part of a program; this means that symbols can usefully be compared using `eq?`. At the same time, symbols have naturally descriptive names. This combination of efficiency and descriptive power makes them ideal for use as discrete data.

6.6.6.2 Symbols as Lookup Keys

Given their efficiency and descriptive power, it is natural to use symbols as the keys in an association list or hash table.

To illustrate this, consider a more structured representation of the car properties example from the preceding subsection. Rather than mixing all the properties up together in a flat list, we could use an association list like this:

```
(define car1-properties '((colour . red)
                          (transmission . manual)
                          (fuel . unleaded)
                          (steering . power-assisted)))
```

Notice how this structure is more explicit and extensible than the flat list. For example it makes clear that `manual` refers to the transmission rather than, say, the windows or the locking of the car. It also allows further properties to use the same symbols among their possible values without becoming ambiguous:

```
(define car1-properties '((colour . red)
                          (transmission . manual)
                          (fuel . unleaded)
                          (steering . power-assisted)
                          (seat-colour . red)
                          (locking . manual)))
```

With a representation like this, it is easy to use the efficient `assq-XXX` family of procedures (see Section 6.6.20 [Association Lists], page 230) to extract or change individual pieces of information:

```
(assq-ref car1-properties 'fuel) ⇒ unleaded
(assq-ref car1-properties 'transmission) ⇒ manual

(assq-set! car1-properties 'seat-colour 'black)
⇒
((colour . red)
 (transmission . manual)
 (fuel . unleaded)
```

```
(steering . power-assisted)
(seat-colour . black)
(locking . manual)))
```

Hash tables also have keys, and exactly the same arguments apply to the use of symbols in hash tables as in association lists. The hash value that Guile uses to decide where to add a symbol-keyed entry to a hash table can be obtained by calling the `symbol-hash` procedure:

`symbol-hash` *symbol* [Scheme Procedure]
`scm_symbol_hash` (*symbol*) [C Function]
Return a hash value for *symbol*.

See Section 6.6.22 [Hash Tables], page 238, for information about hash tables in general, and for why you might choose to use a hash table rather than an association list.

6.6.6.3 Symbols as Denoting Variables

When an unquoted symbol in a Scheme program is evaluated, it is interpreted as a variable reference, and the result of the evaluation is the appropriate variable's value.

For example, when the expression `(string-length "abcd")` is read and evaluated, the sequence of characters `string-length` is read as the symbol whose name is "string-length". This symbol is associated with a variable whose value is the procedure that implements string length calculation. Therefore evaluation of the `string-length` symbol results in that procedure.

The details of the connection between an unquoted symbol and the variable to which it refers are explained elsewhere. See Section 6.12 [Binding Constructs], page 293, for how associations between symbols and variables are created, and Section 6.20 [Modules], page 410, for how those associations are affected by Guile's module system.

6.6.6.4 Operations Related to Symbols

Given any Scheme value, you can determine whether it is a symbol using the `symbol?` primitive:

`symbol?` *obj* [Scheme Procedure]
`scm_symbol_p` (*obj*) [C Function]
Return `#t` if *obj* is a symbol, otherwise return `#f`.

`int scm_is_symbol` (*SCM val*) [C Function]
Equivalent to `scm_is_true (scm_symbol_p (val))`.

Once you know that you have a symbol, you can obtain its name as a string by calling `symbol->string`. Note that Guile differs by default from R5RS on the details of `symbol->string` as regards case-sensitivity:

`symbol->string` *s* [Scheme Procedure]
`scm_symbol_to_string` (*s*) [C Function]
Return the name of symbol *s* as a string. By default, Guile reads symbols case-sensitively, so the string returned will have the same case variation as the sequence of characters that caused *s* to be created.

If Guile is set to read symbols case-insensitively (as specified by R5RS), and *s* comes into being as part of a literal expression (see Section "Literal expressions" in *The Revised^5 Report on Scheme*) or by a call to the `read` or `string-ci->symbol` procedures, Guile converts any alphabetic characters in the symbol's name to lower case before creating the symbol object, so the string returned here will be in lower case.

If *s* was created by `string->symbol`, the case of characters in the string returned will be the same as that in the string that was passed to `string->symbol`, regardless of Guile's case-sensitivity setting at the time *s* was created.

It is an error to apply mutation procedures like `string-set!` to strings returned by this procedure.

Most symbols are created by writing them literally in code. However it is also possible to create symbols programmatically using the following procedures:

`symbol` *char...* [Scheme Procedure]

Return a newly allocated symbol made from the given character arguments.

```
(symbol #\x #\y #\z) ⇒ xyz
```

`list->symbol` *lst* [Scheme Procedure]

Return a newly allocated symbol made from a list of characters.

```
(list->symbol '(#\a #\b #\c)) ⇒ abc
```

`symbol-append` *arg ...* [Scheme Procedure]

Return a newly allocated symbol whose characters form the concatenation of the given symbols, *arg ...*.

```
(let ((h 'hello))
  (symbol-append h 'world))
⇒ helloworld
```

`string->symbol` *string* [Scheme Procedure]
`scm_string_to_symbol` (*string*) [C Function]

Return the symbol whose name is *string*. This procedure can create symbols with names containing special characters or letters in the non-standard case, but it is usually a bad idea to create such symbols because in some implementations of Scheme they cannot be read as themselves.

`string-ci->symbol` *str* [Scheme Procedure]
`scm_string_ci_to_symbol` (*str*) [C Function]

Return the symbol whose name is *str*. If Guile is currently reading symbols case-insensitively, *str* is converted to lowercase before the returned symbol is looked up or created.

The following examples illustrate Guile's detailed behaviour as regards the case-sensitivity of symbols:

```
(read-enable 'case-insensitive)   ; R5RS compliant behaviour

(symbol->string 'flying-fish)     ⇒ "flying-fish"
(symbol->string 'Martin)          ⇒ "martin"
```

```
(symbol->string
   (string->symbol "Malvina"))   ⇒ "Malvina"

(eq? 'mISSISSIppi 'mississippi)  ⇒ #t
(string->symbol "mISSISSIppi")   ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒ #f
(eq? 'LolliPop
  (string->symbol (symbol->string 'LolliPop))) ⇒ #t
(string=? "K. Harper, M.D."
  (symbol->string
    (string->symbol "K. Harper, M.D."))) ⇒ #t

(read-disable 'case-insensitive)   ; Guile default behaviour

(symbol->string 'flying-fish)    ⇒ "flying-fish"
(symbol->string 'Martin)         ⇒ "Martin"
(symbol->string
   (string->symbol "Malvina"))   ⇒ "Malvina"

(eq? 'mISSISSIppi 'mississippi)  ⇒ #f
(string->symbol "mISSISSIppi")   ⇒ mISSISSIppi
(eq? 'bitBlt (string->symbol "bitBlt")) ⇒ #t
(eq? 'LolliPop
  (string->symbol (symbol->string 'LolliPop))) ⇒ #t
(string=? "K. Harper, M.D."
  (symbol->string
    (string->symbol "K. Harper, M.D."))) ⇒ #t
```

From C, there are lower level functions that construct a Scheme symbol from a C string in the current locale encoding.

When you want to do more from C, you should convert between symbols and strings using `scm_symbol_to_string` and `scm_string_to_symbol` and work with the strings.

`SCM scm_from_latin1_symbol` (*const char *name*) [C Function]
`SCM scm_from_utf8_symbol` (*const char *name*) [C Function]

Construct and return a Scheme symbol whose name is specified by the null-terminated C string *name*. These are appropriate when the C string is hard-coded in the source code.

`SCM scm_from_locale_symbol` (*const char *name*) [C Function]
`SCM scm_from_locale_symboln` (*const char *name, size_t len*) [C Function]

Construct and return a Scheme symbol whose name is specified by *name*. For `scm_from_locale_symbol`, *name* must be null terminated; for `scm_from_locale_symboln` the length of *name* is specified explicitly by *len*.

Note that these functions should *not* be used when *name* is a C string constant, because there is no guarantee that the current locale will match that of the execution character set, used for string and character constants. Most modern C compilers use UTF-8 by default, so in such cases we recommend `scm_from_utf8_symbol`.

`SCM scm_take_locale_symbol` (*char *str*) [C Function]
`SCM scm_take_locale_symboln` (*char *str, size_t len*) [C Function]
Like `scm_from_locale_symbol` and `scm_from_locale_symboln`, respectively, but also frees *str* with `free` eventually. Thus, you can use this function when you would free *str* anyway immediately after creating the Scheme string. In certain cases, Guile can then use *str* directly as its internal representation.

The size of a symbol can also be obtained from C:

`size_t scm_c_symbol_length` (*SCM sym*) [C Function]
Return the number of characters in *sym*.

Finally, some applications, especially those that generate new Scheme code dynamically, need to generate symbols for use in the generated code. The `gensym` primitive meets this need:

`gensym` [*prefix*] [Scheme Procedure]
`scm_gensym` (*prefix*) [C Function]
Create a new symbol with a name constructed from a prefix and a counter value. The string *prefix* can be specified as an optional argument. Default prefix is '`g`'. The counter is increased by 1 at each call. There is no provision for resetting the counter.

The symbols generated by `gensym` are *likely* to be unique, since their names begin with a space and it is only otherwise possible to generate such symbols if a programmer goes out of their way to do so. Uniqueness can be guaranteed by instead using uninterned symbols (see Section 6.6.6.7 [Symbol Uninterned], page 172), though they can't be usefully written out and read back in.

6.6.6.5 Function Slots and Property Lists

In traditional Lisp dialects, symbols are often understood as having three kinds of value at once:

- a *variable* value, which is used when the symbol appears in code in a variable reference context
- a *function* value, which is used when the symbol appears in code in a function name position (i.e. as the first element in an unquoted list)
- a *property list* value, which is used when the symbol is given as the first argument to Lisp's `put` or `get` functions.

Although Scheme (as one of its simplifications with respect to Lisp) does away with the distinction between variable and function namespaces, Guile currently retains some elements of the traditional structure in case they turn out to be useful when implementing translators for other languages, in particular Emacs Lisp.

Specifically, Guile symbols have two extra slots, one for a symbol's property list, and one for its "function value." The following procedures are provided to access these slots.

`symbol-fref` *symbol* [Scheme Procedure]
`scm_symbol_fref` (*symbol*) [C Function]
Return the contents of *symbol*'s *function slot*.

`symbol-fset!` *symbol value* [Scheme Procedure]
`scm_symbol_fset_x` (*symbol*, *value*) [C Function]
Set the contents of *symbol*'s function slot to *value*.

`symbol-pref` *symbol* [Scheme Procedure]
`scm_symbol_pref` (*symbol*) [C Function]
Return the *property list* currently associated with *symbol*.

`symbol-pset!` *symbol value* [Scheme Procedure]
`scm_symbol_pset_x` (*symbol*, *value*) [C Function]
Set *symbol*'s property list to *value*.

`symbol-property` *sym prop* [Scheme Procedure]
From *sym*'s property list, return the value for property *prop*. The assumption is that *sym*'s property list is an association list whose keys are distinguished from each other using `equal?`; *prop* should be one of the keys in that list. If the property list has no entry for *prop*, `symbol-property` returns `#f`.

`set-symbol-property!` *sym prop val* [Scheme Procedure]
In *sym*'s property list, set the value for property *prop* to *val*, or add a new entry for *prop*, with value *val*, if none already exists. For the structure of the property list, see `symbol-property`.

`symbol-property-remove!` *sym prop* [Scheme Procedure]
From *sym*'s property list, remove the entry for property *prop*, if there is one. For the structure of the property list, see `symbol-property`.

Support for these extra slots may be removed in a future release, and it is probably better to avoid using them. For a more modern and Schemely approach to properties, see Section 6.11.2 [Object Properties], page 285.

6.6.6.6 Extended Read Syntax for Symbols

The read syntax for a symbol is a sequence of letters, digits, and *extended alphabetic characters*, beginning with a character that cannot begin a number. In addition, the special cases of `+`, `-`, and `...` are read as symbols even though numbers can begin with `+`, `-` or `..`

Extended alphabetic characters may be used within identifiers as if they were letters. The set of extended alphabetic characters is:

```
! $ % & * + - . / : < = > ? @ ^ _ ~
```

In addition to the standard read syntax defined above (which is taken from R5RS (see Section "Formal syntax" in *The Revised^5 Report on Scheme*)), Guile provides an extended symbol read syntax that allows the inclusion of unusual characters such as space characters, newlines and parentheses. If (for whatever reason) you need to write a symbol containing characters not mentioned above, you can do so as follows.

- Begin the symbol with the characters `#{`,
- write the characters of the symbol and
- finish the symbol with the characters `}#`.

Here are a few examples of this form of read syntax. The first symbol needs to use extended syntax because it contains a space character, the second because it contains a line break, and the last because it looks like a number.

```
#{foo bar}#

#{what
ever}#

#{4242}#
```

Although Guile provides this extended read syntax for symbols, widespread usage of it is discouraged because it is not portable and not very readable.

Alternatively, if you enable the `r7rs-symbols` read option (see see Section 6.18.2 [Scheme Read], page 385), you can write arbitrary symbols using the same notation used for strings, except delimited by vertical bars instead of double quotes.

```
|foo bar|
|\x3BB; is a greek lambda|
|\| is a vertical bar|
```

Note that there's also an `r7rs-symbols` print option (see Section 6.18.3 [Scheme Write], page 386). To enable the use of this notation, evaluate one or both of the following expressions:

```
(read-enable  'r7rs-symbols)
(print-enable 'r7rs-symbols)
```

6.6.6.7 Uninterned Symbols

What makes symbols useful is that they are automatically kept unique. There are no two symbols that are distinct objects but have the same name. But of course, there is no rule without exception. In addition to the normal symbols that have been discussed up to now, you can also create special *uninterned* symbols that behave slightly differently.

To understand what is different about them and why they might be useful, we look at how normal symbols are actually kept unique.

Whenever Guile wants to find the symbol with a specific name, for example during `read` or when executing `string->symbol`, it first looks into a table of all existing symbols to find out whether a symbol with the given name already exists. When this is the case, Guile just returns that symbol. When not, a new symbol with the name is created and entered into the table so that it can be found later.

Sometimes you might want to create a symbol that is guaranteed 'fresh', i.e. a symbol that did not exist previously. You might also want to somehow guarantee that no one else will ever unintentionally stumble across your symbol in the future. These properties of a symbol are often needed when generating code during macro expansion. When introducing new temporary variables, you want to guarantee that they don't conflict with variables in other people's code.

The simplest way to arrange for this is to create a new symbol but not enter it into the global table of all symbols. That way, no one will ever get access to your symbol by chance. Symbols that are not in the table are called *uninterned*. Of course, symbols that *are* in the table are called *interned*.

You create new uninterned symbols with the function `make-symbol`. You can test whether a symbol is interned or not with `symbol-interned?`.

Uninterned symbols break the rule that the name of a symbol uniquely identifies the symbol object. Because of this, they can not be written out and read back in like interned symbols. Currently, Guile has no support for reading uninterned symbols. Note that the function `gensym` does not return uninterned symbols for this reason.

`make-symbol` *name* [Scheme Procedure]
`scm_make_symbol` (*name*) [C Function]
Return a new uninterned symbol with the name *name*. The returned symbol is guaranteed to be unique and future calls to `string->symbol` will not return it.

`symbol-interned?` *symbol* [Scheme Procedure]
`scm_symbol_interned_p` (*symbol*) [C Function]
Return `#t` if *symbol* is interned, otherwise return `#f`.

For example:

```
(define foo-1 (string->symbol "foo"))
(define foo-2 (string->symbol "foo"))
(define foo-3 (make-symbol "foo"))
(define foo-4 (make-symbol "foo"))

(eq? foo-1 foo-2)
⇒ #t
; Two interned symbols with the same name are the same object,

(eq? foo-1 foo-3)
⇒ #f
; but a call to make-symbol with the same name returns a
; distinct object.

(eq? foo-3 foo-4)
⇒ #f
; A call to make-symbol always returns a new object, even for
; the same name.

foo-3
⇒ #<uninterned-symbol foo 8085290>
; Uninterned symbols print differently from interned symbols,

(symbol? foo-3)
⇒ #t
; but they are still symbols,

(symbol-interned? foo-3)
⇒ #f
; just not interned.
```

6.6.7 Keywords

Keywords are self-evaluating objects with a convenient read syntax that makes them easy to type.

Guile's keyword support conforms to R5RS, and adds a (switchable) read syntax extension to permit keywords to begin with `:` as well as `#:`, or to end with `:`.

6.6.7.1 Why Use Keywords?

Keywords are useful in contexts where a program or procedure wants to be able to accept a large number of optional arguments without making its interface unmanageable.

To illustrate this, consider a hypothetical `make-window` procedure, which creates a new window on the screen for drawing into using some graphical toolkit. There are many parameters that the caller might like to specify, but which could also be sensibly defaulted, for example:

- color depth – Default: the color depth for the screen
- background color – Default: white
- width – Default: 600
- height – Default: 400

If `make-window` did not use keywords, the caller would have to pass in a value for each possible argument, remembering the correct argument order and using a special value to indicate the default value for that argument:

```
(make-window 'default              ;; Color depth
             'default              ;; Background color
             800                   ;; Width
             100                   ;; Height
             ...)                  ;; More make-window arguments
```

With keywords, on the other hand, defaulted arguments are omitted, and non-default arguments are clearly tagged by the appropriate keyword. As a result, the invocation becomes much clearer:

```
(make-window #:width 800 #:height 100)
```

On the other hand, for a simpler procedure with few arguments, the use of keywords would be a hindrance rather than a help. The primitive procedure `cons`, for example, would not be improved if it had to be invoked as

```
(cons #:car x #:cdr y)
```

So the decision whether to use keywords or not is purely pragmatic: use them if they will clarify the procedure invocation at point of call.

6.6.7.2 Coding With Keywords

If a procedure wants to support keywords, it should take a rest argument and then use whatever means is convenient to extract keywords and their corresponding arguments from the contents of that rest argument.

The following example illustrates the principle: the code for `make-window` uses a helper procedure called `get-keyword-value` to extract individual keyword arguments from the rest argument.

```
(define (get-keyword-value args keyword default)
```

```
  (let ((kv (memq keyword args)))
    (if (and kv (>= (length kv) 2))
        (cadr kv)
        default)))

(define (make-window . args)
  (let ((depth  (get-keyword-value args #:depth  screen-depth))
        (bg     (get-keyword-value args #:bg     "white"))
        (width  (get-keyword-value args #:width  800))
        (height (get-keyword-value args #:height 100))
        ...)
    ...))
```

But you don't need to write `get-keyword-value`. The `(ice-9 optargs)` module provides a set of powerful macros that you can use to implement keyword-supporting procedures like this:

```
(use-modules (ice-9 optargs))

(define (make-window . args)
  (let-keywords args #f ((depth  screen-depth)
                         (bg     "white")
                         (width  800)
                         (height 100))
    ...))
```

Or, even more economically, like this:

```
(use-modules (ice-9 optargs))

(define* (make-window #:key (depth  screen-depth)
                            (bg     "white")
                            (width  800)
                            (height 100))
  ...)
```

For further details on `let-keywords`, `define*` and other facilities provided by the `(ice-9 optargs)` module, see Section 6.9.4 [Optional Arguments], page 252.

To handle keyword arguments from procedures implemented in C, use `scm_c_bind_keyword_arguments` (see Section 6.6.7.4 [Keyword Procedures], page 176).

6.6.7.3 Keyword Read Syntax

Guile, by default, only recognizes a keyword syntax that is compatible with R5RS. A token of the form `#:NAME`, where `NAME` has the same syntax as a Scheme symbol (see Section 6.6.6.6 [Symbol Read Syntax], page 171), is the external representation of the keyword named `NAME`. Keyword objects print using this syntax as well, so values containing keyword objects can be read back into Guile. When used in an expression, keywords are self-quoting objects.

If the `keywords` read option is set to `'prefix`, Guile also recognizes the alternative read syntax `:NAME`. Otherwise, tokens of the form `:NAME` are read as symbols, as required by R5RS.

If the `keywords` read option is set to `'postfix`, Guile recognizes the SRFI-88 read syntax `NAME:` (see Section 7.5.42 [SRFI-88], page 653). Otherwise, tokens of this form are read as symbols.

To enable and disable the alternative non-R5RS keyword syntax, you use the `read-set!` procedure documented Section 6.18.2 [Scheme Read], page 385. Note that the `prefix` and `postfix` syntax are mutually exclusive.

```
(read-set! keywords 'prefix)

#:type
⇒
#:type

:type
⇒
#:type

(read-set! keywords 'postfix)

type:
⇒
#:type

:type
⇒
:type

(read-set! keywords #f)

#:type
⇒
#:type

:type
⊣
ERROR: In expression :type:
ERROR: Unbound variable: :type
ABORT: (unbound-variable)
```

6.6.7.4 Keyword Procedures

`keyword?` *obj* [Scheme Procedure]
`scm_keyword_p` (*obj*) [C Function]
Return `#t` if the argument *obj* is a keyword, else `#f`.

`keyword->symbol` *keyword* [Scheme Procedure]
`scm_keyword_to_symbol` (*keyword*) [C Function]
Return the symbol with the same name as *keyword*.

`symbol->keyword` *symbol* [Scheme Procedure]
`scm_symbol_to_keyword` (*symbol*) [C Function]
Return the keyword with the same name as *symbol*.

`int scm_is_keyword` (*SCM obj*) [C Function]
Equivalent to `scm_is_true (scm_keyword_p (obj))`.

`SCM scm_from_locale_keyword` (*const char *name*) [C Function]
`SCM scm_from_locale_keywordn` (*const char *name, size_t len*) [C Function]
Equivalent to `scm_symbol_to_keyword (scm_from_locale_symbol (name))` and `scm_symbol_to_keyword (scm_from_locale_symboln (name, len))`, respectively.

Note that these functions should *not* be used when *name* is a C string constant, because there is no guarantee that the current locale will match that of the execution character set, used for string and character constants. Most modern C compilers use UTF-8 by default, so in such cases we recommend `scm_from_utf8_keyword`.

`SCM scm_from_latin1_keyword` (*const char *name*) [C Function]
`SCM scm_from_utf8_keyword` (*const char *name*) [C Function]
Equivalent to `scm_symbol_to_keyword (scm_from_latin1_symbol (name))` and `scm_symbol_to_keyword (scm_from_utf8_symbol (name))`, respectively.

`void scm_c_bind_keyword_arguments` (*const char *subr, SCM rest, scm_t_keyword_arguments_flags flags, SCM keyword1, SCM *argp1, ..., SCM keywordN, SCM *argpN,* `SCM_UNDEFINED`) [C Function]
Extract the specified keyword arguments from *rest*, which is not modified. If the keyword argument *keyword1* is present in *rest* with an associated value, that value is stored in the variable pointed to by *argp1*, otherwise the variable is left unchanged. Similarly for the other keywords and argument pointers up to *keywordN* and *argpN*. The argument list to `scm_c_bind_keyword_arguments` must be terminated by `SCM_UNDEFINED`.

Note that since the variables pointed to by *argp1* through *argpN* are left unchanged if the associated keyword argument is not present, they should be initialized to their default values before calling `scm_c_bind_keyword_arguments`. Alternatively, you can initialize them to `SCM_UNDEFINED` before the call, and then use `SCM_UNBNDP` after the call to see which ones were provided.

If an unrecognized keyword argument is present in *rest* and *flags* does not contain `SCM_ALLOW_OTHER_KEYS`, or if non-keyword arguments are present and *flags* does not contain `SCM_ALLOW_NON_KEYWORD_ARGUMENTS`, an exception is raised. *subr* should be the name of the procedure receiving the keyword arguments, for purposes of error reporting.

For example:

```
SCM k_delimiter;
SCM k_grammar;
SCM sym_infix;

SCM my_string_join (SCM strings, SCM rest)
{
```

```
  SCM delimiter = SCM_UNDEFINED;
  SCM grammar   = sym_infix;

  scm_c_bind_keyword_arguments ("my-string-join", rest, 0,
                                k_delimiter, &delimiter,
                                k_grammar, &grammar,
                                SCM_UNDEFINED);

  if (SCM_UNBNDP (delimiter))
    delimiter = scm_from_utf8_string (" ");

  return scm_string_join (strings, delimiter, grammar);
}

void my_init ()
{
  k_delimiter = scm_from_utf8_keyword ("delimiter");
  k_grammar   = scm_from_utf8_keyword ("grammar");
  sym_infix   = scm_from_utf8_symbol  ("infix");
  scm_c_define_gsubr ("my-string-join", 1, 0, 1, my_string_join);
}
```

6.6.8 Pairs

Pairs are used to combine two Scheme objects into one compound object. Hence the name: A pair stores a pair of objects.

The data type *pair* is extremely important in Scheme, just like in any other Lisp dialect. The reason is that pairs are not only used to make two values available as one object, but that pairs are used for constructing lists of values. Because lists are so important in Scheme, they are described in a section of their own (see Section 6.6.9 [Lists], page 181).

Pairs can literally get entered in source code or at the REPL, in the so-called *dotted list* syntax. This syntax consists of an opening parentheses, the first element of the pair, a dot, the second element and a closing parentheses. The following example shows how a pair consisting of the two numbers 1 and 2, and a pair containing the symbols `foo` and `bar` can be entered. It is very important to write the whitespace before and after the dot, because otherwise the Scheme parser would not be able to figure out where to split the tokens.

```
(1 . 2)
(foo . bar)
```

But beware, if you want to try out these examples, you have to *quote* the expressions. More information about quotation is available in the section Section 6.18.1.1 [Expression Syntax], page 382. The correct way to try these examples is as follows.

```
'(1 . 2)
⇒
(1 . 2)
'(foo . bar)
⇒
(foo . bar)
```

A new pair is made by calling the procedure `cons` with two arguments. Then the argument values are stored into a newly allocated pair, and the pair is returned. The name `cons` stands for "construct". Use the procedure `pair?` to test whether a given Scheme object is a pair or not.

`cons` *x y* [Scheme Procedure]
`scm_cons` (*x*, *y*) [C Function]
Return a newly allocated pair whose car is *x* and whose cdr is *y*. The pair is guaranteed to be different (in the sense of `eq?`) from every previously existing object.

`pair?` *x* [Scheme Procedure]
`scm_pair_p` (*x*) [C Function]
Return `#t` if *x* is a pair; otherwise return `#f`.

`int scm_is_pair` (*SCM x*) [C Function]
Return 1 when *x* is a pair; otherwise return 0.

The two parts of a pair are traditionally called *car* and *cdr*. They can be retrieved with procedures of the same name (`car` and `cdr`), and can be modified with the procedures `set-car!` and `set-cdr!`.

Since a very common operation in Scheme programs is to access the car of a car of a pair, or the car of the cdr of a pair, etc., the procedures called `caar`, `cadr` and so on are also predefined. However, using these procedures is often detrimental to readability, and error-prone. Thus, accessing the contents of a list is usually better achieved using pattern matching techniques (see Section 7.8 [Pattern Matching], page 706).

`car` *pair* [Scheme Procedure]
`cdr` *pair* [Scheme Procedure]
`scm_car` (*pair*) [C Function]
`scm_cdr` (*pair*) [C Function]
Return the car or the cdr of *pair*, respectively.

`SCM SCM_CAR` (*SCM pair*) [C Macro]
`SCM SCM_CDR` (*SCM pair*) [C Macro]
These two macros are the fastest way to access the car or cdr of a pair; they can be thought of as compiling into a single memory reference.

These macros do no checking at all. The argument *pair* must be a valid pair.

`cddr` *pair* [Scheme Procedure]
`cdar` *pair* [Scheme Procedure]
`cadr` *pair* [Scheme Procedure]
`caar` *pair* [Scheme Procedure]
`cdddr` *pair* [Scheme Procedure]
`cddar` *pair* [Scheme Procedure]
`cdadr` *pair* [Scheme Procedure]
`cdaar` *pair* [Scheme Procedure]
`caddr` *pair* [Scheme Procedure]
`cadar` *pair* [Scheme Procedure]
`caadr` *pair* [Scheme Procedure]

`caaar` *pair* [Scheme Procedure]
`cddddr` *pair* [Scheme Procedure]
`cdddar` *pair* [Scheme Procedure]
`cddadr` *pair* [Scheme Procedure]
`cddaar` *pair* [Scheme Procedure]
`cdaddr` *pair* [Scheme Procedure]
`cdadar` *pair* [Scheme Procedure]
`cdaadr` *pair* [Scheme Procedure]
`cdaaar` *pair* [Scheme Procedure]
`cadddr` *pair* [Scheme Procedure]
`caddar` *pair* [Scheme Procedure]
`cadadr` *pair* [Scheme Procedure]
`cadaar` *pair* [Scheme Procedure]
`caaddr` *pair* [Scheme Procedure]
`caadar` *pair* [Scheme Procedure]
`caaadr` *pair* [Scheme Procedure]
`caaaar` *pair* [Scheme Procedure]
`scm_cddr` (*pair*) [C Function]
`scm_cdar` (*pair*) [C Function]
`scm_cadr` (*pair*) [C Function]
`scm_caar` (*pair*) [C Function]
`scm_cdddr` (*pair*) [C Function]
`scm_cddar` (*pair*) [C Function]
`scm_cdadr` (*pair*) [C Function]
`scm_cdaar` (*pair*) [C Function]
`scm_caddr` (*pair*) [C Function]
`scm_cadar` (*pair*) [C Function]
`scm_caadr` (*pair*) [C Function]
`scm_caaar` (*pair*) [C Function]
`scm_cddddr` (*pair*) [C Function]
`scm_cdddar` (*pair*) [C Function]
`scm_cddadr` (*pair*) [C Function]
`scm_cddaar` (*pair*) [C Function]
`scm_cdaddr` (*pair*) [C Function]
`scm_cdadar` (*pair*) [C Function]
`scm_cdaadr` (*pair*) [C Function]
`scm_cdaaar` (*pair*) [C Function]
`scm_cadddr` (*pair*) [C Function]
`scm_caddar` (*pair*) [C Function]
`scm_cadadr` (*pair*) [C Function]
`scm_cadaar` (*pair*) [C Function]
`scm_caaddr` (*pair*) [C Function]
`scm_caadar` (*pair*) [C Function]
`scm_caaadr` (*pair*) [C Function]
`scm_caaaar` (*pair*) [C Function]

These procedures are compositions of `car` and `cdr`, where for example `caddr` could be defined by

```
(define caddr (lambda (x) (car (cdr (cdr x)))))
```

`cadr`, `caddr` and `cadddr` pick out the second, third or fourth elements of a list, respectively. SRFI-1 provides the same under the names `second`, `third` and `fourth` (see Section 7.5.3.3 [SRFI-1 Selectors], page 586).

`set-car!` *pair value* [Scheme Procedure]
`scm_set_car_x` (*pair*, *value*) [C Function]
Stores *value* in the car field of *pair*. The value returned by `set-car!` is unspecified.

`set-cdr!` *pair value* [Scheme Procedure]
`scm_set_cdr_x` (*pair*, *value*) [C Function]
Stores *value* in the cdr field of *pair*. The value returned by `set-cdr!` is unspecified.

6.6.9 Lists

A very important data type in Scheme—as well as in all other Lisp dialects—is the data type *list*.[1]

This is the short definition of what a list is:

- Either the empty list `()`,
- or a pair which has a list in its cdr.

6.6.9.1 List Read Syntax

The syntax for lists is an opening parentheses, then all the elements of the list (separated by whitespace) and finally a closing parentheses.[2].

```
(1 2 3)             ; a list of the numbers 1, 2 and 3
("foo" bar 3.1415) ; a string, a symbol and a real number
()                  ; the empty list
```

The last example needs a bit more explanation. A list with no elements, called the *empty list*, is special in some ways. It is used for terminating lists by storing it into the cdr of the last pair that makes up a list. An example will clear that up:

```
(car '(1))
⇒
1
(cdr '(1))
⇒
()
```

This example also shows that lists have to be quoted when written (see Section 6.18.1.1 [Expression Syntax], page 382), because they would otherwise be mistakingly taken as procedure applications (see Section 3.2.2 [Simple Invocation], page 18).

6.6.9.2 List Predicates

Often it is useful to test whether a given Scheme object is a list or not. List-processing procedures could use this information to test whether their input is valid, or they could do different things depending on the datatype of their arguments.

---

[1] Strictly speaking, Scheme does not have a real datatype *list*. Lists are made up of *chained pairs*, and only exist by definition—a list is a chain of pairs which looks like a list.

[2] Note that there is no separation character between the list elements, like a comma or a semicolon.

`list?` *x* [Scheme Procedure]
`scm_list_p` (*x*) [C Function]
Return `#t` if *x* is a proper list, else `#f`.

The predicate `null?` is often used in list-processing code to tell whether a given list has run out of elements. That is, a loop somehow deals with the elements of a list until the list satisfies `null?`. Then, the algorithm terminates.

`null?` *x* [Scheme Procedure]
`scm_null_p` (*x*) [C Function]
Return `#t` if *x* is the empty list, else `#f`.

`int scm_is_null` (*SCM x*) [C Function]
Return 1 when *x* is the empty list; otherwise return 0.

6.6.9.3 List Constructors

This section describes the procedures for constructing new lists. `list` simply returns a list where the elements are the arguments, `cons*` is similar, but the last argument is stored in the cdr of the last pair of the list.

`list` *elem ...* [Scheme Procedure]
`scm_list_1` (*elem1*) [C Function]
`scm_list_2` (*elem1*, *elem2*) [C Function]
`scm_list_3` (*elem1*, *elem2*, *elem3*) [C Function]
`scm_list_4` (*elem1*, *elem2*, *elem3*, *elem4*) [C Function]
`scm_list_5` (*elem1*, *elem2*, *elem3*, *elem4*, *elem5*) [C Function]
`scm_list_n` (*elem1*, *...*, *elemN*, `SCM_UNDEFINED`) [C Function]
Return a new list containing elements *elem ...*.

`scm_list_n` takes a variable number of arguments, terminated by the special `SCM_UNDEFINED`. That final `SCM_UNDEFINED` is not included in the list. None of *elem ...* can themselves be `SCM_UNDEFINED`, or `scm_list_n` will terminate at that point.

`cons*` *arg1 arg2 ...* [Scheme Procedure]
Like `list`, but the last arg provides the tail of the constructed list, returning `(cons` *arg1* `(cons` *arg2* `(cons ...` *argn*`)))`. Requires at least one argument. If given one argument, that argument is returned as result. This function is called `list*` in some other Schemes and in Common LISP.

`list-copy` *lst* [Scheme Procedure]
`scm_list_copy` (*lst*) [C Function]
Return a (newly-created) copy of *lst*.

`make-list` *n* [*init*] [Scheme Procedure]
Create a list containing of *n* elements, where each element is initialized to *init*. *init* defaults to the empty list `()` if not given.

Note that `list-copy` only makes a copy of the pairs which make up the spine of the lists. The list elements are not copied, which means that modifying the elements of the new list also modifies the elements of the old list. On the other hand, applying procedures

like `set-cdr!` or `delv!` to the new list will not alter the old list. If you also need to copy the list elements (making a deep copy), use the procedure `copy-tree` (see Section 6.11.4 [Copying], page 287).

6.6.9.4 List Selection

These procedures are used to get some information about a list, or to retrieve one or more elements of a list.

`length` *lst* [Scheme Procedure]
`scm_length` (*lst*) [C Function]
Return the number of elements in list *lst*.

`last-pair` *lst* [Scheme Procedure]
`scm_last_pair` (*lst*) [C Function]
Return the last pair in *lst*, signalling an error if *lst* is circular.

`list-ref` *list k* [Scheme Procedure]
`scm_list_ref` (*list*, *k*) [C Function]
Return the *k*th element from *list*.

`list-tail` *lst k* [Scheme Procedure]
`list-cdr-ref` *lst k* [Scheme Procedure]
`scm_list_tail` (*lst*, *k*) [C Function]
Return the "tail" of *lst* beginning with its *k*th element. The first element of the list is considered to be element 0.

`list-tail` and `list-cdr-ref` are identical. It may help to think of `list-cdr-ref` as accessing the *k*th cdr of the list, or returning the results of cdring *k* times down *lst*.

`list-head` *lst k* [Scheme Procedure]
`scm_list_head` (*lst*, *k*) [C Function]
Copy the first *k* elements from *lst* into a new list, and return it.

6.6.9.5 Append and Reverse

`append` and `append!` are used to concatenate two or more lists in order to form a new list. `reverse` and `reverse!` return lists with the same elements as their arguments, but in reverse order. The procedure variants with an `!` directly modify the pairs which form the list, whereas the other procedures create new pairs. This is why you should be careful when using the side-effecting variants.

`append` *lst ... obj* [Scheme Procedure]
`append` [Scheme Procedure]
`append!` *lst ... obj* [Scheme Procedure]
`append!` [Scheme Procedure]
`scm_append` (*lstlst*) [C Function]
`scm_append_x` (*lstlst*) [C Function]
Return a list comprising all the elements of lists *lst ... obj*. If called with no arguments, return the empty list.

```
(append '(x) '(y))          ⇒  (x y)
```

```
(append '(a) '(b c d))      ⇒  (a b c d)
(append '(a (b)) '((c)))    ⇒  (a (b) (c))
```

The last argument *obj* may actually be any object; an improper list results if the last argument is not a proper list.

```
(append '(a b) '(c . d))    ⇒  (a b c . d)
(append '() 'a)             ⇒  a
```

`append` doesn't modify the given lists, but the return may share structure with the final *obj*. `append!` is permitted, but not required, to modify the given lists to form its return.

For `scm_append` and `scm_append_x`, *lstlst* is a list of the list operands *lst* ... *obj*. That *lstlst* itself is not modified or used in the return.

`reverse` *lst* [Scheme Procedure]
`reverse!` *lst* [*newtail*] [Scheme Procedure]
`scm_reverse` (*lst*) [C Function]
`scm_reverse_x` (*lst*, *newtail*) [C Function]

Return a list comprising the elements of *lst*, in reverse order.

`reverse` constructs a new list. `reverse!` is permitted, but not required, to modify *lst* in constructing its return.

For `reverse!`, the optional *newtail* is appended to the result. *newtail* isn't reversed, it simply becomes the list tail. For `scm_reverse_x`, the *newtail* parameter is mandatory, but can be `SCM_EOL` if no further tail is required.

6.6.9.6 List Modification

The following procedures modify an existing list, either by changing elements of the list, or by changing the list structure itself.

`list-set!` *list k val* [Scheme Procedure]
`scm_list_set_x` (*list*, *k*, *val*) [C Function]

Set the *k*th element of *list* to *val*.

`list-cdr-set!` *list k val* [Scheme Procedure]
`scm_list_cdr_set_x` (*list*, *k*, *val*) [C Function]

Set the *k*th cdr of *list* to *val*.

`delq` *item lst* [Scheme Procedure]
`scm_delq` (*item*, *lst*) [C Function]

Return a newly-created copy of *lst* with elements `eq?` to *item* removed. This procedure mirrors `memq`: `delq` compares elements of *lst* against *item* with `eq?`.

`delv` *item lst* [Scheme Procedure]
`scm_delv` (*item*, *lst*) [C Function]

Return a newly-created copy of *lst* with elements `eqv?` to *item* removed. This procedure mirrors `memv`: `delv` compares elements of *lst* against *item* with `eqv?`.

`delete` *item lst* [Scheme Procedure]
`scm_delete` (*item*, *lst*) [C Function]

Return a newly-created copy of *lst* with elements `equal?` to *item* removed. This procedure mirrors `member`: `delete` compares elements of *lst* against *item* with `equal?`.

See also SRFI-1 which has an extended `delete` (Section 7.5.3.8 [SRFI-1 Deleting], page 593), and also an `lset-difference` which can delete multiple *item*s in one call (Section 7.5.3.10 [SRFI-1 Set Operations], page 595).

`delq!` *item lst* [Scheme Procedure]
`delv!` *item lst* [Scheme Procedure]
`delete!` *item lst* [Scheme Procedure]
`scm_delq_x` (*item, lst*) [C Function]
`scm_delv_x` (*item, lst*) [C Function]
`scm_delete_x` (*item, lst*) [C Function]

These procedures are destructive versions of `delq`, `delv` and `delete`: they modify the pointers in the existing *lst* rather than creating a new list. Caveat evaluator: Like other destructive list functions, these functions cannot modify the binding of *lst*, and so cannot be used to delete the first element of *lst* destructively.

`delq1!` *item lst* [Scheme Procedure]
`scm_delq1_x` (*item, lst*) [C Function]

Like `delq!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `eq?`. See also `delv1!` and `delete1!`.

`delv1!` *item lst* [Scheme Procedure]
`scm_delv1_x` (*item, lst*) [C Function]

Like `delv!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `eqv?`. See also `delq1!` and `delete1!`.

`delete1!` *item lst* [Scheme Procedure]
`scm_delete1_x` (*item, lst*) [C Function]

Like `delete!`, but only deletes the first occurrence of *item* from *lst*. Tests for equality using `equal?`. See also `delq1!` and `delv1!`.

`filter` *pred lst* [Scheme Procedure]
`filter!` *pred lst* [Scheme Procedure]

Return a list containing all elements from *lst* which satisfy the predicate *pred*. The elements in the result list have the same order as in *lst*. The order in which *pred* is applied to the list elements is not specified.

`filter` does not change *lst*, but the result may share a tail with it. `filter!` may modify *lst* to construct its return.

6.6.9.7 List Searching

The following procedures search lists for particular elements. They use different comparison predicates for comparing list elements with the object to be searched. When they fail, they return `#f`, otherwise they return the sublist whose car is equal to the search object, where equality depends on the equality predicate used.

`memq` *x lst* [Scheme Procedure]
`scm_memq` (*x, lst*) [C Function]

Return the first sublist of *lst* whose car is `eq?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

`memv` *x lst* [Scheme Procedure]
`scm_memv` (*x, lst*) [C Function]

Return the first sublist of *lst* whose car is `eqv?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

`member` *x lst* [Scheme Procedure]
`scm_member` (*x, lst*) [C Function]

Return the first sublist of *lst* whose car is `equal?` to *x* where the sublists of *lst* are the non-empty lists returned by `(list-tail lst k)` for *k* less than the length of *lst*. If *x* does not occur in *lst*, then `#f` (not the empty list) is returned.

See also SRFI-1 which has an extended `member` function (Section 7.5.3.7 [SRFI-1 Searching], page 592).

6.6.9.8 List Mapping

List processing is very convenient in Scheme because the process of iterating over the elements of a list can be highly abstracted. The procedures in this section are the most basic iterating procedures for lists. They take a procedure and one or more lists as arguments, and apply the procedure to each element of the list. They differ in their return value.

`map` *proc arg1 arg2 . . .* [Scheme Procedure]
`map-in-order` *proc arg1 arg2 . . .* [Scheme Procedure]
`scm_map` (*proc, arg1, args*) [C Function]

Apply *proc* to each element of the list *arg1* (if only two arguments are given), or to the corresponding elements of the argument lists (if more than two arguments are given). The result(s) of the procedure applications are saved and returned in a list. For `map`, the order of procedure applications is not specified, `map-in-order` applies the procedure from left to right to the list elements.

`for-each` *proc arg1 arg2 . . .* [Scheme Procedure]

Like `map`, but the procedure is always applied from left to right, and the result(s) of the procedure applications are thrown away. The return value is not specified.

See also SRFI-1 which extends these functions to take lists of unequal lengths (Section 7.5.3.5 [SRFI-1 Fold and Map], page 588).

6.6.10 Vectors

Vectors are sequences of Scheme objects. Unlike lists, the length of a vector, once the vector is created, cannot be changed. The advantage of vectors over lists is that the time required to access one element of a vector given its *position* (synonymous with *index*), a zero-origin number, is constant, whereas lists have an access time linear to the position of the accessed element in the list.

Vectors can contain any kind of Scheme object; it is even possible to have different types of objects in the same vector. For vectors containing vectors, you may wish to use arrays, instead. Note, too, that vectors are the special case of one dimensional non-uniform arrays and that most array procedures operate happily on vectors (see Section 6.6.13 [Arrays], page 200).

Also see Section 7.5.30 [SRFI-43], page 641, for a comprehensive vector library.

6.6.10.1 Read Syntax for Vectors

Vectors can literally be entered in source code, just like strings, characters or some of the other data types. The read syntax for vectors is as follows: A sharp sign (`#`), followed by an opening parentheses, all elements of the vector in their respective read syntax, and finally a closing parentheses. Like strings, vectors do not have to be quoted.

The following are examples of the read syntax for vectors; where the first vector only contains numbers and the second three different object types: a string, a symbol and a number in hexadecimal notation.

```
#(1 2 3)
#("Hello" foo #xdeadbeef)
```

6.6.10.2 Dynamic Vector Creation and Validation

Instead of creating a vector implicitly by using the read syntax just described, you can create a vector dynamically by calling one of the `vector` and `list->vector` primitives with the list of Scheme values that you want to place into a vector. The size of the vector thus created is determined implicitly by the number of arguments given.

`vector` *arg ...* [Scheme Procedure]
`list->vector` *l* [Scheme Procedure]
`scm_vector` (*l*) [C Function]

Return a newly allocated vector composed of the given arguments. Analogous to `list`.

```
(vector 'a 'b 'c) ⇒ #(a b c)
```

The inverse operation is `vector->list`:

`vector->list` *v* [Scheme Procedure]
`scm_vector_to_list` (*v*) [C Function]

Return a newly allocated list composed of the elements of *v*.

```
(vector->list #(dah dah didah)) ⇒  (dah dah didah)
(list->vector '(dididit dah)) ⇒  #(dididit dah)
```

To allocate a vector with an explicitly specified size, use `make-vector`. With this primitive you can also specify an initial value for the vector elements (the same value for all elements, that is):

`make-vector` *len* [*fill*] [Scheme Procedure]
`scm_make_vector` (*len*, *fill*) [C Function]

Return a newly allocated vector of *len* elements. If a second argument is given, then each position is initialized to *fill*. Otherwise the initial contents of each position is unspecified.

`SCM scm_c_make_vector` (*size_t k*, *SCM fill*) [C Function]

Like `scm_make_vector`, but the length is given as a `size_t`.

To check whether an arbitrary Scheme value *is* a vector, use the `vector?` primitive:

`vector?` *obj* [Scheme Procedure]
`scm_vector_p` (*obj*) [C Function]
Return `#t` if *obj* is a vector, otherwise return `#f`.

`int scm_is_vector` (*SCM obj*) [C Function]
Return non-zero when *obj* is a vector, otherwise return `zero`.

6.6.10.3 Accessing and Modifying Vector Contents

`vector-length` and `vector-ref` return information about a given vector, respectively its size and the elements that are contained in the vector.

`vector-length` *vector* [Scheme Procedure]
`scm_vector_length` (*vector*) [C Function]
Return the number of elements in *vector* as an exact integer.

`size_t scm_c_vector_length` (*SCM vec*) [C Function]
Return the number of elements in *vec* as a `size_t`.

`vector-ref` *vec k* [Scheme Procedure]
`scm_vector_ref` (*vec, k*) [C Function]
Return the contents of position *k* of *vec*. *k* must be a valid index of *vec*.

```
(vector-ref #(1 1 2 3 5 8 13 21) 5) ⇒ 8
(vector-ref #(1 1 2 3 5 8 13 21)
    (let ((i (round (* 2 (acos -1)))))
      (if (inexact? i)
        (inexact->exact i)
           i))) ⇒ 13
```

`SCM scm_c_vector_ref` (*SCM vec, size_t k*) [C Function]
Return the contents of position *k* (a `size_t`) of *vec*.

A vector created by one of the dynamic vector constructor procedures (see Section 6.6.10.2 [Vector Creation], page 187) can be modified using the following procedures.

NOTE: According to R5RS, it is an error to use any of these procedures on a literally read vector, because such vectors should be considered as constants. Currently, however, Guile does not detect this error.

`vector-set!` *vec k obj* [Scheme Procedure]
`scm_vector_set_x` (*vec, k, obj*) [C Function]
Store *obj* in position *k* of *vec*. *k* must be a valid index of *vec*. The value returned by '`vector-set!`' is unspecified.

```
(let ((vec (vector 0 '(2 2 2 2) "Anna")))
  (vector-set! vec 1 '("Sue" "Sue"))
  vec) ⇒  #(0 ("Sue" "Sue") "Anna")
```

`void scm_c_vector_set_x` (*SCM vec, size_t k, SCM obj*) [C Function]
Store *obj* in position *k* (a `size_t`) of *vec*.

`vector-fill!` *vec fill* [Scheme Procedure]
`scm_vector_fill_x` (*vec, fill*) [C Function]
Store *fill* in every position of *vec*. The value returned by `vector-fill!` is unspecified.

`vector-copy` *vec* [Scheme Procedure]
`scm_vector_copy` (*vec*) [C Function]
Return a copy of *vec*.

`vector-move-left!` *vec1 start1 end1 vec2 start2* [Scheme Procedure]
`scm_vector_move_left_x` (*vec1, start1, end1, vec2, start2*) [C Function]
Copy elements from *vec1*, positions *start1* to *end1*, to *vec2* starting at position *start2*. *start1* and *start2* are inclusive indices; *end1* is exclusive.

`vector-move-left!` copies elements in leftmost order. Therefore, in the case where *vec1* and *vec2* refer to the same vector, `vector-move-left!` is usually appropriate when *start1* is greater than *start2*.

`vector-move-right!` *vec1 start1 end1 vec2 start2* [Scheme Procedure]
`scm_vector_move_right_x` (*vec1, start1, end1, vec2, start2*) [C Function]
Copy elements from *vec1*, positions *start1* to *end1*, to *vec2* starting at position *start2*. *start1* and *start2* are inclusive indices; *end1* is exclusive.

`vector-move-right!` copies elements in rightmost order. Therefore, in the case where *vec1* and *vec2* refer to the same vector, `vector-move-right!` is usually appropriate when *start1* is less than *start2*.

6.6.10.4 Vector Accessing from C

A vector can be read and modified from C with the functions `scm_c_vector_ref` and `scm_c_vector_set_x`, for example. In addition to these functions, there are two more ways to access vectors from C that might be more efficient in certain situations: you can restrict yourself to *simple vectors* and then use the very fast *simple vector macros*; or you can use the very general framework for accessing all kinds of arrays (see Section 6.6.13.5 [Accessing Arrays from C], page 210), which is more verbose, but can deal efficiently with all kinds of vectors (and arrays). For vectors, you can use the `scm_vector_elements` and `scm_vector_writable_elements` functions as shortcuts.

`int scm_is_simple_vector` (*SCM obj*) [C Function]
Return non-zero if *obj* is a simple vector, else return zero. A simple vector is a vector that can be used with the `SCM_SIMPLE_*` macros below.

The following functions are guaranteed to return simple vectors: `scm_make_vector`, `scm_c_make_vector`, `scm_vector`, `scm_list_to_vector`.

`size_t SCM_SIMPLE_VECTOR_LENGTH` (*SCM vec*) [C Macro]
Evaluates to the length of the simple vector *vec*. No type checking is done.

`SCM SCM_SIMPLE_VECTOR_REF` (*SCM vec, size_t idx*) [C Macro]
Evaluates to the element at position *idx* in the simple vector *vec*. No type or range checking is done.

`void SCM_SIMPLE_VECTOR_SET` (*SCM vec, size_t idx, SCM val*) [C Macro]
Sets the element at position *idx* in the simple vector *vec* to *val*. No type or range checking is done.

`const SCM * scm_vector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Acquire a handle for the vector *vec* and return a pointer to the elements of it. This pointer can only be used to read the elements of *vec*. When *vec* is not a vector, an error is signaled. The handle must eventually be released with `scm_array_handle_release`.

The variables pointed to by *lenp* and *incp* are filled with the number of elements of the vector and the increment (number of elements) between successive elements, respectively. Successive elements of *vec* need not be contiguous in their underlying "root vector" returned here; hence the increment is not necessarily equal to 1 and may well be negative too (see Section 6.6.13.3 [Shared Arrays], page 205).

The following example shows the typical way to use this function. It creates a list of all elements of *vec* (in reverse order).

```
scm_t_array_handle handle;
size_t i, len;
ssize_t inc;
const SCM *elt;
SCM list;

elt = scm_vector_elements (vec, &handle, &len, &inc);
list = SCM_EOL;
for (i = 0; i < len; i++, elt += inc)
  list = scm_cons (*elt, list);
scm_array_handle_release (&handle);
```

`SCM * scm_vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_elements` but the pointer can be used to modify the vector.

The following example shows the typical way to use this function. It fills a vector with `#t`.

```
scm_t_array_handle handle;
size_t i, len;
ssize_t inc;
SCM *elt;

elt = scm_vector_writable_elements (vec, &handle, &len, &inc);
for (i = 0; i < len; i++, elt += inc)
  *elt = SCM_BOOL_T;
scm_array_handle_release (&handle);
```

6.6.10.5 Uniform Numeric Vectors

A uniform numeric vector is a vector whose elements are all of a single numeric type. Guile offers uniform numeric vectors for signed and unsigned 8-bit, 16-bit, 32-bit, and 64-bit integers, two sizes of floating point values, and complex floating-point numbers of these two sizes. See Section 7.5.5 [SRFI-4], page 598, for more information.

For many purposes, bytevectors work just as well as uniform vectors, and have the advantage that they integrate well with binary input and output. See Section 6.6.12 [Bytevectors], page 193, for more information on bytevectors.

6.6.11 Bit Vectors

Bit vectors are zero-origin, one-dimensional arrays of booleans. They are displayed as a sequence of `0`s and `1`s prefixed by `#*`, e.g.,

```
(make-bitvector 8 #f) ⇒
#*00000000
```

Bit vectors are the special case of one dimensional bit arrays, and can thus be used with the array procedures, See Section 6.6.13 [Arrays], page 200.

`bitvector?` *obj* [Scheme Procedure]
Return `#t` when *obj* is a bitvector, else return `#f`.

`make-bitvector` *len* [*fill*] [Scheme Procedure]
Create a new bitvector of length *len* and optionally initialize all elements to *fill*.

`bitvector` *bit ...* [Scheme Procedure]
Create a new bitvector with the arguments as elements.

`bitvector-length` *vec* [Scheme Procedure]
Return the length of the bitvector *vec*.

`bitvector-bit-set?` *vec idx* [Scheme Procedure]
`bitvector-bit-clear?` *vec idx* [Scheme Procedure]
Return `#t` if the bit at index *idx* of the bitvector *vec* is set (for `bitvector-bit-set?`) or clear (for `bitvector-bit-clear?`).

`bitvector-set-bit!` *vec idx* [Scheme Procedure]
`bitvector-clear-bit!` *vec idx* [Scheme Procedure]
Set (for `bitvector-set-bit!`) or clear (for `bitvector-clear-bit!`) the bit at index *idx* of the bitvector *vec*.

`bitvector-set-all-bits!` *vec* [Scheme Procedure]
`bitvector-clear-all-bits!` *vec* [Scheme Procedure]
`bitvector-flip-all-bits!` *vec* [Scheme Procedure]
Set, clear, or flip all bits of *vec*.

`list->bitvector` *list* [Scheme Procedure]
`scm_list_to_bitvector` (*list*) [C Function]
Return a new bitvector initialized with the elements of *list*.

`bitvector->list` *vec* [Scheme Procedure]
`scm_bitvector_to_list` (*vec*) [C Function]
Return a new list initialized with the elements of the bitvector *vec*.

`bitvector-count` *bitvector* [Scheme Procedure]
Return a count of how many entries in *bitvector* are set.

```
(bitvector-count #*000111000)  ⇒ 3
```

`bitvector-count-bits` *bitvector bits* [Scheme Procedure]

Return a count of how many entries in *bitvector* are set, with the bitvector *bits* selecting the entries to consider. *bitvector* must be at least as long as *bits*.

For example,

```
(bitvector-count-bits #*01110111 #*11001101) ⇒ 3
```

`bitvector-position` *bitvector bool start* [Scheme Procedure]
`scm_bitvector_position` (*bitvector, bool, start*) [C Function]

Return the index of the first occurrence of *bool* in *bitvector*, starting from *start*. If there is no *bool* entry between *start* and the end of *bitvector*, then return `#f`. For example,

```
(bitvector-position #*000101 #t 0)  ⇒ 3
(bitvector-position #*0001111 #f 3) ⇒ #f
```

`bitvector-set-bits!` *bitvector bits* [Scheme Procedure]

Set entries of *bitvector* to `#t`, with *bits* selecting the bits to set. The return value is unspecified. *bitvector* must be at least as long as *bits*.

```
(define bv (bitvector-copy #*11000010))
(bitvector-set-bits! bv #*10010001)
bv
⇒ #*11010011
```

`bitvector-clear-bits!` *bitvector bits* [Scheme Procedure]

Set entries of *bitvector* to `#f`, with *bits* selecting the bits to clear. The return value is unspecified. *bitvector* must be at least as long as *bits*.

```
(define bv (bitvector-copy #*11000010))
(bitvector-clear-bits! bv #*10010001)
bv
⇒ #*01000010
```

`int scm_is_bitvector` (*SCM obj*) [C Function]
`SCM scm_c_make_bitvector` (*size_t len, SCM fill*) [C Function]
`int scm_bitvector_bit_is_set` (*SCM vec, size_t idx*) [C Function]
`int scm_bitvector_bit_is_clear` (*SCM vec, size_t idx*) [C Function]
`void scm_c_bitvector_set_bit_x` (*SCM vec, size_t idx*) [C Function]
`void scm_c_bitvector_clear_bit_x` (*SCM vec, size_t idx*) [C Function]
`void scm_c_bitvector_set_bits_x` (*SCM vec, SCM bits*) [C Function]
`void scm_c_bitvector_clear_bits_x` (*SCM vec, SCM bits*) [C Function]
`void scm_c_bitvector_set_all_bits_x` (*SCM vec*) [C Function]
`void scm_c_bitvector_clear_all_bits_x` (*SCM vec*) [C Function]
`void scm_c_bitvector_flip_all_bits_x` (*SCM vec*) [C Function]
`size_t scm_c_bitvector_length` (*SCM bitvector*) [C Function]
`size_t scm_c_bitvector_count` (*SCM bitvector*) [C Function]
`size_t scm_c_bitvector_count_bits` (*SCM bitvector, SCM bits*) [C Function]

C API for the corresponding Scheme bitvector interfaces.

`const scm_t_uint32 * scm_bitvector_elements` (*SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_elements` (see Section 6.6.10.4 [Vector Accessing from C], page 189), but for bitvectors. The variable pointed to by *offp* is set to the value returned by `scm_array_handle_bit_elements_offset`. See `scm_array_handle_bit_elements` for how to use the returned pointer and the offset.

`scm_t_uint32 * scm_bitvector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *offp, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_bitvector_elements`, but the pointer is good for reading and writing.

6.6.12 Bytevectors

A *bytevector* is a raw bit string. The `(rnrs bytevectors)` module provides the programming interface specified by the Revised^6 Report on the Algorithmic Language Scheme (R6RS) (`http://www.r6rs.org/`). It contains procedures to manipulate bytevectors and interpret their contents in a number of ways: bytevector contents can be accessed as signed or unsigned integer of various sizes and endianness, as IEEE-754 floating point numbers, or as strings. It is a useful tool to encode and decode binary data.

The R6RS (Section 4.3.4) specifies an external representation for bytevectors, whereby the octets (integers in the range 0–255) contained in the bytevector are represented as a list prefixed by `#vu8`:

```
#vu8(1 53 204)
```

denotes a 3-byte bytevector containing the octets 1, 53, and 204. Like string literals, booleans, etc., bytevectors are "self-quoting", i.e., they do not need to be quoted:

```
#vu8(1 53 204)
⇒ #vu8(1 53 204)
```

Bytevectors can be used with the binary input/output primitives (see Section 6.14.2 [Binary I/O], page 332).

6.6.12.1 Endianness

Some of the following procedures take an *endianness* parameter. The *endianness* is defined as the order of bytes in multi-byte numbers: numbers encoded in *big endian* have their most significant bytes written first, whereas numbers encoded in *little endian* have their least significant bytes first[3].

Little-endian is the native endianness of the IA32 architecture and its derivatives, while big-endian is native to SPARC and PowerPC, among others. The `native-endianness` procedure returns the native endianness of the machine it runs on.

`native-endianness` [Scheme Procedure]

`scm_native_endianness ()` [C Function]

Return a value denoting the native endianness of the host machine.

[3] Big-endian and little-endian are the most common "endiannesses", but others do exist. For instance, the GNU MP library allows *word order* to be specified independently of *byte order* (see Section "Integer Import and Export" in *The GNU Multiple Precision Arithmetic Library Manual*).

`endianness` *symbol* [Scheme Macro]
Return an object denoting the endianness specified by *symbol*. If *symbol* is neither `big` nor `little` then an error is raised at expand-time.

`scm_endianness_big` [C Variable]
`scm_endianness_little` [C Variable]
The objects denoting big- and little-endianness, respectively.

6.6.12.2 Manipulating Bytevectors

Bytevectors can be created, copied, and analyzed with the following procedures and C functions.

`make-bytevector` *len* [*fill*] [Scheme Procedure]
`scm_make_bytevector` (*len, fill*) [C Function]
`scm_c_make_bytevector` (*size_t len*) [C Function]
Return a new bytevector of *len* bytes. Optionally, if *fill* is given, fill it with *fill*; *fill* must be in the range [-128,255].

`bytevector?` *obj* [Scheme Procedure]
`scm_bytevector_p` (*obj*) [C Function]
Return true if *obj* is a bytevector.

`int scm_is_bytevector` (*SCM obj*) [C Function]
Equivalent to `scm_is_true (scm_bytevector_p (obj))`.

`bytevector-length` *bv* [Scheme Procedure]
`scm_bytevector_length` (*bv*) [C Function]
Return the length in bytes of bytevector *bv*.

`size_t scm_c_bytevector_length` (*SCM bv*) [C Function]
Likewise, return the length in bytes of bytevector *bv*.

`bytevector=?` *bv1 bv2* [Scheme Procedure]
`scm_bytevector_eq_p` (*bv1, bv2*) [C Function]
Return is *bv1* equals to *bv2*—i.e., if they have the same length and contents.

`bytevector-fill!` *bv fill* [Scheme Procedure]
`scm_bytevector_fill_x` (*bv, fill*) [C Function]
Fill bytevector *bv* with *fill*, a byte.

`bytevector-copy!` *source source-start target target-start len* [Scheme Procedure]
`scm_bytevector_copy_x` (*source, source_start, target, target_start, len*) [C Function]
Copy *len* bytes from *source* into *target*, starting reading from *source-start* (a positive index within *source*) and start writing at *target-start*. It is permitted for the *source* and *target* regions to overlap.

`bytevector-copy` *bv* [Scheme Procedure]
`scm_bytevector_copy` (*bv*) [C Function]
Return a newly allocated copy of *bv*.

`scm_t_uint8 scm_c_bytevector_ref` (*SCM bv*, *size_t index*) [C Function]
Return the byte at *index* in bytevector *bv*.

`void scm_c_bytevector_set_x` (*SCM bv*, *size_t index*, [C Function]
scm_t_uint8 value)
Set the byte at *index* in *bv* to *value*.

Low-level C macros are available. They do not perform any type-checking; as such they should be used with care.

`size_t SCM_BYTEVECTOR_LENGTH` (*bv*) [C Macro]
Return the length in bytes of bytevector *bv*.

`signed char * SCM_BYTEVECTOR_CONTENTS` (*bv*) [C Macro]
Return a pointer to the contents of bytevector *bv*.

6.6.12.3 Interpreting Bytevector Contents as Integers

The contents of a bytevector can be interpreted as a sequence of integers of any given size, sign, and endianness.

```
(let ((bv (make-bytevector 4)))
  (bytevector-u8-set! bv 0 #x12)
  (bytevector-u8-set! bv 1 #x34)
  (bytevector-u8-set! bv 2 #x56)
  (bytevector-u8-set! bv 3 #x78)

  (map (lambda (number)
         (number->string number 16))
       (list (bytevector-u8-ref bv 0)
             (bytevector-u16-ref bv 0 (endianness big))
             (bytevector-u32-ref bv 0 (endianness little)))))
```

⇒ `("12" "1234" "78563412")`

The most generic procedures to interpret bytevector contents as integers are described below.

`bytevector-uint-ref` *bv index endianness size* [Scheme Procedure]
`scm_bytevector_uint_ref` (*bv*, *index*, *endianness*, *size*) [C Function]
Return the *size*-byte long unsigned integer at index *index* in *bv*, decoded according to *endianness*.

`bytevector-sint-ref` *bv index endianness size* [Scheme Procedure]
`scm_bytevector_sint_ref` (*bv*, *index*, *endianness*, *size*) [C Function]
Return the *size*-byte long signed integer at index *index* in *bv*, decoded according to *endianness*.

`bytevector-uint-set!` *bv index value endianness size* [Scheme Procedure]
`scm_bytevector_uint_set_x` (*bv*, *index*, *value*, *endianness*, *size*) [C Function]
Set the *size*-byte long unsigned integer at *index* to *value*, encoded according to *endianness*.

`bytevector-sint-set!` *bv index value endianness size* [Scheme Procedure]
`scm_bytevector_sint_set_x` (*bv*, *index*, *value*, *endianness*, *size*) [C Function]
Set the *size*-byte long signed integer at *index* to *value*, encoded according to *endianness*.

The following procedures are similar to the ones above, but specialized to a given integer size:

`bytevector-u8-ref` *bv index* [Scheme Procedure]
`bytevector-s8-ref` *bv index* [Scheme Procedure]
`bytevector-u16-ref` *bv index endianness* [Scheme Procedure]
`bytevector-s16-ref` *bv index endianness* [Scheme Procedure]
`bytevector-u32-ref` *bv index endianness* [Scheme Procedure]
`bytevector-s32-ref` *bv index endianness* [Scheme Procedure]
`bytevector-u64-ref` *bv index endianness* [Scheme Procedure]
`bytevector-s64-ref` *bv index endianness* [Scheme Procedure]
`scm_bytevector_u8_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s8_ref` (*bv*, *index*) [C Function]
`scm_bytevector_u16_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s16_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_u32_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s32_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_u64_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_s64_ref` (*bv*, *index*, *endianness*) [C Function]
Return the unsigned *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) from *bv* at *index*, decoded according to *endianness*.

`bytevector-u8-set!` *bv index value* [Scheme Procedure]
`bytevector-s8-set!` *bv index value* [Scheme Procedure]
`bytevector-u16-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s16-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-u32-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s32-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-u64-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-s64-set!` *bv index value endianness* [Scheme Procedure]
`scm_bytevector_u8_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s8_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u16_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s16_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_u32_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s32_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_u64_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_s64_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
Store *value* as an *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) in *bv* at *index*, encoded according to *endianness*.

Finally, a variant specialized for the host's endianness is available for each of these functions (with the exception of the `u8` and `s8` accessors, as endianness is about byte order and there is only 1 byte):

`bytevector-u16-native-ref` *bv index* [Scheme Procedure]
`bytevector-s16-native-ref` *bv index* [Scheme Procedure]
`bytevector-u32-native-ref` *bv index* [Scheme Procedure]
`bytevector-s32-native-ref` *bv index* [Scheme Procedure]
`bytevector-u64-native-ref` *bv index* [Scheme Procedure]
`bytevector-s64-native-ref` *bv index* [Scheme Procedure]
`scm_bytevector_u16_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s16_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_u32_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s32_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_u64_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_s64_native_ref` (*bv*, *index*) [C Function]

Return the unsigned *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) from *bv* at *index*, decoded according to the host's native endianness.

`bytevector-u16-native-set!` *bv index value* [Scheme Procedure]
`bytevector-s16-native-set!` *bv index value* [Scheme Procedure]
`bytevector-u32-native-set!` *bv index value* [Scheme Procedure]
`bytevector-s32-native-set!` *bv index value* [Scheme Procedure]
`bytevector-u64-native-set!` *bv index value* [Scheme Procedure]
`bytevector-s64-native-set!` *bv index value* [Scheme Procedure]
`scm_bytevector_u16_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s16_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u32_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s32_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_u64_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_s64_native_set_x` (*bv*, *index*, *value*) [C Function]

Store *value* as an *n*-bit (signed) integer (where *n* is 8, 16, 32 or 64) in *bv* at *index*, encoded according to the host's native endianness.

6.6.12.4 Converting Bytevectors to/from Integer Lists

Bytevector contents can readily be converted to/from lists of signed or unsigned integers:

```
(bytevector->sint-list (u8-list->bytevector (make-list 4 255))
                       (endianness little) 2)
⇒ (-1 -1)
```

`bytevector->u8-list` *bv* [Scheme Procedure]
`scm_bytevector_to_u8_list` (*bv*) [C Function]

Return a newly allocated list of unsigned 8-bit integers from the contents of *bv*.

`u8-list->bytevector` *lst* [Scheme Procedure]
`scm_u8_list_to_bytevector` (*lst*) [C Function]

Return a newly allocated bytevector consisting of the unsigned 8-bit integers listed in *lst*.

`bytevector->uint-list` *bv endianness size* [Scheme Procedure]
`scm_bytevector_to_uint_list` (*bv*, *endianness*, *size*) [C Function]

Return a list of unsigned integers of *size* bytes representing the contents of *bv*, decoded according to *endianness*.

`bytevector->sint-list` *bv endianness size* [Scheme Procedure]
`scm_bytevector_to_sint_list` (*bv*, *endianness*, *size*) [C Function]
Return a list of signed integers of *size* bytes representing the contents of *bv*, decoded according to *endianness*.

`uint-list->bytevector` *lst endianness size* [Scheme Procedure]
`scm_uint_list_to_bytevector` (*lst*, *endianness*, *size*) [C Function]
Return a new bytevector containing the unsigned integers listed in *lst* and encoded on *size* bytes according to *endianness*.

`sint-list->bytevector` *lst endianness size* [Scheme Procedure]
`scm_sint_list_to_bytevector` (*lst*, *endianness*, *size*) [C Function]
Return a new bytevector containing the signed integers listed in *lst* and encoded on *size* bytes according to *endianness*.

6.6.12.5 Interpreting Bytevector Contents as Floating Point Numbers

Bytevector contents can also be accessed as IEEE-754 single- or double-precision floating point numbers (respectively 32 and 64-bit long) using the procedures described here.

`bytevector-ieee-single-ref` *bv index endianness* [Scheme Procedure]
`bytevector-ieee-double-ref` *bv index endianness* [Scheme Procedure]
`scm_bytevector_ieee_single_ref` (*bv*, *index*, *endianness*) [C Function]
`scm_bytevector_ieee_double_ref` (*bv*, *index*, *endianness*) [C Function]
Return the IEEE-754 single-precision floating point number from *bv* at *index* according to *endianness*.

`bytevector-ieee-single-set!` *bv index value endianness* [Scheme Procedure]
`bytevector-ieee-double-set!` *bv index value endianness* [Scheme Procedure]
`scm_bytevector_ieee_single_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
`scm_bytevector_ieee_double_set_x` (*bv*, *index*, *value*, *endianness*) [C Function]
Store real number *value* in *bv* at *index* according to *endianness*.

Specialized procedures are also available:

`bytevector-ieee-single-native-ref` *bv index* [Scheme Procedure]
`bytevector-ieee-double-native-ref` *bv index* [Scheme Procedure]
`scm_bytevector_ieee_single_native_ref` (*bv*, *index*) [C Function]
`scm_bytevector_ieee_double_native_ref` (*bv*, *index*) [C Function]
Return the IEEE-754 single-precision floating point number from *bv* at *index* according to the host's native endianness.

`bytevector-ieee-single-native-set!` *bv index value* [Scheme Procedure]
`bytevector-ieee-double-native-set!` *bv index value* [Scheme Procedure]
`scm_bytevector_ieee_single_native_set_x` (*bv*, *index*, *value*) [C Function]
`scm_bytevector_ieee_double_native_set_x` (*bv*, *index*, *value*) [C Function]
Store real number *value* in *bv* at *index* according to the host's native endianness.

6.6.12.6 Interpreting Bytevector Contents as Unicode Strings

Bytevector contents can also be interpreted as Unicode strings encoded in one of the most commonly available encoding formats. See Section 6.6.5.13 [Representing Strings as Bytes], page 159, for a more generic interface.

```
(utf8->string (u8-list->bytevector '(99 97 102 101)))
⇒ "cafe"

(string->utf8 "café") ;; SMALL LATIN LETTER E WITH ACUTE ACCENT
⇒ #vu8(99 97 102 195 169)
```

`string-utf8-length` *str* [Scheme Procedure]
`SCM scm_string_utf8_length` (*str*) [C function]
`size_t scm_c_string_utf8_length` (*str*) [C function]
Return the number of bytes in the UTF-8 representation of *str*.

`string->utf8` *str* [Scheme Procedure]
`string->utf16` *str* [*endianness*] [Scheme Procedure]
`string->utf32` *str* [*endianness*] [Scheme Procedure]
`scm_string_to_utf8` (*str*) [C Function]
`scm_string_to_utf16` (*str*, *endianness*) [C Function]
`scm_string_to_utf32` (*str*, *endianness*) [C Function]
Return a newly allocated bytevector that contains the UTF-8, UTF-16, or UTF-32 (aka. UCS-4) encoding of *str*. For UTF-16 and UTF-32, *endianness* should be the symbol `big` or `little`; when omitted, it defaults to big endian.

`utf8->string` *utf* [Scheme Procedure]
`utf16->string` *utf* [*endianness*] [Scheme Procedure]
`utf32->string` *utf* [*endianness*] [Scheme Procedure]
`scm_utf8_to_string` (*utf*) [C Function]
`scm_utf16_to_string` (*utf*, *endianness*) [C Function]
`scm_utf32_to_string` (*utf*, *endianness*) [C Function]
Return a newly allocated string that contains from the UTF-8-, UTF-16-, or UTF-32-decoded contents of bytevector *utf*. For UTF-16 and UTF-32, *endianness* should be the symbol `big` or `little`; when omitted, it defaults to big endian.

6.6.12.7 Accessing Bytevectors with the Array API

As an extension to the R6RS, Guile allows bytevectors to be manipulated with the *array* procedures (see Section 6.6.13 [Arrays], page 200). When using these APIs, bytes are accessed one at a time as 8-bit unsigned integers:

```
(define bv #vu8(0 1 2 3))

(array? bv)
⇒ #t

(array-rank bv)
⇒ 1
```

```
(array-ref bv 2)
⇒ 2

;; Note the different argument order on array-set!.
(array-set! bv 77 2)
(array-ref bv 2)
⇒ 77

(array-type bv)
⇒ vu8
```

6.6.12.8 Accessing Bytevectors with the SRFI-4 API

Bytevectors may also be accessed with the SRFI-4 API. See Section 7.5.5.3 [SRFI-4 and Bytevectors], page 606, for more information.

6.6.13 Arrays

Arrays are a collection of cells organized into an arbitrary number of dimensions. Each cell can be accessed in constant time by supplying an index for each dimension.

In the current implementation, an array uses a vector of some kind for the actual storage of its elements. Any kind of vector will do, so you can have arrays of uniform numeric values, arrays of characters, arrays of bits, and of course, arrays of arbitrary Scheme values. For example, arrays with an underlying `c64vector` might be nice for digital signal processing, while arrays made from a `u8vector` might be used to hold gray-scale images.

The number of dimensions of an array is called its *rank*. Thus, a matrix is an array of rank 2, while a vector has rank 1. When accessing an array element, you have to specify one exact integer for each dimension. These integers are called the *indices* of the element. An array specifies the allowed range of indices for each dimension via an inclusive lower and upper bound. These bounds can well be negative, but the upper bound must be greater than or equal to the lower bound minus one. When all lower bounds of an array are zero, it is called a *zero-origin* array.

Arrays can be of rank 0, which could be interpreted as a scalar. Thus, a zero-rank array can store exactly one object and the list of indices of this element is the empty list.

Arrays contain zero elements when one of their dimensions has a zero length. These empty arrays maintain information about their shape: a matrix with zero columns and 3 rows is different from a matrix with 3 columns and zero rows, which again is different from a vector of length zero.

The array procedures are all polymorphic, treating strings, uniform numeric vectors, bytevectors, bit vectors and ordinary vectors as one dimensional arrays.

6.6.13.1 Array Syntax

An array is displayed as `#` followed by its rank, followed by a tag that describes the underlying vector, optionally followed by information about its shape, and finally followed by the cells, organized into dimensions using parentheses.

In more words, the array tag is of the form

```
#<rank><vectag><@lower><:len><@lower><:len>...
```

where `<rank>` is a positive integer in decimal giving the rank of the array. It is omitted when the rank is 1 and the array is non-shared and has zero-origin (see below). For shared arrays and for a non-zero origin, the rank is always printed even when it is 1 to distinguish them from ordinary vectors.

The `<vectag>` part is the tag for a uniform numeric vector, like `u8`, `s16`, etc, `b` for bitvectors, or `a` for strings. It is empty for ordinary vectors.

The `<@lower>` part is a '`@`' character followed by a signed integer in decimal giving the lower bound of a dimension. There is one `<@lower>` for each dimension. When all lower bounds are zero, all `<@lower>` parts are omitted.

The `<:len>` part is a '`:`' character followed by an unsigned integer in decimal giving the length of a dimension. Like for the lower bounds, there is one `<:len>` for each dimension, and the `<:len>` part always follows the `<@lower>` part for a dimension. Lengths are only then printed when they can't be deduced from the nested lists of elements of the array literal, which can happen when at least one length is zero.

As a special case, an array of rank 0 is printed as `#0<vectag>(<scalar>)`, where `<scalar>` is the result of printing the single element of the array.

Thus,

`#(1 2 3)` is an ordinary array of rank 1 with lower bound 0 in dimension 0. (I.e., a regular vector.)

`#@2(1 2 3)`
: is an ordinary array of rank 1 with lower bound 2 in dimension 0.

`#2((1 2 3) (4 5 6))`
: is a non-uniform array of rank 2; a 2×3 matrix with index ranges 0..1 and 0..2.

`#u8(0 1 2)`
: is a uniform u8 array of rank 1.

`#2u32@2@3((1 2) (2 3))`
: is a uniform u32 array of rank 2 with index ranges 2..3 and 3..4.

`#2()` is a two-dimensional array with index ranges 0..-1 and 0..-1, i.e. both dimensions have length zero.

`#2:0:2()` is a two-dimensional array with index ranges 0..-1 and 0..1, i.e. the first dimension has length zero, but the second has length 2.

`#0(12)` is a rank-zero array with contents 12.

In addition, bytevectors are also arrays, but use a different syntax (see Section 6.6.12 [Bytevectors], page 193):

`#vu8(1 2 3)`
: is a 3-byte long bytevector, with contents 1, 2, 3.

6.6.13.2 Array Procedures

When an array is created, the range of each dimension must be specified, e.g., to create a 2×3 array with a zero-based index:

```
(make-array 'ho 2 3) ⇒ #2((ho ho ho) (ho ho ho))
```

The range of each dimension can also be given explicitly, e.g., another way to create the same array:

```
(make-array 'ho '(0 1) '(0 2)) ⇒ #2((ho ho ho) (ho ho ho))
```

The following procedures can be used with arrays (or vectors). An argument shown as *idx*... means one parameter for each dimension in the array. A *idxlist* argument means a list of such values, one for each dimension.

`array?` *obj* [Scheme Procedure]
`scm_array_p` (*obj, unused*) [C Function]
Return `#t` if the *obj* is an array, and `#f` if not.

The second argument to scm_array_p is there for historical reasons, but it is not used. You should always pass `SCM_UNDEFINED` as its value.

`typed-array?` *obj type* [Scheme Procedure]
`scm_typed_array_p` (*obj, type*) [C Function]
Return `#t` if the *obj* is an array of type *type*, and `#f` if not.

`int scm_is_array` (*SCM obj*) [C Function]
Return `1` if the *obj* is an array and `0` if not.

`int scm_is_typed_array` (*SCM obj, SCM type*) [C Function]
Return `0` if the *obj* is an array of type *type*, and `1` if not.

`make-array` *fill bound ...* [Scheme Procedure]
`scm_make_array` (*fill, bounds*) [C Function]
Equivalent to `(make-typed-array #t` *`fill bound ...`*`)`.

`make-typed-array` *type fill bound ...* [Scheme Procedure]
`scm_make_typed_array` (*type, fill, bounds*) [C Function]
Create and return an array that has as many dimensions as there are *bound*s and (maybe) fill it with *fill*.

The underlying storage vector is created according to *type*, which must be a symbol whose name is the 'vectag' of the array as explained above, or `#t` for ordinary, non-specialized arrays.

For example, using the symbol `f64` for *type* will create an array that uses a `f64vector` for storing its elements, and `a` will use a string.

When *fill* is not the special *unspecified* value, the new array is filled with *fill*. Otherwise, the initial contents of the array is unspecified. The special *unspecified* value is stored in the variable `*unspecified*` so that for example `(make-typed-array 'u32 *unspecified* 4)` creates a uninitialized `u32` vector of length 4.

Each *bound* may be a positive non-zero integer *n*, in which case the index for that dimension can range from 0 through *n*-1; or an explicit index range specifier in the form `(LOWER UPPER)`, where both *lower* and *upper* are integers, possibly less than zero, and possibly the same number (however, *lower* cannot be greater than *upper*).

`list->array` *dimspec list* [Scheme Procedure]
Equivalent to `(list->typed-array #t` *`dimspec list`*`)`.

`list->typed-array` *type dimspec list* [Scheme Procedure]
`scm_list_to_typed_array` (*type, dimspec, list*) [C Function]

Return an array of the type indicated by *type* with elements the same as those of *list*.

The argument *dimspec* determines the number of dimensions of the array and their lower bounds. When *dimspec* is an exact integer, it gives the number of dimensions directly and all lower bounds are zero. When it is a list of exact integers, then each element is the lower index bound of a dimension, and there will be as many dimensions as elements in the list.

`array-type` *array* [Scheme Procedure]
`scm_array_type` (*array*) [C Function]

Return the type of *array*. This is the 'vectag' used for printing *array* (or `#t` for ordinary arrays) and can be used with `make-typed-array` to create an array of the same kind as *array*.

`array-ref` *array idx ...* [Scheme Procedure]
`scm_array_ref` (*array, idxlist*) [C Function]

Return the element at `(idx ...)` in *array*.

```
(define a (make-array 999 '(1 2) '(3 4)))
(array-ref a 2 4) ⇒ 999
```

`array-in-bounds?` *array idx ...* [Scheme Procedure]
`scm_array_in_bounds_p` (*array, idxlist*) [C Function]

Return `#t` if the given indices would be acceptable to `array-ref`.

```
(define a (make-array #f '(1 2) '(3 4)))
(array-in-bounds? a 2 3) ⇒ #t
(array-in-bounds? a 0 0) ⇒ #f
```

`array-set!` *array obj idx ...* [Scheme Procedure]
`scm_array_set_x` (*array, obj, idxlist*) [C Function]

Set the element at `(idx ...)` in *array* to *obj*. The return value is unspecified.

```
(define a (make-array #f '(0 1) '(0 1)))
(array-set! a #t 1 1)
a ⇒ #2((#f #f) (#f #t))
```

`array-shape` *array* [Scheme Procedure]
`array-dimensions` *array* [Scheme Procedure]
`scm_array_dimensions` (*array*) [C Function]

Return a list of the bounds for each dimension of *array*.

`array-shape` gives `(`*lower upper*`)` for each dimension. `array-dimensions` instead returns just *upper* + 1 for dimensions with a 0 lower bound. Both are suitable as input to `make-array`.

For example,

```
(define a (make-array 'foo '(-1 3) 5))
(array-shape a)      ⇒ ((-1 3) (0 4))
(array-dimensions a) ⇒ ((-1 3) 5)
```

`array-length` *array* [Scheme Procedure]
`scm_array_length` (*array*) [C Function]
`size_t` *scm_c_array_length* (*array*) [C Function]
Return the length of an array: its first dimension. It is an error to ask for the length of an array of rank 0.

`array-rank` *array* [Scheme Procedure]
`scm_array_rank` (*array*) [C Function]
Return the rank of *array*.

`size_t scm_c_array_rank` (*SCM array*) [C Function]
Return the rank of *array* as a `size_t`.

`array->list` *array* [Scheme Procedure]
`scm_array_to_list` (*array*) [C Function]
Return a list consisting of all the elements, in order, of *array*.

`array-copy!` *src dst* [Scheme Procedure]
`array-copy-in-order!` *src dst* [Scheme Procedure]
`scm_array_copy_x` (*src*, *dst*) [C Function]
Copy every element from vector or array *src* to the corresponding element of *dst*. *dst* must have the same rank as *src*, and be at least as large in each dimension. The return value is unspecified.

`array-fill!` *array fill* [Scheme Procedure]
`scm_array_fill_x` (*array*, *fill*) [C Function]
Store *fill* in every element of *array*. The value returned is unspecified.

`array-equal?` *array ...* [Scheme Procedure]
Return `#t` if all arguments are arrays with the same shape, the same type, and have corresponding elements which are either `equal?` or `array-equal?`. This function differs from `equal?` (see Section 6.11.1 [Equality], page 283) in that all arguments must be arrays.

`array-map!` *dst proc src ...* [Scheme Procedure]
`array-map-in-order!` *dst proc src ...* [Scheme Procedure]
`scm_array_map_x` (*dst*, *proc*, *srclist*) [C Function]
Set each element of the *dst* array to values obtained from calls to *proc*. The list of *src* arguments may be empty. The value returned is unspecified.

Each call is (*proc elem ...*), where each *elem* is from the corresponding *src* array, at the *dst* index. `array-map-in-order!` makes the calls in row-major order, `array-map!` makes them in an unspecified order.

The *src* arrays must have the same number of dimensions as *dst*, and must have a range for each dimension which covers the range in *dst*. This ensures all *dst* indices are valid in each *src*.

`array-for-each` *proc src1 src2 ...* [Scheme Procedure]
`scm_array_for_each` (*proc*, *src1*, *srclist*) [C Function]
Apply *proc* to each tuple of elements of *src1 src2 ...*, in row-major order. The value returned is unspecified.

`array-index-map!` *dst proc* [Scheme Procedure]
`scm_array_index_map_x` (*dst, proc*) [C Function]

Set each element of the *dst* array to values returned by calls to *proc*. The value returned is unspecified.

Each call is (*proc i1 ... iN*), where *i1*...*iN* is the destination index, one parameter for each dimension. The order in which the calls are made is unspecified.

For example, to create a 4×4 matrix representing a cyclic group,

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

```
(define a (make-array #f 4 4))
(array-index-map! a (lambda (i j)
                      (modulo (+ i j) 4)))
```

An additional array function is available in the module `(ice-9 arrays)`. It can be used with:

```
(use-modules (ice-9 arrays))
```

`array-copy` *src* [Scheme Procedure]

Return a new array with the same elements, type and shape as *src*. However, the array increments may not be the same as those of *src*. In the current implementation, the returned array will be in row-major order, but that might change in the future. Use `array-copy!` on an array of known order if that is a concern.

6.6.13.3 Shared Arrays

`make-shared-array` *oldarray mapfunc bound ...* [Scheme Procedure]
`scm_make_shared_array` (*oldarray, mapfunc, boundlist*) [C Function]

Return a new array which shares the storage of *oldarray*. Changes made through either affect the same underlying storage. The *bound ...* arguments are the shape of the new array, the same as `make-array` (see Section 6.6.13.2 [Array Procedures], page 201).

mapfunc translates coordinates from the new array to the *oldarray*. It's called as (*mapfunc* `newidx1 ...`) with one parameter for each dimension of the new array, and should return a list of indices for *oldarray*, one for each dimension of *oldarray*.

mapfunc must be affine linear, meaning that each *oldarray* index must be formed by adding integer multiples (possibly negative) of some or all of *newidx1* etc, plus a possible integer offset. The multiples and offset must be the same in each call.

One good use for a shared array is to restrict the range of some dimensions, so as to apply say `array-for-each` or `array-fill!` to only part of an array. The plain `list` function can be used for *mapfunc* in this case, making no changes to the index values. For example,

```
(make-shared-array #2((a b c) (d e f) (g h i)) list 3 2)
⇒ #2((a b) (d e) (g h))
```

The new array can have fewer dimensions than *oldarray*, for example to take a column from an array.

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i) (list i 2))
                   '(0 2))
⇒ #1(c f i)
```

A diagonal can be taken by using the single new array index for both row and column in the old array. For example,

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i) (list i i))
                   '(0 2))
⇒ #1(a e i)
```

Dimensions can be increased by for instance considering portions of a one dimensional array as rows in a two dimensional array. (`array-contents` below can do the opposite, flattening an array.)

```
(make-shared-array #1(a b c d e f g h i j k l)
                   (lambda (i j) (list (+ (* i 3) j)))
                   4 3)
⇒ #2((a b c) (d e f) (g h i) (j k l))
```

By negating an index the order that elements appear can be reversed. The following just reverses the column order,

```
(make-shared-array #2((a b c) (d e f) (g h i))
                   (lambda (i j) (list i (- 2 j)))
                   3 3)
⇒ #2((c b a) (f e d) (i h g))
```

A fixed offset on indexes allows for instance a change from a 0 based to a 1 based array,

```
(define x #2((a b c) (d e f) (g h i)))
(define y (make-shared-array x
                             (lambda (i j) (list (1- i) (1- j)))
                             '(1 3) '(1 3)))
(array-ref x 0 0) ⇒ a
(array-ref y 1 1) ⇒ a
```

A multiple on an index allows every Nth element of an array to be taken. The following is every third element,

```
(make-shared-array #1(a b c d e f g h i j k l)
                   (lambda (i) (list (* i 3)))
                   4)
⇒ #1(a d g j)
```

The above examples can be combined to make weird and wonderful selections from an array, but it's important to note that because *mapfunc* must be affine linear, arbitrary permutations are not possible.

In the current implementation, *mapfunc* is not called for every access to the new array but only on some sample points to establish a base and stride for new array indices in *oldarray* data. A few sample points are enough because *mapfunc* is linear.

`shared-array-increments` *array* [Scheme Procedure]
`scm_shared_array_increments` (*array*) [C Function]
For each dimension, return the distance between elements in the root vector.

`shared-array-offset` *array* [Scheme Procedure]
`scm_shared_array_offset` (*array*) [C Function]
Return the root vector index of the first element in the array.

`shared-array-root` *array* [Scheme Procedure]
`scm_shared_array_root` (*array*) [C Function]
Return the root vector of a shared array.

`array-contents` *array* [*strict*] [Scheme Procedure]
`scm_array_contents` (*array*, *strict*) [C Function]
If *array* may be *unrolled* into a one dimensional shared array without changing their order (last subscript changing fastest), then `array-contents` returns that shared array, otherwise it returns `#f`. All arrays made by `make-array` and `make-typed-array` may be unrolled, some arrays made by `make-shared-array` may not be.

If the optional argument *strict* is provided, a shared array will be returned only if its elements are stored internally contiguous in memory.

`transpose-array` *array dim1 dim2 ...* [Scheme Procedure]
`scm_transpose_array` (*array*, *dimlist*) [C Function]
Return an array sharing contents with *array*, but with dimensions arranged in a different order. There must be one *dim* argument for each dimension of *array*. *dim1*, *dim2*, ... should be integers between 0 and the rank of the array to be returned. Each integer in that range must appear at least once in the argument list.

The values of *dim1*, *dim2*, ... correspond to dimensions in the array to be returned, and their positions in the argument list to dimensions of *array*. Several *dim*s may have the same value, in which case the returned array will have smaller rank than *array*.

```
(transpose-array '#2((a b) (c d)) 1 0) ⇒ #2((a c) (b d))
(transpose-array '#2((a b) (c d)) 0 0) ⇒ #1(a d)
(transpose-array '#3(((a b c) (d e f)) ((1 2 3) (4 5 6))) 1 1 0) ⇒
                 #2((a 4) (b 5) (c 6))
```

6.6.13.4 Arrays as arrays of arrays

Mathematically, one can see an array of rank n (an n-array) as an array of lower rank where the elements are themselves arrays ('cells').

We speak of the first $n-k$ dimensions of the array as the $n-k$-'frame' of the array, while the last k dimensions are the dimensions of the k-'cells'. For example, a 3-array can be seen as a 2-array of vectors (1-arrays) or as a 1-array of matrices (2-arrays). In each case, the vectors or matrices are the 1-cells or 2-cells of the array. This terminology originates in the J language.

The more vague concept of a 'slice' refers to a subset of the array where some indices are fixed and others are left free. As a Guile data object, a cell is the same as a 'prefix slice' (the first $n-k$ indices into the original array are fixed), except that a 0-cell is not a

shared array of the original array, but a 0-slice (where all the indices into the original array are fixed) is.

Before version 2.0, Guile had a feature called 'enclosed arrays' to create special 'array of arrays' objects. The functions in this section do not need special types; instead, the frame rank is stated in each function call, either implicitly or explicitly.

`array-cell-ref` *array idx ...* [Scheme Procedure]
`scm_array_cell_ref` (*array, idxlist*) [C Function]

If the length of *idxlist* equals the rank n of *array*, return the element at `(idx ...)`, just like `(array-ref array idx ...)`. If, however, the length k of *idxlist* is smaller than n, then return the $(n-k)$-cell of *array* given by *idxlist*, as a shared array.

For example:

```
(array-cell-ref #2((a b) (c d)) 0) ⇒ #(a b)
(array-cell-ref #2((a b) (c d)) 1) ⇒ #(c d)
(array-cell-ref #2((a b) (c d)) 1 1) ⇒ d
(array-cell-ref #2((a b) (c d))) ⇒ #2((a b) (c d))
```

`(apply array-cell-ref array indices)` is equivalent to

```
(let ((len (length indices)))
  (if (= (array-rank a) len)
    (apply array-ref a indices)
    (apply make-shared-array a
           (lambda t (append indices t))
           (drop (array-dimensions a) len))))
```

`array-slice` *array idx ...* [Scheme Procedure]
`scm_array_slice` (*array, idxlist*) [C Function]

Like `(array-cell-ref array idx ...)`, but return a 0-rank shared array into *ARRAY* if the length of *idxlist* matches the rank of *array*. This can be useful when using *ARRAY* as a place to write to.

Compare:

```
(array-cell-ref #2((a b) (c d)) 1 1) ⇒ d
(array-slice #2((a b) (c d)) 1 1) ⇒ #0(d)
(define a (make-array 'a 2 2))
(array-fill! (array-slice a 1 1) 'b)
a ⇒ #2((a a) (a b)).
(array-fill! (array-cell-ref a 1 1) 'b) ⇒ error: not an array
```

`(apply array-slice array indices)` is equivalent to

```
(apply make-shared-array a
  (lambda t (append indices t))
  (drop (array-dimensions a) (length indices)))
```

`array-cell-set!` *array x idx ...* [Scheme Procedure]
`scm_array_cell_set_x` (*array, x, idxlist*) [C Function]

If the length of *idxlist* equals the rank n of *array*, set the element at `(idx ...)` of *array* to *x*, just like `(array-set! array x idx ...)`. If, however, the length k of *idxlist* is smaller than n, then copy the $(n-k)$-rank array *x* into the $(n-k)$-cell

of *array* given by *idxlist*. In this case, the last $(n - k)$ dimensions of *array* and the dimensions of *x* must match exactly.

This function returns the modified *array*.

For example:

```
(array-cell-set! (make-array 'a 2 2) b 1 1)
  ⇒ #2((a a) (a b))
(array-cell-set! (make-array 'a 2 2) #(x y) 1)
  ⇒ #2((a a) (x y))
```

Note that `array-cell-set!` will expect elements, not arrays, when the destination has rank 0. Use `array-slice` for the opposite behavior.

```
(array-cell-set! (make-array 'a 2 2) #0(b) 1 1)
  ⇒ #2((a a) (a #0(b)))
(let ((a (make-array 'a 2 2)))
  (array-copy! #0(b) (array-slice a 1 1)) a)
  ⇒ #2((a a) (a b))
```

`(apply array-cell-set! array x indices)` is equivalent to

```
(let ((len (length indices)))
  (if (= (array-rank array) len)
    (apply array-set! array x indices)
    (array-copy! x (apply array-cell-ref array indices)))
  array)
```

`array-slice-for-each` *frame-rank op x . . .* [Scheme Procedure]
`scm_array_slice_for_each` (*array, frame_rank, op, xlist*) [C Function]

Each *x* must be an array of rank ≥ *frame-rank*, and the first *frame-rank* dimensions of each *x* must all be the same. *array-slice-for-each* calls *op* with each set of (rank(*x*) - *frame-rank*)-cells from *x*, in unspecified order.

array-slice-for-each allows you to loop over cells of any rank without having to carry an index list or construct shared arrays manually. The slices passed to *op* are always shared arrays of *X*, even if they are of rank 0, so it is possible to write to them.

This function returns an unspecified value.

For example, to sort the rows of rank-2 array `a`:

```
(array-slice-for-each 1 (lambda (x) (sort! x <)) a)
```

As another example, let `a` be a rank-2 array where each row is a 2-element vector (x, y). Let's compute the arguments of these vectors and store them in rank-1 array `b`.

```
(array-slice-for-each 1
  (lambda (a b)
    (array-set! b (atan (array-ref a 1) (array-ref a 0))))
  a b)
```

`(apply array-slice-for-each frame-rank op x)` is equivalent to

```
(let ((frame (take (array-dimensions (car x)) frank)))
  (unless (every (lambda (x)
                   (equal? frame (take (array-dimensions x) frank)))
```

```
                    (cdr x))
          (error))
        (array-index-map!
          (apply make-shared-array (make-array #t) (const '()) frame)
          (lambda i (apply op (map (lambda (x) (apply array-slice x i)) x)))))
```

`array-slice-for-each-in-order` *frame-rank op x ...* [Scheme Procedure]
`scm_array_slice_for_each_in_order` (*array, frame_rank, op, xlist*) [C Function]

Same as `array-slice-for-each`, but the arguments are traversed sequentially and in row-major order.

6.6.13.5 Accessing Arrays from C

For interworking with external C code, Guile provides an API to allow C code to access the elements of a Scheme array. In particular, for uniform numeric arrays, the API exposes the underlying uniform data as a C array of numbers of the relevant type.

While pointers to the elements of an array are in use, the array itself must be protected so that the pointer remains valid. Such a protected array is said to be *reserved*. A reserved array can be read but modifications to it that would cause the pointer to its elements to become invalid are prevented. When you attempt such a modification, an error is signalled.

(This is similar to locking the array while it is in use, but without the danger of a deadlock. In a multi-threaded program, you will need additional synchronization to avoid modifying reserved arrays.)

You must take care to always unreserve an array after reserving it, even in the presence of non-local exits. If a non-local exit can happen between these two calls, you should install a dynwind context that releases the array when it is left (see Section 6.13.10 [Dynamic Wind], page 320).

In addition, array reserving and unreserving must be properly paired. For instance, when reserving two or more arrays in a certain order, you need to unreserve them in the opposite order.

Once you have reserved an array and have retrieved the pointer to its elements, you must figure out the layout of the elements in memory. Guile allows slices to be taken out of arrays without actually making a copy, such as making an alias for the diagonal of a matrix that can be treated as a vector. Arrays that result from such an operation are not stored contiguously in memory and when working with their elements directly, you need to take this into account.

The layout of array elements in memory can be defined via a *mapping function* that computes a scalar position from a vector of indices. The scalar position then is the offset of the element with the given indices from the start of the storage block of the array.

In Guile, this mapping function is restricted to be *affine*: all mapping functions of Guile arrays can be written as `p = b + c[0]*i[0] + c[1]*i[1] + ... + c[n-1]*i[n-1]` where `i[k]` is the `k`th index and `n` is the rank of the array. For example, a matrix of size 3x3 would have `b == 0`, `c[0] == 3` and `c[1] == 1`. When you transpose this matrix (with `transpose-array`, say), you will get an array whose mapping function has `b == 0`, `c[0] == 1` and `c[1] == 3`.

The function `scm_array_handle_dims` gives you (indirect) access to the coefficients `c[k]`.

Note that there are no functions for accessing the elements of a character array yet. Once the string implementation of Guile has been changed to use Unicode, we will provide them.

`scm_t_array_handle` [C Type]

This is a structure type that holds all information necessary to manage the reservation of arrays as explained above. Structures of this type must be allocated on the stack and must only be accessed by the functions listed below.

`void scm_array_get_handle` (*SCM array, scm_t_array_handle *handle*) [C Function]

Reserve *array*, which must be an array, and prepare *handle* to be used with the functions below. You must eventually call `scm_array_handle_release` on *handle*, and do this in a properly nested fashion, as explained above. The structure pointed to by *handle* does not need to be initialized before calling this function.

`void scm_array_handle_release` (*scm_t_array_handle *handle*) [C Function]

End the array reservation represented by *handle*. After a call to this function, *handle* might be used for another reservation.

`size_t scm_array_handle_rank` (*scm_t_array_handle *handle*) [C Function]

Return the rank of the array represented by *handle*.

`scm_t_array_dim` [C Type]

This structure type holds information about the layout of one dimension of an array. It includes the following fields:

`ssize_t lbnd`
`ssize_t ubnd`

The lower and upper bounds (both inclusive) of the permissible index range for the given dimension. Both values can be negative, but *lbnd* is always less than or equal to *ubnd*.

`ssize_t inc`

The distance from one element of this dimension to the next. Note, too, that this can be negative.

`const scm_t_array_dim * scm_array_handle_dims` (*scm_t_array_handle *handle*) [C Function]

Return a pointer to a C vector of information about the dimensions of the array represented by *handle*. This pointer is valid as long as the array remains reserved. As explained above, the `scm_t_array_dim` structures returned by this function can be used calculate the position of an element in the storage block of the array from its indices.

This position can then be used as an index into the C array pointer returned by the various `scm_array_handle_<foo>_elements` functions, or with `scm_array_handle_ref` and `scm_array_handle_set`.

Here is how one can compute the position *pos* of an element given its indices in the vector *indices*:

```
ssize_t indices[RANK];
scm_t_array_dim *dims;
ssize_t pos;
size_t i;

pos = 0;
for (i = 0; i < RANK; i++)
  {
    if (indices[i] < dims[i].lbnd || indices[i] > dims[i].ubnd)
      out_of_range ();
    pos += (indices[i] - dims[i].lbnd) * dims[i].inc;
  }
```

`ssize_t scm_array_handle_pos` (*scm_t_array_handle *handle, SCM indices*) [C Function]

Compute the position corresponding to *indices*, a list of indices. The position is computed as described above for `scm_array_handle_dims`. The number of the indices and their range is checked and an appropriate error is signalled for invalid indices.

`SCM scm_array_handle_ref` (*scm_t_array_handle *handle, ssize_t pos*) [C Function]

Return the element at position *pos* in the storage block of the array represented by *handle*. Any kind of array is acceptable. No range checking is done on *pos*.

`void scm_array_handle_set` (*scm_t_array_handle *handle, ssize_t pos, SCM val*) [C Function]

Set the element at position *pos* in the storage block of the array represented by *handle* to *val*. Any kind of array is acceptable. No range checking is done on *pos*. An error is signalled when the array can not store *val*.

`const SCM * scm_array_handle_elements` (*scm_t_array_handle *handle*) [C Function]

Return a pointer to the elements of a ordinary array of general Scheme values (i.e., a non-uniform array) for reading. This pointer is valid as long as the array remains reserved.

`SCM * scm_array_handle_writable_elements` (*scm_t_array_handle *handle*) [C Function]

Like `scm_array_handle_elements`, but the pointer is good for reading and writing.

`const void * scm_array_handle_uniform_elements` (*scm_t_array_handle *handle*) [C Function]

Return a pointer to the elements of a uniform numeric array for reading. This pointer is valid as long as the array remains reserved. The size of each element is given by `scm_array_handle_uniform_element_size`.

`void * scm_array_handle_uniform_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_uniform_elements`, but the pointer is good reading and writing.

`size_t scm_array_handle_uniform_element_size` [C Function]
(*scm_t_array_handle *handle*)

Return the size of one element of the uniform numeric array represented by *handle*.

`const scm_t_uint8 * scm_array_handle_u8_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int8 * scm_array_handle_s8_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_uint16 * scm_array_handle_u16_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int16 * scm_array_handle_s16_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_uint32 * scm_array_handle_u32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int32 * scm_array_handle_s32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_uint64 * scm_array_handle_u64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const scm_t_int64 * scm_array_handle_s64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const float * scm_array_handle_f32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const double * scm_array_handle_f64_elements` [C Function]
(*scm_t_array_handle *handle*)
`const float * scm_array_handle_c32_elements` [C Function]
(*scm_t_array_handle *handle*)
`const double * scm_array_handle_c64_elements` [C Function]
(*scm_t_array_handle *handle*)

Return a pointer to the elements of a uniform numeric array of the indicated kind for reading. This pointer is valid as long as the array remains reserved.

The pointers for `c32` and `c64` uniform numeric arrays point to pairs of floating point numbers. The even index holds the real part, the odd index the imaginary part of the complex number.

`scm_t_uint8 * scm_array_handle_u8_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int8 * scm_array_handle_s8_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint16 * scm_array_handle_u16_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int16 * scm_array_handle_s16_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

`scm_t_uint32 * scm_array_handle_u32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int32 * scm_array_handle_s32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_uint64 * scm_array_handle_u64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`scm_t_int64 * scm_array_handle_s64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`float * scm_array_handle_f32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`double * scm_array_handle_f64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`float * scm_array_handle_c32_writable_elements` [C Function]
(*scm_t_array_handle *handle*)
`double * scm_array_handle_c64_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_<kind>_elements`, but the pointer is good for reading and writing.

`const scm_t_uint32 * scm_array_handle_bit_elements` [C Function]
(*scm_t_array_handle *handle*)

Return a pointer to the words that store the bits of the represented array, which must be a bit array.

Unlike other arrays, bit arrays have an additional offset that must be figured into index calculations. That offset is returned by `scm_array_handle_bit_elements_offset`.

To find a certain bit you first need to calculate its position as explained above for `scm_array_handle_dims` and then add the offset. This gives the absolute position of the bit, which is always a non-negative integer.

Each word of the bit array storage block contains exactly 32 bits, with the least significant bit in that word having the lowest absolute position number. The next word contains the next 32 bits.

Thus, the following code can be used to access a bit whose position according to `scm_array_handle_dims` is given in *pos*:

```
SCM bit_array;
scm_t_array_handle handle;
scm_t_uint32 *bits;
ssize_t pos;
size_t abs_pos;
size_t word_pos, mask;

scm_array_get_handle (&bit_array, &handle);
bits = scm_array_handle_bit_elements (&handle);

pos = ...
abs_pos = pos + scm_array_handle_bit_elements_offset (&handle);
word_pos = abs_pos / 32;
```

```
mask = 1L << (abs_pos % 32);

if (bits[word_pos] & mask)
  /* bit is set. */

scm_array_handle_release (&handle);
```

`scm_t_uint32 * scm_array_handle_bit_writable_elements` [C Function]
(*scm_t_array_handle *handle*)

Like `scm_array_handle_bit_elements` but the pointer is good for reading and writing. You must take care not to modify bits outside of the allowed index range of the array, even for contiguous arrays.

6.6.14 VLists

The `(ice-9 vlist)` module provides an implementation of the *VList* data structure designed by Phil Bagwell in 2002. VLists are immutable lists, which can contain any Scheme object. They improve on standard Scheme linked lists in several areas:

- Random access has typically constant-time complexity.
- Computing the length of a VList has time complexity logarithmic in the number of elements.
- VLists use less storage space than standard lists.
- VList elements are stored in contiguous regions, which improves memory locality and leads to more efficient use of hardware caches.

The idea behind VLists is to store vlist elements in increasingly large contiguous blocks (implemented as vectors here). These blocks are linked to one another using a pointer to the next block and an offset within that block. The size of these blocks form a geometric series with ratio `block-growth-factor` (2 by default).

The VList structure also serves as the basis for the *VList-based hash lists* or "vhashes", an immutable dictionary type (see Section 6.6.21 [VHashes], page 236).

However, the current implementation in `(ice-9 vlist)` has several noteworthy shortcomings:

- It is *not* thread-safe. Although operations on vlists are all *referentially transparent* (i.e., purely functional), adding elements to a vlist with `vlist-cons` mutates part of its internal structure, which makes it non-thread-safe. This could be fixed, but it would slow down `vlist-cons`.
- `vlist-cons` always allocates at least as much memory as `cons`. Again, Phil Bagwell describes how to fix it, but that would require tuning the garbage collector in a way that may not be generally beneficial.
- `vlist-cons` is a Scheme procedure compiled to bytecode, and it does not compete with the straightforward C implementation of `cons`, and with the fact that the VM has a special `cons` instruction.

We hope to address these in the future.

The programming interface exported by `(ice-9 vlist)` is defined below. Most of it is the same as SRFI-1 with an added `vlist-` prefix to function names.

`vlist?` *obj* [Scheme Procedure]
Return true if *obj* is a VList.

`vlist-null` [Scheme Variable]
The empty VList. Note that it's possible to create an empty VList not `eq?` to `vlist-null`; thus, callers should always use `vlist-null?` when testing whether a VList is empty.

`vlist-null?` *vlist* [Scheme Procedure]
Return true if *vlist* is empty.

`vlist-cons` *item vlist* [Scheme Procedure]
Return a new vlist with *item* as its head and *vlist* as its tail.

`vlist-head` *vlist* [Scheme Procedure]
Return the head of *vlist*.

`vlist-tail` *vlist* [Scheme Procedure]
Return the tail of *vlist*.

`block-growth-factor` [Scheme Variable]
A fluid that defines the growth factor of VList blocks, 2 by default.

The functions below provide the usual set of higher-level list operations.

`vlist-fold` *proc init vlist* [Scheme Procedure]
`vlist-fold-right` *proc init vlist* [Scheme Procedure]
Fold over *vlist*, calling *proc* for each element, as for SRFI-1 `fold` and `fold-right` (see Section 7.5.3 [SRFI-1], page 584).

`vlist-ref` *vlist index* [Scheme Procedure]
Return the element at index *index* in *vlist*. This is typically a constant-time operation.

`vlist-length` *vlist* [Scheme Procedure]
Return the length of *vlist*. This is typically logarithmic in the number of elements in *vlist*.

`vlist-reverse` *vlist* [Scheme Procedure]
Return a new *vlist* whose content are those of *vlist* in reverse order.

`vlist-map` *proc vlist* [Scheme Procedure]
Map *proc* over the elements of *vlist* and return a new vlist.

`vlist-for-each` *proc vlist* [Scheme Procedure]
Call *proc* on each element of *vlist*. The result is unspecified.

`vlist-drop` *vlist count* [Scheme Procedure]
Return a new vlist that does not contain the *count* first elements of *vlist*. This is typically a constant-time operation.

`vlist-take` *vlist count* [Scheme Procedure]
Return a new vlist that contains only the *count* first elements of *vlist*.

`vlist-filter` *pred vlist* [Scheme Procedure]
Return a new vlist containing all the elements from *vlist* that satisfy *pred*.

`vlist-delete` *x vlist* [*equal?*] [Scheme Procedure]
Return a new vlist corresponding to *vlist* without the elements *equal?* to *x*.

`vlist-unfold` *p f g seed* [*tail-gen*] [Scheme Procedure]
`vlist-unfold-right` *p f g seed* [*tail*] [Scheme Procedure]
Return a new vlist, as for SRFI-1 `unfold` and `unfold-right` (see Section 7.5.3 [SRFI-1], page 584).

`vlist-append` *vlist ...* [Scheme Procedure]
Append the given vlists and return the resulting vlist.

`list->vlist` *lst* [Scheme Procedure]
Return a new vlist whose contents correspond to *lst*.

`vlist->list` *vlist* [Scheme Procedure]
Return a new list whose contents match those of *vlist*.

6.6.15 Record Overview

Records, also called *structures*, are Scheme's primary mechanism to define new disjoint types. A *record type* defines a list of *fields* that instances of the type consist of. This is like C's `struct`.

Historically, Guile has offered several different ways to define record types and to create records, offering different features, and making different trade-offs. Over the years, each "standard" has also come with its own new record interface, leading to a maze of record APIs.

At the highest level is SRFI-9, a high-level record interface implemented by most Scheme implementations (see Section 6.6.16 [SRFI-9 Records], page 218). It defines a simple and efficient syntactic abstraction of record types and their associated type predicate, fields, and field accessors. SRFI-9 is suitable for most uses, and this is the recommended way to create record types in Guile. Similar high-level record APIs include SRFI-35 (see Section 7.5.24 [SRFI-35], page 626) and R6RS records (see Section 7.6.2.9 [rnrs records syntactic], page 675).

Then comes Guile's historical "records" API (see Section 6.6.17 [Records], page 221). Record types defined this way are first-class objects. Introspection facilities are available, allowing users to query the list of fields or the value of a specific field at run-time, without prior knowledge of the type.

Finally, the common denominator of these interfaces is Guile's *structure* API (see Section 6.6.18 [Structures], page 223). Guile's structures are the low-level building block for all other record APIs. Application writers will normally not need to use it.

Records created with these APIs may all be pattern-matched using Guile's standard pattern matcher (see Section 7.8 [Pattern Matching], page 706).

6.6.16 SRFI-9 Records

SRFI-9 standardizes a syntax for defining new record types and creating predicate, constructor, and field getter and setter functions. In Guile this is the recommended option to create new record types (see Section 6.6.15 [Record Overview], page 217). It can be used with:

```
(use-modules (srfi srfi-9))
```

`define-record-type` *type* [Scheme Syntax]
(*constructor fieldname* ...)
predicate
(*fieldname accessor* [*modifier*]) ...

Create a new record type, and make various `define`s for using it. This syntax can only occur at the top-level, not nested within some other form.

type is bound to the record type, which is as per the return from the core `make-record-type`. *type* also provides the name for the record, as per `record-type-name`.

constructor is bound to a function to be called as `(`*`constructor`* `fieldval ...)` to create a new record of this type. The arguments are initial values for the fields, one argument for each field, in the order they appear in the `define-record-type` form.

The *fieldname*s provide the names for the record fields, as per the core `record-type-fields` etc, and are referred to in the subsequent accessor/modifier forms.

predicate is bound to a function to be called as `(`*`predicate`* `obj)`. It returns `#t` or `#f` according to whether *obj* is a record of this type.

Each *accessor* is bound to a function to be called `(`*`accessor`* `record)` to retrieve the respective field from a *record*. Similarly each *modifier* is bound to a function to be called `(`*`modifier`* `record val)` to set the respective field in a *record*.

An example will illustrate typical usage,

```
(define-record-type <employee>
  (make-employee name age salary)
  employee?
  (name    employee-name)
  (age     employee-age    set-employee-age!)
  (salary  employee-salary set-employee-salary!))
```

This creates a new employee data type, with name, age and salary fields. Accessor functions are created for each field, but no modifier function for the name (the intention in this example being that it's established only when an employee object is created). These can all then be used as for example,

```
<employee> ⇒ #<record-type <employee>>

(define fred (make-employee "Fred" 45 20000.00))

(employee? fred)        ⇒ #t
(employee-age fred)     ⇒ 45
```

```
(set-employee-salary! fred 25000.00)  ;; pay rise
```

The functions created by `define-record-type` are ordinary top-level `define`s. They can be redefined or `set!` as desired, exported from a module, etc.

Non-toplevel Record Definitions

The SRFI-9 specification explicitly disallows record definitions in a non-toplevel context, such as inside `lambda` body or inside a *let* block. However, Guile's implementation does not enforce that restriction.

Custom Printers

You may use `set-record-type-printer!` to customize the default printing behavior of records. This is a Guile extension and is not part of SRFI-9. It is located in the `(srfi srfi-9 gnu)` module.

`set-record-type-printer!` *type proc* [Scheme Syntax]

Where *type* corresponds to the first argument of `define-record-type`, and *proc* is a procedure accepting two arguments, the record to print, and an output port.

This example prints the employee's name in brackets, for instance `[Fred]`.

```
(set-record-type-printer! <employee>
  (lambda (record port)
    (write-char #\[ port)
    (display (employee-name record) port)
    (write-char #\] port)))
```

Functional "Setters"

When writing code in a functional style, it is desirable to never alter the contents of records. For such code, a simple way to return new record instances based on existing ones is highly desirable.

The `(srfi srfi-9 gnu)` module extends SRFI-9 with facilities to return new record instances based on existing ones, only with one or more field values changed—*functional setters*. First, the `define-immutable-record-type` works like `define-record-type`, except that fields are immutable and setters are defined as functional setters.

`define-immutable-record-type` *type* [Scheme Syntax]
(*constructor fieldname* . . .)
predicate
(*fieldname accessor* [*modifier*]) . . .

Define *type* as a new record type, like `define-record-type`. However, the record type is made *immutable* (records may not be mutated, even with `struct-set!`), and any *modifier* is defined to be a functional setter—a procedure that returns a new record instance with the specified field changed, and leaves the original unchanged (see example below.)

In addition, the generic `set-field` and `set-fields` macros may be applied to any SRFI-9 record.

`set-field` *record* (*field sub-fields ...*) *value* [Scheme Syntax]
Return a new record of *record*'s type whose fields are equal to the corresponding fields of *record* except for the one specified by *field*.

field must be the name of the getter corresponding to the field of *record* being "set". Subsequent *sub-fields* must be record getters designating sub-fields within that field value to be set (see example below.)

`set-fields` *record* ((*field sub-fields ...*) *value*) ... [Scheme Syntax]
Like `set-field`, but can be used to set more than one field at a time. This expands to code that is more efficient than a series of single `set-field` calls.

To illustrate the use of functional setters, let's assume these two record type definitions:

```
(define-record-type <address>
  (address street city country)
  address?
  (street  address-street)
  (city    address-city)
  (country address-country))

(define-immutable-record-type <person>
  (person age email address)
  person?
  (age     person-age set-person-age)
  (email   person-email set-person-email)
  (address person-address set-person-address))
```

First, note that the `<person>` record type definition introduces named functional setters. These may be used like this:

```
(define fsf-address
  (address "Franklin Street" "Boston" "USA"))

(define rms
  (person 30 "rms@gnu.org" fsf-address))

(and (equal? (set-person-age rms 60)
             (person 60 "rms@gnu.org" fsf-address))
     (= (person-age rms) 30))
⇒ #t
```

Here, the original `<person>` record, to which *rms* is bound, is left unchanged.

Now, suppose we want to change both the street and age of *rms*. This can be achieved using `set-fields`:

```
(set-fields rms
  ((person-age) 60)
  ((person-address address-street) "Temple Place"))
⇒ #<<person> age: 60 email: "rms@gnu.org"
  address: #<<address> street: "Temple Place" city: "Boston" country: "USA">>
```

Notice how the above changed two fields of *rms*, including the `street` field of its `address` field, in a concise way. Also note that `set-fields` works equally well for types defined with just `define-record-type`.

6.6.17 Records

A *record type* is a first class object representing a user-defined data type. A *record* is an instance of a record type.

Note that in many ways, this interface is too low-level for every-day use. Most uses of records are better served by SRFI-9 records. See Section 6.6.16 [SRFI-9 Records], page 218.

`record?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a record of any type and `#f` otherwise.

Note that `record?` may be true of any Scheme value; there is no promise that records are disjoint with other Scheme types.

`make-record-type` *type-name field-names* [*print*] [Scheme Procedure]
[*#:parent*=`#f`] [*#:uid*=`#f`] [*#:extensible?*=`#f`] [*#:opaque?*=`#f`]
[*#:allow-duplicate-field-names?*=`#t`]

Create and return a new *record-type descriptor*.

type-name is a string naming the type. Currently it's only used in the printed representation of records, and in diagnostics. *field-names* is a list of elements of the form `(immutable` *name*`)`, `(mutable` *name*`)`, or *name*, where *name* are symbols naming the fields of a record of the type. Duplicates are not allowed among these symbols, unless *allow-duplicate-field-names?* is true.

```
(make-record-type "employee" '(name age salary))
```

The optional *print* argument is a function used by `display`, `write`, etc, for printing a record of the new type. It's called as `(`*print* `record port)` and should look at *record* and write to *port*.

Pass the `#:parent` keyword to derive a record type from a supertype. A derived record type has the fields from its parent type, followed by fields declared in the `make-record-type` call. Record predicates and field accessors for instance of a parent type will also work on any instance of a subtype.

Allowing record subtyping has a small amount of overhead. To avoid this overhead, prevent extensibility by passing `#:extensible? #f`. By default, record types in Guile are not extensible.

Generally speaking, calling `make-record-type` returns a fresh record type; it *generates* new record types. However sometimes you only want to define a record type if one hasn't been defined already. For a *nongenerative* record type definition, pass a symbol as the `#:uid` keyword parameter. If a record with the given *uid* was already defined, it will be returned instead. The type name, fields, parent (if any), and so on for the previously-defined type must be compatible.

R6RS defines a notion of "opaque" record types. Given an instance of an opaque record type, one cannot obtain a run-time representation of the record type. See Section 7.6.2.10 [rnrs records procedural], page 676, for full details. The `#:opaque?` flag is used by Guile's R6RS layer to record this information. The default is determined by whether the parent type, if any, was opaque.

Fields are mutable by default, meaning that `record-modifier` will return a procedure that can update a record in place. Specifying a field using the form `(immutable name)` instead marks a field as immutable.

`record-constructor` *rtd* [Scheme Procedure]
Return a procedure for constructing new members of the type represented by *rtd*. The result will be a procedure accepting exactly as many arguments as there are fields in the record type.

`record-predicate` *rtd* [Scheme Procedure]
Return a procedure for testing membership in the type represented by *rtd*. The returned procedure accepts exactly one argument and returns a true value if the argument is a member of the indicated record type; it returns a false value otherwise.

`record-accessor` *rtd field-name* [Scheme Procedure]
Return a procedure for reading the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly one argument which must be a record of the appropriate type; it returns the current value of the field named by the symbol *field-name* in that record.

If *field-name* is a symbol, it must be a member of the list of field-names in the call to `make-record-type` that created the type represented by *rtd*. If multiple fields in *rtd* have the same name, `record-accessor` returns the first one.

If *field-name* is an integer, it should be an index into `(record-type-fields rtd)`. This allows accessing fields with duplicate names.

`record-modifier` *rtd field-name* [Scheme Procedure]
Return a procedure for writing the value of a particular field of a member of the type represented by *rtd*. The returned procedure accepts exactly two arguments: first, a record of the appropriate type, and second, an arbitrary Scheme value; it modifies the field named by the symbol *field-name* in that record to contain the given value. The returned value of the modifier procedure is unspecified. The symbol *field-name* is a field name or a field index, as in `record-modifier`.

`record-type-descriptor` *record* [Scheme Procedure]
Return a record-type descriptor representing the type of the given record. That is, for example, if the returned descriptor were passed to `record-predicate`, the resulting predicate would return a true value when passed the given record. Note that it is not necessarily the case that the returned descriptor is the one that was passed to `record-constructor` in the call that created the constructor procedure that created the given record.

`record-type-name` *rtd* [Scheme Procedure]
Return the type-name associated with the type represented by rtd. The returned value is `eqv?` to the *type-name* argument given in the call to `make-record-type` that created the type represented by *rtd*.

`record-type-fields` *rtd* [Scheme Procedure]
Return a list of the symbols naming the fields in members of the type represented by *rtd*. The returned value is `equal?` to the field-names argument given in the call to `make-record-type` that created the type represented by *rtd*.

6.6.18 Structures

A *structure* is a first class data type which holds Scheme values or C words in fields numbered 0 upwards. A *vtable* is a structure that represents a structure type, giving field types and permissions, and an optional print function for `write` etc.

Structures are lower level than records (see Section 6.6.17 [Records], page 221). Usually, when you need to represent structured data, you just want to use records. But sometimes you need to implement new kinds of structured data abstractions, and for that purpose structures are useful. Indeed, records in Guile are implemented with structures.

6.6.18.1 Vtables

A vtable is a structure type, specifying its layout, and other information. A vtable is actually itself a structure, but there's no need to worry about that initially (see Section 6.6.18.3 [Vtable Contents], page 225.)

`make-vtable` *fields* [*print*] [Scheme Procedure]

Create a new vtable.

fields is a string describing the fields in the structures to be created. Each field is represented by two characters, a type letter and a permissions letter, for example `"pw"`. The types are as follows.

- `p` – a Scheme value. "p" stands for "protected" meaning it's protected against garbage collection.
- `u` – an arbitrary word of data (an `scm_t_bits`). At the Scheme level it's read and written as an unsigned integer. "u" stands for "unboxed", as it's stored as a raw value without additional type annotations.

It used to be that the second letter for each field was a permission code, such as `w` for writable or `r` for read-only. However over time structs have become more of a raw low-level facility; access control is better implemented as a layer on top. After all, `struct-set!` is a cross-cutting operator that can bypass abstractions made by higher-level record facilities; it's not generally safe (in the sense of abstraction-preserving) to expose `struct-set!` to "untrusted" code, even if the fields happen to be writable. Additionally, permission checks added overhead to every structure access in a way that couldn't be optimized out, hampering the ability of structs to act as a low-level building block. For all of these reasons, all fields in Guile structs are now writable; attempting to make a read-only field will now issue a deprecation warning, and the field will be writable regardless.

```
(make-vtable "pw")      ;; one scheme field
(make-vtable "pwuwuw")  ;; one scheme and two unboxed fields
```

The optional *print* argument is a function called by `display` and `write` (etc) to give a printed representation of a structure created from this vtable. It's called (`print struct port`) and should look at *struct* and write to *port*. The default print merely gives a form like '`#<struct ADDR:ADDR>`' with a pair of machine addresses.

The following print function for example shows the two fields of its structure.

```
(make-vtable "pwpw"
             (lambda (struct port)
```

```
(format port "#<~a and ~a>"
        (struct-ref struct 0)
        (struct-ref struct 1))))
```

6.6.18.2 Structure Basics

This section describes the basic procedures for working with structures. `make-struct/no-tail` creates a structure, and `struct-ref` and `struct-set!` access its fields.

`make-struct/no-tail` *vtable init ...* [Scheme Procedure]

Create a new structure, with layout per the given *vtable* (see Section 6.6.18.1 [Vtables], page 223).

The optional *init...* arguments are initial values for the fields of the structure. This is the only way to put values in read-only fields. If there are fewer *init* arguments than fields then the defaults are `#f` for a Scheme field (type `p`) or 0 for an unboxed field (type `u`).

The name is a bit strange, we admit. The reason for it is that Guile used to have a `make-struct` that took an additional argument; while we deprecate that old interface, `make-struct/no-tail` is the new name for this functionality.

For example,

```
(define v (make-vtable "pwpwpw"))
(define s (make-struct/no-tail v 123 "abc" 456))
(struct-ref s 0) ⇒ 123
(struct-ref s 1) ⇒ "abc"
```

`SCM scm_make_struct` (*SCM vtable, SCM tail_size, SCM init_list*) [C Function]

`SCM scm_c_make_struct` (*SCM vtable, SCM tail_size, SCM init,* ...) [C Function]

`SCM scm_c_make_structv` (*SCM vtable, SCM tail_size, size_t n_inits, scm_t_bits init*[]) [C Function]

There are a few ways to make structures from C. `scm_make_struct` takes a list, `scm_c_make_struct` takes variable arguments terminated with SCM_UNDEFINED, and `scm_c_make_structv` takes a packed array.

For all of these, *tail_size* should be zero (as a SCM value).

`struct?` *obj* [Scheme Procedure]

`scm_struct_p` (*obj*) [C Function]

Return `#t` if *obj* is a structure, or `#f` if not.

`struct-ref` *struct n* [Scheme Procedure]

`scm_struct_ref` (*struct, n*) [C Function]

Return the contents of field number *n* in *struct*. The first field is number 0.

An error is thrown if *n* is out of range.

`struct-set!` *struct n value* [Scheme Procedure]

`scm_struct_set_x` (*struct, n, value*) [C Function]

Set field number *n* in *struct* to *value*. The first field is number 0.

An error is thrown if *n* is out of range, or if the field cannot be written because it's `r` read-only.

Unboxed fields (those with type `u`) need to be accessed with special procedures.

`struct-ref/unboxed` *struct n* [Scheme Procedure]
`struct-set!/unboxed` *struct n value* [Scheme Procedure]
`scm_struct_ref_unboxed` (*struct, n*) [C Function]
`scm_struct_set_x_unboxed` (*struct, n, value*) [C Function]

Like `struct-ref` and `struct-set!`, except that these may only be used on unboxed fields. `struct-ref/unboxed` will always return a positive integer. Likewise, `struct-set!/unboxed` takes an unsigned integer as the *value* argument, and will signal an error otherwise.

`struct-vtable` *struct* [Scheme Procedure]
`scm_struct_vtable` (*struct*) [C Function]

Return the vtable that describes *struct*.

The vtable is effectively the type of the structure. See Section 6.6.18.3 [Vtable Contents], page 225, for more on vtables.

6.6.18.3 Vtable Contents

A vtable is itself a structure. It has a specific set of fields describing various aspects of its *instances*: the structures created from a vtable. Some of the fields are internal to Guile, some of them are part of the public interface, and there may be additional fields added on by the user.

Every vtable has a field for the layout of their instances, a field for the procedure used to print its instances, and a field for the name of the vtable itself. Access to the layout and printer is exposed directly via field indexes. Access to the vtable name is exposed via accessor procedures.

`vtable-index-layout` [Scheme Variable]
`scm_vtable_index_layout` [C Macro]

The field number of the layout specification in a vtable. The layout specification is a symbol like `pwpw` formed from the fields string passed to `make-vtable`, or created by `make-struct-layout` (see Section 6.6.18.4 [Meta-Vtables], page 226).

```
(define v (make-vtable "pwpw" 0))
(struct-ref v vtable-index-layout) ⇒ pwpw
```

This field is read-only, since the layout of structures using a vtable cannot be changed.

`vtable-index-printer` [Scheme Variable]
`scm_vtable_index_printer` [C Macro]

The field number of the printer function. This field contains `#f` if the default print function should be used.

```
(define (my-print-func struct port)
  ...)
(define v (make-vtable "pwpw" my-print-func))
(struct-ref v vtable-index-printer) ⇒ my-print-func
```

This field is writable, allowing the print function to be changed dynamically.

`struct-vtable-name` *vtable* [Scheme Procedure]
`set-struct-vtable-name!` *vtable name* [Scheme Procedure]
`scm_struct_vtable_name` (*vtable*) [C Function]
`scm_set_struct_vtable_name_x` (*vtable, name*) [C Function]

Get or set the name of *vtable*. *name* is a symbol and is used in the default print function when printing structures created from *vtable*.

```
(define v (make-vtable "pw"))
(set-struct-vtable-name! v 'my-name)

(define s (make-struct v 0))
(display s) ⊣ #<my-name b7ab3ae0:b7ab3730>
```

6.6.18.4 Meta-Vtables

As a structure, a vtable also has a vtable, which is also a structure. Structures, their vtables, the vtables of the vtables, and so on form a tree of structures. Making a new structure adds a leaf to the tree, and if that structure is a vtable, it may be used to create other leaves.

If you traverse up the tree of vtables, via calling `struct-vtable`, eventually you reach a root which is the vtable of itself:

```
scheme@(guile-user)> (current-module)
$1 = #<directory (guile-user) 221b090>
scheme@(guile-user)> (struct-vtable $1)
$2 = #<record-type module>
scheme@(guile-user)> (struct-vtable $2)
$3 = #<<standard-vtable> 12c30a0>
scheme@(guile-user)> (struct-vtable $3)
$4 = #<<standard-vtable> 12c3fa0>
scheme@(guile-user)> (struct-vtable $4)
$5 = #<<standard-vtable> 12c3fa0>
scheme@(guile-user)> <standard-vtable>
$6 = #<<standard-vtable> 12c3fa0>
```

In this example, we can say that `$1` is an instance of `$2`, `$2` is an instance of `$3`, `$3` is an instance of `$4`, and `$4`, strangely enough, is an instance of itself. The value bound to `$4` in this console session also bound to `<standard-vtable>` in the default environment.

`<standard-vtable>` [Scheme Variable]

A meta-vtable, useful for making new vtables.

All of these values are structures. All but `$1` are vtables. As `$2` is an instance of `$3`, and `$3` is a vtable, we can say that `$3` is a *meta-vtable*: a vtable that can create vtables.

With this definition, we can specify more precisely what a vtable is: a vtable is a structure made from a meta-vtable. Making a structure from a meta-vtable runs some special checks to ensure that the first field of the structure is a valid layout. Additionally, if these checks see that the layout of the child vtable contains all the required fields of a vtable, in the correct order, then the child vtable will also be a meta-table, inheriting a magical bit from the parent.

`struct-vtable?` *obj* [Scheme Procedure]
`scm_struct_vtable_p` (*obj*) [C Function]
Return `#t` if *obj* is a vtable structure: an instance of a meta-vtable.

`<standard-vtable>` is a root of the vtable tree. (Normally there is only one root in a given Guile process, but due to some legacy interfaces there may be more than one.)

The set of required fields of a vtable is the set of fields in the `<standard-vtable>`, and is bound to `standard-vtable-fields` in the default environment. It is possible to create a meta-vtable that with additional fields in its layout, which can be used to create vtables with additional data:

```
scheme@(guile-user)> (struct-ref $3 vtable-index-layout)
$6 = pwuhuhpwphuhuhpwpwpw
scheme@(guile-user)> (struct-ref $4 vtable-index-layout)
$7 = pwuhuhpwphuhuh
scheme@(guile-user)> standard-vtable-fields
$8 = "pwuhuhpwphuhuh"
scheme@(guile-user)> (struct-ref $2 vtable-offset-user)
$9 = module
```

In this continuation of our earlier example, `$2` is a vtable that has extra fields, because its vtable, `$3`, was made from a meta-vtable with an extended layout. `vtable-offset-user` is a convenient definition that indicates the number of fields in `standard-vtable-fields`.

`standard-vtable-fields` [Scheme Variable]
A string containing the ordered set of fields that a vtable must have.

`vtable-offset-user` [Scheme Variable]
The first index in a vtable that is available for a user.

`make-struct-layout` *fields* [Scheme Procedure]
`scm_make_struct_layout` (*fields*) [C Function]
Return a structure layout symbol, from a *fields* string. *fields* is as described under `make-vtable` (see Section 6.6.18.1 [Vtables], page 223). An invalid *fields* string is an error.

With these definitions, one can define `make-vtable` in this way:

```
(define* (make-vtable fields #:optional printer)
  (make-struct/no-tail <standard-vtable>
    (make-struct-layout fields)
    printer))
```

6.6.18.5 Vtable Example

Let us bring these points together with an example. Consider a simple object system with single inheritance. Objects will be normal structures, and classes will be vtables with three extra class fields: the name of the class, the parent class, and the list of fields.

So, first we need a meta-vtable that allocates instances with these extra class fields.

```
(define <class>
  (make-vtable
```

```
   (string-append standard-vtable-fields "pwpwpw")
   (lambda (x port)
     (format port "<<class> ~a>" (class-name x)))))

(define (class? x)
  (and (struct? x)
       (eq? (struct-vtable x) <class>)))
```

To make a structure with a specific meta-vtable, we will use `make-struct/no-tail`, passing it the computed instance layout and printer, as with `make-vtable`, and additionally the extra three class fields.

```
(define (make-class name parent fields)
  (let* ((fields (compute-fields parent fields))
         (layout (compute-layout fields)))
    (make-struct/no-tail <class>
      layout
      (lambda (x port)
        (print-instance x port))
      name
      parent
      fields)))
```

Instances will store their associated data in slots in the structure: as many slots as there are fields. The `compute-layout` procedure below can compute a layout, and `field-index` returns the slot corresponding to a field.

```
(define-syntax-rule (define-accessor name n)
  (define (name obj)
    (struct-ref obj n)))

;; Accessors for classes
(define-accessor class-name (+ vtable-offset-user 0))
(define-accessor class-parent (+ vtable-offset-user 1))
(define-accessor class-fields (+ vtable-offset-user 2))

(define (compute-fields parent fields)
  (if parent
      (append (class-fields parent) fields)
      fields))

(define (compute-layout fields)
  (make-struct-layout
   (string-concatenate (make-list (length fields) "pw"))))

(define (field-index class field)
  (list-index (class-fields class) field))

(define (print-instance x port)
  (format port "<~a" (class-name (struct-vtable x)))
```

```
    (for-each (lambda (field idx)
                (format port " ~a: ~a" field (struct-ref x idx)))
              (class-fields (struct-vtable x))
              (iota (length (class-fields (struct-vtable x)))))
    (format port ">"))
```

So, at this point we can actually make a few classes:

```
(define-syntax-rule (define-class name parent field ...)
  (define name (make-class 'name parent '(field ...))))

(define-class <surface> #f
  width height)

(define-class <window> <surface>
  x y)
```

And finally, make an instance:

```
(make-struct/no-tail <window> 400 300 10 20)
⇒ <<window> width: 400 height: 300 x: 10 y: 20>
```

And that's that. Note that there are many possible optimizations and feature enhancements that can be made to this object system, and the included GOOPS system does make most of them. For more simple use cases, the records facility is usually sufficient. But sometimes you need to make new kinds of data abstractions, and for that purpose, structs are here.

6.6.19 Dictionary Types

A *dictionary* object is a data structure used to index information in a user-defined way. In standard Scheme, the main aggregate data types are lists and vectors. Lists are not really indexed at all, and vectors are indexed only by number (e.g. `(vector-ref foo 5)`). Often you will find it useful to index your data on some other type; for example, in a library catalog you might want to look up a book by the name of its author. Dictionaries are used to help you organize information in such a way.

An *association list* (or *alist* for short) is a list of key-value pairs. Each pair represents a single quantity or object; the `car` of the pair is a key which is used to identify the object, and the `cdr` is the object's value.

A *hash table* also permits you to index objects with arbitrary keys, but in a way that makes looking up any one object extremely fast. A well-designed hash system makes hash table lookups almost as fast as conventional array or vector references.

Alists are popular among Lisp programmers because they use only the language's primitive operations (lists, *car*, *cdr* and the equality primitives). No changes to the language core are necessary. Therefore, with Scheme's built-in list manipulation facilities, it is very convenient to handle data stored in an association list. Also, alists are highly portable and can be easily implemented on even the most minimal Lisp systems.

However, alists are inefficient, especially for storing large quantities of data. Because we want Guile to be useful for large software systems as well as small ones, Guile provides a rich set of tools for using either association lists or hash tables.

6.6.20 Association Lists

An association list is a conventional data structure that is often used to implement simple key-value databases. It consists of a list of entries in which each entry is a pair. The *key* of each entry is the `car` of the pair and the *value* of each entry is the `cdr`.

```
ASSOCIATION LIST ::=  '( (KEY1 . VALUE1)
                         (KEY2 . VALUE2)
                         (KEY3 . VALUE3)
                         ...
                       )
```

Association lists are also known, for short, as *alists*.

The structure of an association list is just one example of the infinite number of possible structures that can be built using pairs and lists. As such, the keys and values in an association list can be manipulated using the general list structure procedures `cons`, `car`, `cdr`, `set-car!`, `set-cdr!` and so on. However, because association lists are so useful, Guile also provides specific procedures for manipulating them.

6.6.20.1 Alist Key Equality

All of Guile's dedicated association list procedures, apart from `acons`, come in three flavours, depending on the level of equality that is required to decide whether an existing key in the association list is the same as the key that the procedure call uses to identify the required entry.

- Procedures with *assq* in their name use `eq?` to determine key equality.
- Procedures with *assv* in their name use `eqv?` to determine key equality.
- Procedures with *assoc* in their name use `equal?` to determine key equality.

`acons` is an exception because it is used to build association lists which do not require their entries' keys to be unique.

6.6.20.2 Adding or Setting Alist Entries

`acons` adds a new entry to an association list and returns the combined association list. The combined alist is formed by consing the new entry onto the head of the alist specified in the `acons` procedure call. So the specified alist is not modified, but its contents become shared with the tail of the combined alist that `acons` returns.

In the most common usage of `acons`, a variable holding the original association list is updated with the combined alist:

```
(set! address-list (acons name address address-list))
```

In such cases, it doesn't matter that the old and new values of `address-list` share some of their contents, since the old value is usually no longer independently accessible.

Note that `acons` adds the specified new entry regardless of whether the alist may already contain entries with keys that are, in some sense, the same as that of the new entry. Thus `acons` is ideal for building alists where there is no concept of key uniqueness.

```
(set! task-list (acons 3 "pay gas bill" '()))
task-list
⇒
```

```
((3 . "pay gas bill"))

(set! task-list (acons 3 "tidy bedroom" task-list))
task-list
⇒
((3 . "tidy bedroom") (3 . "pay gas bill"))
```

`assq-set!`, `assv-set!` and `assoc-set!` are used to add or replace an entry in an association list where there *is* a concept of key uniqueness. If the specified association list already contains an entry whose key is the same as that specified in the procedure call, the existing entry is replaced by the new one. Otherwise, the new entry is consed onto the head of the old association list to create the combined alist. In all cases, these procedures return the combined alist.

`assq-set!` and friends *may* destructively modify the structure of the old association list in such a way that an existing variable is correctly updated without having to `set!` it to the value returned:

```
address-list
⇒
(("mary" . "34 Elm Road") ("james" . "16 Bow Street"))

(assoc-set! address-list "james" "1a London Road")
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))

address-list
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))
```

Or they may not:

```
(assoc-set! address-list "bob" "11 Newington Avenue")
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))

address-list
⇒
(("mary" . "34 Elm Road") ("james" . "1a London Road"))
```

The only safe way to update an association list variable when adding or replacing an entry like this is to `set!` the variable to the returned value:

```
(set! address-list
      (assoc-set! address-list "bob" "11 Newington Avenue"))
address-list
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))
```

Because of this slight inconvenience, you may find it more convenient to use hash tables to store dictionary data. If your application will not be modifying the contents of an alist very often, this may not make much difference to you.

If you need to keep the old value of an association list in a form independent from the list that results from modification by `acons`, `assq-set!`, `assv-set!` or `assoc-set!`, use `list-copy` to copy the old association list before modifying it.

`acons` *key value alist* [Scheme Procedure]
`scm_acons` (*key*, *value*, *alist*) [C Function]
Add a new key-value pair to *alist*. A new pair is created whose car is *key* and whose cdr is *value*, and the pair is consed onto *alist*, and the new list is returned. This function is *not* destructive; *alist* is not modified.

`assq-set!` *alist key val* [Scheme Procedure]
`assv-set!` *alist key value* [Scheme Procedure]
`assoc-set!` *alist key value* [Scheme Procedure]
`scm_assq_set_x` (*alist*, *key*, *val*) [C Function]
`scm_assv_set_x` (*alist*, *key*, *val*) [C Function]
`scm_assoc_set_x` (*alist*, *key*, *val*) [C Function]
Reassociate *key* in *alist* with *value*: find any existing *alist* entry for *key* and associate it with the new *value*. If *alist* does not contain an entry for *key*, add a new one. Return the (possibly new) alist.

These functions do not attempt to verify the structure of *alist*, and so may cause unusual results if passed an object that is not an association list.

6.6.20.3 Retrieving Alist Entries

`assq`, `assv` and `assoc` find the entry in an alist for a given key, and return the `(`*`key`* `.` *`value`*`)` pair. `assq-ref`, `assv-ref` and `assoc-ref` do a similar lookup, but return just the *value*.

`assq` *key alist* [Scheme Procedure]
`assv` *key alist* [Scheme Procedure]
`assoc` *key alist* [Scheme Procedure]
`scm_assq` (*key*, *alist*) [C Function]
`scm_assv` (*key*, *alist*) [C Function]
`scm_assoc` (*key*, *alist*) [C Function]
Return the first entry in *alist* with the given *key*. The return is the pair `(KEY . VALUE)` from *alist*. If there's no matching entry the return is `#f`.

`assq` compares keys with `eq?`, `assv` uses `eqv?` and `assoc` uses `equal?`. See also SRFI-1 which has an extended `assoc` (Section 7.5.3.9 [SRFI-1 Association Lists], page 594).

`assq-ref` *alist key* [Scheme Procedure]
`assv-ref` *alist key* [Scheme Procedure]
`assoc-ref` *alist key* [Scheme Procedure]
`scm_assq_ref` (*alist*, *key*) [C Function]
`scm_assv_ref` (*alist*, *key*) [C Function]

`scm_assoc_ref` (*alist*, *key*) [C Function]

Return the value from the first entry in *alist* with the given *key*, or `#f` if there's no such entry.

`assq-ref` compares keys with `eq?`, `assv-ref` uses `eqv?` and `assoc-ref` uses `equal?`.

Notice these functions have the *key* argument last, like other `-ref` functions, but this is opposite to what `assq` etc above use.

When the return is `#f` it can be either *key* not found, or an entry which happens to have value `#f` in the `cdr`. Use `assq` etc above if you need to differentiate these cases.

6.6.20.4 Removing Alist Entries

To remove the element from an association list whose key matches a specified key, use `assq-remove!`, `assv-remove!` or `assoc-remove!` (depending, as usual, on the level of equality required between the key that you specify and the keys in the association list).

As with `assq-set!` and friends, the specified alist may or may not be modified destructively, and the only safe way to update a variable containing the alist is to `set!` it to the value that `assq-remove!` and friends return.

```
address-list
⇒
(("bob" . "11 Newington Avenue") ("mary" . "34 Elm Road")
 ("james" . "1a London Road"))

(set! address-list (assoc-remove! address-list "mary"))
address-list
⇒
(("bob" . "11 Newington Avenue") ("james" . "1a London Road"))
```

Note that, when `assq/v/oc-remove!` is used to modify an association list that has been constructed only using the corresponding `assq/v/oc-set!`, there can be at most one matching entry in the alist, so the question of multiple entries being removed in one go does not arise. If `assq/v/oc-remove!` is applied to an association list that has been constructed using `acons`, or an `assq/v/oc-set!` with a different level of equality, or any mixture of these, it removes only the first matching entry from the alist, even if the alist might contain further matching entries. For example:

```
(define address-list '())
(set! address-list (assq-set! address-list "mary" "11 Elm Street"))
(set! address-list (assq-set! address-list "mary" "57 Pine Drive"))
address-list
⇒
(("mary" . "57 Pine Drive") ("mary" . "11 Elm Street"))

(set! address-list (assoc-remove! address-list "mary"))
address-list
⇒
(("mary" . "11 Elm Street"))
```

In this example, the two instances of the string `"mary"` are not the same when compared using `eq?`, so the two `assq-set!` calls add two distinct entries to `address-list`. When

compared using `equal?`, both "mary"s in `address-list` are the same as the "mary" in the `assoc-remove!` call, but `assoc-remove!` stops after removing the first matching entry that it finds, and so one of the "mary" entries is left in place.

`assq-remove!` *alist key* [Scheme Procedure]
`assv-remove!` *alist key* [Scheme Procedure]
`assoc-remove!` *alist key* [Scheme Procedure]
`scm_assq_remove_x` (*alist, key*) [C Function]
`scm_assv_remove_x` (*alist, key*) [C Function]
`scm_assoc_remove_x` (*alist, key*) [C Function]

Delete the first entry in *alist* associated with *key*, and return the resulting alist.

6.6.20.5 Sloppy Alist Functions

`sloppy-assq`, `sloppy-assv` and `sloppy-assoc` behave like the corresponding non-`sloppy-` procedures, except that they return `#f` when the specified association list is not well-formed, where the non-`sloppy-` versions would signal an error.

Specifically, there are two conditions for which the non-`sloppy-` procedures signal an error, which the `sloppy-` procedures handle instead by returning `#f`. Firstly, if the specified alist as a whole is not a proper list:

```
(assoc "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
ERROR: In procedure assoc in expression (assoc "mary" (quote #)):
ERROR: Wrong type argument in position 2 (expecting
   association list): ((1 . 2) ("key" . "door") . "open sesame")

(sloppy-assoc "mary" '((1 . 2) ("key" . "door") . "open sesame"))
⇒
#f
```

Secondly, if one of the entries in the specified alist is not a pair:

```
(assoc 2 '((1 . 1) 2 (3 . 9)))
⇒
ERROR: In procedure assoc in expression (assoc 2 (quote #)):
ERROR: Wrong type argument in position 2 (expecting
   association list): ((1 . 1) 2 (3 . 9))

(sloppy-assoc 2 '((1 . 1) 2 (3 . 9)))
⇒
#f
```

Unless you are explicitly working with badly formed association lists, it is much safer to use the non-`sloppy-` procedures, because they help to highlight coding and data errors that the `sloppy-` versions would silently cover up.

`sloppy-assq` *key alist* [Scheme Procedure]
`scm_sloppy_assq` (*key, alist*) [C Function]

Behaves like `assq` but does not do any error checking. Recommended only for use in Guile internals.

`sloppy-assv` *key alist* [Scheme Procedure]
`scm_sloppy_assv` (*key*, *alist*) [C Function]
Behaves like `assv` but does not do any error checking. Recommended only for use in Guile internals.

`sloppy-assoc` *key alist* [Scheme Procedure]
`scm_sloppy_assoc` (*key*, *alist*) [C Function]
Behaves like `assoc` but does not do any error checking. Recommended only for use in Guile internals.

6.6.20.6 Alist Example

Here is a longer example of how alists may be used in practice.

```
(define capitals '(("New York" . "Albany")
                   ("Oregon"   . "Salem")
                   ("Florida"  . "Miami")))

;; What's the capital of Oregon?
(assoc "Oregon" capitals)       ⇒ ("Oregon" . "Salem")
(assoc-ref capitals "Oregon")   ⇒ "Salem"

;; We left out South Dakota.
(set! capitals
      (assoc-set! capitals "South Dakota" "Pierre"))
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Oregon" . "Salem")
    ("Florida" . "Miami"))

;; And we got Florida wrong.
(set! capitals
      (assoc-set! capitals "Florida" "Tallahassee"))
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Oregon" . "Salem")
    ("Florida" . "Tallahassee"))

;; After Oregon secedes, we can remove it.
(set! capitals
      (assoc-remove! capitals "Oregon"))
capitals
⇒ (("South Dakota" . "Pierre")
    ("New York" . "Albany")
    ("Florida" . "Tallahassee"))
```

6.6.21 VList-Based Hash Lists or "VHashes"

The `(ice-9 vlist)` module provides an implementation of *VList-based hash lists* (see Section 6.6.14 [VLists], page 215). VList-based hash lists, or *vhashes*, are an immutable dictionary type similar to association lists that maps *keys* to *values*. However, unlike association lists, accessing a value given its key is typically a constant-time operation.

The VHash programming interface of `(ice-9 vlist)` is mostly the same as that of association lists found in SRFI-1, with procedure names prefixed by `vhash-` instead of `alist-` (see Section 7.5.3.9 [SRFI-1 Association Lists], page 594).

In addition, vhashes can be manipulated using VList operations:

```
(vlist-head (vhash-consq 'a 1 vlist-null))
⇒ (a . 1)

(define vh1 (vhash-consq 'b 2 (vhash-consq 'a 1 vlist-null)))
(define vh2 (vhash-consq 'c 3 (vlist-tail vh1)))

(vhash-assq 'a vh2)
⇒ (a . 1)
(vhash-assq 'b vh2)
⇒ #f
(vhash-assq 'c vh2)
⇒ (c . 3)
(vlist->list vh2)
⇒ ((c . 3) (a . 1))
```

However, keep in mind that procedures that construct new VLists (`vlist-map`, `vlist-filter`, etc.) return raw VLists, not vhashes:

```
(define vh (alist->vhash '((a . 1) (b . 2) (c . 3)) hashq))
(vhash-assq 'a vh)
⇒ (a . 1)

(define vl
  ;; This will create a raw vlist.
  (vlist-filter (lambda (key+value) (odd? (cdr key+value))) vh))
(vhash-assq 'a vl)
⇒ ERROR: Wrong type argument in position 2

(vlist->list vl)
⇒ ((a . 1) (c . 3))
```

`vhash?` *obj* [Scheme Procedure]

Return true if *obj* is a vhash.

`vhash-cons` *key value vhash* [*hash-proc*] [Scheme Procedure]
`vhash-consq` *key value vhash* [Scheme Procedure]
`vhash-consv` *key value vhash* [Scheme Procedure]

Return a new hash list based on *vhash* where *key* is associated with *value*, using *hash-proc* to compute the hash of *key*. *vhash* must be either `vlist-null` or a

vhash returned by a previous call to `vhash-cons`. *hash-proc* defaults to `hash` (see Section 6.6.22.2 [Hash Table Reference], page 239). With `vhash-consq`, the `hashq` hash function is used; with `vhash-consv` the `hashv` hash function is used.

All `vhash-cons` calls made to construct a vhash should use the same *hash-proc*. Failing to do that, the result is undefined.

`vhash-assoc` *key vhash* [*equal?* [*hash-proc*]] [Scheme Procedure]
`vhash-assq` *key vhash* [Scheme Procedure]
`vhash-assv` *key vhash* [Scheme Procedure]

Return the first key/value pair from *vhash* whose key is equal to *key* according to the *equal?* equality predicate (which defaults to `equal?`), and using *hash-proc* (which defaults to `hash`) to compute the hash of *key*. The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the last form uses `eqv?` and `hashv`.

Note that it is important to consistently use the same hash function for *hash-proc* as was passed to `vhash-cons`. Failing to do that, the result is unpredictable.

`vhash-delete` *key vhash* [*equal?* [*hash-proc*]] [Scheme Procedure]
`vhash-delq` *key vhash* [Scheme Procedure]
`vhash-delv` *key vhash* [Scheme Procedure]

Remove all associations from *vhash* with *key*, comparing keys with *equal?* (which defaults to `equal?`), and computing the hash of *key* using *hash-proc* (which defaults to `hash`). The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the last one uses `eqv?` and `hashv`.

Again the choice of *hash-proc* must be consistent with previous calls to `vhash-cons`.

`vhash-fold` *proc init vhash* [Scheme Procedure]
`vhash-fold-right` *proc init vhash* [Scheme Procedure]

Fold over the key/value elements of *vhash* in the given direction, with each call to *proc* having the form `(`*proc* `key value result)`, where *result* is the result of the previous call to *proc* and *init* the value of *result* for the first call to *proc*.

`vhash-fold*` *proc init key vhash* [*equal?* [*hash*]] [Scheme Procedure]
`vhash-foldq*` *proc init key vhash* [Scheme Procedure]
`vhash-foldv*` *proc init key vhash* [Scheme Procedure]

Fold over all the values associated with *key* in *vhash*, with each call to *proc* having the form `(proc value result)`, where *result* is the result of the previous call to *proc* and *init* the value of *result* for the first call to *proc*.

Keys in *vhash* are hashed using *hash* are compared using *equal?*. The second form uses `eq?` as the equality predicate and `hashq` as the hash function; the third one uses `eqv?` and `hashv`.

Example:

```
(define vh
  (alist->vhash '((a . 1) (a . 2) (z . 0) (a . 3))))

(vhash-fold* cons '() 'a vh)
⇒ (3 2 1)
```

```
(vhash-fold* cons '() 'z vh)
⇒ (0)
```

`alist->vhash` *alist* [*hash-proc*] [Scheme Procedure]

Return the vhash corresponding to *alist*, an association list, using *hash-proc* to compute key hashes. When omitted, *hash-proc* defaults to `hash`.

6.6.22 Hash Tables

Hash tables are dictionaries which offer similar functionality as association lists: They provide a mapping from keys to values. The difference is that association lists need time linear in the size of elements when searching for entries, whereas hash tables can normally search in constant time. The drawback is that hash tables require a little bit more memory, and that you can not use the normal list procedures (see Section 6.6.9 [Lists], page 181) for working with them.

6.6.22.1 Hash Table Examples

For demonstration purposes, this section gives a few usage examples of some hash table procedures, together with some explanation what they do.

First we start by creating a new hash table with 31 slots, and populate it with two key/value pairs.

```
(define h (make-hash-table 31))

;; This is an opaque object
h
⇒
#<hash-table 0/31>

;; Inserting into a hash table can be done with hashq-set!
(hashq-set! h 'foo "bar")
⇒
"bar"

(hashq-set! h 'braz "zonk")
⇒
"zonk"

;; Or with hash-create-handle!
(hashq-create-handle! h 'frob #f)
⇒
(frob . #f)
```

You can get the value for a given key with the procedure `hashq-ref`, but the problem with this procedure is that you cannot reliably determine whether a key does exists in the table. The reason is that the procedure returns `#f` if the key is not in the table, but it will return the same value if the key is in the table and just happens to have the value `#f`, as you can see in the following examples.

```
(hashq-ref h 'foo)
```

```
⇒
"bar"

(hashq-ref h 'frob)
⇒
#f

(hashq-ref h 'not-there)
⇒
#f
```

It is often better is to use the procedure `hashq-get-handle`, which makes a distinction between the two cases. Just like `assq`, this procedure returns a key/value-pair on success, and `#f` if the key is not found.

```
(hashq-get-handle h 'foo)
⇒
(foo . "bar")

(hashq-get-handle h 'not-there)
⇒
#f
```

Interesting results can be computed by using `hash-fold` to work through each element. This example will count the total number of elements:

```
(hash-fold (lambda (key value seed) (+ 1 seed)) 0 h)
⇒
3
```

The same thing can be done with the procedure `hash-count`, which can also count the number of elements matching a particular predicate. For example, count the number of elements with string values:

```
(hash-count (lambda (key value) (string? value)) h)
⇒
2
```

Counting all the elements is a simple task using `const`:

```
(hash-count (const #t) h)
⇒
3
```

6.6.22.2 Hash Table Reference

Like the association list functions, the hash table functions come in several varieties, according to the equality test used for the keys. Plain `hash-` functions use `equal?`, `hashq-` functions use `eq?`, `hashv-` functions use `eqv?`, and the `hashx-` functions use an application supplied test.

A single `make-hash-table` creates a hash table suitable for use with any set of functions, but it's imperative that just one set is then used consistently, or results will be unpredictable.

Hash tables are implemented as a vector indexed by a hash value formed from the key, with an association list of key/value pairs for each bucket in case distinct keys hash together. Direct access to the pairs in those lists is provided by the `-handle-` functions.

When the number of entries in a hash table goes above a threshold, the vector is made larger and the entries are rehashed, to prevent the bucket lists from becoming too long and slowing down accesses. When the number of entries goes below a threshold, the vector is shrunk to save space.

For the `hashx-` "extended" routines, an application supplies a *hash* function producing an integer index like `hashq` etc below, and an *assoc* alist search function like `assq` etc (see Section 6.6.20.3 [Retrieving Alist Entries], page 232). Here's an example of such functions implementing case-insensitive hashing of string keys,

```
(use-modules (srfi srfi-1)
             (srfi srfi-13))

(define (my-hash str size)
  (remainder (string-hash-ci str) size))
(define (my-assoc str alist)
  (find (lambda (pair) (string-ci=? str (car pair))) alist))

(define my-table (make-hash-table))
(hashx-set! my-hash my-assoc my-table "foo" 123)

(hashx-ref my-hash my-assoc my-table "FOO")
⇒ 123
```

In a `hashx-` *hash* function the aim is to spread keys across the vector, so bucket lists don't become long. But the actual values are arbitrary as long as they're in the range 0 to *size* − 1. Helpful functions for forming a hash value, in addition to `hashq` etc below, include `symbol-hash` (see Section 6.6.6.2 [Symbol Keys], page 166), `string-hash` and `string-hash-ci` (see Section 6.6.5.7 [String Comparison], page 148), and `char-set-hash` (see Section 6.6.4.1 [Character Set Predicates/Comparison], page 134).

`make-hash-table` [*size*] [Scheme Procedure]

Create a new hash table object, with an optional minimum vector *size*.

When *size* is given, the table vector will still grow and shrink automatically, as described above, but with *size* as a minimum. If an application knows roughly how many entries the table will hold then it can use *size* to avoid rehashing when initial entries are added.

`alist->hash-table` *alist* [Scheme Procedure]
`alist->hashq-table` *alist* [Scheme Procedure]
`alist->hashv-table` *alist* [Scheme Procedure]
`alist->hashx-table` *hash assoc alist* [Scheme Procedure]

Convert *alist* into a hash table. When keys are repeated in *alist*, the leftmost association takes precedence.

```
(use-modules (ice-9 hash-table))
(alist->hash-table '((foo . 1) (bar . 2)))
```

When converting to an extended hash table, custom *hash* and *assoc* procedures must be provided.

```
(alist->hashx-table hash assoc '((foo . 1) (bar . 2)))
```

`hash-table?` *obj* [Scheme Procedure]
`scm_hash_table_p` (*obj*) [C Function]
Return `#t` if *obj* is a abstract hash table object.

`hash-clear!` *table* [Scheme Procedure]
`scm_hash_clear_x` (*table*) [C Function]
Remove all items from *table* (without triggering a resize).

`hash-ref` *table key* [*dflt*] [Scheme Procedure]
`hashq-ref` *table key* [*dflt*] [Scheme Procedure]
`hashv-ref` *table key* [*dflt*] [Scheme Procedure]
`hashx-ref` *hash assoc table key* [*dflt*] [Scheme Procedure]
`scm_hash_ref` (*table, key, dflt*) [C Function]
`scm_hashq_ref` (*table, key, dflt*) [C Function]
`scm_hashv_ref` (*table, key, dflt*) [C Function]
`scm_hashx_ref` (*hash, assoc, table, key, dflt*) [C Function]
Lookup *key* in the given hash *table*, and return the associated value. If *key* is not found, return *dflt*, or `#f` if *dflt* is not given.

`hash-set!` *table key val* [Scheme Procedure]
`hashq-set!` *table key val* [Scheme Procedure]
`hashv-set!` *table key val* [Scheme Procedure]
`hashx-set!` *hash assoc table key val* [Scheme Procedure]
`scm_hash_set_x` (*table, key, val*) [C Function]
`scm_hashq_set_x` (*table, key, val*) [C Function]
`scm_hashv_set_x` (*table, key, val*) [C Function]
`scm_hashx_set_x` (*hash, assoc, table, key, val*) [C Function]
Associate *val* with *key* in the given hash *table*. If *key* is already present then it's associated value is changed. If it's not present then a new entry is created.

`hash-remove!` *table key* [Scheme Procedure]
`hashq-remove!` *table key* [Scheme Procedure]
`hashv-remove!` *table key* [Scheme Procedure]
`hashx-remove!` *hash assoc table key* [Scheme Procedure]
`scm_hash_remove_x` (*table, key*) [C Function]
`scm_hashq_remove_x` (*table, key*) [C Function]
`scm_hashv_remove_x` (*table, key*) [C Function]
`scm_hashx_remove_x` (*hash, assoc, table, key*) [C Function]
Remove any association for *key* in the given hash *table*. If *key* is not in *table* then nothing is done.

`hash` *key size* [Scheme Procedure]
`hashq` *key size* [Scheme Procedure]
`hashv` *key size* [Scheme Procedure]
`scm_hash` (*key, size*) [C Function]

`scm_hashq` (*key, size*) [C Function]
`scm_hashv` (*key, size*) [C Function]

Return a hash value for *key*. This is a number in the range 0 to *size* − 1, which is suitable for use in a hash table of the given *size*.

Note that `hashq` and `hashv` may use internal addresses of objects, so if an object is garbage collected and re-created it can have a different hash value, even when the two are notionally `eq?`. For instance with symbols,

```
(hashq 'something 123)    ⇒ 19
(gc)
(hashq 'something 123)    ⇒ 62
```

In normal use this is not a problem, since an object entered into a hash table won't be garbage collected until removed. It's only if hashing calculations are somehow separated from normal references that its lifetime needs to be considered.

`hash-get-handle` *table key* [Scheme Procedure]
`hashq-get-handle` *table key* [Scheme Procedure]
`hashv-get-handle` *table key* [Scheme Procedure]
`hashx-get-handle` *hash assoc table key* [Scheme Procedure]
`scm_hash_get_handle` (*table, key*) [C Function]
`scm_hashq_get_handle` (*table, key*) [C Function]
`scm_hashv_get_handle` (*table, key*) [C Function]
`scm_hashx_get_handle` (*hash, assoc, table, key*) [C Function]

Return the (`key . value`) pair for *key* in the given hash *table*, or `#f` if *key* is not in *table*.

`hash-create-handle!` *table key init* [Scheme Procedure]
`hashq-create-handle!` *table key init* [Scheme Procedure]
`hashv-create-handle!` *table key init* [Scheme Procedure]
`hashx-create-handle!` *hash assoc table key init* [Scheme Procedure]
`scm_hash_create_handle_x` (*table, key, init*) [C Function]
`scm_hashq_create_handle_x` (*table, key, init*) [C Function]
`scm_hashv_create_handle_x` (*table, key, init*) [C Function]
`scm_hashx_create_handle_x` (*hash, assoc, table, key, init*) [C Function]

Return the (`key . value`) pair for *key* in the given hash *table*. If *key* is not in *table* then create an entry for it with *init* as the value, and return that pair.

`hash-map->list` *proc table* [Scheme Procedure]
`hash-for-each` *proc table* [Scheme Procedure]
`scm_hash_map_to_list` (*proc, table*) [C Function]
`scm_hash_for_each` (*proc, table*) [C Function]

Apply *proc* to the entries in the given hash *table*. Each call is (`proc key value`). `hash-map->list` returns a list of the results from these calls, `hash-for-each` discards the results and returns an unspecified value.

Calls are made over the table entries in an unspecified order, and for `hash-map->list` the order of the values in the returned list is unspecified. Results will be unpredictable if *table* is modified while iterating.

For example the following returns a new alist comprising all the entries from `mytable`, in no particular order.

```
(hash-map->list cons mytable)
```

`hash-for-each-handle` *proc table* [Scheme Procedure]
`scm_hash_for_each_handle` (*proc, table*) [C Function]

Apply *proc* to the entries in the given hash *table*. Each call is `(proc handle)`, where *handle* is a `(key . value)` pair. Return an unspecified value.

`hash-for-each-handle` differs from `hash-for-each` only in the argument list of *proc*.

`hash-fold` *proc init table* [Scheme Procedure]
`scm_hash_fold` (*proc, init, table*) [C Function]

Accumulate a result by applying *proc* to the elements of the given hash *table*. Each call is `(proc key value prior-result)`, where *key* and *value* are from the *table* and *prior-result* is the return from the previous *proc* call. For the first call, *prior-result* is the given *init* value.

Calls are made over the table entries in an unspecified order. Results will be unpredictable if *table* is modified while `hash-fold` is running.

For example, the following returns a count of how many keys in `mytable` are strings.

```
(hash-fold (lambda (key value prior)
             (if (string? key) (1+ prior) prior))
           0 mytable)
```

`hash-count` *pred table* [Scheme Procedure]
`scm_hash_count` (*pred, table*) [C Function]

Return the number of elements in the given hash *table* that cause `(pred key value)` to return true. To quickly determine the total number of elements, use `(const #t)` for *pred*.

6.6.23 Other Types

Procedures are documented in their own section. See Section 6.9 [Procedures], page 248.

Variable objects are documented as part of the description of Guile's module system: see Section 6.20.7 [Variables], page 419.

See Section 6.22 [Scheduling], page 442, for discussion of threads, mutexes, and so on.

Ports are described in the section on I/O: see Section 6.14 [Input and Output], page 331.

Regular expressions are described in their own section: see Section 6.15 [Regular Expressions], page 358.

There are quite a number of additional data types documented in this manual; if you feel a link is missing here, please file a bug.

6.7 Foreign Objects

This chapter contains reference information related to defining and working with foreign objects. See Section 5.5 [Defining New Foreign Object Types], page 74, for a tutorial-like introduction to foreign objects.

`scm_t_struct_finalize` [C Type]
This function type returns `void` and takes one `SCM` argument.

`SCM scm_make_foreign_object_type` (*SCM name, SCM slots, scm_t_struct_finalize finalizer*) [C Function]
Create a fresh foreign object type. *name* is a symbol naming the type. *slots* is a list of symbols, each one naming a field in the foreign object type. *finalizer* indicates the finalizer, and may be `NULL`.

We recommend that finalizers be avoided if possible. See Section 5.5.4 [Foreign Object Memory Management], page 77. Finalizers must be async-safe and thread-safe. Again, see Section 5.5.4 [Foreign Object Memory Management], page 77. If you are embedding Guile in an application that is not thread-safe, and you define foreign object types that need finalization, you might want to disable automatic finalization, and arrange to call `scm_manually_run_finalizers ()` yourself.

`int scm_set_automatic_finalization_enabled` (*int enabled_p*) [C Function]
Enable or disable automatic finalization. By default, Guile arranges to invoke object finalizers automatically, in a separate thread if possible. Passing a zero value for *enabled_p* will disable automatic finalization for Guile as a whole. If you disable automatic finalization, you will have to call `scm_run_finalizers ()` periodically.

Unlike most other Guile functions, you can call `scm_set_automatic_finalization_enabled` before Guile has been initialized.

Return the previous status of automatic finalization.

`int scm_run_finalizers` (*void*) [C Function]
Invoke any pending finalizers. Returns the number of finalizers that were invoked. This function should be called when automatic finalization is disabled, though it may be called if it is enabled as well.

`void scm_assert_foreign_object_type` (*SCM type, SCM val*) [C Function]
When *val* is a foreign object of the given *type*, do nothing. Otherwise, signal an error.

`SCM scm_make_foreign_object_0` (*SCM type*) [C Function]
`SCM scm_make_foreign_object_1` (*SCM type, void *val0*) [C Function]
`SCM scm_make_foreign_object_2` (*SCM type, void *val0, void *val1*) [C Function]
`SCM scm_make_foreign_object_3` (*SCM type, void *val0, void *val1, void *val2*) [C Function]
`SCM scm_make_foreign_object_n` (*SCM type, size_t n, void *vals*[]) [C Function]
Make a new foreign object of the type with type *type* and initialize the first *n* fields to the given values, as appropriate.

The number of fields for objects of a given type is fixed when the type is created. It is an error to give more initializers than there are fields in the value. It is perfectly fine to give fewer initializers than needed; this is convenient when some fields are of non-pointer types, and would be easier to initialize with the setters described below.

`void* scm_foreign_object_ref` (*SCM obj, size_t n*); [C Function]
`scm_t_bits scm_foreign_object_unsigned_ref` (*SCM obj, size_t n*); [C Function]
`scm_t_signed_bits scm_foreign_object_signed_ref` (*SCM obj, size_t n*); [C Function]

Return the value of the *n*th field of the foreign object *obj*. The backing store for the fields is as wide as a `scm_t_bits` value, which is at least as wide as a pointer. The different variants handle casting in a portable way.

`void scm_foreign_object_set_x` (*SCM obj, size_t n, void *val*); [C Function]
`void scm_foreign_object_unsigned_set_x` (*SCM obj, size_t n, scm_t_bits val*); [C Function]
`void scm_foreign_object_signed_set_x` (*SCM obj, size_t n, scm_t_signed_bits val*); [C Function]

Set the value of the *n*th field of the foreign object *obj* to *val*, after portably converting to a `scm_t_bits` value, if needed.

One can also access foreign objects from Scheme. See Section 5.5.5 [Foreign Objects and Scheme], page 80, for some examples.

```
(use-modules (system foreign-object))
```

`make-foreign-object-type` *name slots* [*#:finalizer=#f*] [Scheme Procedure]

Make a new foreign object type. See the above documentation for `scm_make_foreign_object_type`; these functions are exactly equivalent, except for the way in which the finalizer gets attached to instances (an internal detail).

The resulting value is a GOOPS class. See Chapter 8 [GOOPS], page 773, for more on classes in Guile.

`define-foreign-object-type` *name constructor* (*slot ...*) [*#:finalizer=#f*] [Scheme Syntax]

A convenience macro to define a type, using `make-foreign-object-type`, and bind it to *name*. A constructor will be bound to *constructor*, and getters will be bound to each of *slot...*.

6.8 Smobs

A *smob* is a "small object". Before foreign objects were introduced in Guile 2.0.12 (see Section 6.7 [Foreign Objects], page 243), smobs were the preferred way to for C code to define new kinds of Scheme objects. With the exception of the so-called "applicable SMOBs" discussed below, smobs are now a legacy interface and are headed for eventual deprecation. See Section 6.2 [Deprecation], page 100. New code should use the foreign object interface.

This section contains reference information related to defining and working with smobs. For a tutorial-like introduction to smobs, see "Defining New Types (Smobs)" in previous versions of this manual.

`scm_t_bits scm_make_smob_type` (*const char *name, size_t size*) [Function]

This function adds a new smob type, named *name*, with instance size *size*, to the system. The return value is a tag that is used in creating instances of the type.

If *size* is 0, the default *free* function will do nothing.

If *size* is not 0, the default *free* function will deallocate the memory block pointed to by `SCM_SMOB_DATA` with `scm_gc_free`. The *what* parameter in the call to `scm_gc_free` will be *name*.

Default values are provided for the *mark*, *free*, *print*, and *equalp* functions. If you want to customize any of these functions, the call to `scm_make_smob_type` should be immediately followed by calls to one or several of `scm_set_smob_mark`, `scm_set_smob_free`, `scm_set_smob_print`, and/or `scm_set_smob_equalp`.

`void scm_set_smob_free` (*scm_t_bits tc*, *size_t* (**free*) (*SCM obj*)) [C Function]

This function sets the smob freeing procedure (sometimes referred to as a *finalizer*) for the smob type specified by the tag *tc*. *tc* is the tag returned by `scm_make_smob_type`.

The *free* procedure must deallocate all resources that are directly associated with the smob instance *obj*. It must assume that all `SCM` values that it references have already been freed and are thus invalid.

It must also not call any libguile function or macro except `scm_gc_free`, `SCM_SMOB_FLAGS`, `SCM_SMOB_DATA`, `SCM_SMOB_DATA_2`, and `SCM_SMOB_DATA_3`.

The *free* procedure must return 0.

Note that defining a freeing procedure is not necessary if the resources associated with *obj* consists only of memory allocated with `scm_gc_malloc` or `scm_gc_malloc_pointerless` because this memory is automatically reclaimed by the garbage collector when it is no longer needed (see Section 6.19.2 [Memory Blocks], page 405).

Smob free functions must be thread-safe. See Section 5.5.4 [Foreign Object Memory Management], page 77, for a discussion on finalizers and concurrency. If you are embedding Guile in an application that is not thread-safe, and you define smob types that need finalization, you might want to disable automatic finalization, and arrange to call `scm_manually_run_finalizers ()` yourself. See Section 6.7 [Foreign Objects], page 243.

`void scm_set_smob_mark` (*scm_t_bits tc*, *SCM* (**mark*) (*SCM obj*)) [C Function]

This function sets the smob marking procedure for the smob type specified by the tag *tc*. *tc* is the tag returned by `scm_make_smob_type`.

Defining a marking procedure is almost always the wrong thing to do. It is much, much preferable to allocate smob data with the `scm_gc_malloc` and `scm_gc_malloc_pointerless` functions, and allow the GC to trace pointers automatically.

Any mark procedures you see currently almost surely date from the time of Guile 1.8, before the switch to the Boehm-Demers-Weiser collector. Such smob implementations should be changed to just use `scm_gc_malloc` and friends, and to lose their mark function.

If you decide to keep the mark function, note that it may be called on objects that are on the free list. Please read and digest the comments from the BDW GC's `gc/gc_mark.h` header.

The *mark* procedure must cause `scm_gc_mark` to be called for every `SCM` value that is directly referenced by the smob instance *obj*. One of these `SCM` values can be returned

from the procedure and Guile will call `scm_gc_mark` for it. This can be used to avoid deep recursions for smob instances that form a list.

It must not call any libguile function or macro except `scm_gc_mark`, `SCM_SMOB_FLAGS`, `SCM_SMOB_DATA`, `SCM_SMOB_DATA_2`, and `SCM_SMOB_DATA_3`.

`void scm_set_smob_print` (*scm_t_bits tc, int (*print) (SCM obj, SCM port, scm_print_state* pstate)*) [C Function]

This function sets the smob printing procedure for the smob type specified by the tag *tc*. *tc* is the tag returned by `scm_make_smob_type`.

The *print* procedure should output a textual representation of the smob instance *obj* to *port*, using information in *pstate*.

The textual representation should be of the form `#<name ...>`. This ensures that `read` will not interpret it as some other Scheme value.

It is often best to ignore *pstate* and just print to *port* with `scm_display`, `scm_write`, `scm_simple_format`, and `scm_puts`.

`void scm_set_smob_equalp` (*scm_t_bits tc, SCM (*equalp) (SCM obj1, SCM obj2)*) [C Function]

This function sets the smob equality-testing predicate for the smob type specified by the tag *tc*. *tc* is the tag returned by `scm_make_smob_type`.

The *equalp* procedure should return `SCM_BOOL_T` when *obj1* is `equal?` to *obj2*. Else it should return `SCM_BOOL_F`. Both *obj1* and *obj2* are instances of the smob type *tc*.

`void scm_assert_smob_type` (*scm_t_bits tag, SCM val*) [C Function]

When *val* is a smob of the type indicated by *tag*, do nothing. Else, signal an error.

`int SCM_SMOB_PREDICATE` (*scm_t_bits tag, SCM exp*) [C Macro]

Return true if *exp* is a smob instance of the type indicated by *tag*, or false otherwise. The expression *exp* can be evaluated more than once, so it shouldn't contain any side effects.

`SCM scm_new_smob` (*scm_t_bits tag, void *data*) [C Function]

`SCM scm_new_double_smob` (*scm_t_bits tag, void *data, void *data2, void *data3*) [C Function]

Make a new smob of the type with tag *tag* and smob data *data*, *data2*, and *data3*, as appropriate.

The *tag* is what has been returned by `scm_make_smob_type`. The initial values *data*, *data2*, and *data3* are of type `scm_t_bits`; when you want to use them for `SCM` values, these values need to be converted to a `scm_t_bits` first by using `SCM_UNPACK`.

The flags of the smob instance start out as zero.

`scm_t_bits SCM_SMOB_FLAGS` (*SCM obj*) [C Macro]

Return the 16 extra bits of the smob *obj*. No meaning is predefined for these bits, you can use them freely.

`scm_t_bits SCM_SET_SMOB_FLAGS` (*SCM obj, scm_t_bits flags*) [C Macro]

Set the 16 extra bits of the smob *obj* to *flags*. No meaning is predefined for these bits, you can use them freely.

`scm_t_bits SCM_SMOB_DATA` (*SCM obj*) [C Macro]
`scm_t_bits SCM_SMOB_DATA_2` (*SCM obj*) [C Macro]
`scm_t_bits SCM_SMOB_DATA_3` (*SCM obj*) [C Macro]
Return the first (second, third) immediate word of the smob *obj* as a `scm_t_bits` value. When the word contains a `SCM` value, use `SCM_SMOB_OBJECT` (etc.) instead.

`void SCM_SET_SMOB_DATA` (*SCM obj, scm_t_bits val*) [C Macro]
`void SCM_SET_SMOB_DATA_2` (*SCM obj, scm_t_bits val*) [C Macro]
`void SCM_SET_SMOB_DATA_3` (*SCM obj, scm_t_bits val*) [C Macro]
Set the first (second, third) immediate word of the smob *obj* to *val*. When the word should be set to a `SCM` value, use `SCM_SMOB_SET_OBJECT` (etc.) instead.

`SCM SCM_SMOB_OBJECT` (*SCM obj*) [C Macro]
`SCM SCM_SMOB_OBJECT_2` (*SCM obj*) [C Macro]
`SCM SCM_SMOB_OBJECT_3` (*SCM obj*) [C Macro]
Return the first (second, third) immediate word of the smob *obj* as a `SCM` value. When the word contains a `scm_t_bits` value, use `SCM_SMOB_DATA` (etc.) instead.

`void SCM_SET_SMOB_OBJECT` (*SCM obj, SCM val*) [C Macro]
`void SCM_SET_SMOB_OBJECT_2` (*SCM obj, SCM val*) [C Macro]
`void SCM_SET_SMOB_OBJECT_3` (*SCM obj, SCM val*) [C Macro]
Set the first (second, third) immediate word of the smob *obj* to *val*. When the word should be set to a `scm_t_bits` value, use `SCM_SMOB_SET_DATA` (etc.) instead.

`SCM * SCM_SMOB_OBJECT_LOC` (*SCM obj*) [C Macro]
`SCM * SCM_SMOB_OBJECT_2_LOC` (*SCM obj*) [C Macro]
`SCM * SCM_SMOB_OBJECT_3_LOC` (*SCM obj*) [C Macro]
Return a pointer to the first (second, third) immediate word of the smob *obj*. Note that this is a pointer to `SCM`. If you need to work with `scm_t_bits` values, use `SCM_PACK` and `SCM_UNPACK`, as appropriate.

`SCM scm_markcdr` (*SCM x*) [Function]
Mark the references in the smob *x*, assuming that *x*'s first data word contains an ordinary Scheme object, and *x* refers to no other objects. This function simply returns *x*'s first data word.

6.9 Procedures

6.9.1 Lambda: Basic Procedure Creation

A `lambda` expression evaluates to a procedure. The environment which is in effect when a `lambda` expression is evaluated is enclosed in the newly created procedure, this is referred to as a *closure* (see Section 3.4 [About Closure], page 26).

When a procedure created by `lambda` is called with some actual arguments, the environment enclosed in the procedure is extended by binding the variables named in the formal argument list to new locations and storing the actual arguments into these locations. Then the body of the `lambda` expression is evaluated sequentially. The result of the last expression in the procedure body is then the result of the procedure invocation.

The following examples will show how procedures can be created using `lambda`, and what you can do with these procedures.

```
(lambda (x) (+ x x))        ⇒ a procedure
((lambda (x) (+ x x)) 4)    ⇒ 8
```

The fact that the environment in effect when creating a procedure is enclosed in the procedure is shown with this example:

```
(define add4
  (let ((x 4))
    (lambda (y) (+ x y))))
(add4 6)                    ⇒ 10
```

`lambda` *formals body* [syntax]

formals should be a formal argument list as described in the following table.

`(`*variable1* `...)`
: The procedure takes a fixed number of arguments; when the procedure is called, the arguments will be stored into the newly created location for the formal variables.

variable
: The procedure takes any number of arguments; when the procedure is called, the sequence of actual arguments will converted into a list and stored into the newly created location for the formal variable.

`(`*variable1 ... variablen . variablen+1*`)`
: If a space-delimited period precedes the last variable, then the procedure takes *n* or more variables where *n* is the number of formal arguments before the period. There must be at least one argument before the period. The first *n* actual arguments will be stored into the newly allocated locations for the first *n* formal arguments and the sequence of the remaining actual arguments is converted into a list and the stored into the location for the last formal argument. If there are exactly *n* actual arguments, the empty list is stored into the location of the last formal argument.

The list in *variable* or *variablen+1* is always newly created and the procedure can modify it if desired. This is the case even when the procedure is invoked via `apply`, the required part of the list argument there will be copied (see Section 6.18.4 [Procedures for On the Fly Evaluation], page 387).

body is a sequence of Scheme expressions which are evaluated in order when the procedure is invoked.

6.9.2 Primitive Procedures

Procedures written in C can be registered for use from Scheme, provided they take only arguments of type `SCM` and return `SCM` values. `scm_c_define_gsubr` is likely to be the most useful mechanism, combining the process of registration (`scm_c_make_gsubr`) and definition (`scm_define`).

`SCM scm_c_make_gsubr` (*const char *name, int req, int opt, int rst, fcn*) [Function]

Register a C procedure *fcn* as a "subr" — a primitive subroutine that can be called from Scheme. It will be associated with the given *name* but no environment binding

will be created. The arguments *req*, *opt* and *rst* specify the number of required, optional and "rest" arguments respectively. The total number of these arguments should match the actual number of arguments to *fcn*, but may not exceed 10. The number of rest arguments should be 0 or 1. `scm_c_make_gsubr` returns a value of type `SCM` which is a "handle" for the procedure.

`SCM scm_c_define_gsubr` (*const char *name, int req, int opt, int rst, fcn*) [Function]

Register a C procedure *fcn*, as for `scm_c_make_gsubr` above, and additionally create a top-level Scheme binding for the procedure in the "current environment" using `scm_define`. `scm_c_define_gsubr` returns a handle for the procedure in the same way as `scm_c_make_gsubr`, which is usually not further required.

6.9.3 Compiled Procedures

The evaluation strategy given in Section 6.9.1 [Lambda], page 248, describes how procedures are *interpreted*. Interpretation operates directly on expanded Scheme source code, recursively calling the evaluator to obtain the value of nested expressions.

Most procedures are compiled, however. This means that Guile has done some precomputation on the procedure, to determine what it will need to do each time the procedure runs. Compiled procedures run faster than interpreted procedures.

Loading files is the normal way that compiled procedures come to being. If Guile sees that a file is uncompiled, or that its compiled file is out of date, it will attempt to compile the file when it is loaded, and save the result to disk. Procedures can be compiled at runtime as well. See Section 6.18 [Read/Load/Eval/Compile], page 382, for more information on runtime compilation.

Compiled procedures, also known as *programs*, respond to all procedures that operate on procedures: you can pass a program to `procedure?`, `procedure-name`, and so on (see Section 6.9.7 [Procedure Properties], page 258). In addition, there are a few more accessors for low-level details on programs.

Most people won't need to use the routines described in this section, but it's good to have them documented. You'll have to include the appropriate module first, though:

```
(use-modules (system vm program))
```

`program?` *obj* [Scheme Procedure]
`scm_program_p` (*obj*) [C Function]

Returns `#t` if *obj* is a compiled procedure, or `#f` otherwise.

`program-code` *program* [Scheme Procedure]
`scm_program_code` (*program*) [C Function]

Returns the address of the program's entry, as an integer. This address is mostly useful to procedures in `(system vm debug)`.

`program-num-free-variable` *program* [Scheme Procedure]
`scm_program_num_free_variables` (*program*) [C Function]

Return the number of free variables captured by this program.

`program-free-variable-ref` *program n* [Scheme Procedure]
`scm_program_free_variable-ref` (*program, n*) [C Function]
`program-free-variable-set!` *program n val* [Scheme Procedure]
`scm_program_free_variable_set_x` (*program, n, val*) [C Function]

Accessors for a program's free variables. Some of the values captured are actually in variable "boxes". See Section 9.3.4 [Variables and the VM], page 830, for more information.

Users must not modify the returned value unless they think they're really clever.

`program-bindings` *program* [Scheme Procedure]
`make-binding` *name boxed? index start end* [Scheme Procedure]
`binding:name` *binding* [Scheme Procedure]
`binding:boxed?` *binding* [Scheme Procedure]
`binding:index` *binding* [Scheme Procedure]
`binding:start` *binding* [Scheme Procedure]
`binding:end` *binding* [Scheme Procedure]

Bindings annotations for programs, along with their accessors.

Bindings declare names and liveness extents for block-local variables. The best way to see what these are is to play around with them at a REPL. See Section 9.3.2 [VM Concepts], page 828, for more information.

Note that bindings information is stored in a program as part of its metadata thunk, so including it in the generated object code does not impose a runtime performance penalty.

`program-sources` *program* [Scheme Procedure]
`source:addr` *source* [Scheme Procedure]
`source:line` *source* [Scheme Procedure]
`source:column` *source* [Scheme Procedure]
`source:file` *source* [Scheme Procedure]

Source location annotations for programs, along with their accessors.

Source location information propagates through the compiler and ends up being serialized to the program's metadata. This information is keyed by the offset of the instruction pointer within the object code of the program. Specifically, it is keyed on the `ip` *just following* an instruction, so that backtraces can find the source location of a call that is in progress.

`program-arities` *program* [Scheme Procedure]
`scm_program_arities` (*program*) [C Function]
`program-arity` *program ip* [Scheme Procedure]
`arity:start` *arity* [Scheme Procedure]
`arity:end` *arity* [Scheme Procedure]
`arity:nreq` *arity* [Scheme Procedure]
`arity:nopt` *arity* [Scheme Procedure]
`arity:rest?` *arity* [Scheme Procedure]
`arity:kw` *arity* [Scheme Procedure]
`arity:allow-other-keys?` *arity* [Scheme Procedure]

Accessors for a representation of the "arity" of a program.

The normal case is that a procedure has one arity. For example, `(lambda (x) x)`, takes one required argument, and that's it. One could access that number of required arguments via `(arity:nreq (program-arities (lambda (x) x)))`. Similarly, `arity:nopt` gets the number of optional arguments, and `arity:rest?` returns a true value if the procedure has a rest arg.

`arity:kw` returns a list of `(`*kw* `.` *idx*`)` pairs, if the procedure has keyword arguments. The *idx* refers to the *idx*th local variable; See Section 9.3.4 [Variables and the VM], page 830, for more information. Finally `arity:allow-other-keys?` returns a true value if other keys are allowed. See Section 6.9.4 [Optional Arguments], page 252, for more information.

So what about `arity:start` and `arity:end`, then? They return the range of bytes in the program's bytecode for which a given arity is valid. You see, a procedure can actually have more than one arity. The question, "what is a procedure's arity" only really makes sense at certain points in the program, delimited by these `arity:start` and `arity:end` values.

`program-arguments-alist` *program* [*ip*] [Scheme Procedure]

Return an association list describing the arguments that *program* accepts, or `#f` if the information cannot be obtained.

The alist keys that are currently defined are 'required', 'optional', 'keyword', 'allow-other-keys?', and 'rest'. For example:

```
(program-arguments-alist
 (lambda* (a b #:optional c #:key (d 1) #:rest e)
   #t)) ⇒
((required . (a b))
 (optional . (c))
 (keyword . ((#:d . 4)))
 (allow-other-keys? . #f)
 (rest . d))
```

`program-lambda-list` *program* [*ip*] [Scheme Procedure]

Return a representation of the arguments of *program* as a lambda list, or `#f` if this information is not available.

For example:

```
(program-lambda-list
 (lambda* (a b #:optional c #:key (d 1) #:rest e)
   #t)) ⇒
```

6.9.4 Optional Arguments

Scheme procedures, as defined in R5RS, can either handle a fixed number of actual arguments, or a fixed number of actual arguments followed by arbitrarily many additional arguments. Writing procedures of variable arity can be useful, but unfortunately, the syntactic means for handling argument lists of varying length is a bit inconvenient. It is possible to give names to the fixed number of arguments, but the remaining (optional) arguments can be only referenced as a list of values (see Section 6.9.1 [Lambda], page 248).

For this reason, Guile provides an extension to `lambda`, `lambda*`, which allows the user to define procedures with optional and keyword arguments. In addition, Guile's virtual machine has low-level support for optional and keyword argument dispatch. Calls to procedures with optional and keyword arguments can be made cheaply, without allocating a rest list.

6.9.4.1 lambda* and define*.

`lambda*` is like `lambda`, except with some extensions to allow optional and keyword arguments.

`lambda*` ([*var*...] [library syntax]
[*#:optional vardef*...]
[*#:key vardef*... [*#:allow-other-keys*]]
[*#:rest var* | . *var*])
body1 body2 ...

Create a procedure which takes optional and/or keyword arguments specified with `#:optional` and `#:key`. For example,

```
(lambda* (a b #:optional c d . e) '())
```

is a procedure with fixed arguments *a* and *b*, optional arguments *c* and *d*, and rest argument *e*. If the optional arguments are omitted in a call, the variables for them are bound to `#f`.

Likewise, `define*` is syntactic sugar for defining procedures using `lambda*`.

`lambda*` can also make procedures with keyword arguments. For example, a procedure defined like this:

```
(define* (sir-yes-sir #:key action how-high)
  (list action how-high))
```

can be called as `(sir-yes-sir #:action 'jump)`, `(sir-yes-sir #:how-high 13)`, `(sir-yes-sir #:action 'lay-down #:how-high 0)`, or just `(sir-yes-sir)`. Whichever arguments are given as keywords are bound to values (and those not given are `#f`).

Optional and keyword arguments can also have default values to take when not present in a call, by giving a two-element list of variable name and expression. For example in

```
(define* (frob foo #:optional (bar 42) #:key (baz 73))
  (list foo bar baz))
```

foo is a fixed argument, *bar* is an optional argument with default value 42, and baz is a keyword argument with default value 73. Default value expressions are not evaluated unless they are needed, and until the procedure is called.

Normally it's an error if a call has keywords other than those specified by `#:key`, but adding `#:allow-other-keys` to the definition (after the keyword argument declarations) will ignore unknown keywords.

If a call has a keyword given twice, the last value is used. For example,

```
(define* (flips #:key (heads 0) (tails 0))
```

```
  (display (list heads tails)))

(flips #:heads 37 #:tails 42 #:heads 99)
⊣ (99 42)
```

`#:rest` is a synonym for the dotted syntax rest argument. The argument lists `(a . b)` and `(a #:rest b)` are equivalent in all respects. This is provided for more similarity to DSSSL, MIT-Scheme and Kawa among others, as well as for refugees from other Lisp dialects.

When `#:key` is used together with a rest argument, the keyword parameters in a call all remain in the rest list. This is the same as Common Lisp. For example,

```
((lambda* (#:key (x 0) #:allow-other-keys #:rest r)
   (display r))
 #:x 123 #:y 456)
⊣ (#:x 123 #:y 456)
```

`#:optional` and `#:key` establish their bindings successively, from left to right. This means default expressions can refer back to prior parameters, for example

```
(lambda* (start #:optional (end (+ 10 start)))
  (do ((i start (1+ i)))
      ((> i end))
    (display i)))
```

The exception to this left-to-right scoping rule is the rest argument. If there is a rest argument, it is bound after the optional arguments, but before the keyword arguments.

6.9.4.2 (ice-9 optargs)

Before Guile 2.0, `lambda*` and `define*` were implemented using macros that processed rest list arguments. This was not optimal, as calling procedures with optional arguments had to allocate rest lists at every procedure invocation. Guile 2.0 improved this situation by bringing optional and keyword arguments into Guile's core.

However there are occasions in which you have a list and want to parse it for optional or keyword arguments. Guile's `(ice-9 optargs)` provides some macros to help with that task.

The syntax `let-optional` and `let-optional*` are for destructuring rest argument lists and giving names to the various list elements. `let-optional` binds all variables simultaneously, while `let-optional*` binds them sequentially, consistent with `let` and `let*` (see Section 6.12.2 [Local Bindings], page 294).

`let-optional` *rest-arg* (*binding ...*) *body1 body2 ...* [library syntax]
`let-optional*` *rest-arg* (*binding ...*) *body1 body2 ...* [library syntax]

These two macros give you an optional argument interface that is very *Schemey* and introduces no fancy syntax. They are compatible with the scsh macros of the same name, but are slightly extended. Each of *binding* may be of one of the forms *var* or (*var default-value*). *rest-arg* should be the rest-argument of the procedures these are used from. The items in *rest-arg* are sequentially bound to the variable names are given. When *rest-arg* runs out, the remaining vars are bound either to the default

values or `#f` if no default value was specified. *rest-arg* remains bound to whatever may have been left of *rest-arg*.

After binding the variables, the expressions *body1 body2 ...* are evaluated in order.

Similarly, `let-keywords` and `let-keywords*` extract values from keyword style argument lists, binding local variables to those values or to defaults.

`let-keywords` *args allow-other-keys? (binding ...) body1 body2 ...* [library syntax]

`let-keywords*` *args allow-other-keys? (binding ...) body1 body2 ...* [library syntax]

args is evaluated and should give a list of the form `(#:keyword1 value1 #:keyword2 value2 ...)`. The *binding*s are variables and default expressions, with the variables to be set (by name) from the keyword values. The *body1 body2 ...* forms are then evaluated and the last is the result. An example will make the syntax clearest,

```
(define args '(#:xyzzy "hello" #:foo "world"))

(let-keywords args #t
      ((foo  "default for foo")
       (bar  (string-append "default" "for" "bar")))
  (display foo)
  (display ", ")
  (display bar))
⊣ world, defaultforbar
```

The binding for `foo` comes from the `#:foo` keyword in `args`. But the binding for `bar` is the default in the `let-keywords`, since there's no `#:bar` in the args.

allow-other-keys? is evaluated and controls whether unknown keywords are allowed in the *args* list. When true other keys are ignored (such as `#:xyzzy` in the example), when `#f` an error is thrown for anything unknown.

`(ice-9 optargs)` also provides some more `define*` sugar, which is not so useful with modern Guile coding, but still supported: `define*-public` is the `lambda*` version of `define-public`; `defmacro*` and `defmacro*-public` exist for defining macros with the improved argument list handling possibilities. The `-public` versions not only define the procedures/macros, but also export them from the current module.

`define*-public` *formals body1 body2 ...* [library syntax]

Like a mix of `define*` and `define-public`.

`defmacro*` *name formals body1 body2 ...* [library syntax]

`defmacro*-public` *name formals body1 body2 ...* [library syntax]

These are just like `defmacro` and `defmacro-public` except that they take `lambda*`-style extended parameter lists, where `#:optional`, `#:key`, `#:allow-other-keys` and `#:rest` are allowed with the usual semantics. Here is an example of a macro with an optional argument:

```
(defmacro* transmogrify (a #:optional b)
  (a 1))
```

6.9.5 Case-lambda

R5RS's rest arguments are indeed useful and very general, but they often aren't the most appropriate or efficient means to get the job done. For example, `lambda*` is a much better solution to the optional argument problem than `lambda` with rest arguments.

Likewise, `case-lambda` works well for when you want one procedure to do double duty (or triple, or ...), without the penalty of consing a rest list.

For example:

```
(define (make-accum n)
  (case-lambda
    (() n)
    ((m) (set! n (+ n m)) n)))

(define a (make-accum 20))
(a) ⇒ 20
(a 10) ⇒ 30
(a) ⇒ 30
```

The value returned by a `case-lambda` form is a procedure which matches the number of actual arguments against the formals in the various clauses, in order. The first matching clause is selected, the corresponding values from the actual parameter list are bound to the variable names in the clauses and the body of the clause is evaluated. If no clause matches, an error is signalled.

The syntax of the `case-lambda` form is defined in the following EBNF grammar. *Formals* means a formal argument list just like with `lambda` (see Section 6.9.1 [Lambda], page 248).

```
<case-lambda>
   --> (case-lambda <case-lambda-clause>*)
   --> (case-lambda <docstring> <case-lambda-clause>*)
<case-lambda-clause>
   --> (<formals> <definition-or-command>*)
<formals>
   --> (<identifier>*)
     | (<identifier>* . <identifier>)
     | <identifier>
```

Rest lists can be useful with `case-lambda`:

```
(define plus
  (case-lambda
    "Return the sum of all arguments."
    (() 0)
    ((a) a)
    ((a b) (+ a b))
    ((a b . rest) (apply plus (+ a b) rest))))
(plus 1 2 3) ⇒ 6
```

Also, for completeness. Guile defines `case-lambda*` as well, which is like `case-lambda`, except with `lambda*` clauses. A `case-lambda*` clause matches if the arguments fill the required arguments, but are not too many for the optional and/or rest arguments.

Keyword arguments are possible with `case-lambda*` as well, but they do not contribute to the "matching" behavior, and their interactions with required, optional, and rest arguments can be surprising.

For the purposes of `case-lambda*` (and of `case-lambda`, as a special case), a clause *matches* if it has enough required arguments, and not too many positional arguments. The required arguments are any arguments before the `#:optional`, `#:key`, and `#:rest` arguments. *Positional* arguments are the required arguments, together with the optional arguments.

In the absence of `#:key` or `#:rest` arguments, it's easy to see how there could be too many positional arguments: you pass 5 arguments to a function that only takes 4 arguments, including optional arguments. If there is a `#:rest` argument, there can never be too many positional arguments: any application with enough required arguments for a clause will match that clause, even if there are also `#:key` arguments.

Otherwise, for applications to a clause with `#:key` arguments (and without a `#:rest` argument), a clause will match there only if there are enough required arguments and if the next argument after binding required and optional arguments, if any, is a keyword. For efficiency reasons, Guile is currently unable to include keyword arguments in the matching algorithm. Clauses match on positional arguments only, not by comparing a given keyword to the available set of keyword arguments that a function has.

Some examples follow.

```
(define f
  (case-lambda*
    ((a #:optional b) 'clause-1)
    ((a #:optional b #:key c) 'clause-2)
    ((a #:key d) 'clause-3)
    ((#:key e #:rest f) 'clause-4)))

(f) ⇒ clause-4
(f 1) ⇒ clause-1
(f) ⇒ clause-4
(f #:e 10) clause-1
(f 1 #:foo) clause-1
(f 1 #:c 2) clause-2
(f #:a #:b #:c #:d #:e) clause-4

;; clause-2 will match anything that clause-3 would match.
(f 1 #:d 2) ⇒ error: bad keyword args in clause 2
```

Don't forget that the clauses are matched in order, and the first matching clause will be taken. This can result in a keyword being bound to a required argument, as in the case of `f #:e 10`.

6.9.6 Higher-Order Functions

As a functional programming language, Scheme allows the definition of *higher-order functions*, i.e., functions that take functions as arguments and/or return functions. Utilities to derive procedures from other procedures are provided and described below.

`const` *value* [Scheme Procedure]

Return a procedure that accepts any number of arguments and returns *value*.

```
(procedure? (const 3))        ⇒ #t
((const 'hello))              ⇒ hello
((const 'hello) 'world)       ⇒ hello
```

`negate` *proc* [Scheme Procedure]

Return a procedure with the same arity as *proc* that returns the `not` of *proc*'s result.

```
(procedure? (negate number?)) ⇒ #t
((negate odd?) 2)             ⇒ #t
((negate real?) 'dream)       ⇒ #t
((negate string-prefix?) "GNU" "GNU Guile")
                              ⇒ #f
(filter (negate number?) '(a 2 "b"))
                              ⇒ (a "b")
```

`compose` *proc1 proc2 ...* [Scheme Procedure]

Compose *proc1* with the procedures *proc2 ...* such that the last *proc* argument is applied first and *proc1* last, and return the resulting procedure. The given procedures must have compatible arity.

```
(procedure? (compose 1+ 1-)) ⇒ #t
((compose sqrt 1+ 1+) 2)     ⇒ 2.0
((compose 1+ sqrt) 3)        ⇒ 2.73205080756888
(eq? (compose 1+) 1+)        ⇒ #t

((compose zip unzip2) '((1 2) (a b)))
                             ⇒ ((1 2) (a b))
```

`identity` *x* [Scheme Procedure]

Return X.

`and=>` *value proc* [Scheme Procedure]

When *value* is `#f`, return `#f`. Otherwise, return (*proc value*).

6.9.7 Procedure Properties and Meta-information

In addition to the information that is strictly necessary to run, procedures may have other associated information. For example, the name of a procedure is information not for the procedure, but about the procedure. This meta-information can be accessed via the procedure properties interface.

The first group of procedures in this meta-interface are predicates to test whether a Scheme object is a procedure, or a special procedure, respectively. `procedure?` is the most general predicates, it returns `#t` for any kind of procedure.

`procedure?` *obj* [Scheme Procedure]
`scm_procedure_p` (*obj*) [C Function]

Return `#t` if *obj* is a procedure.

`thunk?` *obj* [Scheme Procedure]
`scm_thunk_p` (*obj*) [C Function]
Return `#t` if *obj* is a procedure that can be called with zero arguments.

Procedure properties are general properties associated with procedures. These can be the name of a procedure or other relevant information, such as debug hints.

`procedure-name` *proc* [Scheme Procedure]
`scm_procedure_name` (*proc*) [C Function]
Return the name of the procedure *proc*

`procedure-source` *proc* [Scheme Procedure]
`scm_procedure_source` (*proc*) [C Function]
Return the source of the procedure *proc*. Returns `#f` if the source code is not available.

`procedure-properties` *proc* [Scheme Procedure]
`scm_procedure_properties` (*proc*) [C Function]
Return the properties associated with *proc*, as an association list.

`procedure-property` *proc key* [Scheme Procedure]
`scm_procedure_property` (*proc*, *key*) [C Function]
Return the property of *proc* with name *key*.

`set-procedure-properties!` *proc alist* [Scheme Procedure]
`scm_set_procedure_properties_x` (*proc*, *alist*) [C Function]
Set *proc*'s property list to *alist*.

`set-procedure-property!` *proc key value* [Scheme Procedure]
`scm_set_procedure_property_x` (*proc*, *key*, *value*) [C Function]
In *proc*'s property list, set the property named *key* to *value*.

Documentation for a procedure can be accessed with the procedure `procedure-documentation`.

`procedure-documentation` *proc* [Scheme Procedure]
`scm_procedure_documentation` (*proc*) [C Function]
Return the documentation string associated with `proc`. By convention, if a procedure contains more than one expression and the first expression is a string constant, that string is assumed to contain documentation for that procedure.

6.9.8 Procedures with Setters

A *procedure with setter* is a special kind of procedure which normally behaves like any accessor procedure, that is a procedure which accesses a data structure. The difference is that this kind of procedure has a so-called *setter* attached, which is a procedure for storing something into a data structure.

Procedures with setters are treated specially when the procedure appears in the special form `set!` (REFFIXME). How it works is best shown by example.

Suppose we have a procedure called `foo-ref`, which accepts two arguments, a value of type `foo` and an integer. The procedure returns the value stored at the given index in the `foo` object. Let `f` be a variable containing such a `foo` data structure.[4]

```
(foo-ref f 0)       ⇒ bar
(foo-ref f 1)       ⇒ braz
```

Also suppose that a corresponding setter procedure called `foo-set!` does exist.

```
(foo-set! f 0 'bla)
(foo-ref f 0)       ⇒ bla
```

Now we could create a new procedure called `foo`, which is a procedure with setter, by calling `make-procedure-with-setter` with the accessor and setter procedures `foo-ref` and `foo-set!`. Let us call this new procedure `foo`.

```
(define foo (make-procedure-with-setter foo-ref foo-set!))
```

`foo` can from now on be used to either read from the data structure stored in `f`, or to write into the structure.

```
(set! (foo f 0) 'dum)
(foo f 0)           ⇒ dum
```

`make-procedure-with-setter` *procedure setter* [Scheme Procedure]
`scm_make_procedure_with_setter` (*procedure, setter*) [C Function]
Create a new procedure which behaves like *procedure*, but with the associated setter *setter*.

`procedure-with-setter?` *obj* [Scheme Procedure]
`scm_procedure_with_setter_p` (*obj*) [C Function]
Return `#t` if *obj* is a procedure with an associated setter procedure.

`procedure` *proc* [Scheme Procedure]
`scm_procedure` (*proc*) [C Function]
Return the procedure of *proc*, which must be an applicable struct.

`setter` *proc* [Scheme Procedure]
Return the setter of *proc*, which must be either a procedure with setter or an operator struct.

6.9.9 Inlinable Procedures

You can define an *inlinable procedure* by using `define-inlinable` instead of `define`. An inlinable procedure behaves the same as a regular procedure, but direct calls will result in the procedure body being inlined into the caller.

Bear in mind that starting from version 2.0.3, Guile has a partial evaluator that can inline the body of inner procedures when deemed appropriate:

```
scheme@(guile-user)> ,optimize (define (foo x)
```

[4] Working definitions would be:

```
(define foo-ref vector-ref)
(define foo-set! vector-set!)
(define f (make-vector 2 #f))
```

```
                                        (define (bar) (+ x 3))
                                        (* (bar) 2))
$1 = (define foo
       (lambda (#{x 94}#) (* (+ #{x 94}# 3) 2)))
```

The partial evaluator does not inline top-level bindings, though, so this is a situation where you may find it interesting to use `define-inlinable`.

Procedures defined with `define-inlinable` are *always* inlined, at all direct call sites. This eliminates function call overhead at the expense of an increase in code size. Additionally, the caller will not transparently use the new definition if the inline procedure is redefined. It is not possible to trace an inlined procedures or install a breakpoint in it (see Section 6.26.4 [Traps], page 484). For these reasons, you should not make a procedure inlinable unless it demonstrably improves performance in a crucial way.

In general, only small procedures should be considered for inlining, as making large procedures inlinable will probably result in an increase in code size. Additionally, the elimination of the call overhead rarely matters for large procedures.

`define-inlinable` (*name parameter ...*) *body1 body2 ...* [Scheme Syntax]

Define *name* as a procedure with parameters *parameter*s and bodies *body1*, *body2*, ...

6.10 Macros

At its best, programming in Lisp is an iterative process of building up a language appropriate to the problem at hand, and then solving the problem in that language. Defining new procedures is part of that, but Lisp also allows the user to extend its syntax, with its famous *macros*.

Macros are syntactic extensions which cause the expression that they appear in to be transformed in some way *before* being evaluated. In expressions that are intended for macro transformation, the identifier that names the relevant macro must appear as the first element, like this:

```
(macro-name macro-args ...)
```

Macro expansion is a separate phase of evaluation, run before code is interpreted or compiled. A macro is a program that runs on programs, translating an embedded language into core Scheme[5].

6.10.1 Defining Macros

A macro is a binding between a keyword and a syntax transformer. Since it's difficult to discuss `define-syntax` without discussing the format of transformers, consider the following example macro definition:

```
(define-syntax when
  (syntax-rules ()
    ((when condition exp ...)
     (if condition
```

[5] These days such embedded languages are often referred to as *embedded domain-specific languages*, or EDSLs.

```
          (begin exp ...)))))

(when #t
  (display "hey ho\n")
  (display "let's go\n"))
⊣ hey ho
⊣ let's go
```

In this example, the `when` binding is bound with `define-syntax`. Syntax transformers are discussed in more depth in Section 6.10.2 [Syntax Rules], page 263, and Section 6.10.3 [Syntax Case], page 268.

`define-syntax` *keyword transformer* [Syntax]

Bind *keyword* to the syntax transformer obtained by evaluating *transformer*.

After a macro has been defined, further instances of *keyword* in Scheme source code will invoke the syntax transformer defined by *transformer*.

One can also establish local syntactic bindings with `let-syntax`.

`let-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Syntax]

Bind each *keyword* to its corresponding *transformer* while expanding *exp1 exp2 ...*.

A `let-syntax` binding only exists at expansion-time.

```
(let-syntax ((unless
              (syntax-rules ()
                ((unless condition exp ...)
                 (if (not condition)
                     (begin exp ...))))))
  (unless #t
    (primitive-exit 1))
  "rock rock rock")
⇒ "rock rock rock"
```

A `define-syntax` form is valid anywhere a definition may appear: at the top-level, or locally. Just as a local `define` expands out to an instance of `letrec`, a local `define-syntax` expands out to `letrec-syntax`.

`letrec-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Syntax]

Bind each *keyword* to its corresponding *transformer* while expanding *exp1 exp2 ...*.

In the spirit of `letrec` versus `let`, an expansion produced by *transformer* may reference a *keyword* bound by the same *letrec-syntax*.

```
(letrec-syntax ((my-or
                 (syntax-rules ()
                   ((my-or)
                    #t)
                   ((my-or exp)
                    exp)
                   ((my-or exp rest ...)
                    (let ((t exp))
```

```
                          (if t
                              t
                              (my-or rest ...)))))))
  (my-or #f "rockaway beach"))
⇒ "rockaway beach"
```

6.10.2 Syntax-rules Macros

`syntax-rules` macros are simple, pattern-driven syntax transformers, with a beauty worthy of Scheme.

`syntax-rules` *literals* (*pattern template*) ... [Syntax]

Create a syntax transformer that will rewrite an expression using the rules embodied in the *pattern* and *template* clauses.

A `syntax-rules` macro consists of three parts: the literals (if any), the patterns, and as many templates as there are patterns.

When the syntax expander sees the invocation of a `syntax-rules` macro, it matches the expression against the patterns, in order, and rewrites the expression using the template from the first matching pattern. If no pattern matches, a syntax error is signalled.

6.10.2.1 Patterns

We have already seen some examples of patterns in the previous section: `(unless condition exp ...)`, `(my-or exp)`, and so on. A pattern is structured like the expression that it is to match. It can have nested structure as well, like `(let ((var val) ...) exp exp* ...)`. Broadly speaking, patterns are made of lists, improper lists, vectors, identifiers, and datums. Users can match a sequence of patterns using the ellipsis (`...`).

Identifiers in a pattern are called *literals* if they are present in the `syntax-rules` literals list, and *pattern variables* otherwise. When building up the macro output, the expander replaces instances of a pattern variable in the template with the matched subexpression.

```
(define-syntax kwote
  (syntax-rules ()
    ((kwote exp)
     (quote exp))))
(kwote (foo . bar))
⇒ (foo . bar)
```

An improper list of patterns matches as rest arguments do:

```
(define-syntax let1
  (syntax-rules ()
    ((_ (var val) . exps)
     (let ((var val)) . exps))))
```

However this definition of `let1` probably isn't what you want, as the tail pattern *exps* will match non-lists, like `(let1 (foo 'bar) . baz)`. So often instead of using improper lists as patterns, ellipsized patterns are better. Instances of a pattern variable in the template must be followed by an ellipsis.

```
(define-syntax let1
  (syntax-rules ()
```

```
    ((_ (var val) exp ...)
     (let ((var val)) exp ...))))
```

This `let1` probably still doesn't do what we want, because the body matches sequences of zero expressions, like `(let1 (foo 'bar))`. In this case we need to assert we have at least one body expression. A common idiom for this is to name the ellipsized pattern variable with an asterisk:

```
(define-syntax let1
  (syntax-rules ()
    ((_ (var val) exp exp* ...)
     (let ((var val)) exp exp* ...))))
```

A vector of patterns matches a vector whose contents match the patterns, including ellipsizing and tail patterns.

```
(define-syntax letv
  (syntax-rules ()
    ((_ #((var val) ...) exp exp* ...)
     (let ((var val) ...) exp exp* ...))))
(letv #((foo 'bar)) foo)
⇒ bar
```

Literals are used to match specific datums in an expression, like the use of `=>` and `else` in `cond` expressions.

```
(define-syntax cond1
  (syntax-rules (=> else)
    ((cond1 test => fun)
     (let ((exp test))
       (if exp (fun exp) #f)))
    ((cond1 test exp exp* ...)
     (if test (begin exp exp* ...)))
    ((cond1 else exp exp* ...)
     (begin exp exp* ...))))

(define (square x) (* x x))
(cond1 10 => square)
⇒ 100
(let ((=> #t))
  (cond1 10 => square))
⇒ #<procedure square (x)>
```

A literal matches an input expression if the input expression is an identifier with the same name as the literal, and both are unbound[6].

Although literals can be unbound, usually they are bound to allow them to be imported, exported, and renamed. See Section 6.20 [Modules], page 410, for more information on imports and exports. In Guile there are a few standard auxiliary syntax definitions, as specified by R6RS and R7RS:

[6] Language lawyers probably see the need here for use of `literal-identifier=?` rather than `free-identifier=?`, and would probably be correct. Patches accepted.

`else` [Scheme Syntax]
`=>` [Scheme Syntax]
`_` [Scheme Syntax]
`...` [Scheme Syntax]

Auxiliary syntax definitions.

These are defined as if with a macro that never matches, e.g.:

```
(define-syntax else (syntax-rules ()))
```

If a pattern is not a list, vector, or an identifier, it matches as a literal, with `equal?`.

```
(define-syntax define-matcher-macro
  (syntax-rules ()
    ((_ name lit)
     (define-syntax name
       (syntax-rules ()
        ((_ lit) #t)
        ((_ else) #f))))))

(define-matcher-macro is-literal-foo? "foo")

(is-literal-foo? "foo")
⇒ #t
(is-literal-foo? "bar")
⇒ #f
(let ((foo "foo"))
  (is-literal-foo? foo))
⇒ #f
```

The last example indicates that matching happens at expansion-time, not at run-time.

Syntax-rules macros are always used as (*macro* . *args*), and the *macro* will always be a symbol. Correspondingly, a `syntax-rules` pattern must be a list (proper or improper), and the first pattern in that list must be an identifier. Incidentally it can be any identifier – it doesn't have to actually be the name of the macro. Thus the following three are equivalent:

```
(define-syntax when
  (syntax-rules ()
    ((when c e ...)
     (if c (begin e ...)))))

(define-syntax when
  (syntax-rules ()
    ((_ c e ...)
     (if c (begin e ...)))))

(define-syntax when
  (syntax-rules ()
    ((something-else-entirely c e ...)
     (if c (begin e ...)))))
```

For clarity, use one of the first two variants. Also note that since the pattern variable will always match the macro itself (e.g., `cond1`), it is actually left unbound in the template.

6.10.2.2 Hygiene

`syntax-rules` macros have a magical property: they preserve referential transparency. When you read a macro definition, any free bindings in that macro are resolved relative to the macro definition; and when you read a macro instantiation, all free bindings in that expression are resolved relative to the expression.

This property is sometimes known as *hygiene*, and it does aid in code cleanliness. In your macro definitions, you can feel free to introduce temporary variables, without worrying about inadvertently introducing bindings into the macro expansion.

Consider the definition of `my-or` from the previous section:

```
(define-syntax my-or
  (syntax-rules ()
    ((my-or)
     #t)
    ((my-or exp)
     exp)
    ((my-or exp rest ...)
     (let ((t exp))
       (if t
           t
           (my-or rest ...))))))
```

A naive expansion of `(let ((t #t)) (my-or #f t))` would yield:

```
(let ((t #t))
  (let ((t #f))
    (if t t t)))
⇒ #f
```

Which clearly is not what we want. Somehow the `t` in the definition is distinct from the `t` at the site of use; and it is indeed this distinction that is maintained by the syntax expander, when expanding hygienic macros.

This discussion is mostly relevant in the context of traditional Lisp macros (see Section 6.10.5 [Defmacros], page 275), which do not preserve referential transparency. Hygiene adds to the expressive power of Scheme.

6.10.2.3 Shorthands

One often ends up writing simple one-clause `syntax-rules` macros. There is a convenient shorthand for this idiom, in the form of `define-syntax-rule`.

`define-syntax-rule` (*keyword . pattern*) [*docstring*] *template* [Syntax]
Define *keyword* as a new `syntax-rules` macro with one clause.

Cast into this form, our `when` example is significantly shorter:

```
(define-syntax-rule (when c e ...)
  (if c (begin e ...)))
```

6.10.2.4 Reporting Syntax Errors in Macros

`syntax-error` *message* [*arg ...*] [Syntax]

Report an error at macro-expansion time. *message* must be a string literal, and the optional *arg* operands can be arbitrary expressions providing additional information.

`syntax-error` is intended to be used within `syntax-rules` templates. For example:

```
(define-syntax simple-let
  (syntax-rules ()
    ((_ (head ... ((x . y) val) . tail)
        body1 body2 ...)
     (syntax-error
      "expected an identifier but got"
      (x . y)))
    ((_ ((name val) ...) body1 body2 ...)
     ((lambda (name ...) body1 body2 ...)
      val ...))))
```

6.10.2.5 Specifying a Custom Ellipsis Identifier

When writing macros that generate macro definitions, it is convenient to use a different ellipsis identifier at each level. Guile allows the desired ellipsis identifier to be specified as the first operand to `syntax-rules`, as specified by SRFI-46 and R7RS. For example:

```
(define-syntax define-quotation-macros
  (syntax-rules ()
    ((_ (macro-name head-symbol) ...)
     (begin (define-syntax macro-name
              (syntax-rules ::: ()
                ((_ x :::)
                 (quote (head-symbol x :::)))))
            ...))))
(define-quotation-macros (quote-a a) (quote-b b) (quote-c c))
(quote-a 1 2 3) ⇒ (a 1 2 3)
```

6.10.2.6 Further Information

For a formal definition of `syntax-rules` and its pattern language, see See Section "Macros" in *Revised(5) Report on the Algorithmic Language Scheme*.

`syntax-rules` macros are simple and clean, but do they have limitations. They do not lend themselves to expressive error messages: patterns either match or they don't. Their ability to generate code is limited to template-driven expansion; often one needs to define a number of helper macros to get real work done. Sometimes one wants to introduce a binding into the lexical context of the generated code; this is impossible with `syntax-rules`. Relatedly, they cannot programmatically generate identifiers.

The solution to all of these problems is to use `syntax-case` if you need its features. But if for some reason you're stuck with `syntax-rules`, you might enjoy Joe Marshall's `syntax-rules` Primer for the Merely Eccentric (`http://sites.google.com/site/evalapply/eccentric.txt`).

6.10.3 Support for the `syntax-case` System

`syntax-case` macros are procedural syntax transformers, with a power worthy of Scheme.

`syntax-case` *syntax literals* (*pattern* [*guard*] *exp*) ... [Syntax]
Match the syntax object *syntax* against the given patterns, in order. If a *pattern* matches, return the result of evaluating the associated *exp*.

Compare the following definitions of `when`:

```
(define-syntax when
  (syntax-rules ()
    ((_ test e e* ...)
     (if test (begin e e* ...)))))

(define-syntax when
  (lambda (x)
    (syntax-case x ()
      ((_ test e e* ...)
       #'(if test (begin e e* ...))))))
```

Clearly, the `syntax-case` definition is similar to its `syntax-rules` counterpart, and equally clearly there are some differences. The `syntax-case` definition is wrapped in a `lambda`, a function of one argument; that argument is passed to the `syntax-case` invocation; and the "return value" of the macro has a `#'` prefix.

All of these differences stem from the fact that `syntax-case` does not define a syntax transformer itself – instead, `syntax-case` expressions provide a way to destructure a *syntax object*, and to rebuild syntax objects as output.

So the `lambda` wrapper is simply a leaky implementation detail, that syntax transformers are just functions that transform syntax to syntax. This should not be surprising, given that we have already described macros as "programs that write programs". `syntax-case` is simply a way to take apart and put together program text, and to be a valid syntax transformer it needs to be wrapped in a procedure.

Unlike traditional Lisp macros (see Section 6.10.5 [Defmacros], page 275), `syntax-case` macros transform syntax objects, not raw Scheme forms. Recall the naive expansion of `my-or` given in the previous section:

```
(let ((t #t))
  (my-or #f t))
;; naive expansion:
(let ((t #t))
  (let ((t #f))
    (if t t t)))
```

Raw Scheme forms simply don't have enough information to distinguish the first two `t` instances in `(if t t t)` from the third `t`. So instead of representing identifiers as symbols, the syntax expander represents identifiers as annotated syntax objects, attaching such information to those syntax objects as is needed to maintain referential transparency.

`syntax` *form* [Syntax]
Create a syntax object wrapping *form* within the current lexical context.

Syntax objects are typically created internally to the process of expansion, but it is possible to create them outside of syntax expansion:

```
(syntax (foo bar baz))
⇒ #<some representation of that syntax>
```

However it is more common, and useful, to create syntax objects when building output from a `syntax-case` expression.

```
(define-syntax add1
  (lambda (x)
    (syntax-case x ()
      ((_ exp)
       (syntax (+ exp 1))))))
```

It is not strictly necessary for a `syntax-case` expression to return a syntax object, because `syntax-case` expressions can be used in helper functions, or otherwise used outside of syntax expansion itself. However a syntax transformer procedure must return a syntax object, so most uses of `syntax-case` do end up returning syntax objects.

Here in this case, the form that built the return value was `(syntax (+ exp 1))`. The interesting thing about this is that within a `syntax` expression, any appearance of a pattern variable is substituted into the resulting syntax object, carrying with it all relevant metadata from the source expression, such as lexical identity and source location.

Indeed, a pattern variable may only be referenced from inside a `syntax` form. The syntax expander would raise an error when defining `add1` if it found *exp* referenced outside a `syntax` form.

Since `syntax` appears frequently in macro-heavy code, it has a special reader macro: `#'`. `#'foo` is transformed by the reader into `(syntax foo)`, just as `'foo` is transformed into `(quote foo)`.

The pattern language used by `syntax-case` is conveniently the same language used by `syntax-rules`. Given this, Guile actually defines `syntax-rules` in terms of `syntax-case`:

```
(define-syntax syntax-rules
  (lambda (x)
    (syntax-case x ()
      ((_ (k ...) ((keyword . pattern) template) ...)
       #'(lambda (x)
           (syntax-case x (k ...)
             ((dummy . pattern) #'template)
             ...))))))
```

And that's that.

6.10.3.1 Why `syntax-case`?

The examples we have shown thus far could just as well have been expressed with `syntax-rules`, and have just shown that `syntax-case` is more verbose, which is true. But there is a difference: `syntax-case` creates *procedural* macros, giving the full power of Scheme to the macro expander. This has many practical applications.

A common desire is to be able to match a form only if it is an identifier. This is impossible with `syntax-rules`, given the datum matching forms. But with `syntax-case` it is easy:

`identifier?` *syntax-object* [Scheme Procedure]

Returns `#t` if *syntax-object* is an identifier, or `#f` otherwise.

```
;; relying on previous add1 definition
(define-syntax add1!
  (lambda (x)
    (syntax-case x ()
      ((_ var) (identifier? #'var)
       #'(set! var (add1 var))))))

(define foo 0)
(add1! foo)
foo ⇒ 1
(add1! "not-an-identifier") ⇒ error
```

With `syntax-rules`, the error for `(add1! "not-an-identifier")` would be something like "invalid `set!`". With `syntax-case`, it will say something like "invalid `add1!`", because we attach the *guard clause* to the pattern: `(identifier? #'var)`. This becomes more important with more complicated macros. It is necessary to use `identifier?`, because to the expander, an identifier is more than a bare symbol.

Note that even in the guard clause, we reference the *var* pattern variable within a `syntax` form, via `#'var`.

Another common desire is to introduce bindings into the lexical context of the output expression. One example would be in the so-called "anaphoric macros", like `aif`. Anaphoric macros bind some expression to a well-known identifier, often `it`, within their bodies. For example, in `(aif (foo) (bar it))`, `it` would be bound to the result of `(foo)`.

To begin with, we should mention a solution that doesn't work:

```
;; doesn't work
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       #'(let ((it test))
           (if it then else))))))
```

The reason that this doesn't work is that, by default, the expander will preserve referential transparency; the *then* and *else* expressions won't have access to the binding of `it`.

But they can, if we explicitly introduce a binding via `datum->syntax`.

`datum->syntax` *template-id datum* [Scheme Procedure]

Create a syntax object that wraps *datum*, within the lexical context corresponding to the identifier *template-id*.

For completeness, we should mention that it is possible to strip the metadata from a syntax object, returning a raw Scheme datum:

`syntax->datum` *syntax-object* [Scheme Procedure]

Strip the metadata from *syntax-object*, returning its contents as a raw Scheme datum.

In this case we want to introduce `it` in the context of the whole expression, so we can create a syntax object as `(datum->syntax x 'it)`, where `x` is the whole expression, as passed to the transformer procedure.

Here's another solution that doesn't work:

```
;; doesn't work either
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       (let ((it (datum->syntax x 'it)))
         #'(let ((it test))
             (if it then else)))))))
```

The reason that this one doesn't work is that there are really two environments at work here – the environment of pattern variables, as bound by `syntax-case`, and the environment of lexical variables, as bound by normal Scheme. The outer let form establishes a binding in the environment of lexical variables, but the inner let form is inside a syntax form, where only pattern variables will be substituted. Here we need to introduce a piece of the lexical environment into the pattern variable environment, and we can do so using `syntax-case` itself:

```
;; works, but is obtuse
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       ;; invoking syntax-case on the generated
       ;; syntax object to expose it to 'syntax'
       (syntax-case (datum->syntax x 'it) ()
         (it
           #'(let ((it test))
               (if it then else))))))))

(aif (getuid) (display it) (display "none")) (newline)
⊣ 500
```

However there are easier ways to write this. `with-syntax` is often convenient:

`with-syntax` ((*pat val*) ...) *exp* ... [Syntax]

Bind patterns *pat* from their corresponding values *val*, within the lexical context of *exp* ....

```
;; better
(define-syntax aif
  (lambda (x)
    (syntax-case x ()
      ((_ test then else)
       (with-syntax ((it (datum->syntax x 'it)))
         #'(let ((it test))
             (if it then else)))))))
```

As you might imagine, `with-syntax` is defined in terms of `syntax-case`. But even that might be off-putting to you if you are an old Lisp macro hacker, used to building macro output with `quasiquote`. The issue is that `with-syntax` creates a separation between the point of definition of a value and its point of substitution.

So for cases in which a `quasiquote` style makes more sense, `syntax-case` also defines `quasisyntax`, and the related `unsyntax` and `unsyntax-splicing`, abbreviated by the reader as `#‘`, `#,`, and `#,@`, respectively.

For example, to define a macro that inserts a compile-time timestamp into a source file, one may write:

```
(define-syntax display-compile-timestamp
  (lambda (x)
    (syntax-case x ()
      ((_)
       #‘(begin
          (display "The compile timestamp was: ")
          (display #,(current-time))
          (newline))))))
```

Readers interested in further information on `syntax-case` macros should see R. Kent Dybvig's excellent *The Scheme Programming Language*, either edition 3 or 4, in the chapter on syntax. Dybvig was the primary author of the `syntax-case` system. The book itself is available online at `http://scheme.com/tspl4/`.

6.10.3.2 Custom Ellipsis Identifiers for syntax-case Macros

When writing procedural macros that generate macro definitions, it is convenient to use a different ellipsis identifier at each level. Guile supports this for procedural macros using the `with-ellipsis` special form:

`with-ellipsis` *ellipsis body ...* [Syntax]

ellipsis must be an identifier. Evaluate *body* in a special lexical environment such that all macro patterns and templates within *body* will use *ellipsis* as the ellipsis identifier instead of the usual three dots (`...`).

For example:

```
(define-syntax define-quotation-macros
  (lambda (x)
    (syntax-case x ()
      ((_ (macro-name head-symbol) ...)
       #’(begin (define-syntax macro-name
                  (lambda (x)
                    (with-ellipsis :::
                      (syntax-case x ()
                        ((_ x :::)
                         #’(quote (head-symbol x :::)))))))))
                ...)))))
(define-quotation-macros (quote-a a) (quote-b b) (quote-c c))
(quote-a 1 2 3) ⇒ (a 1 2 3)
```

Note that `with-ellipsis` does not affect the ellipsis identifier of the generated code, unless `with-ellipsis` is included around the generated code.

6.10.4 Syntax Transformer Helpers

As noted in the previous section, Guile's syntax expander operates on syntax objects. Procedural macros consume and produce syntax objects. This section describes some of the auxiliary helpers that procedural macros can use to compare, generate, and query objects of this data type.

`bound-identifier=?` *a b* [Scheme Procedure]
Return `#t` if the syntax objects *a* and *b* refer to the same lexically-bound identifier, or `#f` otherwise.

`free-identifier=?` *a b* [Scheme Procedure]
Return `#t` if the syntax objects *a* and *b* refer to the same free identifier, or `#f` otherwise.

`generate-temporaries` *ls* [Scheme Procedure]
Return a list of temporary identifiers as long as *ls* is long.

`syntax-source` *x* [Scheme Procedure]
Return the source properties that correspond to the syntax object *x*. See Section 6.26.2 [Source Properties], page 477, for more information.

Guile also offers some more experimental interfaces in a separate module. As was the case with the Large Hadron Collider, it is unclear to our senior macrologists whether adding these interfaces will result in awesomeness or in the destruction of Guile via the creation of a singularity. We will preserve their functionality through the 2.0 series, but we reserve the right to modify them in a future stable series, to a more than usual degree.

```
(use-modules (system syntax))
```

`syntax-module` *id* [Scheme Procedure]
Return the name of the module whose source contains the identifier *id*.

`syntax-local-binding` *id* [Scheme Procedure]
[*#:resolve-syntax-parameters?=#t*]
Resolve the identifer *id*, a syntax object, within the current lexical environment, and return two values, the binding type and a binding value. The binding type is a symbol, which may be one of the following:

`lexical` A lexically-bound variable. The value is a unique token (in the sense of `eq?`) identifying this binding.

`macro` A syntax transformer, either local or global. The value is the transformer procedure.

`syntax-parameter`
A syntax parameter (see Section 6.10.7 [Syntax Parameters], page 277). By default, `syntax-local-binding` will resolve syntax parameters, so that this value will not be returned. Pass `#:resolve-syntax-parameters? #f` to indicate that you are interested in syntax parameters. The value is the default transformer procedure, as in `macro`.

`pattern-variable`
: A pattern variable, bound via `syntax-case`. The value is an opaque object, internal to the expander.

`ellipsis`
: An internal binding, bound via `with-ellipsis`. The value is the (anti-marked) local ellipsis identifier.

`displaced-lexical`
: A lexical variable that has gone out of scope. This can happen if a badly-written procedural macro saves a syntax object, then attempts to introduce it in a context in which it is unbound. The value is `#f`.

`global`
: A global binding. The value is a pair, whose head is the symbol, and whose tail is the name of the module in which to resolve the symbol.

`other`
: Some other binding, like `lambda` or other core bindings. The value is `#f`.

This is a very low-level procedure, with limited uses. One case in which it is useful is to build abstractions that associate auxiliary information with macros:

```
(define aux-property (make-object-property))
(define-syntax-rule (with-aux aux value)
  (let ((trans value))
    (set! (aux-property trans) aux)
    trans))
(define-syntax retrieve-aux
  (lambda (x)
    (syntax-case x ()
      ((x id)
       (call-with-values (lambda () (syntax-local-binding #'id))
         (lambda (type val)
           (with-syntax ((aux (datum->syntax #'here
                                             (and (eq? type 'macro)
                                                  (aux-property val)))))
             #''aux)))))))
(define-syntax foo
  (with-aux 'bar
    (syntax-rules () ((_) 'foo))))
(foo)
⇒ foo
(retrieve-aux foo)
⇒ bar
```

`syntax-local-binding` must be called within the dynamic extent of a syntax transformer; to call it otherwise will signal an error.

`syntax-locally-bound-identifiers` *id* [Scheme Procedure]

Return a list of identifiers that were visible lexically when the identifier *id* was created, in order from outermost to innermost.

This procedure is intended to be used in specialized procedural macros, to provide a macro with the set of bound identifiers that the macro can reference.

As a technical implementation detail, the identifiers returned by `syntax-locally-bound-identifiers` will be anti-marked, like the syntax object that is given as input to a macro. This is to signal to the macro expander that these bindings were present in the original source, and do not need to be hygienically renamed, as would be the case with other introduced identifiers. See the discussion of hygiene in section 12.1 of the R6RS, for more information on marks.

```
(define (local-lexicals id)
  (filter (lambda (x)
            (eq? (syntax-local-binding x) 'lexical))
          (syntax-locally-bound-identifiers id)))
(define-syntax lexicals
  (lambda (x)
    (syntax-case x ()
      ((lexicals) #'(lexicals lexicals))
      ((lexicals scope)
       (with-syntax (((id ...) (local-lexicals #'scope)))
         #'(list (cons 'id id) ...))))))

(let* ((x 10) (x 20)) (lexicals))
⇒ ((x . 10) (x . 20))
```

6.10.5 Lisp-style Macro Definitions

The traditional way to define macros in Lisp is very similar to procedure definitions. The key differences are that the macro definition body should return a list that describes the transformed expression, and that the definition is marked as a macro definition (rather than a procedure definition) by the use of a different definition keyword: in Lisp, `defmacro` rather than `defun`, and in Scheme, `define-macro` rather than `define`.

Guile supports this style of macro definition using both `defmacro` and `define-macro`. The only difference between them is how the macro name and arguments are grouped together in the definition:

```
(defmacro name (args ...) body ...)
```

is the same as

```
(define-macro (name args ...) body ...)
```

The difference is analogous to the corresponding difference between Lisp's `defun` and Scheme's `define`.

Having read the previous section on `syntax-case`, it's probably clear that Guile actually implements defmacros in terms of `syntax-case`, applying the transformer on the expression between invocations of `syntax->datum` and `datum->syntax`. This realization leads us to the problem with defmacros, that they do not preserve referential transparency. One can be careful to not introduce bindings into expanded code, via liberal use of `gensym`, but there is no getting around the lack of referential transparency for free bindings in the macro itself.

Even a macro as simple as our `when` from before is difficult to get right:

```
(define-macro (when cond exp . rest)
  `(if ,cond
       (begin ,exp . ,rest)))
```

```
(when #f (display "Launching missiles!\n"))
⇒ #f

(let ((if list))
  (when #f (display "Launching missiles!\n")))
⊣ Launching missiles!
⇒ (#f #<unspecified>)
```

Guile's perspective is that defmacros have had a good run, but that modern macros should be written with `syntax-rules` or `syntax-case`. There are still many uses of defmacros within Guile itself, but we will be phasing them out over time. Of course we won't take away `defmacro` or `define-macro` themselves, as there is lots of code out there that uses them.

6.10.6 Identifier Macros

When the syntax expander sees a form in which the first element is a macro, the whole form gets passed to the macro's syntax transformer. One may visualize this as:

```
(define-syntax foo foo-transformer)
(foo arg...)
;; expands via
(foo-transformer #'(foo arg...))
```

If, on the other hand, a macro is referenced in some other part of a form, the syntax transformer is invoked with only the macro reference, not the whole form.

```
(define-syntax foo foo-transformer)
foo
;; expands via
(foo-transformer #'foo)
```

This allows bare identifier references to be replaced programmatically via a macro. `syntax-rules` provides some syntax to effect this transformation more easily.

`identifier-syntax` *exp* [Syntax]

Returns a macro transformer that will replace occurrences of the macro with *exp*.

For example, if you are importing external code written in terms of `fx+`, the fixnum addition operator, but Guile doesn't have `fx+`, you may use the following to replace `fx+` with `+`:

```
(define-syntax fx+ (identifier-syntax +))
```

There is also special support for recognizing identifiers on the left-hand side of a `set!` expression, as in the following:

```
(define-syntax foo foo-transformer)
(set! foo val)
;; expands via
(foo-transformer #'(set! foo val))
;; if foo-transformer is a "variable transformer"
```

As the example notes, the transformer procedure must be explicitly marked as being a "variable transformer", as most macros aren't written to discriminate on the form in the operator position.

`make-variable-transformer` *transformer* [Scheme Procedure]

Mark the *transformer* procedure as being a "variable transformer". In practice this means that, when bound to a syntactic keyword, it may detect references to that keyword on the left-hand-side of a `set!`.

```
(define bar 10)
(define-syntax bar-alias
  (make-variable-transformer
   (lambda (x)
     (syntax-case x (set!)
       ((set! var val) #'(set! bar val))
       ((var arg ...) #'(bar arg ...))
       (var (identifier? #'var) #'bar)))))

bar-alias ⇒ 10
(set! bar-alias 20)
bar ⇒ 20
(set! bar 30)
bar-alias ⇒ 30
```

There is an extension to identifier-syntax which allows it to handle the `set!` case as well:

`identifier-syntax` (*var exp1*) ((*set! var val*) *exp2*) [Syntax]

Create a variable transformer. The first clause is used for references to the variable in operator or operand position, and the second for appearances of the variable on the left-hand-side of an assignment.

For example, the previous `bar-alias` example could be expressed more succinctly like this:

```
(define-syntax bar-alias
  (identifier-syntax
    (var bar)
    ((set! var val) (set! bar val))))
```

As before, the templates in `identifier-syntax` forms do not need wrapping in `#'` syntax forms.

6.10.7 Syntax Parameters

Syntax parameters[7] are a mechanism for rebinding a macro definition within the dynamic extent of a macro expansion. This provides a convenient solution to one of the most common types of unhygienic macro: those that introduce a unhygienic binding each time the macro is used. Examples include a `lambda` form with a `return` keyword, or class macros that introduce a special `self` binding.

[7] Described in the paper *Keeping it Clean with Syntax Parameters* by Barzilay, Culpepper and Flatt.

With syntax parameters, instead of introducing the binding unhygienically each time, we instead create one binding for the keyword, which we can then adjust later when we want the keyword to have a different meaning. As no new bindings are introduced, hygiene is preserved. This is similar to the dynamic binding mechanisms we have at run-time (see Section 7.5.27 [SRFI-39], page 631), except that the dynamic binding only occurs during macro expansion. The code after macro expansion remains lexically scoped.

`define-syntax-parameter` *keyword transformer* [Syntax]

Binds *keyword* to the value obtained by evaluating *transformer*. The *transformer* provides the default expansion for the syntax parameter, and in the absence of `syntax-parameterize`, is functionally equivalent to `define-syntax`. Usually, you will just want to have the *transformer* throw a syntax error indicating that the *keyword* is supposed to be used in conjunction with another macro, for example:

```
(define-syntax-parameter return
  (lambda (stx)
    (syntax-violation 'return "return used outside of a lambda^" stx)))
```

`syntax-parameterize` ((*keyword transformer*) ...) *exp* ... [Syntax]

Adjusts *keyword* ... to use the values obtained by evaluating their *transformer* ..., in the expansion of the *exp* ... forms. Each *keyword* must be bound to a syntax-parameter. `syntax-parameterize` differs from `let-syntax`, in that the binding is not shadowed, but adjusted, and so uses of the keyword in the expansion of *exp* ... use the new transformers. This is somewhat similar to how `parameterize` adjusts the values of regular parameters, rather than creating new bindings.

```
(define-syntax lambda^
  (syntax-rules ()
    [(lambda^ argument-list body body* ...)
     (lambda argument-list
       (call-with-current-continuation
        (lambda (escape)
          ;; In the body we adjust the 'return' keyword so that calls
          ;; to 'return' are replaced with calls to the escape
          ;; continuation.
          (syntax-parameterize ([return (syntax-rules ()
                                          [(return vals (... ...))
                                           (escape vals (... ...))])])
            body body* ...))))]))

;; Now we can write functions that return early.  Here, 'product' will
;; return immediately if it sees any 0 element.
(define product
  (lambda^ (list)
           (fold (lambda (n o)
                   (if (zero? n)
                       (return 0)
                       (* n o)))
                 1
```

```
                    list)))
```

6.10.8 Eval-when

As `syntax-case` macros have the whole power of Scheme available to them, they present a problem regarding time: when a macro runs, what parts of the program are available for the macro to use?

The default answer to this question is that when you import a module (via `define-module` or `use-modules`), that module will be loaded up at expansion-time, as well as at run-time. Additionally, top-level syntactic definitions within one compilation unit made by `define-syntax` are also evaluated at expansion time, in the order that they appear in the compilation unit (file).

But if a syntactic definition needs to call out to a normal procedure at expansion-time, it might well need need special declarations to indicate that the procedure should be made available at expansion-time.

For example, the following code will work at a REPL, but not in a file:

```
;; incorrect
(use-modules (srfi srfi-19))
(define (date) (date->string (current-date)))
(define-syntax %date (identifier-syntax (date)))
(define *compilation-date* %date)
```

It works at a REPL because the expressions are evaluated one-by-one, in order, but if placed in a file, the expressions are expanded one-by-one, but not evaluated until the compiled file is loaded.

The fix is to use `eval-when`.

```
;; correct: using eval-when
(use-modules (srfi srfi-19))
(eval-when (expand load eval)
  (define (date) (date->string (current-date))))
(define-syntax %date (identifier-syntax (date)))
(define *compilation-date* %date)
```

`eval-when` *conditions exp...* [Syntax]

Evaluate *exp...* under the given *conditions*. Valid conditions include:

`expand` Evaluate during macro expansion, whether compiling or not.

`load` Evaluate during the evaluation phase of compiled code, e.g. when loading a compiled module or running compiled code at the REPL.

`eval` Evaluate during the evaluation phase of non-compiled code.

`compile` Evaluate during macro expansion, but only when compiling.

In other words, when using the primitive evaluator, `eval-when` expressions with `expand` are run during macro expansion, and those with `eval` are run during the evaluation phase.

When using the compiler, `eval-when` expressions with either `expand` or `compile` are run during macro expansion, and those with `load` are run during the evaluation phase.

When in doubt, use the three conditions `(expand load eval)`, as in the example above. Other uses of `eval-when` may void your warranty or poison your cat.

6.10.9 Macro Expansion

Usually, macros are expanded on behalf of the user as needed. Macro expansion is an integral part of `eval` and `compile`. Users can also expand macros at the REPL prompt via the `expand` REPL command; See Section 4.4.4.4 [Compile Commands], page 51.

Macros can also be expanded programmatically, via `macroexpand`, but the details get a bit hairy for two reasons.

The first complication is that the result of macro-expansion isn't Scheme: it's Tree-IL, Guile's high-level intermediate language. See Section 9.4.3 [Tree-IL], page 859. As "hygienic macros" can produce identifiers that are distinct but have the same name, the output format needs to be able to represent distinctions between variable identities and names. Again, See Section 9.4.3 [Tree-IL], page 859, for all the details. The easiest thing is to just run `tree-il->scheme` on the result of macro-expansion:

```
(macroexpand '(+ 1 2))
⇒
#<tree-il (call (toplevel +) (const 1) (const 2))>

(use-modules (language tree-il))
(tree-il->scheme (macroexpand '(+ 1 2)))
⇒
(+ 1 2)
```

The second complication involves `eval-when`. As an example, what would it mean to macro-expand the definition of a macro?

```
(macroexpand '(define-syntax qux (identifier-syntax 'bar)))
⇒
?
```

The answer is that it depends who is macro-expanding, and why. Do you define the macro in the current environment? Residualize a macro definition? Both? Neither? The default is to expand in "eval" mode, which means an `eval-when` clauses will only proceed when `eval` (or `expand`) is in its condition set. Top-level macros will be `eval`'d in the top-level environment.

In this way `(macroexpand` *foo*`)` is equivalent to `(macroexpand` *foo* `'e '(eval))`. The second argument is the mode (`'e` for "eval") and the third is the eval-syntax-expanders-when parameter (only `eval` in this default setting).

But if you are compiling the macro definition, probably you want to reify the macro definition itself. In that case you pass `'c` as the second argument to `macroexpand`. But probably you want the macro definition to be present at compile time as well, so you pass `'(compile load eval)` as the *esew* parameter. In fact `(compile` *foo* `#:to 'tree-il)` is entirely equivalent to `(macroexpand` *foo* `'c '(compile load eval))`; See Section 9.4.2 [The Scheme Compiler], page 858.

It's a terrible interface; we know. The macroexpander is somewhat tricksy regarding modes, so unless you are building a macro-expanding tool, we suggest to avoid invoking it directly.

6.10.10 Hygiene and the Top-Level

Consider the following macro.

```
(define-syntax-rule (defconst name val)
  (begin
    (define t val)
    (define-syntax-rule (name) t)))
```

If we use it to make a couple of bindings:

```
(defconst foo 42)
(defconst bar 37)
```

The expansion would look something like this:

```
(begin
  (define t 42)
  (define-syntax-rule (foo) t))
(begin
  (define t 37)
  (define-syntax-rule (bar) t))
```

As the two `t` bindings were introduced by the macro, they should be introduced hygienically – and indeed they are, inside a lexical contour (a `let` or some other lexical scope). The `t` reference in `foo` is distinct to the reference in `bar`.

At the top-level things are more complicated. Before Guile 2.2, a use of `defconst` at the top-level would not introduce a fresh binding for `t`. This was consistent with a weaselly interpretation of the Scheme standard, in which all possible bindings may be assumed to exist, at the top-level, and in which we merely take advantage of toplevel `define` of an existing binding being equivalent to `set!`. But it's not a good reason.

The solution is to create fresh names for all bindings introduced by macros – not just bindings in lexical contours, but also bindings introduced at the top-level.

However, the obvious strategy of just giving random names to introduced toplevel identifiers poses a problem for separate compilation. Consider without loss of generality a `defconst` of `foo` in module `a` that introduces the fresh top-level name `t-1`. If we then compile a module `b` that uses `foo`, there is now a reference to `t-1` in module `b`. If module `a` is then expanded again, for whatever reason, for example in a simple recompilation, the introduced `t` gets a fresh name; say, `t-2`. Now module `b` has broken because module `a` no longer has a binding for `t-1`.

If introduced top-level identifiers "escape" a module, in whatever way, they then form part of the binary interface (ABI) of a module. It is unacceptable from an engineering point of view to allow the ABI to change randomly. (It also poses practical problems in meeting the recompilation conditions of the Lesser GPL license, for such modules.) For this reason many people prefer to never use identifier-introducing macros at the top-level, instead making those macros receive the names for their introduced identifiers as part of their arguments, or to construct them programmatically and use `datum->syntax`. But this approach requires omniscience as to the implementation of all macros one might use, and also limits the expressive power of Scheme macros.

There is no perfect solution to this issue. Guile does a terrible thing here. When it goes to introduce a top-level identifier, Guile gives the identifier a pseudo-fresh name: a name

that depends on the hash of the source expression in which the name occurs. The result in this case is that the introduced definitions expand as:

```
(begin
  (define t-1dc5e42de7c1050c 42)
  (define-syntax-rule (foo) t-1dc5e42de7c1050c))
(begin
  (define t-10cb8ce9fdddd6e9 37)
  (define-syntax-rule (bar) t-10cb8ce9fdddd6e9))
```

However, note that as the hash depends solely on the expression introducing the definition, we also have:

```
(defconst baz 42)
⇒ (begin
    (define t-1dc5e42de7c1050c 42)
    (define-syntax-rule (baz) t-1dc5e42de7c1050c))
```

Note that the introduced binding has the same name! This is because the source expression, `(define t 42)`, was the same. Probably you will never see an error in this area, but it is important to understand the components of the interface of a module, and that interface may include macro-introduced identifiers.

6.10.11 Internal Macros

`make-syntax-transformer` *name type binding* [Scheme Procedure]

Construct a syntax transformer object. This is part of Guile's low-level support for syntax-case.

`macro?` *obj* [Scheme Procedure]
`scm_macro_p` (*obj*) [C Function]

Return `#t` if *obj* is a syntax transformer, or `#f` otherwise.

Note that it's a bit difficult to actually get a macro as a first-class object; simply naming it (like `case`) will produce a syntax error. But it is possible to get these objects using `module-ref`:

```
(macro? (module-ref (current-module) 'case))
⇒ #t
```

`macro-type` *m* [Scheme Procedure]
`scm_macro_type` (*m*) [C Function]

Return the *type* that was given when *m* was constructed, via `make-syntax-transformer`.

`macro-name` *m* [Scheme Procedure]
`scm_macro_name` (*m*) [C Function]

Return the name of the macro *m*.

`macro-binding` *m* [Scheme Procedure]
`scm_macro_binding` (*m*) [C Function]

Return the binding of the macro *m*.

`macro-transformer` *m* [Scheme Procedure]
`scm_macro_transformer` (*m*) [C Function]

Return the transformer of the macro *m*. This will return a procedure, for which one may ask the docstring. That's the whole reason this section is documented. Actually a part of the result of `macro-binding`.

6.11 General Utility Functions

This chapter contains information about procedures which are not cleanly tied to a specific data type. Because of their wide range of applications, they are collected in a *utility* chapter.

6.11.1 Equality

There are three kinds of core equality predicates in Scheme, described below. The same kinds of comparisons arise in other functions, like `memq` and friends (see Section 6.6.9.7 [List Searching], page 185).

For all three tests, objects of different types are never equal. So for instance a list and a vector are not `equal?`, even if their contents are the same. Exact and inexact numbers are considered different types too, and are hence not equal even if their values are the same.

`eq?` tests just for the same object (essentially a pointer comparison). This is fast, and can be used when searching for a particular object, or when working with symbols or keywords (which are always unique objects).

`eqv?` extends `eq?` to look at the value of numbers and characters. It can for instance be used somewhat like `=` (see Section 6.6.2.8 [Comparison], page 117) but without an error if one operand isn't a number.

`equal?` goes further, it looks (recursively) into the contents of lists, vectors, etc. This is good for instance on lists that have been read or calculated in various places and are the same, just not made up of the same pairs. Such lists look the same (when printed), and `equal?` will consider them the same.

`eq?` *x y* [Scheme Procedure]
`scm_eq_p` (*x*, *y*) [C Function]

Return `#t` if *x* and *y* are the same object, except for numbers and characters. For example,

```
(define x (vector 1 2 3))
(define y (vector 1 2 3))

(eq? x x)  ⇒ #t
(eq? x y)  ⇒ #f
```

Numbers and characters are not equal to any other object, but the problem is they're not necessarily `eq?` to themselves either. This is even so when the number comes directly from a variable,

```
(let ((n (+ 2 3)))
  (eq? n n))       ⇒ *unspecified*
```

Generally `eqv?` below should be used when comparing numbers or characters. `=` (see Section 6.6.2.8 [Comparison], page 117) or `char=?` (see Section 6.6.3 [Characters], page 129) can be used too.

It's worth noting that end-of-list `()`, `#t`, `#f`, a symbol of a given name, and a keyword of a given name, are unique objects. There's just one of each, so for instance no matter how `()` arises in a program, it's the same object and can be compared with `eq?`,

```
(define x (cdr '(123)))
(define y (cdr '(456)))
(eq? x y) ⇒ #t

(define x (string->symbol "foo"))
(eq? x 'foo) ⇒ #t
```

`int scm_is_eq` (*SCM x, SCM y*) [C Function]

Return `1` when *x* and *y* are equal in the sense of `eq?`, otherwise return `0`.

The `==` operator should not be used on `SCM` values, an `SCM` is a C type which cannot necessarily be compared using `==` (see Section 6.3 [The SCM Type], page 100).

`eqv?` *x y* [Scheme Procedure]
`scm_eqv_p` (*x, y*) [C Function]

Return `#t` if *x* and *y* are the same object, or for characters and numbers the same value.

On objects except characters and numbers, `eqv?` is the same as `eq?` above, it's true if *x* and *y* are the same object.

If *x* and *y* are numbers or characters, `eqv?` compares their type and value. An exact number is not `eqv?` to an inexact number (even if their value is the same).

```
(eqv? 3 (+ 1 2)) ⇒ #t
(eqv? 1 1.0)     ⇒ #f
```

`equal?` *x y* [Scheme Procedure]
`scm_equal_p` (*x, y*) [C Function]

Return `#t` if *x* and *y* are the same type, and their contents or value are equal.

For a pair, string, vector, array or structure, `equal?` compares the contents, and does so using the same `equal?` recursively, so a deep structure can be traversed.

```
(equal? (list 1 2 3) (list 1 2 3))   ⇒ #t
(equal? (list 1 2 3) (vector 1 2 3)) ⇒ #f
```

For other objects, `equal?` compares as per `eqv?` above, which means characters and numbers are compared by type and value (and like `eqv?`, exact and inexact numbers are not `equal?`, even if their value is the same).

```
(equal? 3 (+ 1 2)) ⇒ #t
(equal? 1 1.0)     ⇒ #f
```

Hash tables are currently only compared as per `eq?`, so two different tables are not `equal?`, even if their contents are the same.

`equal?` does not support circular data structures, it may go into an infinite loop if asked to compare two circular lists or similar.

GOOPS object types (see Chapter 8 [GOOPS], page 773), including foreign object types (see Section 5.5 [Defining New Foreign Object Types], page 74), can have an `equal?` implementation specialized on two values of the same type. If `equal?` is called on two GOOPS objects of the same type, `equal?` will dispatch out to a generic function. This lets an application traverse the contents or control what is considered `equal?` for two objects of such a type. If there's no such handler, the default is to just compare as per `eq?`.

6.11.2 Object Properties

It's often useful to associate a piece of additional information with a Scheme object even though that object does not have a dedicated slot available in which the additional information could be stored. Object properties allow you to do just that.

Guile's representation of an object property is a procedure-with-setter (see Section 6.9.8 [Procedures with Setters], page 259) that can be used with the generalized form of `set!` (REFFIXME) to set and retrieve that property for any Scheme object. So, setting a property looks like this:

```
(set! (my-property obj1) value-for-obj1)
(set! (my-property obj2) value-for-obj2)
```

And retrieving values of the same property looks like this:

```
(my-property obj1)
⇒
value-for-obj1

(my-property obj2)
⇒
value-for-obj2
```

To create an object property in the first place, use the `make-object-property` procedure:

```
(define my-property (make-object-property))
```

`make-object-property` [Scheme Procedure]

Create and return an object property. An object property is a procedure-with-setter that can be called in two ways. `(set! (`*property obj*`) val)` sets *obj*'s *property* to *val*. `(`*property obj*`)` returns the current setting of *obj*'s *property*.

A single object property created by `make-object-property` can associate distinct property values with all Scheme values that are distinguishable by `eq?` (ruling out numeric values).

Internally, object properties are implemented using a weak key hash table. This means that, as long as a Scheme value with property values is protected from garbage collection, its property values are also protected. When the Scheme value is collected, its entry in the property table is removed and so the (ex-) property values are no longer protected by the table.

Guile also implements a more traditional Lispy interface to properties, in which each object has an list of key-value pairs associated with it. Properties in that list are keyed by

symbols. This is a legacy interface; you should use weak hash tables or object properties instead.

`object-properties` *obj* [Scheme Procedure]
`scm_object_properties` (*obj*) [C Function]
Return *obj*'s property list.

`set-object-properties!` *obj alist* [Scheme Procedure]
`scm_set_object_properties_x` (*obj, alist*) [C Function]
Set *obj*'s property list to *alist*.

`object-property` *obj key* [Scheme Procedure]
`scm_object_property` (*obj, key*) [C Function]
Return the property of *obj* with name *key*.

`set-object-property!` *obj key value* [Scheme Procedure]
`scm_set_object_property_x` (*obj, key, value*) [C Function]
In *obj*'s property list, set the property named *key* to *value*.

6.11.3 Sorting

Sorting is very important in computer programs. Therefore, Guile comes with several sorting procedures built-in. As always, procedures with names ending in `!` are side-effecting, that means that they may modify their parameters in order to produce their results.

The first group of procedures can be used to merge two lists (which must be already sorted on their own) and produce sorted lists containing all elements of the input lists.

`merge` *alist blist less* [Scheme Procedure]
`scm_merge` (*alist, blist, less*) [C Function]
Merge two already sorted lists into one. Given two lists *alist* and *blist*, such that `(sorted? alist less?)` and `(sorted? blist less?)`, return a new list in which the elements of *alist* and *blist* have been stably interleaved so that `(sorted? (merge alist blist less?) less?)`. Note: this does _not_ accept vectors.

`merge!` *alist blist less* [Scheme Procedure]
`scm_merge_x` (*alist, blist, less*) [C Function]
Takes two lists *alist* and *blist* such that `(sorted? alist less?)` and `(sorted? blist less?)` and returns a new list in which the elements of *alist* and *blist* have been stably interleaved so that `(sorted? (merge alist blist less?) less?)`. This is the destructive variant of `merge` Note: this does _not_ accept vectors.

The following procedures can operate on sequences which are either vectors or list. According to the given arguments, they return sorted vectors or lists, respectively. The first of the following procedures determines whether a sequence is already sorted, the other sort a given sequence. The variants with names starting with `stable-` are special in that they maintain a special property of the input sequences: If two or more elements are the same according to the comparison predicate, they are left in the same order as they appeared in the input.

`sorted?` *items less* [Scheme Procedure]
`scm_sorted_p` (*items, less*) [C Function]
Return `#t` if *items* is a list or vector such that, for each element x and the next element y of *items*, `(less y x)` returns `#f`. Otherwise return `#f`.

`sort` *items less* [Scheme Procedure]
`scm_sort` (*items, less*) [C Function]
Sort the sequence *items*, which may be a list or a vector. *less* is used for comparing the sequence elements. This is not a stable sort.

`sort!` *items less* [Scheme Procedure]
`scm_sort_x` (*items, less*) [C Function]
Sort the sequence *items*, which may be a list or a vector. *less* is used for comparing the sequence elements. The sorting is destructive, that means that the input sequence is modified to produce the sorted result. This is not a stable sort.

`stable-sort` *items less* [Scheme Procedure]
`scm_stable_sort` (*items, less*) [C Function]
Sort the sequence *items*, which may be a list or a vector. *less* is used for comparing the sequence elements. This is a stable sort.

`stable-sort!` *items less* [Scheme Procedure]
`scm_stable_sort_x` (*items, less*) [C Function]
Sort the sequence *items*, which may be a list or a vector. *less* is used for comparing the sequence elements. The sorting is destructive, that means that the input sequence is modified to produce the sorted result. This is a stable sort.

The procedures in the last group only accept lists or vectors as input, as their names indicate.

`sort-list` *items less* [Scheme Procedure]
`scm_sort_list` (*items, less*) [C Function]
Sort the list *items*, using *less* for comparing the list elements. This is a stable sort.

`sort-list!` *items less* [Scheme Procedure]
`scm_sort_list_x` (*items, less*) [C Function]
Sort the list *items*, using *less* for comparing the list elements. The sorting is destructive, that means that the input list is modified to produce the sorted result. This is a stable sort.

`restricted-vector-sort!` *vec less startpos endpos* [Scheme Procedure]
`scm_restricted_vector_sort_x` (*vec, less, startpos, endpos*) [C Function]
Sort the vector *vec*, using *less* for comparing the vector elements. *startpos* (inclusively) and *endpos* (exclusively) delimit the range of the vector which gets sorted. The return value is not specified.

6.11.4 Copying Deep Structures

The procedures for copying lists (see Section 6.6.9 [Lists], page 181) only produce a flat copy of the input list, and currently Guile does not even contain procedures for copying vectors. `copy-tree` can be used for these application, as it does not only copy the spine of a list, but also copies any pairs in the cars of the input lists.

`copy-tree` *obj* [Scheme Procedure]
`scm_copy_tree` (*obj*) [C Function]
Recursively copy the data tree that is bound to *obj*, and return the new data structure. `copy-tree` recurses down the contents of both pairs and vectors (since both cons cells and vector cells may point to arbitrary objects), and stops recursing when it hits any other object.

6.11.5 General String Conversion

When debugging Scheme programs, but also for providing a human-friendly interface, a procedure for converting any Scheme object into string format is very useful. Conversion from/to strings can of course be done with specialized procedures when the data type of the object to convert is known, but with this procedure, it is often more comfortable.

`object->string` converts an object by using a print procedure for writing to a string port, and then returning the resulting string. Converting an object back from the string is only possible if the object type has a read syntax and the read syntax is preserved by the printing procedure.

`object->string` *obj* [*printer*] [Scheme Procedure]
`scm_object_to_string` (*obj, printer*) [C Function]
Return a Scheme string obtained by printing *obj*. Printing function can be specified by the optional second argument *printer* (default: `write`).

6.11.6 Hooks

A hook is a list of procedures to be called at well defined points in time. Typically, an application provides a hook *h* and promises its users that it will call all of the procedures in *h* at a defined point in the application's processing. By adding its own procedure to *h*, an application user can tap into or even influence the progress of the application.

Guile itself provides several such hooks for debugging and customization purposes: these are listed in a subsection below.

When an application first creates a hook, it needs to know how many arguments will be passed to the hook's procedures when the hook is run. The chosen number of arguments (which may be none) is declared when the hook is created, and all the procedures that are added to that hook must be capable of accepting that number of arguments.

A hook is created using `make-hook`. A procedure can be added to or removed from a hook using `add-hook!` or `remove-hook!`, and all of a hook's procedures can be removed together using `reset-hook!`. When an application wants to run a hook, it does so using `run-hook`.

6.11.6.1 Hook Usage by Example

Hook usage is shown by some examples in this section. First, we will define a hook of arity 2 — that is, the procedures stored in the hook will have to accept two arguments.

```
(define hook (make-hook 2))
hook
⇒ #<hook 2 40286c90>
```

Now we are ready to add some procedures to the newly created hook with `add-hook!`. In the following example, two procedures are added, which print different messages and do different things with their arguments.

```
(add-hook! hook (lambda (x y)
                  (display "Foo: ")
                  (display (+ x y))
                  (newline)))
(add-hook! hook (lambda (x y)
                  (display "Bar: ")
                  (display (* x y))
                  (newline)))
```

Once the procedures have been added, we can invoke the hook using `run-hook`.

```
(run-hook hook 3 4)
⊣ Bar: 12
⊣ Foo: 7
```

Note that the procedures are called in the reverse of the order with which they were added. This is because the default behaviour of `add-hook!` is to add its procedure to the *front* of the hook's procedure list. You can force `add-hook!` to add its procedure to the *end* of the list instead by providing a third `#t` argument on the second call to `add-hook!`.

```
(add-hook! hook (lambda (x y)
                  (display "Foo: ")
                  (display (+ x y))
                  (newline)))
(add-hook! hook (lambda (x y)
                  (display "Bar: ")
                  (display (* x y))
                  (newline))
                  #t)             ; <- Change here!

(run-hook hook 3 4)
⊣ Foo: 7
⊣ Bar: 12
```

6.11.6.2 Hook Reference

When you create a hook with `make-hook`, you must specify the arity of the procedures which can be added to the hook. If the arity is not given explicitly as an argument to `make-hook`, it defaults to zero. All procedures of a given hook must have the same arity, and when the procedures are invoked using `run-hook`, the number of arguments passed must match the arity specified at hook creation time.

The order in which procedures are added to a hook matters. If the third parameter to `add-hook!` is omitted or is equal to `#f`, the procedure is added in front of the procedures which might already be on that hook, otherwise the procedure is added at the end. The procedures are always called from the front to the end of the list when they are invoked via `run-hook`.

The ordering of the list of procedures returned by `hook->list` matches the order in which those procedures would be called if the hook was run using `run-hook`.

Note that the C functions in the following entries are for handling *Scheme-level* hooks in C. There are also *C-level* hooks which have their own interface (see Section 6.11.6.3 [C Hooks], page 291).

`make-hook` [*n_args*] [Scheme Procedure]
`scm_make_hook` (*n_args*) [C Function]
Create a hook for storing procedure of arity *n_args*. *n_args* defaults to zero. The returned value is a hook object to be used with the other hook procedures.

`hook?` *x* [Scheme Procedure]
`scm_hook_p` (*x*) [C Function]
Return `#t` if *x* is a hook, `#f` otherwise.

`hook-empty?` *hook* [Scheme Procedure]
`scm_hook_empty_p` (*hook*) [C Function]
Return `#t` if *hook* is an empty hook, `#f` otherwise.

`add-hook!` *hook proc* [*append_p*] [Scheme Procedure]
`scm_add_hook_x` (*hook, proc, append_p*) [C Function]
Add the procedure *proc* to the hook *hook*. The procedure is added to the end if *append_p* is true, otherwise it is added to the front. The return value of this procedure is not specified.

`remove-hook!` *hook proc* [Scheme Procedure]
`scm_remove_hook_x` (*hook, proc*) [C Function]
Remove the procedure *proc* from the hook *hook*. The return value of this procedure is not specified.

`reset-hook!` *hook* [Scheme Procedure]
`scm_reset_hook_x` (*hook*) [C Function]
Remove all procedures from the hook *hook*. The return value of this procedure is not specified.

`hook->list` *hook* [Scheme Procedure]
`scm_hook_to_list` (*hook*) [C Function]
Convert the procedure list of *hook* to a list.

`run-hook` *hook arg . . .* [Scheme Procedure]
`scm_run_hook` (*hook, args*) [C Function]
Apply all procedures from the hook *hook* to the arguments *arg . . .*. The order of the procedure application is first to last. The return value of this procedure is not specified.

If, in C code, you are certain that you have a hook object and well formed argument list for that hook, you can also use `scm_c_run_hook`, which is identical to `scm_run_hook` but does no type checking.

`void scm_c_run_hook` (*SCM hook, SCM args*) [C Function]
The same as `scm_run_hook` but without any type checking to confirm that *hook* is actually a hook object and that *args* is a well-formed list matching the arity of the hook.

For C code, `SCM_HOOKP` is a faster alternative to `scm_hook_p`:

`int SCM_HOOKP` (*x*) [C Macro]
Return 1 if *x* is a Scheme-level hook, 0 otherwise.

6.11.6.3 Hooks For C Code.

The hooks already described are intended to be populated by Scheme-level procedures. In addition to this, the Guile library provides an independent set of interfaces for the creation and manipulation of hooks that are designed to be populated by functions implemented in C.

The original motivation here was to provide a kind of hook that could safely be invoked at various points during garbage collection. Scheme-level hooks are unsuitable for this purpose as running them could itself require memory allocation, which would then invoke garbage collection recursively . . . However, it is also the case that these hooks are easier to work with than the Scheme-level ones if you only want to register C functions with them. So if that is mainly what your code needs to do, you may prefer to use this interface.

To create a C hook, you should allocate storage for a structure of type `scm_t_c_hook` and then initialize it using `scm_c_hook_init`.

`scm_t_c_hook` [C Type]
Data type for a C hook. The internals of this type should be treated as opaque.

`scm_t_c_hook_type` [C Enum]
Enumeration of possible hook types, which are:

`SCM_C_HOOK_NORMAL`
Type of hook for which all the registered functions will always be called.

`SCM_C_HOOK_OR`
Type of hook for which the sequence of registered functions will be called only until one of them returns C true (a non-NULL pointer).

`SCM_C_HOOK_AND`
Type of hook for which the sequence of registered functions will be called only until one of them returns C false (a NULL pointer).

`void scm_c_hook_init` (*scm_t_c_hook *hook, void *hook_data, scm_t_c_hook_type type*) [C Function]
Initialize the C hook at memory pointed to by *hook*. *type* should be one of the values of the `scm_t_c_hook_type` enumeration, and controls how the hook functions will be called. *hook_data* is a closure parameter that will be passed to all registered hook functions when they are called.

To add or remove a C function from a C hook, use `scm_c_hook_add` or `scm_c_hook_remove`. A hook function must expect three `void *` parameters which are, respectively:

hook_data The hook closure data that was specified at the time the hook was initialized by `scm_c_hook_init`.

func_data The function closure data that was specified at the time that that function was registered with the hook by `scm_c_hook_add`.

data The call closure data specified by the `scm_c_hook_run` call that runs the hook.

`scm_t_c_hook_function` [C Type]
Function type for a C hook function: takes three `void *` parameters and returns a `void *` result.

`void scm_c_hook_add` (*scm_t_c_hook *hook, scm_t_c_hook_function func, void *func_data, int appendp*) [C Function]
Add function *func*, with function closure data *func_data*, to the C hook *hook*. The new function is appended to the hook's list of functions if *appendp* is non-zero, otherwise prepended.

`void scm_c_hook_remove` (*scm_t_c_hook *hook, scm_t_c_hook_function func, void *func_data*) [C Function]
Remove function *func*, with function closure data *func_data*, from the C hook *hook*. `scm_c_hook_remove` checks both *func* and *func_data* so as to allow for the same *func* being registered multiple times with different closure data.

Finally, to invoke a C hook, call the `scm_c_hook_run` function specifying the hook and the call closure data for this run:

`void * scm_c_hook_run` (*scm_t_c_hook *hook, void *data*) [C Function]
Run the C hook *hook* will call closure data *data*. Subject to the variations for hook types `SCM_C_HOOK_OR` and `SCM_C_HOOK_AND`, `scm_c_hook_run` calls *hook*'s registered functions in turn, passing them the hook's closure data, each function's closure data, and the call closure data.

`scm_c_hook_run`'s return value is the return value of the last function to be called.

6.11.6.4 Hooks for Garbage Collection

Whenever Guile performs a garbage collection, it calls the following hooks in the order shown.

`scm_before_gc_c_hook` [C Hook]
C hook called at the very start of a garbage collection, after setting `scm_gc_running_p` to 1, but before entering the GC critical section.

If garbage collection is blocked because `scm_block_gc` is non-zero, GC exits early soon after calling this hook, and no further hooks will be called.

`scm_before_mark_c_hook` [C Hook]
C hook called before beginning the mark phase of garbage collection, after the GC thread has entered a critical section.

`scm_before_sweep_c_hook` [C Hook]
: C hook called before beginning the sweep phase of garbage collection. This is the same as at the end of the mark phase, since nothing else happens between marking and sweeping.

`scm_after_sweep_c_hook` [C Hook]
: C hook called after the end of the sweep phase of garbage collection, but while the GC thread is still inside its critical section.

`scm_after_gc_c_hook` [C Hook]
: C hook called at the very end of a garbage collection, after the GC thread has left its critical section.

`after-gc-hook` [Scheme Hook]
: Scheme hook with arity 0. This hook is run asynchronously (see Section 6.22.3 [Asyncs], page 445) soon after the GC has completed and any other events that were deferred during garbage collection have been processed. (Also accessible from C with the name `scm_after_gc_hook`.)

All the C hooks listed here have type `SCM_C_HOOK_NORMAL`, are initialized with hook closure data NULL, are invoked by `scm_c_hook_run` with call closure data NULL.

The Scheme hook `after-gc-hook` is particularly useful in conjunction with guardians (see Section 6.19.4 [Guardians], page 409). Typically, if you are using a guardian, you want to call the guardian after garbage collection to see if any of the objects added to the guardian have been collected. By adding a thunk that performs this call to `after-gc-hook`, you can ensure that your guardian is tested after every garbage collection cycle.

6.11.6.5 Hooks into the Guile REPL

6.12 Definitions and Variable Bindings

Scheme supports the definition of variables in different contexts. Variables can be defined at the top level, so that they are visible in the entire program, and variables can be defined locally to procedures and expressions. This is important for modularity and data abstraction.

6.12.1 Top Level Variable Definitions

At the top level of a program (i.e., not nested within any other expression), a definition of the form

```
(define a value)
```

defines a variable called `a` and sets it to the value *value*.

If the variable already exists in the current module, because it has already been created by a previous `define` expression with the same name, its value is simply changed to the new *value*. In this case, then, the above form is completely equivalent to

```
(set! a value)
```

This equivalence means that `define` can be used interchangeably with `set!` to change the value of variables at the top level of the REPL or a Scheme source file. It is useful during

interactive development when reloading a Scheme file that you have modified, because it allows the `define` expressions in that file to work as expected both the first time that the file is loaded and on subsequent occasions.

Note, though, that `define` and `set!` are not always equivalent. For example, a `set!` is not allowed if the named variable does not already exist, and the two expressions can behave differently in the case where there are imported variables visible from another module.

`define` *name value* [Scheme Syntax]
: Create a top level variable named *name* with value *value*. If the named variable already exists, just change its value. The return value of a `define` expression is unspecified.

The C API equivalents of `define` are `scm_define` and `scm_c_define`, which differ from each other in whether the variable name is specified as a `SCM` symbol or as a null-terminated C string.

`scm_define` (*sym, value*) [C Function]
`scm_c_define` (*const char *name, value*) [C Function]
: C equivalents of `define`, with variable name specified either by *sym*, a symbol, or by *name*, a null-terminated C string. Both variants return the new or preexisting variable object.

`define` (when it occurs at top level), `scm_define` and `scm_c_define` all create or set the value of a variable in the top level environment of the current module. If there was not already a variable with the specified name belonging to the current module, but a similarly named variable from another module was visible through having been imported, the newly created variable in the current module will shadow the imported variable, such that the imported variable is no longer visible.

Attention: Scheme definitions inside local binding constructs (see Section 6.12.2 [Local Bindings], page 294) act differently (see Section 6.12.3 [Internal Definitions], page 296).

Many people end up in a development style of adding and changing definitions at runtime, building out their program without restarting it. (You can do this using `reload-module`, the `reload` REPL command, the `load` procedure, or even just pasting code into a REPL.) If you are one of these people, you will find that sometimes there are some variables that you *don't* want to redefine all the time. For these, use `define-once`.

`define-once` *name value* [Scheme Syntax]
: Create a top level variable named *name* with value *value*, but only if *name* is not already bound in the current module.

Old Lispers probably know `define-once` under its Lisp name, `defvar`.

6.12.2 Local Variable Bindings

As opposed to definitions at the top level, which creates bindings that are visible to all code in a module, it is also possible to define variables which are only visible in a well-defined part of the program. Normally, this part of a program will be a procedure or a subexpression of a procedure.

With the constructs for local binding (`let`, `let*`, `letrec`, and `letrec*`), the Scheme language has a block structure like most other programming languages since the days of ALGOL 60. Readers familiar to languages like C or Java should already be used to this concept, but the family of `let` expressions has a few properties which are well worth knowing.

The most basic local binding construct is `let`.

`let` *bindings body* [syntax]

bindings has the form

```
((variable1 init1) ...)
```

that is zero or more two-element lists of a variable and an arbitrary expression each. All *variable* names must be distinct.

A `let` expression is evaluated as follows.

- All *init* expressions are evaluated.
- New storage is allocated for the *variables*.
- The values of the *init* expressions are stored into the variables.
- The expressions in *body* are evaluated in order, and the value of the last expression is returned as the value of the `let` expression.

The *init* expressions are not allowed to refer to any of the *variables*.

The other binding constructs are variations on the same theme: making new values, binding them to variables, and executing a body in that new, extended lexical context.

`let*` *bindings body* [syntax]

Similar to `let`, but the variable bindings are performed sequentially, that means that all *init* expression are allowed to use the variables defined on their left in the binding list.

A `let*` expression can always be expressed with nested `let` expressions.

```
(let* ((a 1) (b a))
   b)
≡
(let ((a 1))
  (let ((b a))
    b))
```

`letrec` *bindings body* [syntax]

Similar to `let`, but it is possible to refer to the *variable* from lambda expression created in any of the *inits*. That is, procedures created in the *init* expression can recursively refer to the defined variables.

```
(letrec ((even? (lambda (n)
                  (if (zero? n)
                      #t
                      (odd? (- n 1)))))
         (odd? (lambda (n)
                  (if (zero? n)
                      #f
```

```
                          (even? (- n 1))))))
  (even? 88))
⇒
#t
```

Note that while the *init* expressions may refer to the new variables, they may not access their values. For example, making the `even?` function above creates a closure (see Section 3.4 [About Closure], page 26) referencing the `odd?` variable. But `odd?` can't be called until after execution has entered the body.

`letrec*` *bindings body* [syntax]

Similar to `letrec`, except the *init* expressions are bound to their variables in order.

`letrec*` thus relaxes the letrec restriction, in that later *init* expressions may refer to the values of previously bound variables.

```
(letrec ((a 42)
         (b (+ a 10)))  ;; Illegal access
  (* a b))
;; The behavior of the expression above is unspecified

(letrec* ((a 42)
          (b (+ a 10)))
  (* a b))
⇒ 2184
```

There is also an alternative form of the `let` form, which is used for expressing iteration. Because of the use as a looping construct, this form (the *named let*) is documented in the section about iteration (see Section 6.13.4 [while do], page 301)

6.12.3 Internal definitions

A `define` form which appears inside the body of a `lambda`, `let`, `let*`, `letrec`, `letrec*` or equivalent expression is called an *internal definition*. An internal definition differs from a top level definition (see Section 6.12.1 [Top Level], page 293), because the definition is only visible inside the complete body of the enclosing form. Let us examine the following example.

```
(let ((frumble "froz"))
  (define banana (lambda () (apple 'peach)))
  (define apple (lambda (x) x))
  (banana))
⇒
peach
```

Here the enclosing form is a `let`, so the `define`s in the `let`-body are internal definitions. Because the scope of the internal definitions is the **complete** body of the `let`-expression, the `lambda`-expression which gets bound to the variable `banana` may refer to the variable `apple`, even though its definition appears lexically *after* the definition of `banana`. This is because a sequence of internal definition acts as if it were a `letrec*` expression.

```
(let ()
  (define a 1)
```

```
  (define b 2)
  (+ a b))
```

is equivalent to

```
(let ()
  (letrec* ((a 1) (b 2))
    (+ a b)))
```

Internal definitions may be mixed with non-definition expressions. If an expression precedes a definition, it is treated as if it were a definition of an unreferenced variable. So this:

```
(let ()
  (define a 1)
  (foo)
  (define b 2)
  (+ a b))
```

is equivalent to

```
(let ()
  (letrec* ((a 1) (_ (begin (foo) #f)) (b 2))
    (+ a b)))
```

Another noteworthy difference to top level definitions is that within one group of internal definitions all variable names must be distinct. Whereas on the top level a second define for a given variable acts like a `set!`, for internal definitions, duplicate bound identifiers signals an error.

As a historical note, it used to be that internal bindings were expanded in terms of `letrec`, not `letrec*`. This was the situation for the R5RS report and before. However with the R6RS, it was recognized that sequential definition was a more intuitive expansion, as in the following case:

```
(let ()
  (define a 1)
  (define b (+ a a))
  (+ a b))
```

Guile decided to follow the R6RS in this regard, and now expands internal definitions using `letrec*`. Relatedly, it used to be that internal definitions had to precede all expressions in the body; this restriction was relaxed in Guile 3.0.

6.12.4 Querying variable bindings

Guile provides a procedure for checking whether a symbol is bound in the top level environment.

`defined?` *sym* [*module*] [Scheme Procedure]
`scm_defined_p` (*sym*, *module*) [C Function]

Return `#t` if *sym* is defined in the module *module* or the current module when *module* is not specified; otherwise return `#f`.

6.12.5 Binding multiple return values

`define-values` *formals expression* [Syntax]

The *expression* is evaluated, and the *formals* are bound to the return values in the same way that the formals in a `lambda` expression are matched to the arguments in a procedure call.

```
(define-values (q r) (floor/ 10 3))
(list q r) ⇒ (3 1)

(define-values (x . y) (values 1 2 3))
x ⇒ 1
y ⇒ (2 3)

(define-values x (values 1 2 3))
x ⇒ (1 2 3)
```

6.13 Controlling the Flow of Program Execution

See Section 5.4.3 [Control Flow], page 68, for a discussion of how the more general control flow of Scheme affects C code.

6.13.1 Sequencing and Splicing

As an expression, the `begin` syntax is used to evaluate a sequence of sub-expressions in order. Consider the conditional expression below:

```
(if (> x 0)
    (begin (display "greater") (newline)))
```

If the test is true, we want to display "greater" to the current output port, then display a newline. We use `begin` to form a compound expression out of this sequence of sub-expressions.

`begin` *expr ...* [syntax]

The expression(s) are evaluated in left-to-right order and the value of the last expression is returned as the value of the `begin`-expression. This expression type is used when the expressions before the last one are evaluated for their side effects.

The `begin` syntax has another role in definition context (see Section 6.12.3 [Internal Definitions], page 296). A `begin` form in a definition context *splices* its subforms into its place. For example, consider the following procedure:

```
(define (make-seal)
  (define-sealant seal open)
  (values seal open))
```

Let us assume the existence of a `define-sealant` macro that expands out to some definitions wrapped in a `begin`, like so:

```
(define (make-seal)
  (begin
    (define seal-tag
```

```
      (list 'seal))
    (define (seal x)
      (cons seal-tag x))
    (define (sealed? x)
      (and (pair? x) (eq? (car x) seal-tag)))
    (define (open x)
      (if (sealed? x)
          (cdr x)
          (error "Expected a sealed value:" x))))
  (values seal open))
```

Here, because the `begin` is in definition context, its subforms are *spliced* into the place of the `begin`. This allows the definitions created by the macro to be visible to the following expression, the `values` form.

It is a fine point, but splicing and sequencing are different. It can make sense to splice zero forms, because it can make sense to have zero internal definitions before the expressions in a procedure or lexical binding form. However it does not make sense to have a sequence of zero expressions, because in that case it would not be clear what the value of the sequence would be, because in a sequence of zero expressions, there can be no last value. Sequencing zero expressions is an error.

It would be more elegant in some ways to eliminate splicing from the Scheme language, and without macros (see Section 6.10 [Macros], page 261), that would be a good idea. But it is useful to be able to write macros that expand out to multiple definitions, as in `define-sealant` above, so Scheme abuses the `begin` form for these two tasks.

6.13.2 Simple Conditional Evaluation

Guile provides three syntactic constructs for conditional evaluation. `if` is the normal if-then-else expression (with an optional else branch), `cond` is a conditional expression with multiple branches and `case` branches if an expression has one of a set of constant values.

`if` *test consequent* [*alternate*] [syntax]

All arguments may be arbitrary expressions. First, *test* is evaluated. If it returns a true value, the expression *consequent* is evaluated and *alternate* is ignored. If *test* evaluates to `#f`, *alternate* is evaluated instead. The values of the evaluated branch (*consequent* or *alternate*) are returned as the values of the `if` expression.

When *alternate* is omitted and the *test* evaluates to `#f`, the value of the expression is not specified.

When you go to write an `if` without an alternate (a *one-armed* `if`), part of what you are expressing is that you don't care about the return value (or values) of the expression. As such, you are more interested in the *effect* of evaluating the consequent expression. (By convention, we use the word *statement* to refer to an expression that is evaluated for effect, not for value).

In such a case, it is considered more clear to express these intentions with these special forms, `when` and `unless`. As an added bonus, these forms accept multiple statements to evaluate, which are implicitly wrapped in a `begin`.

`when` *test statement1 statement2 ...* [Scheme Syntax]
`unless` *test statement1 statement2 ...* [Scheme Syntax]

The actual definitions of these forms are in many ways their most clear documentation:

```
(define-syntax-rule (when test stmt stmt* ...)
  (if test (begin stmt stmt* ...)))

(define-syntax-rule (unless condition stmt stmt* ...)
  (if (not test) (begin stmt stmt* ...)))
```

That is to say, `when` evaluates its consequent statements in order if *test* is true. `unless` is the opposite: it evaluates the statements if *test* is false.

`cond` *clause1 clause2 . . .* [syntax]

Each `cond`-clause must look like this:

```
(test expression ...)
```

where *test* and *expression* are arbitrary expressions, or like this

```
(test => expression)
```

where *expression* must evaluate to a procedure.

The *test*s of the clauses are evaluated in order and as soon as one of them evaluates to a true value, the corresponding *expression*s are evaluated in order and the last value is returned as the value of the `cond`-expression. For the `=>` clause type, *expression* is evaluated and the resulting procedure is applied to the value of *test*. The result of this procedure application is then the result of the `cond`-expression.

One additional `cond`-clause is available as an extension to standard Scheme:

```
(test guard => expression)
```

where *guard* and *expression* must evaluate to procedures. For this clause type, *test* may return multiple values, and `cond` ignores its boolean state; instead, `cond` evaluates *guard* and applies the resulting procedure to the value(s) of *test*, as if *guard* were the *consumer* argument of `call-with-values`. If the result of that procedure call is a true value, it evaluates *expression* and applies the resulting procedure to the value(s) of *test*, in the same manner as the *guard* was called.

The *test* of the last *clause* may be the symbol `else`. Then, if none of the preceding *test*s is true, the *expression*s following the `else` are evaluated to produce the result of the `cond`-expression.

`case` *key clause1 clause2 . . .* [syntax]

key may be any expression, and the *clause*s must have the form

```
((datum1 ...) expr1 expr2 ...)
```

or

```
((datum1 ...) => expression)
```

and the last *clause* may have the form

```
(else expr1 expr2 ...)
```

or

```
(else => expression)
```

All *datum*s must be distinct. First, *key* is evaluated. The result of this evaluation is compared against all *datum* values using `eqv?`. When this comparison succeeds, the expression(s) following the *datum* are evaluated from left to right, returning the value of the last expression as the result of the `case` expression.

If the *key* matches no *datum* and there is an `else`-clause, the expressions following the `else` are evaluated. If there is no such clause, the result of the expression is unspecified.

For the `=>` clause types, *expression* is evaluated and the resulting procedure is applied to the value of *key*. The result of this procedure application is then the result of the `case`-expression.

6.13.3 Conditional Evaluation of a Sequence of Expressions

`and` and `or` evaluate all their arguments in order, similar to `begin`, but evaluation stops as soon as one of the expressions evaluates to false or true, respectively.

`and` *expr ...* [syntax]

Evaluate the *expr*s from left to right and stop evaluation as soon as one expression evaluates to `#f`; the remaining expressions are not evaluated. The value of the last evaluated expression is returned. If no expression evaluates to `#f`, the value of the last expression is returned.

If used without expressions, `#t` is returned.

`or` *expr ...* [syntax]

Evaluate the *expr*s from left to right and stop evaluation as soon as one expression evaluates to a true value (that is, a value different from `#f`); the remaining expressions are not evaluated. The value of the last evaluated expression is returned. If all expressions evaluate to `#f`, `#f` is returned.

If used without expressions, `#f` is returned.

6.13.4 Iteration mechanisms

Scheme has only few iteration mechanisms, mainly because iteration in Scheme programs is normally expressed using recursion. Nevertheless, R5RS defines a construct for programming loops, calling `do`. In addition, Guile has an explicit looping syntax called `while`.

`do` ((*variable init* [*step*]) ...) (*test expr ...*) *body ...* [syntax]

Bind *variable*s and evaluate *body* until *test* is true. The return value is the last *expr* after *test*, if given. A simple example will illustrate the basic form,

```
(do ((i 1 (1+ i)))
    ((> i 4))
  (display i))
⊣ 1234
```

Or with two variables and a final return value,

```
(do ((i 1 (1+ i))
     (p 3 (* 3 p)))
    ((> i 4)
     p)
```

```
  (format #t "3**~s is ~s\n" i p))
⊣
3**1 is 3
3**2 is 9
3**3 is 27
3**4 is 81
⇒
789
```

The *variable* bindings are established like a `let`, in that the expressions are all evaluated and then all bindings made. When iterating, the optional *step* expressions are evaluated with the previous bindings in scope, then new bindings all made.

The *test* expression is a termination condition. Looping stops when the *test* is true. It's evaluated before running the *body* each time, so if it's true the first time then *body* is not run at all.

The optional *expr*s after the *test* are evaluated at the end of looping, with the final *variable* bindings available. The last *expr* gives the return value, or if there are no *expr*s the return value is unspecified.

Each iteration establishes bindings to fresh locations for the *variable*s, like a new `let` for each iteration. This is done for *variable*s without *step* expressions too. The following illustrates this, showing how a new `i` is captured by the `lambda` in each iteration (see Section 3.4 [The Concept of Closure], page 26).

```
(define lst '())
(do ((i 1 (1+ i)))
    ((> i 4))
  (set! lst (cons (lambda () i) lst)))
(map (lambda (proc) (proc)) lst)
⇒
(4 3 2 1)
```

`while` *cond body ...* [syntax]

Run a loop executing the *body* forms while *cond* is true. *cond* is tested at the start of each iteration, so if it's `#f` the first time then *body* is not executed at all.

Within `while`, two extra bindings are provided, they can be used from both *cond* and *body*.

`break` *break-arg ...* [Scheme Procedure]

Break out of the `while` form.

`continue` [Scheme Procedure]

Abandon the current iteration, go back to the start and test *cond* again, etc.

If the loop terminates normally, by the *cond* evaluating to `#f`, then the `while` expression as a whole evaluates to `#f`. If it terminates by a call to `break` with some number of arguments, those arguments are returned from the `while` expression, as multiple values. Otherwise if it terminates by a call to `break` with no arguments, then return value is `#t`.

```
(while #f (error "not reached")) ⇒ #f
```

```
(while #t (break)) ⇒ #t
(while #t (break 1 2 3)) ⇒ 1 2 3
```

Each `while` form gets its own `break` and `continue` procedures, operating on that `while`. This means when loops are nested the outer `break` can be used to escape all the way out. For example,

```
(while (test1)
  (let ((outer-break break))
    (while (test2)
      (if (something)
        (outer-break #f))
      ...)))
```

Note that each `break` and `continue` procedure can only be used within the dynamic extent of its `while`. Outside the `while` their behaviour is unspecified.

Another very common way of expressing iteration in Scheme programs is the use of the so-called *named let*.

Named let is a variant of `let` which creates a procedure and calls it in one step. Because of the newly created procedure, named let is more powerful than `do`–it can be used for iteration, but also for arbitrary recursion.

`let` *variable bindings body* [syntax]

For the definition of *bindings* see the documentation about `let` (see Section 6.12.2 [Local Bindings], page 294).

Named `let` works as follows:

- A new procedure which accepts as many arguments as are in *bindings* is created and bound locally (using `let`) to *variable*. The new procedure's formal argument names are the name of the *variables*.
- The *body* expressions are inserted into the newly created procedure.
- The procedure is called with the *init* expressions as the formal arguments.

The next example implements a loop which iterates (by recursion) 1000 times.

```
(let lp ((x 1000))
  (if (positive? x)
      (lp (- x 1))
      x))
⇒
0
```

6.13.5 Prompts

Prompts are control-flow barriers between different parts of a program. In the same way that a user sees a shell prompt (e.g., the Bash prompt) as a barrier between the operating system and her programs, Scheme prompts allow the Scheme programmer to treat parts of programs as if they were running in different operating systems.

We use this roundabout explanation because, unless you're a functional programming junkie, you probably haven't heard the term, "delimited, composable continuation". That's OK; it's a relatively recent topic, but a very useful one to know about.

6.13.5.1 Prompt Primitives

Guile's primitive delimited control operators are `call-with-prompt` and `abort-to-prompt`.

`call-with-prompt` *tag thunk handler* [Scheme Procedure]
Set up a prompt, and call *thunk* within that prompt.

During the dynamic extent of the call to *thunk*, a prompt named *tag* will be present in the dynamic context, such that if a user calls `abort-to-prompt` (see below) with that tag, control rewinds back to the prompt, and the *handler* is run.

handler must be a procedure. The first argument to *handler* will be the state of the computation begun when *thunk* was called, and ending with the call to `abort-to-prompt`. The remaining arguments to *handler* are those passed to `abort-to-prompt`.

`make-prompt-tag` [*stem*] [Scheme Procedure]
Make a new prompt tag. A prompt tag is simply a unique object. Currently, a prompt tag is a fresh pair. This may change in some future Guile version.

`default-prompt-tag` [Scheme Procedure]
Return the default prompt tag. Having a distinguished default prompt tag allows some useful prompt and abort idioms, discussed in the next section. Note that `default-prompt-tag` is actually a parameter, and so may be dynamically rebound using `parameterize`. See Section 6.13.12 [Parameters], page 326.

`abort-to-prompt` *tag val1 val2 ...* [Scheme Procedure]
Unwind the dynamic and control context to the nearest prompt named *tag*, also passing the given values.

C programmers may recognize `call-with-prompt` and `abort-to-prompt` as a fancy kind of `setjmp` and `longjmp`, respectively. Prompts are indeed quite useful as non-local escape mechanisms. Guile's `with-exception-handler` and `raise-exception` are implemented in terms of prompts. Prompts are more convenient than `longjmp`, in that one has the opportunity to pass multiple values to the jump target.

Also unlike `longjmp`, the prompt handler is given the full state of the process that was aborted, as the first argument to the prompt's handler. That state is the *continuation* of the computation wrapped by the prompt. It is a *delimited continuation*, because it is not the whole continuation of the program; rather, just the computation initiated by the call to `call-with-prompt`.

The continuation is a procedure, and may be reinstated simply by invoking it, with any number of values. Here's where things get interesting, and complicated as well. Besides being described as delimited, continuations reified by prompts are also *composable*, because invoking a prompt-saved continuation composes that continuation with the current one.

Imagine you have saved a continuation via call-with-prompt:

```
(define cont
  (call-with-prompt
   ;; tag
   'foo
   ;; thunk
   (lambda ()
```

```
    (+ 34 (abort-to-prompt 'foo)))
  ;; handler
  (lambda (k) k)))
```

The resulting continuation is the addition of 34. It's as if you had written:

```
(define cont
  (lambda (x)
    (+ 34 x)))
```

So, if we call `cont` with one numeric value, we get that number, incremented by 34:

```
(cont 8)
⇒ 42
(* 2 (cont 8))
⇒ 84
```

The last example illustrates what we mean when we say, "composes with the current continuation". We mean that there is a current continuation – some remaining things to compute, like `(lambda (x) (* x 2))` – and that calling the saved continuation doesn't wipe out the current continuation, it composes the saved continuation with the current one.

We're belaboring the point here because traditional Scheme continuations, as discussed in the next section, aren't composable, and are actually less expressive than continuations captured by prompts. But there's a place for them both.

Before moving on, we should mention that if the handler of a prompt is a `lambda` expression, and the first argument isn't referenced, an abort to that prompt will not cause a continuation to be reified. This can be an important efficiency consideration to keep in mind.

One example where this optimization matters is *escape continuations*. Escape continuations are delimited continuations whose only use is to make a non-local exit—i.e., to escape from the current continuation. A common use of escape continuations is when handling an exception (see Section 6.13.8 [Exceptions], page 311).

The constructs below are syntactic sugar atop prompts to simplify the use of escape continuations.

`call-with-escape-continuation` *proc* [Scheme Procedure]
`call/ec` *proc* [Scheme Procedure]

Call *proc* with an escape continuation.

In the example below, the *return* continuation is used to escape the continuation of the call to `fold`.

```
(use-modules (ice-9 control)
             (srfi srfi-1))

(define (prefix x lst)
  ;; Return all the elements before the first occurrence
  ;; of X in LST.
  (call/ec
    (lambda (return)
      (fold (lambda (element prefix)
              (if (equal? element x)
```

```
                    (return (reverse prefix))  ; escape 'fold'
                    (cons element prefix)))
                '()
                lst))))

(prefix 'a '(0 1 2 a 3 4 5))
⇒ (0 1 2)
```

`let-escape-continuation` *k body ...* [Scheme Syntax]
`let/ec` *k body ...* [Scheme Syntax]

Bind *k* within *body* to an escape continuation.

This is equivalent to `(call/ec (lambda (k) body ...))`.

Additionally there is another helper primitive exported by `(ice-9 control)`, so load up that module for `suspendable-continuation?`:

```
(use-modules (ice-9 control))
```

`suspendable-continuation?` *tag* [Scheme Procedure]

Return `#t` if a call to `abort-to-prompt` with the prompt tag *tag* would produce a delimited continuation that could be resumed later.

Almost all continuations have this property. The exception is where some code between the `call-with-prompt` and the `abort-to-prompt` recursed through C for some reason, the `abort-to-prompt` will succeed but any attempt to resume the continuation (by calling it) would fail. This is because composing a saved continuation with the current continuation involves relocating the stack frames that were saved from the old stack onto a (possibly) new position on the new stack, and Guile can only do this for stack frames that it created for Scheme code, not stack frames created by the C compiler. It's a bit gnarly but if you stick with Scheme, you won't have any problem.

If no prompt is found with the given tag, this procedure just returns `#f`.

6.13.5.2 Shift, Reset, and All That

There is a whole zoo of delimited control operators, and as it does not seem to be a bounded set, Guile implements support for them in a separate module:

```
(use-modules (ice-9 control))
```

Firstly, we have a helpful abbreviation for the `call-with-prompt` operator.

`%` *expr* [Scheme Syntax]
`%` *expr handler* [Scheme Syntax]
`%` *tag expr handler* [Scheme Syntax]

Evaluate *expr* in a prompt, optionally specifying a tag and a handler. If no tag is given, the default prompt tag is used.

If no handler is given, a default handler is installed. The default handler accepts a procedure of one argument, which will be called on the captured continuation, within a prompt.

Sometimes it's easier just to show code, as in this case:

```
(define (default-prompt-handler k proc)
```

```
(% (default-prompt-tag)
   (proc k)
   default-prompt-handler))
```

The `%` symbol is chosen because it looks like a prompt.

Likewise there is an abbreviation for `abort-to-prompt`, which assumes the default prompt tag:

`abort` *val1 val2 ...* [Scheme Procedure]

Abort to the default prompt tag, passing *val1 val2 ...* to the handler.

As mentioned before, `(ice-9 control)` also provides other delimited control operators. This section is a bit technical, and first-time users of delimited continuations should probably come back to it after some practice with `%`.

Still here? So, when one implements a delimited control operator like `call-with-prompt`, one needs to make two decisions. Firstly, does the handler run within or outside the prompt? Having the handler run within the prompt allows an abort inside the handler to return to the same prompt handler, which is often useful. However it prevents tail calls from the handler, so it is less general.

Similarly, does invoking a captured continuation reinstate a prompt? Again we have the tradeoff of convenience versus proper tail calls.

These decisions are captured in the Felleisen *F* operator. If neither the continuations nor the handlers implicitly add a prompt, the operator is known as –*F*–. This is the case for Guile's `call-with-prompt` and `abort-to-prompt`.

If both continuation and handler implicitly add prompts, then the operator is +*F*+. `shift` and `reset` are such operators.

`reset` *body1 body2 ...* [Scheme Syntax]

Establish a prompt, and evaluate *body1 body2 ...* within that prompt.

The prompt handler is designed to work with `shift`, described below.

`shift` *cont body1 body2 ...* [Scheme Syntax]

Abort to the nearest `reset`, and evaluate *body1 body2 ...* in a context in which the captured continuation is bound to *cont*.

As mentioned above, taken together, the *body1 body2 ...* expressions and the invocations of *cont* implicitly establish a prompt.

Interested readers are invited to explore Oleg Kiselyov's wonderful web site at `http://okmij.org/ftp/`, for more information on these operators.

6.13.6 Continuations

A "continuation" is the code that will execute when a given function or expression returns. For example, consider

```
(define (foo)
  (display "hello\n")
  (display (bar)) (newline)
  (exit))
```

The continuation from the call to `bar` comprises a `display` of the value returned, a `newline` and an `exit`. This can be expressed as a function of one argument.

```
(lambda (r)
  (display r) (newline)
  (exit))
```

In Scheme, continuations are represented as special procedures just like this. The special property is that when a continuation is called it abandons the current program location and jumps directly to that represented by the continuation.

A continuation is like a dynamic label, capturing at run-time a point in program execution, including all the nested calls that have lead to it (or rather the code that will execute when those calls return).

Continuations are created with the following functions.

`call-with-current-continuation` *proc* [Scheme Procedure]
`call/cc` *proc* [Scheme Procedure]

Capture the current continuation and call `(`*proc cont*`)` with it. The return value is the value returned by *proc*, or when `(`*cont value*`)` is later invoked, the return is the *value* passed.

Normally *cont* should be called with one argument, but when the location resumed is expecting multiple values (see Section 6.13.7 [Multiple Values], page 309) then they should be passed as multiple arguments, for instance `(`*cont x y z*`)`.

cont may only be used from the same side of a continuation barrier as it was created (see Section 6.13.14 [Continuation Barriers], page 331), and in a multi-threaded program only from the thread in which it was created.

The call to *proc* is not part of the continuation captured, it runs only when the continuation is created. Often a program will want to store *cont* somewhere for later use; this can be done in *proc*.

The `call` in the name `call-with-current-continuation` refers to the way a call to *proc* gives the newly created continuation. It's not related to the way a call is used later to invoke that continuation.

`call/cc` is an alias for `call-with-current-continuation`. This is in common use since the latter is rather long.

Here is a simple example,

```
(define kont #f)
(format #t "the return is ~a\n"
        (call/cc (lambda (k)
                   (set! kont k)
                   1)))
⇒ the return is 1

(kont 2)
⇒ the return is 2
```

`call/cc` captures a continuation in which the value returned is going to be displayed by `format`. The `lambda` stores this in `kont` and gives an initial return `1` which is displayed. The later invocation of `kont` resumes the captured point, but this time returning `2`, which is displayed.

When Guile is run interactively, a call to `format` like this has an implicit return back to the read-eval-print loop. `call/cc` captures that like any other return, which is why interactively `kont` will come back to read more input.

C programmers may note that `call/cc` is like `setjmp` in the way it records at runtime a point in program execution. A call to a continuation is like a `longjmp` in that it abandons the present location and goes to the recorded one. Like `longjmp`, the value passed to the continuation is the value returned by `call/cc` on resuming there. However `longjmp` can only go up the program stack, but the continuation mechanism can go anywhere.

When a continuation is invoked, `call/cc` and subsequent code effectively "returns" a second time. It can be confusing to imagine a function returning more times than it was called. It may help instead to think of it being stealthily re-entered and then program flow going on as normal.

`dynamic-wind` (see Section 6.13.10 [Dynamic Wind], page 320) can be used to ensure setup and cleanup code is run when a program locus is resumed or abandoned through the continuation mechanism.

Continuations are a powerful mechanism, and can be used to implement almost any sort of control structure, such as loops, coroutines, or exception handlers.

However the implementation of continuations in Guile is not as efficient as one might hope, because Guile is designed to cooperate with programs written in other languages, such as C, which do not know about continuations. Basically continuations are captured by a block copy of the stack, and resumed by copying back.

For this reason, continuations captured by `call/cc` should be used only when there is no other simple way to achieve the desired result, or when the elegance of the continuation mechanism outweighs the need for performance.

Escapes upwards from loops or nested functions are generally best handled with prompts (see Section 6.13.5 [Prompts], page 303). Coroutines can be efficiently implemented with cooperating threads (a thread holds a full program stack but doesn't copy it around the way continuations do).

6.13.7 Returning and Accepting Multiple Values

Scheme allows a procedure to return more than one value to its caller. This is quite different to other languages which only allow single-value returns. Returning multiple values is different from returning a list (or pair or vector) of values to the caller, because conceptually not *one* compound object is returned, but several distinct values.

The primitive procedures for handling multiple values are `values` and `call-with-values`. `values` is used for returning multiple values from a procedure. This is done by placing a call to `values` with zero or more arguments in tail position in a procedure body. `call-with-values` combines a procedure returning multiple values with a procedure which accepts these values as parameters.

`values` *arg ...* [Scheme Procedure]
`scm_values` (*args*) [C Function]
Delivers all of its arguments to its continuation. Except for continuations created by the `call-with-values` procedure, all continuations take exactly one value. The effect of passing no value or more than one value to continuations that were not created by `call-with-values` is unspecified.

For `scm_values`, *args* is a list of arguments and the return is a multiple-values object which the caller can return. In the current implementation that object shares structure with *args*, so *args* should not be modified subsequently.

`SCM scm_c_values` (*SCM *base, size_t n*) [C Function]
`scm_c_values` is an alternative to `scm_values`. It creates a new values object, and copies into it the *n* values starting from *base*.

Currently this creates a list and passes it to `scm_values`, but we expect that in the future we will be able to use a more efficient representation.

`size_t scm_c_nvalues` (*SCM obj*) [C Function]
If *obj* is a multiple-values object, returns the number of values it contains. Otherwise returns 1.

`SCM scm_c_value_ref` (*SCM obj, size_t idx*) [C Function]
Returns the value at the position specified by *idx* in *obj*. Note that *obj* will ordinarily be a multiple-values object, but it need not be. Any other object represents a single value (itself), and is handled appropriately.

`call-with-values` *producer consumer* [Scheme Procedure]
Calls its *producer* argument with no values and a continuation that, when passed some values, calls the *consumer* procedure with those values as arguments. The continuation for the call to *consumer* is the continuation of the call to `call-with-values`.

```
(call-with-values (lambda () (values 4 5))
                  (lambda (a b) b))
⇒ 5

(call-with-values * -)
⇒ -1
```

In addition to the fundamental procedures described above, Guile has a module which exports a syntax called `receive`, which is much more convenient. This is in the `(ice-9 receive)` and is the same as specified by SRFI-8 (see Section 7.5.7 [SRFI-8], page 607).

```
(use-modules (ice-9 receive))
```

`receive` *formals expr body ...* [library syntax]
Evaluate the expression *expr*, and bind the result values (zero or more) to the formal arguments in *formals*. *formals* is a list of symbols, like the argument list in a `lambda` (see Section 6.9.1 [Lambda], page 248). After binding the variables, the expressions in *body ...* are evaluated in order, the return value is the result from the last expression.

For example getting results from `partition` in SRFI-1 (see Section 7.5.3 [SRFI-1], page 584),

```
(receive (odds evens)
    (partition odd? '(7 4 2 8 3))
  (display odds)
  (display " and ")
  (display evens))
⊣ (7 3) and (4 2 8)
```

6.13.8 Exceptions

What happens when things go wrong? Guile's exception facility exists to help answer this question, allowing programs to describe the problem and to handle the situation in a flexible way.

When a program runs into a problem, such as division by zero, it will raise an exception. Sometimes exceptions get raised by Guile on a program's behalf. Sometimes a program will want to raise exceptions of its own. Raising an exception stops the current computation and instead invokes the current exception handler, passing it an exception object describing the unexpected situation.

Usually an exception handler will unwind the computation back to some kind of safe point. For example, typical logic for a key press driven application might look something like this:

```
main-loop:
  read the next key press and call dispatch-key

dispatch-key:
  lookup the key in a keymap and call an appropriate procedure,
  say find-file

find-file:
  interactively read the required file name, then call
  find-specified-file

find-specified-file:
  check whether file exists; if not, raise an exception
  ...
```

In this case, `main-loop` can install an exception handler that would cause any exception raised inside `dispatch-key` to print a warning and jump back to the main loop.

The following subsections go into more detail about exception objects, raising exceptions, and handling exceptions. It also presents a historical interface that was used in Guile's first 25 years and which won't be going away any time soon.

6.13.8.1 Exception Objects

When Guile encounters an exceptional situation, it raises an exception, where the exception is an object that describes the exceptional situation. Exception objects are structured data, built on the record facility (see Section 6.6.17 [Records], page 221).

`&exception` [Exception Type]
: The base exception type. All exception objects are composed of instances of subtypes of `&exception`.

`exception-type?` *obj* [Scheme Procedure]
: Return true if *obj* is an exception type.

Exception types exist in a hierarchy. New exception types can be defined using `make-exception-type`.

`make-exception-type` *id parent field-names* [Scheme Procedure]
: Return a new exception type named *id*, inheriting from *parent*, and with the fields whose names are listed in *field-names*. *field-names* must be a list of symbols and must not contain names already used by *parent* or one of its supertypes.

Exception type objects are record type objects, and as such, one can use `record-constructor` on an exception type to get its constructor. The constructor will take as many arguments as the exception has fields (including supertypes). See Section 6.6.17 [Records], page 221.

However, `record-predicate` and `record-accessor` aren't usually what you want to use as exception type predicates and field accessors. The reason is, instances of exception types can be composed into *compound exceptions*. Exception accessors should pick out the specific component of a compound exception, and then access the field on that specific component.

`make-exception` *exceptions ...* [Scheme Procedure]
: Return an exception object composed of *exceptions*.

`exception?` *obj* [Scheme Procedure]
: Return true if *obj* is an exception object.

`exception-predicate` *type* [Scheme Procedure]
: Return a procedure that will return true if its argument is a simple exception that is an instance of *type*, or a compound exception composed of such an instance.

`exception-accessor` *rtd proc* [Scheme Procedure]
: Return a procedure that will tail-call *proc* on an instance of the exception type *rtd*, or on the component of a compound exception that is an instance of *rtd*.

Compound exceptions are useful to separately express the different aspects of a situation. For example, compound exceptions allow a programmer to say that "this situation is a programming error, and also here's a useful message to show to the user, and here are some relevant objects that can give more information about the error". This error could be composed of instances of the `&programming-error`, `&message`, and `&irritants` exception types.

The subtyping relationship in exceptions is useful to let different-but-similar situations to be treated the same; for example there are many varieties of programming errors (for example, divide-by-zero or type mismatches), but perhaps there are common ways that the user would like to handle them all, and that common way might be different than how one might handle an error originating outside the program (for example, a file-not-found error).

The standard exception hierarchy in Guile takes its cues from R6RS, though the names of some of the types are different. See Section 7.6.2.12 [rnrs exceptions], page 678, for more details.

To have access to Guile's exception type hierarchy, import the `(ice-9 exceptions)` module:

```
(use-modules (ice-9 exceptions))
```

The following diagram gives an overview of the standard exception type hierarchy.

```
&exception
|- &warning
|- &message
|- &irritants
|- &origin
\- &error
   |- &external-error
   \- &programming-error
      |- &assertion-failure
      |- &non-continuable
      |- &implementation-restriction
      |- &lexical
      |- &syntax
      \- &undefined-variable
```

`&warning` [Exception Type]

An exception type denoting warnings. These are usually raised using `#:continuable? #t`; see the `raise-exception` documentation for more.

`make-warning` [Scheme Procedure]
`warning?` *obj* [Scheme Procedure]

Constructor and predicate for `&warning` exception objects.

`&message` *message* [Exception Type]

An exception type that provides a message to display to the user. Usually used as a component of a compound exception.

`make-exception-with-message` *message* [Scheme Procedure]
`exception-with-message?` *obj* [Scheme Procedure]
`exception-message` *exn* [Scheme Procedure]

Constructor, predicate, and accessor for `&message` exception objects.

`&irritants` *irritants* [Exception Type]

An exception type that provides a list of objects that were unexpected in some way. Usually used as a component of a compound exception.

`make-exception-with-irritants` *irritants* [Scheme Procedure]
`exception-with-irritants?` *obj* [Scheme Procedure]
`exception-irritants` *exn* [Scheme Procedure]

Constructor, predicate, and accessor for `&irritants` exception objects.

`&origin` *origin* [Exception Type]
An exception type that indicates the origin of an exception, typically expressed as a procedure name, as a symbol. Usually used as a component of a compound exception.

`make-exception-with-origin` *origin* [Scheme Procedure]
`exception-with-origin?` *obj* [Scheme Procedure]
`exception-origin` *exn* [Scheme Procedure]
Constructor, predicate, and accessor for `&origin` exception objects.

`&error` [Exception Type]
An exception type denoting errors: situations that are not just exceptional, but wrong.

`make-error` [Scheme Procedure]
`error?` *obj* [Scheme Procedure]
Constructor and predicate for `&error` exception objects.

`&external-error` [Exception Type]
An exception type denoting errors that proceed from the interaction of the program with the world, for example a "file not found" error.

`make-external-error` [Scheme Procedure]
`external-error?` *obj* [Scheme Procedure]
Constructor and predicate for `&external-error` exception objects.

`&programming-error` [Exception Type]
An exception type denoting errors that proceed from inside a program: type mismatches and so on.

`make-programming-error` [Scheme Procedure]
`programming-error?` *obj* [Scheme Procedure]
Constructor and predicate for `&programming-error` exception objects.

`&non-continuable` [Exception Type]
An exception type denoting errors that proceed from inside a program: type mismatches and so on.

`make-non-continuable-error` [Scheme Procedure]
`non-continuable-error?` *obj* [Scheme Procedure]
Constructor and predicate for `&non-continuable` exception objects.

`&lexical` [Exception Type]
An exception type denoting lexical errors, for example unbalanced parentheses.

`make-lexical-error` [Scheme Procedure]
`lexical-error?` *obj* [Scheme Procedure]
Constructor and predicate for `&lexical` exception objects.

`&syntax` *form subform* [Exception Type]
An exception type denoting syntax errors, for example a `cond` expression with invalid syntax. The *form* field indicates the form containing the error, and *subform* indicates the unexpected subcomponent, or `#f` if unavailable.

`make-syntax-error` *form subform* [Scheme Procedure]
`syntax-error?` *obj* [Scheme Procedure]
`syntax-error-form` *exn* [Scheme Procedure]
`syntax-error-subform` *exn* [Scheme Procedure]
Constructor, predicate, and accessors for `&syntax` exception objects.

`&undefined-variable` [Exception Type]
An exception type denoting undefined variables.

`make-undefine-variable-error` [Scheme Procedure]
`undefined-variable-error?` *obj* [Scheme Procedure]
Constructor and predicate for `&undefined-variable` exception objects.

Incidentally, the `(ice-9 exceptions)` module also includes a `define-exception-type` macro that can be used to conveniently add new exception types to the hierarchy.

`define-exception-type` *name parent constructor predicate* (*field accessor*) ... [Syntax]
Define *name* to be a new exception type, inheriting from *parent*. Define *constructor* and *predicate* to be the exception constructor and predicate, respectively, and define an *accessor* for each *field*.

6.13.8.2 Raising and Handling Exceptions

An exception object describes an exceptional situation. To bring that description to the attention of the user or to handle the situation programmatically, the first step is to *raise* the exception.

`raise-exception` *obj* [*#:continuable=#f*] [Scheme Procedure]
Raise an exception by invoking the current exception handler on *obj*. The handler is called with a continuation whose dynamic environment is that of the call to `raise`, except that the current exception handler is the one that was in place when the handler being called was installed.

If *continuable?* is true, the handler is invoked in tail position relative to the `raise-exception` call. Otherwise if the handler returns, a non-continuable exception of type `&non-continuable` is raised in the same dynamic environment as the handler.

As the above description notes, Guile has a notion of a *current exception handler*. At the REPL, this exception handler may enter a recursive debugger; in a standalone program, it may simply print a representation of the error and exit.

To establish an exception handler within the dynamic extent of a call, use `with-exception-handler`.

`with-exception-handler` *handler thunk* [*#:unwind?=#f*] [*#:unwind-for-type=#t*] [Scheme Procedure]
Establish *handler*, a procedure of one argument, as the current exception handler during the dynamic extent of invoking *thunk*.

If `raise-exception` is called during the dynamic extent of invoking *thunk*, *handler* will be invoked on the argument of `raise-exception`.

There are two kinds of exception handlers: unwinding and non-unwinding.

By default, exception handlers are non-unwinding. Unless `with-exception-handler` was invoked with `#:unwind? #t`, exception handlers are invoked within the continuation of the error, without unwinding the stack. The dynamic environment of the handler call will be that of the `raise-exception` call, with the difference that the current exception handler will be "unwound" to the `\"outer\"` handler (the one that was in place when the corresponding `with-exception-handler` was called).

However, it's often the case that one would like to handle an exception by unwinding the computation to an earlier state and running the error handler there. After all, unless the `raise-exception` call is continuable, the exception handler needs to abort the continuation. To support this use case, if `with-exception-handler` was invoked with `#:unwind? #t` is true, `raise-exception` will first unwind the stack by invoking an *escape continuation* (see Section 6.13.5.1 [Prompt Primitives], page 304), and then invoke the handler with the continuation of the `with-exception-handler` call.

Finally, one more wrinkle: for unwinding exception handlers, it can be useful to Guile if it can determine whether an exception handler would indeed handle a particular exception or not. This is especially the case for exceptions raised in resource-exhaustion scenarios like `stack-overflow` or `out-of-memory`, where you want to immediately shrink resource use before recovering. See Section 6.26.3.4 [Stack Overflow], page 481. For this purpose, the `#:unwind-for-type` keyword argument allows users to specify the kind of exception handled by an exception handler; if `#t`, all exceptions will be handled; if an exception type object, only exceptions of that type will be handled; otherwise if a symbol, only that exceptions with the given `exception-kind` will be handled.

6.13.8.3 Throw and Catch

Guile only adopted `with-exception-handler` and `raise-exception` as its primary exception-handling facility in 2019. Before then, exception handling was fundamentally based on three other primitives with a somewhat more complex interface: `catch`, `with-throw-handler`, and `throw`.

`catch` *key thunk handler* [*pre-unwind-handler*] [Scheme Procedure]

`scm_catch_with_pre_unwind_handler` (*key, thunk, handler, pre_unwind_handler*) [C Function]

`scm_catch` (*key, thunk, handler*) [C Function]

Establish an exception handler during the dynamic extent of the call to *thunk*. *key* is either `#t`, indicating that all exceptions should be handled, or a symbol, restricting the exceptions handled to those having the *key* as their `exception-kind`.

If *thunk* executes normally, meaning without throwing any exceptions, the handler procedures are not called at all and the result of the `thunk` call is the result of the `catch`. Otherwise if an exception is thrown that matches *key*, *handler* is called with the continuation of the `catch` call.

Given the discussion from the previous section, it is most precise and concise to specify what `catch` does by expressing it in terms of `with-exception-handler`. Calling `catch` with the three arguments is the same as:

```
(define (catch key thunk handler)
```

```
(with-exception-handler
 (lambda (exn)
   (apply handler (exception-kind exn) (exception-args exn)))
 thunk
 #:unwind? #t
 #:unwind-for-type key))
```

By invoking `with-exception-handler` with `#:unwind? #t`, `catch` sets up an escape continuation that will be invoked in an exceptional situation before the handler is called.

If `catch` is called with four arguments, then the use of *thunk* should be replaced with:

```
(lambda ()
  (with-throw-handler key thunk pre-unwind-handler))
```

As can be seen above, if a pre-unwind-handler is passed to `catch`, it's like calling `with-throw-handler` inside the body thunk.

`with-throw-handler` is the second of the older primitives, and is used to be able to intercept an exception that is being thrown before the stack is unwound. This could be to clean up some related state, to print a backtrace, or to pass information about the exception to a debugger, for example.

`with-throw-handler` *key thunk handler* [Scheme Procedure]
`scm_with_throw_handler` (*key, thunk, handler*) [C Function]
Add *handler* to the dynamic context as a throw handler for key *key*, then invoke *thunk*.

It's not possible to exactly express `with-throw-handler` in terms of `with-exception-handler`, but we can get close.

```
(define (with-throw-handler key thunk handler)
  (with-exception-handler
   (lambda (exn)
     (when (or (eq? key #t) (eq? key (exception-kind exn)))
       (apply handler (exception-kind exn) (exception-args exn)))
     (raise-exception exn))
   thunk))
```

As you can see, unlike in the case of `catch`, the handler for `with-throw-handler` is invoked within the continuation of `raise-exception`, before unwinding the stack. If the throw handler returns normally, the exception will be re-raised, to be handled by the next exception handler.

The special wrinkle of `with-throw-handler` that can't be shown above is that if invoking the handler causes a `raise-exception` instead of completing normally, the exception is thrown in the *original* dynamic environment of the `raise-exception`. Any inner exception handler will get another shot at handling the exception. Here is an example to illustrate this behavior:

```
(catch 'a
  (lambda ()
    (with-throw-handler 'b
      (lambda ()
```

```
          (catch 'a
            (lambda ()
              (throw 'b))
            inner-handler))
        (lambda (key . args)
          (throw 'a))))
    outer-handler)
```

This code will call `inner-handler` and then continue with the continuation of the inner `catch`.

Finally, we get to `throw`, which is the older equivalent to `raise-exception`.

`throw` *key arg ...* [Scheme Procedure]
`scm_throw` (*key, args*) [C Function]
Raise an exception with kind *key* and arguments *args*. *key* is a symbol, denoting the "kind" of the exception.

Again, we can specify what `throw` does by expressing it in terms of `raise-exception`.

```
(define (throw key . args)
  (raise-exception (make-exception-from-throw key args)))
```

At this point, we should mention the primitive that manage the relationship between structured exception objects `throw`.

`make-exception-from-throw` *key args* [Scheme Procedure]
Create an exception object for the given *key* and *args* passed to `throw`. This may be a specific type of exception, for example `&programming-error`; Guile maintains a set of custom transformers for the various *key* values that have been used historically.

`exception-kind` *exn* [Scheme Procedure]
If *exn* is an exception created via `make-exception-from-throw`, return the corresponding *key* for the exception. Otherwise, unless *exn* is an exception of a type with a known mapping to `throw`, return the symbol `%exception`.

`exception-args` *exn* [Scheme Procedure]
If *exn* is an exception created via `make-exception-from-throw`, return the corresponding *args* for the exception. Otherwise, unless *exn* is an exception of a type with a known mapping to `throw`, return `(list exn)`.

6.13.8.4 Exceptions and C

There are some specific versions of Guile's original `catch` and `with-throw-handler` exception-handling primitives that are still widely used in C code.

`SCM scm_c_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data, scm_t_catch_handler pre_unwind_handler, void *pre_unwind_handler_data*) [C Function]

`SCM scm_internal_catch` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data*) [C Function]

The above `scm_catch_with_pre_unwind_handler` and `scm_catch` take Scheme procedures as body and handler arguments. `scm_c_catch` and `scm_internal_catch` are equivalents taking C functions.

body is called as `body (body_data)` with a catch on exceptions of the given *tag* type. If an exception is caught, *pre_unwind_handler* and *handler* are called as `handler (handler_data, key, args)`. *key* and *args* are the `SCM` key and argument list from the `throw`.

body and *handler* should have the following prototypes. `scm_t_catch_body` and `scm_t_catch_handler` are pointer typedefs for these.

```
SCM body (void *data);
SCM handler (void *data, SCM key, SCM args);
```

The *body_data* and *handler_data* parameters are passed to the respective calls so an application can communicate extra information to those functions.

If the data consists of an `SCM` object, care should be taken that it isn't garbage collected while still required. If the `SCM` is a local C variable, one way to protect it is to pass a pointer to that variable as the data parameter, since the C compiler will then know the value must be held on the stack. Another way is to use `scm_remember_upto_here_1` (see Section 5.5.4 [Foreign Object Memory Management], page 77).

`SCM scm_c_with_throw_handler` (*SCM tag, scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data, int lazy_catch_p*) [C Function]

The above `scm_with_throw_handler` takes Scheme procedures as body (thunk) and handler arguments. `scm_c_with_throw_handler` is an equivalent taking C functions. See `scm_c_catch` (see Section 6.13.8.4 [Exceptions and C], page 318) for a description of the parameters, the behaviour however of course follows `with-throw-handler`.

6.13.9 Procedures for Signaling Errors

Guile provides a set of convenience procedures for signaling error conditions that are implemented on top of the exception primitives just described.

`error` *msg arg ...* [Scheme Procedure]

Raise an error with key `misc-error` and a message constructed by displaying *msg* and writing *arg ...*.

`scm-error` *key subr message args data* [Scheme Procedure]

`scm_error_scm` (*key, subr, message, args, data*) [C Function]

Raise an error with key *key*. *subr* can be a string naming the procedure associated with the error, or `#f`. *message* is the error message string, possibly containing `~S` and `~A` escapes. When an error is reported, these are replaced by formatting the corresponding members of *args*: `~A` (was `%s` in older versions of Guile) formats using

`display` and `~S` (was `%S`) formats using `write`. *data* is a list or `#f` depending on *key*: if *key* is `system-error` then it should be a list containing the Unix `errno` value; If *key* is `signal` then it should be a list containing the Unix signal number; If *key* is `out-of-range`, `wrong-type-arg`, or `keyword-argument-error`, it is a list containing the bad value; otherwise it will usually be `#f`.

`strerror` *err* [Scheme Procedure]
`scm_strerror` (*err*) [C Function]

Return the Unix error message corresponding to *err*, an integer `errno` value.

When `setlocale` has been called (see Section 7.2.13 [Locales], page 547), the message is in the language and charset of `LC_MESSAGES`. (This is done by the C library.)

`false-if-exception` *expr* [syntax]

Returns the result of evaluating its argument; however if an exception occurs then `#f` is returned instead.

6.13.10 Dynamic Wind

For Scheme code, the fundamental procedure to react to non-local entry and exits of dynamic contexts is `dynamic-wind`. C code could use `scm_internal_dynamic_wind`, but since C does not allow the convenient construction of anonymous procedures that close over lexical variables, this will be, well, inconvenient.

Therefore, Guile offers the functions `scm_dynwind_begin` and `scm_dynwind_end` to delimit a dynamic extent. Within this dynamic extent, which is called a *dynwind context*, you can perform various *dynwind actions* that control what happens when the dynwind context is entered or left. For example, you can register a cleanup routine with `scm_dynwind_unwind_handler` that is executed when the context is left. There are several other more specialized dynwind actions as well, for example to temporarily block the execution of asyncs or to temporarily change the current output port. They are described elsewhere in this manual.

Here is an example that shows how to prevent memory leaks.

```
/* Suppose there is a function called FOO in some library that you
   would like to make available to Scheme code (or to C code that
   follows the Scheme conventions).

   FOO takes two C strings and returns a new string.  When an error has
   occurred in FOO, it returns NULL.
*/

char *foo (char *s1, char *s2);

/* SCM_FOO interfaces the C function FOO to the Scheme way of life.
   It takes care to free up all temporary strings in the case of
   non-local exits.
 */
```

```
SCM
scm_foo (SCM s1, SCM s2)
{
  char *c_s1, *c_s2, *c_res;

  scm_dynwind_begin (0);

  c_s1 = scm_to_locale_string (s1);

  /* Call 'free (c_s1)' when the dynwind context is left.
  */
  scm_dynwind_unwind_handler (free, c_s1, SCM_F_WIND_EXPLICITLY);

  c_s2 = scm_to_locale_string (s2);

  /* Same as above, but more concisely.
  */
  scm_dynwind_free (c_s2);

  c_res = foo (c_s1, c_s2);
  if (c_res == NULL)
    scm_report_out_of_memory ();

  scm_dynwind_end ();

  return scm_take_locale_string (res);
}
```

`dynamic-wind` *in_guard thunk out_guard* [Scheme Procedure]
`scm_dynamic_wind` (*in_guard, thunk, out_guard*) [C Function]

All three arguments must be 0-argument procedures. *in_guard* is called, then *thunk*, then *out_guard*.

If, any time during the execution of *thunk*, the dynamic extent of the `dynamic-wind` expression is escaped non-locally, *out_guard* is called. If the dynamic extent of the dynamic-wind is re-entered, *in_guard* is called. Thus *in_guard* and *out_guard* may be called any number of times.

```
(define x 'normal-binding)
⇒ x
(define a-cont
  (call-with-current-continuation
   (lambda (escape)
     (let ((old-x x))
       (dynamic-wind
           ;; in-guard:
           ;;
           (lambda () (set! x 'special-binding))
```

```
                ;; thunk
                ;;
                (lambda () (display x) (newline)
                           (call-with-current-continuation escape)
                           (display x) (newline)
                           x)

                ;; out-guard:
                ;;
                (lambda () (set! x old-x)))))))
          ;; Prints:
          special-binding
          ;; Evaluates to:
          ⇒ a-cont
          x
          ⇒ normal-binding
          (a-cont #f)
          ;; Prints:
          special-binding
          ;; Evaluates to:
          ⇒ a-cont  ;; the value of the (define a-cont...)
          x
          ⇒ normal-binding
          a-cont
          ⇒ special-binding
```

`scm_t_dynwind_flags` [C Type]

This is an enumeration of several flags that modify the behavior of `scm_dynwind_begin`. The flags are listed in the following table.

`SCM_F_DYNWIND_REWINDABLE`

The dynamic context is *rewindable*. This means that it can be reentered non-locally (via the invocation of a continuation). The default is that a dynwind context can not be reentered non-locally.

`void scm_dynwind_begin` (*scm_t_dynwind_flags flags*) [C Function]

The function `scm_dynwind_begin` starts a new dynamic context and makes it the 'current' one.

The *flags* argument determines the default behavior of the context. Normally, use 0. This will result in a context that can not be reentered with a captured continuation. When you are prepared to handle reentries, include `SCM_F_DYNWIND_REWINDABLE` in *flags*.

Being prepared for reentry means that the effects of unwind handlers can be undone on reentry. In the example above, we want to prevent a memory leak on non-local exit and thus register an unwind handler that frees the memory. But once the memory is freed, we can not get it back on reentry. Thus reentry can not be allowed.

The consequence is that continuations become less useful when non-reentrant contexts are captured, but you don't need to worry about that too much.

The context is ended either implicitly when a non-local exit happens, or explicitly with `scm_dynwind_end`. You must make sure that a dynwind context is indeed ended properly. If you fail to call `scm_dynwind_end` for each `scm_dynwind_begin`, the behavior is undefined.

`void scm_dynwind_end ()` [C Function]
End the current dynamic context explicitly and make the previous one current.

`scm_t_wind_flags` [C Type]
This is an enumeration of several flags that modify the behavior of `scm_dynwind_unwind_handler` and `scm_dynwind_rewind_handler`. The flags are listed in the following table.

`SCM_F_WIND_EXPLICITLY`
: The registered action is also carried out when the dynwind context is entered or left locally.

`void scm_dynwind_unwind_handler (`*void (*func)(void *), void *data, scm_t_wind_flags flags*`)` [C Function]
`void scm_dynwind_unwind_handler_with_scm (`*void (*func)(SCM), SCM data, scm_t_wind_flags flags*`)` [C Function]
Arranges for *func* to be called with *data* as its arguments when the current context ends implicitly. If *flags* contains `SCM_F_WIND_EXPLICITLY`, *func* is also called when the context ends explicitly with `scm_dynwind_end`.

The function `scm_dynwind_unwind_handler_with_scm` takes care that *data* is protected from garbage collection.

`void scm_dynwind_rewind_handler (`*void (*func)(void *), void *data, scm_t_wind_flags flags*`)` [C Function]
`void scm_dynwind_rewind_handler_with_scm (`*void (*func)(SCM), SCM data, scm_t_wind_flags flags*`)` [C Function]
Arrange for *func* to be called with *data* as its argument when the current context is restarted by rewinding the stack. When *flags* contains `SCM_F_WIND_EXPLICITLY`, *func* is called immediately as well.

The function `scm_dynwind_rewind_handler_with_scm` takes care that *data* is protected from garbage collection.

`void scm_dynwind_free (`*void *mem*`)` [C Function]
Arrange for *mem* to be freed automatically whenever the current context is exited, whether normally or non-locally. `scm_dynwind_free (mem)` is an equivalent shorthand for `scm_dynwind_unwind_handler (free, mem, SCM_F_WIND_EXPLICITLY)`.

6.13.11 Fluids and Dynamic States

A *fluid* is a variable whose value is associated with the dynamic extent of a function call. In the same way that an operating system runs a process with a given set of current input and output ports (or file descriptors), in Guile you can arrange to call a function while binding a fluid to a particular value. That association between fluid and value will exist during the dynamic extent of the function call.

Fluids are therefore a building block for implementing dynamically scoped variables. Dynamically scoped variables are useful when you want to set a variable to a value during some dynamic extent in the execution of your program and have them revert to their original value when the control flow is outside of this dynamic extent. See the description of `with-fluids` below for details. This association between fluids, values, and dynamic extents is robust to multiple entries (as when a captured continuation is invoked more than once) and early exits (for example, when throwing exceptions).

Guile uses fluids to implement parameters (see Section 6.13.12 [Parameters], page 326). Usually you just want to use parameters directly. However it can be useful to know what a fluid is and how it works, so that's what this section is about.

The current set of fluid-value associations can be captured in a *dynamic state* object. A dynamic extent is simply that: a snapshot of the current fluid-value associations. Guile users can capture the current dynamic state with `current-dynamic-state` and restore it later via `with-dynamic-state` or similar procedures. This facility is especially useful when implementing lightweight thread-like abstractions.

New fluids are created with `make-fluid` and `fluid?` is used for testing whether an object is actually a fluid. The values stored in a fluid can be accessed with `fluid-ref` and `fluid-set!`.

See Section 6.22.2 [Thread Local Variables], page 444, for further notes on fluids, threads, parameters, and dynamic states.

`make-fluid` [*dflt*] [Scheme Procedure]
`scm_make_fluid` () [C Function]
`scm_make_fluid_with_default` (*dflt*) [C Function]

Return a newly created fluid, whose initial value is *dflt*, or `#f` if *dflt* is not given. Fluids are objects that can hold one value per dynamic state. That is, modifications to this value are only visible to code that executes with the same dynamic state as the modifying code. When a new dynamic state is constructed, it inherits the values from its parent. Because each thread normally executes with its own dynamic state, you can use fluids for thread local storage.

`make-unbound-fluid` [Scheme Procedure]
`scm_make_unbound_fluid` () [C Function]

Return a new fluid that is initially unbound (instead of being implicitly bound to some definite value).

`fluid?` *obj* [Scheme Procedure]
`scm_fluid_p` (*obj*) [C Function]

Return `#t` if *obj* is a fluid; otherwise, return `#f`.

`fluid-ref` *fluid* [Scheme Procedure]
`scm_fluid_ref` (*fluid*) [C Function]

Return the value associated with *fluid* in the current dynamic root. If *fluid* has not been set, then return its default value. Calling `fluid-ref` on an unbound fluid produces a runtime error.

`fluid-set!` *fluid value* [Scheme Procedure]
`scm_fluid_set_x` (*fluid, value*) [C Function]

Set the value associated with *fluid* in the current dynamic root.

`fluid-ref*` *fluid depth* [Scheme Procedure]
`scm_fluid_ref_star` (*fluid, depth*) [C Function]
Return the *depth*th oldest value associated with *fluid* in the current thread. If *depth* equals or exceeds the number of values that have been assigned to *fluid*, return the default value of the fluid. `(fluid-ref* f 0)` is equivalent to `(fluid-ref f)`.

`fluid-ref*` is useful when you want to maintain a stack-like structure in a fluid, such as the stack of current exception handlers. Using `fluid-ref*` instead of an explicit stack allows any partial continuation captured by `call-with-prompt` to only capture the bindings made within the limits of the prompt instead of the entire continuation. See Section 6.13.5 [Prompts], page 303, for more on delimited continuations.

`fluid-unset!` *fluid* [Scheme Procedure]
`scm_fluid_unset_x` (*fluid*) [C Function]
Disassociate the given fluid from any value, making it unbound.

`fluid-bound?` *fluid* [Scheme Procedure]
`scm_fluid_bound_p` (*fluid*) [C Function]
Returns `#t` if the given fluid is bound to a value, otherwise `#f`.

`with-fluids*` temporarily changes the values of one or more fluids, so that the given procedure and each procedure called by it access the given values. After the procedure returns, the old values are restored.

`with-fluid*` *fluid value thunk* [Scheme Procedure]
`scm_with_fluid` (*fluid, value, thunk*) [C Function]
Set *fluid* to *value* temporarily, and call *thunk*. *thunk* must be a procedure with no argument.

`with-fluids*` *fluids values thunk* [Scheme Procedure]
`scm_with_fluids` (*fluids, values, thunk*) [C Function]
Set *fluids* to *values* temporary, and call *thunk*. *fluids* must be a list of fluids and *values* must be the same number of their values to be applied. Each substitution is done in the order given. *thunk* must be a procedure with no argument. It is called inside a `dynamic-wind` and the fluids are set/restored when control enter or leaves the established dynamic extent.

`with-fluids` ((*fluid value*) ...) *body1 body2 ...* [Scheme Macro]
Execute body *body1 body2 ...* while each *fluid* is set to the corresponding *value*. Both *fluid* and *value* are evaluated and *fluid* must yield a fluid. The body is executed inside a `dynamic-wind` and the fluids are set/restored when control enter or leaves the established dynamic extent.

`SCM scm_c_with_fluids` (*SCM fluids, SCM vals, SCM (*cproc)(void *), void *data*) [C Function]
`SCM scm_c_with_fluid` (*SCM fluid, SCM val, SCM (*cproc)(void *), void *data*) [C Function]
The function `scm_c_with_fluids` is like `scm_with_fluids` except that it takes a C function to call instead of a Scheme thunk.

The function `scm_c_with_fluid` is similar but only allows one fluid to be set instead of a list.

`void scm_dynwind_fluid` (*SCM fluid, SCM val*) [C Function]
This function must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (see Section 6.13.10 [Dynamic Wind], page 320). During the dynwind context, the fluid *fluid* is set to *val*.

More precisely, the value of the fluid is swapped with a 'backup' value whenever the dynwind context is entered or left. The backup value is initialized with the *val* argument.

`dynamic-state?` *obj* [Scheme Procedure]
`scm_dynamic_state_p` (*obj*) [C Function]
Return `#t` if *obj* is a dynamic state object; return `#f` otherwise.

`int scm_is_dynamic_state` (*SCM obj*) [C Procedure]
Return non-zero if *obj* is a dynamic state object; return zero otherwise.

`current-dynamic-state` [Scheme Procedure]
`scm_current_dynamic_state` () [C Function]
Return a snapshot of the current fluid-value associations as a fresh dynamic state object.

`set-current-dynamic-state` *state* [Scheme Procedure]
`scm_set_current_dynamic_state` (*state*) [C Function]
Restore the saved fluid-value associations from *state*, replacing the current fluid-value associations. Return the current fluid-value associatoins as a dynamic state object, as in `current-dynamic-state`.

`with-dynamic-state` *state proc* [Scheme Procedure]
`scm_with_dynamic_state` (*state, proc*) [C Function]
Call *proc* while the fluid bindings from *state* have been made current, saving the current fluid bindings. When control leaves the invocation of *proc*, restore the saved bindings, saving instead the fluid bindings from inside the call. If control later re-enters *proc*, restore those saved bindings, saving the current bindings, and so on.

`void scm_dynwind_current_dynamic_state` (*SCM state*) [C Procedure]
Set the current dynamic state to *state* for the current dynwind context. Like `with-dynamic-state`, but in terms of Guile's "dynwind" C API.

`void * scm_c_with_dynamic_state` (*SCM state, void *(*func)(void *), void *data*) [C Procedure]
Like `scm_with_dynamic_state`, but call *func* with *data*.

6.13.12 Parameters

Parameters are Guile's facility for dynamically bound variables.

On the most basic level, a parameter object is a procedure. Calling it with no arguments returns its value. Calling it with one argument sets the value.

```
(define my-param (make-parameter 123))
(my-param) ⇒ 123
(my-param 456)
```

```
(my-param) ⇒ 456
```

The `parameterize` special form establishes new locations for parameters, those new locations having effect within the dynamic extent of the `parameterize` body. Leaving restores the previous locations. Re-entering (through a saved continuation) will again use the new locations.

```
(parameterize ((my-param 789))
  (my-param)) ⇒ 789
(my-param) ⇒ 456
```

Parameters are like dynamically bound variables in other Lisp dialects. They allow an application to establish parameter settings (as the name suggests) just for the execution of a particular bit of code, restoring when done. Examples of such parameters might be case-sensitivity for a search, or a prompt for user input.

Global variables are not as good as parameter objects for this sort of thing. Changes to them are visible to all threads, but in Guile parameter object locations are per-thread, thereby truly limiting the effect of `parameterize` to just its dynamic execution.

Passing arguments to functions is thread-safe, but that soon becomes tedious when there's more than a few or when they need to pass down through several layers of calls before reaching the point they should affect. Introducing a new setting to existing code is often easier with a parameter object than adding arguments.

`make-parameter` *init* [*converter*] [Scheme Procedure]

Return a new parameter object, with initial value *init*.

If a *converter* is given, then a call `(`*converter* `val)` is made for each value set, its return is the value stored. Such a call is made for the *init* initial value too.

A *converter* allows values to be validated, or put into a canonical form. For example,

```
(define my-param (make-parameter 123
                   (lambda (val)
                     (if (not (number? val))
                         (error "must be a number"))
                     (inexact->exact val))))
(my-param 0.75)
(my-param) ⇒ 3/4
```

`parameterize` ((*param value*) . . .) *body1 body2* . . . [library syntax]

Establish a new dynamic scope with the given *param*s bound to new locations and set to the given *value*s. *body1 body2* . . . is evaluated in that environment. The value returned is that of last body form.

Each *param* is an expression which is evaluated to get the parameter object. Often this will just be the name of a variable holding the object, but it can be anything that evaluates to a parameter.

The *param* expressions and *value* expressions are all evaluated before establishing the new dynamic bindings, and they're evaluated in an unspecified order.

For example,

```
(define prompt (make-parameter "Type something: "))
(define (get-input)
```

```
  (display (prompt))
  ...)

(parameterize ((prompt "Type a number: "))
  (get-input)
  ...)
```

Parameter objects are implemented using fluids (see Section 6.13.11 [Fluids and Dynamic States], page 323), so each dynamic state has its own parameter locations. That includes the separate locations when outside any `parameterize` form. When a parameter is created it gets a separate initial location in each dynamic state, all initialized to the given *init* value.

New code should probably just use parameters instead of fluids, because the interface is better. But for migrating old code or otherwise providing interoperability, Guile provides the `fluid->parameter` procedure:

`fluid->parameter` *fluid* [*conv*] [Scheme Procedure]

Make a parameter that wraps a fluid.

The value of the parameter will be the same as the value of the fluid. If the parameter is rebound in some dynamic extent, perhaps via `parameterize`, the new value will be run through the optional *conv* procedure, as with any parameter. Note that unlike `make-parameter`, *conv* is not applied to the initial value.

As alluded to above, because each thread usually has a separate dynamic state, each thread has its own locations behind parameter objects, and changes in one thread are not visible to any other. When a new dynamic state or thread is created, the values of parameters in the originating context are copied, into new locations.

Guile's parameters conform to SRFI-39 (see Section 7.5.27 [SRFI-39], page 631).

6.13.13 How to Handle Errors

Guile is currently in a transition from its historical `catch` and `throw` error handling and signaling operators to the new structured exception facility; See Section 6.13.8 [Exceptions], page 311. However in the meantime, here is some documentation on errors and the older `catch` and `throw` interface.

Errors are always thrown with a *key* and four arguments:

- *key*: a symbol which indicates the type of error. The symbols used by libguile are listed below.
- *subr*: the name of the procedure from which the error is thrown, or `#f`.
- *message*: a string (possibly language and system dependent) describing the error. The tokens `~A` and `~S` can be embedded within the message: they will be replaced with members of the *args* list when the message is printed. `~A` indicates an argument printed using `display`, while `~S` indicates an argument printed using `write`. *message* can also be `#f`, to allow it to be derived from the *key* by the error handler (may be useful if the *key* is to be thrown from both C and Scheme).
- *args*: a list of arguments to be used to expand `~A` and `~S` tokens in *message*. Can also be `#f` if no arguments are required.

- *rest*: a list of any additional objects required. e.g., when the key is `'system-error`, this contains the C errno value. Can also be `#f` if no additional objects are required.

In addition to `catch` and `throw`, the following Scheme facilities are available:

`display-error` *frame port subr message args rest* [Scheme Procedure]
`scm_display_error` (*frame*, *port*, *subr*, *message*, *args*, *rest*) [C Function]
Display an error message to the output port *port*. *frame* is the frame in which the error occurred, *subr* is the name of the procedure in which the error occurred and *message* is the actual error message, which may contain formatting instructions. These will format the arguments in the list *args* accordingly. *rest* is currently ignored.

The following are the error keys defined by libguile and the situations in which they are used:

- `error-signal`: thrown after receiving an unhandled fatal signal such as SIGSEGV, SIGBUS, SIGFPE etc. The *rest* argument in the throw contains the coded signal number (at present this is not the same as the usual Unix signal number).
- `system-error`: thrown after the operating system indicates an error condition. The *rest* argument in the throw contains the errno value.
- `numerical-overflow`: numerical overflow.
- `out-of-range`: the arguments to a procedure do not fall within the accepted domain.
- `wrong-type-arg`: an argument to a procedure has the wrong type.
- `wrong-number-of-args`: a procedure was called with the wrong number of arguments.
- `memory-allocation-error`: memory allocation error.
- `stack-overflow`: stack overflow error.
- `regular-expression-syntax`: errors generated by the regular expression library.
- `misc-error`: other errors.

6.13.13.1 C Support

In the following C functions, *SUBR* and *MESSAGE* parameters can be `NULL` to give the effect of `#f` described above.

`SCM scm_error` (*SCM key*, *char *subr*, *char *message*, *SCM args*, *SCM rest*) [C Function]
Throw an error, as per `scm-error` (see Section 6.13.9 [Error Reporting], page 319).

`void scm_syserror` (*char *subr*) [C Function]
`void scm_syserror_msg` (*char *subr*, *char *message*, *SCM args*) [C Function]
Throw an error with key `system-error` and supply `errno` in the *rest* argument. For `scm_syserror` the message is generated using `strerror`.

Care should be taken that any code in between the failing operation and the call to these routines doesn't change `errno`.

`void scm_num_overflow` (*char *subr*) [C Function]
`void scm_out_of_range` (*char *subr*, *SCM bad_value*) [C Function]
`void scm_wrong_num_args` (*SCM proc*) [C Function]

`void scm_wrong_type_arg` (*char* *`subr`, *int* `argnum`, *SCM* `bad_value`) [C Function]

`void scm_wrong_type_arg_msg` (*char* *`subr`, *int* `argnum`, *SCM* `bad_value`, *const char* *`expected`) [C Function]

`void scm_misc_error` (*const char* *`subr`, *const char* *`message`, *SCM* `args`) [C Function]

Throw an error with the various keys described above.

In `scm_wrong_num_args`, *proc* should be a Scheme symbol which is the name of the procedure incorrectly invoked. The other routines take the name of the invoked procedure as a C string.

In `scm_wrong_type_arg_msg`, *expected* is a C string describing the type of argument that was expected.

In `scm_misc_error`, *message* is the error message string, possibly containing `simple-format` escapes (see Section 6.14.5 [Simple Output], page 338), and the corresponding arguments in the *args* list.

6.13.13.2 Signalling Type Errors

Every function visible at the Scheme level should aggressively check the types of its arguments, to avoid misinterpreting a value, and perhaps causing a segmentation fault. Guile provides some macros to make this easier.

`void SCM_ASSERT` (*int* `test`, *SCM* `obj`, *unsigned int* `position`, *const char* *`subr`) [Macro]

`void SCM_ASSERT_TYPE` (*int* `test`, *SCM* `obj`, *unsigned int* `position`, *const char* *`subr`, *const char* *`expected`) [Macro]

If *test* is zero, signal a "wrong type argument" error, attributed to the subroutine named *subr*, operating on the value *obj*, which is the *position*'th argument of *subr*.

In `SCM_ASSERT_TYPE`, *expected* is a C string describing the type of argument that was expected.

`int SCM_ARG1` [Macro]
`int SCM_ARG2` [Macro]
`int SCM_ARG3` [Macro]
`int SCM_ARG4` [Macro]
`int SCM_ARG5` [Macro]
`int SCM_ARG6` [Macro]
`int SCM_ARG7` [Macro]

One of the above values can be used for *position* to indicate the number of the argument of *subr* which is being checked. Alternatively, a positive integer number can be used, which allows to check arguments after the seventh. However, for parameter numbers up to seven it is preferable to use `SCM_ARGN` instead of the corresponding raw number, since it will make the code easier to understand.

`int SCM_ARGn` [Macro]

Passing a value of zero or `SCM_ARGn` for *position* allows to leave it unspecified which argument's type is incorrect. Again, `SCM_ARGn` should be preferred over a raw zero constant.

6.13.14 Continuation Barriers

The non-local flow of control caused by continuations might sometimes not be wanted. You can use `with-continuation-barrier` to erect fences that continuations can not pass.

`with-continuation-barrier` *proc* [Scheme Procedure]
`scm_with_continuation_barrier` (*proc*) [C Function]

Call *proc* and return its result. Do not allow the invocation of continuations that would leave or enter the dynamic extent of the call to `with-continuation-barrier`. Such an attempt causes an error to be signaled.

Throws (such as errors) that are not caught from within *proc* are caught by `with-continuation-barrier`. In that case, a short message is printed to the current error port and `#f` is returned.

Thus, `with-continuation-barrier` returns exactly once.

`void * scm_c_with_continuation_barrier` (*void *(*func) (void *), void *data*) [C Function]

Like `scm_with_continuation_barrier` but call *func* on *data*. When an error is caught, `NULL` is returned.

6.14 Input and Output

6.14.1 Ports

Ports are the way that Guile performs input and output. Guile can read in characters or bytes from an *input port*, or write them out to an *output port*. Some ports support both interfaces.

There are a number of different port types implemented in Guile. File ports provide input and output over files, as you might imagine. For example, we might display a string to a file like this:

```
(let ((port (open-output-file "foo.txt")))
  (display "Hello, world!\n" port)
  (close-port port))
```

There are also string ports, for taking input from a string, or collecting output to a string; bytevector ports, for doing the same but using a bytevector as a source or sink of data; and soft ports, for arranging to call Scheme functions to provide input or handle output. See Section 6.14.10 [Port Types], page 344.

Ports should be *closed* when they are not needed by calling `close-port` on them, as in the example above. This will make sure that any pending output is successfully written out to disk, in the case of a file port, or otherwise to whatever mutable store is backed by the port. Any error that occurs while writing out that buffered data would also be raised promptly at the `close-port`, and not later when the port is closed by the garbage collector. See Section 6.14.6 [Buffering], page 339, for more on buffered output.

Closing a port also releases any precious resource the file might have. Usually in Scheme a programmer doesn't have to clean up after their data structures (see Section 6.19 [Memory Management], page 404), but most systems have strict limits on how many files can be open, both on a per-process and a system-wide basis. A program that uses many files should take

care not to hit those limits. The same applies to similar system resources such as pipes and sockets.

Indeed for these reasons the above example is not the most idiomatic way to use ports. It is more common to acquire ports via procedures like `call-with-output-file`, which handle the `close-port` automatically:

```
(call-with-output-file "foo.txt"
  (lambda (port)
    (display "Hello, world!\n" port)))
```

Finally, all ports have associated input and output buffers, as appropriate. Buffering is a common strategy to limit the overhead of small reads and writes: without buffering, each character fetched from a file would involve at least one call into the kernel, and maybe more depending on the character and the encoding. Instead, Guile will batch reads and writes into internal buffers. However, sometimes you want to make output on a port show up immediately. See Section 6.14.6 [Buffering], page 339, for more on interfaces to control port buffering.

`port?` *x* [Scheme Procedure]
`scm_port_p` (*x*) [C Function]
Return a boolean indicating whether *x* is a port.

`input-port?` *x* [Scheme Procedure]
`scm_input_port_p` (*x*) [C Function]
Return `#t` if *x* is an input port, otherwise return `#f`. Any object satisfying this predicate also satisfies `port?`.

`output-port?` *x* [Scheme Procedure]
`scm_output_port_p` (*x*) [C Function]
Return `#t` if *x* is an output port, otherwise return `#f`. Any object satisfying this predicate also satisfies `port?`.

`close-port` *port* [Scheme Procedure]
`scm_close_port` (*port*) [C Function]
Close the specified port object. Return `#t` if it successfully closes a port or `#f` if it was already closed. An exception may be raised if an error occurs, for example when flushing buffered output. See Section 6.14.6 [Buffering], page 339, for more on buffered output. See Section 7.2.2 [Ports and File Descriptors], page 497, for a procedure which can close file descriptors.

`port-closed?` *port* [Scheme Procedure]
`scm_port_closed_p` (*port*) [C Function]
Return `#t` if *port* is closed or `#f` if it is open.

6.14.2 Binary I/O

Guile's ports are fundamentally binary in nature: at the lowest level, they work on bytes. This section describes Guile's core binary I/O operations. See Section 6.14.4 [Textual I/O], page 336, for input and output of strings and characters.

To use these routines, first include the binary I/O module:

```
(use-modules (ice-9 binary-ports))
```

Note that although this module's name suggests that binary ports are some different kind of port, that's not the case: all ports in Guile are both binary and textual ports.

`get-u8` *port* [Scheme Procedure]
`scm_get_u8` (*port*) [C Function]
Return an octet read from *port*, an input port, blocking as necessary, or the end-of-file object.

`lookahead-u8` *port* [Scheme Procedure]
`scm_lookahead_u8` (*port*) [C Function]
Like `get-u8` but does not update *port*'s position to point past the octet.

The end-of-file object is unlike any other kind of object: it's not a pair, a symbol, or anything else. To check if a value is the end-of-file object, use the `eof-object?` predicate.

`eof-object?` *x* [Scheme Procedure]
`scm_eof_object_p` (*x*) [C Function]
Return `#t` if *x* is an end-of-file object, or `#f` otherwise.

Note that unlike other procedures in this module, `eof-object?` is defined in the default environment.

`get-bytevector-n` *port count* [Scheme Procedure]
`scm_get_bytevector_n` (*port, count*) [C Function]
Read *count* octets from *port*, blocking as necessary and return a bytevector containing the octets read. If fewer bytes are available, a bytevector smaller than *count* is returned.

`get-bytevector-n!` *port bv start count* [Scheme Procedure]
`scm_get_bytevector_n_x` (*port, bv, start, count*) [C Function]
Read *count* bytes from *port* and store them in *bv* starting at index *start*. Return either the number of bytes actually read or the end-of-file object.

`get-bytevector-some` *port* [Scheme Procedure]
`scm_get_bytevector_some` (*port*) [C Function]
Read from *port*, blocking as necessary, until bytes are available or an end-of-file is reached. Return either the end-of-file object or a new bytevector containing some of the available bytes (at least one), and update the port position to point just past these bytes.

`get-bytevector-some!` *port bv start count* [Scheme Procedure]
`scm_get_bytevector_some_x` (*port, bv, start, count*) [C Function]
Read up to *count* bytes from *port*, blocking as necessary until at least one byte is available or an end-of-file is reached. Store them in *bv* starting at index *start*. Return the number of bytes actually read, or an end-of-file object.

`get-bytevector-all` *port* [Scheme Procedure]
`scm_get_bytevector_all` (*port*) [C Function]
Read from *port*, blocking as necessary, until the end-of-file is reached. Return either a new bytevector containing the data read or the end-of-file object (if no data were available).

`unget-bytevector` *port bv* [*start* [*count*]] [Scheme Procedure]
`scm_unget_bytevector` (*port, bv, start, count*) [C Function]
Place the contents of *bv* in *port*, optionally starting at index *start* and limiting to *count* octets, so that its bytes will be read from left-to-right as the next bytes from *port* during subsequent read operations. If called multiple times, the unread bytes will be read again in last-in first-out order.

To perform binary output on a port, use `put-u8` or `put-bytevector`.

`put-u8` *port octet* [Scheme Procedure]
`scm_put_u8` (*port, octet*) [C Function]
Write *octet*, an integer in the 0–255 range, to *port*, a binary output port.

`put-bytevector` *port bv* [*start* [*count*]] [Scheme Procedure]
`scm_put_bytevector` (*port, bv, start, count*) [C Function]
Write the contents of *bv* to *port*, optionally starting at index *start* and limiting to *count* octets.

6.14.3 Encoding

Textual input and output on Guile ports is layered on top of binary operations. To this end, each port has an associated character encoding that controls how bytes read from the port are converted to characters, and how characters written to the port are converted to bytes.

`port-encoding` *port* [Scheme Procedure]
`scm_port_encoding` (*port*) [C Function]
Returns, as a string, the character encoding that *port* uses to interpret its input and output.

`set-port-encoding!` *port enc* [Scheme Procedure]
`scm_set_port_encoding_x` (*port, enc*) [C Function]
Sets the character encoding that will be used to interpret I/O to *port*. *enc* is a string containing the name of an encoding. Valid encoding names are those defined by IANA (`http://www.iana.org/assignments/character-sets`), for example `"UTF-8"` or `"ISO-8859-1"`.

When ports are created, they are assigned an encoding. The usual process to determine the initial encoding for a port is to take the value of the `%default-port-encoding` fluid.

`%default-port-encoding` [Scheme Variable]
A fluid containing name of the encoding to be used by default for newly created ports (see Section 6.13.11 [Fluids and Dynamic States], page 323). As a special case, the value `#f` is equivalent to `"ISO-8859-1"`.

The `%default-port-encoding` itself defaults to the encoding appropriate for the current locale, if `setlocale` has been called. See Section 7.2.13 [Locales], page 547, for more on locales and when you might need to call `setlocale` explicitly.

Some port types have other ways of determining their initial locales. String ports, for example, default to the UTF-8 encoding, in order to be able to represent all characters

regardless of the current locale. File ports can optionally sniff their file for a `coding:` declaration; See Section 6.14.10.1 [File Ports], page 344. Binary ports might be initialized to the ISO-8859-1 encoding in which each codepoint between 0 and 255 corresponds to a byte with that value.

Currently, the ports only work with *non-modal* encodings. Most encodings are non-modal, meaning that the conversion of bytes to a string doesn't depend on its context: the same byte sequence will always return the same string. A couple of modal encodings are in common use, like ISO-2022-JP and ISO-2022-KR, and they are not yet supported.

Each port also has an associated conversion strategy, which determines what to do when a Guile character can't be converted to the port's encoded character representation for output. There are three possible strategies: to raise an error, to replace the character with a hex escape, or to replace the character with a substitute character. Port conversion strategies are also used when decoding characters from an input port.

`port-conversion-strategy` *port* [Scheme Procedure]
`scm_port_conversion_strategy` (*port*) [C Function]
Returns the behavior of the port when outputting a character that is not representable in the port's current encoding.

If *port* is `#f`, then the current default behavior will be returned. New ports will have this default behavior when they are created.

`set-port-conversion-strategy!` *port sym* [Scheme Procedure]
`scm_set_port_conversion_strategy_x` (*port*, *sym*) [C Function]
Sets the behavior of Guile when outputting a character that is not representable in the port's current encoding, or when Guile encounters a decoding error when trying to read a character. *sym* can be either `error`, `substitute`, or `escape`.

If *port* is an open port, the conversion error behavior is set for that port. If it is `#f`, it is set as the default behavior for any future ports that get created in this thread.

As with port encodings, there is a fluid which determines the initial conversion strategy for a port.

`%default-port-conversion-strategy` [Scheme Variable]
The fluid that defines the conversion strategy for newly created ports, and also for other conversion routines such as `scm_to_stringn`, `scm_from_stringn`, `string->pointer`, and `pointer->string`.

Its value must be one of the symbols described above, with the same semantics: `error`, `substitute`, or `escape`.

When Guile starts, its value is `substitute`.

Note that `(set-port-conversion-strategy! #f` *sym*`)` is equivalent to `(fluid-set! %default-port-conversion-strategy` *sym*`)`.

As mentioned above, for an output port there are three possible port conversion strategies. The `error` strategy will throw an error when a nonconvertible character is encountered. The `substitute` strategy will replace nonconvertible characters with a question mark (‘`?`’). Finally the `escape` strategy will print nonconvertible characters as a hex escape, using the

escaping that is recognized by Guile's string syntax. Note that if the port's encoding is a Unicode encoding, like `UTF-8`, then encoding errors are impossible.

For an input port, the `error` strategy will cause Guile to throw an error if it encounters an invalid encoding, such as might happen if you tried to read `ISO-8859-1` as `UTF-8`. The error is thrown before advancing the read position. The `substitute` strategy will replace the bad bytes with a U+FFFD replacement character, in accordance with Unicode recommendations. When reading from an input port, the `escape` strategy is treated as if it were `error`.

6.14.4 Textual I/O

This section describes Guile's core textual I/O operations on characters and strings. See Section 6.14.2 [Binary I/O], page 332, for input and output of bytes and bytevectors. See Section 6.14.3 [Encoding], page 334, for more on how characters relate to bytes. To read general S-expressions from ports, See Section 6.18.2 [Scheme Read], page 385. See Section 6.18.3 [Scheme Write], page 386, for interfaces that write generic Scheme datums.

To use these routines, first include the textual I/O module:

```
(use-modules (ice-9 textual-ports))
```

Note that although this module's name suggests that textual ports are some different kind of port, that's not the case: all ports in Guile are both binary and textual ports.

`get-char` *input-port* [Scheme Procedure]
: Reads from *input-port*, blocking as necessary, until a complete character is available from *input-port*, or until an end of file is reached.

  If a complete character is available before the next end of file, `get-char` returns that character and updates the input port to point past the character. If an end of file is reached before any character is read, `get-char` returns the end-of-file object.

`lookahead-char` *input-port* [Scheme Procedure]
: The `lookahead-char` procedure is like `get-char`, but it does not update *input-port* to point past the character.

In the same way that it's possible to "unget" a byte or bytes, it's possible to "unget" the bytes corresponding to an encoded character.

`unget-char` *port char* [Scheme Procedure]
: Place character *char* in *port* so that it will be read by the next read operation. If called multiple times, the unread characters will be read again in last-in first-out order.

`unget-string` *port str* [Scheme Procedure]
: Place the string *str* in *port* so that its characters will be read from left-to-right as the next characters from *port* during subsequent read operations. If called multiple times, the unread characters will be read again in last-in first-out order.

Reading in a character at a time can be inefficient. If it's possible to perform I/O over multiple characters at a time, via strings, that might be faster.

`get-string-n` *input-port count* [Scheme Procedure]

The `get-string-n` procedure reads from *input-port*, blocking as necessary, until *count* characters are available, or until an end of file is reached. *count* must be an exact, non-negative integer, representing the number of characters to be read.

If *count* characters are available before end of file, `get-string-n` returns a string consisting of those *count* characters. If fewer characters are available before an end of file, but one or more characters can be read, `get-string-n` returns a string containing those characters. In either case, the input port is updated to point just past the characters read. If no characters can be read before an end of file, the end-of-file object is returned.

`get-string-n!` *input-port string start count* [Scheme Procedure]

The `get-string-n!` procedure reads from *input-port* in the same manner as `get-string-n`. *start* and *count* must be exact, non-negative integer objects, with *count* representing the number of characters to be read. *string* must be a string with at least $*start* + *count*$ characters.

If *count* characters are available before an end of file, they are written into *string* starting at index *start*, and *count* is returned. If fewer characters are available before an end of file, but one or more can be read, those characters are written into *string* starting at index *start* and the number of characters actually read is returned as an exact integer object. If no characters can be read before an end of file, the end-of-file object is returned.

`get-string-all` *input-port* [Scheme Procedure]

Reads from *input-port* until an end of file, decoding characters in the same manner as `get-string-n` and `get-string-n!`.

If characters are available before the end of file, a string containing all the characters decoded from that data are returned. If no character precedes the end of file, the end-of-file object is returned.

`get-line` *input-port* [Scheme Procedure]

Reads from *input-port* up to and including the linefeed character or end of file, decoding characters in the same manner as `get-string-n` and `get-string-n!`.

If a linefeed character is read, a string containing all of the text up to (but not including) the linefeed character is returned, and the port is updated to point just past the linefeed character. If an end of file is encountered before any linefeed character is read, but some characters have been read and decoded as characters, a string containing those characters is returned. If an end of file is encountered before any characters are read, the end-of-file object is returned.

Finally, there are just two core procedures to write characters to a port.

`put-char` *port char* [Scheme Procedure]

Writes *char* to the port. The `put-char` procedure returns an unspecified value.

`put-string` *port string* [Scheme Procedure]
`put-string` *port string start* [Scheme Procedure]

`put-string` *port string start count* [Scheme Procedure]
Write the *count* characters of *string* starting at index *start* to the port.

start and *count* must be non-negative exact integer objects. *string* must have a length of at least *start* + *count*. *start* defaults to 0. *count* defaults to (`string` − `lengthstring`) − *start*$.

Calling `put-string` is equivalent in all respects to calling `put-char` on the relevant sequence of characters, except that it will attempt to write multiple characters to the port at a time, even if the port is unbuffered.

The `put-string` procedure returns an unspecified value.

Textual ports have a textual position associated with them: a line and a column. Reading in characters or writing them out advances the line and the column appropriately.

`port-column` *port* [Scheme Procedure]
`port-line` *port* [Scheme Procedure]
`scm_port_column` (*port*) [C Function]
`scm_port_line` (*port*) [C Function]
Return the current column number or line number of *port*.

Port lines and positions are represented as 0-origin integers, which is to say that the the first character of the first line is line 0, column 0. However, when you display a line number, for example in an error message, we recommend you add 1 to get 1-origin integers. This is because lines numbers traditionally start with 1, and that is what non-programmers will find most natural.

`set-port-column!` *port column* [Scheme Procedure]
`set-port-line!` *port line* [Scheme Procedure]
`scm_set_port_column_x` (*port, column*) [C Function]
`scm_set_port_line_x` (*port, line*) [C Function]
Set the current column or line number of *port*.

6.14.5 Simple Textual Output

Guile exports a simple formatted output function, `simple-format`. For a more capable formatted output facility, See Section 7.11 [Formatted Output], page 716.

`simple-format` *destination message . args* [Scheme Procedure]
`scm_simple_format` (*destination, message, args*) [C Function]
Write *message* to *destination*, defaulting to the current output port. *message* can contain `~A` and `~S` escapes. When printed, the escapes are replaced with corresponding members of *args*: `~A` formats using `display` and `~S` formats using `write`. If *destination* is `#t`, then use the current output port, if *destination* is `#f`, then return a string containing the formatted text. Does not add a trailing newline.

Somewhat confusingly, Guile binds the `format` identifier to `simple-format` at startup. Once `(ice-9 format)` loads, it actually replaces the core `format` binding, so depending on whether you or a module you use has loaded `(ice-9 format)`, you may be using the simple or the more capable version.

6.14.6 Buffering

Every port has associated input and output buffers. You can think of ports as being backed by some mutable store, and that store might be far away. For example, ports backed by file descriptors have to go all the way to the kernel to read and write their data. To avoid this round-trip cost, Guile usually reads in data from the mutable store in chunks, and then services small requests like `get-char` out of that intermediate buffer. Similarly, small writes like `write-char` first go to a buffer, and are sent to the store when the buffer is full (or when port is flushed). Buffered ports speed up your program by reducing the number of round-trips to the mutable store, and they do so in a way that is mostly transparent to the user.

There are two major ways, however, in which buffering affects program semantics. Building correct, performant programs requires understanding these situations.

The first case is in random-access read/write ports (see Section 6.14.7 [Random Access], page 340). These ports, usually backed by a file, logically operate over the same mutable store when both reading and writing. So, if you read a character, causing the buffer to fill, then write a character, the bytes you filled in your read buffer are now invalid. Every time you switch between reading and writing, Guile has to flush any pending buffer. If this happens frequently, the cost can be high. In that case you should reduce the amount that you buffer, in both directions. Similarly, Guile has to flush buffers before seeking. None of these considerations apply to sockets, which don't logically read from and write to the same mutable store, and are not seekable. Note also that sockets are unbuffered by default. See Section 7.2.11.4 [Network Sockets and Communication], page 540.

The second case is the more pernicious one. If you write data to a buffered port, it probably doesn't go out to the mutable store directly. (This "probably" introduces some indeterminism in your program: what goes to the store, and when, depends on how full the buffer is. It is something that the user needs to explicitly be aware of.) The data is written to the store later – when the buffer fills up due to another write, or when `force-output` is called, or when `close-port` is called, or when the program exits, or even when the garbage collector runs. The salient point is, *the errors are signalled then too*. Buffered writes defer error detection (and defer the side effects to the mutable store), perhaps indefinitely if the port type does not need to be closed at GC.

One common heuristic that works well for textual ports is to flush output when a newline (`\n`) is written. This *line buffering* mode is on by default for TTY ports. Most other ports are *block buffered*, meaning that once the output buffer reaches the block size, which depends on the port and its configuration, the output is flushed as a block, without regard to what is in the block. Likewise reads are read in at the block size, though if there are fewer bytes available to read, the buffer may not be entirely filled.

Note that binary reads or writes that are larger than the buffer size go directly to the mutable store without passing through the buffers. If your access pattern involves many big reads or writes, buffering might not matter so much to you.

To control the buffering behavior of a port, use `setvbuf`.

`setvbuf` *port mode* [*size*] [Scheme Procedure]
`scm_setvbuf` (*port, mode, size*) [C Function]

Set the buffering mode for *port*. *mode* can be one of the following symbols:

`none` non-buffered

`line` line buffered

`block` block buffered, using a newly allocated buffer of *size* bytes. If *size* is omitted, a default size will be used.

Another way to set the buffering, for file ports, is to open the file with `0` or `l` as part of the mode string, for unbuffered or line-buffered ports, respectively. See Section 6.14.10.1 [File Ports], page 344, for more.

Any buffered output data will be written out when the port is closed. To make sure to flush it at specific points in your program, use `force-otput`.

`force-output` [*port*] [Scheme Procedure]
`scm_force_output` (*port*) [C Function]
Flush the specified output port, or the current output port if *port* is omitted. The current output buffer contents, if any, are passed to the underlying port implementation.

The return value is unspecified.

`flush-all-ports` [Scheme Procedure]
`scm_flush_all_ports` () [C Function]
Equivalent to calling `force-output` on all open output ports. The return value is unspecified.

Similarly, sometimes you might want to switch from using Guile's ports to working directly on file descriptors. In that case, for input ports use `drain-input` to get any buffered input from that port.

`drain-input` *port* [Scheme Procedure]
`scm_drain_input` (*port*) [C Function]
This procedure clears a port's input buffers, similar to the way that force-output clears the output buffer. The contents of the buffers are returned as a single string, e.g.,

```
(define p (open-input-file ...))
(drain-input p) => empty string, nothing buffered yet.
(unread-char (read-char p) p)
(drain-input p) => initial chars from p, up to the buffer size.
```

All of these considerations are very similar to those of streams in the C library, although Guile's ports are not built on top of C streams. Still, it is useful to read what other systems do. See Section "Streams" in *The GNU C Library Reference Manual*, for more discussion on C streams.

6.14.7 Random Access

`seek` *fd_port offset whence* [Scheme Procedure]
`scm_seek` (*fd_port, offset, whence*) [C Function]
Sets the current position of *fd_port* to the integer *offset*. For a file port, *offset* is expressed as a number of bytes; for other types of ports, such as string ports, *offset*

is an abstract representation of the position within the port's data, not necessarily expressed as a number of bytes. *offset* is interpreted according to the value of *whence*.

One of the following variables should be supplied for *whence*:

`SEEK_SET` [Variable]
Seek from the beginning of the file.

`SEEK_CUR` [Variable]
Seek from the current position.

`SEEK_END` [Variable]
Seek from the end of the file.

If *fd_port* is a file descriptor, the underlying system call is `lseek`. *port* may be a string port.

The value returned is the new position in *fd_port*. This means that the current position of a port can be obtained using:

```
(seek port 0 SEEK_CUR)
```

`ftell` *fd_port* [Scheme Procedure]
`scm_ftell` (*fd_port*) [C Function]
Return an integer representing the current position of *fd_port*, measured from the beginning. Equivalent to:

```
(seek port 0 SEEK_CUR)
```

`truncate-file` *file* [*length*] [Scheme Procedure]
`scm_truncate_file` (*file*, *length*) [C Function]
Truncate *file* to *length* bytes. *file* can be a filename string, a port object, or an integer file descriptor. The return value is unspecified.

For a port or file descriptor *length* can be omitted, in which case the file is truncated at the current position (per `ftell` above).

On most systems a file can be extended by giving a length greater than the current size, but this is not mandatory in the POSIX standard.

6.14.8 Line Oriented and Delimited Text

The delimited-I/O module can be accessed with:

```
(use-modules (ice-9 rdelim))
```

It can be used to read or write lines of text, or read text delimited by a specified set of characters.

`read-line` [*port*] [*handle-delim*] [Scheme Procedure]
Return a line of text from *port* if specified, otherwise from the value returned by `(current-input-port)`. Under Unix, a line of text is terminated by the first end-of-line character or by end-of-file.

If *handle-delim* is specified, it should be one of the following symbols:

`trim` Discard the terminating delimiter. This is the default, but it will be impossible to tell whether the read terminated with a delimiter or end-of-file.

`concat` Append the terminating delimiter (if any) to the returned string.

`peek` Push the terminating delimiter (if any) back on to the port.

`split` Return a pair containing the string read from the port and the terminating delimiter or end-of-file object.

`read-line!` *buf* [*port*] [Scheme Procedure]
Read a line of text into the supplied string *buf* and return the number of characters added to *buf*. If *buf* is filled, then `#f` is returned. Read from *port* if specified, otherwise from the value returned by `(current-input-port)`.

`read-delimited` *delims* [*port*] [*handle-delim*] [Scheme Procedure]
Read text until one of the characters in the string *delims* is found or end-of-file is reached. Read from *port* if supplied, otherwise from the value returned by `(current-input-port)`. *handle-delim* takes the same values as described for `read-line`.

`read-delimited!` *delims buf* [*port*] [*handle-delim*] [*start*] [*end*] [Scheme Procedure]
Read text into the supplied string *buf*.

If a delimiter was found, return the number of characters written, except if *handle-delim* is `split`, in which case the return value is a pair, as noted above.

As a special case, if *port* was already at end-of-stream, the EOF object is returned. Also, if no characters were written because the buffer was full, `#f` is returned.

It's something of a wacky interface, to be honest.

`%read-delimited!` *delims str gobble* [*port* [*start* [*end*]]] [Scheme Procedure]
`scm_read_delimited_x` (*delims*, *str*, *gobble*, *port*, *start*, *end*) [C Function]
Read characters from *port* into *str* until one of the characters in the *delims* string is encountered. If *gobble* is true, discard the delimiter character; otherwise, leave it in the input stream for the next read. If *port* is not specified, use the value of `(current-input-port)`. If *start* or *end* are specified, store data only into the substring of *str* bounded by *start* and *end* (which default to the beginning and end of the string, respectively).

Return a pair consisting of the delimiter that terminated the string and the number of characters read. If reading stopped at the end of file, the delimiter returned is the *eof-object*; if the string was filled without encountering a delimiter, this value is `#f`.

`%read-line` [*port*] [Scheme Procedure]
`scm_read_line` (*port*) [C Function]
Read a newline-terminated line from *port*, allocating storage as necessary. The newline terminator (if any) is removed from the string, and a pair consisting of the line and its delimiter is returned. The delimiter may be either a newline or the *eof-object*; if `%read-line` is called at the end of file, it returns the pair `(#<eof> . #<eof>)`.

6.14.9 Default Ports for Input, Output and Errors

`current-input-port` [Scheme Procedure]
`scm_current_input_port ()` [C Function]

Return the current input port. This is the default port used by many input procedures.

Initially this is the *standard input* in Unix and C terminology. When the standard input is a tty the port is unbuffered, otherwise it's fully buffered.

Unbuffered input is good if an application runs an interactive subprocess, since any type-ahead input won't go into Guile's buffer and be unavailable to the subprocess.

Note that Guile buffering is completely separate from the tty "line discipline". In the usual cooked mode on a tty Guile only sees a line of input once the user presses `Return`.

`current-output-port` [Scheme Procedure]
`scm_current_output_port ()` [C Function]

Return the current output port. This is the default port used by many output procedures.

Initially this is the *standard output* in Unix and C terminology. When the standard output is a tty this port is unbuffered, otherwise it's fully buffered.

Unbuffered output to a tty is good for ensuring progress output or a prompt is seen. But an application which always prints whole lines could change to line buffered, or an application with a lot of output could go fully buffered and perhaps make explicit `force-output` calls (see Section 6.14.6 [Buffering], page 339) at selected points.

`current-error-port` [Scheme Procedure]
`scm_current_error_port ()` [C Function]

Return the port to which errors and warnings should be sent.

Initially this is the *standard error* in Unix and C terminology. When the standard error is a tty this port is unbuffered, otherwise it's fully buffered.

`set-current-input-port` *port* [Scheme Procedure]
`set-current-output-port` *port* [Scheme Procedure]
`set-current-error-port` *port* [Scheme Procedure]
`scm_set_current_input_port` (*port*) [C Function]
`scm_set_current_output_port` (*port*) [C Function]
`scm_set_current_error_port` (*port*) [C Function]

Change the ports returned by `current-input-port`, `current-output-port` and `current-error-port`, respectively, so that they use the supplied *port* for input or output.

`with-input-from-port` *port thunk* [Scheme Procedure]
`with-output-to-port` *port thunk* [Scheme Procedure]
`with-error-to-port` *port thunk* [Scheme Procedure]

Call *thunk* in a dynamic environment in which `current-input-port`, `current-output-port` or `current-error-port` is rebound to the given *port*.

`void scm_dynwind_current_input_port` (*SCM port*) [C Function]
`void scm_dynwind_current_output_port` (*SCM port*) [C Function]
`void scm_dynwind_current_error_port` (*SCM port*) [C Function]

These functions must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (see Section 6.13.10 [Dynamic Wind], page 320). During the dynwind context, the indicated port is set to *port*.

More precisely, the current port is swapped with a 'backup' value whenever the dynwind context is entered or left. The backup value is initialized with the *port* argument.

6.14.10 Types of Port

6.14.10.1 File Ports

The following procedures are used to open file ports. See also Section 7.2.2 [Ports and File Descriptors], page 497, for an interface to the Unix `open` system call.

All file access uses the "LFS" large file support functions when available, so files bigger than 2 Gbytes (2^31 bytes) can be read and written on a 32-bit system.

Most systems have limits on how many files can be open, so it's strongly recommended that file ports be closed explicitly when no longer required (see Section 6.14.1 [Ports], page 331).

`open-file` *filename mode* [*#:guess-encoding=#f*] [*#:encoding=#f*] [Scheme Procedure]
`scm_open_file_with_encoding` (*filename, mode, guess_encoding, encoding*) [C Function]
`scm_open_file` (*filename, mode*) [C Function]

Open the file whose name is *filename*, and return a port representing that file. The attributes of the port are determined by the *mode* string. The way in which this is interpreted is similar to C stdio. The first character must be one of the following:

'`r`' Open an existing file for input.

'`w`' Open a file for output, creating it if it doesn't already exist or removing its contents if it does.

'`a`' Open a file for output, creating it if it doesn't already exist. All writes to the port will go to the end of the file. The "append mode" can be turned off while the port is in use see Section 7.2.2 [Ports and File Descriptors], page 497,

The following additional characters can be appended:

'`+`' Open the port for both input and output. E.g., `r+`: open an existing file for both input and output.

'`0`' Create an "unbuffered" port. In this case input and output operations are passed directly to the underlying port implementation without additional buffering. This is likely to slow down I/O operations. The buffering mode can be changed while a port is in use (see Section 6.14.6 [Buffering], page 339).

'`l`' Add line-buffering to the port. The port output buffer will be automatically flushed whenever a newline character is written.

'`b`' Use binary mode, ensuring that each byte in the file will be read as one Scheme character.

To provide this property, the file will be opened with the 8-bit character encoding `"ISO-8859-1"`, ignoring the default port encoding. See Section 6.14.1 [Ports], page 331, for more information on port encodings.

Note that while it is possible to read and write binary data as characters or strings, it is usually better to treat bytes as octets, and byte sequences as bytevectors. See Section 6.14.2 [Binary I/O], page 332, for more.

This option had another historical meaning, for DOS compatibility: in the default (textual) mode, DOS reads a CR-LF sequence as one LF byte. The `b` flag prevents this from happening, adding `O_BINARY` to the underlying `open` call. Still, the flag is generally useful because of its port encoding ramifications.

Unless binary mode is requested, the character encoding of the new port is determined as follows: First, if *guess-encoding* is true, the `file-encoding` procedure is used to guess the encoding of the file (see Section 6.18.8 [Character Encoding of Source Files], page 395). If *guess-encoding* is false or if `file-encoding` fails, *encoding* is used unless it is also false. As a last resort, the default port encoding is used. See Section 6.14.1 [Ports], page 331, for more information on port encodings. It is an error to pass a non-false *guess-encoding* or *encoding* if binary mode is requested.

If a file cannot be opened with the access requested, `open-file` throws an exception.

`open-input-file` *filename* [*#:guess-encoding=#f*] [*#:encoding=#f*] [*#:binary=#f*] [Scheme Procedure]

Open *filename* for input. If *binary* is true, open the port in binary mode, otherwise use text mode. *encoding* and *guess-encoding* determine the character encoding as described above for `open-file`. Equivalent to

```
(open-file filename
           (if binary "rb" "r")
           #:guess-encoding guess-encoding
           #:encoding encoding)
```

`open-output-file` *filename* [*#:encoding=#f*] [*#:binary=#f*] [Scheme Procedure]

Open *filename* for output. If *binary* is true, open the port in binary mode, otherwise use text mode. *encoding* specifies the character encoding as described above for `open-file`. Equivalent to

```
(open-file filename
           (if binary "wb" "w")
           #:encoding encoding)
```

`call-with-input-file` *filename proc* [Scheme Procedure]
[*#:guess-encoding=#f*] [*#:encoding=#f*] [*#:binary=#f*]
`call-with-output-file` *filename proc* [*#:encoding=#f*] [Scheme Procedure]
[*#:binary=#f*]

Open *filename* for input or output, and call `(proc port)` with the resulting port. Return the value returned by *proc*. *filename* is opened as per `open-input-file` or `open-output-file` respectively, and an error is signaled if it cannot be opened.

When *proc* returns, the port is closed. If *proc* does not return (e.g. if it throws an error), then the port might not be closed automatically, though it will be garbage collected in the usual way if not otherwise referenced.

`with-input-from-file` *filename thunk* [Scheme Procedure]
[*#:guess-encoding=#f*] [*#:encoding=#f*] [*#:binary=#f*]
`with-output-to-file` *filename thunk* [*#:encoding=#f*] [Scheme Procedure]
[*#:binary=#f*]
`with-error-to-file` *filename thunk* [*#:encoding=#f*] [Scheme Procedure]
[*#:binary=#f*]

Open *filename* and call `(thunk)` with the new port setup as respectively the `current-input-port`, `current-output-port`, or `current-error-port`. Return the value returned by *thunk*. *filename* is opened as per `open-input-file` or `open-output-file` respectively, and an error is signaled if it cannot be opened.

When *thunk* returns, the port is closed and the previous setting of the respective current port is restored.

The current port setting is managed with `dynamic-wind`, so the previous value is restored no matter how *thunk* exits (eg. an exception), and if *thunk* is re-entered (via a captured continuation) then it's set again to the *filename* port.

The port is closed when *thunk* returns normally, but not when exited via an exception or new continuation. This ensures it's still ready for use if *thunk* is re-entered by a captured continuation. Of course the port is always garbage collected and closed in the usual way when no longer referenced anywhere.

`port-mode` *port* [Scheme Procedure]
`scm_port_mode` (*port*) [C Function]

Return the port modes associated with the open port *port*. These will not necessarily be identical to the modes used when the port was opened, since modes such as "append" which are used only during port creation are not retained.

`port-filename` *port* [Scheme Procedure]
`scm_port_filename` (*port*) [C Function]

Return the filename associated with *port*, or `#f` if no filename is associated with the port.

port must be open; `port-filename` cannot be used once the port is closed.

`set-port-filename!` *port filename* [Scheme Procedure]
`scm_set_port_filename_x` (*port, filename*) [C Function]

Change the filename associated with *port*, using the current input port if none is specified. Note that this does not change the port's source of data, but only the value that is returned by `port-filename` and reported in diagnostic output.

`file-port?` *obj* [Scheme Procedure]
`scm_file_port_p` (*obj*) [C Function]
Determine whether *obj* is a port that is related to a file.

6.14.10.2 Bytevector Ports

`open-bytevector-input-port` *bv* [*transcoder*] [Scheme Procedure]
`scm_open_bytevector_input_port` (*bv*, *transcoder*) [C Function]
Return an input port whose contents are drawn from bytevector *bv* (see Section 6.6.12 [Bytevectors], page 193).

The *transcoder* argument is currently not supported.

`open-bytevector-output-port` [*transcoder*] [Scheme Procedure]
`scm_open_bytevector_output_port` (*transcoder*) [C Function]
Return two values: a binary output port and a procedure. The latter should be called with zero arguments to obtain a bytevector containing the data accumulated by the port, as illustrated below.

```
(call-with-values
  (lambda ()
    (open-bytevector-output-port))
  (lambda (port get-bytevector)
    (display "hello" port)
    (get-bytevector)))

⇒ #vu8(104 101 108 108 111)
```

The *transcoder* argument is currently not supported.

6.14.10.3 String Ports

`call-with-output-string` *proc* [Scheme Procedure]
`scm_call_with_output_string` (*proc*) [C Function]
Calls the one-argument procedure *proc* with a newly created output port. When the function returns, the string composed of the characters written into the port is returned. *proc* should not close the port.

`call-with-input-string` *string proc* [Scheme Procedure]
`scm_call_with_input_string` (*string*, *proc*) [C Function]
Calls the one-argument procedure *proc* with a newly created input port from which *string*'s contents may be read. The value yielded by the *proc* is returned.

`with-output-to-string` *thunk* [Scheme Procedure]
Calls the zero-argument procedure *thunk* with the current output port set temporarily to a new string port. It returns a string composed of the characters written to the current output.

`with-input-from-string` *string thunk* [Scheme Procedure]
Calls the zero-argument procedure *thunk* with the current input port set temporarily to a string port opened on the specified *string*. The value yielded by *thunk* is returned.

`open-input-string` *str* [Scheme Procedure]
`scm_open_input_string` (*str*) [C Function]
Take a string and return an input port that delivers characters from the string. The port can be closed by `close-input-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

`open-output-string` [Scheme Procedure]
`scm_open_output_string` () [C Function]
Return an output port that will accumulate characters for retrieval by `get-output-string`. The port can be closed by the procedure `close-output-port`, though its storage will be reclaimed by the garbage collector if it becomes inaccessible.

`get-output-string` *port* [Scheme Procedure]
`scm_get_output_string` (*port*) [C Function]
Given an output port created by `open-output-string`, return a string consisting of the characters that have been output to the port so far.

`get-output-string` must be used before closing *port*, once closed the string cannot be obtained.

With string ports, the port-encoding is treated differently than other types of ports. When string ports are created, they do not inherit a character encoding from the current locale. They are given a default locale that allows them to handle all valid string characters. Typically one should not modify a string port's character encoding away from its default. See Section 6.14.3 [Encoding], page 334.

6.14.10.4 Custom Ports

Custom ports allow the user to provide input and handle output via user-supplied procedures. Guile currently only provides custom binary ports, not textual ports; for custom textual ports, See Section 6.14.10.5 [Soft Ports], page 350. We should add the R6RS custom textual port interfaces though. Contributions are appreciated.

`make-custom-binary-input-port` *id read! get-position set-position! close* [Scheme Procedure]
Return a new custom binary input port[8] named *id* (a string) whose input is drained by invoking *read!* and passing it a bytevector, an index where bytes should be written, and the number of bytes to read. The `read!` procedure must return an integer indicating the number of bytes read, or `0` to indicate the end-of-file.

Optionally, if *get-position* is not `#f`, it must be a thunk that will be called when `port-position` is invoked on the custom binary port and should return an integer indicating the position within the underlying data stream; if *get-position* was not supplied, the returned port does not support `port-position`.

Likewise, if *set-position!* is not `#f`, it should be a one-argument procedure. When `set-port-position!` is invoked on the custom binary input port, *set-position!* is passed an integer indicating the position of the next byte is to read.

Finally, if *close* is not `#f`, it must be a thunk. It is invoked when the custom binary input port is closed.

[8] This is similar in spirit to Guile's *soft ports* (see Section 6.14.10.5 [Soft Ports], page 350).

The returned port is fully buffered by default, but its buffering mode can be changed using `setvbuf` (see Section 6.14.6 [Buffering], page 339).

Using a custom binary input port, the `open-bytevector-input-port` procedure (see Section 6.14.10.2 [Bytevector Ports], page 347) could be implemented as follows:

```
(define (open-bytevector-input-port source)
  (define position 0)
  (define length (bytevector-length source))

  (define (read! bv start count)
    (let ((count (min count (- length position))))
      (bytevector-copy! source position
                        bv start count)
      (set! position (+ position count))
      count))

  (define (get-position) position)

  (define (set-position! new-position)
    (set! position new-position))

  (make-custom-binary-input-port "the port" read!
                                 get-position set-position!
                                 #f))

(read (open-bytevector-input-port (string->utf8 "hello")))
⇒ hello
```

`make-custom-binary-output-port` *id write! get-position set-position! close* [Scheme Procedure]

Return a new custom binary output port named *id* (a string) whose output is sunk by invoking *write!* and passing it a bytevector, an index where bytes should be read from this bytevector, and the number of bytes to be "written". The `write!` procedure must return an integer indicating the number of bytes actually written; when it is passed `0` as the number of bytes to write, it should behave as though an end-of-file was sent to the byte sink.

The other arguments are as for `make-custom-binary-input-port`.

`make-custom-binary-input/output-port` *id read! write! get-position set-position! close* [Scheme Procedure]

Return a new custom binary input/output port named *id* (a string). The various arguments are the same as for The other arguments are as for `make-custom-binary-input-port` and `make-custom-binary-output-port`. If buffering is enabled on the port, as is the case by default, input will be buffered in both directions; See Section 6.14.6 [Buffering], page 339. If the *set-position!* function is provided and not `#f`, then the port will also be marked as random-access, causing the buffer to be flushed between reads and writes.

6.14.10.5 Soft Ports

A *soft port* is a port based on a vector of procedures capable of accepting or delivering characters. It allows emulation of I/O ports.

`make-soft-port` *pv modes* [Scheme Procedure]

Return a port capable of receiving or delivering characters as specified by the *modes* string (see Section 6.14.10.1 [File Ports], page 344). *pv* must be a vector of length 5 or 6. Its components are as follows:

0. procedure accepting one character for output
1. procedure accepting a string for output
2. thunk for flushing output
3. thunk for getting one character
4. thunk for closing port (not by garbage collection)
5. (if present and not `#f`) thunk for computing the number of characters that can be read from the port without blocking.

For an output-only port only elements 0, 1, 2, and 4 need be procedures. For an input-only port only elements 3 and 4 need be procedures. Thunks 2 and 4 can instead be `#f` if there is no useful operation for them to perform.

If thunk 3 returns `#f` or an `eof-object` (see Section "Input" in *The Revised^5 Report on Scheme*) it indicates that the port has reached end-of-file. For example:

```
(define stdout (current-output-port))
(define p (make-soft-port
           (vector
            (lambda (c) (write c stdout))
            (lambda (s) (display s stdout))
            (lambda () (display "." stdout))
            (lambda () (char-upcase (read-char)))
            (lambda () (display "@" stdout)))
           "rw"))

(write p p) ⇒ #<input-output: soft 8081e20>
```

6.14.10.6 Void Ports

This kind of port causes any data to be discarded when written to, and always returns the end-of-file object when read from.

`%make-void-port` *mode* [Scheme Procedure]
`scm_sys_make_void_port` (*mode*) [C Function]

Create and return a new void port. A void port acts like `/dev/null`. The *mode* argument specifies the input/output modes for this port: see the documentation for `open-file` in Section 6.14.10.1 [File Ports], page 344.

6.14.11 Venerable Port Interfaces

Over the 25 years or so that Guile has been around, its port system has evolved, adding many useful features. At the same time there have been four major Scheme standards

released in those 25 years, which also evolve the common Scheme understanding of what a port interface should be. Alas, it would be too much to ask for all of these evolutionary branches to be consistent. Some of Guile's original interfaces don't mesh with the later Scheme standards, and yet Guile can't just drop old interfaces. Sadly as well, the R6RS and R7RS standards both part from a base of R5RS, but end up in different and somewhat incompatible designs.

Guile's approach is to pick a set of port primitives that make sense together. We document that set of primitives, design our internal interfaces around them, and recommend them to users. As the R6RS I/O system is the most capable standard that Scheme has yet produced in this domain, we mostly recommend that; `(ice-9 binary-ports)` and `(ice-9 textual-ports)` are wholly modelled on `(rnrs io ports)`. Guile does not wholly copy R6RS, however; See Section 7.6.1 [R6RS Incompatibilities], page 662.

At the same time, we have many venerable port interfaces, lore handed down to us from our hacker ancestors. Most of these interfaces even predate the expectation that Scheme should have modules, so they are present in the default environment. In Guile we support them as well and we have no plans to remove them, but again we don't recommend them for new users.

`char-ready?` [*port*] [Scheme Procedure]

Return `#t` if a character is ready on input *port* and return `#f` otherwise. If `char-ready?` returns `#t` then the next `read-char` operation on *port* is guaranteed not to hang. If *port* is a file port at end of file then `char-ready?` returns `#t`.

`char-ready?` exists to make it possible for a program to accept characters from interactive ports without getting stuck waiting for input. Any input editors associated with such ports must make sure that characters whose existence has been asserted by `char-ready?` cannot be rubbed out. If `char-ready?` were to return `#f` at end of file, a port at end of file would be indistinguishable from an interactive port that has no ready characters.

Note that `char-ready?` only works reliably for terminals and sockets with one-byte encodings. Under the hood it will return `#t` if the port has any input buffered, or if the file descriptor that backs the port polls as readable, indicating that Guile can fetch more bytes from the kernel. However being able to fetch one byte doesn't mean that a full character is available; See Section 6.14.3 [Encoding], page 334. Also, on many systems it's possible for a file descriptor to poll as readable, but then block when it comes time to read bytes. Note also that on Linux kernels, all file ports backed by files always poll as readable. For non-file ports, this procedure always returns `#t`, except for soft ports, which have a `char-ready?` handler. See Section 6.14.10.5 [Soft Ports], page 350.

In short, this is a legacy procedure whose semantics are hard to provide. However it is a useful check to see if any input is buffered. See Section 6.14.14 [Non-Blocking I/O], page 356.

`read-char` [*port*] [Scheme Procedure]

The same as `get-char`, except that *port* defaults to the current input port. See Section 6.14.4 [Textual I/O], page 336.

`peek-char` [*port*] [Scheme Procedure]
The same as `lookahead-char`, except that *port* defaults to the current input port. See Section 6.14.4 [Textual I/O], page 336.

`unread-char` *cobj* [*port*] [Scheme Procedure]
The same as `unget-char`, except that *port* defaults to the current input port, and the arguments are swapped. See Section 6.14.4 [Textual I/O], page 336.

`unread-string` *str port* [Scheme Procedure]
`scm_unread_string` (*str, port*) [C Function]
The same as `unget-string`, except that *port* defaults to the current input port, and the arguments are swapped. See Section 6.14.4 [Textual I/O], page 336.

`newline` [*port*] [Scheme Procedure]
Send a newline to *port*. If *port* is omitted, send to the current output port. Equivalent to `(put-char port #\newline)`.

`write-char` *chr* [*port*] [Scheme Procedure]
The same as `put-char`, except that *port* defaults to the current input port, and the arguments are swapped. See Section 6.14.4 [Textual I/O], page 336.

6.14.12 Using Ports from C

Guile's C interfaces provides some niceties for sending and receiving bytes and characters in a way that works better with C.

`size_t scm_c_read` (*SCM port, void *buffer, size_t size*) [C Function]
Read up to *size* bytes from *port* and store them in *buffer*. The return value is the number of bytes actually read, which can be less than *size* if end-of-file has been reached.

Note that as this is a binary input procedure, this function does not update `port-line` and `port-column` (see Section 6.14.4 [Textual I/O], page 336).

`void scm_c_write` (*SCM port, const void *buffer, size_t size*) [C Function]
Write *size* bytes at *buffer* to *port*.

Note that as this is a binary output procedure, this function does not update `port-line` and `port-column` (see Section 6.14.4 [Textual I/O], page 336).

`size_t scm_c_read_bytes` (*SCM port, SCM bv, size_t start, size_t count*) [C Function]
`void scm_c_write_bytes` (*SCM port, SCM bv, size_t start, size_t count*) [C Function]
Like `scm_c_read` and `scm_c_write`, but reading into or writing from the bytevector *bv*. *count* indicates the byte index at which to start in the bytevector, and the read or write will continue for *count* bytes.

`void scm_unget_bytes` (*const unsigned char *buf, size_t len, SCM port*) [C Function]
`void scm_unget_byte` (*int c, SCM port*) [C Function]
`void scm_ungetc` (*scm_t_wchar c, SCM port*) [C Function]
Like `unget-bytevector`, `unget-byte`, and `unget-char`, respectively. See Section 6.14.4 [Textual I/O], page 336.

`void scm_c_put_latin1_chars` (*SCM port, const scm_t_uint8 *buf, size_t len*) [C Function]

`void scm_c_put_utf32_chars` (*SCM port, const scm_t_uint32 *buf, size_t len*); [C Function]

Write a string to *port*. In the first case, the `scm_t_uint8*` buffer is a string in the latin-1 encoding. In the second, the `scm_t_uint32*` buffer is a string in the UTF-32 encoding. These routines will update the port's line and column.

6.14.13 Implementing New Port Types in C

This section describes how to implement a new port type in C. Although ports support many operations, as a data structure they present an opaque interface to the user. To the port implementor, you have two pieces of information to work with: the port type, and the port's "stream". The port type is an opaque pointer allocated when defining your port type. It is your key into the port API, and it helps you identify which ports are actually yours. The "stream" is a pointer you control, and which you set when you create a port. Get a stream from a port using the `SCM_STREAM` macro. Note that your port methods are only ever called with ports of your type.

A port type is created by calling `scm_make_port_type`. Once you have your port type, you can create ports with `scm_c_make_port`, or `scm_c_make_port_with_encoding`.

`scm_t_port_type* scm_make_port_type` (*char *name, size_t (*read) (SCM port, SCM dst, size_t start, size_t count), size_t (*write) (SCM port, SCM src, size_t start, size_t count)*) [Function]

Define a new port type. The *name*, *read* and *write* parameters are initial values for those port type fields, as described below. The other fields are initialized with default values and can be changed later.

`SCM scm_c_make_port_with_encoding` (*scm_t_port_type *type, unsigned long mode_bits, SCM encoding, SCM conversion_strategy, scm_t_bits stream*) [Function]

`SCM scm_c_make_port` (*scm_t_port_type *type, unsigned long mode_bits, scm_t_bits stream*) [Function]

Make a port with the given *type*. The *stream* indicates the private data associated with the port, which your port implementation may later retrieve with `SCM_STREAM`. The mode bits should include one or more of the flags `SCM_RDNG` or `SCM_WRTNG`, indicating that the port is an input and/or an output port, respectively. The mode bits may also include `SCM_BUF0` or `SCM_BUFLINE`, indicating that the port should be unbuffered or line-buffered, respectively. The default is that the port will be block-buffered. See Section 6.14.6 [Buffering], page 339.

As you would imagine, *encoding* and *conversion_strategy* specify the port's initial textual encoding and conversion strategy. Both are symbols. `scm_c_make_port` is the same as `scm_c_make_port_with_encoding`, except it uses the default port encoding and conversion strategy.

The port type has a number of associate procedures and properties which collectively implement the port's behavior. Creating a new port type mostly involves writing these procedures.

`name` A pointer to a NUL terminated string: the name of the port type. This property is initialized via the first argument to `scm_make_port_type`.

`read` A port's `read` implementation fills read buffers. It should copy bytes to the supplied bytevector `dst`, starting at offset `start` and continuing for `count` bytes, returning the number of bytes read.

`write` A port's `write` implementation flushes write buffers to the mutable store. It should write out bytes from the supplied bytevector `src`, starting at offset `start` and continuing for `count` bytes, and return the number of bytes that were written.

`read_wait_fd`
`write_wait_fd`
If a port's `read` or `write` function returns `(size_t) -1`, that indicates that reading or writing would block. In that case to preserve the illusion of a blocking read or write operation, Guile's C port run-time will `poll` on the file descriptor returned by either the port's `read_wait_fd` or `write_wait_fd` function. Set using

`void scm_set_port_read_wait_fd (`*scm_t_port_type *type, int (*wait_fd) (SCM port)*`)` [Function]

`void scm_set_port_write_wait_fd (`*scm_t_port_type *type, int (*wait_fd) (SCM port)*`)` [Function]

Only a port type which implements the `read_wait_fd` or `write_wait_fd` port methods can usefully return `(size_t) -1` from a read or write function. See Section 6.14.14 [Non-Blocking I/O], page 356, for more on non-blocking I/O in Guile.

`print` Called when `write` is called on the port, to print a port description. For example, for a file port it may produce something like: `#<input: /etc/passwd 3>`. Set using

`void scm_set_port_print (`*scm_t_port_type *type, int (*print) (SCM port, SCM dest_port, scm_print_state *pstate)*`)` [Function]

The first argument *port* is the port being printed, the second argument *dest_port* is where its description should go.

`close` Called when the port is closed. It should free any resources used by the port. Set using

`void scm_set_port_close (`*scm_t_port_type *type, void (*close) (SCM port)*`)` [Function]

By default, ports that are garbage collected just go away without closing. If your port type needs to release some external resource like a file descriptor, or needs to make sure that its internal buffers are flushed even if the port is collected while it was open, then mark the port type as needing a close on GC.

`void scm_set_port_needs_close_on_gc (`*scm_t_port_type *type, int needs_close_p*`)` [Function]

`seek` Set the current position of the port. Guile will flush read and/or write buffers before seeking, as appropriate.

`void scm_set_port_seek (`*scm_t_port_type *type, scm_t_off (*seek) (SCM port, scm_t_off offset, int whence)*`)` [Function]

`truncate` Truncate the port data to be specified length. Guile will flush buffers before hand, as appropriate. Set using

`void scm_set_port_truncate (`*scm_t_port_type *type, void (*truncate) (SCM port, scm_t_off length)*`)` [Function]

`random_access_p`

Determine whether this port is a random-access port.

Seeking on a random-access port with buffered input, or switching to writing after reading, will cause the buffered input to be discarded and Guile will seek the port back the buffered number of bytes. Likewise seeking on a random-access port with buffered output, or switching to reading after writing, will flush pending bytes with a call to the `write` procedure. See Section 6.14.6 [Buffering], page 339.

Indicate to Guile that your port needs this behavior by returning a nonzero value from your `random_access_p` function. The default implementation of this function returns nonzero if the port type supplies a seek implementation.

`void scm_set_port_random_access_p (`*scm_t_port_type *type, int (*random_access_p) (SCM port)*`);` [Function]

`get_natural_buffer_sizes`

Guile will internally attach buffers to ports. An input port always has a read buffer and an output port always has a write buffer. See Section 6.14.6 [Buffering], page 339. A port buffer consists of a bytevector, along with some cursors into that bytevector denoting where to get and put data.

Port implementations generally don't have to be concerned with buffering: a port type's `read` or `write` function will receive the buffer's bytevector as an argument, along with an offset and a length into that bytevector, and should then either fill or empty that bytevector. However in some cases, port implementations may be able to provide an appropriate default buffer size to Guile.

`void scm_set_port_get_natural_buffer_sizes (`*scm_t_port_type *type, void (*get_natural_buffer_sizes) (SCM, size_t *read_buf_size, size_t *write_buf_size)*`)` [Function]

Fill in *read_buf_size* and *write_buf_size* with an appropriate buffer size for this port, if one is known.

File ports implement a `get_natural_buffer_sizes` to let the operating system inform Guile about the appropriate buffer sizes for the particular file opened by the port.

Note that calls to all of these methods can proceed in parallel and concurrently and from any thread up until the point that the port is closed. The call to `close` will happen when no other method is running, and no method will be called after the `close` method is called. If your port implementation needs mutual exclusion to prevent concurrency, it is responsible for locking appropriately.

6.14.14 Non-Blocking I/O

Most ports in Guile are *blocking*: when you try to read a character from a port, Guile will block on the read until a character is ready, or end-of-stream is detected. Likewise whenever Guile goes to write (possibly buffered) data to an output port, Guile will block until all the data is written.

Interacting with ports in blocking mode is very convenient: you can write straightforward, sequential algorithms whose code flow reflects the flow of data. However, blocking I/O has two main limitations.

The first is that it's easy to get into a situation where code is waiting on data. Time spent waiting on data when code could be doing something else is wasteful and prevents your program from reaching its peak throughput. If you implement a web server that sequentially handles requests from clients, it's very easy for the server to end up waiting on a client to finish its HTTP request, or waiting on it to consume the response. The end result is that you are able to serve fewer requests per second than you'd like to serve.

The second limitation is related: a blocking parser over user-controlled input is a denial-of-service vulnerability. Indeed the so-called "slow loris" attack of the early 2010s was just that: an attack on common web servers that drip-fed HTTP requests, one character at a time. All it took was a handful of slow loris connections to occupy an entire web server.

In Guile we would like to preserve the ability to write straightforward blocking networking processes of all kinds, but under the hood to allow those processes to suspend their requests if they would block.

To do this, the first piece is to allow Guile ports to declare themselves as being nonblocking. This is currently supported only for file ports, which also includes sockets, terminals, or any other port that is backed by a file descriptor. To do that, we use an arcane UNIX incantation:

```
(let ((flags (fcntl socket F_GETFL)))
  (fcntl socket F_SETFL (logior O_NONBLOCK flags)))
```

Now the file descriptor is open in non-blocking mode. If Guile tries to read or write from this file and the read or write returns a result indicating that more data can only be had by doing a blocking read or write, Guile will block by polling on the socket's `read-wait-fd` or `write-wait-fd`, to preserve the illusion of a blocking read or write. See Section 6.14.13 [I/O Extensions], page 353, for more on those internal interfaces.

So far we have just reproduced the status quo: the file descriptor is non-blocking, but the operations on the port do block. To go farther, it would be nice if we could suspend the "thread" using delimited continuations, and only resume the thread once the file descriptor is readable or writable. (See Section 6.13.5 [Prompts], page 303).

But here we run into a difficulty. The ports code is implemented in C, which means that although we can suspend the computation to some outer prompt, we can't resume it because Guile can't resume delimited continuations that capture the C stack.

To overcome this difficulty we have created a compatible but entirely parallel implementation of port operations. To use this implementation, do the following:

```
(use-modules (ice-9 suspendable-ports))
(install-suspendable-ports!)
```

This will replace the core I/O primitives like `get-char` and `put-bytevector` with new versions that are exactly the same as the ones in the standard library, but with two differences. One is that when a read or a write would block, the suspendable port operations call out the value of the `current-read-waiter` or `current-write-waiter` parameter, as appropriate. See Section 6.13.12 [Parameters], page 326. The default read and write waiters do the same thing that the C read and write waiters do, which is to poll. User code can parameterize the waiters, though, enabling the computation to suspend and allow the program to process other I/O operations. Because the new suspendable ports implementation is written in Scheme, that suspended computation can resume again later when it is able to make progress. Success!

The other main difference is that because the new ports implementation is written in Scheme, it is slower than C, currently by a factor of 3 or 4, though it depends on many factors. For this reason we have to keep the C implementations as the default ones. One day when Guile's compiler is better, we can close this gap and have only one port operation implementation again.

Note that Guile does not currently include an implementation of the facility to suspend the current thread and schedule other threads in the meantime. Before adding such a thing, we want to make sure that we're providing the right primitives that can be used to build schedulers and other user-space concurrency patterns, and that the patterns that we settle on are the right patterns. In the meantime, have a look at 8sync (`https://gnu.org/software/8sync`) for a prototype of an asynchronous I/O and concurrency facility.

`install-suspendable-ports!` [Scheme Procedure]
: Replace the core ports implementation with suspendable ports, as described above. This will mutate the values of the bindings like `get-char`, `put-u8`, and so on in place.

`uninstall-suspendable-ports!` [Scheme Procedure]
: Restore the original core ports implementation, un-doing the effect of `install-suspendable-ports!`.

`current-read-waiter` [Scheme Parameter]
`current-write-waiter` [Scheme Parameter]
: Parameters whose values are procedures of one argument, called when a suspendable port operation would block on a port while reading or writing, respectively. The default values of these parameters do a blocking `poll` on the port's file descriptor. The procedures are passed the port in question as their one argument.

6.14.15 Handling of Unicode Byte Order Marks

This section documents the finer points of Guile's handling of Unicode byte order marks (BOMs). A byte order mark (U+FEFF) is typically found at the start of a UTF-16 or UTF-32 stream, to allow readers to reliably determine the byte order. Occasionally, a BOM is found at the start of a UTF-8 stream, but this is much less common and not generally recommended.

Guile attempts to handle BOMs automatically, and in accordance with the recommendations of the Unicode Standard, when the port encoding is set to `UTF-8`, `UTF-16`, or `UTF-32`. In brief, Guile automatically writes a BOM at the start of a UTF-16 or UTF-32 stream, and automatically consumes one from the start of a UTF-8, UTF-16, or UTF-32 stream.

As specified in the Unicode Standard, a BOM is only handled specially at the start of a stream, and only if the port encoding is set to `UTF-8`, `UTF-16` or `UTF-32`. If the port encoding is set to `UTF-16BE`, `UTF-16LE`, `UTF-32BE`, or `UTF-32LE`, then BOMs are *not* handled specially, and none of the special handling described in this section applies.

- To ensure that Guile will properly detect the byte order of a UTF-16 or UTF-32 stream, you must perform a textual read before any writes, seeks, or binary I/O. Guile will not attempt to read a BOM unless a read is explicitly requested at the start of the stream.
- If a textual write is performed before the first read, then an arbitrary byte order will be chosen. Currently, big endian is the default on all platforms, but that may change in the future. If you wish to explicitly control the byte order of an output stream, set the port encoding to `UTF-16BE`, `UTF-16LE`, `UTF-32BE`, or `UTF-32LE`, and explicitly write a BOM (`#\xFEFF`) if desired.
- If `set-port-encoding!` is called in the middle of a stream, Guile treats this as a new logical "start of stream" for purposes of BOM handling, and will forget about any BOMs that had previously been seen. Therefore, it may choose a different byte order than had been used previously. This is intended to support multiple logical text streams embedded within a larger binary stream.
- Binary I/O operations are not guaranteed to update Guile's notion of whether the port is at the "start of the stream", nor are they guaranteed to produce or consume BOMs.
- For ports that support seeking (e.g. normal files), the input and output streams are considered linked: if the user reads first, then a BOM will be consumed (if appropriate), but later writes will *not* produce a BOM. Similarly, if the user writes first, then later reads will *not* consume a BOM.
- For ports that are not random access (e.g. pipes, sockets, and terminals), the input and output streams are considered *independent* for purposes of BOM handling: the first read will consume a BOM (if appropriate), and the first write will *also* produce a BOM (if appropriate). However, the input and output streams will always use the same byte order.
- Seeks to the beginning of a file will set the "start of stream" flags. Therefore, a subsequent textual read or write will consume or produce a BOM. However, unlike `set-port-encoding!`, if a byte order had already been chosen for the port, it will remain in effect after a seek, and cannot be changed by the presence of a BOM. Seeks anywhere other than the beginning of a file clear the "start of stream" flags.

6.15 Regular Expressions

A *regular expression* (or *regexp*) is a pattern that describes a whole class of strings. A full description of regular expressions and their syntax is beyond the scope of this manual.

If your system does not include a POSIX regular expression library, and you have not linked Guile with a third-party regexp library such as Rx, these functions will not be available. You can tell whether your Guile installation includes regular expression support by checking whether `(provided? 'regex)` returns true.

The following regexp and string matching features are provided by the `(ice-9 regex)` module. Before using the described functions, you should load this module by executing `(use-modules (ice-9 regex))`.

6.15.1 Regexp Functions

By default, Guile supports POSIX extended regular expressions. That means that the characters '`(`', '`)`', '`+`' and '`?`' are special, and must be escaped if you wish to match the literal characters and there is no support for "non-greedy" variants of '`*`', '`+`' or '`?`'.

This regular expression interface was modeled after that implemented by SCSH, the Scheme Shell. It is intended to be upwardly compatible with SCSH regular expressions.

Zero bytes (`#\nul`) cannot be used in regex patterns or input strings, since the underlying C functions treat that as the end of string. If there's a zero byte an error is thrown.

Internally, patterns and input strings are converted to the current locale's encoding, and then passed to the C library's regular expression routines (see Section "Regular Expressions" in *The GNU C Library Reference Manual*). The returned match structures always point to characters in the strings, not to individual bytes, even in the case of multi-byte encodings.

`string-match` *pattern str* [*start*] [Scheme Procedure]

Compile the string *pattern* into a regular expression and compare it with *str*. The optional numeric argument *start* specifies the position of *str* at which to begin matching.

`string-match` returns a *match structure* which describes what, if anything, was matched by the regular expression. See Section 6.15.2 [Match Structures], page 363. If *str* does not match *pattern* at all, `string-match` returns `#f`.

Two examples of a match follow. In the first example, the pattern matches the four digits in the match string. In the second, the pattern matches nothing.

```
(string-match "[0-9][0-9][0-9][0-9]" "blah2002")
⇒ #("blah2002" (4 . 8))

(string-match "[A-Za-z]" "123456")
⇒ #f
```

Each time `string-match` is called, it must compile its *pattern* argument into a regular expression structure. This operation is expensive, which makes `string-match` inefficient if the same regular expression is used several times (for example, in a loop). For better performance, you can compile a regular expression in advance and then match strings against the compiled regexp.

`make-regexp` *pat flag...* [Scheme Procedure]
`scm_make_regexp` (*pat*, *flaglst*) [C Function]

Compile the regular expression described by *pat*, and return the compiled regexp structure. If *pat* does not describe a legal regular expression, `make-regexp` throws a `regular-expression-syntax` error.

The *flag* arguments change the behavior of the compiled regular expression. The following values may be supplied:

`regexp/icase` [Variable]

Consider uppercase and lowercase letters to be the same when matching.

`regexp/newline` [Variable]
: If a newline appears in the target string, then permit the '`^`' and '`$`' operators to match immediately after or immediately before the newline, respectively. Also, the '`.`' and '`[^...]`' operators will never match a newline character. The intent of this flag is to treat the target string as a buffer containing many lines of text, and the regular expression as a pattern that may match a single one of those lines.

`regexp/basic` [Variable]
: Compile a basic ("obsolete") regexp instead of the extended ("modern") regexps that are the default. Basic regexps do not consider '`|`', '`+`' or '`?`' to be special characters, and require the '`{...}`' and '`(...)`' metacharacters to be backslash-escaped (see Section 6.15.3 [Backslash Escapes], page 364). There are several other differences between basic and extended regular expressions, but these are the most significant.

`regexp/extended` [Variable]
: Compile an extended regular expression rather than a basic regexp. This is the default behavior; this flag will not usually be needed. If a call to `make-regexp` includes both `regexp/basic` and `regexp/extended` flags, the one which comes last will override the earlier one.

`regexp-exec` *rx str* [*start* [*flags*]] [Scheme Procedure]
`scm_regexp_exec` (*rx, str, start, flags*) [C Function]
: Match the compiled regular expression *rx* against `str`. If the optional integer *start* argument is provided, begin matching from that position in the string. Return a match structure describing the results of the match, or `#f` if no match could be found.

  The *flags* argument changes the matching behavior. The following flag values may be supplied, use `logior` (see Section 6.6.2.13 [Bitwise Operations], page 125) to combine them,

  `regexp/notbol` [Variable]
  : Consider that the *start* offset into *str* is not the beginning of a line and should not match operator '`^`'.

    If *rx* was created with the `regexp/newline` option above, '`^`' will still match after a newline in *str*.

  `regexp/noteol` [Variable]
  : Consider that the end of *str* is not the end of a line and should not match operator '`$`'.

    If *rx* was created with the `regexp/newline` option above, '`$`' will still match before a newline in *str*.

```
;; Regexp to match uppercase letters
(define r (make-regexp "[A-Z]*"))

;; Regexp to match letters, ignoring case
```

```
(define ri (make-regexp "[A-Z]*" regexp/icase))

;; Search for bob using regexp r
(match:substring (regexp-exec r "bob"))
⇒ ""                    ; no match

;; Search for bob using regexp ri
(match:substring (regexp-exec ri "Bob"))
⇒ "Bob"                 ; matched case insensitive
```

`regexp?` *obj* [Scheme Procedure]
`scm_regexp_p` (*obj*) [C Function]

Return `#t` if *obj* is a compiled regular expression, or `#f` otherwise.

`list-matches` *regexp str* [*flags*] [Scheme Procedure]

Return a list of match structures which are the non-overlapping matches of *regexp* in *str*. *regexp* can be either a pattern string or a compiled regexp. The *flags* argument is as per `regexp-exec` above.

```
(map match:substring (list-matches "[a-z]+" "abc 42 def 78"))
⇒ ("abc" "def")
```

`fold-matches` *regexp str init proc* [*flags*] [Scheme Procedure]

Apply *proc* to the non-overlapping matches of *regexp* in *str*, to build a result. *regexp* can be either a pattern string or a compiled regexp. The *flags* argument is as per `regexp-exec` above.

proc is called as `(`*proc* `match prev)` where *match* is a match structure and *prev* is the previous return from *proc*. For the first call *prev* is the given *init* parameter. `fold-matches` returns the final value from *proc*.

For example to count matches,

```
(fold-matches "[a-z][0-9]" "abc x1 def y2" 0
              (lambda (match count)
                (1+ count)))
⇒ 2
```

Regular expressions are commonly used to find patterns in one string and replace them with the contents of another string. The following functions are convenient ways to do this.

`regexp-substitute` *port match item ...* [Scheme Procedure]

Write to *port* selected parts of the match structure *match*. Or if *port* is `#f` then form a string from those parts and return that.

Each *item* specifies a part to be written, and may be one of the following,

- A string. String arguments are written out verbatim.
- An integer. The submatch with that number is written (`match:substring`). Zero is the entire match.

- The symbol '`pre`'. The portion of the matched string preceding the regexp match is written (`match:prefix`).
- The symbol '`post`'. The portion of the matched string following the regexp match is written (`match:suffix`).

For example, changing a match and retaining the text before and after,

```
(regexp-substitute #f (string-match "[0-9]+" "number 25 is good")
                   'pre "37" 'post)
⇒ "number 37 is good"
```

Or matching a YYYYMMDD format date such as '20020828' and re-ordering and hyphenating the fields.

```
(define date-regex
   "([0-9][0-9][0-9][0-9])([0-9][0-9])([0-9][0-9])")
(define s "Date 20020429 12am.")
(regexp-substitute #f (string-match date-regex s)
                   'pre 2 "-" 3 "-" 1 'post " (" 0 ")")
⇒ "Date 04-29-2002 12am. (20020429)"
```

`regexp-substitute/global` *port regexp target item...* [Scheme Procedure]

Write to *port* selected parts of matches of *regexp* in *target*. If *port* is `#f` then form a string from those parts and return that. *regexp* can be a string or a compiled regex.

This is similar to `regexp-substitute`, but allows global substitutions on *target*. Each *item* behaves as per `regexp-substitute`, with the following differences,

- A function. Called as (*item* `match`) with the match structure for the *regexp* match, it should return a string to be written to *port*.
- The symbol '`post`'. This doesn't output anything, but instead causes `regexp-substitute/global` to recurse on the unmatched portion of *target*.

  This *must* be supplied to perform a global search and replace on *target*; without it `regexp-substitute/global` returns after a single match and output.

For example, to collapse runs of tabs and spaces to a single hyphen each,

```
(regexp-substitute/global #f "[ \t]+"  "this   is   the text"
                          'pre "-" 'post)
⇒ "this-is-the-text"
```

Or using a function to reverse the letters in each word,

```
(regexp-substitute/global #f "[a-z]+"  "to do and not-do"
  'pre (lambda (m) (string-reverse (match:substring m))) 'post)
⇒ "ot od dna ton-od"
```

Without the `post` symbol, just one regexp match is made. For example the following is the date example from `regexp-substitute` above, without the need for the separate `string-match` call.

```
(define date-regex
   "([0-9][0-9][0-9][0-9])([0-9][0-9])([0-9][0-9])")
(define s "Date 20020429 12am.")
(regexp-substitute/global #f date-regex s
```

```
                                    'pre 2 "-" 3 "-" 1 'post " (" 0 ")")

     ⇒ "Date 04-29-2002 12am. (20020429)"
```

6.15.2 Match Structures

A *match structure* is the object returned by `string-match` and `regexp-exec`. It describes which portion of a string, if any, matched the given regular expression. Match structures include: a reference to the string that was checked for matches; the starting and ending positions of the regexp match; and, if the regexp included any parenthesized subexpressions, the starting and ending positions of each submatch.

In each of the regexp match functions described below, the `match` argument must be a match structure returned by a previous call to `string-match` or `regexp-exec`. Most of these functions return some information about the original target string that was matched against a regular expression; we will call that string *target* for easy reference.

`regexp-match?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a match structure returned by a previous call to `regexp-exec`, or `#f` otherwise.

`match:substring` *match* [*n*] [Scheme Procedure]

Return the portion of *target* matched by subexpression number *n*. Submatch 0 (the default) represents the entire regexp match. If the regular expression as a whole matched, but the subexpression number *n* did not match, return `#f`.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:substring s)
⇒ "2002"

;; match starting at offset 6 in the string
(match:substring
  (string-match "[0-9][0-9][0-9][0-9]" "blah987654" 6))
⇒ "7654"
```

`match:start` *match* [*n*] [Scheme Procedure]

Return the starting position of submatch number *n*.

In the following example, the result is 4, since the match starts at character index 4:

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:start s)
⇒ 4
```

`match:end` *match* [*n*] [Scheme Procedure]

Return the ending position of submatch number *n*.

In the following example, the result is 8, since the match runs between characters 4 and 8 (i.e. the "2002").

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:end s)
⇒ 8
```

`match:prefix` *match* [Scheme Procedure]
Return the unmatched portion of *target* preceding the regexp match.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:prefix s)
⇒ "blah"
```

`match:suffix` *match* [Scheme Procedure]
Return the unmatched portion of *target* following the regexp match.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:suffix s)
⇒ "foo"
```

`match:count` *match* [Scheme Procedure]
Return the number of parenthesized subexpressions from *match*. Note that the entire regular expression match itself counts as a subexpression, and failed submatches are included in the count.

`match:string` *match* [Scheme Procedure]
Return the original *target* string.

```
(define s (string-match "[0-9][0-9][0-9][0-9]" "blah2002foo"))
(match:string s)
⇒ "blah2002foo"
```

6.15.3 Backslash Escapes

Sometimes you will want a regexp to match characters like '`*`' or '`$`' exactly. For example, to check whether a particular string represents a menu entry from an Info node, it would be useful to match it against a regexp like '`^* [^:]*::`'. However, this won't work; because the asterisk is a metacharacter, it won't match the '`*`' at the beginning of the string. In this case, we want to make the first asterisk un-magic.

You can do this by preceding the metacharacter with a backslash character '`\`'. (This is also called *quoting* the metacharacter, and is known as a *backslash escape*.) When Guile sees a backslash in a regular expression, it considers the following glyph to be an ordinary character, no matter what special meaning it would ordinarily have. Therefore, we can make the above example work by changing the regexp to '`^\* [^:]*::`'. The '`\*`' sequence tells the regular expression engine to match only a single asterisk in the target string.

Since the backslash is itself a metacharacter, you may force a regexp to match a backslash in the target string by preceding the backslash with itself. For example, to find variable references in a TEX program, you might want to find occurrences of the string '`\let\`' followed by any number of alphabetic characters. The regular expression '`\\let\\[A-Za-z]*`' would do this: the double backslashes in the regexp each match a single backslash in the target string.

`regexp-quote` *str* [Scheme Procedure]
Quote each special character found in *str* with a backslash, and return the resulting string.

Very important: Using backslash escapes in Guile source code (as in Emacs Lisp or C) can be tricky, because the backslash character has special meaning for the Guile reader. For example, if Guile encounters the character sequence '`\n`' in the middle of a string while processing Scheme code, it replaces those characters with a newline character. Similarly, the character sequence '`\t`' is replaced by a horizontal tab. Several of these *escape sequences* are processed by the Guile reader before your code is executed. Unrecognized escape sequences are ignored: if the characters '`\*`' appear in a string, they will be translated to the single character '`*`'.

This translation is obviously undesirable for regular expressions, since we want to be able to include backslashes in a string in order to escape regexp metacharacters. Therefore, to make sure that a backslash is preserved in a string in your Guile program, you must use *two* consecutive backslashes:

```
(define Info-menu-entry-pattern (make-regexp "^\\* [^:]*"))
```

The string in this example is preprocessed by the Guile reader before any code is executed. The resulting argument to `make-regexp` is the string '`^\* [^:]*`', which is what we really want.

This also means that in order to write a regular expression that matches a single backslash character, the regular expression string in the source code must include *four* backslashes. Each consecutive pair of backslashes gets translated by the Guile reader to a single backslash, and the resulting double-backslash is interpreted by the regexp engine as matching a single backslash character. Hence:

```
(define tex-variable-pattern (make-regexp "\\\\let\\\\=[A-Za-z]*"))
```

The reason for the unwieldiness of this syntax is historical. Both regular expression pattern matchers and Unix string processing systems have traditionally used backslashes with the special meanings described above. The POSIX regular expression specification and ANSI C standard both require these semantics. Attempting to abandon either convention would cause other kinds of compatibility problems, possibly more severe ones. Therefore, without extending the Scheme reader to support strings with different quoting conventions (an ungainly and confusing extension when implemented in other languages), we must adhere to this cumbersome escape syntax.

6.16 LALR(1) Parsing

The `(system base lalr)` module provides the `lalr-scm` LALR(1) parser generator by Dominique Boucher (`https://github.com/schemeway/lalr-scm/`). `lalr-scm` uses the same algorithm as GNU Bison (see Section "Introduction" in *Bison, The Yacc-compatible Parser Generator*). Parsers are defined using the `lalr-parser` macro.

`lalr-parser` [*options*] *tokens rules...* [Scheme Syntax]

Generate an LALR(1) syntax analyzer. *tokens* is a list of symbols representing the terminal symbols of the grammar. *rules* are the grammar production rules.

Each rule has the form `(`*non-terminal* `(`*rhs* `...) :` *action* `...)`, where *non-terminal* is the name of the rule, *rhs* are the right-hand sides, i.e., the production rule, and *action* is a semantic action associated with the rule.

The generated parser is a two-argument procedure that takes a *tokenizer* and a *syntax error procedure*. The tokenizer should be a thunk that returns lexical tokens as

produced by `make-lexical-token`. The syntax error procedure may be called with at least an error message (a string), and optionally the lexical token that caused the error.

Please refer to the `lalr-scm` documentation for details.

6.17 PEG Parsing

Parsing Expression Grammars (PEGs) are a way of specifying formal languages for text processing. They can be used either for matching (like regular expressions) or for building recursive descent parsers (like lex/yacc). Guile uses a superset of PEG syntax that allows more control over what information is preserved during parsing.

Wikipedia has a clear and concise introduction to PEGs if you want to familiarize yourself with the syntax: `http://en.wikipedia.org/wiki/Parsing_expression_grammar`.

The `(ice-9 peg)` module works by compiling PEGs down to lambda expressions. These can either be stored in variables at compile-time by the define macros (`define-peg-pattern` and `define-peg-string-patterns`) or calculated explicitly at runtime with the compile functions (`compile-peg-pattern` and `peg-string-compile`).

They can then be used for either parsing (`match-pattern`) or searching (`search-for-pattern`). For convenience, `search-for-pattern` also takes pattern literals in case you want to inline a simple search (people often use regular expressions this way).

The rest of this documentation consists of a syntax reference, an API reference, and a tutorial.

6.17.1 PEG Syntax Reference

Normal PEG Syntax:

`sequence` *a b* [PEG Pattern]

Parses *a*. If this succeeds, continues to parse *b* from the end of the text parsed as *a*. Succeeds if both *a* and *b* succeed.

```
"a b"
```

```
(and a b)
```

`ordered choice` *a b* [PEG Pattern]

Parses *a*. If this fails, backtracks and parses *b*. Succeeds if either *a* or *b* succeeds.

```
"a/b"
```

```
(or a b)
```

`zero or more` *a* [PEG Pattern]

Parses *a* as many times in a row as it can, starting each *a* at the end of the text parsed by the previous *a*. Always succeeds.

```
"a*"
```

```
(* a)
```

`one or more` *a* [PEG Pattern]

Parses *a* as many times in a row as it can, starting each *a* at the end of the text parsed by the previous *a*. Succeeds if at least one *a* was parsed.

```
"a+"
```

```
(+ a)
```

`optional` *a* [PEG Pattern]

Tries to parse *a*. Succeeds if *a* succeeds.

```
"a?"
```

```
(? a)
```

`followed by` *a* [PEG Pattern]

Makes sure it is possible to parse *a*, but does not actually parse it. Succeeds if *a* would succeed.

```
"&a"
```

```
(followed-by a)
```

`not followed by` *a* [PEG Pattern]

Makes sure it is impossible to parse *a*, but does not actually parse it. Succeeds if *a* would fail.

```
"!a"
```

```
(not-followed-by a)
```

`string literal` *"abc"* [PEG Pattern]

Parses the string "*abc*". Succeeds if that parsing succeeds.

```
"'abc'"
```

```
"abc"
```

`any character` [PEG Pattern]

Parses any single character. Succeeds unless there is no more text to be parsed.

```
"."
```

```
peg-any
```

`character class` *a b* [PEG Pattern]

Alternative syntax for "Ordered Choice *a b*" if *a* and *b* are characters.

```
"[ab]"
```

```
(or "a" "b")
```

`range of characters` *a z* [PEG Pattern]

Parses any character falling between *a* and *z*.

```
"[a-z]"
```

```
(range #\a #\z)
```

Example:

```
"(a !b / c &d*) 'e'+"
```

Would be:

```
(and
 (or
  (and a (not-followed-by b))
  (and c (followed-by (* d))))
 (+ "e"))
```

Extended Syntax

There is some extra syntax for S-expressions.

`ignore` *a* [PEG Pattern]
: Ignore the text matching *a*

`capture` *a* [PEG Pattern]
: Capture the text matching *a*.

`peg` *a* [PEG Pattern]
: Embed the PEG pattern *a* using string syntax.

Example:

```
"!a / 'b'"
```

Is equivalent to

```
(or (peg "!a") "b")
```

and

```
(or (not-followed-by a) "b")
```

6.17.2 PEG API Reference

Define Macros

The most straightforward way to define a PEG is by using one of the define macros (both of these macroexpand into `define` expressions). These macros bind parsing functions to variables. These parsing functions may be invoked by `match-pattern` or `search-for-pattern`, which return a PEG match record. Raw data can be retrieved from this record with the PEG match deconstructor functions. More complicated (and perhaps enlightening) examples can be found in the tutorial.

`define-peg-string-patterns` *peg-string* [Scheme Macro]
: Defines all the nonterminals in the PEG *peg-string*. More precisely, `define-peg-string-patterns` takes a superset of PEGs. A normal PEG has a `<-` between the nonterminal and the pattern. `define-peg-string-patterns` uses this symbol to determine what information it should propagate up the parse tree. The normal `<-` propagates the matched text up the parse tree, `<--` propagates the matched text up the parse tree tagged with the name of the nonterminal, and `<` discards that matched text and propagates nothing up the parse tree. Also, nonterminals may consist of any

alphanumeric character or a "-" character (in normal PEGs nonterminals can only be alphabetic).

For example, if we:

```
(define-peg-string-patterns
  "as <- 'a'+
bs <- 'b'+
as-or-bs <- as/bs")
(define-peg-string-patterns
  "as-tag <-- 'a'+
bs-tag <-- 'b'+
as-or-bs-tag <-- as-tag/bs-tag")
```

Then:

```
(match-pattern as-or-bs "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
(match-pattern as-or-bs-tag "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as-or-bs-tag (as-tag aa))>
```

Note that in doing this, we have bound 6 variables at the toplevel (*as*, *bs*, *as-or-bs*, *as-tag*, *bs-tag*, and *as-or-bs-tag*).

`define-peg-pattern` *name capture-type peg-sexp* [Scheme Macro]

Defines a single nonterminal *name*. *capture-type* determines how much information is passed up the parse tree. *peg-sexp* is a PEG in S-expression form.

Possible values for capture-type:

`all` passes the matched text up the parse tree tagged with the name of the nonterminal.

`body` passes the matched text up the parse tree.

`none` passes nothing up the parse tree.

For Example, if we:

```
(define-peg-pattern as body (+ "a"))
(define-peg-pattern bs body (+ "b"))
(define-peg-pattern as-or-bs body (or as bs))
(define-peg-pattern as-tag all (+ "a"))
(define-peg-pattern bs-tag all (+ "b"))
(define-peg-pattern as-or-bs-tag all (or as-tag bs-tag))
```

Then:

```
(match-pattern as-or-bs "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
(match-pattern as-or-bs-tag "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as-or-bs-tag (as-tag aa))>
```

Note that in doing this, we have bound 6 variables at the toplevel (*as*, *bs*, *as-or-bs*, *as-tag*, *bs-tag*, and *as-or-bs-tag*).

Compile Functions

It is sometimes useful to be able to compile anonymous PEG patterns at runtime. These functions let you do that using either syntax.

`peg-string-compile` *peg-string capture-type* [Scheme Procedure]
Compiles the PEG pattern in *peg-string* propagating according to *capture-type* (capture-type can be any of the values from `define-peg-pattern`).

`compile-peg-pattern` *peg-sexp capture-type* [Scheme Procedure]
Compiles the PEG pattern in *peg-sexp* propagating according to *capture-type* (capture-type can be any of the values from `define-peg-pattern`).

The functions return syntax objects, which can be useful if you want to use them in macros. If all you want is to define a new nonterminal, you can do the following:

```
(define exp '(+ "a"))
(define as (compile (compile-peg-pattern exp 'body)))
```

You can use this nonterminal with all of the regular PEG functions:

```
(match-pattern as "aaaaa") ⇒
#<peg start: 0 end: 5 string: bbbbb tree: bbbbb>
```

Parsing & Matching Functions

For our purposes, "parsing" means parsing a string into a tree starting from the first character, while "matching" means searching through the string for a substring. In practice, the only difference between the two functions is that `match-pattern` gives up if it can't find a valid substring starting at index 0 and `search-for-pattern` keeps looking. They are both equally capable of "parsing" and "matching" given those constraints.

`match-pattern` *nonterm string* [Scheme Procedure]
Parses *string* using the PEG stored in *nonterm*. If no match was found, `match-pattern` returns false. If a match was found, a PEG match record is returned.

The `capture-type` argument to `define-peg-pattern` allows you to choose what information to hold on to while parsing. The options are:

`all` tag the matched text with the nonterminal

`body` just the matched text

`none` nothing

```
(define-peg-pattern as all (+ "a"))
(match-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as aa)>

(define-peg-pattern as body (+ "a"))
(match-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>

(define-peg-pattern as none (+ "a"))
(match-pattern as "aabbcc") ⇒
```

```
#<peg start: 0 end: 2 string: aabbcc tree: ()>

(define-peg-pattern bs body (+ "b"))
(match-pattern bs "aabbcc") ⇒
#f
```

`search-for-pattern` *nonterm-or-peg string* [Scheme Macro]

Searches through *string* looking for a matching subexpression. *nonterm-or-peg* can either be a nonterminal or a literal PEG pattern. When a literal PEG pattern is provided, `search-for-pattern` works very similarly to the regular expression searches many hackers are used to. If no match was found, `search-for-pattern` returns false. If a match was found, a PEG match record is returned.

```
(define-peg-pattern as body (+ "a"))
(search-for-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
(search-for-pattern (+ "a") "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>
(search-for-pattern "'a'+" "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: aa>

(define-peg-pattern as all (+ "a"))
(search-for-pattern as "aabbcc") ⇒
#<peg start: 0 end: 2 string: aabbcc tree: (as aa)>

(define-peg-pattern bs body (+ "b"))
(search-for-pattern bs "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: bb>
(search-for-pattern (+ "b") "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: bb>
(search-for-pattern "'b'+" "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: bb>

(define-peg-pattern zs body (+ "z"))
(search-for-pattern zs "aabbcc") ⇒
#f
(search-for-pattern (+ "z") "aabbcc") ⇒
#f
(search-for-pattern "'z'+" "aabbcc") ⇒
#f
```

PEG Match Records

The `match-pattern` and `search-for-pattern` functions both return PEG match records. Actual information can be extracted from these with the following functions.

`peg:string` *match-record* [Scheme Procedure]

Returns the original string that was parsed in the creation of `match-record`.

`peg:start` *match-record* [Scheme Procedure]
Returns the index of the first parsed character in the original string (from `peg:string`). If this is the same as `peg:end`, nothing was parsed.

`peg:end` *match-record* [Scheme Procedure]
Returns one more than the index of the last parsed character in the original string (from `peg:string`). If this is the same as `peg:start`, nothing was parsed.

`peg:substring` *match-record* [Scheme Procedure]
Returns the substring parsed by `match-record`. This is equivalent to `(substring (peg:string match-record) (peg:start match-record) (peg:end match-record))`.

`peg:tree` *match-record* [Scheme Procedure]
Returns the tree parsed by `match-record`.

`peg-record?` *match-record* [Scheme Procedure]
Returns true if `match-record` is a PEG match record, or false otherwise.

Example:

```
(define-peg-pattern bs all (peg "'b'+"))

(search-for-pattern bs "aabbcc") ⇒
#<peg start: 2 end: 4 string: aabbcc tree: (bs bb)>

(let ((pm (search-for-pattern bs "aabbcc")))
   `((string ,(peg:string pm))
     (start ,(peg:start pm))
     (end ,(peg:end pm))
     (substring ,(peg:substring pm))
     (tree ,(peg:tree pm))
     (record? ,(peg-record? pm)))) ⇒
((string "aabbcc")
 (start 2)
 (end 4)
 (substring "bb")
 (tree (bs "bb"))
 (record? #t))
```

Miscellaneous

`context-flatten` *tst lst* [Scheme Procedure]
Takes a predicate *tst* and a list *lst*. Flattens *lst* until all elements are either atoms or satisfy *tst*. If *lst* itself satisfies *tst*, `(list lst)` is returned (this is a flat list whose only element satisfies *tst*).

```
(context-flatten (lambda (x) (and (number? (car x)) (= (car x) 1))) '(2 2 (1 1 (2
(2 2 (1 1 (2 2)) 2 2 (1 1))
(context-flatten (lambda (x) (and (number? (car x)) (= (car x) 1))) '(1 1 (1 1 (2
```

```
((1 1 (1 1 (2 2)) (2 2 (1 1))))
```

If you're wondering why this is here, take a look at the tutorial.

`keyword-flatten` *terms lst* [Scheme Procedure]

A less general form of `context-flatten`. Takes a list of terminal atoms `terms` and flattens *lst* until all elements are either atoms, or lists which have an atom from `terms` as their first element.

```
(keyword-flatten '(a b) '(c a b (a c) (b c) (c (b a) (c a)))) ⇒
(c a b (a c) (b c) c (b a) c a)
```

If you're wondering why this is here, take a look at the tutorial.

6.17.3 PEG Tutorial

Parsing /etc/passwd

This example will show how to parse /etc/passwd using PEGs.

First we define an example /etc/passwd file:

```
(define *etc-passwd*
  "root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
messagebus:x:103:107::/var/run/dbus:/bin/false
")
```

As a first pass at this, we might want to have all the entries in /etc/passwd in a list.

Doing this with string-based PEG syntax would look like this:

```
(define-peg-string-patterns
  "passwd <- entry* !.
entry <-- (! NL .)* NL*
NL < '\n'")
```

A `passwd` file is 0 or more entries (`entry*`) until the end of the file (`!.` (`.` is any character, so `!.` means "not anything")). We want to capture the data in the nonterminal `passwd`, but not tag it with the name, so we use `<-`.

An entry is a series of 0 or more characters that aren't newlines (`(! NL .)*`) followed by 0 or more newlines (`NL*`). We want to tag all the entries with `entry`, so we use `<--`.

A newline is just a literal newline (`'\n'`). We don't want a bunch of newlines cluttering up the output, so we use `<` to throw away the captured data.

Here is the same PEG defined using S-expressions:

```
(define-peg-pattern passwd body (and (* entry) (not-followed-by peg-any)))
(define-peg-pattern entry all (and (* (and (not-followed-by NL) peg-any))
       (* NL)))
(define-peg-pattern NL none "\n")
```

Obviously this is much more verbose. On the other hand, it's more explicit, and thus easier to build automatically. However, there are some tricks that make S-expressions easier

to use in some cases. One is the `ignore` keyword; the string syntax has no way to say "throw away this text" except breaking it out into a separate nonterminal. For instance, to throw away the newlines we had to define `NL`. In the S-expression syntax, we could have simply written `(ignore "\n")`. Also, for the cases where string syntax is really much cleaner, the `peg` keyword can be used to embed string syntax in S-expression syntax. For instance, we could have written:

```
(define-peg-pattern passwd body (peg "entry* !."))
```

However we define it, parsing `*etc-passwd*` with the `passwd` nonterminal yields the same results:

```
(peg:tree (match-pattern passwd *etc-passwd*)) ⇒
((entry "root:x:0:0:root:/root:/bin/bash")
 (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
 (entry "bin:x:2:2:bin:/bin:/bin/sh")
 (entry "sys:x:3:3:sys:/dev:/bin/sh")
 (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
 (entry "messagebus:x:103:107::/var/run/dbus:/bin/false"))
```

However, here is something to be wary of:

```
(peg:tree (match-pattern passwd "one entry")) ⇒
(entry "one entry")
```

By default, the parse trees generated by PEGs are compressed as much as possible without losing information. It may not look like this is what you want at first, but uncompressed parse trees are an enormous headache (there's no easy way to predict how deep particular lists will nest, there are empty lists littered everywhere, etc. etc.). One side-effect of this, however, is that sometimes the compressor is too aggressive. No information is discarded when `((entry "one entry"))` is compressed to `(entry "one entry")`, but in this particular case it probably isn't what we want.

There are two functions for easily dealing with this: `keyword-flatten` and `context-flatten`. The `keyword-flatten` function takes a list of keywords and a list to flatten, then tries to coerce the list such that the first element of all sublists is one of the keywords. The `context-flatten` function is similar, but instead of a list of keywords it takes a predicate that should indicate whether a given sublist is good enough (refer to the API reference for more details).

What we want here is `keyword-flatten`.

```
(keyword-flatten '(entry) (peg:tree (match-pattern passwd *etc-passwd*))) ⇒
((entry "root:x:0:0:root:/root:/bin/bash")
 (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
 (entry "bin:x:2:2:bin:/bin:/bin/sh")
 (entry "sys:x:3:3:sys:/dev:/bin/sh")
 (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
 (entry "messagebus:x:103:107::/var/run/dbus:/bin/false"))
(keyword-flatten '(entry) (peg:tree (match-pattern passwd "one entry"))) ⇒
((entry "one entry"))
```

Of course, this is a somewhat contrived example. In practice we would probably just tag the `passwd` nonterminal to remove the ambiguity (using either the `all` keyword for S-expressions or the `<--` symbol for strings)..

```
(define-peg-pattern tag-passwd all (peg "entry* !."))
(peg:tree (match-pattern tag-passwd *etc-passwd*)) ⇒
(tag-passwd
  (entry "root:x:0:0:root:/root:/bin/bash")
  (entry "daemon:x:1:1:daemon:/usr/sbin:/bin/sh")
  (entry "bin:x:2:2:bin:/bin:/bin/sh")
  (entry "sys:x:3:3:sys:/dev:/bin/sh")
  (entry "nobody:x:65534:65534:nobody:/nonexistent:/bin/sh")
  (entry "messagebus:x:103:107::/var/run/dbus:/bin/false"))
(peg:tree (match-pattern tag-passwd "one entry"))
(tag-passwd
  (entry "one entry"))
```

If you're ever uncertain about the potential results of parsing something, remember the two absolute rules:

1. No parsing information will ever be discarded.
2. There will never be any lists with fewer than 2 elements.

For the purposes of (1), "parsing information" means things tagged with the `any` keyword or the `<--` symbol. Plain strings will be concatenated.

Let's extend this example a bit more and actually pull some useful information out of the passwd file:

```
(define-peg-string-patterns
  "passwd <-- entry* !.
entry <-- login C pass C uid C gid C nameORcomment C homedir C shell NL*
login <-- text
pass <-- text
uid <-- [0-9]*
gid <-- [0-9]*
nameORcomment <-- text
homedir <-- path
shell <-- path
path <-- (SLASH pathELEMENT)*
pathELEMENT <-- (!NL !C  !'/' .)*
text <- (!NL !C  .)*
C < ':'
NL < '\n'
SLASH < '/'")
```

This produces rather pretty parse trees:

```
(passwd
  (entry (login "root")
         (pass "x")
         (uid "0")
         (gid "0")
         (nameORcomment "root")
         (homedir (path (pathELEMENT "root")))
         (shell (path (pathELEMENT "bin") (pathELEMENT "bash")))))
```

```
(entry (login "daemon")
       (pass "x")
       (uid "1")
       (gid "1")
       (nameORcomment "daemon")
       (homedir
         (path (pathELEMENT "usr") (pathELEMENT "sbin")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "bin")
       (pass "x")
       (uid "2")
       (gid "2")
       (nameORcomment "bin")
       (homedir (path (pathELEMENT "bin")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "sys")
       (pass "x")
       (uid "3")
       (gid "3")
       (nameORcomment "sys")
       (homedir (path (pathELEMENT "dev")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "nobody")
       (pass "x")
       (uid "65534")
       (gid "65534")
       (nameORcomment "nobody")
       (homedir (path (pathELEMENT "nonexistent")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "sh"))))
(entry (login "messagebus")
       (pass "x")
       (uid "103")
       (gid "107")
       nameORcomment
       (homedir
         (path (pathELEMENT "var")
               (pathELEMENT "run")
               (pathELEMENT "dbus")))
       (shell (path (pathELEMENT "bin") (pathELEMENT "false")))))
```

Notice that when there's no entry in a field (e.g. `nameORcomment` for messagebus) the symbol is inserted. This is the "don't throw away any information" rule—we succesfully matched a `nameORcomment` of 0 characters (since we used `*` when defining it). This is usually what you want, because it allows you to e.g. use `list-ref` to pull out elements (since they all have known offsets).

If you'd prefer not to have symbols for empty matches, you can replace the `*` with a `+` and add a `?` after the `nameORcomment` in `entry`. Then it will try to parse 1 or more

characters, fail (inserting nothing into the parse tree), but continue because it didn't have to match the nameORcomment to continue.

Embedding Arithmetic Expressions

We can parse simple mathematical expressions with the following PEG:

```
(define-peg-string-patterns
  "expr <- sum
sum <-- (product ('+' / '-') sum) / product
product <-- (value ('*' / '/') product) / value
value <-- number / '(' expr ')'
number <-- [0-9]+")
```

Then:

```
(peg:tree (match-pattern expr "1+1/2*3+(1+1)/2")) ⇒
(sum (product (value (number "1")))
     "+"
     (sum (product
            (value (number "1"))
            "/"
            (product
              (value (number "2"))
              "*"
              (product (value (number "3")))))
          "+"
          (sum (product
                 (value "("
                        (sum (product (value (number "1")))
                             "+"
                             (sum (product (value (number "1")))))
                        ")")
                 "/"
                 (product (value (number "2")))))))
```

There is very little wasted effort in this PEG. The `number` nonterminal has to be tagged because otherwise the numbers might run together with the arithmetic expressions during the string concatenation stage of parse-tree compression (the parser will see "1" followed by "/" and decide to call it "1/"). When in doubt, tag.

It is very easy to turn these parse trees into lisp expressions:

```
(define (parse-sum sum left . rest)
  (if (null? rest)
      (apply parse-product left)
      (list (string->symbol (car rest))
    (apply parse-product left)
    (apply parse-sum (cadr rest)))))

(define (parse-product product left . rest)
  (if (null? rest)
```

```
      (apply parse-value left)
      (list (string->symbol (car rest))
    (apply parse-value left)
    (apply parse-product (cadr rest)))))

(define (parse-value value first . rest)
  (if (null? rest)
      (string->number (cadr first))
      (apply parse-sum (car rest))))

(define parse-expr parse-sum)
```

(Notice all these functions look very similar; for a more complicated PEG, it would be worth abstracting.)

Then:

```
(apply parse-expr (peg:tree (match-pattern expr "1+1/2*3+(1+1)/2"))) ⇒
(+ 1 (+ (/ 1 (* 2 3)) (/ (+ 1 1) 2)))
```

But wait! The associativity is wrong! Where it says `(/ 1 (* 2 3))`, it should say `(* (/ 1 2) 3)`.

It's tempting to try replacing e.g. `"sum <-- (product ('+' / '-') sum) / product"` with `"sum <-- (sum ('+' / '-') product) / product"`, but this is a Bad Idea. PEGs don't support left recursion. To see why, imagine what the parser will do here. When it tries to parse `sum`, it first has to try and parse `sum`. But to do that, it first has to try and parse `sum`. This will continue until the stack gets blown off.

So how does one parse left-associative binary operators with PEGs? Honestly, this is one of their major shortcomings. There's no general-purpose way of doing this, but here the repetition operators are a good choice:

```
(use-modules (srfi srfi-1))

(define-peg-string-patterns
  "expr <- sum
sum <-- (product ('+' / '-'))* product
product <-- (value ('*' / '/'))* value
value <-- number / '(' expr ')'
number <-- [0-9]+")

;; take a deep breath...
(define (make-left-parser next-func)
  (lambda (sum first . rest) ;; general form, comments below assume
    ;; that we're dealing with a sum expression
    (if (null? rest) ;; form (sum (product ...))
      (apply next-func first)
      (if (string? (cadr first));; form (sum ((product ...) "+") (product ...))
  (list (string->symbol (cadr first))
(apply next-func (car first))
(apply next-func (car rest)))
```

```
          ;; form (sum (((product ...) "+") ((product ...) "+")) (product ...))
   (car
    (reduce ;; walk through the list and build a left-associative tree
     (lambda (l r)
       (list (list (cadr r) (car r) (apply next-func (car l)))
     (string->symbol (cadr l))))
     'ignore
     (append ;; make a list of all the products
             ;; the first one should be pre-parsed
      (list (list (apply next-func (caar first))
 (string->symbol (cadar first))))
      (cdr first)
             ;; the last one has to be added in
      (list (append rest '("done")))))))))

(define (parse-value value first . rest)
  (if (null? rest)
      (string->number (cadr first))
      (apply parse-sum (car rest))))
(define parse-product (make-left-parser parse-value))
(define parse-sum (make-left-parser parse-product))
(define parse-expr parse-sum)
```

Then:

```
(apply parse-expr (peg:tree (match-pattern expr "1+1/2*3+(1+1)/2"))) ⇒
(+ (+ 1 (* (/ 1 2) 3)) (/ (+ 1 1) 2))
```

As you can see, this is much uglier (it could be made prettier by using `context-flatten`, but the way it's written above makes it clear how we deal with the three ways the zero-or-more * expression can parse). Fortunately, most of the time we can get away with only using right-associativity.

Simplified Functions

For a more tantalizing example, consider the following grammar that parses (highly) simplified C functions:

```
(define-peg-string-patterns
  "cfunc <-- cSP ctype cSP cname cSP cargs cLB cSP cbody cRB
ctype <-- cidentifier
cname <-- cidentifier
cargs <-- cLP (! (cSP cRP) carg cSP (cCOMMA / cRP) cSP)* cSP
carg <-- cSP ctype cSP cname
cbody <-- cstatement *
cidentifier <- [a-zA-z][a-zA-Z0-9_]*
cstatement <-- (!';'.)*cSC cSP
cSC < ';'
cCOMMA < ','
cLP < '('
cRP < ')'
```

```
cLB < '{'
cRB < '}'
cSP < [ \t\n]*")
```

Then:

```
(match-pattern cfunc "int square(int a) { return a*a;}") ⇒
(32
 (cfunc (ctype "int")
        (cname "square")
        (cargs (carg (ctype "int") (cname "a")))
        (cbody (cstatement "return a*a"))))
```

And:

```
(match-pattern cfunc "int mod(int a, int b) { int c = a/b;return a-b*c; }") ⇒
(52
 (cfunc (ctype "int")
        (cname "mod")
        (cargs (carg (ctype "int") (cname "a"))
               (carg (ctype "int") (cname "b")))
        (cbody (cstatement "int c = a/b")
               (cstatement "return a- b*c"))))
```

By wrapping all the `carg` nonterminals in a `cargs` nonterminal, we were able to remove any ambiguity in the parsing structure and avoid having to call `context-flatten` on the output of `match-pattern`. We used the same trick with the `cstatement` nonterminals, wrapping them in a `cbody` nonterminal.

The whitespace nonterminal `cSP` used here is a (very) useful instantiation of a common pattern for matching syntactically irrelevant information. Since it's tagged with `<` and ends with `*` it won't clutter up the parse trees (all the empty lists will be discarded during the compression step) and it will never cause parsing to fail.

6.17.4 PEG Internals

A PEG parser takes a string as input and attempts to parse it as a given nonterminal. The key idea of the PEG implementation is that every nonterminal is just a function that takes a string as an argument and attempts to parse that string as its nonterminal. The functions always start from the beginning, but a parse is considered successful if there is material left over at the end.

This makes it easy to model different PEG parsing operations. For instance, consider the PEG grammar `"ab"`, which could also be written `(and "a" "b")`. It matches the string "ab". Here's how that might be implemented in the PEG style:

```
(define (match-and-a-b str)
  (match-a str)
  (match-b str))
```

As you can see, the use of functions provides an easy way to model sequencing. In a similar way, one could model `(or a b)` with something like the following:

```
(define (match-or-a-b str)
  (or (match-a str) (match-b str)))
```

Here the semantics of a PEG `or` expression map naturally onto Scheme's `or` operator. This function will attempt to run `(match-a str)`, and return its result if it succeeds. Otherwise it will run `(match-b str)`.

Of course, the code above wouldn't quite work. We need some way for the parsing functions to communicate. The actual interface used is below.

Parsing Function Interface

A parsing function takes three arguments - a string, the length of that string, and the position in that string it should start parsing at. In effect, the parsing functions pass around substrings in pieces - the first argument is a buffer of characters, and the second two give a range within that buffer that the parsing function should look at.

Parsing functions return either #f, if they failed to match their nonterminal, or a list whose first element must be an integer representing the final position in the string they matched and whose cdr can be any other data the function wishes to return, or '() if it doesn't have any more data.

The one caveat is that if the extra data it returns is a list, any adjacent strings in that list will be appended by `match-pattern`. For instance, if a parsing function returns `(13 ("a" "b" "c"))`, `match-pattern` will take `(13 ("abc"))` as its value.

For example, here is a function to match "ab" using the actual interface.

```
(define (match-a-b str len pos)
   (and (<= (+ pos 2) len)
        (string= str "ab" pos (+ pos 2))
        (list (+ pos 2) '())))  ; we return no extra information
```

The above function can be used to match a string by running `(match-pattern match-a-b "ab")`.

Code Generators and Extensible Syntax

PEG expressions, such as those in a `define-peg-pattern` form, are interpreted internally in two steps.

First, any string PEG is expanded into an s-expression PEG by the code in the `(ice-9 peg string-peg)` module.

Then, the s-expression PEG that results is compiled into a parsing function by the `(ice-9 peg codegen)` module. In particular, the function `compile-peg-pattern` is called on the s-expression. It then decides what to do based on the form it is passed.

The PEG syntax can be expanded by providing `compile-peg-pattern` more options for what to do with its forms. The extended syntax will be associated with a symbol, for instance `my-parsing-form`, and will be called on all PEG expressions of the form

```
(my-parsing-form ...)
```

The parsing function should take two arguments. The first will be a syntax object containing a list with all of the arguments to the form (but not the form's name), and the second will be the `capture-type` argument that is passed to `define-peg-pattern`.

New functions can be registered by calling `(add-peg-compiler! symbol function)`, where `symbol` is the symbol that will indicate a form of this type and `function` is the code generating function described above. The function `add-peg-compiler!` is exported from the `(ice-9 peg codegen)` module.

6.18 Reading and Evaluating Scheme Code

This chapter describes Guile functions that are concerned with reading, loading, evaluating, and compiling Scheme code at run time.

6.18.1 Scheme Syntax: Standard and Guile Extensions

6.18.1.1 Expression Syntax

An expression to be evaluated takes one of the following forms.

symbol A symbol is evaluated by dereferencing. A binding of that symbol is sought and the value there used. For example,

```
(define x 123)
x ⇒ 123
```

(*proc args...*)

A parenthesised expression is a function call. *proc* and each argument are evaluated, then the function (which *proc* evaluated to) is called with those arguments.

The order in which *proc* and the arguments are evaluated is unspecified, so be careful when using expressions with side effects.

```
(max 1 2 3) ⇒ 3

(define (get-some-proc)  min)
((get-some-proc) 1 2 3) ⇒ 1
```

The same sort of parenthesised form is used for a macro invocation, but in that case the arguments are not evaluated. See the descriptions of macros for more on this (see Section 6.10 [Macros], page 261, and see Section 6.10.2 [Syntax Rules], page 263).

constant Number, string, character and boolean constants evaluate "to themselves", so can appear as literals.

```
123     ⇒ 123
99.9    ⇒ 99.9
"hello" ⇒ "hello"
#\z     ⇒ #\z
#t      ⇒ #t
```

Note that an application must not attempt to modify literal strings, since they may be in read-only memory.

(quote *data*)

'*data* Quoting is used to obtain a literal symbol (instead of a variable reference), a literal list (instead of a function call), or a literal vector. ' is simply a shorthand for a `quote` form. For example,

```
'x                  ⇒ x
'(1 2 3)            ⇒ (1 2 3)
'#(1 (2 3) 4)       ⇒ #(1 (2 3) 4)
(quote x)           ⇒ x
```

```
(quote (1 2 3))       ⇒ (1 2 3)
(quote #(1 (2 3) 4)) ⇒ #(1 (2 3) 4)
```

Note that an application must not attempt to modify literal lists or vectors obtained from a `quote` form, since they may be in read-only memory.

`(quasiquote` *data*`)`

`‘`*data* Backquote quasi-quotation is like `quote`, but selected sub-expressions are evaluated. This is a convenient way to construct a list or vector structure most of which is constant, but at certain points should have expressions substituted.

The same effect can always be had with suitable `list`, `cons` or `vector` calls, but quasi-quoting is often easier.

`(unquote` *expr*`)`

`,`*expr* Within the quasiquote *data*, `unquote` or `,` indicates an expression to be evaluated and inserted. The comma syntax `,` is simply a shorthand for an `unquote` form. For example,

```
‘(1 2 (* 9 9) 3 4)       ⇒ (1 2 (* 9 9) 3 4)
‘(1 2 ,(* 9 9) 3 4)      ⇒ (1 2 81 3 4)
‘(1 (unquote (+ 1 1)) 3) ⇒ (1 2 3)
‘#(1 ,(/ 12 2))          ⇒ #(1 6)
```

`(unquote-splicing` *expr*`)`

`,@`*expr* Within the quasiquote *data*, `unquote-splicing` or `,@` indicates an expression to be evaluated and the elements of the returned list inserted. *expr* must evaluate to a list. The “comma-at” syntax `,@` is simply a shorthand for an `unquote-splicing` form.

```
(define x '(2 3))
‘(1 ,x 4)                          ⇒ (1 (2 3) 4)
‘(1 ,@x 4)                         ⇒ (1 2 3 4)
‘(1 (unquote-splicing (map 1+ x))) ⇒ (1 3 4)
‘#(9 ,@x 9)                        ⇒ #(9 2 3 9)
```

Notice `,@` differs from plain `,` in the way one level of nesting is stripped. For `,@` the elements of a returned list are inserted, whereas with `,` it would be the list itself inserted.

6.18.1.2 Comments

Comments in Scheme source files are written by starting them with a semicolon character (`;`). The comment then reaches up to the end of the line. Comments can begin at any column, and the may be inserted on the same line as Scheme code.

```
; Comment
;; Comment too
(define x 1)        ; Comment after expression
(let ((y 1))
  ;; Display something.
  (display y)
;;; Comment at left margin.
  (display (+ y 1)))
```

It is common to use a single semicolon for comments following expressions on a line, to use two semicolons for comments which are indented like code, and three semicolons for comments which start at column 0, even if they are inside an indented code block. This convention is used when indenting code in Emacs' Scheme mode.

6.18.1.3 Block Comments

In addition to the standard line comments defined by R5RS, Guile has another comment type for multiline comments, called *block comments*. This type of comment begins with the character sequence `#!` and ends with the characters `!#`.

These comments are compatible with the block comments in the Scheme Shell `scsh` (see Section 7.18 [The Scheme shell (scsh)], page 743). The characters `#!` were chosen because they are the magic characters used in shell scripts for indicating that the name of the program for executing the script follows on the same line.

Thus a Guile script often starts like this.

```
#! /usr/local/bin/guile -s
!#
```

More details on Guile scripting can be found in the scripting section (see Section 4.3 [Guile Scripting], page 41).

Similarly, Guile (starting from version 2.0) supports nested block comments as specified by R6RS and SRFI-30 (`http://srfi.schemers.org/srfi-30/srfi-30.html`):

```
(+ 1 #| this is a #| nested |# block comment |# 2)
⇒ 3
```

For backward compatibility, this syntax can be overridden with `read-hash-extend` (see Section 6.18.1.6 [Reader Extensions], page 385).

There is one special case where the contents of a comment can actually affect the interpretation of code. When a character encoding declaration, such as `coding: utf-8` appears in one of the first few lines of a source file, it indicates to Guile's default reader that this source code file is not ASCII. For details see Section 6.18.8 [Character Encoding of Source Files], page 395.

6.18.1.4 Case Sensitivity

Scheme as defined in R5RS is not case sensitive when reading symbols. Guile, on the contrary is case sensitive by default, so the identifiers

```
guile-whuzzy
Guile-Whuzzy
```

are the same in R5RS Scheme, but are different in Guile.

It is possible to turn off case sensitivity in Guile by setting the reader option `case-insensitive`. For more information on reader options, See Section 6.18.2 [Scheme Read], page 385.

```
(read-enable 'case-insensitive)
```

It is also possible to disable (or enable) case sensitivity within a single file by placing the reader directives `#!fold-case` (or `#!no-fold-case`) within the file itself.

6.18.1.5 Keyword Syntax

6.18.1.6 Reader Extensions

`read-hash-extend` *chr proc* [Scheme Procedure]
`scm_read_hash_extend` (*chr, proc*) [C Function]

Install the procedure *proc* for reading expressions starting with the character sequence `#` and *chr*. *proc* will be called with two arguments: the character *chr* and the port to read further data from. The object returned will be the return value of `read`. Passing `#f` for *proc* will remove a previous setting.

6.18.2 Reading Scheme Code

`read` [*port*] [Scheme Procedure]
`scm_read` (*port*) [C Function]

Read an s-expression from the input port *port*, or from the current input port if *port* is not specified. Any whitespace before the next token is discarded.

The behaviour of Guile's Scheme reader can be modified by manipulating its read options.

`read-options` [*setting*] [Scheme Procedure]

Display the current settings of the global read options. If *setting* is omitted, only a short form of the current read options is printed. Otherwise if *setting* is the symbol `help`, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking `read-options` at the prompt.

```
scheme@(guile-user)> (read-options)
(square-brackets keywords #f positions)
scheme@(guile-user)> (read-options 'help)
copy              no    Copy source code expressions.
positions         yes   Record positions of source code expressions.
case-insensitive  no    Convert symbols to lower case.
keywords          #f    Style of keyword recognition: #f, 'prefix or 'postfix.
r6rs-hex-escapes  no    Use R6RS variable-length character and string hex escapes.
square-brackets   yes   Treat `[' and `]' as parentheses, for R6RS compatibility.
hungry-eol-escapes no   In strings, consume leading whitespace after an
                        escaped end-of-line.
curly-infix       no    Support SRFI-105 curly infix expressions.
r7rs-symbols      no    Support R7RS |...| symbol notation.
```

Note that Guile also includes a preliminary mechanism for setting read options on a per-port basis. For instance, the `case-insensitive` read option is set (or unset) on the port when the reader encounters the `#!fold-case` or `#!no-fold-case` reader directives. Similarly, the `#!curly-infix` reader directive sets the `curly-infix` read option on the port, and `#!curly-infix-and-bracket-lists` sets `curly-infix` and unsets `square-brackets` on the port (see Section 7.5.44 [SRFI-105], page 654). There is currently no other way to access or set the per-port read options.

The boolean options may be toggled with `read-enable` and `read-disable`. The non-boolean `keywords` option must be set using `read-set!`.

`read-enable` *option-name* [Scheme Procedure]
`read-disable` *option-name* [Scheme Procedure]

`read-set!` *option-name value* [Scheme Syntax]

Modify the read options. `read-enable` should be used with boolean options and switches them on, `read-disable` switches them off.

`read-set!` can be used to set an option to a specific value. Due to historical oddities, it is a macro that expects an unquoted option name.

For example, to make `read` fold all symbols to their lower case (perhaps for compatibility with older Scheme code), you can enter:

```
(read-enable 'case-insensitive)
```

For more information on the effect of the `r6rs-hex-escapes` and `hungry-eol-escapes` options, see (see Section 6.6.5.1 [String Syntax], page 141).

For more information on the `r7rs-symbols` option, see (see Section 6.6.6.6 [Symbol Read Syntax], page 171).

6.18.3 Writing Scheme Values

Any scheme value may be written to a port. Not all values may be read back in (see Section 6.18.2 [Scheme Read], page 385), however.

`write` *obj* [*port*] [Scheme Procedure]

Send a representation of *obj* to *port* or to the current output port if not given.

The output is designed to be machine readable, and can be read back with `read` (see Section 6.18.2 [Scheme Read], page 385). Strings are printed in double quotes, with escapes if necessary, and characters are printed in '`#\`' notation.

`display` *obj* [*port*] [Scheme Procedure]

Send a representation of *obj* to *port* or to the current output port if not given.

The output is designed for human readability, it differs from `write` in that strings are printed without double quotes and escapes, and characters are printed as per `write-char`, not in '`#\`' form.

As was the case with the Scheme reader, there are a few options that affect the behavior of the Scheme printer.

`print-options` [*setting*] [Scheme Procedure]

Display the current settings of the read options. If *setting* is omitted, only a short form of the current read options is printed. Otherwise if *setting* is the symbol `help`, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking `print-options` at the prompt.

```
scheme@(guile-user)> (print-options)
(quote-keywordish-symbols reader highlight-suffix "}" highlight-prefix "{")
scheme@(guile-user)> (print-options 'help)
highlight-prefix          {       The string to print before highlighted values.
highlight-suffix          }       The string to print after highlighted values.
quote-keywordish-symbols  reader  How to print symbols that have a colon
                                  as their first or last character. The
                                  value '#f' does not quote the colons;
                                  '#t' quotes them; 'reader' quotes them
```

```
                                  when the reader option 'keywords' is
                                  not '#f'.
escape-newlines          yes      Render newlines as \n when printing
                                  using 'write'.
r7rs-symbols             no       Escape symbols using R7RS |...| symbol
                                  notation.
```

These options may be modified with the print-set! syntax.

`print-set!` *option-name value* [Scheme Syntax]

Modify the print options. Due to historical oddities, `print-set!` is a macro that expects an unquoted option name.

6.18.4 Procedures for On the Fly Evaluation

Scheme has the lovely property that its expressions may be represented as data. The `eval` procedure takes a Scheme datum and evaluates it as code.

`eval` *exp module_or_state* [Scheme Procedure]
`scm_eval` (*exp, module_or_state*) [C Function]

Evaluate *exp*, a list representing a Scheme expression, in the top-level environment specified by *module_or_state*. While *exp* is evaluated (using `primitive-eval`), *module_or_state* is made the current module. The current module is reset to its previous value when `eval` returns. XXX - dynamic states. Example: (eval '(+ 1 2) (interaction-environment))

`interaction-environment` [Scheme Procedure]
`scm_interaction_environment` () [C Function]

Return a specifier for the environment that contains implementation–defined bindings, typically a superset of those listed in the report. The intent is that this procedure will return the environment in which the implementation would evaluate expressions dynamically typed by the user.

See Section 6.20.12 [Environments], page 427, for other environments.

One does not always receive code as Scheme data, of course, and this is especially the case for Guile's other language implementations (see Section 6.24 [Other Languages], page 461). For the case in which all you have is a string, we have `eval-string`. There is a legacy version of this procedure in the default environment, but you really want the one from `(ice-9 eval-string)`, so load it up:

```
(use-modules (ice-9 eval-string))
```

`eval-string` *string* [*#:module=#f*] [*#:file=#f*] [*#:line=#f*] [*#:column=#f*] [*#:lang=*(*current-language*)] [*#:compile?=#f*] [Scheme Procedure]

Parse *string* according to the current language, normally Scheme. Evaluate or compile the expressions it contains, in order, returning the last expression.

If the *module* keyword argument is set, save a module excursion (see Section 6.20.8 [Module System Reflection], page 420) and set the current module to *module* before evaluation.

The *file*, *line*, and *column* keyword arguments can be used to indicate that the source string begins at a particular source location.

Finally, *lang* is a language, defaulting to the current language, and the expression is compiled if *compile?* is true or there is no evaluator for the given language.

`scm_eval_string` (*string*) [C Function]
`scm_eval_string_in_module` (*string, module*) [C Function]
These C bindings call `eval-string` from `(ice-9 eval-string)`, evaluating within *module* or the current module.

`SCM scm_c_eval_string` (*const char *string*) [C Function]
`scm_eval_string`, but taking a C string in locale encoding instead of an `SCM`.

`apply` *proc arg ... arglst* [Scheme Procedure]
`scm_apply_0` (*proc, arglst*) [C Function]
`scm_apply_1` (*proc, arg1, arglst*) [C Function]
`scm_apply_2` (*proc, arg1, arg2, arglst*) [C Function]
`scm_apply_3` (*proc, arg1, arg2, arg3, arglst*) [C Function]
`scm_apply` (*proc, arg, rest*) [C Function]
Call *proc* with arguments *arg ...* and the elements of the *arglst* list.

`scm_apply` takes parameters corresponding to a Scheme level `(lambda (proc arg1 . rest) ...)`. So *arg1* and all but the last element of the *rest* list make up *arg ...*, and the last element of *rest* is the *arglst* list. Or if *rest* is the empty list `SCM_EOL` then there's no *arg ...*, and (*arg1*) is the *arglst*.

arglst is not modified, but the *rest* list passed to `scm_apply` is modified.

`scm_call_0` (*proc*) [C Function]
`scm_call_1` (*proc, arg1*) [C Function]
`scm_call_2` (*proc, arg1, arg2*) [C Function]
`scm_call_3` (*proc, arg1, arg2, arg3*) [C Function]
`scm_call_4` (*proc, arg1, arg2, arg3, arg4*) [C Function]
`scm_call_5` (*proc, arg1, arg2, arg3, arg4, arg5*) [C Function]
`scm_call_6` (*proc, arg1, arg2, arg3, arg4, arg5, arg6*) [C Function]
`scm_call_7` (*proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7*) [C Function]
`scm_call_8` (*proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8*) [C Function]
`scm_call_9` (*proc, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8, arg9*) [C Function]
Call *proc* with the given arguments.

`scm_call` (*proc, ...*) [C Function]
Call *proc* with any number of arguments. The argument list must be terminated by `SCM_UNDEFINED`. For example:

```
scm_call (scm_c_public_ref ("guile", "+"),
          scm_from_int (1),
          scm_from_int (2),
          SCM_UNDEFINED);
```

`scm_call_n` (*proc, argv, nargs*) [C Function]
Call *proc* with the array of arguments *argv*, as a `SCM*`. The length of the arguments should be passed in *nargs*, as a `size_t`.

`primitive-eval` *exp* [Scheme Procedure]
`scm_primitive_eval` (*exp*) [C Function]
Evaluate *exp* in the top-level environment specified by the current module.

6.18.5 Compiling Scheme Code

The `eval` procedure directly interprets the S-expression representation of Scheme. An alternate strategy for evaluation is to determine ahead of time what computations will be necessary to evaluate the expression, and then use that recipe to produce the desired results. This is known as *compilation*.

While it is possible to compile simple Scheme expressions such as `(+ 2 2)` or even `"Hello world!"`, compilation is most interesting in the context of procedures. Compiling a lambda expression produces a compiled procedure, which is just like a normal procedure except typically much faster, because it can bypass the generic interpreter.

Functions from system modules in a Guile installation are normally compiled already, so they load and run quickly.

Note that well-written Scheme programs will not typically call the procedures in this section, for the same reason that it is often bad taste to use `eval`. By default, Guile automatically compiles any files it encounters that have not been compiled yet (see Section 4.2 [Invoking Guile], page 35). The compiler can also be invoked explicitly from the shell as `guild compile foo.scm`.

(Why are calls to `eval` and `compile` usually in bad taste? Because they are limited, in that they can only really make sense for top-level expressions. Also, most needs for "compile-time" computation are fulfilled by macros and closures. Of course one good counterexample is the REPL itself, or any code that reads expressions from a port.)

Automatic compilation generally works transparently, without any need for user intervention. However Guile does not yet do proper dependency tracking, so that if file `a.scm` uses macros from `b.scm`, and *b.scm* changes, `a.scm` would not be automatically recompiled. To forcibly invalidate the auto-compilation cache, pass the `--fresh-auto-compile` option to Guile, or set the `GUILE_AUTO_COMPILE` environment variable to `fresh` (instead of to `0` or `1`).

For more information on the compiler itself, see Section 9.4 [Compiling to the Virtual Machine], page 856. For information on the virtual machine, see Section 9.3 [A Virtual Machine for Guile], page 827.

The command-line interface to Guile's compiler is the `guild compile` command:

`guild compile` [`option...`] *`file...`* [Command]
Compile *file*, a source file, and store bytecode in the compilation cache or in the file specified by the `-o` option. The following options are available:

`-L` *`dir`*
`--load-path=`*`dir`*
Add *dir* to the front of the module load path.

`-o` *ofile*
`--output=`*ofile*
: Write output bytecode to *ofile*. By convention, bytecode file names end in `.go`. When `-o` is omitted, the output file name is as for `compile-file` (see below).

`-x` *extension*
: Recognize *extension* as a valid source file name extension.

  For example, to compile R6RS code, you might want to pass `-x .sls` so that files ending in `.sls` can be found.

`-W` *warning*
`--warn=`*warning*
: Emit warnings of type *warning*; use `--warn=help` for a list of available warnings and their description. Currently recognized warnings include `unused-variable`, `unused-toplevel`, `shadowed-toplevel`, `unbound-variable`, `macro-use-before-definition`, `arity-mismatch`, `format`, `duplicate-case-datum`, and `bad-case-datum`.

`-O` *opt*
`--optimize=`*opt*
: Enable or disable specific compiler optimizations; use `-Ohelp` for a list of available options. The default is `-O2`, which enables most optimizations. `-O0` is recommended if compilation speed is more important than the speed of the compiled code. Pass `-Ono-`*opt* to disable a specific compiler pass. Any number of `-O` options can be passed to the compiler, with later ones taking precedence.

`--r6rs`
`--r7rs`
: Compile in an environment whose default bindings, reader options, and load paths are adapted for specific Scheme standards. See Section 7.6 [R6RS Support], page 662, and See Section 7.7 [R7RS Support], page 704.

`-f` *lang*
`--from=`*lang*
: Use *lang* as the source language of *file*. If this option is omitted, `scheme` is assumed.

`-t` *lang*
`--to=`*lang*
: Use *lang* as the target language of *file*. If this option is omitted, `rtl` is assumed.

`-T` *target*
`--target=`*target*
: Produce code for *target* instead of *%host-type* (see Section 6.23.1 [Build Config], page 456). Target must be a valid GNU triplet, such as `armv5tel-unknown-linux-gnueabi` (see Section "Specifying Target Triplets" in *GNU Autoconf Manual*).

Each *file* is assumed to be UTF-8-encoded, unless it contains a coding declaration as recognized by `file-encoding` (see Section 6.18.8 [Character Encoding of Source Files], page 395).

The compiler can also be invoked directly by Scheme code. These interfaces are in their own module:

```
(use-modules (system base compile))
```

`compile` *exp* [*#:env=#f*] [*#:from=(current-language)*] [Scheme Procedure]
[*#:to=value*] [*#:opts='()*]
[*#:optimization-level=(default-optimization-level)*]
[*#:warning-level=(default-warning-level)*]

Compile the expression *exp* in the environment *env*. If *exp* is a procedure, the result will be a compiled procedure; otherwise `compile` is mostly equivalent to `eval`.

For a discussion of languages and compiler options, See Section 9.4 [Compiling to the Virtual Machine], page 856.

`compile-file` *file* [*#:output-file=#f*] [Scheme Procedure]
[*#:from=(current-language)*] [*#:to='rtl*] [*#:env=(default-environment from)*] [*#:opts='()*] [*#:optimization-level=(default-optimization-level)*]
[*#:warning-level=(default-warning-level)*] [*#:canonicalization='relative*]

Compile the file named *file*.

Output will be written to a *output-file*. If you do not supply an output file name, output is written to a file in the cache directory, as computed by `(compiled-file-name file)`.

from and *to* specify the source and target languages. See Section 9.4 [Compiling to the Virtual Machine], page 856, for more information on these options, and on *env* and *opts*.

As with `guild compile`, *file* is assumed to be UTF-8-encoded unless it contains a coding declaration.

`default-optimization-level` [Scheme Parameter]

The default optimization level, as an integer from 0 to 9. The default is 2.

`default-warning-level` [Scheme Parameter]

The default warning level, as an integer from 0 to 9. The default is 1.

See Section 6.13.12 [Parameters], page 326, for more on how to set parameters.

`compiled-file-name` *file* [Scheme Procedure]

Compute a cached location for a compiled version of a Scheme file named *file*.

This file will usually be below the `$HOME/.cache/guile/ccache` directory, depending on the value of the `XDG_CACHE_HOME` environment variable. The intention is that `compiled-file-name` provides a fallback location for caching auto-compiled files. If you want to place a compile file in the `%load-compiled-path`, you should pass the *output-file* option to `compile-file`, explicitly.

`%auto-compilation-options` [Scheme Variable]
This variable contains the options passed to the `compile-file` procedure when auto-compiling source files. By default, it enables useful compilation warnings. It can be customized from `~/.guile`.

6.18.6 Loading Scheme Code from File

`load` *filename* [*reader*] [Scheme Procedure]
Load *filename* and evaluate its contents in the top-level environment.

reader if provided should be either `#f`, or a procedure with the signature `(lambda (port) ...)` which reads the next expression from *port*. If *reader* is `#f` or absent, Guile's built-in `read` procedure is used (see Section 6.18.2 [Scheme Read], page 385).

The *reader* argument takes effect by setting the value of the `current-reader` fluid (see below) before loading the file, and restoring its previous value when loading is complete. The Scheme code inside *filename* can itself change the current reader procedure on the fly by setting `current-reader` fluid.

If the variable `%load-hook` is defined, it should be bound to a procedure that will be called before any code is loaded. See documentation for `%load-hook` later in this section.

`load-compiled` *filename* [Scheme Procedure]
Load the compiled file named *filename*.

Compiling a source file (see Section 6.18 [Read/Load/Eval/Compile], page 382) and then calling `load-compiled` on the resulting file is equivalent to calling `load` on the source file.

`primitive-load` *filename* [Scheme Procedure]
`scm_primitive_load` (*filename*) [C Function]
Load the file named *filename* and evaluate its contents in the top-level environment. *filename* must either be a full pathname or be a pathname relative to the current directory. If the variable `%load-hook` is defined, it should be bound to a procedure that will be called before any code is loaded. See the documentation for `%load-hook` later in this section.

`SCM scm_c_primitive_load` (*const char *filename*) [C Function]
`scm_primitive_load`, but taking a C string instead of an `SCM`.

`current-reader` [Variable]
`current-reader` holds the read procedure that is currently being used by the above loading procedures to read expressions (from the file that they are loading). `current-reader` is a fluid, so it has an independent value in each dynamic root and should be read and set using `fluid-ref` and `fluid-set!` (see Section 6.13.11 [Fluids and Dynamic States], page 323).

Changing `current-reader` is typically useful to introduce local syntactic changes, such that code following the `fluid-set!` call is read using the newly installed reader. The `current-reader` change should take place at evaluation time when the code is evaluated, or at compilation time when the code is compiled:

```
(eval-when (compile eval)
```

```
(fluid-set! current-reader my-own-reader))
```

The `eval-when` form above ensures that the `current-reader` change occurs at the right time.

`%load-hook` [Variable]

A procedure to be called (`%load-hook` *filename*) whenever a file is loaded, or `#f` for no such call. `%load-hook` is used by all of the loading functions (`load` and `primitive-load`, and `load-from-path` and `primitive-load-path` documented in the next section).

For example an application can set this to show what's loaded,

```
(set! %load-hook (lambda (filename)
                   (format #t "Loading ~a ...\n" filename)))
(load-from-path "foo.scm")
⊣ Loading /usr/local/share/guile/site/foo.scm ...
```

`current-load-port` [Scheme Procedure]
`scm_current_load_port ()` [C Function]

Return the current-load-port. The load port is used internally by `primitive-load`.

6.18.7 Load Paths

The procedure in the previous section look for Scheme code in the file system at specific location. Guile also has some procedures to search the load path for code.

`%load-path` [Variable]

List of directories which should be searched for Scheme modules and libraries. When Guile starts up, `%load-path` is initialized to the default load path (`list` (`%library-dir`) (`%site-dir`) (`%global-site-dir`) (`%package-data-dir`)). The `GUILE_LOAD_PATH` environment variable can be used to prepend or append additional directories (see Section 4.2.2 [Environment Variables], page 38).

See Section 6.23.1 [Build Config], page 456, for more on `%site-dir` and related procedures.

`load-from-path` *filename* [Scheme Procedure]

Similar to `load`, but searches for *filename* in the load paths. Preferentially loads a compiled version of the file, if it is available and up-to-date.

A user can extend the load path by calling `add-to-load-path`.

`add-to-load-path` *dir* [Scheme Syntax]

Add *dir* to the load path.

For example, a script might include this form to add the directory that it is in to the load path:

```
(add-to-load-path (dirname (current-filename)))
```

It's better to use `add-to-load-path` than to modify `%load-path` directly, because `add-to-load-path` takes care of modifying the path both at compile-time and at run-time.

`primitive-load-path` *filename* [*exception-on-not-found*] [Scheme Procedure]
`scm_primitive_load_path` (*filename*) [C Function]

Search `%load-path` for the file named *filename* and load it into the top-level environment. If *filename* is a relative pathname and is not found in the list of search paths, an error is signalled. Preferentially loads a compiled version of the file, if it is available and up-to-date.

If *filename* is a relative pathname and is not found in the list of search paths, one of three things may happen, depending on the optional second argument, *exception-on-not-found*. If it is `#f`, `#f` will be returned. If it is a procedure, it will be called with no arguments. (This allows a distinction to be made between exceptions raised by loading a file, and exceptions related to the loader itself.) Otherwise an error is signalled.

For compatibility with Guile 1.8 and earlier, the C function takes only one argument, which can be either a string (the file name) or an argument list.

`%search-load-path` *filename* [Scheme Procedure]
`scm_sys_search_load_path` (*filename*) [C Function]

Search `%load-path` for the file named *filename*, which must be readable by the current user. If *filename* is found in the list of paths to search or is an absolute pathname, return its full pathname. Otherwise, return `#f`. Filenames may have any of the optional extensions in the `%load-extensions` list; `%search-load-path` will try each extension automatically.

`%load-extensions` [Variable]

A list of default file extensions for files containing Scheme code. `%search-load-path` tries each of these extensions when looking for a file to load. By default, `%load-extensions` is bound to the list `("" ".scm")`.

As mentioned above, when Guile searches the `%load-path` for a source file, it will also search the `%load-compiled-path` for a corresponding compiled file. If the compiled file is as new or newer than the source file, it will be loaded instead of the source file, using `load-compiled`.

`%load-compiled-path` [Variable]

Like `%load-path`, but for compiled files. By default, this path has two entries: one for compiled files from Guile itself, and one for site packages. The `GUILE_LOAD_COMPILED_PATH` environment variable can be used to prepend or append additional directories (see Section 4.2.2 [Environment Variables], page 38).

When `primitive-load-path` searches the `%load-compiled-path` for a corresponding compiled file for a relative path it does so by appending `.go` to the relative path. For example, searching for `ice-9/popen` could find `/usr/lib/guile/3.0/ccache/ice-9/popen.go`, and use it instead of `/usr/share/guile/3.0/ice-9/popen.scm`.

If `primitive-load-path` does not find a corresponding `.go` file in the `%load-compiled-path`, or the `.go` file is out of date, it will search for a corresponding auto-compiled file in the fallback path, possibly creating one if one does not exist.

See Section 4.7 [Installing Site Packages], page 57, for more on how to correctly install site packages. See Section 6.20.4 [Modules and the File System], page 416, for more on the

relationship between load paths and modules. See Section 6.18.5 [Compilation], page 389, for more on the fallback path and auto-compilation.

Finally, there are a couple of helper procedures for general path manipulation.

`parse-path` *path* [*tail*] [Scheme Procedure]
`scm_parse_path` (*path, tail*) [C Function]
Parse *path*, which is expected to be a colon-separated string, into a list and return the resulting list with *tail* appended. If *path* is `#f`, *tail* is returned.

`parse-path-with-ellipsis` *path base* [Scheme Procedure]
`scm_parse_path_with_ellipsis` (*path, base*) [C Function]
Parse *path*, which is expected to be a colon-separated string, into a list and return the resulting list with *base* (a list) spliced in place of the `...` path component, if present, or else *base* is added to the end. If *path* is `#f`, *base* is returned.

`search-path` *path filename* [*extensions* [*require-exts?*]] [Scheme Procedure]
`scm_search_path` (*path, filename, rest*) [C Function]
Search *path* for a directory containing a file named *filename*. The file must be readable, and not a directory. If we find one, return its full filename; otherwise, return `#f`. If *filename* is absolute, return it unchanged. If given, *extensions* is a list of strings; for each directory in *path*, we search for *filename* concatenated with each *extension*. If *require-exts?* is true, require that the returned file name have one of the given extensions; if *require-exts?* is not given, it defaults to `#f`.

For compatibility with Guile 1.8 and earlier, the C function takes only three arguments.

6.18.8 Character Encoding of Source Files

Scheme source code files are usually encoded in ASCII or UTF-8, but the built-in reader can interpret other character encodings as well. When Guile loads Scheme source code, it uses the `file-encoding` procedure (described below) to try to guess the encoding of the file. In the absence of any hints, UTF-8 is assumed. One way to provide a hint about the encoding of a source file is to place a coding declaration in the top 500 characters of the file.

A coding declaration has the form `coding: XXXXXX`, where `XXXXXX` is the name of a character encoding in which the source code file has been encoded. The coding declaration must appear in a scheme comment. It can either be a semicolon-initiated comment, or the first block `#!` comment in the file.

The name of the character encoding in the coding declaration is typically lower case and containing only letters, numbers, and hyphens, as recognized by `set-port-encoding!` (see Section 6.14.1 [Ports], page 331). Common examples of character encoding names are `utf-8` and `iso-8859-1`, as defined by IANA (`http://www.iana.org/assignments/character-sets`). Thus, the coding declaration is mostly compatible with Emacs.

However, there are some differences in encoding names recognized by Emacs and encoding names defined by IANA, the latter being essentially a subset of the former. For instance, `latin-1` is a valid encoding name for Emacs, but it's not according to the IANA standard, which Guile follows; instead, you should use `iso-8859-1`, which is both understood by

Emacs and dubbed by IANA (IANA writes it uppercase but Emacs wants it lowercase and Guile is case insensitive.)

For source code, only a subset of all possible character encodings can be interpreted by the built-in source code reader. Only those character encodings in which ASCII text appears unmodified can be used. This includes `UTF-8` and `ISO-8859-1` through `ISO-8859-15`. The multi-byte character encodings `UTF-16` and `UTF-32` may not be used because they are not compatible with ASCII.

There might be a scenario in which one would want to read non-ASCII code from a port, such as with the function `read`, instead of with `load`. If the port's character encoding is the same as the encoding of the code to be read by the port, not other special handling is necessary. The port will automatically do the character encoding conversion. The functions `setlocale` or by `set-port-encoding!` are used to set port encodings (see Section 6.14.1 [Ports], page 331).

If a port is used to read code of unknown character encoding, it can accomplish this in three steps. First, the character encoding of the port should be set to ISO-8859-1 using `set-port-encoding!`. Then, the procedure `file-encoding`, described below, is used to scan for a coding declaration when reading from the port. As a side effect, it rewinds the port after its scan is complete. After that, the port's character encoding should be set to the encoding returned by `file-encoding`, if any, again by using `set-port-encoding!`. Then the code can be read as normal.

Alternatively, one can use the `#:guess-encoding` keyword argument of `open-file` and related procedures. See Section 6.14.10.1 [File Ports], page 344.

`file-encoding` *port* [Scheme Procedure]
`scm_file_encoding` (*port*) [C Function]

Attempt to scan the first few hundred bytes from the *port* for hints about its character encoding. Return a string containing the encoding name or `#f` if the encoding cannot be determined. The port is rewound.

Currently, the only supported method is to look for an Emacs-like character coding declaration (see Section "Recognize Coding" in *The GNU Emacs Reference Manual*). The coding declaration is of the form `coding: XXXXX` and must appear in a Scheme comment. Additional heuristics may be added in the future.

6.18.9 Delayed Evaluation

Promises are a convenient way to defer a calculation until its result is actually needed, and to run such a calculation only once. Also see Section 7.5.31 [SRFI-45], page 646.

`delay` *expr* [syntax]

Return a promise object which holds the given *expr* expression, ready to be evaluated by a later `force`.

`promise?` *obj* [Scheme Procedure]
`scm_promise_p` (*obj*) [C Function]

Return true if *obj* is a promise.

`force` *p* [Scheme Procedure]
`scm_force` (*p*) [C Function]

Return the value obtained from evaluating the *expr* in the given promise *p*. If *p* has previously been forced then its *expr* is not evaluated again, instead the value obtained at that time is simply returned.

During a `force`, an *expr* can call `force` again on its own promise, resulting in a recursive evaluation of that *expr*. The first evaluation to return gives the value for the promise. Higher evaluations run to completion in the normal way, but their results are ignored, `force` always returns the first value.

6.18.10 Local Evaluation

Guile includes a facility to capture a lexical environment, and later evaluate a new expression within that environment. This code is implemented in a module.

```
(use-modules (ice-9 local-eval))
```

`the-environment` [syntax]

Captures and returns a lexical environment for use with `local-eval` or `local-compile`.

`local-eval` *exp env* [Scheme Procedure]
`scm_local_eval` (*exp*, *env*) [C Function]
`local-compile` *exp env* [*opts*=()] [Scheme Procedure]

Evaluate or compile the expression *exp* in the lexical environment *env*.

Here is a simple example, illustrating that it is the variable that gets captured, not just its value at one point in time.

```
(define e (let ((x 100)) (the-environment)))
(define fetch-x (local-eval '(lambda () x) e))
(fetch-x)
⇒ 100
(local-eval '(set! x 42) e)
(fetch-x)
⇒ 42
```

While *exp* is evaluated within the lexical environment of `(the-environment)`, it has the dynamic environment of the call to `local-eval`.

`local-eval` and `local-compile` can only evaluate expressions, not definitions.

```
(local-eval '(define foo 42)
            (let ((x 100)) (the-environment)))
⇒ syntax error: definition in expression context
```

Note that the current implementation of `(the-environment)` only captures "normal" lexical bindings, and pattern variables bound by `syntax-case`. It does not currently capture local syntax transformers bound by `let-syntax`, `letrec-syntax` or non-top-level `define-syntax` forms. Any attempt to reference such captured syntactic keywords via `local-eval` or `local-compile` produces an error.

6.18.11 Local Inclusion

This section has discussed various means of linking Scheme code together: fundamentally, loading up files at run-time using `load` and `load-compiled`. Guile provides another option to compose parts of programs together at expansion-time instead of at run-time.

`include` *file-name* [Scheme Syntax]

Open *file-name*, at expansion-time, and read the Scheme forms that it contains, splicing them into the location of the `include`, within a `begin`.

If *file-name* is a relative path, it is searched for relative to the path that contains the file that the `include` form appears in.

If you are a C programmer, if `load` in Scheme is like `dlopen` in C, consider `include` to be like the C preprocessor's `#include`. When you use `include`, it is as if the contents of the included file were typed in instead of the `include` form.

Because the code is included at compile-time, it is available to the macroexpander. Syntax definitions in the included file are available to later code in the form in which the `include` appears, without the need for `eval-when`. (See Section 6.10.8 [Eval When], page 279.)

For the same reason, compiling a form that uses `include` results in one compilation unit, composed of multiple files. Loading the compiled file is one `stat` operation for the compilation unit, instead of `2*n` in the case of `load` (once for each loaded source file, and once each corresponding compiled file, in the best case).

Unlike `load`, `include` also works within nested lexical contexts. It so happens that the optimizer works best within a lexical context, because all of the uses of bindings in a lexical context are visible, so composing files by including them within a `(let () ...)` can sometimes lead to important speed improvements.

On the other hand, `include` does have all the disadvantages of early binding: once the code with the `include` is compiled, no change to the included file is reflected in the future behavior of the including form.

Also, the particular form of `include`, which requires an absolute path, or a path relative to the current directory at compile-time, is not very amenable to compiling the source in one place, but then installing the source to another place. For this reason, Guile provides another form, `include-from-path`, which looks for the source file to include within a load path.

`include-from-path` *file-name* [Scheme Syntax]

Like `include`, but instead of expecting `file-name` to be an absolute file name, it is expected to be a relative path to search in the `%load-path`.

`include-from-path` is more useful when you want to install all of the source files for a package (as you should!). It makes it possible to evaluate an installed file from source, instead of relying on the `.go` file being up to date.

6.18.12 Sandboxed Evaluation

Sometimes you would like to evaluate code that comes from an untrusted party. The safest way to do this is to buy a new computer, evaluate the code on that computer, then throw

the machine away. However if you are unwilling to take this simple approach, Guile does include a limited "sandbox" facility that can allow untrusted code to be evaluated with some confidence.

To use the sandboxed evaluator, load its module:

```
(use-modules (ice-9 sandbox))
```

Guile's sandboxing facility starts with the ability to restrict the time and space used by a piece of code.

`call-with-time-limit` *limit thunk limit-reached* [Scheme Procedure]
Call *thunk*, but cancel it if *limit* seconds of wall-clock time have elapsed. If the computation is cancelled, call *limit-reached* in tail position. *thunk* must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.

`call-with-allocation-limit` *limit thunk limit-reached* [Scheme Procedure]
Call *thunk*, but cancel it if *limit* bytes have been allocated. If the computation is cancelled, call *limit-reached* in tail position. *thunk* must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.

This limit applies to both stack and heap allocation. The computation will not be aborted before *limit* bytes have been allocated, but for the heap allocation limit, the check may be postponed until the next garbage collection.

Note that as a current shortcoming, the heap size limit applies to all threads; concurrent allocation by other unrelated threads counts towards the allocation limit.

`call-with-time-and-allocation-limits` *time-limit allocation-limit thunk* [Scheme Procedure]
Invoke *thunk* in a dynamic extent in which its execution is limited to *time-limit* seconds of wall-clock time, and its allocation to *allocation-limit* bytes. *thunk* must not disable interrupts or prevent an abort via a `dynamic-wind` unwind handler.

If successful, return all values produced by invoking *thunk*. Any uncaught exception thrown by the thunk will propagate out. If the time or allocation limit is exceeded, an exception will be thrown to the `limit-exceeded` key.

The time limit and stack limit are both very precise, but the heap limit only gets checked asynchronously, after a garbage collection. In particular, if the heap is already very large, the number of allocated bytes between garbage collections will be large, and therefore the precision of the check is reduced.

Additionally, due to the mechanism used by the allocation limit (the `after-gc-hook`), large single allocations like `(make-vector #e1e7)` are only detected after the allocation completes, even if the allocation itself causes garbage collection. It's possible therefore for user code to not only exceed the allocation limit set, but also to exhaust all available memory, causing out-of-memory conditions at any allocation site. Failure to allocate memory in Guile itself should be safe and cause an exception to be thrown, but most systems are not designed to handle `malloc` failures. An allocation failure may therefore exercise unexpected code paths in your system, so it is a weakness of the sandbox (and therefore an interesting point of attack).

The main sandbox interface is `eval-in-sandbox`.

`eval-in-sandbox` *exp* [*#:time-limit 0.1*] [*#:allocation-limit #e10e6*] [*#:bindings all-pure-bindings*] [*#:module (make-sandbox-module bindings)*] [*#:sever-module? #t*] [Scheme Procedure]

Evaluate the Scheme expression *exp* within an isolated "sandbox". Limit its execution to *time-limit* seconds of wall-clock time, and limit its allocation to *allocation-limit* bytes.

The evaluation will occur in *module*, which defaults to the result of calling `make-sandbox-module` on *bindings*, which itself defaults to `all-pure-bindings`. This is the core of the sandbox: creating a scope for the expression that is *safe*.

A safe sandbox module has two characteristics. Firstly, it will not allow the expression being evaluated to avoid being cancelled due to time or allocation limits. This ensures that the expression terminates in a timely fashion.

Secondly, a safe sandbox module will prevent the evaluation from receiving information from previous evaluations, or from affecting future evaluations. All combinations of binding sets exported by `(ice-9 sandbox)` form safe sandbox modules.

The *bindings* should be given as a list of import sets. One import set is a list whose car names an interface, like `(ice-9 q)`, and whose cdr is a list of imports. An import is either a bare symbol or a pair of `(out . in)`, where *out* and *in* are both symbols and denote the name under which a binding is exported from the module, and the name under which to make the binding available, respectively. Note that *bindings* is only used as an input to the default initializer for the *module* argument; if you pass `#:module`, *bindings* is unused. If *sever-module?* is true (the default), the module will be unlinked from the global module tree after the evaluation returns, to allow *mod* to be garbage-collected.

If successful, return all values produced by *exp*. Any uncaught exception thrown by the expression will propagate out. If the time or allocation limit is exceeded, an exception will be thrown to the `limit-exceeded` key.

Constructing a safe sandbox module is tricky in general. Guile defines an easy way to construct safe modules from predefined sets of bindings. Before getting to that interface, here are some general notes on safety.

1. The time and allocation limits rely on the ability to interrupt and cancel a computation. For this reason, no binding included in a sandbox module should be able to indefinitely postpone interrupt handling, nor should a binding be able to prevent an abort. In practice this second consideration means that `dynamic-wind` should not be included in any binding set.
2. The time and allocation limits apply only to the `eval-in-sandbox` call. If the call returns a procedure which is later called, no limit is "automatically" in place. Users of `eval-in-sandbox` have to be very careful to reimpose limits when calling procedures that escape from sandboxes.
3. Similarly, the dynamic environment of the `eval-in-sandbox` call is not necessarily in place when any procedure that escapes from the sandbox is later called.

   This detail prevents us from exposing `primitive-eval` to the sandbox, for two reasons. The first is that it's possible for legacy code to forge references to any binding, if the `allow-legacy-syntax-objects?` parameter is true. The default for this parameter is

true; see Section 6.10.4 [Syntax Transformer Helpers], page 273, for the details. The parameter is bound to `#f` for the duration of the `eval-in-sandbox` call itself, but that will not be in place during calls to escaped procedures.

The second reason we don't expose `primitive-eval` is that `primitive-eval` implicitly works in the current module, which for an escaped procedure will probably be different than the module that is current for the `eval-in-sandbox` call itself.

The common denominator here is that if an interface exposed to the sandbox relies on dynamic environments, it is easy to mistakenly grant the sandboxed procedure additional capabilities in the form of bindings that it should not have access to. For this reason, the default sets of predefined bindings do not depend on any dynamically scoped value.

4. Mutation may allow a sandboxed evaluation to break some invariant in users of data supplied to it. A lot of code culturally doesn't expect mutation, but if you hand mutable data to a sandboxed evaluation and you also grant mutating capabilities to that evaluation, then the sandboxed code may indeed mutate that data. The default set of bindings to the sandbox do not include any mutating primitives.

   Relatedly, `set!` may allow a sandbox to mutate a primitive, invalidating many system-wide invariants. Guile is currently quite permissive when it comes to imported bindings and mutability. Although `set!` to a module-local or lexically bound variable would be fine, we don't currently have an easy way to disallow `set!` to an imported binding, so currently no binding set includes `set!`.

5. Mutation may allow a sandboxed evaluation to keep state, or make a communication mechanism with other code. On the one hand this sounds cool, but on the other hand maybe this is part of your threat model. Again, the default set of bindings doesn't include mutating primitives, preventing sandboxed evaluations from keeping state.

6. The sandbox should probably not be able to open a network connection, or write to a file, or open a file from disk. The default binding set includes no interaction with the operating system.

If you, dear reader, find the above discussion interesting, you will enjoy Jonathan Rees' dissertation, "A Security Kernel Based on the Lambda Calculus".

`all-pure-bindings` [Scheme Variable]

All "pure" bindings that together form a safe subset of those bindings available by default to Guile user code.

`all-pure-and-impure-bindings` [Scheme Variable]

Like `all-pure-bindings`, but additionally including mutating primitives like `vector-set!`. This set is still safe in the sense mentioned above, with the caveats about mutation.

The components of these composite sets are as follows:

`alist-bindings` [Scheme Variable]
`array-bindings` [Scheme Variable]
`bit-bindings` [Scheme Variable]
`bitvector-bindings` [Scheme Variable]

`char-bindings` [Scheme Variable]
`char-set-bindings` [Scheme Variable]
`clock-bindings` [Scheme Variable]
`core-bindings` [Scheme Variable]
`error-bindings` [Scheme Variable]
`fluid-bindings` [Scheme Variable]
`hash-bindings` [Scheme Variable]
`iteration-bindings` [Scheme Variable]
`keyword-bindings` [Scheme Variable]
`list-bindings` [Scheme Variable]
`macro-bindings` [Scheme Variable]
`nil-bindings` [Scheme Variable]
`number-bindings` [Scheme Variable]
`pair-bindings` [Scheme Variable]
`predicate-bindings` [Scheme Variable]
`procedure-bindings` [Scheme Variable]
`promise-bindings` [Scheme Variable]
`prompt-bindings` [Scheme Variable]
`regexp-bindings` [Scheme Variable]
`sort-bindings` [Scheme Variable]
`srfi-4-bindings` [Scheme Variable]
`string-bindings` [Scheme Variable]
`symbol-bindings` [Scheme Variable]
`unspecified-bindings` [Scheme Variable]
`variable-bindings` [Scheme Variable]
`vector-bindings` [Scheme Variable]
`version-bindings` [Scheme Variable]

The components of `all-pure-bindings`.

`mutating-alist-bindings` [Scheme Variable]
`mutating-array-bindings` [Scheme Variable]
`mutating-bitvector-bindings` [Scheme Variable]
`mutating-fluid-bindings` [Scheme Variable]
`mutating-hash-bindings` [Scheme Variable]
`mutating-list-bindings` [Scheme Variable]
`mutating-pair-bindings` [Scheme Variable]
`mutating-sort-bindings` [Scheme Variable]
`mutating-srfi-4-bindings` [Scheme Variable]
`mutating-string-bindings` [Scheme Variable]
`mutating-variable-bindings` [Scheme Variable]
`mutating-vector-bindings` [Scheme Variable]

The additional components of `all-pure-and-impure-bindings`.

Finally, what do you do with a binding set? What is a binding set anyway? `make-sandbox-module` is here for you.

`make-sandbox-module` *bindings* [Scheme Procedure]

Return a fresh module that only contains *bindings*.

The *bindings* should be given as a list of import sets. One import set is a list whose car names an interface, like `(ice-9 q)`, and whose cdr is a list of imports. An import is either a bare symbol or a pair of `(out . in)`, where *out* and *in* are both symbols and denote the name under which a binding is exported from the module, and the name under which to make the binding available, respectively.

So you see that binding sets are just lists, and `all-pure-and-impure-bindings` is really just the result of appending all of the component binding sets.

6.18.13 REPL Servers

The procedures in this section are provided by

```
(use-modules (system repl server))
```

When an application is written in Guile, it is often convenient to allow the user to be able to interact with it by evaluating Scheme expressions in a REPL.

The procedures of this module allow you to spawn a *REPL server*, which permits interaction over a local or TCP connection. Guile itself uses them internally to implement the `--listen` switch, Section 4.2.1 [Command-line Options], page 35.

`make-tcp-server-socket` [*#:host=#f*] [*#:addr*] [*#:port=37146*] [Scheme Procedure]

Return a stream socket bound to a given address *addr* and port number *port*. If the *host* is given, and *addr* is not, then the *host* string is converted to an address. If neither is given, we use the loopback address.

`make-unix-domain-server-socket` [*#:path="/tmp/guile-socket"*] [Scheme Procedure]

Return a UNIX domain socket, bound to a given *path*.

`run-server` [*server-socket*] [Scheme Procedure]

`spawn-server` [*server-socket*] [Scheme Procedure]

Create and run a REPL, making it available over the given *server-socket*. If *server-socket* is not provided, it defaults to the socket created by calling `make-tcp-server-socket` with no arguments.

`run-server` runs the server in the current thread, whereas `spawn-server` runs the server in a new thread.

`stop-server-and-clients!` [Scheme Procedure]

Closes the connection on all running server sockets.

Please note that in the current implementation, the REPL threads are cancelled without unwinding their stacks. If any of them are holding mutexes or are within a critical section, the results are unspecified.

6.18.14 Cooperative REPL Servers

The procedures in this section are provided by

```
(use-modules (system repl coop-server))
```

Whereas ordinary REPL servers run in their own threads (see Section 6.18.13 [REPL Servers], page 403), sometimes it is more convenient to provide REPLs that run at specified

times within an existing thread, for example in programs utilizing an event loop or in single-threaded programs. This allows for safe access and mutation of a program's data structures from the REPL, without concern for thread synchronization.

Although the REPLs are run in the thread that calls `spawn-coop-repl-server` and `poll-coop-repl-server`, dedicated threads are spawned so that the calling thread is not blocked. The spawned threads read input for the REPLs and to listen for new connections.

Cooperative REPL servers must be polled periodically to evaluate any pending expressions by calling `poll-coop-repl-server` with the object returned from `spawn-coop-repl-server`. The thread that calls `poll-coop-repl-server` will be blocked for as long as the expression takes to be evaluated or if the debugger is entered.

`spawn-coop-repl-server` [*server-socket*] [Scheme Procedure]
: Create and return a new cooperative REPL server object, and spawn a new thread to listen for connections on *server-socket*. Proper functioning of the REPL server requires that `poll-coop-repl-server` be called periodically on the returned server object.

`poll-coop-repl-server` *coop-server* [Scheme Procedure]
: Poll the cooperative REPL server *coop-server* and apply a pending operation if there is one, such as evaluating an expression typed at the REPL prompt. This procedure must be called from the same thread that called `spawn-coop-repl-server`.

6.19 Memory Management and Garbage Collection

Guile uses a *garbage collector* to manage most of its objects. While the garbage collector is designed to be mostly invisible, you sometimes need to interact with it explicitly.

See Section 5.4.2 [Garbage Collection], page 67, for a general discussion of how garbage collection relates to using Guile from C.

6.19.1 Function related to Garbage Collection

`gc` [Scheme Procedure]
`scm_gc ()` [C Function]
: Finds all of the "live" `SCM` objects and reclaims for further use those that are no longer accessible. You normally don't need to call this function explicitly. Its functionality is invoked automatically as needed.

`SCM scm_gc_protect_object (`*SCM obj*`)` [C Function]
: Protects *obj* from being freed by the garbage collector, when it otherwise might be. When you are done with the object, call `scm_gc_unprotect_object` on the object. Calls to `scm_gc_protect_object`/`scm_gc_unprotect_object` can be nested, and the object remains protected until it has been unprotected as many times as it was protected. It is an error to unprotect an object more times than it has been protected. Returns the SCM object it was passed.

  Note that storing *obj* in a C global variable has the same effect[9].

[9] In Guile up to version 1.8, C global variables were not visited by the garbage collector in the mark phase; hence, `scm_gc_protect_object` was the only way in C to prevent a Scheme object from being freed.

`SCM scm_gc_unprotect_object (SCM obj)` [C Function]
: Unprotects an object from the garbage collector which was protected by `scm_gc_unprotect_object`. Returns the SCM object it was passed.

`SCM scm_permanent_object (SCM obj)` [C Function]
: Similar to `scm_gc_protect_object` in that it causes the collector to always mark the object, except that it should not be nested (only call `scm_permanent_object` on an object once), and it has no corresponding unpermanent function. Once an object is declared permanent, it will never be freed. Returns the SCM object it was passed.

`void scm_remember_upto_here_1 (SCM obj)` [C Macro]
`void scm_remember_upto_here_2 (SCM obj1, SCM obj2)` [C Macro]
: Create a reference to the given object or objects, so they're certain to be present on the stack or in a register and hence will not be freed by the garbage collector before this point.

  Note that these functions can only be applied to ordinary C local variables (ie. "automatics"). Objects held in global or static variables or some malloced block or the like cannot be protected with this mechanism.

`gc-stats` [Scheme Procedure]
`scm_gc_stats ()` [C Function]
: Return an association list of statistics about Guile's current use of storage.

`gc-live-object-stats` [Scheme Procedure]
`scm_gc_live_object_stats ()` [C Function]
: Return an alist of statistics of the current live objects.

`void scm_gc_mark (SCM x)` [Function]
: Mark the object *x*, and recurse on any objects *x* refers to. If *x*'s mark bit is already set, return immediately. This function must only be called during the mark-phase of garbage collection, typically from a smob *mark* function.

6.19.2 Memory Blocks

In C programs, dynamic management of memory blocks is normally done with the functions malloc, realloc, and free. Guile has additional functions for dynamic memory allocation that are integrated into the garbage collector and the error reporting system.

Memory blocks that are associated with Scheme objects (for example a foreign object) should be allocated with `scm_gc_malloc` or `scm_gc_malloc_pointerless`. These two functions will either return a valid pointer or signal an error. Memory blocks allocated this way may be released explicitly; however, this is not strictly needed, and we recommend *not* calling `scm_gc_free`. All memory allocated with `scm_gc_malloc` or `scm_gc_malloc_pointerless` is automatically reclaimed when the garbage collector no longer sees any live reference to it[10].

When garbage collection occurs, Guile will visit the words in memory allocated with `scm_gc_malloc`, looking for live pointers. This means that if `scm_gc_malloc`-allocated memory contains a pointer to some other part of the memory, the garbage collector notices

[10] In Guile up to version 1.8, memory allocated with `scm_gc_malloc` *had* to be freed with `scm_gc_free`.

it and prevents it from being reclaimed[11]. Conversely, memory allocated with `scm_gc_malloc_pointerless` is assumed to be "pointer-less" and is not scanned for pointers.

For memory that is not associated with a Scheme object, you can use `scm_malloc` instead of `malloc`. Like `scm_gc_malloc`, it will either return a valid pointer or signal an error. However, it will not assume that the new memory block can be freed by a garbage collection. The memory must be explicitly freed with `free`.

There is also `scm_gc_realloc` and `scm_realloc`, to be used in place of `realloc` when appropriate, and `scm_gc_calloc` and `scm_calloc`, to be used in place of `calloc` when appropriate.

The function `scm_dynwind_free` can be useful when memory should be freed with libc's `free` when leaving a dynwind context, See Section 6.13.10 [Dynamic Wind], page 320.

`void * scm_malloc (`*size_t* `size``)` [C Function]
`void * scm_calloc (`*size_t* `size``)` [C Function]
Allocate *size* bytes of memory and return a pointer to it. When *size* is 0, return `NULL`. When not enough memory is available, signal an error. This function runs the GC to free up some memory when it deems it appropriate.

The memory is allocated by the libc `malloc` function and can be freed with `free`. There is no `scm_free` function to go with `scm_malloc` to make it easier to pass memory back and forth between different modules.

The function `scm_calloc` is similar to `scm_malloc`, but initializes the block of memory to zero as well.

These functions will (indirectly) call `scm_gc_register_allocation`.

`void * scm_realloc (`*void *mem*, *size_t* `new_size``)` [C Function]
Change the size of the memory block at *mem* to *new_size* and return its new location. When *new_size* is 0, this is the same as calling `free` on *mem* and `NULL` is returned. When *mem* is `NULL`, this function behaves like `scm_malloc` and allocates a new block of size *new_size*.

When not enough memory is available, signal an error. This function runs the GC to free up some memory when it deems it appropriate.

This function will call `scm_gc_register_allocation`.

`void * scm_gc_malloc (`*size_t* `size`, *const char* *`what``)` [C Function]
`void * scm_gc_malloc_pointerless (`*size_t* `size`, *const char* *`what``)` [C Function]
`void * scm_gc_realloc (`*void *mem*, *size_t* `old_size`, *size_t* `new_size`, *const char* *`what`*);* [C Function]
`void * scm_gc_calloc (`*size_t* `size`, *const char* *`what``)` [C Function]
Allocate *size* bytes of automatically-managed memory. The memory is automatically freed when no longer referenced from any live memory block.

When garbage collection occurs, Guile will visit the words in memory allocated with `scm_gc_malloc` or `scm_gc_calloc`, looking for pointers to other memory allocations

[11] In Guile up to 1.8, memory allocated with `scm_gc_malloc` was *not* visited by the collector in the mark phase. Consequently, the GC had to be told explicitly about pointers to live objects contained in the memory block, e.g., *via* SMOB mark functions (see Section 6.8 [Smobs], page 245)

that are managed by the GC. In contrast, memory allocated by `scm_gc_malloc_pointerless` is not scanned for pointers.

The `scm_gc_realloc` call preserves the "pointerlessness" of the memory area pointed to by *mem*. Note that you need to pass the old size of a reallocated memory block as well. See below for a motivation.

`void scm_gc_free` (*void *mem, size_t size, const char *what*) [C Function]
Explicitly free the memory block pointed to by *mem*, which was previously allocated by one of the above `scm_gc` functions. This function is almost always unnecessary, except for codebases that still need to compile on Guile 1.8.

Note that you need to explicitly pass the *size* parameter. This is done since it should normally be easy to provide this parameter (for memory that is associated with GC controlled objects) and help keep the memory management overhead very low. However, in Guile 2.x, *size* is always ignored.

`void scm_gc_register_allocation` (*size_t size*) [C Function]
Informs the garbage collector that *size* bytes have been allocated, which the collector would otherwise not have known about.

In general, Scheme will decide to collect garbage only after some amount of memory has been allocated. Calling this function will make the Scheme garbage collector know about more allocation, and thus run more often (as appropriate).

It is especially important to call this function when large unmanaged allocations, like images, may be freed by small Scheme allocations, like foreign objects.

`void scm_dynwind_free` (*void *mem*) [C Function]
Equivalent to `scm_dynwind_unwind_handler (free, mem, SCM_F_WIND_EXPLICITLY)`. That is, the memory block at *mem* will be freed (using `free` from the C library) when the current dynwind is left.

`malloc-stats` [Scheme Procedure]
Return an alist ((*what* . *n*) ...) describing number of malloced objects. *what* is the second argument to `scm_gc_malloc`, *n* is the number of objects of that type currently allocated.

This function is only available if the `GUILE_DEBUG_MALLOC` preprocessor macro was defined when Guile was compiled.

6.19.3 Weak References

[FIXME: This chapter is based on Mikael Djurfeldt's answer to a question by Michael Livshin. Any mistakes are not theirs, of course.]

Weak references let you attach bookkeeping information to data so that the additional information automatically disappears when the original data is no longer in use and gets garbage collected. In a weak key hash, the hash entry for that key disappears as soon as the key is no longer referenced from anywhere else. For weak value hashes, the same happens as soon as the value is no longer in use. Entries in a doubly weak hash disappear when either the key or the value are not used anywhere else anymore.

Object properties offer the same kind of functionality as weak key hashes in many situations. (see Section 6.11.2 [Object Properties], page 285)

Here's an example (a little bit strained perhaps, but one of the examples is actually used in Guile):

Assume that you're implementing a debugging system where you want to associate information about filename and position of source code expressions with the expressions themselves.

Hashtables can be used for that, but if you use ordinary hash tables it will be impossible for the scheme interpreter to "forget" old source when, for example, a file is reloaded.

To implement the mapping from source code expressions to positional information it is necessary to use weak-key tables since we don't want the expressions to be remembered just because they are in our table.

To implement a mapping from source file line numbers to source code expressions you would use a weak-value table.

To implement a mapping from source code expressions to the procedures they constitute a doubly-weak table has to be used.

6.19.3.1 Weak hash tables

`make-weak-key-hash-table` [*size*] [Scheme Procedure]
`make-weak-value-hash-table` [*size*] [Scheme Procedure]
`make-doubly-weak-hash-table` [*size*] [Scheme Procedure]
`scm_make_weak_key_hash_table` (*size*) [C Function]
`scm_make_weak_value_hash_table` (*size*) [C Function]
`scm_make_doubly_weak_hash_table` (*size*) [C Function]

Return a weak hash table with *size* buckets. As with any hash table, choosing a good size for the table requires some caution.

You can modify weak hash tables in exactly the same way you would modify regular hash tables, with the exception of the routines that act on handles. Weak tables have a different implementation behind the scenes that doesn't have handles. see Section 6.6.22 [Hash Tables], page 238, for more on `hashq-ref` et al.

Note that in a weak-key hash table, the reference to the value is strong. This means that if the value references the key, even indirectly, the key will never be collected, which can lead to a memory leak. The reverse is true for weak value tables.

`weak-key-hash-table?` *obj* [Scheme Procedure]
`weak-value-hash-table?` *obj* [Scheme Procedure]
`doubly-weak-hash-table?` *obj* [Scheme Procedure]
`scm_weak_key_hash_table_p` (*obj*) [C Function]
`scm_weak_value_hash_table_p` (*obj*) [C Function]
`scm_doubly_weak_hash_table_p` (*obj*) [C Function]

Return `#t` if *obj* is the specified weak hash table. Note that a doubly weak hash table is neither a weak key nor a weak value hash table.

6.19.3.2 Weak vectors

`make-weak-vector` *size* [*fill*] [Scheme Procedure]

`scm_make_weak_vector` (*size*, *fill*) [C Function]
: Return a weak vector with *size* elements. If the optional argument *fill* is given, all entries in the vector will be set to *fill*. The default value for *fill* is the empty list.

`weak-vector` *elem ...* [Scheme Procedure]
`list->weak-vector` *l* [Scheme Procedure]
`scm_weak_vector` (*l*) [C Function]
: Construct a weak vector from a list: `weak-vector` uses the list of its arguments while `list->weak-vector` uses its only argument *l* (a list) to construct a weak vector the same way `list->vector` would.

`weak-vector?` *obj* [Scheme Procedure]
`scm_weak_vector_p` (*obj*) [C Function]
: Return `#t` if *obj* is a weak vector.

`weak-vector-ref` *wvect k* [Scheme Procedure]
`scm_weak_vector_ref` (*wvect*, *k*) [C Function]
: Return the *k*th element of the weak vector *wvect*, or `#f` if that element has been collected.

`weak-vector-set!` *wvect k elt* [Scheme Procedure]
`scm_weak_vector_set_x` (*wvect*, *k*, *elt*) [C Function]
: Set the *k*th element of the weak vector *wvect* to *elt*.

6.19.4 Guardians

Guardians provide a way to be notified about objects that would otherwise be collected as garbage. Guarding them prevents the objects from being collected and cleanup actions can be performed on them, for example.

See R. Kent Dybvig, Carl Bruggeman, and David Eby (1993) "Guardians in a Generation-Based Garbage Collector". ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1993.

`make-guardian` [Scheme Procedure]
`scm_make_guardian` () [C Function]
: Create a new guardian. A guardian protects a set of objects from garbage collection, allowing a program to apply cleanup or other actions.

  `make-guardian` returns a procedure representing the guardian. Calling the guardian procedure with an argument adds the argument to the guardian's set of protected objects. Calling the guardian procedure without an argument returns one of the protected objects which are ready for garbage collection, or `#f` if no such object is available. Objects which are returned in this way are removed from the guardian.

  You can put a single object into a guardian more than once and you can put a single object into more than one guardian. The object will then be returned multiple times by the guardian procedures.

  An object is eligible to be returned from a guardian when it is no longer referenced from outside any guardian.

  There is no guarantee about the order in which objects are returned from a guardian. If you want to impose an order on finalization actions, for example, you can do that

by keeping objects alive in some global data structure until they are no longer needed for finalizing other objects.

Being an element in a weak vector, a key in a hash table with weak keys, or a value in a hash table with weak values does not prevent an object from being returned by a guardian. But as long as an object can be returned from a guardian it will not be removed from such a weak vector or hash table. In other words, a weak link does not prevent an object from being considered collectable, but being inside a guardian prevents a weak link from being broken.

A key in a weak key hash table can be thought of as having a strong reference to its associated value as long as the key is accessible. Consequently, when the key is only accessible from within a guardian, the reference from the key to the value is also considered to be coming from within a guardian. Thus, if there is no other reference to the value, it is eligible to be returned from a guardian.

6.20 Modules

When programs become large, naming conflicts can occur when a function or global variable defined in one file has the same name as a function or global variable in another file. Even just a *similarity* between function names can cause hard-to-find bugs, since a programmer might type the wrong function name.

The approach used to tackle this problem is called *information encapsulation*, which consists of packaging functional units into a given name space that is clearly separated from other name spaces.

The language features that allow this are usually called *the module system* because programs are broken up into modules that are compiled separately (or loaded separately in an interpreter).

Older languages, like C, have limited support for name space manipulation and protection. In C a variable or function is public by default, and can be made local to a module with the `static` keyword. But you cannot reference public variables and functions from another module with different names.

More advanced module systems have become a common feature in recently designed languages: ML, Python, Perl, and Modula 3 all allow the *renaming* of objects from a foreign module, so they will not clutter the global name space.

In addition, Guile offers variables as first-class objects. They can be used for interacting with the module system.

6.20.1 General Information about Modules

A Guile module can be thought of as a collection of named procedures, variables and macros. More precisely, it is a set of *bindings* of symbols (names) to Scheme objects.

Within a module, all bindings are visible. Certain bindings can be declared *public*, in which case they are added to the module's so-called *export list*; this set of public bindings is called the module's *public interface* (see Section 6.20.3 [Creating Guile Modules], page 413).

A client module *uses* a providing module's bindings by either accessing the providing module's public interface, or by building a custom interface (and then accessing that). In a custom interface, the client module can *select* which bindings to access and can also

algorithmically *rename* bindings. In contrast, when using the providing module's public interface, the entire export list is available without renaming (see Section 6.20.2 [Using Guile Modules], page 411).

All Guile modules have a unique *module name*, for example `(ice-9 popen)` or `(srfi srfi-11)`. Module names are lists of one or more symbols.

When Guile goes to use an interface from a module, for example `(ice-9 popen)`, Guile first looks to see if it has loaded `(ice-9 popen)` for any reason. If the module has not been loaded yet, Guile searches a *load path* for a file that might define it, and loads that file.

The following subsections go into more detail on using, creating, installing, and otherwise manipulating modules and the module system.

6.20.2 Using Guile Modules

To use a Guile module is to access either its public interface or a custom interface (see Section 6.20.1 [General Information about Modules], page 410). Both types of access are handled by the syntactic form `use-modules`, which accepts one or more interface specifications and, upon evaluation, arranges for those interfaces to be available to the current module. This process may include locating and loading code for a given module if that code has not yet been loaded, following `%load-path` (see Section 6.20.4 [Modules and the File System], page 416).

An *interface specification* has one of two forms. The first variation is simply to name the module, in which case its public interface is the one accessed. For example:

```
(use-modules (ice-9 popen))
```

Here, the interface specification is `(ice-9 popen)`, and the result is that the current module now has access to `open-pipe`, `close-pipe`, `open-input-pipe`, and so on (see Section 7.2.10 [Pipes], page 529).

Note in the previous example that if the current module had already defined `open-pipe`, that definition would be overwritten by the definition in `(ice-9 popen)`. For this reason (and others), there is a second variation of interface specification that not only names a module to be accessed, but also selects bindings from it and renames them to suit the current module's needs. For example:

```
(use-modules ((ice-9 popen)
              #:select ((open-pipe . pipe-open) close-pipe)
              #:renamer (symbol-prefix-proc 'unixy:)))
```

or more simply:

```
(use-modules ((ice-9 popen)
              #:select ((open-pipe . pipe-open) close-pipe)
              #:prefix unixy:))
```

Here, the interface specification is more complex than before, and the result is that a custom interface with only two bindings is created and subsequently accessed by the current module. The mapping of old to new names is as follows:

```
(ice-9 popen) sees:           current module sees:
open-pipe                     unixy:pipe-open
close-pipe                    unixy:close-pipe
```

This example also shows how to use the convenience procedure `symbol-prefix-proc`.

You can also directly refer to bindings in a module by using the `@` syntax. For example, instead of using the `use-modules` statement from above and writing `unixy:pipe-open` to refer to the `pipe-open` from the `(ice-9 popen)`, you could also write `(@ (ice-9 popen) open-pipe)`. Thus an alternative to the complete `use-modules` statement would be

```
(define unixy:pipe-open (@ (ice-9 popen) open-pipe))
(define unixy:close-pipe (@ (ice-9 popen) close-pipe))
```

There is also `@@`, which can be used like `@`, but does not check whether the variable that is being accessed is actually exported. Thus, `@@` can be thought of as the impolite version of `@` and should only be used as a last resort or for debugging, for example.

Note that just as with a `use-modules` statement, any module that has not yet been loaded will be loaded when referenced by a `@` or `@@` form.

You can also use the `@` and `@@` syntaxes as the target of a `set!` when the binding refers to a variable.

`symbol-prefix-proc` *prefix-sym* [Scheme Procedure]

Return a procedure that prefixes its arg (a symbol) with *prefix-sym*.

`use-modules` *spec ...* [syntax]

Resolve each interface specification *spec* into an interface and arrange for these to be accessible by the current module. The return value is unspecified.

spec can be a list of symbols, in which case it names a module whose public interface is found and used.

spec can also be of the form:

```
(MODULE-NAME [#:select SELECTION]
             [#:prefix PREFIX]
             [#:renamer RENAMER])
```

in which case a custom interface is newly created and used. *module-name* is a list of symbols, as above; *selection* is a list of selection-specs; *prefix* is a symbol that is prepended to imported names; and *renamer* is a procedure that takes a symbol and returns its new name. A selection-spec is either a symbol or a pair of symbols `(ORIG . SEEN)`, where *orig* is the name in the used module and *seen* is the name in the using module. Note that *seen* is also modified by *prefix* and *renamer*.

The `#:select`, `#:prefix`, and `#:renamer` clauses are optional. If all are omitted, the returned interface has no bindings. If the `#:select` clause is omitted, *prefix* and *renamer* operate on the used module's public interface.

In addition to the above, *spec* can also include a `#:version` clause, of the form:

```
#:version VERSION-SPEC
```

where *version-spec* is an R6RS-compatible version reference. An error will be signaled in the case in which a module with the same name has already been loaded, if that module specifies a version and that version is not compatible with *version-spec*. See Section 6.20.5 [R6RS Version References], page 416, for more on version references.

If the module name is not resolvable, `use-modules` will signal an error.

`@` *module-name binding-name* [syntax]

Refer to the binding named *binding-name* in module *module-name*. The binding must have been exported by the module.

`@@` *module-name binding-name* [syntax]

Refer to the binding named *binding-name* in module *module-name*. The binding must not have been exported by the module. This syntax is only intended for debugging purposes or as a last resort. See Section 6.20.9 [Declarative Modules], page 422, for some limitations on the use of `@@`.

6.20.3 Creating Guile Modules

When you want to create your own modules, you have to take the following steps:

- Create a Scheme source file and add all variables and procedures you wish to export, or which are required by the exported procedures.
- Add a `define-module` form at the beginning.
- Export all bindings which should be in the public interface, either by using `define-public` or `export` (both documented below).

`define-module` *module-name option ...* [syntax]

module-name is a list of one or more symbols.

```
(define-module (ice-9 popen))
```

`define-module` makes this module available to Guile programs under the given *module-name*.

option ... are keyword/value pairs which specify more about the defined module. The recognized options and their meaning are shown in the following table.

`#:use-module` *interface-specification*

Equivalent to a `(use-modules` *interface-specification*`)` (see Section 6.20.2 [Using Guile Modules], page 411).

`#:autoload` *module symbol-list*

Load *module* when any of *symbol-list* are accessed. For example,

```
(define-module (my mod)
  #:autoload (srfi srfi-1) (partition delete-duplicates))
...
(when something
  (set! foo (delete-duplicates ...)))
```

When a module is autoloaded, only the bindings in *symbol-list* become available[12].

An autoload is a good way to put off loading a big module until it's really needed, for instance for faster startup or if it will only be needed in certain circumstances.

`#:export` *list*

Export all identifiers in *list* which must be a list of symbols or pairs of symbols. This is equivalent to `(export` *list*`)` in the module body.

[12] In Guile 2.2 and earlier, *all* the module bindings would become available; *symbol-list* was just the list of bindings that will first trigger the load.

`#:re-export` *list*
: Re-export all identifiers in *list* which must be a list of symbols or pairs of symbols. The symbols in *list* must be imported by the current module from other modules. This is equivalent to `re-export` below.

`#:replace` *list*
: Export all identifiers in *list* (a list of symbols or pairs of symbols) and mark them as *replacing bindings*. In the module user's name space, this will have the effect of replacing any binding with the same name that is not also "replacing". Normally a replacement results in an "override" warning message, `#:replace` avoids that.

  In general, a module that exports a binding for which the `(guile)` module already has a definition should use `#:replace` instead of `#:export`. `#:replace`, in a sense, lets Guile know that the module *purposefully* replaces a core binding. It is important to note, however, that this binding replacement is confined to the name space of the module user. In other words, the value of the core binding in question remains unchanged for other modules.

  Note that although it is often a good idea for the replaced binding to remain compatible with a binding in `(guile)`, to avoid surprising the user, sometimes the bindings will be incompatible. For example, SRFI-19 exports its own version of `current-time` (see Section 7.5.16.2 [SRFI-19 Time], page 615) which is not compatible with the core `current-time` function (see Section 7.2.5 [Time], page 513). Guile assumes that a user importing a module knows what she is doing, and uses `#:replace` for this binding rather than `#:export`.

  A `#:replace` clause is equivalent to `(export!` *list*`)` in the module body.

  The `#:duplicates` (see below) provides fine-grain control about duplicate binding handling on the module-user side.

`#:re-export-and-replace` *list*
: Like `#:re-export`, but also marking the bindings as replacements in the sense of `#:replace`.

`#:version` *list*
: Specify a version for the module in the form of *list*, a list of zero or more exact, nonnegative integers. The corresponding `#:version` option in the `use-modules` form allows callers to restrict the value of this option in various ways.

`#:duplicates` *list*
: Tell Guile to handle duplicate bindings for the bindings imported by the current module according to the policy defined by *list*, a list of symbols. *list* must contain symbols representing a duplicate binding handling policy chosen among the following:

  `check`
  : Raises an error when a binding is imported from more than one place.

`warn` Issue a warning when a binding is imported from more than one place and leave the responsibility of actually handling the duplication to the next duplicate binding handler.

`replace` When a new binding is imported that has the same name as a previously imported binding, then do the following:

1. If the old binding was said to be *replacing* (via the `#:replace` option above) and the new binding is not replacing, the keep the old binding.
2. If the old binding was not said to be replacing and the new binding is replacing, then replace the old binding with the new one.
3. If neither the old nor the new binding is replacing, then keep the old one.

`warn-override-core`
Issue a warning when a core binding is being overwritten and actually override the core binding with the new one.

`first` In case of duplicate bindings, the firstly imported binding is always the one which is kept.

`last` In case of duplicate bindings, the lastly imported binding is always the one which is kept.

`noop` In case of duplicate bindings, leave the responsibility to the next duplicate handler.

If *list* contains more than one symbol, then the duplicate binding handlers which appear first will be used first when resolving a duplicate binding situation. As mentioned above, some resolution policies may explicitly leave the responsibility of handling the duplication to the next handler in *list*.

If GOOPS has been loaded before the `#:duplicates` clause is processed, there are additional strategies available for dealing with generic functions. See Section 8.6.3 [Merging Generics], page 782, for more information.

The default duplicate binding resolution policy is given by the `default-duplicate-binding-handler` procedure, and is

```
(replace warn-override-core warn last)
```

`#:pure` Create a *pure* module, that is a module which does not contain any of the standard procedure bindings except for the syntax forms. This is useful if you want to create *safe* modules, that is modules which do not know anything about dangerous procedures.

`export` *variable ...* [syntax]
Add all *variable*s (which must be symbols or pairs of symbols) to the list of exported bindings of the current module. If *variable* is a pair, its `car` gives the name of the variable as seen by the current module and its `cdr` specifies a name for the binding in the current module's public interface.

`define-public` ... [syntax]
Equivalent to `(begin (define foo ...) (export foo))`.

`re-export` *variable* ... [syntax]
Add all *variable*s (which must be symbols or pairs of symbols) to the list of re-exported bindings of the current module. Pairs of symbols are handled as in `export`. Re-exported bindings must be imported by the current module from some other module.

`export!` *variable* ... [syntax]
Like `export`, but marking the exported variables as replacing. Using a module with replacing bindings will cause any existing bindings to be replaced without issuing any warnings. See the discussion of `#:replace` above.

6.20.4 Modules and the File System

Typical programs only use a small subset of modules installed on a Guile system. In order to keep startup time down, Guile only loads modules when a program uses them, on demand.

When a program evaluates `(use-modules (ice-9 popen))`, and the module is not loaded, Guile searches for a conventionally-named file from in the *load path*.

In this case, loading `(ice-9 popen)` will eventually cause Guile to run `(primitive-load-path "ice-9/popen")`. `primitive-load-path` will search for a file `ice-9/popen` in the `%load-path` (see Section 6.18.7 [Load Paths], page 393). For each directory in `%load-path`, Guile will try to find the file name, concatenated with the extensions from `%load-extensions`. By default, this will cause Guile to `stat ice-9/popen.scm`, and then `ice-9/popen`. See Section 6.18.7 [Load Paths], page 393, for more on `primitive-load-path`.

If a corresponding compiled `.go` file is found in the `%load-compiled-path` or in the fallback path, and is as fresh as the source file, it will be loaded instead of the source file. If no compiled file is found, Guile may try to compile the source file and cache away the resulting `.go` file. See Section 6.18.5 [Compilation], page 389, for more on compilation.

Once Guile finds a suitable source or compiled file is found, the file will be loaded. If, after loading the file, the module under consideration is still not defined, Guile will signal an error.

For more information on where and how to install Scheme modules, See Section 4.7 [Installing Site Packages], page 57.

6.20.5 R6RS Version References

Guile's module system includes support for locating modules based on a declared version specifier of the same form as the one described in R6RS (see Section "Library form" in *The Revised^6 Report on the Algorithmic Language Scheme*). By using the `#:version` keyword in a `define-module` form, a module may specify a version as a list of zero or more exact, nonnegative integers.

This version can then be used to locate the module during the module search process. Client modules and callers of the `use-modules` function may specify constraints on the versions of target modules by providing a *version reference*, which has one of the following forms:

```
(sub-version-reference ...)
```

```
(and version-reference ...)
(or version-reference ...)
(not version-reference)
```

in which *sub-version-reference* is in turn one of:

```
(sub-version)
(>= sub-version)
(<= sub-version)
(and sub-version-reference ...)
(or sub-version-reference ...)
(not sub-version-reference)
```

in which *sub-version* is an exact, nonnegative integer as above. A version reference matches a declared module version if each element of the version reference matches a corresponding element of the module version, according to the following rules:

- The `and` sub-form matches a version or version element if every element in the tail of the sub-form matches the specified version or version element.
- The `or` sub-form matches a version or version element if any element in the tail of the sub-form matches the specified version or version element.
- The `not` sub-form matches a version or version element if the tail of the sub-form does not match the version or version element.
- The `>=` sub-form matches a version element if the element is greater than or equal to the *sub-version* in the tail of the sub-form.
- The `<=` sub-form matches a version element if the version is less than or equal to the *sub-version* in the tail of the sub-form.
- A *sub-version* matches a version element if one is *eqv?* to the other.

For example, a module declared as:

```
(define-module (mylib mymodule) #:version (1 2 0))
```

would be successfully loaded by any of the following `use-modules` expressions:

```
(use-modules ((mylib mymodule) #:version (1 2 (>= 0))))
(use-modules ((mylib mymodule) #:version (or (1 2 0) (1 2 1))))
(use-modules ((mylib mymodule) #:version ((and (>= 1) (not 2)) 2 0)))
```

6.20.6 R6RS Libraries

In addition to the API described in the previous sections, you also have the option to create modules using the portable `library` form described in R6RS (see Section "Library form" in *The Revised^6 Report on the Algorithmic Language Scheme*), and to import libraries created in this format by other programmers. Guile's R6RS library implementation takes advantage of the flexibility built into the module system by expanding the R6RS library form into a corresponding Guile `define-module` form that specifies equivalent import and export requirements and includes the same body expressions. The library expression:

```
(library (mylib (1 2))
  (export mybinding)
  (import (otherlib (3))))
```

is equivalent to the module definition:

```
(define-module (mylib)
```

```
#:version (1 2)
#:use-module ((otherlib) #:version (3))
#:export (mybinding))
```

Central to the mechanics of R6RS libraries is the concept of import and export *levels*, which control the visibility of bindings at various phases of a library's lifecycle — macros necessary to expand forms in the library's body need to be available at expand time; variables used in the body of a procedure exported by the library must be available at runtime. R6RS specifies the optional `for` sub-form of an *import set* specification (see below) as a mechanism by which a library author can indicate that a particular library import should take place at a particular phase with respect to the lifecycle of the importing library.

Guile's library implementation uses a technique called *implicit phasing* (first described by Abdulaziz Ghuloum and R. Kent Dybvig), which allows the expander and compiler to automatically determine the necessary visibility of a binding imported from another library. As such, the `for` sub-form described below is ignored by Guile (but may be required by Schemes in which phasing is explicit).

`library` *name* (*export export-spec ...*) (*import import-spec ...*) *body ...* [Scheme Syntax]

Defines a new library with the specified name, exports, and imports, and evaluates the specified body expressions in this library's environment.

The library *name* is a non-empty list of identifiers, optionally ending with a version specification of the form described above (see Section 6.20.3 [Creating Guile Modules], page 413).

Each *export-spec* is the name of a variable defined or imported by the library, or must take the form `(rename (internal-name external-name) ...)`, where the identifier *internal-name* names a variable defined or imported by the library and *external-name* is the name by which the variable is seen by importing libraries.

Each *import-spec* must be either an *import set* (see below) or must be of the form `(for import-set import-level ...)`, where each *import-level* is one of:

```
run
expand
(meta level)
```

where *level* is an integer. Note that since Guile does not require explicit phase specification, any *import-set*s found inside of `for` sub-forms will be "unwrapped" during expansion and processed as if they had been specified directly.

Import sets in turn take one of the following forms:

```
library-reference
(library library-reference)
(only import-set identifier ...)
(except import-set identifier ...)
(prefix import-set identifier)
(rename import-set (internal-identifier external-identifier) ...)
```

where *library-reference* is a non-empty list of identifiers ending with an optional version reference (see Section 6.20.5 [R6RS Version References], page 416), and the other sub-forms have the following semantics, defined recursively on nested *import-set*s:

- The `library` sub-form is used to specify libraries for import whose names begin with the identifier "library."
- The `only` sub-form imports only the specified *identifier*s from the given *import-set*.
- The `except` sub-form imports all of the bindings exported by *import-set* except for those that appear in the specified list of *identifier*s.
- The `prefix` sub-form imports all of the bindings exported by *import-set*, first prefixing them with the specified *identifier*.
- The `rename` sub-form imports all of the identifiers exported by *import-set*. The binding for each *internal-identifier* among these identifiers is made visible to the importing library as the corresponding *external-identifier*; all other bindings are imported using the names provided by *import-set*.

Note that because Guile translates R6RS libraries into module definitions, an import specification may be used to declare a dependency on a native Guile module — although doing so may make your libraries less portable to other Schemes.

`import` *import-spec ...* [Scheme Syntax]

Import into the current environment the libraries specified by the given import specifications, where each *import-spec* takes the same form as in the `library` form described above.

6.20.7 Variables

Each module has its own hash table, sometimes known as an *obarray*, that maps the names defined in that module to their corresponding variable objects.

A variable is a box-like object that can hold any Scheme value. It is said to be *undefined* if its box holds a special Scheme value that denotes undefined-ness (which is different from all other Scheme values, including for example `#f`); otherwise the variable is *defined*.

On its own, a variable object is anonymous. A variable is said to be *bound* when it is associated with a name in some way, usually a symbol in a module obarray. When this happens, the name is said to be bound to the variable, in that module.

(That's the theory, anyway. In practice, defined-ness and bound-ness sometimes get confused, because Lisp and Scheme implementations have often conflated — or deliberately drawn no distinction between — a name that is unbound and a name that is bound to a variable whose value is undefined. We will try to be clear about the difference and explain any confusion where it is unavoidable.)

Variables do not have a read syntax. Most commonly they are created and bound implicitly by `define` expressions: a top-level `define` expression of the form

```
(define name value)
```

creates a variable with initial value *value* and binds it to the name *name* in the current module. But they can also be created dynamically by calling one of the constructor procedures `make-variable` and `make-undefined-variable`.

`make-undefined-variable` [Scheme Procedure]
`scm_make_undefined_variable ()` [C Function]

Return a variable that is initially unbound.

`make-variable` *init* [Scheme Procedure]
`scm_make_variable` (*init*) [C Function]
Return a variable initialized to value *init*.

`variable-bound?` *var* [Scheme Procedure]
`scm_variable_bound_p` (*var*) [C Function]
Return `#t` if *var* is bound to a value, or `#f` otherwise. Throws an error if *var* is not a variable object.

`variable-ref` *var* [Scheme Procedure]
`scm_variable_ref` (*var*) [C Function]
Dereference *var* and return its value. *var* must be a variable object; see `make-variable` and `make-undefined-variable`.

`variable-set!` *var val* [Scheme Procedure]
`scm_variable_set_x` (*var*, *val*) [C Function]
Set the value of the variable *var* to *val*. *var* must be a variable object, *val* can be any value. Return an unspecified value.

`variable-unset!` *var* [Scheme Procedure]
`scm_variable_unset_x` (*var*) [C Function]
Unset the value of the variable *var*, leaving *var* unbound.

`variable?` *obj* [Scheme Procedure]
`scm_variable_p` (*obj*) [C Function]
Return `#t` if *obj* is a variable object, else return `#f`.

6.20.8 Module System Reflection

The previous sections have described a declarative view of the module system. You can also work with it programmatically by accessing and modifying various parts of the Scheme objects that Guile uses to implement the module system.

At any time, there is a *current module*. This module is the one where a top-level `define` and similar syntax will add new bindings. You can find other module objects with `resolve-module`, for example.

These module objects can be used as the second argument to `eval`.

`current-module` [Scheme Procedure]
`scm_current_module` () [C Function]
Return the current module object.

`set-current-module` *module* [Scheme Procedure]
`scm_set_current_module` (*module*) [C Function]
Set the current module to *module* and return the previous current module.

`save-module-excursion` *thunk* [Scheme Procedure]
Call *thunk* within a `dynamic-wind` such that the module that is current at invocation time is restored when *thunk*'s dynamic extent is left (see Section 6.13.10 [Dynamic Wind], page 320).

More precisely, if *thunk* escapes non-locally, the current module (at the time of escape) is saved, and the original current module (at the time *thunk*'s dynamic extent was last entered) is restored. If *thunk*'s dynamic extent is re-entered, then the current module is saved, and the previously saved inner module is set current again.

`resolve-module` *name* [*autoload=#t*] [*version=#f*] [*#:ensure=#t*] [Scheme Procedure]

`scm_resolve_module` (*name*) [C Function]

Find the module named *name* and return it. When it has not already been defined and *autoload* is true, try to auto-load it. When it can't be found that way either, create an empty module if *ensure* is true, otherwise return `#f`. If *version* is true, ensure that the resulting module is compatible with the given version reference (see Section 6.20.5 [R6RS Version References], page 416). The name is a list of symbols.

`resolve-interface` *name* [*#:select=#f*] [*#:hide='()*] [*#:prefix=#f*] [*#:renamer=#f*] [*#:version=#f*] [Scheme Procedure]

Find the module named *name* as with `resolve-module` and return its interface. The interface of a module is also a module object, but it contains only the exported bindings.

`module-uses` *module* [Scheme Procedure]

Return a list of the interfaces used by *module*.

`module-use!` *module interface* [Scheme Procedure]

Add *interface* to the front of the use-list of *module*. Both arguments should be module objects, and *interface* should very likely be a module returned by `resolve-interface`.

`reload-module` *module* [Scheme Procedure]

Revisit the source file that corresponds to *module*. Raises an error if no source file is associated with the given module.

As mentioned in the previous section, modules contain a mapping between identifiers (as symbols) and storage locations (as variables). Guile defines a number of procedures to allow access to this mapping. If you are programming in C, Section 6.20.10 [Accessing Modules from C], page 424.

`module-variable` *module name* [Scheme Procedure]

Return the variable bound to *name* (a symbol) in *module*, or `#f` if *name* is unbound.

`module-add!` *module name var* [Scheme Procedure]

Define a new binding between *name* (a symbol) and *var* (a variable) in *module*.

`module-ref` *module name* [Scheme Procedure]

Look up the value bound to *name* in *module*. Like `module-variable`, but also does a `variable-ref` on the resulting variable, raising an error if *name* is unbound.

`module-define!` *module name value* [Scheme Procedure]

Locally bind *name* to *value* in *module*. If *name* was already locally bound in *module*, i.e., defined locally and not by an imported module, the value stored in the existing variable will be updated. Otherwise, a new variable will be added to the module, via `module-add!`.

`module-set!` *module name value* [Scheme Procedure]
Update the binding of *name* in *module* to *value*, raising an error if *name* is not already bound in *module*.

There are many other reflective procedures available in the default environment. If you find yourself using one of them, please contact the Guile developers so that we can commit to stability for that interface.

6.20.9 Declarative Modules

The first-class access to modules and module variables described in the previous subsection is very powerful and allows Guile users to build many tools to dynamically learn things about their Guile systems. However, as Scheme godparent Mathias Felleisen wrote in "On the Expressive Power of Programming Languages", a more expressive language is necessarily harder to reason about. There are transformations that Guile's compiler would like to make which can't be done if every top-level definition is subject to mutation at any time.

Consider this module:

```
(define-module (boxes)
  #:export (make-box box-ref box-set! box-swap!))

(define (make-box x) (list x))
(define (box-ref box) (car box))
(define (box-set! box x) (set-car! box x))
(define (box-swap! box x)
  (let ((y (box-ref box)))
    (box-set! box x)
    y))
```

Ideally you'd like for the `box-ref` in `box-swap!` to be inlined to `car`. Guile's compiler can do this, but only if it knows that `box-ref`'s definition is what it appears to be in the text. However, in the general case it could be that a programmer could reach into the `(boxes)` module at any time and change the value of `box-ref`.

To allow Guile to reason about the values of top-levels from a module, a module can be marked as *declarative*. This flag applies only to the subset of top-level definitions that are themselves declarative: those that are defined within the compilation unit, and not assigned (`set!`) or redefined within the compilation unit.

To explicitly mark a module as being declarative, pass the `#:declarative?` keyword argument when declaring a module:

```
(define-module (boxes)
  #:export (make-box box-ref box-set! box-swap!)
  #:declarative? #t)
```

By default, modules are compiled declaratively if the `user-modules-declarative?` parameter is true when the module is compiled.

`user-modules-declarative?` [Scheme Parameter]
A boolean indicating whether definitions in modules created by `define-module` or implicitly as part of a compilation unit without an explicit module can be treated as declarative.

Because it's usually what you want, the default value of `user-modules-declarative?` is `#t`.

Should I Mark My Module As Declarative?

In the vast majority of use cases, declarative modules are what you want. However, there are exceptions.

Consider the `(boxes)` module above. Let's say you want to be able to go in and change the definition of `box-set!` at run-time:

```
scheme@(guile-user)> (use-modules (boxes))
scheme@(guile-user)> ,module boxes
scheme@(boxes)> (define (box-set! x y) (set-car! x (pk y)))
```

However, considering that `(boxes)` is a declarative module, it could be that `box-swap!` inlined the call to `box-set!` – so it may be that you are surprised if you call `(box-swap! x y)` and you don't see the new definition being used. (Note, however, that Guile has no guarantees about what definitions its compiler will or will not inline.)

If you want to allow the definition of `box-set!` to be changed and to have all of its uses updated, then probably the best option is to edit the module and reload the whole thing:

```
scheme@(guile-user)> ,reload (boxes)
```

The advantage of the reloading approach is that you maintain the optimizations that declarative modules enable, while also being able to live-update the code. If the module keeps precious program state, those definitions can be marked as `define-once` to prevent reloads from overwriting them. See Section 6.12.1 [Top Level], page 293, for more on `define-once`. Incidentally, `define-once` also prevents declarative-definition optimizations, so if there's a limited subset of redefinable bindings, `define-once` could be an interesting tool to mark those definitions as works-in-progress for interactive program development.

To users, whether a module is declarative or not is mostly immaterial: besides normal use via `use-modules`, users can reference and redefine public or private bindings programmatically or interactively. The only difference is that changing a declarative definition may not change all of its uses. If this use-case is important to you, and if reloading whole modules is insufficient, then you can mark all definitions in a module as non-declarative by adding `#:declarative? #f` to the module definition.

The default of whether modules are declarative or not can be controlled via the `(user-modules-declarative?)` parameter mentioned above, but care should be taken to set this parameter when the modules are compiled, e.g. via `(eval-when (expand) (user-modules-declarative? #f))`. See Section 6.10.8 [Eval When], page 279.

Alternately you can prevent declarative-definition optimizations by compiling at the `-O1` optimization level instead of the default `-O2`, or via explicitly passing `-Ono-letrectify` to the `guild compile` invocation. See Section 6.18.5 [Compilation], page 389, for more on compiler options.

One final note. Currently, definitions from declarative modules can only be inlined within the module they are defined in, and within a compilation unit. This may change in the future to allow Guile to inline imported declarative definitions as well (cross-module inlining). To Guile, whether a definition is inlinable or not is a property of the definition, not its use. We hope to improve compiler tooling in the future to allow the user to identify definitions that are out of date when a declarative binding is redefined.

6.20.10 Accessing Modules from C

The last sections have described how modules are used in Scheme code, which is the recommended way of creating and accessing modules. You can also work with modules from C, but it is more cumbersome.

The following procedures are available.

`SCM scm_c_call_with_current_module (SCM module, SCM (*func)(void *), void *data)` [C Function]

Call *func* and make *module* the current module during the call. The argument *data* is passed to *func*. The return value of `scm_c_call_with_current_module` is the return value of *func*.

`SCM scm_public_variable (SCM module_name, SCM name)` [C Function]
`SCM scm_c_public_variable (const char *module_name, const char *name)` [C Function]

Find a the variable bound to the symbol *name* in the public interface of the module named *module_name*.

module_name should be a list of symbols, when represented as a Scheme object, or a space-separated string, in the `const char *` case. See `scm_c_define_module` below, for more examples.

Signals an error if no module was found with the given name. If *name* is not bound in the module, just returns `#f`.

`SCM scm_private_variable (SCM module_name, SCM name)` [C Function]
`SCM scm_c_private_variable (const char *module_name, const char *name)` [C Function]

Like `scm_public_variable`, but looks in the internals of the module named *module_name* instead of the public interface. Logically, these procedures should only be called on modules you write.

`SCM scm_public_lookup (SCM module_name, SCM name)` [C Function]
`SCM scm_c_public_lookup (const char *module_name, const char *name)` [C Function]
`SCM scm_private_lookup (SCM module_name, SCM name)` [C Function]
`SCM scm_c_private_lookup (const char *module_name, const char *name)` [C Function]

Like `scm_public_variable` or `scm_private_variable`, but if the *name* is not bound in the module, signals an error. Returns a variable, always.

```
static SCM eval_string_var;

/* NOTE: It is important that the call to 'my_init'
   happens-before all calls to 'my_eval_string'. */
void my_init (void)
{
  eval_string_var = scm_c_public_lookup ("ice-9 eval-string",
                                         "eval-string");
}
```

```
SCM my_eval_string (SCM str)
{
  return scm_call_1 (scm_variable_ref (eval_string_var), str);
}
```

SCM scm_public_ref (*SCM module_name, SCM name*) [C Function]

SCM scm_c_public_ref (*const char *module_name, const char *name*) [C Function]

SCM scm_private_ref (*SCM module_name, SCM name*) [C Function]

SCM scm_c_private_ref (*const char *module_name, const char *name*) [C Function]

Like scm_public_lookup or scm_private_lookup, but additionally dereferences the variable. If the variable object is unbound, signals an error. Returns the value bound to *name* in *module_name*.

In addition, there are a number of other lookup-related procedures. We suggest that you use the scm_public_ and scm_private_ family of procedures instead, if possible.

SCM scm_c_lookup (*const char *name*) [C Function]

Return the variable bound to the symbol indicated by *name* in the current module. If there is no such binding or the symbol is not bound to a variable, signal an error.

SCM scm_lookup (*SCM name*) [C Function]

Like scm_c_lookup, but the symbol is specified directly.

SCM scm_c_module_lookup (*SCM module, const char *name*) [C Function]

SCM scm_module_lookup (*SCM module, SCM name*) [C Function]

Like scm_c_lookup and scm_lookup, but the specified module is used instead of the current one.

SCM scm_module_variable (*SCM module, SCM name*) [C Function]

Like scm_module_lookup, but if the binding does not exist, just returns #f instead of raising an error.

To define a value, use scm_define:

SCM scm_c_define (*const char *name, SCM val*) [C Function]

Bind the symbol indicated by *name* to a variable in the current module and set that variable to *val*. When *name* is already bound to a variable, use that. Else create a new variable.

SCM scm_define (*SCM name, SCM val*) [C Function]

Like scm_c_define, but the symbol is specified directly.

SCM scm_c_module_define (*SCM module, const char *name, SCM val*) [C Function]

SCM scm_module_define (*SCM module, SCM name, SCM val*) [C Function]

Like scm_c_define and scm_define, but the specified module is used instead of the current one.

In some rare cases, you may need to access the variable that `scm_module_define` would have accessed, without changing the binding of the existing variable, if one is present. In that case, use `scm_module_ensure_local_variable`:

`SCM scm_module_ensure_local_variable (SCM module, SCM sym)` [C Function]
Like `scm_module_define`, but if the *sym* is already locally bound in that module, the variable's existing binding is not reset. Returns a variable.

`SCM scm_module_reverse_lookup (SCM module, SCM variable)` [C Function]
Find the symbol that is bound to *variable* in *module*. When no such binding is found, return `#f`.

`SCM scm_c_define_module (const char *name, void (*init)(void *), void *data)` [C Function]
Define a new module named *name* and make it current while *init* is called, passing it *data*. Return the module.

The parameter *name* is a string with the symbols that make up the module name, separated by spaces. For example, '`"foo bar"`' names the module '`(foo bar)`'.

When there already exists a module named *name*, it is used unchanged, otherwise, an empty module is created.

`SCM scm_c_resolve_module (const char *name)` [C Function]
Find the module name *name* and return it. When it has not already been defined, try to auto-load it. When it can't be found that way either, create an empty module. The name is interpreted as for `scm_c_define_module`.

`SCM scm_c_use_module (const char *name)` [C Function]
Add the module named *name* to the uses list of the current module, as with `(use-modules name)`. The name is interpreted as for `scm_c_define_module`.

`void scm_c_export (const char *name, ...)` [C Function]
Add the bindings designated by *name*, ... to the public interface of the current module. The list of names is terminated by `NULL`.

6.20.11 provide and require

Aubrey Jaffer, mostly to support his portable Scheme library SLIB, implemented a provide/require mechanism for many Scheme implementations. Library files in SLIB *provide* a feature, and when user programs *require* that feature, the library file is loaded in.

For example, the file `random.scm` in the SLIB package contains the line

```
(provide 'random)
```

so to use its procedures, a user would type

```
(require 'random)
```

and they would magically become available, *but still have the same names!* So this method is nice, but not as good as a full-featured module system.

When SLIB is used with Guile, provide and require can be used to access its facilities.

6.20.12 Environments

Scheme, as defined in R5RS, does *not* have a full module system. However it does define the concept of a top-level *environment*. Such an environment maps identifiers (symbols) to Scheme objects such as procedures and lists: Section 3.4 [About Closure], page 26. In other words, it implements a set of *bindings*.

Environments in R5RS can be passed as the second argument to `eval` (see Section 6.18.4 [Fly Evaluation], page 387). Three procedures are defined to return environments: `scheme-report-environment`, `null-environment` and `interaction-environment` (see Section 6.18.4 [Fly Evaluation], page 387).

In addition, in Guile any module can be used as an R5RS environment, i.e., passed as the second argument to `eval`.

Note: the following two procedures are available only when the `(ice-9 r5rs)` module is loaded:

```
(use-modules (ice-9 r5rs))
```

`scheme-report-environment` *version* [Scheme Procedure]
`null-environment` *version* [Scheme Procedure]

version must be the exact integer '5', corresponding to revision 5 of the Scheme report (the Revised^5 Report on Scheme). `scheme-report-environment` returns a specifier for an environment that is empty except for all bindings defined in the report that are either required or both optional and supported by the implementation. `null-environment` returns a specifier for an environment that is empty except for the (syntactic) bindings for all syntactic keywords defined in the report that are either required or both optional and supported by the implementation.

Currently Guile does not support values of *version* for other revisions of the report.

The effect of assigning (through the use of `eval`) a variable bound in a `scheme-report-environment` (for example `car`) is unspecified. Currently the environments specified by `scheme-report-environment` are not immutable in Guile.

6.21 Foreign Function Interface

The more one hacks in Scheme, the more one realizes that there are actually two computational worlds: one which is warm and alive, that land of parentheses, and one cold and dead, the land of C and its ilk.

But yet we as programmers live in both worlds, and Guile itself is half implemented in C. So it is that Guile's living half pays respect to its dead counterpart, via a spectrum of interfaces to C ranging from dynamic loading of Scheme primitives to dynamic binding of stock C library procedures.

6.21.1 Foreign Libraries

Most modern Unices have something called *shared libraries*. This ordinarily means that they have the capability to share the executable image of a library between several running programs to save memory and disk space. But generally, shared libraries give a lot of additional flexibility compared to the traditional static libraries. In fact, calling them 'dynamic' libraries is as correct as calling them 'shared'.

Shared libraries really give you a lot of flexibility in addition to the memory and disk space savings. When you link a program against a shared library, that library is not closely incorporated into the final executable. Instead, the executable of your program only contains enough information to find the needed shared libraries when the program is actually run. Only then, when the program is starting, is the final step of the linking process performed. This means that you need not recompile all programs when you install a new, only slightly modified version of a shared library. The programs will pick up the changes automatically the next time they are run.

Now, when all the necessary machinery is there to perform part of the linking at run-time, why not take the next step and allow the programmer to explicitly take advantage of it from within their program? Of course, many operating systems that support shared libraries do just that, and chances are that Guile will allow you to access this feature from within your Scheme programs. As you might have guessed already, this feature is called *dynamic linking*.[13]

We titled this section "foreign libraries" because although the name "foreign" doesn't leak into the API, the world of C really is foreign to Scheme – and that estrangement extends to components of foreign libraries as well, as we see in future sections.

`dynamic-link` [*library*] [Scheme Procedure]
`scm_dynamic_link` (*library*) [C Function]

Find the shared library denoted by *library* (a string) and link it into the running Guile application. When everything works out, return a Scheme object suitable for representing the linked object file. Otherwise an error is thrown. How object files are searched is system dependent.

Guile first tries to load *library* as the absolute file name of a shared library. If that fails, it then falls back to interpret *library* as just the name of some shared library that will be searched for in the places where shared libraries usually reside, such as `/usr/lib` and `/usr/local/lib`.

library should not contain an extension such as `.so`, unless *library* represents the absolute file name to the shared library. The correct file name extension for the host operating system is provided automatically, according to libltdl's rules (see Section "`lt_dlopenext`" in *Shared Library Support for GNU*).

When *library* is omitted, a *global symbol handle* is returned. This handle provides access to the symbols available to the program at run-time, including those exported by the program itself and the shared libraries already loaded.

Note that on hosts that use dynamic-link libraries (DLLs), the global symbol handle may not be able to provide access to symbols from recursively-loaded DLLs. Only exported symbols from those DLLs directly loaded by the program may be available.

`dynamic-object?` *obj* [Scheme Procedure]
`scm_dynamic_object_p` (*obj*) [C Function]

Return `#t` if *obj* is a dynamic library handle, or `#f` otherwise.

[13] Some people also refer to the final linking stage at program startup as 'dynamic linking', so if you want to make yourself perfectly clear, it is probably best to use the more technical term *dlopening*, as suggested by Gordon Matzigkeit in his libtool documentation.

`dynamic-unlink` *dobj* [Scheme Procedure]
`scm_dynamic_unlink` (*dobj*) [C Function]

Unlink the indicated object file from the application. The argument *dobj* must have been obtained by a call to `dynamic-link`. After `dynamic-unlink` has been called on *dobj*, its content is no longer accessible.

```
(define libgl-obj (dynamic-link "libGL"))
libgl-obj
⇒ #<dynamic-object "libGL">
(dynamic-unlink libGL-obj)
libGL-obj
⇒ #<dynamic-object "libGL" (unlinked)>
```

As you can see, after calling `dynamic-unlink` on a dynamically linked library, it is marked as '`(unlinked)`' and you are no longer able to use it with `dynamic-call`, etc. Whether the library is really removed from you program is system-dependent and will generally not happen when some other parts of your program still use it.

When dynamic linking is disabled or not supported on your system, the above functions throw errors, but they are still available.

6.21.2 Foreign Functions

The most natural thing to do with a dynamic library is to grovel around in it for a function pointer: a *foreign function*. `dynamic-func` exists for that purpose.

`dynamic-func` *name dobj* [Scheme Procedure]
`scm_dynamic_func` (*name*, *dobj*) [C Function]

Return a "handle" for the func *name* in the shared object referred to by *dobj*. The handle can be passed to `dynamic-call` to actually call the function.

Regardless whether your C compiler prepends an underscore '_' to the global names in a program, you should **not** include this underscore in *name* since it will be added automatically when necessary.

Guile has static support for calling functions with no arguments, `dynamic-call`.

`dynamic-call` *func dobj* [Scheme Procedure]
`scm_dynamic_call` (*func*, *dobj*) [C Function]

Call the C function indicated by *func* and *dobj*. The function is passed no arguments and its return value is ignored. When *function* is something returned by `dynamic-func`, call that function and ignore *dobj*. When *func* is a string , look it up in *dynobj*; this is equivalent to

```
(dynamic-call (dynamic-func func dobj) #f)
```

`dynamic-call` is not very powerful. It is mostly intended to be used for calling specially written initialization functions that will then add new primitives to Guile. For example, we do not expect that you will dynamically link `libX11` with `dynamic-link` and then construct a beautiful graphical user interface just by using `dynamic-call`. Instead, the usual way would be to write a special Guile-to-X11 glue library that has intimate knowledge about both Guile and X11 and does whatever is necessary to make them inter-operate smoothly. This glue library could then be dynamically linked into a vanilla Guile interpreter and activated by calling its initialization function. That function would add all the new types and primitives to the Guile interpreter that it has to offer.

(There is actually another, better option: simply to create a `libX11` wrapper in Scheme via the dynamic FFI. See Section 6.21.6 [Dynamic FFI], page 439, for more information.)

Given some set of C extensions to Guile, the next logical step is to integrate these glue libraries into the module system of Guile so that you can load new primitives into a running system just as you can load new Scheme code.

`load-extension` *lib init* [Scheme Procedure]
`scm_load_extension` (*lib, init*) [C Function]

Load and initialize the extension designated by LIB and INIT. When there is no pre-registered function for LIB/INIT, this is equivalent to

```
(dynamic-call INIT (dynamic-link LIB))
```

When there is a pre-registered function, that function is called instead.

Normally, there is no pre-registered function. This option exists only for situations where dynamic linking is unavailable or unwanted. In that case, you would statically link your program with the desired library, and register its init function right after Guile has been initialized.

As for `dynamic-link`, *lib* should not contain any suffix such as `.so` (see Section 6.21.1 [Foreign Libraries], page 427). It should also not contain any directory components. Libraries that implement Guile Extensions should be put into the normal locations for shared libraries. We recommend to use the naming convention `libguile-bla-blum` for a extension related to a module `(bla blum)`.

The normal way for a extension to be used is to write a small Scheme file that defines a module, and to load the extension into this module. When the module is auto-loaded, the extension is loaded as well. For example,

```
(define-module (bla blum))

(load-extension "libguile-bla-blum" "bla_init_blum")
```

6.21.3 C Extensions

The most interesting application of dynamically linked libraries is probably to use them for providing *compiled code modules* to Scheme programs. As much fun as programming in Scheme is, every now and then comes the need to write some low-level C stuff to make Scheme even more fun.

Not only can you put these new primitives into their own module (see the previous section), you can even put them into a shared library that is only then linked to your running Guile image when it is actually needed.

An example will hopefully make everything clear. Suppose we want to make the Bessel functions of the C library available to Scheme in the module '`(math bessel)`'. First we need to write the appropriate glue code to convert the arguments and return values of the functions from Scheme to C and back. Additionally, we need a function that will add them to the set of Guile primitives. Because this is just an example, we will only implement this for the `j0` function.

```
#include <math.h>
#include <libguile.h>

SCM
```

```
j0_wrapper (SCM x)
{
  return scm_from_double (j0 (scm_to_double (x, "j0")));
}

void
init_math_bessel ()
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
}
```

We can already try to bring this into action by manually calling the low level functions for performing dynamic linking. The C source file needs to be compiled into a shared library. Here is how to do it on GNU/Linux, please refer to the `libtool` documentation for how to create dynamically linkable libraries portably.

```
gcc -shared -o libbessel.so -fPIC bessel.c
```

Now fire up Guile:

```
(define bessel-lib (dynamic-link "./libbessel.so"))
(dynamic-call "init_math_bessel" bessel-lib)
(j0 2)
⇒ 0.223890779141236
```

The filename `./libbessel.so` should be pointing to the shared library produced with the `gcc` command above, of course. The second line of the Guile interaction will call the `init_math_bessel` function which in turn will register the C function `j0_wrapper` with the Guile interpreter under the name `j0`. This function becomes immediately available and we can call it from Scheme.

Fun, isn't it? But we are only half way there. This is what `apropos` has to say about `j0`:

```
(apropos "j0")
⊣ (guile-user): j0     #<primitive-procedure j0>
```

As you can see, `j0` is contained in the root module, where all the other Guile primitives like `display`, etc live. In general, a primitive is put into whatever module is the *current module* at the time `scm_c_define_gsubr` is called.

A compiled module should have a specially named *module init function*. Guile knows about this special name and will call that function automatically after having linked in the shared library. For our example, we replace `init_math_bessel` with the following code in `bessel.c`:

```
void
init_math_bessel (void *unused)
{
  scm_c_define_gsubr ("j0", 1, 0, 0, j0_wrapper);
  scm_c_export ("j0", NULL);
}

void
scm_init_math_bessel_module ()
{
  scm_c_define_module ("math bessel", init_math_bessel, NULL);
}
```

The general pattern for the name of a module init function is: '`scm_init_`', followed by the name of the module where the individual hierarchical components are concatenated with underscores, followed by '`_module`'.

After `libbessel.so` has been rebuilt, we need to place the shared library into the right place.

Once the module has been correctly installed, it should be possible to use it like this:

```
guile> (load-extension "./libbessel.so" "scm_init_math_bessel_module")
guile> (use-modules (math bessel))
guile> (j0 2)
0.223890779141236
guile> (apropos "j0")
⊣ (math bessel): j0      #<primitive-procedure j0>
```

That's it!

6.21.4 Modules and Extensions

The new primitives that you add to Guile with `scm_c_define_gsubr` (see Section 6.9.2 [Primitive Procedures], page 249) or with any of the other mechanisms are placed into the module that is current when the `scm_c_define_gsubr` is executed. Extensions loaded from the REPL, for example, will be placed into the `(guile-user)` module, if the REPL module was not changed.

To define C primitives within a specific module, the simplest way is:

```
(define-module (foo bar))
(load-extension "foobar-c-code" "foo_bar_init")
```

When loaded with `(use-modules (foo bar))`, the `load-extension` call looks for the `foobar-c-code.so` (etc) object file in Guile's `extensiondir`, which is usually a subdirectory of the `libdir`. For example, if your libdir is `/usr/lib`, the `extensiondir` for the Guile 3.0.*x* series will be `/usr/lib/guile/3.0/`.

The extension path includes the major and minor version of Guile (the "effective version"), because Guile guarantees compatibility within a given effective version. This allows you to install different versions of the same extension for different versions of Guile.

If the extension is not found in the `extensiondir`, Guile will also search the standard system locations, such as `/usr/lib` or `/usr/local/lib`. It is preferable, however, to keep your extension out of the system library path, to prevent unintended interference with other dynamically-linked C libraries.

If someone installs your module to a non-standard location then the object file won't be found. You can address this by inserting the install location in the `foo/bar.scm` file. This is convenient for the user and also guarantees the intended object is read, even if stray older or newer versions are in the loader's path.

The usual way to specify an install location is with a `prefix` at the configure stage, for instance '`./configure prefix=/opt`' results in library files as say `/opt/lib/foobar-c-code.so`. When using Autoconf (see Section "Introduction" in *The GNU Autoconf Manual*), the library location is in a `libdir` variable. Its value is intended to be expanded by `make`, and can by substituted into a source file like `foo.scm.in`

```
(define-module (foo bar))
(load-extension "XXextensiondirXX/foobar-c-code" "foo_bar_init")
```

with the following in a `Makefile`, using `sed` (see Section "Introduction" in *SED*),

```
foo.scm: foo.scm.in
        sed 's|XXextensiondirXX|$(libdir)/guile/3.0|' <foo.scm.in >foo.scm
```

The actual pattern `XXextensiondirXX` is arbitrary, it's only something which doesn't otherwise occur. If several modules need the value, it can be easier to create one `foo/config.scm` with a define of the `extensiondir` location, and use that as required.

```
(define-module (foo config))
(define-public foo-config-extensiondir "XXextensiondirXX"")
```

Such a file might have other locations too, for instance a data directory for auxiliary files, or `localedir` if the module has its own `gettext` message catalogue (see Section 6.25 [Internationalization], page 465).

It will be noted all of the above requires that the Scheme code to be found in `%load-path` (see Section 6.18.7 [Load Paths], page 393). Presently it's left up to the system administrator or each user to augment that path when installing Guile modules in non-default locations. But having reached the Scheme code, that code should take care of hitting any of its own private files etc.

6.21.5 Foreign Pointers

The previous sections have shown how Guile can be extended at runtime by loading compiled C extensions. This approach is all well and good, but wouldn't it be nice if we didn't have to write any C at all? This section takes up the problem of accessing C values from Scheme, and the next discusses C functions.

6.21.5.1 Foreign Types

The first impedance mismatch that one sees between C and Scheme is that in C, the storage locations (variables) are typed, but in Scheme types are associated with values, not variables. See Section 3.1.2 [Values and Variables], page 15.

So when describing a C function or a C structure so that it can be accessed from Scheme, the data types of the parameters or fields must be passed explicitly.

These "C type values" may be constructed using the constants and procedures from the `(system foreign)` module, which may be loaded like this:

```
(use-modules (system foreign))
```

`(system foreign)` exports a number of values expressing the basic C types:

`int8` [Scheme Variable]
`uint8` [Scheme Variable]
`uint16` [Scheme Variable]
`int16` [Scheme Variable]
`uint32` [Scheme Variable]
`int32` [Scheme Variable]
`uint64` [Scheme Variable]
`int64` [Scheme Variable]
`float` [Scheme Variable]
`double` [Scheme Variable]

These values represent the C numeric types of the specified sizes and signednesses.

In addition there are some convenience bindings for indicating types of platform-dependent size:

`int` [Scheme Variable]
`unsigned-int` [Scheme Variable]
`long` [Scheme Variable]
`unsigned-long` [Scheme Variable]
`short` [Scheme Variable]
`unsigned-short` [Scheme Variable]
`size_t` [Scheme Variable]
`ssize_t` [Scheme Variable]
`ptrdiff_t` [Scheme Variable]
`intptr_t` [Scheme Variable]
`uintptr_t` [Scheme Variable]

Values exported by the `(system foreign)` module, representing C numeric types. For example, `long` may be `equal?` to `int64` on a 64-bit platform.

`void` [Scheme Variable]

The `void` type. It can be used as the first argument to `pointer->procedure` to wrap a C function that returns nothing.

In addition, the symbol `*` is used by convention to denote pointer types. Procedures detailed in the following sections, such as `pointer->procedure`, accept it as a type descriptor.

6.21.5.2 Foreign Variables

Pointers to variables in the current address space may be looked up dynamically using `dynamic-pointer`.

`dynamic-pointer` *name dobj* [Scheme Procedure]
`scm_dynamic_pointer` (*name, dobj*) [C Function]

Return a "wrapped pointer" for the symbol *name* in the shared object referred to by *dobj*. The returned pointer points to a C object.

Regardless whether your C compiler prepends an underscore '`_`' to the global names in a program, you should **not** include this underscore in *name* since it will be added automatically when necessary.

For example, currently Guile has a variable, `scm_numptob`, as part of its API. It is declared as a C `long`. So, to create a handle pointing to that foreign value, we do:

```
(use-modules (system foreign))
(define numptob (dynamic-pointer "scm_numptob" (dynamic-link)))
numptob
⇒ #<pointer 0x7fb35b1b4688>
```

(The next section discusses ways to dereference pointers.)

A value returned by `dynamic-pointer` is a Scheme wrapper for a C pointer.

`pointer-address` *pointer* [Scheme Procedure]
`scm_pointer_address` (*pointer*) [C Function]
Return the numerical value of *pointer*.

```
(pointer-address numptob)
⇒ 139984413364296 ; YMMV
```

`make-pointer` *address* [*finalizer*] [Scheme Procedure]
Return a foreign pointer object pointing to *address*. If *finalizer* is passed, it should be a pointer to a one-argument C function that will be called when the pointer object becomes unreachable.

`pointer?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a pointer object, `#f` otherwise.

`%null-pointer` [Scheme Variable]
A foreign pointer whose value is 0.

`null-pointer?` *pointer* [Scheme Procedure]
Return `#t` if *pointer* is the null pointer, `#f` otherwise.

For the purpose of passing SCM values directly to foreign functions, and allowing them to return SCM values, Guile also supports some unsafe casting operators.

`scm->pointer` *scm* [Scheme Procedure]
Return a foreign pointer object with the `object-address` of *scm*.

`pointer->scm` *pointer* [Scheme Procedure]
Unsafely cast *pointer* to a Scheme object. Cross your fingers!

Sometimes you want to give C extensions access to the dynamic FFI. At that point, the names get confusing, because "pointer" can refer to a `SCM` object that wraps a pointer, or to a `void*` value. We will try to use "pointer object" to refer to Scheme objects, and "pointer value" to refer to `void *` values.

`SCM scm_from_pointer` (*void *ptr, void (*finalizer) (void*)*) [C Function]
Create a pointer object from a pointer value.

If *finalizer* is non-null, Guile arranges to call it on the pointer value at some point after the pointer object becomes collectable.

`void* scm_to_pointer` (*SCM obj*) [C Function]
Unpack the pointer value from a pointer object.

6.21.5.3 Void Pointers and Byte Access

Wrapped pointers are untyped, so they are essentially equivalent to C `void` pointers. As in C, the memory region pointed to by a pointer can be accessed at the byte level. This is achieved using *bytevectors* (see Section 6.6.12 [Bytevectors], page 193). The `(rnrs bytevectors)` module contains procedures that can be used to convert byte sequences to Scheme objects such as strings, floating point numbers, or integers.

`pointer->bytevector` *pointer len* [*offset* [*uvec_type*]] [Scheme Procedure]
`scm_pointer_to_bytevector` (*pointer, len, offset, uvec_type*) [C Function]
Return a bytevector aliasing the *len* bytes pointed to by *pointer*.

The user may specify an alternate default interpretation for the memory by passing the *uvec_type* argument, to indicate that the memory is an array of elements of that type. *uvec_type* should be something that `array-type` would return, like `f32` or `s16`.

When *offset* is passed, it specifies the offset in bytes relative to *pointer* of the memory region aliased by the returned bytevector.

Mutating the returned bytevector mutates the memory pointed to by *pointer*, so buckle your seatbelts.

`bytevector->pointer` *bv* [*offset*] [Scheme Procedure]
`scm_bytevector_to_pointer` (*bv, offset*) [C Function]
Return a pointer pointer aliasing the memory pointed to by *bv* or *offset* bytes after *bv* when *offset* is passed.

In addition to these primitives, convenience procedures are available:

`dereference-pointer` *pointer* [Scheme Procedure]
Assuming *pointer* points to a memory region that holds a pointer, return this pointer.

`string->pointer` *string* [*encoding*] [Scheme Procedure]
Return a foreign pointer to a nul-terminated copy of *string* in the given *encoding*, defaulting to the current locale encoding. The C string is freed when the returned foreign pointer becomes unreachable.

This is the Scheme equivalent of `scm_to_stringn`.

`pointer->string` *pointer* [*length*] [*encoding*] [Scheme Procedure]
Return the string representing the C string pointed to by *pointer*. If *length* is omitted or `-1`, the string is assumed to be nul-terminated. Otherwise *length* is the number of bytes in memory pointed to by *pointer*. The C string is assumed to be in the given *encoding*, defaulting to the current locale encoding.

This is the Scheme equivalent of `scm_from_stringn`.

Most object-oriented C libraries use pointers to specific data structures to identify objects. It is useful in such cases to reify the different pointer types as disjoint Scheme types. The `define-wrapped-pointer-type` macro simplifies this.

`define-wrapped-pointer-type` *type-name pred wrap unwrap print* [Scheme Syntax]
Define helper procedures to wrap pointer objects into Scheme objects with a disjoint type. Specifically, this macro defines:

- *pred*, a predicate for the new Scheme type;
- *wrap*, a procedure that takes a pointer object and returns an object that satisfies *pred*;
- *unwrap*, which does the reverse.

wrap preserves pointer identity, for two pointer objects *p1* and *p2* that are `equal?`, `(eq? (`*wrap p1*`) (`*wrap p2*`))` ⇒ `#t`.

Finally, *print* should name a user-defined procedure to print such objects. The procedure is passed the wrapped object and a port to write to.

For example, assume we are wrapping a C library that defines a type, `bottle_t`, and functions that can be passed `bottle_t *` pointers to manipulate them. We could write:

```
(define-wrapped-pointer-type bottle
  bottle?
  wrap-bottle unwrap-bottle
  (lambda (b p)
    (format p "#<bottle of ~a ~x>"
            (bottle-contents b)
            (pointer-address (unwrap-bottle b)))))

(define grab-bottle
  ;; Wrapper for `bottle_t *grab (void)'.
  (let ((grab (pointer->procedure '*
                                  (dynamic-func "grab_bottle" libbottle)
                                  '())))
    (lambda ()
      "Return a new bottle."
      (wrap-bottle (grab)))))

(define bottle-contents
  ;; Wrapper for `const char *bottle_contents (bottle_t *)'.
  (let ((contents (pointer->procedure '*
                                      (dynamic-func "bottle_contents"
                                                    libbottle)
                                      '(*))))
    (lambda (b)
      "Return the contents of B."
      (pointer->string (contents (unwrap-bottle b))))))

(write (grab-bottle))
⇒ #<bottle of Château Haut-Brion 803d36>
```

In this example, `grab-bottle` is guaranteed to return a genuine `bottle` object satisfying `bottle?`. Likewise, `bottle-contents` errors out when its argument is not a genuine `bottle` object.

Going back to the `scm_numptob` example above, here is how we can read its value as a C `long` integer:

```
(use-modules (rnrs bytevectors))

(bytevector-uint-ref (pointer->bytevector numptob (sizeof long))
                     0 (native-endianness)
```

```
                        (sizeof long))
    ⇒ 8
```

If we wanted to corrupt Guile's internal state, we could set `scm_numptob` to another value; but we shouldn't, because that variable is not meant to be set. Indeed this point applies more widely: the C API is a dangerous place to be. Not only might setting a value crash your program, simply accessing the data pointed to by a dangling pointer or similar can prove equally disastrous.

6.21.5.4 Foreign Structs

Finally, one last note on foreign values before moving on to actually calling foreign functions. Sometimes you need to deal with C structs, which requires interpreting each element of the struct according to the its type, offset, and alignment. Guile has some primitives to support this.

`sizeof` *type* [Scheme Procedure]
`scm_sizeof` (*type*) [C Function]

Return the size of *type*, in bytes.

type should be a valid C type, like `int`. Alternately *type* may be the symbol `*`, in which case the size of a pointer is returned. *type* may also be a list of types, in which case the size of a `struct` with ABI-conventional packing is returned.

`alignof` *type* [Scheme Procedure]
`scm_alignof` (*type*) [C Function]

Return the alignment of *type*, in bytes.

type should be a valid C type, like `int`. Alternately *type* may be the symbol `*`, in which case the alignment of a pointer is returned. *type* may also be a list of types, in which case the alignment of a `struct` with ABI-conventional packing is returned.

Guile also provides some convenience methods to pack and unpack foreign pointers wrapping C structs.

`make-c-struct` *types vals* [Scheme Procedure]

Create a foreign pointer to a C struct containing *vals* with types `types`.

vals and `types` should be lists of the same length.

`parse-c-struct` *foreign types* [Scheme Procedure]

Parse a foreign pointer to a C struct, returning a list of values.

`types` should be a list of C types.

For example, to create and parse the equivalent of a `struct { int64_t a; uint8_t b; }`:

```
(parse-c-struct (make-c-struct (list int64 uint8)
                               (list 300 43))
                (list int64 uint8))
⇒ (300 43)
```

As yet, Guile only has convenience routines to support conventionally-packed structs. But given the `bytevector->pointer` and `pointer->bytevector` routines, one can create and parse tightly packed structs and unions by hand. See the code for `(system foreign)` for details.

6.21.6 Dynamic FFI

Of course, the land of C is not all nouns and no verbs: there are functions too, and Guile allows you to call them.

`pointer->procedure` *return_type func_ptr arg_types* [*#:return-errno?=#f*] [Scheme Procedure]

`scm_pointer_to_procedure` (*return_type, func_ptr, arg_types*) [C Function]

`scm_pointer_to_procedure_with_errno` (*return_type, func_ptr, arg_types*) [C Function]

Make a foreign function.

Given the foreign void pointer *func_ptr*, its argument and return types *arg_types* and *return_type*, return a procedure that will pass arguments to the foreign function and return appropriate values.

arg_types should be a list of foreign types. `return_type` should be a foreign type. See Section 6.21.5.1 [Foreign Types], page 433, for more information on foreign types.

If *return-errno?* is true, or when calling `scm_pointer_to_procedure_with_errno`, the returned procedure will return two values, with `errno` as the second value.

Here is a better definition of `(math bessel)`:

```
(define-module (math bessel)
  #:use-module (system foreign)
  #:export (j0))

(define libm (dynamic-link "libm"))

(define j0
  (pointer->procedure double
                      (dynamic-func "j0" libm)
                      (list double)))
```

That's it! No C at all.

Numeric arguments and return values from foreign functions are represented as Scheme values. For example, `j0` in the above example takes a Scheme number as its argument, and returns a Scheme number.

Pointers may be passed to and returned from foreign functions as well. In that case the type of the argument or return value should be the symbol `*`, indicating a pointer. For example, the following code makes `memcpy` available to Scheme:

```
(define memcpy
  (let ((this (dynamic-link)))
    (pointer->procedure '*
                        (dynamic-func "memcpy" this)
                        (list '* '* size_t))))
```

To invoke `memcpy`, one must pass it foreign pointers:

```
(use-modules (rnrs bytevectors))

(define src-bits
```

```
  (u8-list->bytevector '(0 1 2 3 4 5 6 7)))
(define src
  (bytevector->pointer src-bits))
(define dest
  (bytevector->pointer (make-bytevector 16 0)))

(memcpy dest src (bytevector-length src-bits))

(bytevector->u8-list (pointer->bytevector dest 16))
⇒ (0 1 2 3 4 5 6 7 0 0 0 0 0 0 0 0)
```

One may also pass structs as values, passing structs as foreign pointers. See Section 6.21.5.4 [Foreign Structs], page 438, for more information on how to express struct types and struct values.

"Out" arguments are passed as foreign pointers. The memory pointed to by the foreign pointer is mutated in place.

```
;; struct timeval {
;;      time_t      tv_sec;     /* seconds */
;;      suseconds_t tv_usec;    /* microseconds */
;; };
;; assuming fields are of type "long"

(define gettimeofday
  (let ((f (pointer->procedure
            int
            (dynamic-func "gettimeofday" (dynamic-link))
            (list '* '*)))
        (tv-type (list long long)))
    (lambda ()
      (let* ((timeval (make-c-struct tv-type (list 0 0)))
             (ret (f timeval %null-pointer)))
        (if (zero? ret)
            (apply values (parse-c-struct timeval tv-type))
            (error "gettimeofday returned an error" ret))))))

(gettimeofday)
⇒ 1270587589
⇒ 499553
```

As you can see, this interface to foreign functions is at a very low, somewhat dangerous level[14].

The FFI can also work in the opposite direction: making Scheme procedures callable from C. This makes it possible to use Scheme procedures as "callbacks" expected by C function.

[14] A contribution to Guile in the form of a high-level FFI would be most welcome.

`procedure->pointer` *return-type proc arg-types* [Scheme Procedure]
`scm_procedure_to_pointer` (*return_type, proc, arg_types*) [C Function]

Return a pointer to a C function of type *return-type* taking arguments of types *arg-types* (a list) and behaving as a proxy to procedure *proc*. Thus *proc*'s arity, supported argument types, and return type should match *return-type* and *arg-types*.

As an example, here's how the C library's `qsort` array sorting function can be made accessible to Scheme (see Section "Array Sort Function" in *The GNU C Library Reference Manual*):

```
(define qsort!
  (let ((qsort (pointer->procedure void
                                   (dynamic-func "qsort"
                                                 (dynamic-link))
                                   (list '* size_t size_t '*))))
    (lambda (bv compare)
      ;; Sort bytevector BV in-place according to comparison
      ;; procedure COMPARE.
      (let ((ptr (procedure->pointer int
                                     (lambda (x y)
                                       ;; X and Y are pointers so,
                                       ;; for convenience, dereference
                                       ;; them before calling COMPARE.
                                       (compare (dereference-uint8* x)
                                                (dereference-uint8* y)))
                                     (list '* '*))))
        (qsort (bytevector->pointer bv)
               (bytevector-length bv) 1 ;; we're sorting bytes
               ptr)))))

(define (dereference-uint8* ptr)
  ;; Helper function: dereference the byte pointed to by PTR.
  (let ((b (pointer->bytevector ptr 1)))
    (bytevector-u8-ref b 0)))

(define bv
  ;; An unsorted array of bytes.
  (u8-list->bytevector '(7 1 127 3 5 4 77 2 9 0)))

;; Sort BV.
(qsort! bv (lambda (x y) (- x y)))

;; Let's see what the sorted array looks like:
(bytevector->u8-list bv)
⇒ (0 1 2 3 4 5 7 9 77 127)
```

And voilà!

Note that `procedure->pointer` is not supported (and not defined) on a few exotic architectures. Thus, user code may need to check `(defined? 'procedure->pointer)`. Never-

theless, it is available on many architectures, including (as of libffi 3.0.9) x86, ia64, SPARC, PowerPC, ARM, and MIPS, to name a few.

6.22 Threads, Mutexes, Asyncs and Dynamic Roots

6.22.1 Threads

Guile supports POSIX threads, unless it was configured with `--without-threads` or the host lacks POSIX thread support. When thread support is available, the `threads` feature is provided (see Section 6.23.2.1 [Feature Manipulation], page 458).

The procedures below manipulate Guile threads, which are wrappers around the system's POSIX threads. For application-level parallelism, using higher-level constructs, such as futures, is recommended (see Section 6.22.7 [Futures], page 453).

To use these facilities, load the `(ice-9 threads)` module.

```
(use-modules (ice-9 threads))
```

`all-threads` [Scheme Procedure]
`scm_all_threads ()` [C Function]
Return a list of all threads.

`current-thread` [Scheme Procedure]
`scm_current_thread ()` [C Function]
Return the thread that called this function.

`call-with-new-thread` *thunk* [*handler*] [Scheme Procedure]
Call `thunk` in a new thread and with a new dynamic state, returning the new thread. The procedure *thunk* is called via `with-continuation-barrier`.

When *handler* is specified, then *thunk* is called from within a `catch` with tag `#t` that has *handler* as its handler. This catch is established inside the continuation barrier.

Once *thunk* or *handler* returns, the return value is made the *exit value* of the thread and the thread is terminated.

`SCM scm_spawn_thread` (*scm_t_catch_body body, void *body_data, scm_t_catch_handler handler, void *handler_data*) [C Function]
Call *body* in a new thread, passing it *body_data*, returning the new thread. The function *body* is called via `scm_c_with_continuation_barrier`.

When *handler* is non-`NULL`, *body* is called via `scm_internal_catch` with tag `SCM_BOOL_T` that has *handler* and *handler_data* as the handler and its data. This catch is established inside the continuation barrier.

Once *body* or *handler* returns, the return value is made the *exit value* of the thread and the thread is terminated.

`thread?` *obj* [Scheme Procedure]
`scm_thread_p` (*obj*) [C Function]
Return `#t` ff *obj* is a thread; otherwise, return `#f`.

`join-thread` *thread* [*timeout* [*timeoutval*]] [Scheme Procedure]
`scm_join_thread` (*thread*) [C Function]
`scm_join_thread_timed` (*thread, timeout, timeoutval*) [C Function]

Wait for *thread* to terminate and return its exit value. Only threads that were created with `call-with-new-thread` or `scm_spawn_thread` can be joinable; attempting to join a foreign thread will raise an error.

When *timeout* is given, it specifies a point in time where the waiting should be aborted. It can be either an integer as returned by `current-time` or a pair as returned by `gettimeofday`. When the waiting is aborted, *timeoutval* is returned (if it is specified; `#f` is returned otherwise).

`thread-exited?` *thread* [Scheme Procedure]
`scm_thread_exited_p` (*thread*) [C Function]

Return `#t` if *thread* has exited, or `#f` otherwise.

`yield` [Scheme Procedure]
`scm_yield` (*thread*) [C Function]

If one or more threads are waiting to execute, calling yield forces an immediate context switch to one of them. Otherwise, yield has no effect.

`cancel-thread` *thread . values* [Scheme Procedure]
`scm_cancel_thread` (*thread*) [C Function]

Asynchronously interrupt *thread* and ask it to terminate. `dynamic-wind` post thunks will run, but throw handlers will not. If *thread* has already terminated or been signaled to terminate, this function is a no-op. Calling `join-thread` on the thread will return the given *values*, if the cancel succeeded.

Under the hood, thread cancellation uses `system-async-mark` and `abort-to-prompt`. See Section 6.22.3 [Asyncs], page 445, for more on asynchronous interrupts.

`make-thread` *proc arg ...* [macro]

Apply *proc* to *arg ...* in a new thread formed by `call-with-new-thread` using a default error handler that display the error to the current error port. The *arg ...* expressions are evaluated in the new thread.

`begin-thread` *expr1 expr2 ...* [macro]

Evaluate forms *expr1 expr2 ...* in a new thread formed by `call-with-new-thread` using a default error handler that display the error to the current error port.

One often wants to limit the number of threads running to be proportional to the number of available processors. These interfaces are therefore exported by (ice-9 threads) as well.

`total-processor-count` [Scheme Procedure]
`scm_total_processor_count` () [C Function]

Return the total number of processors of the machine, which is guaranteed to be at least 1. A "processor" here is a thread execution unit, which can be either:

- an execution core in a (possibly multi-core) chip, in a (possibly multi- chip) module, in a single computer, or
- a thread execution unit inside a core in the case of *hyper-threaded* CPUs.

Which of the two definitions is used, is unspecified.

`current-processor-count` [Scheme Procedure]
`scm_current_processor_count ()` [C Function]
Like `total-processor-count`, but return the number of processors available to the current process. See `setaffinity` and `getaffinity` for more information.

6.22.2 Thread-Local Variables

Sometimes you want to establish a variable binding that is only valid for a given thread: a "thread-local variable".

You would think that fluids or parameters would be Guile's answer for thread-local variables, since establishing a new fluid binding doesn't affect bindings in other threads. See Section 6.13.11 [Fluids and Dynamic States], page 323, or See Section 6.13.12 [Parameters], page 326. However, new threads inherit the fluid bindings that were in place in their creator threads. In this way, a binding established using a fluid (or a parameter) in a thread can escape to other threads, which might not be what you want. Or, it might escape via explicit reification via `current-dynamic-state`.

Of course, this dynamic scoping might be exactly what you want; that's why fluids and parameters work this way, and is what you want for for many common parameters such as the current input and output ports, the current locale conversion parameters, and the like. Perhaps this is the case for most parameters, even. If your use case for thread-local bindings comes from a desire to isolate a binding from its setting in unrelated threads, then fluids and parameters apply nicely.

On the other hand, if your use case is to prevent concurrent access to a value from multiple threads, then using vanilla fluids or parameters is not appropriate. For this purpose, Guile has *thread-local fluids*. A fluid created with `make-thread-local-fluid` won't be captured by `current-dynamic-state` and won't be propagated to new threads.

`make-thread-local-fluid` [*dflt*] [Scheme Procedure]
`scm_make_thread_local_fluid` (*dflt*) [C Function]
Return a newly created fluid, whose initial value is *dflt*, or `#f` if *dflt* is not given. Unlike fluids made with `make-fluid`, thread local fluids are not captured by `make-dynamic-state`. Similarly, a newly spawned child thread does not inherit thread-local fluid values from the parent thread.

`fluid-thread-local?` *fluid* [Scheme Procedure]
`scm_fluid_thread_local_p` (*fluid*) [C Function]
Return `#t` if the fluid *fluid* is is thread-local, or `#f` otherwise.

For example:

```
(define %thread-local (make-thread-local-fluid))

(with-fluids ((%thread-local (compute-data)))
  ... (fluid-ref %thread-local) ...)
```

You can also make a thread-local parameter out of a thread-local fluid using the normal `fluid->parameter`:

```
(define param (fluid->parameter (make-thread-local-fluid)))
```

```
(parameterize ((param (compute-data)))
  ... (param) ...)
```

6.22.3 Asynchronous Interrupts

Every Guile thread can be interrupted. Threads running Guile code will periodically check if there are pending interrupts and run them if necessary. To interrupt a thread, call `system-async-mark` on that thread.

`system-async-mark` *proc* [*thread*] [Scheme Procedure]
`scm_system_async_mark` (*proc*) [C Function]
`scm_system_async_mark_for_thread` (*proc, thread*) [C Function]

Enqueue *proc* (a procedure with zero arguments) for future execution in *thread*. When *proc* has already been enqueued for *thread* but has not been executed yet, this call has no effect. When *thread* is omitted, the thread that called `system-async-mark` is used.

Note that `scm_system_async_mark_for_thread` is not "async-signal-safe" and so cannot be called from a C signal handler. (Indeed in general, `libguile` functions are not safe to call from C signal handlers.)

Though an interrupt procedure can have any side effect permitted to Guile code, asynchronous interrupts are generally used either for profiling or for prematurely cancelling a computation. The former case is mostly transparent to the program being run, by design, but the latter case can introduce bugs. Like finalizers (see Section 5.5.4 [Foreign Object Memory Management], page 77), asynchronous interrupts introduce concurrency in a program. An asyncronous interrupt can run in the middle of some mutex-protected operation, for example, and potentially corrupt the program's state.

If some bit of Guile code needs to temporarily inhibit interrupts, it can use `call-with-blocked-asyncs`. This function works by temporarily increasing the *async blocking level* of the current thread while a given procedure is running. The blocking level starts out at zero, and whenever a safe point is reached, a blocking level greater than zero will prevent the execution of queued asyncs.

Analogously, the procedure `call-with-unblocked-asyncs` will temporarily decrease the blocking level of the current thread. You can use it when you want to disable asyncs by default and only allow them temporarily.

In addition to the C versions of `call-with-blocked-asyncs` and `call-with-unblocked-asyncs`, C code can use `scm_dynwind_block_asyncs` and `scm_dynwind_unblock_asyncs` inside a *dynamic context* (see Section 6.13.10 [Dynamic Wind], page 320) to block or unblock asyncs temporarily.

`call-with-blocked-asyncs` *proc* [Scheme Procedure]
`scm_call_with_blocked_asyncs` (*proc*) [C Function]

Call *proc* and block the execution of asyncs by one level for the current thread while it is running. Return the value returned by *proc*. For the first two variants, call *proc* with no arguments; for the third, call it with *data*.

`void * scm_c_call_with_blocked_asyncs` (*void * (*proc) (void *data), void *data*) [C Function]

The same but with a C function *proc* instead of a Scheme thunk.

`call-with-unblocked-asyncs` *proc* [Scheme Procedure]
`scm_call_with_unblocked_asyncs` (*proc*) [C Function]
Call *proc* and unblock the execution of asyncs by one level for the current thread while it is running. Return the value returned by *proc*. For the first two variants, call *proc* with no arguments; for the third, call it with *data*.

`void * scm_c_call_with_unblocked_asyncs` (*void *(*proc) (void *data), void *data*) [C Function]
The same but with a C function *proc* instead of a Scheme thunk.

`void scm_dynwind_block_asyncs` () [C Function]
During the current dynwind context, increase the blocking of asyncs by one level. This function must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (see Section 6.13.10 [Dynamic Wind], page 320).

`void scm_dynwind_unblock_asyncs` () [C Function]
During the current dynwind context, decrease the blocking of asyncs by one level. This function must be used inside a pair of calls to `scm_dynwind_begin` and `scm_dynwind_end` (see Section 6.13.10 [Dynamic Wind], page 320).

Sometimes you want to interrupt a thread that might be waiting for something to happen, for example on a file descriptor or a condition variable. In that case you can inform Guile of how to interrupt that wait using the following procedures:

`int scm_c_prepare_to_wait_on_fd` (*int fd*) [C Function]
Inform Guile that the current thread is about to sleep, and that if an asynchronous interrupt is signalled on this thread, Guile should wake up the thread by writing a zero byte to *fd*. Returns zero if the prepare succeeded, or nonzero if the thread already has a pending async and that it should avoid waiting.

`int scm_c_prepare_to_wait_on_cond` (*scm_i_pthread_mutex_t *mutex, scm_i_pthread_cond_t *cond*) [C Function]
Inform Guile that the current thread is about to sleep, and that if an asynchronous interrupt is signalled on this thread, Guile should wake up the thread by acquiring *mutex* and signalling *cond*. The caller must already hold *mutex* and only drop it as part of the `pthread_cond_wait` call. Returns zero if the prepare succeeded, or nonzero if the thread already has a pending async and that it should avoid waiting.

`void scm_c_wait_finished` (*void*) [C Function]
Inform Guile that the current thread has finished waiting, and that asynchronous interrupts no longer need any special wakeup action; the current thread will periodically poll its internal queue instead.

Guile's own interface to `sleep`, `wait-condition-variable`, `select`, and so on all call the above routines as appropriate.

Finally, note that threads can also be interrupted via POSIX signals. See Section 7.2.8 [Signals], page 524. As an implementation detail, signal handlers will effectively call `system-async-mark` in a signal-safe way, eventually running the signal handler using the same async mechanism. In this way you can temporarily inhibit signal handlers from running using the above interfaces.

6.22.4 Atomics

When accessing data in parallel from multiple threads, updates made by one thread are not generally guaranteed to be visible by another thread. It could be that your hardware requires special instructions to be emitted to propagate a change from one CPU core to another. Or, it could be that your hardware updates values with a sequence of instructions, and a parallel thread could see a value that is in the process of being updated but not fully updated.

Atomic references solve this problem. Atomics are a standard, primitive facility to allow for concurrent access and update of mutable variables from multiple threads with guaranteed forward-progress and well-defined intermediate states.

Atomic references serve not only as a hardware memory barrier but also as a compiler barrier. Normally a compiler might choose to reorder or elide certain memory accesses due to optimizations like common subexpression elimination. Atomic accesses however will not be reordered relative to each other, and normal memory accesses will not be reordered across atomic accesses.

As an implementation detail, currently all atomic accesses and updates use the sequential consistency memory model from C11. We may relax this in the future to the acquire/release semantics, which still issues a memory barrier so that non-atomic updates are not reordered across atomic accesses or updates.

To use Guile's atomic operations, load the `(ice-9 atomic)` module:

```
(use-modules (ice-9 atomic))
```

`make-atomic-box` *init* [Scheme Procedure]
Return an atomic box initialized to value *init*.

`atomic-box?` *obj* [Scheme Procedure]
Return `#t` if *obj* is an atomic-box object, else return `#f`.

`atomic-box-ref` *box* [Scheme Procedure]
Fetch the value stored in the atomic box *box* and return it.

`atomic-box-set!` *box val* [Scheme Procedure]
Store *val* into the atomic box *box*.

`atomic-box-swap!` *box val* [Scheme Procedure]
Store *val* into the atomic box *box*, and return the value that was previously stored in the box.

`atomic-box-compare-and-swap!` *box expected desired* [Scheme Procedure]
If the value of the atomic box *box* is the same as, *expected* (in the sense of `eq?`), replace the contents of the box with *desired*. Otherwise does not update the box. Returns the previous value of the box in either case, so you can know if the swap worked by checking if the return value is `eq?` to *expected*.

6.22.5 Mutexes and Condition Variables

Mutexes are low-level primitives used to coordinate concurrent access to mutable data. Short for "mutual exclusion", the name "mutex" indicates that only one thread at a time can acquire access to data that is protected by a mutex – threads are excluded from accessing data at the same time. If one thread has locked a mutex, then another thread attempting to lock that same mutex will wait until the first thread is done.

Mutexes can be used to build robust multi-threaded programs that take advantage of multiple cores. However, they provide very low-level functionality and are somewhat dangerous; usually you end up wanting to acquire multiple mutexes at the same time to perform a multi-object access, but this can easily lead to deadlocks if the program is not carefully written. For example, if objects A and B are protected by associated mutexes M and N, respectively, then to access both of them then you need to acquire both mutexes. But what if one thread acquires M first and then N, at the same time that another thread acquires N them M? You can easily end up in a situation where one is waiting for the other.

There's no easy way around this problem on the language level. A function A that uses mutexes does not necessarily compose nicely with a function B that uses mutexes. For this reason we suggest using atomic variables when you can (see Section 6.22.4 [Atomics], page 447), as they do not have this problem.

Still, if you as a programmer are responsible for a whole system, then you can use mutexes as a primitive to provide safe concurrent abstractions to your users. (For example, given all locks in a system, if you establish an order such that M is consistently acquired before N, you can avoid the "deadly-embrace" deadlock described above. The problem is enumerating all mutexes and establishing this order from a system perspective.) Guile gives you the low-level facilities to build such systems.

In Guile there are additional considerations beyond the usual ones in other programming languages: non-local control flow and asynchronous interrupts. What happens if you hold a mutex, but somehow you cause an exception to be thrown? There is no one right answer. You might want to keep the mutex locked to prevent any other code from ever entering that critical section again. Or, your critical section might be fine if you unlock the mutex "on the way out", via an exception handler or `dynamic-wind`. See Section 6.13.8 [Exceptions], page 311, and See Section 6.13.10 [Dynamic Wind], page 320.

But if you arrange to unlock the mutex when leaving a dynamic extent via `dynamic-wind`, what to do if control re-enters that dynamic extent via a continuation invocation? Surely re-entering the dynamic extent without the lock is a bad idea, so there are two options on the table: either prevent re-entry via `with-continuation-barrier` or similar, or reacquire the lock in the entry thunk of a `dynamic-wind`.

You might think that because you don't use continuations, that you don't have to think about this, and you might be right. If you control the whole system, you can reason about continuation use globally. Or, if you know all code that can be called in a dynamic extent, and none of that code can call continuations, then you don't have to worry about re-entry, and you might not have to worry about early exit either.

However, do consider the possibility of asynchronous interrupts (see Section 6.22.3 [Asyncs], page 445). If the user interrupts your code interactively, that can cause an exception; or your thread might be cancelled, which does the same; or the user could be running your code under some pre-emptive system that periodically causes lightweight task

switching. (Guile does not currently include such a system, but it's possible to implement as a library.) Probably you also want to defer asynchronous interrupt processing while you hold the mutex, and probably that also means that you should not hold the mutex for very long.

All of these additional Guile-specific considerations mean that from a system perspective, you would do well to avoid these hazards if you can by not requiring mutexes. Instead, work with immutable data that can be shared between threads without hazards, or use persistent data structures with atomic updates based on the atomic variable library (see Section 6.22.4 [Atomics], page 447).

There are three types of mutexes in Guile: "standard", "recursive", and "unowned".

Calling `make-mutex` with no arguments makes a standard mutex. A standard mutex can only be locked once. If you try to lock it again from the thread that locked it to begin with (the `"owner"` thread), it throws an error. It can only be unlocked from the thread that locked it in the first place.

Calling `make-mutex` with the symbol `recursive` as the argument, or calling `make-recursive-mutex`, will give you a recursive mutex. A recursive mutex can be locked multiple times by its owner. It then has to be unlocked the corresponding number of times, and like standard mutexes can only be unlocked by the owner thread.

Finally, calling `make-mutex` with the symbol `allow-external-unlock` creates an unowned mutex. An unowned mutex is like a standard mutex, except that it can be unlocked by any thread. A corollary of this behavior is that a thread's attempt to lock a mutex that it already owns will block instead of signalling an error, as it could be that some other thread unlocks the mutex, allowing the owner thread to proceed. This kind of mutex is a bit strange and is here for use by SRFI-18.

The mutex procedures in Guile can operate on all three kinds of mutexes.

To use these facilities, load the `(ice-9 threads)` module.

```
(use-modules (ice-9 threads))
```

`make-mutex` [*kind*] [Scheme Procedure]
`scm_make_mutex` () [C Function]
`scm_make_mutex_with_kind` (*SCM kind*) [C Function]

Return a new mutex. It will be a standard non-recursive mutex, unless the `recursive` symbol is passed as the optional *kind* argument, in which case it will be recursive. It's also possible to pass `unowned` for semantics tailored to SRFI-18's use case; see above for details.

`mutex?` *obj* [Scheme Procedure]
`scm_mutex_p` (*obj*) [C Function]

Return `#t` if *obj* is a mutex; otherwise, return `#f`.

`make-recursive-mutex` [Scheme Procedure]
`scm_make_recursive_mutex` () [C Function]

Create a new recursive mutex. It is initially unlocked. Calling this function is equivalent to calling `make-mutex` with the `recursive` kind.

`lock-mutex` *mutex* [*timeout*] [Scheme Procedure]
`scm_lock_mutex` (*mutex*) [C Function]
`scm_timed_lock_mutex` (*mutex, timeout*) [C Function]

Lock *mutex* and return `#t`. If the mutex is already locked, then block and return only when *mutex* has been acquired.

When *timeout* is given, it specifies a point in time where the waiting should be aborted. It can be either an integer as returned by `current-time` or a pair as returned by `gettimeofday`. When the waiting is aborted, `#f` is returned.

For standard mutexes (`make-mutex`), an error is signalled if the thread has itself already locked *mutex*.

For a recursive mutex (`make-recursive-mutex`), if the thread has itself already locked *mutex*, then a further `lock-mutex` call increments the lock count. An additional `unlock-mutex` will be required to finally release.

When an asynchronous interrupt (see Section 6.22.3 [Asyncs], page 445) is scheduled for a thread blocked in `lock-mutex`, Guile will interrupt the wait, run the interrupts, and then resume the wait.

`void scm_dynwind_lock_mutex` (*SCM mutex*) [C Function]

Arrange for *mutex* to be locked whenever the current dynwind context is entered and to be unlocked when it is exited.

`try-mutex` *mx* [Scheme Procedure]
`scm_try_mutex` (*mx*) [C Function]

Try to lock *mutex* and return `#t` if successful, or `#f` otherwise. This is like calling `lock-mutex` with an expired timeout.

`unlock-mutex` *mutex* [Scheme Procedure]
`scm_unlock_mutex` (*mutex*) [C Function]

Unlock *mutex*. An error is signalled if *mutex* is not locked.

"Standard" and "recursive" mutexes can only be unlocked by the thread that locked them; Guile detects this situation and signals an error. "Unowned" mutexes can be unlocked by any thread.

`mutex-owner` *mutex* [Scheme Procedure]
`scm_mutex_owner` (*mutex*) [C Function]

Return the current owner of *mutex*, in the form of a thread or `#f` (indicating no owner). Note that a mutex may be unowned but still locked.

`mutex-level` *mutex* [Scheme Procedure]
`scm_mutex_level` (*mutex*) [C Function]

Return the current lock level of *mutex*. If *mutex* is currently unlocked, this value will be 0; otherwise, it will be the number of times *mutex* has been recursively locked by its current owner.

`mutex-locked?` *mutex* [Scheme Procedure]
`scm_mutex_locked_p` (*mutex*) [C Function]

Return `#t` if *mutex* is locked, regardless of ownership; otherwise, return `#f`.

`make-condition-variable` [Scheme Procedure]
`scm_make_condition_variable ()` [C Function]
Return a new condition variable.

`condition-variable?` *obj* [Scheme Procedure]
`scm_condition_variable_p` (*obj*) [C Function]
Return `#t` if *obj* is a condition variable; otherwise, return `#f`.

`wait-condition-variable` *condvar mutex* [*time*] [Scheme Procedure]
`scm_wait_condition_variable` (*condvar, mutex, time*) [C Function]
Wait until *condvar* has been signalled. While waiting, *mutex* is atomically unlocked (as with `unlock-mutex`) and is locked again when this function returns. When *time* is given, it specifies a point in time where the waiting should be aborted. It can be either a integer as returned by `current-time` or a pair as returned by `gettimeofday`. When the waiting is aborted, `#f` is returned. When the condition variable has in fact been signalled, `#t` is returned. The mutex is re-locked in any case before `wait-condition-variable` returns.

When an async is activated for a thread that is blocked in a call to `wait-condition-variable`, the waiting is interrupted, the mutex is locked, and the async is executed. When the async returns, the mutex is unlocked again and the waiting is resumed. When the thread block while re-acquiring the mutex, execution of asyncs is blocked.

`signal-condition-variable` *condvar* [Scheme Procedure]
`scm_signal_condition_variable` (*condvar*) [C Function]
Wake up one thread that is waiting for *condvar*.

`broadcast-condition-variable` *condvar* [Scheme Procedure]
`scm_broadcast_condition_variable` (*condvar*) [C Function]
Wake up all threads that are waiting for *condvar*.

Guile also includes some higher-level abstractions for working with mutexes.

`with-mutex` *mutex body1 body2 ...* [macro]
Lock *mutex*, evaluate the body *body1 body2 ...*, then unlock *mutex*. The return value is that returned by the last body form.

The lock, body and unlock form the branches of a `dynamic-wind` (see Section 6.13.10 [Dynamic Wind], page 320), so *mutex* is automatically unlocked if an error or new continuation exits the body, and is re-locked if the body is re-entered by a captured continuation.

`monitor` *body1 body2 ...* [macro]
Evaluate the body form *body1 body2 ...* with a mutex locked so only one thread can execute that code at any one time. The return value is the return from the last body form.

Each `monitor` form has its own private mutex and the locking and evaluation is as per `with-mutex` above. A standard mutex (`make-mutex`) is used, which means the body must not recursively re-enter the `monitor` form.

The term "monitor" comes from operating system theory, where it means a particular bit of code managing access to some resource and which only ever executes on behalf of one process at any one time.

6.22.6 Blocking in Guile Mode

Up to Guile version 1.8, a thread blocked in guile mode would prevent the garbage collector from running. Thus threads had to explicitly leave guile mode with `scm_without_guile ()` before making a potentially blocking call such as a mutex lock, a `select ()` system call, etc. The following functions could be used to temporarily leave guile mode or to perform some common blocking operations in a supported way.

Starting from Guile 2.0, blocked threads no longer hinder garbage collection. Thus, the functions below are not needed anymore. They can still be used to inform the GC that a thread is about to block, giving it a (small) optimization opportunity for "stop the world" garbage collections, should they occur while the thread is blocked.

`void * scm_without_guile` (*void *(*func) (void *), void *data*) [C Function]
Leave guile mode, call *func* on *data*, enter guile mode and return the result of calling *func*.

While a thread has left guile mode, it must not call any libguile functions except `scm_with_guile` or `scm_without_guile` and must not use any libguile macros. Also, local variables of type `SCM` that are allocated while not in guile mode are not protected from the garbage collector.

When used from non-guile mode, calling `scm_without_guile` is still allowed: it simply calls *func*. In that way, you can leave guile mode without having to know whether the current thread is in guile mode or not.

`int scm_pthread_mutex_lock` (*pthread_mutex_t *mutex*) [C Function]
Like `pthread_mutex_lock`, but leaves guile mode while waiting for the mutex.

`int scm_pthread_cond_wait` (*pthread_cond_t *cond, pthread_mutex_t *mutex*) [C Function]
`int scm_pthread_cond_timedwait` (*pthread_cond_t *cond, pthread_mutex_t *mutex, struct timespec *abstime*) [C Function]
Like `pthread_cond_wait` and `pthread_cond_timedwait`, but leaves guile mode while waiting for the condition variable.

`int scm_std_select` (*int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout*) [C Function]
Like `select` but leaves guile mode while waiting. Also, the delivery of an async causes this function to be interrupted with error code `EINTR`.

`unsigned int scm_std_sleep` (*unsigned int seconds*) [C Function]
Like `sleep`, but leaves guile mode while sleeping. Also, the delivery of an async causes this function to be interrupted.

`unsigned long scm_std_usleep` (*unsigned long usecs*) [C Function]
Like `usleep`, but leaves guile mode while sleeping. Also, the delivery of an async causes this function to be interrupted.

6.22.7 Futures

The `(ice-9 futures)` module provides *futures*, a construct for fine-grain parallelism. A future is a wrapper around an expression whose computation may occur in parallel with the code of the calling thread, and possibly in parallel with other futures. Like promises, futures are essentially proxies that can be queried to obtain the value of the enclosed expression:

```
(touch (future (+ 2 3)))
⇒ 5
```

However, unlike promises, the expression associated with a future may be evaluated on another CPU core, should one be available. This supports *fine-grain parallelism*, because even relatively small computations can be embedded in futures. Consider this sequential code:

```
(define (find-prime lst1 lst2)
  (or (find prime? lst1)
      (find prime? lst2)))
```

The two arms of `or` are potentially computation-intensive. They are independent of one another, yet, they are evaluated sequentially when the first one returns `#f`. Using futures, one could rewrite it like this:

```
(define (find-prime lst1 lst2)
  (let ((f (future (find prime? lst2))))
    (or (find prime? lst1)
        (touch f))))
```

This preserves the semantics of `find-prime`. On a multi-core machine, though, the computation of `(find prime? lst2)` may be done in parallel with that of the other `find` call, which can reduce the execution time of `find-prime`.

Futures may be nested: a future can itself spawn and then `touch` other futures, leading to a directed acyclic graph of futures. Using this facility, a parallel `map` procedure can be defined along these lines:

```
(use-modules (ice-9 futures) (ice-9 match))

(define (par-map proc lst)
  (match lst
    (()
     '())
    ((head tail ...)
     (let ((tail (future (par-map proc tail)))
           (head (proc head)))
       (cons head (touch tail))))))
```

Note that futures are intended for the evaluation of purely functional expressions. Expressions that have side-effects or rely on I/O may require additional care, such as explicit synchronization (see Section 6.22.5 [Mutexes and Condition Variables], page 448).

Guile's futures are implemented on top of POSIX threads (see Section 6.22.1 [Threads], page 442). Internally, a fixed-size pool of threads is used to evaluate futures, such that offloading the evaluation of an expression to another thread doesn't incur thread creation costs. By default, the pool contains one thread per available CPU core, minus one, to

account for the main thread. The number of available CPU cores is determined using `current-processor-count` (see Section 7.2.7 [Processes], page 518).

When a thread touches a future that has not completed yet, it processes any pending future while waiting for it to complete, or just waits if there are no pending futures. When `touch` is called from within a future, the execution of the calling future is suspended, allowing its host thread to process other futures, and resumed when the touched future has completed. This suspend/resume is achieved by capturing the calling future's continuation, and later reinstating it (see Section 6.13.5 [Prompts], page 303).

`future` *exp* [Scheme Syntax]

Return a future for expression *exp*. This is equivalent to:

```
(make-future (lambda () exp))
```

`make-future` *thunk* [Scheme Procedure]

Return a future for *thunk*, a zero-argument procedure.

This procedure returns immediately. Execution of *thunk* may begin in parallel with the calling thread's computations, if idle CPU cores are available, or it may start when `touch` is invoked on the returned future.

If the execution of *thunk* throws an exception, that exception will be re-thrown when `touch` is invoked on the returned future.

`future?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a future.

`touch` *f* [Scheme Procedure]

Return the result of the expression embedded in future *f*.

If the result was already computed in parallel, `touch` returns instantaneously. Otherwise, it waits for the computation to complete, if it already started, or initiates it. In the former case, the calling thread may process other futures in the meantime.

6.22.8 Parallel forms

The functions described in this section are available from

```
(use-modules (ice-9 threads))
```

They provide high-level parallel constructs. The following functions are implemented in terms of futures (see Section 6.22.7 [Futures], page 453). Thus they are relatively cheap as they re-use existing threads, and portable, since they automatically use one thread per available CPU core.

`parallel` *expr ...* [syntax]

Evaluate each *expr* expression in parallel, each in its own thread. Return the results of *n* expressions as a set of *n* multiple values (see Section 6.13.7 [Multiple Values], page 309).

`letpar` ((*var expr*) *...*) *body1 body2 ...* [syntax]

Evaluate each *expr* in parallel, each in its own thread, then bind the results to the corresponding *var* variables, and then evaluate *body1 body2 ...*

`letpar` is like `let` (see Section 6.12.2 [Local Bindings], page 294), but all the expressions for the bindings are evaluated in parallel.

`par-map` *proc lst1 lst2 ...* [Scheme Procedure]
`par-for-each` *proc lst1 lst2 ...* [Scheme Procedure]

Call *proc* on the elements of the given lists. `par-map` returns a list comprising the return values from *proc*. `par-for-each` returns an unspecified value, but waits for all calls to complete.

The *proc* calls are (`proc elem1 elem2 ...`), where each *elem* is from the corresponding *lst* . Each *lst* must be the same length. The calls are potentially made in parallel, depending on the number of CPU cores available.

These functions are like `map` and `for-each` (see Section 6.6.9.8 [List Mapping], page 186), but make their *proc* calls in parallel.

Unlike those above, the functions described below take a number of threads as an argument. This makes them inherently non-portable since the specified number of threads may differ from the number of available CPU cores as returned by `current-processor-count` (see Section 7.2.7 [Processes], page 518). In addition, these functions create the specified number of threads when they are called and terminate them upon completion, which makes them quite expensive.

Therefore, they should be avoided.

`n-par-map` *n proc lst1 lst2 ...* [Scheme Procedure]
`n-par-for-each` *n proc lst1 lst2 ...* [Scheme Procedure]

Call *proc* on the elements of the given lists, in the same way as `par-map` and `par-for-each` above, but use no more than *n* threads at any one time. The order in which calls are initiated within that threads limit is unspecified.

These functions are good for controlling resource consumption if *proc* calls might be costly, or if there are many to be made. On a dual-CPU system for instance $n = 4$ might be enough to keep the CPUs utilized, and not consume too much memory.

`n-for-each-par-map` *n sproc pproc lst1 lst2 ...* [Scheme Procedure]

Apply *pproc* to the elements of the given lists, and apply *sproc* to each result returned by *pproc*. The final return value is unspecified, but all calls will have been completed before returning.

The calls made are (`sproc (pproc elem1 ... elemN)`), where each *elem* is from the corresponding *lst*. Each *lst* must have the same number of elements.

The *pproc* calls are made in parallel, in separate threads. No more than *n* threads are used at any one time. The order in which *pproc* calls are initiated within that limit is unspecified.

The *sproc* calls are made serially, in list element order, one at a time. *pproc* calls on later elements may execute in parallel with the *sproc* calls. Exactly which thread makes each *sproc* call is unspecified.

This function is designed for individual calculations that can be done in parallel, but with results needing to be handled serially, for instance to write them to a file. The *n* limit on threads controls system resource usage when there are many calculations or when they might be costly.

It will be seen that `n-for-each-par-map` is like a combination of `n-par-map` and `for-each`,

```
(for-each sproc (n-par-map n pproc lst1 ... lstN))
```

But the actual implementation is more efficient since each *sproc* call, in turn, can be initiated once the relevant *pproc* call has completed, it doesn't need to wait for all to finish.

6.23 Configuration, Features and Runtime Options

Why is my Guile different from your Guile? There are three kinds of possible variation:

- build differences — different versions of the Guile source code, installation directories, configuration flags that control pieces of functionality being included or left out, etc.
- differences in dynamically loaded code — behaviour and features provided by modules that can be dynamically loaded into a running Guile
- different runtime options — some of the options that are provided for controlling Guile's behaviour may be set differently.

Guile provides "introspective" variables and procedures to query all of these possible variations at runtime. For runtime options, it also provides procedures to change the settings of options and to obtain documentation on what the options mean.

6.23.1 Configuration, Build and Installation

The following procedures and variables provide information about how Guile was configured, built and installed on your system.

`version` [Scheme Procedure]
`effective-version` [Scheme Procedure]
`major-version` [Scheme Procedure]
`minor-version` [Scheme Procedure]
`micro-version` [Scheme Procedure]
`scm_version ()` [C Function]
`scm_effective_version ()` [C Function]
`scm_major_version ()` [C Function]
`scm_minor_version ()` [C Function]
`scm_micro_version ()` [C Function]

Return a string describing Guile's full version number, effective version number, major, minor or micro version number, respectively. The `effective-version` function returns the version name that should remain unchanged during a stable series. Currently that means that it omits the micro version. The effective version should be used for items like the versioned share directory name i.e. `/usr/share/guile/3.0/`

```
(version) ⇒ "3.0.0"
(effective-version) ⇒ "3.0"
(major-version) ⇒ "3"
(minor-version) ⇒ "0"
(micro-version) ⇒ "0"
```

`%package-data-dir` [Scheme Procedure]
`scm_sys_package_data_dir ()` [C Function]

Return the name of the directory under which Guile Scheme files in general are stored. On Unix-like systems, this is usually `/usr/local/share/guile` or `/usr/share/guile`.

`%library-dir` [Scheme Procedure]
`scm_sys_library_dir ()` [C Function]

Return the name of the directory where the Guile Scheme files that belong to the core Guile installation (as opposed to files from a 3rd party package) are installed. On Unix-like systems this is usually `/usr/local/share/guile/`*`GUILE_EFFECTIVE_VERSION`* or `/usr/share/guile/`*`GUILE_EFFECTIVE_VERSION`*;

for example `/usr/local/share/guile/3.0`.

`%site-dir` [Scheme Procedure]
`scm_sys_site_dir ()` [C Function]

Return the name of the directory where Guile Scheme files specific to your site should be installed. On Unix-like systems, this is usually `/usr/local/share/guile/site` or `/usr/share/guile/site`.

`%site-ccache-dir` [Scheme Procedure]
`scm_sys_site_ccache_dir ()` [C Function]

Return the directory where users should install compiled `.go` files for use with this version of Guile. Might look something like `/usr/lib/guile/3.0/site-ccache`.

`%guile-build-info` [Variable]

Alist of information collected during the building of a particular Guile. Entries can be grouped into one of several categories: directories, env vars, and versioning info.

Briefly, here are the keys in `%guile-build-info`, by group:

directories srcdir, top_srcdir, prefix, exec_prefix, bindir, sbindir, libexecdir, datadir, sysconfdir, sharedstatedir, localstatedir, libdir, infodir, mandir, includedir, pkgdatadir, pkglibdir, pkgincludedir

env vars LIBS

versioning info
guileversion, libguileinterface, buildstamp

Values are all strings. The value for `LIBS` is typically found also as a part of `pkg-config --libs guile-3.0` output. The value for `guileversion` has form X.Y.Z, and should be the same as returned by `(version)`. The value for `libguileinterface` is libtool compatible and has form CURRENT:REVISION:AGE (see Section "Library interface versions" in *GNU Libtool*). The value for `buildstamp` is the output of the command '`date -u +'%Y-%m-%d %T'`' (UTC).

In the source, `%guile-build-info` is initialized from libguile/libpath.h, which is completely generated, so deleting this file before a build guarantees up-to-date values for that build.

`%host-type` [Variable]

The canonical host type (GNU triplet) of the host Guile was configured for, e.g., `"x86_64-unknown-linux-gnu"` (see Section "Canonicalizing" in *The GNU Autoconf Manual*).

6.23.2 Feature Tracking

Guile has a Scheme level variable `*features*` that keeps track to some extent of the features that are available in a running Guile. `*features*` is a list of symbols, for example `threads`, each of which describes a feature of the running Guile process.

`*features*` [Variable]
A list of symbols describing available features of the Guile process.

You shouldn't modify the `*features*` variable directly using `set!`. Instead, see the procedures that are provided for this purpose in the following subsection.

6.23.2.1 Feature Manipulation

To check whether a particular feature is available, use the `provided?` procedure:

`provided?` *feature* [Scheme Procedure]
`feature?` *feature* [Deprecated Scheme Procedure]
Return `#t` if the specified *feature* is available, otherwise `#f`.

To advertise a feature from your own Scheme code, you can use the `provide` procedure:

`provide` *feature* [Scheme Procedure]
Add *feature* to the list of available features in this Guile process.

For C code, the equivalent function takes its feature name as a `char *` argument for convenience:

`void scm_add_feature` (*const char *str*) [C Function]
Add a symbol with name *str* to the list of available features in this Guile process.

6.23.2.2 Common Feature Symbols

In general, a particular feature may be available for one of two reasons. Either because the Guile library was configured and compiled with that feature enabled — i.e. the feature is built into the library on your system. Or because some C or Scheme code that was dynamically loaded by Guile has added that feature to the list.

In the first category, here are the features that the current version of Guile may define (depending on how it is built), and what they mean.

`array` Indicates support for arrays (see Section 6.6.13 [Arrays], page 200).

`array-for-each`
Indicates availability of `array-for-each` and other array mapping procedures (see Section 6.6.13 [Arrays], page 200).

`char-ready?`
Indicates that the `char-ready?` function is available (see Section 6.14.11 [Venerable Port Interfaces], page 350).

`complex` Indicates support for complex numbers.

`current-time`
Indicates availability of time-related functions: `times`, `get-internal-run-time` and so on (see Section 7.2.5 [Time], page 513).

`debug-extensions`
: Indicates that the debugging evaluator is available, together with the options for controlling it.

`delay`
: Indicates support for promises (see Section 6.18.9 [Delayed Evaluation], page 396).

`EIDs`
: Indicates that the `geteuid` and `getegid` really return effective user and group IDs (see Section 7.2.7 [Processes], page 518).

`inexact`
: Indicates support for inexact numbers.

`i/o-extensions`
: Indicates availability of the following extended I/O procedures: `ftell`, `redirect-port`, `dup->fdes`, `dup2`, `fileno`, `isatty?`, `fdopen`, `primitive-move->fdes` and `fdes->ports` (see Section 7.2.2 [Ports and File Descriptors], page 497).

`net-db`
: Indicates availability of network database functions: `scm_gethost`, `scm_getnet`, `scm_getproto`, `scm_getserv`, `scm_sethost`, `scm_setnet`, `scm_setproto`, `scm_setserv`, and their 'byXXX' variants (see Section 7.2.11.2 [Network Databases], page 532).

`posix`
: Indicates support for POSIX functions: `pipe`, `getgroups`, `kill`, `execl` and so on (see Section 7.2 [POSIX], page 496).

`fork`
: Indicates support for the POSIX `fork` function (see Section 7.2.7 [Processes], page 518).

`popen`
: Indicates support for `open-pipe` in the `(ice-9 popen)` module (see Section 7.2.10 [Pipes], page 529).

`random`
: Indicates availability of random number generation functions: `random`, `copy-random-state`, `random-uniform` and so on (see Section 6.6.2.14 [Random], page 127).

`reckless`
: Indicates that Guile was built with important checks omitted — you should never see this!

`regex`
: Indicates support for POSIX regular expressions using `make-regexp`, `regexp-exec` and friends (see Section 6.15.1 [Regexp Functions], page 359).

`socket`
: Indicates availability of socket-related functions: `socket`, `bind`, `connect` and so on (see Section 7.2.11.4 [Network Sockets and Communication], page 540).

`sort`
: Indicates availability of sorting and merging functions (see Section 6.11.3 [Sorting], page 286).

`system`
: Indicates that the `system` function is available (see Section 7.2.7 [Processes], page 518).

`threads`
: Indicates support for multithreading (see Section 6.22.1 [Threads], page 442).

`values`
: Indicates support for multiple return values using `values` and `call-with-values` (see Section 6.13.7 [Multiple Values], page 309).

Available features in the second category depend, by definition, on what additional code your Guile process has loaded in. The following table lists features that you might encounter for this reason.

`defmacro` Indicates that the `defmacro` macro is available (see Section 6.10 [Macros], page 261).

`describe` Indicates that the `(oop goops describe)` module has been loaded, which provides a procedure for describing the contents of GOOPS instances.

`readline` Indicates that Guile has loaded in Readline support, for command line editing (see Section 7.9 [Readline Support], page 711).

`record` Indicates support for record definition using `make-record-type` and friends (see Section 6.6.17 [Records], page 221).

Although these tables may seem exhaustive, it is probably unwise in practice to rely on them, as the correspondences between feature symbols and available procedures/behaviour are not strictly defined. If you are writing code that needs to check for the existence of some procedure, it is probably safer to do so directly using the `defined?` procedure than to test for the corresponding feature using `provided?`.

6.23.3 Runtime Options

There are a number of runtime options available for paramaterizing built-in procedures, like `read`, and built-in behavior, like what happens on an uncaught error.

For more information on reader options, See Section 6.18.2 [Scheme Read], page 385.

For more information on print options, See Section 6.18.3 [Scheme Write], page 386.

Finally, for more information on debugger options, See Section 6.26.3.5 [Debug Options], page 484.

6.23.3.1 Examples of option use

Here is an example of a session in which some read and debug option handling procedures are used. In this example, the user

1. Notices that the symbols `abc` and `aBc` are not the same
2. Examines the `read-options`, and sees that `case-insensitive` is set to "no".
3. Enables `case-insensitive`
4. Quits the recursive prompt
5. Verifies that now `aBc` and `abc` are the same

```
scheme@(guile-user)> (define abc "hello")
scheme@(guile-user)> abc
$1 = "hello"
scheme@(guile-user)> aBc
<unknown-location>: warning: possibly unbound variable ‘aBc’
ERROR: In procedure module-lookup:
ERROR: Unbound variable: aBc
Entering a new prompt.  Type ‘,bt’ for a backtrace or ‘,q’ to continue.
scheme@(guile-user) [1]> (read-options 'help)
copy              no    Copy source code expressions.
positions         yes   Record positions of source code expressions.
case-insensitive  no    Convert symbols to lower case.
```

```
keywords          #f    Style of keyword recognition: #f, 'prefix or 'postfix.
r6rs-hex-escapes  no    Use R6RS variable-length character and string hex escapes.
square-brackets   yes   Treat `[' and `]' as parentheses, for R6RS compatibility.
hungry-eol-escapes no   In strings, consume leading whitespace after an
                        escaped end-of-line.
curly-infix       no    Support SRFI-105 curly infix expressions.
scheme@(guile-user) [1]> (read-enable 'case-insensitive)
$2 = (square-brackets keywords #f case-insensitive positions)
scheme@(guile-user) [1]> ,q
scheme@(guile-user)> aBc
$3 = "hello"
```

6.24 Support for Other Languages

In addition to Scheme, a user may write a Guile program in an increasing number of other languages. Currently supported languages include Emacs Lisp and ECMAScript.

Guile is still fundamentally a Scheme, but it tries to support a wide variety of language building-blocks, so that other languages can be implemented on top of Guile. This allows users to write or extend applications in languages other than Scheme, too. This section describes the languages that have been implemented.

(For details on how to implement a language, See Section 9.4 [Compiling to the Virtual Machine], page 856.)

6.24.1 Using Other Languages

There are currently only two ways to access other languages from within Guile: at the REPL, and programmatically, via `compile`, `read-and-compile`, and `compile-file`.

The REPL is Guile's command prompt (see Section 4.4 [Using Guile Interactively], page 47). The REPL has a concept of the "current language", which defaults to Scheme. The user may change that language, via the meta-command `,language`.

For example, the following meta-command enables Emacs Lisp input:

```
scheme@(guile-user)> ,language elisp
Happy hacking with Emacs Lisp!  To switch back, type `,L scheme'.
elisp@(guile-user)> (eq 1 2)
$1 = #nil
```

Each language has its short name: for example, `elisp`, for Elisp. The same short name may be used to compile source code programmatically, via `compile`:

```
elisp@(guile-user)> ,L scheme
Happy hacking with Guile Scheme!  To switch back, type `,L elisp'.
scheme@(guile-user)> (compile '(eq 1 2) #:from 'elisp)
$2 = #nil
```

Granted, as the input to `compile` is a datum, this works best for Lispy languages, which have a straightforward datum representation. Other languages that need more parsing are better dealt with as strings.

The easiest way to deal with syntax-heavy language is with files, via `compile-file` and friends. However it is possible to invoke a language's reader on a port, and then compile the resulting expression (which is a datum at that point). For more information, See Section 6.18.5 [Compilation], page 389.

For more details on introspecting aspects of different languages, See Section 9.4.1 [Compiler Tower], page 856.

6.24.2 Emacs Lisp

Emacs Lisp (Elisp) is a dynamically-scoped Lisp dialect used in the Emacs editor. See Section "Overview" in *Emacs Lisp*, for more information on Emacs Lisp.

We hope that eventually Guile's implementation of Elisp will be good enough to replace Emacs' own implementation of Elisp. For that reason, we have thought long and hard about how to support the various features of Elisp in a performant and compatible manner.

Readers familiar with Emacs Lisp might be curious about how exactly these various Elisp features are supported in Guile. The rest of this section focuses on addressing these concerns of the Elisp elect.

6.24.2.1 Nil

`nil` in ELisp is an amalgam of Scheme's `#f` and `'()`. It is false, and it is the end-of-list; thus it is a boolean, and a list as well.

Guile has chosen to support `nil` as a separate value, distinct from `#f` and `'()`. This allows existing Scheme and Elisp code to maintain their current semantics. `nil`, which in Elisp would just be written and read as `nil`, in Scheme has the external representation `#nil`.

In Elisp code, `#nil`, `#f`, and `'()` behave like `nil`, in the sense that they are all interpreted as `nil` by Elisp `if`, `cond`, `when`, `not`, `null`, etc. To test whether Elisp would interpret an object as `nil` from within Scheme code, use `nil?`:

`nil?` *obj* [Scheme Procedure]

Return `#t` if *obj* would be interpreted as `nil` by Emacs Lisp code, else return `#f`.

```
(nil? #nil) ⇒ #t
(nil? #f)   ⇒ #t
(nil? '())  ⇒ #t
(nil? 3)    ⇒ #f
```

This decision to have `nil` as a low-level distinct value facilitates interoperability between the two languages. Guile has chosen to have Scheme deal with `nil` as follows:

```
(boolean? #nil) ⇒ #t
(not #nil) ⇒ #t
(null? #nil) ⇒ #t
```

And in C, one has:

```
scm_is_bool (SCM_ELISP_NIL) ⇒ 1
scm_is_false (SCM_ELISP_NIL) ⇒ 1
scm_is_null (SCM_ELISP_NIL) ⇒ 1
```

In this way, a version of `fold` written in Scheme can correctly fold a function written in Elisp (or in fact any other language) over a nil-terminated list, as Elisp makes. The converse holds as well; a version of `fold` written in Elisp can fold over a `'()`-terminated list, as made by Scheme.

On a low level, the bit representations for `#f`, `#t`, `nil`, and `'()` are made in such a way that they differ by only one bit, and so a test for, for example, `#f`-or-`nil` may be made very efficiently. See `libguile/boolean.h`, for more information.

Equality

Since Scheme's `equal?` must be transitive, and `'()` is not `equal?` to `#f`, to Scheme `nil` is not `equal?` to `#f` or `'()`.

```
(eq? #f '()) ⇒ #f
(eq? #nil '()) ⇒ #f
(eq? #nil #f) ⇒ #f
(eqv? #f '()) ⇒ #f
(eqv? #nil '()) ⇒ #f
(eqv? #nil #f) ⇒ #f
(equal? #f '()) ⇒ #f
(equal? #nil '()) ⇒ #f
(equal? #nil #f) ⇒ #f
```

However, in Elisp, `'()`, `#f`, and `nil` are all `equal` (though not `eq`).

```
(defvar f (make-scheme-false))
(defvar eol (make-scheme-null))
(eq f eol) ⇒ nil
(eq nil eol) ⇒ nil
(eq nil f) ⇒ nil
(equal f eol) ⇒ t
(equal nil eol) ⇒ t
(equal nil f) ⇒ t
```

These choices facilitate interoperability between Elisp and Scheme code, but they are not perfect. Some code that is correct standard Scheme is not correct in the presence of a second false and null value. For example:

```
(define (truthiness x)
  (if (eq? x #f)
      #f
      #t))
```

This code seems to be meant to test a value for truth, but now that there are two false values, `#f` and `nil`, it is no longer correct.

Similarly, there is the loop:

```
(define (my-length l)
  (let lp ((l l) (len 0))
    (if (eq? l '())
        len
        (lp (cdr l) (1+ len)))))
```

Here, `my-length` will raise an error if *l* is a `nil`-terminated list.

Both of these examples are correct standard Scheme, but, depending on what they really want to do, they are not correct Guile Scheme. Correctly written, they would test the *properties* of falsehood or nullity, not the individual members of that set. That is to

say, they should use `not` or `null?` to test for falsehood or nullity, not `eq?` or `memv` or the like.

Fortunately, using `not` and `null?` is in good style, so all well-written standard Scheme programs are correct, in Guile Scheme.

Here are correct versions of the above examples:

```
(define (truthiness* x)
  (if (not x)
      #f
      #t))
;; or: (define (t* x) (not (not x)))
;; or: (define (t** x) x)

(define (my-length* l)
  (let lp ((l l) (len 0))
    (if (null? l)
        len
        (lp (cdr l) (1+ len)))))
```

This problem has a mirror-image case in Elisp:

```
(defun my-falsep (x)
  (if (eq x nil)
      t
      nil))
```

Guile can warn when compiling code that has equality comparisons with `#f`, `'()`, or `nil`. See Section 6.18.5 [Compilation], page 389, for details.

6.24.2.2 Dynamic Binding

In contrast to Scheme, which uses "lexical scoping", Emacs Lisp scopes its variables dynamically. Guile supports dynamic scoping with its "fluids" facility. See Section 6.13.11 [Fluids and Dynamic States], page 323, for more information.

6.24.2.3 Other Elisp Features

Buffer-local and mode-local variables should be mentioned here, along with buckybits on characters, Emacs primitive data types, the Lisp-2-ness of Elisp, and other things. Contributions to the documentation are most welcome!

6.24.3 ECMAScript

ECMAScript (`http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`) was not the first non-Schemey language implemented by Guile, but it was the first implemented for Guile's bytecode compiler. The goal was to support ECMAScript version 3.1, a relatively small language, but the implementor was completely irresponsible and got distracted by other things before finishing the standard library, and even some bits of the syntax. So, ECMAScript does deserve a mention in the manual, but it doesn't deserve an endorsement until its implementation is completed, perhaps by some more responsible hacker.

In the meantime, the charitable user might investigate such invocations as `,L ecmascript` and `cat test-suite/tests/ecmascript.test`.

6.25 Support for Internationalization

Guile provides internationalization[15] support for Scheme programs in two ways. First, procedures to manipulate text and data in a way that conforms to particular cultural conventions (i.e., in a "locale-dependent" way) are provided in the `(ice-9 i18n)`. Second, Guile allows the use of GNU `gettext` to translate program message strings.

6.25.1 Internationalization with Guile

In order to make use of the functions described thereafter, the `(ice-9 i18n)` module must be imported in the usual way:

```
(use-modules (ice-9 i18n))
```

The `(ice-9 i18n)` module provides procedures to manipulate text and other data in a way that conforms to the cultural conventions chosen by the user. Each region of the world or language has its own customs to, for instance, represent real numbers, classify characters, collate text, etc. All these aspects comprise the so-called "cultural conventions" of that region or language.

Computer systems typically refer to a set of cultural conventions as a *locale*. For each particular aspect that comprise those cultural conventions, a *locale category* is defined. For instance, the way characters are classified is defined by the `LC_CTYPE` category, while the language in which program messages are issued to the user is defined by the `LC_MESSAGES` category (see Section 7.2.13 [Locales], page 547, for details).

The procedures provided by this module allow the development of programs that adapt automatically to any locale setting. As we will see later, many of these procedures can optionally take a *locale object* argument. This additional argument defines the locale settings that must be followed by the invoked procedure. When it is omitted, then the current locale settings of the process are followed (see Section 7.2.13 [Locales], page 547).

The following procedures allow the manipulation of such locale objects.

`make-locale` *category-list locale-name* [*base-locale*] [Scheme Procedure]
`scm_make_locale` (*category_list, locale_name, base_locale*) [C Function]

Return a reference to a data structure representing a set of locale datasets. *locale-name* should be a string denoting a particular locale (e.g., `"aa_DJ"`) and *category-list* should be either a list of locale categories or a single category as used with `setlocale` (see Section 7.2.13 [Locales], page 547). Optionally, if `base-locale` is passed, it should be a locale object denoting settings for categories not listed in *category-list*.

The following invocation creates a locale object that combines the use of Swedish for messages and character classification with the default settings for the other categories (i.e., the settings of the default `C` locale which usually represents conventions in use in the USA):

```
(make-locale (list LC_MESSAGES LC_CTYPE) "sv_SE")
```

[15] For concision and style, programmers often like to refer to internationalization as "i18n".

The following example combines the use of Esperanto messages and conventions with monetary conventions from Croatia:

```
(make-locale LC_MONETARY "hr_HR"
             (make-locale LC_ALL "eo_EO"))
```

A `system-error` exception (see Section 6.13.13 [Handling Errors], page 328) is raised by `make-locale` when *locale-name* does not match any of the locales compiled on the system. Note that on non-GNU systems, this error may be raised later, when the locale object is actually used.

`locale?` *obj* [Scheme Procedure]
`scm_locale_p` (*obj*) [C Function]
Return true if *obj* is a locale object.

`%global-locale` [Scheme Variable]
`scm_global_locale` [C Variable]
This variable is bound to a locale object denoting the current process locale as installed using `setlocale ()` (see Section 7.2.13 [Locales], page 547). It may be used like any other locale object, including as a third argument to `make-locale`, for instance.

6.25.2 Text Collation

The following procedures provide support for text collation, i.e., locale-dependent string and character sorting.

`string-locale<?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_lt` (*s1, s2, locale*) [C Function]
`string-locale>?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_gt` (*s1, s2, locale*) [C Function]
`string-locale-ci<?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_lt` (*s1, s2, locale*) [C Function]
`string-locale-ci>?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_gt` (*s1, s2, locale*) [C Function]
Compare strings *s1* and *s2* in a locale-dependent way. If *locale* is provided, it should be locale object (as returned by `make-locale`) and will be used to perform the comparison; otherwise, the current system locale is used. For the `-ci` variants, the comparison is made in a case-insensitive way.

`string-locale-ci=?` *s1 s2* [*locale*] [Scheme Procedure]
`scm_string_locale_ci_eq` (*s1, s2, locale*) [C Function]
Compare strings *s1* and *s2* in a case-insensitive, and locale-dependent way. If *locale* is provided, it should be a locale object (as returned by `make-locale`) and will be used to perform the comparison; otherwise, the current system locale is used.

`char-locale<?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_lt` (*c1, c2, locale*) [C Function]
`char-locale>?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_gt` (*c1, c2, locale*) [C Function]
`char-locale-ci<?` *c1 c2* [*locale*] [Scheme Procedure]

`scm_char_locale_ci_lt` (*c1, c2, locale*) [C Function]
`char-locale-ci>?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_ci_gt` (*c1, c2, locale*) [C Function]
Compare characters *c1* and *c2* according to either *locale* (a locale object as returned by `make-locale`) or the current locale. For the `-ci` variants, the comparison is made in a case-insensitive way.

`char-locale-ci=?` *c1 c2* [*locale*] [Scheme Procedure]
`scm_char_locale_ci_eq` (*c1, c2, locale*) [C Function]
Return true if character *c1* is equal to *c2*, in a case insensitive way according to *locale* or to the current locale.

6.25.3 Character Case Mapping

The procedures below provide support for "character case mapping", i.e., to convert characters or strings to their upper-case or lower-case equivalent. Note that SRFI-13 provides procedures that look similar (see Section 6.6.5.9 [Alphabetic Case Mapping], page 154). However, the SRFI-13 procedures are locale-independent. Therefore, they do not take into account specificities of the customs in use in a particular language or region of the world. For instance, while most languages using the Latin alphabet map lower-case letter "i" to upper-case letter "I", Turkish maps lower-case "i" to "Latin capital letter I with dot above". The following procedures allow programmers to provide idiomatic character mapping.

`char-locale-downcase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_upcase` (*chr, locale*) [C Function]
Return the lowercase character that corresponds to *chr* according to either *locale* or the current locale.

`char-locale-upcase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_downcase` (*chr, locale*) [C Function]
Return the uppercase character that corresponds to *chr* according to either *locale* or the current locale.

`char-locale-titlecase` *chr* [*locale*] [Scheme Procedure]
`scm_char_locale_titlecase` (*chr, locale*) [C Function]
Return the titlecase character that corresponds to *chr* according to either *locale* or the current locale.

`string-locale-upcase` *str* [*locale*] [Scheme Procedure]
`scm_string_locale_upcase` (*str, locale*) [C Function]
Return a new string that is the uppercase version of *str* according to either *locale* or the current locale.

`string-locale-downcase` *str* [*locale*] [Scheme Procedure]
`scm_string_locale_downcase` (*str, locale*) [C Function]
Return a new string that is the down-case version of *str* according to either *locale* or the current locale.

`string-locale-titlecase` *str* [*locale*] [Scheme Procedure]
`scm_string_locale_titlecase` (*str, locale*) [C Function]
Return a new string that is the titlecase version of *str* according to either *locale* or the current locale.

6.25.4 Number Input and Output

The following procedures allow programs to read and write numbers written according to a particular locale. As an example, in English, "ten thousand and a half" is usually written `10,000.5` while in French it is written `10 000,5`. These procedures allow such differences to be taken into account.

`locale-string->integer` *str* [*base* [*locale*]] [Scheme Procedure]
`scm_locale_string_to_integer` (*str, base, locale*) [C Function]
Convert string *str* into an integer according to either *locale* (a locale object as returned by `make-locale`) or the current process locale. If *base* is specified, then it determines the base of the integer being read (e.g., `16` for an hexadecimal number, `10` for a decimal number); by default, decimal numbers are read. Return two values (see Section 6.13.7 [Multiple Values], page 309): an integer (on success) or `#f`, and the number of characters read from *str* (`0` on failure).

This function is based on the C library's `strtol` function (see Section "Parsing of Integers" in *The GNU C Library Reference Manual*).

`locale-string->inexact` *str* [*locale*] [Scheme Procedure]
`scm_locale_string_to_inexact` (*str, locale*) [C Function]
Convert string *str* into an inexact number according to either *locale* (a locale object as returned by `make-locale`) or the current process locale. Return two values (see Section 6.13.7 [Multiple Values], page 309): an inexact number (on success) or `#f`, and the number of characters read from *str* (`0` on failure).

This function is based on the C library's `strtod` function (see Section "Parsing of Floats" in *The GNU C Library Reference Manual*).

`number->locale-string` *number* [*fraction-digits* [*locale*]] [Scheme Procedure]
Convert *number* (an inexact) into a string according to the cultural conventions of either *locale* (a locale object) or the current locale. By default, print as many fractional digits as necessary, up to an upper bound. Optionally, *fraction-digits* may be bound to an integer specifying the number of fractional digits to be displayed.

`monetary-amount->locale-string` *amount intl?* [*locale*] [Scheme Procedure]
Convert *amount* (an inexact denoting a monetary amount) into a string according to the cultural conventions of either *locale* (a locale object) or the current locale. If *intl?* is true, then the international monetary format for the given locale is used (see Section "Currency Symbol" in *The GNU C Library Reference Manual*).

6.25.5 Accessing Locale Information

It is sometimes useful to obtain very specific information about a locale such as the word it uses for days or months, its format for representing floating-point figures, etc. The `(ice-9`

`i18n)` module provides support for this in a way that is similar to the libc functions `nl_langinfo ()` and `localeconv ()` (see Section "Locale Information" in *The GNU C Library Reference Manual*). The available functions are listed below.

`locale-encoding` [*locale*] [Scheme Procedure]
: Return the name of the encoding (a string whose interpretation is system-dependent) of either *locale* or the current locale.

The following functions deal with dates and times.

`locale-day` *day* [*locale*] [Scheme Procedure]
`locale-day-short` *day* [*locale*] [Scheme Procedure]
`locale-month` *month* [*locale*] [Scheme Procedure]
`locale-month-short` *month* [*locale*] [Scheme Procedure]
: Return the word (a string) used in either *locale* or the current locale to name the day (or month) denoted by *day* (or *month*), an integer between 1 and 7 (or 1 and 12). The `-short` variants provide an abbreviation instead of a full name.

`locale-am-string` [*locale*] [Scheme Procedure]
`locale-pm-string` [*locale*] [Scheme Procedure]
: Return a (potentially empty) string that is used to denote *ante meridiem* (or *post meridiem*) hours in 12-hour format.

`locale-date+time-format` [*locale*] [Scheme Procedure]
`locale-date-format` [*locale*] [Scheme Procedure]
`locale-time-format` [*locale*] [Scheme Procedure]
`locale-time+am/pm-format` [*locale*] [Scheme Procedure]
`locale-era-date-format` [*locale*] [Scheme Procedure]
`locale-era-date+time-format` [*locale*] [Scheme Procedure]
`locale-era-time-format` [*locale*] [Scheme Procedure]
: These procedures return format strings suitable to `strftime` (see Section 7.2.5 [Time], page 513) that may be used to display (part of) a date/time according to certain constraints and to the conventions of either *locale* or the current locale (see Section "The Elegant and Fast Way" in *The GNU C Library Reference Manual*).

`locale-era` [*locale*] [Scheme Procedure]
`locale-era-year` [*locale*] [Scheme Procedure]
: These functions return, respectively, the era and the year of the relevant era used in *locale* or the current locale. Most locales do not define this value. In this case, the empty string is returned. An example of a locale that does define this value is the Japanese one.

The following procedures give information about number representation.

`locale-decimal-point` [*locale*] [Scheme Procedure]
`locale-thousands-separator` [*locale*] [Scheme Procedure]
: These functions return a string denoting the representation of the decimal point or that of the thousand separator (respectively) for either *locale* or the current locale.

`locale-digit-grouping` [*locale*] [Scheme Procedure]

Return a (potentially circular) list of integers denoting how digits of the integer part of a number are to be grouped, starting at the decimal point and going to the left. The list contains integers indicating the size of the successive groups, from right to left. If the list is non-circular, then no grouping occurs for digits beyond the last group.

For instance, if the returned list is a circular list that contains only `3` and the thousand separator is `","` (as is the case with English locales), then the number `12345678` should be printed `12,345,678`.

The following procedures deal with the representation of monetary amounts. Some of them take an additional *intl?* argument (a boolean) that tells whether the international or local monetary conventions for the given locale are to be used.

`locale-monetary-decimal-point` [*locale*] [Scheme Procedure]
`locale-monetary-thousands-separator` [*locale*] [Scheme Procedure]
`locale-monetary-grouping` [*locale*] [Scheme Procedure]

These are the monetary counterparts of the above procedures. These procedures apply to monetary amounts.

`locale-currency-symbol` *intl?* [*locale*] [Scheme Procedure]

Return the currency symbol (a string) of either *locale* or the current locale.

The following example illustrates the difference between the local and international monetary formats:

```
(define us (make-locale LC_MONETARY "en_US"))
(locale-currency-symbol #f us)
⇒ "-$"
(locale-currency-symbol #t us)
⇒ "USD "
```

`locale-monetary-fractional-digits` *intl?* [*locale*] [Scheme Procedure]

Return the number of fractional digits to be used when printing monetary amounts according to either *locale* or the current locale. If the locale does not specify it, then `#f` is returned.

`locale-currency-symbol-precedes-positive?` *intl?* [*locale*] [Scheme Procedure]
`locale-currency-symbol-precedes-negative?` *intl?* [*locale*] [Scheme Procedure]
`locale-positive-separated-by-space?` *intl?* [*locale*] [Scheme Procedure]
`locale-negative-separated-by-space?` *intl?* [*locale*] [Scheme Procedure]

These procedures return a boolean indicating whether the currency symbol should precede a positive/negative number, and whether a whitespace should be inserted between the currency symbol and a positive/negative amount.

`locale-monetary-positive-sign` [*locale*] [Scheme Procedure]
`locale-monetary-negative-sign` [*locale*] [Scheme Procedure]

Return a string denoting the positive (respectively negative) sign that should be used when printing a monetary amount.

`locale-positive-sign-position` [Scheme Procedure]
`locale-negative-sign-position` [Scheme Procedure]

These functions return a symbol telling where a sign of a positive/negative monetary amount is to appear when printing it. The possible values are:

`parenthesize`
: The currency symbol and quantity should be surrounded by parentheses.

`sign-before`
: Print the sign string before the quantity and currency symbol.

`sign-after`
: Print the sign string after the quantity and currency symbol.

`sign-before-currency-symbol`
: Print the sign string right before the currency symbol.

`sign-after-currency-symbol`
: Print the sign string right after the currency symbol.

`unspecified`
: Unspecified. We recommend you print the sign after the currency symbol.

Finally, the two following procedures may be helpful when programming user interfaces:

`locale-yes-regexp` [*locale*] [Scheme Procedure]
`locale-no-regexp` [*locale*] [Scheme Procedure]

Return a string that can be used as a regular expression to recognize a positive (respectively, negative) response to a yes/no question. For the C locale, the default values are typically `"^[yY]"` and `"^[nN]"`, respectively.

Here is an example:

```
(use-modules (ice-9 rdelim))
(format #t "Does Guile rock?~%")
(let lp ((answer (read-line)))
  (cond ((string-match (locale-yes-regexp) answer)
         (format #t "High fives!~%"))
        ((string-match (locale-no-regexp) answer)
         (format #t "How about now? Does it rock yet?~%")
         (lp (read-line)))
        (else
         (format #t "What do you mean?~%")
         (lp (read-line)))))
```

For an internationalized yes/no string output, `gettext` should be used (see Section 6.25.6 [Gettext Support], page 472).

Example uses of some of these functions are the implementation of the `number->locale-string` and `monetary-amount->locale-string` procedures (see Section 6.25.4 [Number Input and Output], page 468), as well as that the SRFI-19 date and time conversion to/from strings (see Section 7.5.16 [SRFI-19], page 614).

6.25.6 Gettext Support

Guile provides an interface to GNU `gettext` for translating message strings (see Section "Introduction" in *GNU* `gettext` *utilities*).

Messages are collected in domains, so different libraries and programs maintain different message catalogues. The *domain* parameter in the functions below is a string (it becomes part of the message catalog filename).

When `gettext` is not available, or if Guile was configured '`--without-nls`', dummy functions doing no translation are provided. When `gettext` support is available in Guile, the `i18n` feature is provided (see Section 6.23.2 [Feature Tracking], page 458).

`gettext` *msg* [*domain* [*category*]] [Scheme Procedure]
`scm_gettext` (*msg*, *domain*, *category*) [C Function]

Return the translation of *msg* in *domain*. *domain* is optional and defaults to the domain set through `textdomain` below. *category* is optional and defaults to `LC_MESSAGES` (see Section 7.2.13 [Locales], page 547).

Normal usage is for *msg* to be a literal string. `xgettext` can extract those from the source to form a message catalogue ready for translators (see Section "Invoking the `xgettext` Program" in *GNU* `gettext` *utilities*).

```
(display (gettext "You are in a maze of twisty passages."))
```

It is conventional to use `G_` as a shorthand for `gettext`.[16] Libraries can define `G_` in such a way to look up translations using its specific *domain*, allowing different parts of a program to have different translation sources.

```
(define (G_ msg) (gettext msg "mylibrary"))
(display (G_ "File not found."))
```

`G_` is also a good place to perhaps strip disambiguating extra text from the message string, as for instance in Section "How to use `gettext` in GUI programs" in *GNU* `gettext` *utilities*.

`ngettext` *msg msgplural n* [*domain* [*category*]] [Scheme Procedure]
`scm_ngettext` (*msg*, *msgplural*, *n*, *domain*, *category*) [C Function]

Return the translation of *msg*/*msgplural* in *domain*, with a plural form chosen appropriately for the number *n*. *domain* is optional and defaults to the domain set through `textdomain` below. *category* is optional and defaults to `LC_MESSAGES` (see Section 7.2.13 [Locales], page 547).

msg is the singular form, and *msgplural* the plural. When no translation is available, *msg* is used if $n = 1$, or *msgplural* otherwise. When translated, the message catalogue can have a different rule, and can have more than two possible forms.

As per `gettext` above, normal usage is for *msg* and *msgplural* to be literal strings, since `xgettext` can extract them from the source to build a message catalogue. For example,

```
(define (done n)
```

[16] Users of `gettext` might be a bit surprised that `G_` is the conventional abbreviation for `gettext`. In most other languages, the conventional shorthand is `_`. Guile uses `G_` because `_` is already taken, as it is bound to a syntactic keyword used by `syntax-rules`, `match`, and other macros.

```
(format #t (ngettext "~a file processed\n"
                     "~a files processed\n" n)
           n))

(done 1) ⊣ 1 file processed
(done 3) ⊣ 3 files processed
```

It's important to use `ngettext` rather than plain `gettext` for plurals, since the rules for singular and plural forms in English are not the same in other languages. Only `ngettext` will allow translators to give correct forms (see Section "Additional functions for plural forms" in *GNU* `gettext` *utilities*).

`textdomain` [*domain*] [Scheme Procedure]
`scm_textdomain` (*domain*) [C Function]

Get or set the default gettext domain. When called with no parameter the current domain is returned. When called with a parameter, *domain* is set as the current domain, and that new value returned. For example,

```
(textdomain "myprog")
⇒ "myprog"
```

`bindtextdomain` *domain* [*directory*] [Scheme Procedure]
`scm_bindtextdomain` (*domain, directory*) [C Function]

Get or set the directory under which to find message files for *domain*. When called without a *directory* the current setting is returned. When called with a *directory*, *directory* is set for *domain* and that new setting returned. For example,

```
(bindtextdomain "myprog" "/my/tree/share/locale")
⇒ "/my/tree/share/locale"
```

When using Autoconf/Automake, an application should arrange for the configured `localedir` to get into the program (by substituting, or by generating a config file) and set that for its domain. This ensures the catalogue can be found even when installed in a non-standard location.

`bind-textdomain-codeset` *domain* [*encoding*] [Scheme Procedure]
`scm_bind_textdomain_codeset` (*domain, encoding*) [C Function]

Get or set the text encoding to be used by `gettext` for messages from *domain*. *encoding* is a string, the name of a coding system, for instance `"8859_1"`. (On a Unix/POSIX system the `iconv` program can list all available encodings.)

When called without an *encoding* the current setting is returned, or `#f` if none yet set. When called with an *encoding*, it is set for *domain* and that new setting returned. For example,

```
(bind-textdomain-codeset "myprog")
⇒ #f
(bind-textdomain-codeset "myprog" "latin-9")
⇒ "latin-9"
```

The encoding requested can be different from the translated data file, messages will be recoded as necessary. But note that when there is no translation, `gettext` returns its *msg* unchanged, ie. without any recoding. For that reason source message strings are best as plain ASCII.

Currently Guile has no understanding of multi-byte characters, and string functions won't recognise character boundaries in multi-byte strings. An application will at least be able to pass such strings through to some output though. Perhaps this will change in the future.

6.26 Debugging Infrastructure

In order to understand Guile's debugging facilities, you first need to understand a little about how Guile represents the Scheme control stack. With that in place we explain the low level trap calls that the virtual machine can be configured to make, and the trap and breakpoint infrastructure that builds on top of those calls.

6.26.1 Evaluation and the Scheme Stack

The idea of the Scheme stack is central to a lot of debugging. The Scheme stack is a reified representation of the pending function returns in an expression's continuation. As Guile implements function calls using a stack, this reification takes the form of a number of nested stack frames, each of which corresponds to the application of a procedure to a set of arguments.

A Scheme stack always exists implicitly, and can be summoned into concrete existence as a first-class Scheme value by the `make-stack` call, so that an introspective Scheme program – such as a debugger – can present it in some way and allow the user to query its details. The first thing to understand, therefore, is how Guile's function call convention creates the stack.

Broadly speaking, Guile represents all control flow on a stack. Calling a function involves pushing an empty frame on the stack, then evaluating the procedure and its arguments, then fixing up the new frame so that it points to the old one. Frames on the stack are thus linked together. A tail call is the same, except it reuses the existing frame instead of pushing on a new one.

In this way, the only frames that are on the stack are "active" frames, frames which need to do some work before the computation is complete. On the other hand, a function that has tail-called another function will not be on the stack, as it has no work left to do.

Therefore, when an error occurs in a running program, or the program hits a breakpoint, or in fact at any point that the programmer chooses, its state at that point can be represented by a *stack* of all the procedure applications that are logically in progress at that time, each of which is known as a *frame*. The programmer can learn more about the program's state at that point by inspecting the stack and its frames.

6.26.1.1 Stack Capture

A Scheme program can use the `make-stack` primitive anywhere in its code, with first arg `#t`, to construct a Scheme value that describes the Scheme stack at that point.

```
(make-stack #t)
⇒
#<stack 25205a0>
```

Use `start-stack` to limit the stack extent captured by future `make-stack` calls.

`make-stack` *obj arg ...* [Scheme Procedure]
`scm_make_stack` (*obj, args*) [C Function]

Create a new stack. If *obj* is `#t`, the current evaluation stack is used for creating the stack frames, otherwise the frames are taken from *obj* (which must be a continuation or a frame object).

arg ... can be any combination of integer, procedure, address range, and prompt tag values.

These values specify various ways of cutting away uninteresting stack frames from the top and bottom of the stack that `make-stack` returns. They come in pairs like this: (*inner_cut_1 outer_cut_1 inner_cut_2 outer_cut_2 ...*).

Each *inner_cut_i* can be an integer, a procedure, an address range, or a prompt tag. An integer means to cut away exactly that number of frames. A procedure means to cut away all frames up to but excluding the frame whose procedure matches the specified one. An address range is a pair of integers indicating the low and high addresses of a procedure's code, and is the same as cutting away to a procedure (though with less work). Anything else is interpreted as a prompt tag which cuts away all frames that are inside a prompt with the given tag.

Each *outer_cut_i* can likewise be an integer, a procedure, an address range, or a prompt tag. An integer means to cut away that number of frames. A procedure means to cut away frames down to but excluding the frame whose procedure matches the specified one. An address range is the same, but with the procedure's code specified as an address range. Anything else is taken to be a prompt tag, which cuts away all frames that are outside a prompt with the given tag.

If the *outer_cut_i* of the last pair is missing, it is taken as 0.

`start-stack` *id exp* [Scheme Syntax]

Evaluate *exp* on a new calling stack with identity *id*. If *exp* is interrupted during evaluation, backtraces will not display frames farther back than *exp*'s top-level form. This macro is a way of artificially limiting backtraces and stack procedures, largely as a convenience to the user.

6.26.1.2 Stacks

`stack?` *obj* [Scheme Procedure]
`scm_stack_p` (*obj*) [C Function]

Return `#t` if *obj* is a calling stack.

`stack-id` *stack* [Scheme Procedure]
`scm_stack_id` (*stack*) [C Function]

Return the identifier given to *stack* by `start-stack`.

`stack-length` *stack* [Scheme Procedure]
`scm_stack_length` (*stack*) [C Function]

Return the length of *stack*.

`stack-ref` *stack index* [Scheme Procedure]
`scm_stack_ref` (*stack, index*) [C Function]

Return the *index*'th frame from *stack*.

`display-backtrace` *stack port* [*first* [*depth* [*highlights*]]] [Scheme Procedure]
`scm_display_backtrace_with_highlights` (*stack, port, first, depth, highlights*) [C Function]
`scm_display_backtrace` (*stack, port, first, depth*) [C Function]
Display a backtrace to the output port *port*. *stack* is the stack to take the backtrace from, *first* specifies where in the stack to start and *depth* how many frames to display. *first* and *depth* can be `#f`, which means that default values will be used. If *highlights* is given it should be a list; the elements of this list will be highlighted wherever they appear in the backtrace.

6.26.1.3 Frames

`frame?` *obj* [Scheme Procedure]
`scm_frame_p` (*obj*) [C Function]
Return `#t` if *obj* is a stack frame.

`frame-previous` *frame* [Scheme Procedure]
`scm_frame_previous` (*frame*) [C Function]
Return the previous frame of *frame*, or `#f` if *frame* is the first frame in its stack.

`frame-procedure-name` *frame* [Scheme Procedure]
`scm_frame_procedure_name` (*frame*) [C Function]
Return the name of the procedure being applied in *frame*, as a symbol, or `#f` if the procedure has no name.

`frame-arguments` *frame* [Scheme Procedure]
`scm_frame_arguments` (*frame*) [C Function]
Return the arguments of *frame*.

`frame-address` *frame* [Scheme Procedure]
`frame-instruction-pointer` *frame* [Scheme Procedure]
`frame-stack-pointer` *frame* [Scheme Procedure]
Accessors for the three VM registers associated with this frame: the frame pointer (fp), instruction pointer (ip), and stack pointer (sp), respectively. See Section 9.3.2 [VM Concepts], page 828, for more information.

`frame-dynamic-link` *frame* [Scheme Procedure]
`frame-return-address` *frame* [Scheme Procedure]
`frame-mv-return-address` *frame* [Scheme Procedure]
Accessors for the three saved VM registers in a frame: the previous frame pointer, the single-value return address, and the multiple-value return address. See Section 9.3.3 [Stack Layout], page 829, for more information.

`frame-bindings` *frame* [Scheme Procedure]
Return a list of binding records indicating the local variables that are live in a frame.

`frame-lookup-binding` *frame var* [Scheme Procedure]
Fetch the bindings in *frame*, and return the first one whose name is *var*, or `#f` otherwise.

`binding-index` *binding* [Scheme Procedure]
`binding-name` *binding* [Scheme Procedure]
`binding-slot` *binding* [Scheme Procedure]
`binding-representation` *binding* [Scheme Procedure]
Accessors for the various fields in a binding. The implicit "callee" argument is index 0, the first argument is index 1, and so on to the end of the arguments. After that are temporary variables. Note that if a variable is dead, it might not be available.

`binding-ref` *binding* [Scheme Procedure]
`binding-set!` *binding val* [Scheme Procedure]
Accessors for the values of local variables in a frame.

`display-application` *frame* [*port* [*indent*]] [Scheme Procedure]
`scm_display_application` (*frame, port, indent*) [C Function]
Display a procedure application *frame* to the output port *port*. *indent* specifies the indentation of the output.

Additionally, the `(system vm frame)` module defines a number of higher-level introspective procedures, for example to retrieve the names of local variables, and the source location to correspond to a frame. See its source code for more details.

6.26.2 Source Properties

As Guile reads in Scheme code from file or from standard input, it remembers the file name, line number and column number where each expression begins. These pieces of information are known as the *source properties* of the expression. Syntax expanders and the compiler propagate these source properties to compiled procedures, so that, if an error occurs when evaluating the transformed expression, Guile's debugger can point back to the file and location where the expression originated.

The way that source properties are stored means that Guile cannot associate source properties with individual symbols, keywords, characters, booleans, or small integers. This can be seen by typing `(xxx)` and `xxx` at the Guile prompt (where the variable `xxx` has not been defined):

```
scheme@(guile-user)> (xxx)
<unnamed port>:4:1: In procedure module-lookup:
<unnamed port>:4:1: Unbound variable: xxx

scheme@(guile-user)> xxx
ERROR: In procedure module-lookup:
ERROR: Unbound variable: xxx
```

In the latter case, no source properties were stored, so the error doesn't have any source information.

`supports-source-properties?` *obj* [Scheme Procedure]
`scm_supports_source_properties_p` (*obj*) [C Function]
Return #t if source properties can be associated with *obj*, otherwise return #f.

The recording of source properties is controlled by the read option named "positions" (see Section 6.18.2 [Scheme Read], page 385). This option is switched *on* by default.

The following procedures can be used to access and set the source properties of read expressions.

`set-source-properties!` *obj alist* [Scheme Procedure]
`scm_set_source_properties_x` (*obj, alist*) [C Function]
Install the association list *alist* as the source property list for *obj*.

`set-source-property!` *obj key datum* [Scheme Procedure]
`scm_set_source_property_x` (*obj, key, datum*) [C Function]
Set the source property of object *obj*, which is specified by *key* to *datum*. Normally, the key will be a symbol.

`source-properties` *obj* [Scheme Procedure]
`scm_source_properties` (*obj*) [C Function]
Return the source property association list of *obj*.

`source-property` *obj key* [Scheme Procedure]
`scm_source_property` (*obj, key*) [C Function]
Return the property specified by *key* from *obj*'s source properties.

If the `positions` reader option is enabled, supported expressions will have values set for the `filename`, `line` and `column` properties.

Source properties are also associated with syntax objects. Procedural macros can get at the source location of their input using the `syntax-source` accessor. See Section 6.10.4 [Syntax Transformer Helpers], page 273, for more.

Guile also defines a couple of convenience macros built on `syntax-source`:

`current-source-location` [Scheme Syntax]
Expands to the source properties corresponding to the location of the `(current-source-location)` form.

`current-filename` [Scheme Syntax]
Expands to the current filename: the filename that the `(current-filename)` form appears in. Expands to `#f` if this information is unavailable.

If you're stuck with defmacros (see Section 6.10.5 [Defmacros], page 275), and want to preserve source information, the following helper function might be useful to you:

`cons-source` *xorig x y* [Scheme Procedure]
`scm_cons_source` (*xorig, x, y*) [C Function]
Create and return a new pair whose car and cdr are *x* and *y*. Any source properties associated with *xorig* are also associated with the new pair.

6.26.3 Programmatic Error Handling

For better or for worse, all programs have bugs, and dealing with bugs is part of programming. This section deals with that class of bugs that causes an exception to be raised – from your own code, from within a library, or from Guile itself.

6.26.3.1 Catching Exceptions

A common requirement is to be able to show as much useful context as possible when a Scheme program hits an error. The most immediate information about an error is the kind of error that it is – such as "division by zero" – and any parameters that the code which signalled the error chose explicitly to provide. This information originates with the `error` or `raise-exception` call (or their C code equivalents, if the error is detected by C code) that signals the error, and is passed automatically to the handler procedure of the innermost applicable exception handler.

Therefore, to catch errors that occur within a chunk of Scheme code, and to intercept basic information about those errors, you need to execute that code inside the dynamic context of a `with-exception-handler`, or the equivalent in C.

For example, to print out a message and return #f when an error occurs, you might use:

```
(define (catch-all thunk)
  (with-exception-handler
    (lambda (exn)
      (format (current-error-port)
              "Uncaught exception: ~s\n" exn)
      #f)
    thunk
    #:unwind? #t))

(catch-all
 (lambda () (error "Not a vegetable: tomato")))
⊣ Uncaught exception: #<&exception-with-kind-and-args ...>
⇒ #f
```

See Section 6.13.8 [Exceptions], page 311, for full details.

6.26.3.2 Pre-Unwind Debugging

Sometimes when something goes wrong, what you want is not just a representation of the exceptional situation, but the context that brought about that situation. The example in the previous section passed `#:unwind #t` to `with-exception-handler`, indicating that `raise-exception` should unwind the stack before invoking the exception handler. However if you don't take this approach and instead let the exception handler be invoked in the context of the `raise-exception`, you can print a backtrace, launch a recursive debugger, or take other "pre-unwind" actions.

The most basic idea would be to simply print a backtrace:

```
(define (call-with-backtrace thunk)
  (with-exception-handler
    (lambda (exn)
      (backtrace)
      (raise-exception exn))
    thunk))
```

Here we use the built-in `backtrace` procedure to print the backtrace.

`backtrace` [*highlights*] [Scheme Procedure]
`scm_backtrace_with_highlights` (*highlights*) [C Function]

`scm_backtrace ()` [C Function]
Display a backtrace of the current stack to the current output port. If *highlights* is given it should be a list; the elements of this list will be highlighted wherever they appear in the backtrace.

By re-raising the exception, `call-with-backtrace` doesn't actually handle the error. We could define a version that instead aborts the computation:

```
(use-modules (ice-9 control))
(define (call-with-backtrace thunk)
  (let/ec cancel
    (with-exception-handler
      (lambda (exn)
        (backtrace)
        (cancel #f))
      thunk)))
```

In this second example, we use an escape continuation to abort the computation after printing the backtrace, returning `#f` instead.

It could be that you want to only print a limited backtrace. In that case, use `start-stack`:

```
(use-modules (ice-9 control))
(define (call-with-backtrace thunk)
  (let/ec cancel
    (start-stack 'stack-with-backtrace
      (with-exception-handler
        (lambda (exn)
          (backtrace)
          (cancel #f))
        thunk))))
```

There are also more powerful, programmatic ways to walk the stack using `make-stack` and friends; see the API described in Section 6.26.1.2 [Stacks], page 475, and Section 6.26.1.3 [Frames], page 476.

6.26.3.3 call-with-error-handling

The Guile REPL code (in `system/repl/repl.scm` and related files) uses a `catch` with a pre-unwind handler to capture the stack when an error occurs in an expression that was typed into the REPL, and debug that stack interactively in the context of the error.

These procedures are available for use by user programs, in the `(system repl error-handling)` module.

```
(use-modules (system repl error-handling))
```

`call-with-error-handling` *thunk* [*#:on-error on-error='debug*] [*#:post-error post-error='catch*] [*#:pass-keys pass-keys='(quit)*] [*#:report-keys report-keys='(stack-overflow)*] [*#:trap-handler trap-handler='debug*] [Scheme Procedure]
Call a thunk in a context in which errors are handled.

Note that this function was written when `throw/catch` were the fundamental exception handling primitives in Guile, and so exposes some aspects of that interface (notably in the form of the procedural handlers). Guile will probably replace this function with a `call-with-standard-exception-handling` in the future.

There are five keyword arguments:

on-error Specifies what to do before the stack is unwound.

Valid options are `debug` (the default), which will enter a debugger; `pass`, in which case nothing is done, and the exception is rethrown; or a procedure, which will be the pre-unwind handler.

post-error Specifies what to do after the stack is unwound.

Valid options are `catch` (the default), which will silently catch errors, returning the unspecified value; `report`, which prints out a description of the error (via `display-error`), and then returns the unspecified value; or a procedure, which will be the catch handler.

trap-handler

Specifies a trap handler: what to do when a breakpoint is hit.

Valid options are `debug`, which will enter the debugger; `pass`, which does nothing; or `disabled`, which disables traps entirely. See Section 6.26.4 [Traps], page 484, for more information.

pass-keys A set of keys to ignore, as a list.

report-keys

A set of keys to always report even if the post-error handler is `catch`, as a list.

6.26.3.4 Stack Overflow

Every time a Scheme program makes a call that is not in tail position, it pushes a new frame onto the stack. Returning a value from a function pops the top frame off the stack. Stack frames take up memory, and as nobody has an infinite amount of memory, deep recursion could cause Guile to run out of memory. Running out of stack memory is called *stack overflow*.

Stack Limits

Most languages have a terrible stack overflow story. For example, in C, if you use too much stack, your program will exhibit "undefined behavior", which if you are lucky means that it will crash. It's especially bad in C, as you neither know ahead of time how much stack your functions use, nor the stack limit imposed by the user's system, and the stack limit is often quite small relative to the total memory size.

Managed languages like Python have a better error story, as they are defined to raise an exception on stack overflow – but like C, Python and most dynamic languages still have a fixed stack size limit that is usually much smaller than the heap.

Arbitrary stack limits would have an unfortunate effect on Guile programs. For example, the following implementation of the inner loop of `map` is clean and elegant:

```
(define (map f l)
```

```
(if (pair? l)
    (cons (f (car l))
          (map f (cdr l)))
    '())
```

However, if there were a stack limit, that would limit the size of lists that can be processed with this `map`. Eventually, you would have to rewrite it to use iteration with an accumulator:

```
(define (map f l)
  (let lp ((l l) (out '()))
    (if (pair? l)
        (lp (cdr l) (cons (f (car l)) out))
        (reverse out))))
```

This second version is sadly not as clear, and it also allocates more heap memory (once to build the list in reverse, and then again to reverse the list). You would be tempted to use the destructive `reverse!` to save memory and time, but then your code would not be continuation-safe – if *f* returned again after the map had finished, it would see an *out* list that had already been reversed. The recursive `map` has none of these problems.

Guile has no stack limit for Scheme code. When a thread makes its first Guile call, a small stack is allocated – just one page of memory. Whenever that memory limit would be reached, Guile arranges to grow the stack by a factor of two. When garbage collection happens, Guile arranges to return the unused part of the stack to the operating system, but without causing the stack to shrink. In this way, the stack can grow to consume up to all memory available to the Guile process, and when the recursive computation eventually finishes, that stack memory is returned to the system.

Exceptional Situations

Of course, it's still possible to run out of stack memory. The most common cause of this is program bugs that cause unbounded recursion, as in:

```
(define (faulty-map f l)
  (if (pair? l)
      (cons (f (car l)) (faulty-map f l))
      '()))
```

Did you spot the bug? The recursive call to `faulty-map` recursed on *l*, not `(cdr l)`. Running this program would cause Guile to use up all memory in your system, and eventually Guile would fail to grow the stack. At that point you have a problem: Guile needs to raise an exception to unwind the stack and return memory to the system, but the user might have exception handlers in place (see Section 6.13.8.2 [Raising and Handling Exceptions], page 315) that want to run before the stack is unwound, and we don't have any stack in which to run them.

Therefore in this case, Guile raises an unwind-only exception that does not run pre-unwind handlers. Because this is such an odd case, Guile prints out a message on the console, in case the user was expecting to be able to get a backtrace from any pre-unwind handler.

Runaway Recursion

Still, this failure mode is not so nice. If you are running an environment in which you are interactively building a program while it is running, such as at a REPL, you might want to impose an artificial stack limit on the part of your program that you are building to detect accidental runaway recursion. For that purpose, there is `call-with-stack-overflow-handler`, from `(system vm vm)`.

```
(use-module (system vm vm))
```

`call-with-stack-overflow-handler` *limit thunk handler* [Scheme Procedure]

Call *thunk* in an environment in which the stack limit has been reduced to *limit* additional words. If the limit is reached, *handler* (a thunk) will be invoked in the dynamic environment of the error. For the extent of the call to *handler*, the stack limit and handler are restored to the values that were in place when `call-with-stack-overflow-handler` was called.

Usually, *handler* should raise an exception or abort to an outer prompt. However if *handler* does return, it should return a number of additional words of stack space to allow to the inner environment.

A stack overflow handler may only ever "credit" the inner thunk with stack space that was available when the handler was instated. When Guile first starts, there is no stack limit in place, so the outer handler may allow the inner thunk an arbitrary amount of space, but any nested stack overflow handler will not be able to consume more than its limit.

Unlike the unwind-only exception that is thrown if Guile is unable to grow its stack, any exception thrown by a stack overflow handler might invoke pre-unwind handlers. Indeed, the stack overflow handler is itself a pre-unwind handler of sorts. If the code imposing the stack limit wants to protect itself against malicious pre-unwind handlers from the inner thunk, it should abort to a prompt of its own making instead of throwing an exception that might be caught by the inner thunk.

C Stack Usage

It is also possible for Guile to run out of space on the C stack. If you call a primitive procedure which then calls a Scheme procedure in a loop, you will consume C stack space. Guile tries to detect excessive consumption of C stack space, throwing an error when you have hit 80% of the process' available stack (as allocated by the operating system), or 160 kilowords in the absence of a strict limit.

For example, looping through `call-with-vm`, a primitive that calls a thunk, gives us the following:

```
scheme@(guile-user)> (use-modules (system vm vm))
scheme@(guile-user)> (let lp () (call-with-vm lp))
ERROR: Stack overflow
```

Unfortunately, that's all the information we get. Overrunning the C stack will throw an unwind-only exception, because it's not safe to do very much when you are close to the C stack limit.

If you get an error like this, you can either try rewriting your code to use less stack space, or increase the maximum stack size. To increase the maximum stack size, use `debug-set!`, for example:

```
(debug-set! stack 200000)
```

The next section describes `debug-set!` more thoroughly. Of course the best thing is to have your code operate without so much resource consumption by avoiding loops through C trampolines.

6.26.3.5 Debug options

The behavior of the `backtrace` procedure and of the default error handler can be parameterized via the debug options.

`debug-options` [*setting*] [Scheme Procedure]

Display the current settings of the debug options. If *setting* is omitted, only a short form of the current read options is printed. Otherwise if *setting* is the symbol `help`, a complete options description is displayed.

The set of available options, and their default values, may be had by invoking `debug-options` at the prompt.

```
scheme@(guile-user)>
backwards       no      Display backtrace in anti-chronological order.
width           79      Maximal width of backtrace.
depth           20      Maximal length of printed backtrace.
backtrace       yes     Show backtrace on error.
stack           1048576 Stack size limit (measured in words;
                        0 = no check).
show-file-name  #t      Show file names and line numbers in backtraces
                        when not `#f'.  A value of `base' displays only
                        base names, while `#t' displays full names.
warn-deprecated no      Warn when deprecated features are used.
```

The boolean options may be toggled with `debug-enable` and `debug-disable`. The non-boolean options must be set using `debug-set!`.

`debug-enable` *option-name* [Scheme Procedure]
`debug-disable` *option-name* [Scheme Procedure]
`debug-set!` *option-name value* [Scheme Syntax]

Modify the debug options. `debug-enable` should be used with boolean options and switches them on, `debug-disable` switches them off.

`debug-set!` can be used to set an option to a specific value. Due to historical oddities, it is a macro that expects an unquoted option name.

6.26.4 Traps

Guile's virtual machine can be configured to call out at key points to arbitrary user-specified procedures.

In principle, these *hooks* allow Scheme code to implement any model it chooses for examining the evaluation stack as program execution proceeds, and for suspending execution to be resumed later.

VM hooks are very low-level, though, and so Guile also has a library of higher-level *traps* on top of the VM hooks. A trap is an execution condition that, when fulfilled, will fire a handler. For example, Guile defines a trap that fires when control reaches a certain source location.

Finally, Guile also defines a third level of abstractions: per-thread *trap states*. A trap state exists to give names to traps, and to hold on to the set of traps so that they can be enabled, disabled, or removed. The trap state infrastructure defines the most useful abstractions for most cases. For example, Guile's REPL uses trap state functions to set breakpoints and tracepoints.

The following subsections describe all this in detail, for both the user wanting to use traps, and the developer interested in understanding how the interface hangs together.

6.26.4.1 VM Hooks

Everything that runs in Guile runs on its virtual machine, a C program that defines a number of operations that Scheme programs can perform.

Note that there are multiple VM "engines" for Guile. Only some of them have support for hooks compiled in. Normally the deal is that you get hooks if you are running interactively, and otherwise they are disabled, as they do have some overhead (about 10 or 20 percent).

To ensure that you are running with hooks, pass `--debug` to Guile when running your program, or otherwise use the `call-with-vm` and `set-vm-engine!` procedures to ensure that you are running in a VM with the `debug` engine.

To digress, Guile's VM has 4 different hooks that can be fired at different times. For implementation reasons, these hooks are not actually implemented with first-class Scheme hooks (see Section 6.11.6 [Hooks], page 288); they are managed using an ad-hoc interface.

VM hooks are called with one argument: the current frame. See Section 6.26.1.3 [Frames], page 476. Since these hooks may be fired very frequently, Guile does a terrible thing: it allocates the frames on the C stack instead of the garbage-collected heap.

The upshot here is that the frames are only valid within the dynamic extent of the call to the hook. If a hook procedure keeps a reference to the frame outside the extent of the hook, bad things will happen.

The interface to hooks is provided by the `(system vm vm)` module:

```
(use-modules (system vm vm))
```

All of these functions implicitly act on the VM for the current thread only.

`vm-add-next-hook!` *f* [Scheme Procedure]
: Arrange to call *f* when before an instruction is retired (and executed).

`vm-add-apply-hook!` *f* [Scheme Procedure]
: Arrange to call *f* whenever a procedure is applied. The frame locals will be the callee, followed by the arguments to the call.

  Note that procedure application is somewhat orthogonal to continuation pushes and pops. To know whether a call is a tail call or not, with respect to the frame previously in place, check the value of the frame pointer compared the previous frame pointer.

`vm-add-return-hook!` *f* [Scheme Procedure]
: Arrange to call *f* before returning from a frame. The values in the frame will be the frame's return values.

  Note that it's possible to return from an "inner" frame: one that was not immediately proceeded by a call with that frame pointer. In that case, it corresponds to a non-local

control flow jump, either because of applying a composable continuation or because of restoring a saved undelimited continuation.

`vm-add-abort-hook!` [Scheme Procedure]
Arrange to call *f* after aborting to a prompt. See Section 6.13.5 [Prompts], page 303.

Unfortunately, the values passed to the prompt handler are not easily available to *f*.

`vm-remove-next-hook!` *f* [Scheme Procedure]
`vm-remove-apply-hook!` *f* [Scheme Procedure]
`vm-remove-return-hook!` *f* [Scheme Procedure]
`vm-remove-abort-hook!` *f* [Scheme Procedure]
Remove *f* from the corresponding VM hook for the current thread.

These hooks do impose a performance penalty, if they are on. Obviously, the `vm-next-hook` has quite an impact, performance-wise. Therefore Guile exposes a single, heavy-handed knob to turn hooks on or off, the *VM trace level*. If the trace level is positive, hooks run; otherwise they don't.

For convenience, when the VM fires a hook, it does so with the trap level temporarily set to 0. That way the hooks don't fire while you're handling a hook. The trace level is restored to whatever it was once the hook procedure finishes.

`vm-trace-level` [Scheme Procedure]
Retrieve the "trace level" of the VM. If positive, the trace hooks associated with *vm* will be run. The initial trace level is 0.

`set-vm-trace-level!` *level* [Scheme Procedure]
Set the "trace level" of the VM.

See Section 9.3 [A Virtual Machine for Guile], page 827, for more information on Guile's virtual machine.

6.26.4.2 Trap Interface

The capabilities provided by hooks are great, but hooks alone rarely correspond to what users want to do.

For example, if a user wants to break when and if control reaches a certain source location, how do you do it? If you install a "next" hook, you get unacceptable overhead for the execution of the entire program. It would be possible to install an "apply" hook, then if the procedure encompasses those source locations, install a "next" hook, but already you're talking about one concept that might be implemented by a varying number of lower-level concepts.

It's best to be clear about things and define one abstraction for all such conditions: the *trap*.

Considering the myriad capabilities offered by the hooks though, there is only a minimum of functionality shared by all traps. Guile's current take is to reduce this to the absolute minimum, and have the only standard interface of a trap be "turn yourself on" or "turn yourself off".

This interface sounds a bit strange, but it is useful to procedurally compose higher-level traps from lower-level building blocks. For example, Guile defines a trap that calls one

handler when control enters a procedure, and another when control leaves the procedure. Given that trap, one can define a trap that adds to the next-hook only when within a given procedure. Building further, one can define a trap that fires when control reaches particular instructions within a procedure.

Or of course you can stop at any of these intermediate levels. For example, one might only be interested in calls to a given procedure. But the point is that a simple enable/disable interface is all the commonality that exists between the various kinds of traps, and furthermore that such an interface serves to allow "higher-level" traps to be composed from more primitive ones.

Specifically, a trap, in Guile, is a procedure. When a trap is created, by convention the trap is enabled; therefore, the procedure that is the trap will, when called, disable the trap, and return a procedure that will enable the trap, and so on.

Trap procedures take one optional argument: the current frame. (A trap may want to add to different sets of hooks depending on the frame that is current at enable-time.)

If this all sounds very complicated, it's because it is. Some of it is essential, but probably most of it is not. The advantage of using this minimal interface is that composability is more lexically apparent than when, for example, using a stateful interface based on GOOPS. But perhaps this reflects the cognitive limitations of the programmer who made the current interface more than anything else.

6.26.4.3 Low-Level Traps

To summarize the last sections, traps are enabled or disabled, and when they are enabled, they add to various VM hooks.

Note, however, that *traps do not increase the VM trace level*. So if you create a trap, it will be enabled, but unless something else increases the VM's trace level (see Section 6.26.4.1 [VM Hooks], page 485), the trap will not fire. It turns out that getting the VM trace level right is tricky without a global view of what traps are enabled. See Section 6.26.4.5 [Trap States], page 491, for Guile's answer to this problem.

Traps are created by calling procedures. Most of these procedures share a set of common keyword arguments, so rather than document them separately, we discuss them all together here:

`#:vm` The VM to instrument. Defaults to the current thread's VM.

`#:current-frame`
: For traps that enable more hooks depending on their dynamic context, this argument gives the current frame that the trap is running in. Defaults to `#f`.

To have access to these procedures, you'll need to have imported the `(system vm traps)` module:

```
(use-modules (system vm traps))
```

`trap-at-procedure-call` *proc handler* [*#:vm*] [Scheme Procedure]
: A trap that calls *handler* when *proc* is applied.

`trap-in-procedure` *proc enter-handler exit-handler* [*#:current-frame*] [*#:vm*] [Scheme Procedure]
: A trap that calls *enter-handler* when control enters *proc*, and *exit-handler* when control leaves *proc*.

Control can enter a procedure via:

- A procedure call.
- A return to a procedure's frame on the stack.
- A continuation returning directly to an application of this procedure.

Control can leave a procedure via:

- A normal return from the procedure.
- An application of another procedure.
- An invocation of a continuation.
- An abort.

`trap-instructions-in-procedure` *proc next-handler exit-handler* [*#:current-frame*] [*#:vm*] [Scheme Procedure]

A trap that calls *next-handler* for every instruction executed in *proc*, and *exit-handler* when execution leaves *proc*.

`trap-at-procedure-ip-in-range` *proc range handler* [*#:current-frame*] [*#:vm*] [Scheme Procedure]

A trap that calls *handler* when execution enters a range of instructions in *proc*. *range* is a simple of pairs, `((start . end) ...)`. The *start* addresses are inclusive, and *end* addresses are exclusive.

`trap-at-source-location` *file user-line handler* [*#:current-frame*] [*#:vm*] [Scheme Procedure]

A trap that fires when control reaches a given source location. The *user-line* parameter is one-indexed, as a user counts lines, instead of zero-indexed, as Guile counts lines.

`trap-frame-finish` *frame return-handler abort-handler* [*#:vm*] [Scheme Procedure]

A trap that fires when control leaves the given frame. *frame* should be a live frame in the current continuation. *return-handler* will be called on a normal return, and *abort-handler* on a nonlocal exit.

`trap-in-dynamic-extent` *proc enter-handler return-handler abort-handler* [*#:vm*] [Scheme Procedure]

A more traditional dynamic-wind trap, which fires *enter-handler* when control enters *proc*, *return-handler* on a normal return, and *abort-handler* on a nonlocal exit.

Note that rewinds are not handled, so there is no rewind handler.

`trap-calls-in-dynamic-extent` *proc apply-handler return-handler* [*#:current-frame*] [*#:vm*] [Scheme Procedure]

A trap that calls *apply-handler* every time a procedure is applied, and *return-handler* for returns, but only during the dynamic extent of an application of *proc*.

`trap-instructions-in-dynamic-extent` *proc next-handler* [*#:current-frame*] [*#:vm*] [Scheme Procedure]

A trap that calls *next-handler* for all retired instructions within the dynamic extent of a call to *proc*.

`trap-calls-to-procedure` *proc apply-handler return-handler* [*#:vm*] [Scheme Procedure]

A trap that calls *apply-handler* whenever *proc* is applied, and *return-handler* when it returns, but with an additional argument, the call depth.

That is to say, the handlers will get two arguments: the frame in question, and the call depth (a non-negative integer).

`trap-matching-instructions` *frame-pred handler* [*#:vm*] [Scheme Procedure]

A trap that calls *frame-pred* at every instruction, and if *frame-pred* returns a true value, calls *handler* on the frame.

6.26.4.4 Tracing Traps

The `(system vm trace)` module defines a number of traps for tracing of procedure applications. When a procedure is *traced*, it means that every call to that procedure is reported to the user during a program run. The idea is that you can mark a collection of procedures for tracing, and Guile will subsequently print out a line of the form

```
|  |  (procedure args ...)
```

whenever a marked procedure is about to be applied to its arguments. This can help a programmer determine whether a function is being called at the wrong time or with the wrong set of arguments.

In addition, the indentation of the output is useful for demonstrating how the traced applications are or are not tail recursive with respect to each other. Thus, a trace of a non-tail recursive factorial implementation looks like this:

```
scheme@(guile-user)> (define (fact1 n)
                       (if (zero? n) 1
                           (* n (fact1 (1- n)))))
scheme@(guile-user)> ,trace (fact1 4)
trace: (fact1 4)
trace: |  (fact1 3)
trace: |  |  (fact1 2)
trace: |  |  |  (fact1 1)
trace: |  |  |  |  (fact1 0)
trace: |  |  |  |  1
trace: |  |  |  1
trace: |  |  2
trace: |  6
trace: 24
```

While a typical tail recursive implementation would look more like this:

```
scheme@(guile-user)> (define (facti acc n)
                       (if (zero? n) acc
                           (facti (* n acc) (1- n))))
scheme@(guile-user)> (define (fact2 n) (facti 1 n))
scheme@(guile-user)> ,trace (fact2 4)
trace: (fact2 4)
trace: (facti 1 4)
```

```
trace: (facti 4 3)
trace: (facti 12 2)
trace: (facti 24 1)
trace: (facti 24 0)
trace: 24
```

The low-level traps below (see Section 6.26.4.3 [Low-Level Traps], page 487) share some common options:

`#:width` The maximum width of trace output. Trace printouts will try not to exceed this column, but for highly nested procedure calls, it may be unavoidable. Defaults to 80.

`#:vm` The VM on which to add the traps. Defaults to the current thread's VM.

`#:prefix` A string to print out before each trace line. As seen above in the examples, defaults to `"trace: "`.

To have access to these procedures, you'll need to have imported the `(system vm trace)` module:

```
(use-modules (system vm trace))
```

`trace-calls-to-procedure` *proc* [*#:width*] [*#:vm*] [*#:prefix*] [Scheme Procedure]

Print a trace at applications of and returns from *proc*.

`trace-calls-in-procedure` *proc* [*#:width*] [*#:vm*] [*#:prefix*] [Scheme Procedure]

Print a trace at all applications and returns within the dynamic extent of calls to *proc*.

`trace-instructions-in-procedure` *proc* [*#:width*] [*#:vm*] [Scheme Procedure]

Print a trace at all instructions executed in the dynamic extent of calls to *proc*.

In addition, Guile defines a procedure to call a thunk, tracing all procedure calls and returns within the thunk.

`call-with-trace` *thunk* [*#:calls?=#t*] [*#:instructions?=#f*] [*#:width=80*] [Scheme Procedure]

Call *thunk*, tracing all execution within its dynamic extent.

If *calls?* is true, Guile will print a brief report at each procedure call and return, as given above.

If *instructions?* is true, Guile will also print a message each time an instruction is executed. This is a lot of output, but it is sometimes useful when doing low-level optimization.

Note that because this procedure manipulates the VM trace level directly, it doesn't compose well with traps at the REPL.

See Section 4.4.4.5 [Profile Commands], page 51, for more information on tracing at the REPL.

6.26.4.5 Trap States

When multiple traps are present in a system, we begin to have a bookkeeping problem. How are they named? How does one disable, enable, or delete them?

Guile's answer to this is to keep an implicit per-thread *trap state*. The trap state object is not exposed to the user; rather, API that works on trap states fetches the current trap state from the dynamic environment.

Traps are identified by integers. A trap can be enabled, disabled, or removed, and can have an associated user-visible name.

These procedures have their own module:

```
(use-modules (system vm trap-state))
```

`add-trap!` *trap name* [Scheme Procedure]
Add a trap to the current trap state, associating the given *name* with it. Returns a fresh trap identifier (an integer).

Note that usually the more specific functions detailed in Section 6.26.4.6 [High-Level Traps], page 491, are used in preference to this one.

`list-traps` [Scheme Procedure]
List the current set of traps, both enabled and disabled. Returns a list of integers.

`trap-name` *idx* [Scheme Procedure]
Returns the name associated with trap *idx*, or `#f` if there is no such trap.

`trap-enabled?` *idx* [Scheme Procedure]
Returns `#t` if trap *idx* is present and enabled, or `#f` otherwise.

`enable-trap!` *idx* [Scheme Procedure]
Enables trap *idx*.

`disable-trap!` *idx* [Scheme Procedure]
Disables trap *idx*.

`delete-trap!` *idx* [Scheme Procedure]
Removes trap *idx*, disabling it first, if necessary.

6.26.4.6 High-Level Traps

The low-level trap API allows one to make traps that call procedures, and the trap state API allows one to keep track of what traps are there. But neither of these APIs directly helps you when you want to set a breakpoint, because it's unclear what to do when the trap fires. Do you enter a debugger, or mail a summary of the situation to your great-aunt, or what?

So for the common case in which you just want to install breakpoints, and then have them all result in calls to one parameterizable procedure, we have the high-level trap interface.

Perhaps we should have started this section with this interface, as it's clearly the one most people should use. But as its capabilities and limitations proceed from the lower layers, we felt that the character-building exercise of building a mental model might be helpful.

These procedures share a module with trap states:

```
(use-modules (system vm trap-state))
```

`with-default-trap-handler` *handler thunk* [Scheme Procedure]
Call *thunk* in a dynamic context in which *handler* is the current trap handler.

Additionally, during the execution of *thunk*, the VM trace level (see Section 6.26.4.1 [VM Hooks], page 485) is set to the number of enabled traps. This ensures that traps will in fact fire.

handler may be `#f`, in which case VM hooks are not enabled as they otherwise would be, as there is nothing to handle the traps.

The trace-level-setting behavior of `with-default-trap-handler` is one of its more useful aspects, but if you are willing to forgo that, and just want to install a global trap handler, there's a function for that too:

`install-trap-handler!` *handler* [Scheme Procedure]
Set the current thread's trap handler to *handler*.

Trap handlers are called when traps installed by procedures from this module fire. The current "consumer" of this API is Guile's REPL, but one might easily imagine other trap handlers being used to integrate with other debugging tools.

`add-trap-at-procedure-call!` *proc* [Scheme Procedure]
Install a trap that will fire when *proc* is called.

This is a breakpoint.

`add-trace-at-procedure-call!` *proc* [Scheme Procedure]
Install a trap that will print a tracing message when *proc* is called. See Section 6.26.4.4 [Tracing Traps], page 489, for more information.

This is a tracepoint.

`add-trap-at-source-location!` *file user-line* [Scheme Procedure]
Install a trap that will fire when control reaches the given source location. *user-line* is one-indexed, as users count lines, instead of zero-indexed, as Guile counts lines.

This is a source breakpoint.

`add-ephemeral-trap-at-frame-finish!` *frame handler* [Scheme Procedure]
Install a trap that will call *handler* when *frame* finishes executing. The trap will be removed from the trap state after firing, or on nonlocal exit.

This is a finish trap, used to implement the "finish" REPL command.

`add-ephemeral-stepping-trap!` *frame handler* [*#:into?*] [*#:instruction?*] [Scheme Procedure]
Install a trap that will call *handler* after stepping to a different source line or instruction. The trap will be removed from the trap state after firing, or on nonlocal exit.

If *instruction?* is false (the default), the trap will fire when control reaches a new source line. Otherwise it will fire when control reaches a new instruction.

Additionally, if *into?* is false (not the default), the trap will only fire for frames at or prior to the given frame. If *into?* is true (the default), the trap may step into nested procedure invocations.

This is a stepping trap, used to implement the "step", "next", "step-instruction", and "next-instruction" REPL commands.

6.26.5 GDB Support

Sometimes, you may find it necessary to debug Guile applications at the C level. Doing so can be tedious, in particular because the debugger is oblivious to Guile's `SCM` type, and thus unable to display `SCM` values in any meaningful way:

```
(gdb) frame
#0  scm_display (obj=0xf04310, port=0x6f9f30) at print.c:1437
```

To address that, Guile comes with an extension of the GNU Debugger (GDB) that contains a "pretty-printer" for `SCM` values. With this GDB extension, the C frame in the example above shows up like this:

```
(gdb) frame
#0  scm_display (obj=("hello" GDB!), port=#<port file 6f9f30>) at print.c:1437
```

Here GDB was able to decode the list pointed to by *obj*, and to print it using Scheme's read syntax.

That extension is a `.scm` file installed alongside the `libguile` shared library. When GDB 7.8 or later is installed and compiled with support for extensions written in Guile, the extension is automatically loaded when debugging a program linked against `libguile` (see Section "Auto-loading" in *Debugging with GDB*). Note that the directory where `libguile` is installed must be among GDB's auto-loading "safe directories" (see Section "Auto-loading safe path" in *Debugging with GDB*).

6.27 Code Coverage Reports

When writing a test suite for a program or library, it is desirable to know what part of the code is *covered* by the test suite. The `(system vm coverage)` module provides tools to gather code coverage data and to present them, as detailed below.

`with-code-coverage` *thunk* [Scheme Procedure]

Run *thunk*, a zero-argument procedure, while instrumenting Guile's virtual machine to collect code coverage data. Return code coverage data and the values returned by *thunk*.

`coverage-data?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a *coverage data* object as returned by `with-code-coverage`.

`coverage-data->lcov` *data port #:key modules* [Scheme Procedure]

Traverse code coverage information *data*, as obtained with `with-code-coverage`, and write coverage information to port in the `.info` format used by LCOV (`http://ltp.sourceforge.net/coverage/lcov.php`). The report will include all of *modules* (or, by default, all the currently loaded modules) even if their code was not executed.

The generated data can be fed to LCOV's `genhtml` command to produce an HTML report, which aids coverage data visualization.

Here's an example use:

```
(use-modules (system vm coverage)
             (system vm vm))
```

```
(call-with-values (lambda ()
                    (with-code-coverage
                      (lambda ()
                        (do-something-tricky))))
  (lambda (data result)
    (let ((port (open-output-file "lcov.info")))
      (coverage-data->lcov data port)
      (close file))))
```

In addition, the module provides low-level procedures that would make it possible to write other user interfaces to the coverage data.

`instrumented-source-files` *data* [Scheme Procedures]
Return the list of "instrumented" source files, i.e., source files whose code was loaded at the time *data* was collected.

`line-execution-counts` *data file* [Scheme Procedures]
Return a list of line number/execution count pairs for *file*, or `#f` if *file* is not among the files covered by *data*. This includes lines with zero count.

`instrumented/executed-lines` *data file* [Scheme Procedures]
Return the number of instrumented and the number of executed source lines in *file* according to *data*.

`procedure-execution-count` *data proc* [Scheme Procedures]
Return the number of times *proc*'s code was executed, according to *data*, or `#f` if *proc* was not executed. When *proc* is a closure, the number of times its code was executed is returned, not the number of times this code associated with this particular closure was executed.

7 Guile Modules

7.1 SLIB

SLIB is a portable library of Scheme packages which can be used with Guile and other Scheme implementations. SLIB is not included in the Guile distribution, but can be installed separately (see Section 7.1.1 [SLIB installation], page 495). It is available from `http://people.csail.mit.edu/jaffer/SLIB.html`.

After SLIB is installed, the following Scheme expression must be executed before the SLIB facilities can be used:

```
(use-modules (ice-9 slib))
```

`require` can then be used in the usual way (see Section "Require" in *The SLIB Manual*). For example,

```
(use-modules (ice-9 slib))
(require 'primes)
(prime? 13)
⇒ #t
```

A few Guile core functions are overridden by the SLIB setups; for example the SLIB version of `delete-file` returns a boolean indicating success or failure, whereas the Guile core version throws an error for failure. In general (and as might be expected) when SLIB is loaded it's the SLIB specifications that are followed.

7.1.1 SLIB installation

The following procedure works, e.g., with SLIB version 3a3 (see Section "Installation" in *The SLIB Portable Scheme Library*):

1. Unpack SLIB and install it using `make install` from its directory. By default, this will install SLIB in `/usr/local/lib/slib/`. Running `make install-info` installs its documentation, by default under `/usr/local/info/`.
2. Define the `SCHEME_LIBRARY_PATH` environment variable:

   ```
   $ SCHEME_LIBRARY_PATH=/usr/local/lib/slib/
   $ export SCHEME_LIBRARY_PATH
   ```

   Alternatively, you can create a symlink in the Guile directory to SLIB, e.g.:

   ```
   ln -s /usr/local/lib/slib /usr/local/share/guile/3.0/slib
   ```
3. Use Guile to create the catalog file, e.g.,:

   ```
   # guile
   guile> (use-modules (ice-9 slib))
   guile> (require 'new-catalog)
   guile> (quit)
   ```

   The catalog data should now be in `/usr/local/share/guile/3.0/slibcat`.

   If instead you get an error such as:

   ```
   Unbound variable: scheme-implementation-type
   ```

   then a solution is to get a newer version of Guile, or to modify `ice-9/slib.scm` to use `define-public` for the offending variables.

7.1.2 JACAL

Jacal is a symbolic math package written in Scheme by Aubrey Jaffer. It is usually installed as an extra package in SLIB.

You can use Guile's interface to SLIB to invoke Jacal:

```
(use-modules (ice-9 slib))
(slib:load "math")
(math)
```

For complete documentation on Jacal, please read the Jacal manual. If it has been installed on line, you can look at Section "Jacal" in *JACAL Symbolic Mathematics System*. Otherwise you can find it on the web at `http://www-swiss.ai.mit.edu/~jaffer/JACAL.html`

7.2 POSIX System Calls and Networking

7.2.1 POSIX Interface Conventions

These interfaces provide access to operating system facilities. They provide a simple wrapping around the underlying C interfaces to make usage from Scheme more convenient. They are also used to implement the Guile port of scsh (see Section 7.18 [The Scheme shell (scsh)], page 743).

Generally there is a single procedure for each corresponding Unix facility. There are some exceptions, such as procedures implemented for speed and convenience in Scheme with no primitive Unix equivalent, e.g. `copy-file`.

The interfaces are intended as far as possible to be portable across different versions of Unix. In some cases procedures which can't be implemented on particular systems may become no-ops, or perform limited actions. In other cases they may throw errors.

General naming conventions are as follows:

- The Scheme name is often identical to the name of the underlying Unix facility.
- Underscores in Unix procedure names are converted to hyphens.
- Procedures which destructively modify Scheme data have exclamation marks appended, e.g., `recv!`.
- Predicates (returning only `#t` or `#f`) have question marks appended, e.g., `access?`.
- Some names are changed to avoid conflict with dissimilar interfaces defined by scsh, e.g., `primitive-fork`.
- Unix preprocessor names such as `EPERM` or `R_OK` are converted to Scheme variables of the same name (underscores are not replaced with hyphens).

Unexpected conditions are generally handled by raising exceptions. There are a few procedures which return a special value if they don't succeed, e.g., `getenv` returns `#f` if it the requested string is not found in the environment. These cases are noted in the documentation.

For ways to deal with exceptions, see Section 6.13.8 [Exceptions], page 311.

Errors which the C library would report by returning a null pointer or through some other means are reported by raising a `system-error` exception with `scm-error` (see Section 6.13.9

[Error Reporting], page 319). The *data* parameter is a list containing the Unix `errno` value (an integer). For example,

```
(define (my-handler key func fmt fmtargs data)
  (display key) (newline)
  (display func) (newline)
  (apply format #t fmt fmtargs) (newline)
  (display data) (newline))

(catch 'system-error
  (lambda () (dup2 -123 -456))
  my-handler)

⊣
system-error
dup2
Bad file descriptor
(9)
```

`system-error-errno` *arglist* [Function]

Return the `errno` value from a list which is the arguments to an exception handler. If the exception is not a `system-error`, then the return is `#f`. For example,

```
(catch
 'system-error
 (lambda ()
   (mkdir "/this-ought-to-fail-if-I'm-not-root"))
 (lambda stuff
   (let ((errno (system-error-errno stuff)))
     (cond
      ((= errno EACCES)
       (display "You're not allowed to do that."))
      ((= errno EEXIST)
       (display "Already exists."))
      (#t
       (display (strerror errno))))
     (newline))))
```

7.2.2 Ports and File Descriptors

Conventions generally follow those of scsh, Section 7.18 [The Scheme shell (scsh)], page 743.

Each open file port has an associated operating system file descriptor. File descriptors are generally not useful in Scheme programs; however they may be needed when interfacing with foreign code and the Unix environment.

A file descriptor can be extracted from a port and a new port can be created from a file descriptor. However a file descriptor is just an integer and the garbage collector doesn't recognize it as a reference to the port. If all other references to the port were dropped, then it's likely that the garbage collector would free the port, with the side-effect of closing the file descriptor prematurely.

To assist the programmer in avoiding this problem, each port has an associated *revealed count* which can be used to keep track of how many times the underlying file descriptor has been stored in other places. If a port's revealed count is greater than zero, the file descriptor will not be closed when the port is garbage collected. A programmer can therefore ensure that the revealed count will be greater than zero if the file descriptor is needed elsewhere.

For the simple case where a file descriptor is "imported" once to become a port, it does not matter if the file descriptor is closed when the port is garbage collected. There is no need to maintain a revealed count. Likewise when "exporting" a file descriptor to the external environment, setting the revealed count is not required provided the port is kept open (i.e., is pointed to by a live Scheme binding) while the file descriptor is in use.

To correspond with traditional Unix behaviour, three file descriptors (0, 1, and 2) are automatically imported when a program starts up and assigned to the initial values of the current/standard input, output, and error ports, respectively. The revealed count for each is initially set to one, so that dropping references to one of these ports will not result in its garbage collection: it could be retrieved with `fdopen` or `fdes->ports`.

Guile's ports can be buffered. This means that writing a byte to a file port goes to the internal buffer first, and only when the buffer is full (or the user invokes `force-output` on the port) is the data actually written to the file descriptor. Likewise on input, bytes are read in from the file descriptor in blocks and placed in a buffer. Reading a character via `read-char` first goes to the buffer, filling it as needed. Usually read buffering is more or less transparent, but write buffering can sometimes cause writes to be delayed unexpectedly, if you forget to call `force-output`. See Section 6.14.6 [Buffering], page 339, for more on how to control port buffers.

Note however that some procedures (e.g., `recv!`) will accept ports as arguments, but will actually operate directly on the file descriptor underlying the port. Any port buffering is ignored, including the buffer which implements `peek-char` and `unread-char`.

`port-revealed` *port* [Scheme Procedure]
`scm_port_revealed` (*port*) [C Function]
Return the revealed count for *port*.

`set-port-revealed!` *port rcount* [Scheme Procedure]
`scm_set_port_revealed_x` (*port*, *rcount*) [C Function]
Sets the revealed count for a *port* to *rcount*. The return value is unspecified.

`fileno` *port* [Scheme Procedure]
`scm_fileno` (*port*) [C Function]
Return the integer file descriptor underlying *port*. Does not change its revealed count.

`port->fdes` *port* [Scheme Procedure]
Returns the integer file descriptor underlying *port*. As a side effect the revealed count of *port* is incremented.

`fdopen` *fdes modes* [Scheme Procedure]
`scm_fdopen` (*fdes*, *modes*) [C Function]
Return a new port based on the file descriptor *fdes*. Modes are given by the string *modes*. The revealed count of the port is initialized to zero. The *modes* string is the same as that accepted by `open-file` (see Section 6.14.10.1 [File Ports], page 344).

`fdes->ports` *fdes* [Scheme Procedure]
`scm_fdes_to_ports` (*fdes*) [C Function]
Return a list of existing ports which have *fdes* as an underlying file descriptor, without changing their revealed counts.

`fdes->inport` *fdes* [Scheme Procedure]
Returns an existing input port which has *fdes* as its underlying file descriptor, if one exists, and increments its revealed count. Otherwise, returns a new input port with a revealed count of 1.

`fdes->outport` *fdes* [Scheme Procedure]
Returns an existing output port which has *fdes* as its underlying file descriptor, if one exists, and increments its revealed count. Otherwise, returns a new output port with a revealed count of 1.

`primitive-move->fdes` *port fdes* [Scheme Procedure]
`scm_primitive_move_to_fdes` (*port*, *fdes*) [C Function]
Moves the underlying file descriptor for *port* to the integer value *fdes* without changing the revealed count of *port*. Any other ports already using this descriptor will be automatically shifted to new descriptors and their revealed counts reset to zero. The return value is `#f` if the file descriptor already had the required value or `#t` if it was moved.

`move->fdes` *port fdes* [Scheme Procedure]
Moves the underlying file descriptor for *port* to the integer value *fdes* and sets its revealed count to one. Any other ports already using this descriptor will be automatically shifted to new descriptors and their revealed counts reset to zero. The return value is unspecified.

`release-port-handle` *port* [Scheme Procedure]
Decrements the revealed count for a port.

`fsync` *port_or_fd* [Scheme Procedure]
`scm_fsync` (*port_or_fd*) [C Function]
Copies any unwritten data for the specified output file descriptor to disk. If *port_or_fd* is a port, its buffer is flushed before the underlying file descriptor is fsync'd. The return value is unspecified.

`open` *path flags* [*mode*] [Scheme Procedure]
`scm_open` (*path, flags, mode*) [C Function]
Open the file named by *path* for reading and/or writing. *flags* is an integer specifying how the file should be opened. *mode* is an integer specifying the permission bits of the file, if it needs to be created, before the umask (see Section 7.2.7 [Processes], page 518) is applied. The default is 666 (Unix itself has no default).

flags can be constructed by combining variables using `logior`. Basic flags are:

`O_RDONLY` [Variable]
Open the file read-only.

`O_WRONLY` [Variable]
Open the file write-only.

`O_RDWR` [Variable]
Open the file read/write.

`O_APPEND` [Variable]
Append to the file instead of truncating.

`O_CREAT` [Variable]
Create the file if it does not already exist.

See Section "File Status Flags" in *The GNU C Library Reference Manual*, for additional flags.

`open-fdes` *path flags* [*mode*] [Scheme Procedure]
`scm_open_fdes` (*path, flags, mode*) [C Function]
Similar to `open` but return a file descriptor instead of a port.

`close` *fd_or_port* [Scheme Procedure]
`scm_close` (*fd_or_port*) [C Function]
Similar to `close-port` (see Section 6.14.1 [Ports], page 331), but also works on file descriptors. A side effect of closing a file descriptor is that any ports using that file descriptor are moved to a different file descriptor and have their revealed counts set to zero.

`close-fdes` *fd* [Scheme Procedure]
`scm_close_fdes` (*fd*) [C Function]
A simple wrapper for the `close` system call. Close file descriptor *fd*, which must be an integer. Unlike `close`, the file descriptor will be closed even if a port is using it. The return value is unspecified.

`pipe` [Scheme Procedure]
`scm_pipe` () [C Function]
Return a newly created pipe: a pair of ports which are linked together on the local machine. The CAR is the input port and the CDR is the output port. Data written (and flushed) to the output port can be read from the input port. Pipes are commonly used for communication with a newly forked child process. The need to flush the output port can be avoided by making it unbuffered using `setvbuf` (see Section 6.14.6 [Buffering], page 339).

`PIPE_BUF` [Variable]
A write of up to `PIPE_BUF` many bytes to a pipe is atomic, meaning when done it goes into the pipe instantaneously and as a contiguous block (see Section "Atomicity of Pipe I/O" in *The GNU C Library Reference Manual*).

Note that the output port is likely to block if too much data has been written but not yet read from the input port. Typically the capacity is `PIPE_BUF` bytes.

The next group of procedures perform a `dup2` system call, if *newfd* (an integer) is supplied, otherwise a `dup`. The file descriptor to be duplicated can be supplied as an integer or contained in a port. The type of value returned varies depending on which procedure is used.

All procedures also have the side effect when performing `dup2` that any ports using *newfd* are moved to a different file descriptor and have their revealed counts set to zero.

`dup->fdes` *fd_or_port* [*fd*] [Scheme Procedure]
`scm_dup_to_fdes` (*fd_or_port*, *fd*) [C Function]
Return a new integer file descriptor referring to the open file designated by *fd_or_port*, which must be either an open file port or a file descriptor.

`dup->inport` *port/fd* [*newfd*] [Scheme Procedure]
Returns a new input port using the new file descriptor.

`dup->outport` *port/fd* [*newfd*] [Scheme Procedure]
Returns a new output port using the new file descriptor.

`dup` *port/fd* [*newfd*] [Scheme Procedure]
Returns a new port if *port/fd* is a port, with the same mode as the supplied port, otherwise returns an integer file descriptor.

`dup->port` *port/fd mode* [*newfd*] [Scheme Procedure]
Returns a new port using the new file descriptor. *mode* supplies a mode string for the port (see Section 6.14.10.1 [File Ports], page 344).

`duplicate-port` *port modes* [Scheme Procedure]
Returns a new port which is opened on a duplicate of the file descriptor underlying *port*, with mode string *modes* as for Section 6.14.10.1 [File Ports], page 344. The two ports will share a file position and file status flags.

Unexpected behaviour can result if both ports are subsequently used and the original and/or duplicate ports are buffered. The mode string can include `0` to obtain an unbuffered duplicate port.

This procedure is equivalent to (`dup->port` *port modes*).

`redirect-port` *old_port new_port* [Scheme Procedure]
`scm_redirect_port` (*old_port*, *new_port*) [C Function]
This procedure takes two ports and duplicates the underlying file descriptor from *old_port* into *new_port*. The current file descriptor in *new_port* will be closed. After the redirection the two ports will share a file position and file status flags.

The return value is unspecified.

Unexpected behaviour can result if both ports are subsequently used and the original and/or duplicate ports are buffered.

This procedure does not have any side effects on other ports or revealed counts.

`dup2` *oldfd newfd* [Scheme Procedure]
`scm_dup2` (*oldfd*, *newfd*) [C Function]
A simple wrapper for the `dup2` system call. Copies the file descriptor *oldfd* to descriptor number *newfd*, replacing the previous meaning of *newfd*. Both *oldfd* and *newfd*

must be integers. Unlike for `dup->fdes` or `primitive-move->fdes`, no attempt is made to move away ports which are using *newfd*. The return value is unspecified.

`port-for-each` *proc* [Scheme Procedure]
`scm_port_for_each` (*SCM proc*) [C Function]
`scm_c_port_for_each` (*void* (**proc*)(*void **, *SCM*), *void *data*) [C Function]

Apply *proc* to each port in the Guile port table (FIXME: what is the Guile port table?) in turn. The return value is unspecified. More specifically, *proc* is applied exactly once to every port that exists in the system at the time `port-for-each` is invoked. Changes to the port table while `port-for-each` is running have no effect as far as `port-for-each` is concerned.

The C function `scm_port_for_each` takes a Scheme procedure encoded as a `SCM` value, while `scm_c_port_for_each` takes a pointer to a C function and passes along a arbitrary *data* cookie.

`fcntl` *port/fd cmd* [*value*] [Scheme Procedure]
`scm_fcntl` (*object, cmd, value*) [C Function]

Apply *cmd* on *port/fd*, either a port or file descriptor. The *value* argument is used by the `SET` commands described below, it's an integer value.

Values for *cmd* are:

`F_DUPFD` [Variable]

Duplicate the file descriptor, the same as `dup->fdes` above does.

`F_GETFD` [Variable]
`F_SETFD` [Variable]

Get or set flags associated with the file descriptor. The only flag is the following,

`FD_CLOEXEC` [Variable]

"Close on exec", meaning the file descriptor will be closed on an `exec` call (a successful such call). For example to set that flag,

```
(fcntl port F_SETFD FD_CLOEXEC)
```

Or better, set it but leave any other possible future flags unchanged,

```
(fcntl port F_SETFD (logior FD_CLOEXEC
                            (fcntl port F_GETFD)))
```

`F_GETFL` [Variable]
`F_SETFL` [Variable]

Get or set flags associated with the open file. These flags are `O_RDONLY` etc described under `open` above.

A common use is to set `O_NONBLOCK` on a network socket. The following sets that flag, and leaves other flags unchanged.

```
(fcntl sock F_SETFL (logior O_NONBLOCK
                            (fcntl sock F_GETFL)))
```

`F_GETOWN` [Variable]
`F_SETOWN` [Variable]

Get or set the process ID of a socket's owner, for `SIGIO` signals.

`flock` *file operation* [Scheme Procedure]
`scm_flock` (*file, operation*) [C Function]

Apply or remove an advisory lock on an open file. *operation* specifies the action to be done:

`LOCK_SH` [Variable]
Shared lock. More than one process may hold a shared lock for a given file at a given time.

`LOCK_EX` [Variable]
Exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.

`LOCK_UN` [Variable]
Unlock the file.

`LOCK_NB` [Variable]
Don't block when locking. This is combined with one of the other operations using `logior` (see Section 6.6.2.13 [Bitwise Operations], page 125). If `flock` would block an `EWOULDBLOCK` error is thrown (see Section 7.2.1 [Conventions], page 496).

The return value is not specified. *file* may be an open file descriptor or an open file descriptor port.

Note that `flock` does not lock files across NFS.

`select` *reads writes excepts* [*secs* [*usecs*]] [Scheme Procedure]
`scm_select` (*reads, writes, excepts, secs, usecs*) [C Function]

This procedure has a variety of uses: waiting for the ability to provide input, accept output, or the existence of exceptional conditions on a collection of ports or file descriptors, or waiting for a timeout to occur.

When an error occurs, this procedure throws a `system-error` exception (see Section 7.2.1 [Conventions], page 496). Note that `select` may return early for other reasons, for example due to pending interrupts. See Section 6.22.3 [Asyncs], page 445, for more on interrupts.

reads, *writes* and *excepts* can be lists or vectors, with each member a port or a file descriptor. The value returned is a list of three corresponding lists or vectors containing only the members which meet the specified requirement. The ability of port buffers to provide input or accept output is taken into account. Ordering of the input lists or vectors is not preserved.

The optional arguments *secs* and *usecs* specify the timeout. Either *secs* can be specified alone, as either an integer or a real number, or both *secs* and *usecs* can be specified as integers, in which case *usecs* is an additional timeout expressed in microseconds. If *secs* is omitted or is `#f` then select will wait for as long as it takes for one of the other conditions to be satisfied.

The scsh version of `select` differs as follows: Only vectors are accepted for the first three arguments. The *usecs* argument is not supported. Multiple values are returned instead of a list. Duplicates in the input vectors appear only once in output. An additional `select!` interface is provided.

While it is sometimes necessary to operate at the level of file descriptors, this is an operation whose correctness can only be considered as part of a whole program. So for example while the effects of `(string-set! x 34 #\y)` are limited to the bits of code that can access *x*, `(close-fdes 34)` mutates the state of the entire process. In particular if another thread is using file descriptor 34 then their state might be corrupted; and another thread which opens a file might cause file descriptor 34 to be re-used, so that corruption could manifest itself in a strange way.

However when working with file descriptors, it's common to want to associate information with the file descriptor, perhaps in a side table. To support this use case and to allow user code to remove an association when a file descriptor is closed, Guile offers *fdes finalizers*.

As the name indicates, fdes finalizers are finalizers – they can run in response to garbage collection, and they can also run in response to explicit calls to `close-port`, `close-fdes`, or the like. As such they inherit many of the pitfalls of finalizers: they may be invoked from concurrent threads, or not at all. See Section 5.5.4 [Foreign Object Memory Management], page 77, for more on finalizers.

To use fdes finalizers, import their module;

```
(use-modules (ice-9 fdes-finalizers))
```

`add-fdes-finalizer!` *fdes finalizer* [Scheme Procedure]
`remove-fdes-finalizer!` *fdes finalizer* [Scheme Procedure]

Add or remove a finalizer for *fdes*. A finalizer is a procedure that is called by Guile when a file descriptor is closed. The file descriptor being closed is passed as the one argument to the finalizer. If a finalizer has been added multiple times to a file descriptor, to remove it would require that number of calls to `remove-fdes-finalizer!`.

The finalizers added to a file descriptor are called by Guile in an unspecified order, and their return values are ignored.

7.2.3 File System

These procedures allow querying and setting file system attributes (such as owner, permissions, sizes and types of files); deleting, copying, renaming and linking files; creating and removing directories and querying their contents; syncing the file system and creating special files.

`access?` *path how* [Scheme Procedure]
`scm_access` (*path, how*) [C Function]

Test accessibility of a file under the real UID and GID of the calling process. The return is `#t` if *path* exists and the permissions requested by *how* are all allowed, or `#f` if not.

how is an integer which is one of the following values, or a bitwise-OR (`logior`) of multiple values.

`R_OK` [Variable]

Test for read permission.

`W_OK` [Variable]

Test for write permission.

`X_OK` [Variable]
Test for execute permission.

`F_OK` [Variable]
Test for existence of the file. This is implied by each of the other tests, so there's no need to combine it with them.

It's important to note that `access?` does not simply indicate what will happen on attempting to read or write a file. In normal circumstances it does, but in a set-UID or set-GID program it doesn't because `access?` tests the real ID, whereas an open or execute attempt uses the effective ID.

A program which will never run set-UID/GID can ignore the difference between real and effective IDs, but for maximum generality, especially in library functions, it's best not to use `access?` to predict the result of an open or execute, instead simply attempt that and catch any exception.

The main use for `access?` is to let a set-UID/GID program determine what the invoking user would have been allowed to do, without the greater (or perhaps lesser) privileges afforded by the effective ID. For more on this, see Section "Testing File Access" in *The GNU C Library Reference Manual*.

`stat` *object* [*exception-on-error?*] [Scheme Procedure]
`scm_stat` (*object*, *exception_on_error*) [C Function]
Return an object containing various information about the file determined by *object*. *object* can be a string containing a file name or a port or integer file descriptor which is open on a file (in which case `fstat` is used as the underlying system call).

If the optional *exception_on_error* argument is true, which is the default, an exception will be raised if the underlying system call returns an error, for example if the file is not found or is not readable. Otherwise, an error will cause `stat` to return `#f`.

The object returned by `stat` can be passed as a single parameter to the following procedures, all of which return integers:

`stat:dev` *st* [Scheme Procedure]
The device number containing the file.

`stat:ino` *st* [Scheme Procedure]
The file serial number, which distinguishes this file from all other files on the same device.

`stat:mode` *st* [Scheme Procedure]
The mode of the file. This is an integer which incorporates file type information and file permission bits. See also `stat:type` and `stat:perms` below.

`stat:nlink` *st* [Scheme Procedure]
The number of hard links to the file.

`stat:uid` *st* [Scheme Procedure]
The user ID of the file's owner.

`stat:gid` *st* [Scheme Procedure]
The group ID of the file.

`stat:rdev` *st* [Scheme Procedure]
: Device ID; this entry is defined only for character or block special files. On some systems this field is not available at all, in which case `stat:rdev` returns `#f`.

`stat:size` *st* [Scheme Procedure]
: The size of a regular file in bytes.

`stat:atime` *st* [Scheme Procedure]
: The last access time for the file, in seconds.

`stat:mtime` *st* [Scheme Procedure]
: The last modification time for the file, in seconds.

`stat:ctime` *st* [Scheme Procedure]
: The last modification time for the attributes of the file, in seconds.

`stat:atimensec` *st* [Scheme Procedure]
`stat:mtimensec` *st* [Scheme Procedure]
`stat:ctimensec` *st* [Scheme Procedure]
: The fractional part of a file's access, modification, or attribute modification time, in nanoseconds. Nanosecond timestamps are only available on some operating systems and file systems. If Guile cannot retrieve nanosecond-level timestamps for a file, these fields will be set to 0.

`stat:blksize` *st* [Scheme Procedure]
: The optimal block size for reading or writing the file, in bytes. On some systems this field is not available, in which case `stat:blksize` returns a sensible suggested block size.

`stat:blocks` *st* [Scheme Procedure]
: The amount of disk space that the file occupies measured in units of 512 byte blocks. On some systems this field is not available, in which case `stat:blocks` returns `#f`.

In addition, the following procedures return the information from `stat:mode` in a more convenient form:

`stat:type` *st* [Scheme Procedure]
: A symbol representing the type of file. Possible values are '`regular`', '`directory`', '`symlink`', '`block-special`', '`char-special`', '`fifo`', '`socket`', and '`unknown`'.

`stat:perms` *st* [Scheme Procedure]
: An integer representing the access permission bits.

`lstat` *path* [Scheme Procedure]
`scm_lstat` (*path*) [C Function]
: Similar to `stat`, but does not follow symbolic links, i.e., it will return information about a symbolic link itself, not the file it points to. *path* must be a string.

`readlink` *path* [Scheme Procedure]
`scm_readlink` (*path*) [C Function]
Return the value of the symbolic link named by *path* (a string), i.e., the file that the link points to.

`chown` *object owner group* [Scheme Procedure]
`scm_chown` (*object, owner, group*) [C Function]
Change the ownership and group of the file referred to by *object* to the integer values *owner* and *group*. *object* can be a string containing a file name or, if the platform supports `fchown` (see Section "File Owner" in *The GNU C Library Reference Manual*), a port or integer file descriptor which is open on the file. The return value is unspecified.

If *object* is a symbolic link, either the ownership of the link or the ownership of the referenced file will be changed depending on the operating system (lchown is unsupported at present). If *owner* or *group* is specified as `-1`, then that ID is not changed.

`chmod` *object mode* [Scheme Procedure]
`scm_chmod` (*object, mode*) [C Function]
Changes the permissions of the file referred to by *object*. *object* can be a string containing a file name or a port or integer file descriptor which is open on a file (in which case `fchmod` is used as the underlying system call). *mode* specifies the new permissions as a decimal number, e.g., `(chmod "foo" #o755)`. The return value is unspecified.

`utime` *pathname* [*actime* [*modtime* [*actimens* [*modtimens* [*flags*]]]]] [Scheme Procedure]
`scm_utime` (*pathname, actime, modtime, actimens, modtimens, flags*) [C Function]
`utime` sets the access and modification times for the file named by *pathname*. If *actime* or *modtime* is not supplied, then the current time is used. *actime* and *modtime* must be integer time values as returned by the `current-time` procedure.

The optional *actimens* and *modtimens* are nanoseconds to add *actime* and *modtime*. Nanosecond precision is only supported on some combinations of file systems and operating systems.

```
(utime "foo" (- (current-time) 3600))
```

will set the access time to one hour in the past and the modification time to the current time.

Last, *flags* may be either `0` or the `AT_SYMLINK_NOFOLLOW` constant, to set the time of *pathname* even if it is a symbolic link.

`delete-file` *str* [Scheme Procedure]
`scm_delete_file` (*str*) [C Function]
Deletes (or "unlinks") the file whose path is specified by *str*.

`copy-file` *oldfile newfile* [Scheme Procedure]
`scm_copy_file` (*oldfile, newfile*) [C Function]
Copy the file specified by *oldfile* to *newfile*. The return value is unspecified.

`sendfile` *out in count* [*offset*] [Scheme Procedure]
`scm_sendfile` (*out, in, count, offset*) [C Function]
Send *count* bytes from *in* to *out*, both of which must be either open file ports or file descriptors. When *offset* is omitted, start reading from *in*'s current position; otherwise, start reading at *offset*. Return the number of bytes actually sent.

When *in* is a port, it is often preferable to specify *offset*, because *in*'s offset as a port may be different from the offset of its underlying file descriptor.

On systems that support it, such as GNU/Linux, this procedure uses the `sendfile` libc function, which usually corresponds to a system call. This is faster than doing a series of `read` and `write` system calls. A typical application is to send a file over a socket.

In some cases, the `sendfile` libc function may return `EINVAL` or `ENOSYS`. In that case, Guile's `sendfile` procedure automatically falls back to doing a series of `read` and `write` calls.

In other cases, the libc function may send fewer bytes than *count*—for instance because *out* is a slow or limited device, such as a pipe. When that happens, Guile's `sendfile` automatically retries until exactly *count* bytes were sent or an error occurs.

`rename-file` *oldname newname* [Scheme Procedure]
`scm_rename` (*oldname, newname*) [C Function]
Renames the file specified by *oldname* to *newname*. The return value is unspecified.

`link` *oldpath newpath* [Scheme Procedure]
`scm_link` (*oldpath, newpath*) [C Function]
Creates a new name *newpath* in the file system for the file named by *oldpath*. If *oldpath* is a symbolic link, the link may or may not be followed depending on the system.

`symlink` *oldpath newpath* [Scheme Procedure]
`scm_symlink` (*oldpath, newpath*) [C Function]
Create a symbolic link named *newpath* with the value (i.e., pointing to) *oldpath*. The return value is unspecified.

`mkdir` *path* [*mode*] [Scheme Procedure]
`scm_mkdir` (*path, mode*) [C Function]
Create a new directory named by *path*. If *mode* is omitted then the permissions of the directory are set to `#o777` masked with the current umask (see Section 7.2.7 [Processes], page 518). Otherwise they are set to the value specified with *mode*. The return value is unspecified.

`rmdir` *path* [Scheme Procedure]
`scm_rmdir` (*path*) [C Function]
Remove the existing directory named by *path*. The directory must be empty for this to succeed. The return value is unspecified.

`opendir` *dirname* [Scheme Procedure]
`scm_opendir` (*dirname*) [C Function]
Open the directory specified by *dirname* and return a directory stream.

Before using this and the procedures below, make sure to see the higher-level procedures for directory traversal that are available (see Section 7.12 [File Tree Walk], page 727).

`directory-stream?` *object* [Scheme Procedure]
`scm_directory_stream_p` (*object*) [C Function]
Return a boolean indicating whether *object* is a directory stream as returned by `opendir`.

`readdir` *stream* [Scheme Procedure]
`scm_readdir` (*stream*) [C Function]
Return (as a string) the next directory entry from the directory stream *stream*. If there is no remaining entry to be read then the end of file object is returned.

`rewinddir` *stream* [Scheme Procedure]
`scm_rewinddir` (*stream*) [C Function]
Reset the directory port *stream* so that the next call to `readdir` will return the first directory entry.

`closedir` *stream* [Scheme Procedure]
`scm_closedir` (*stream*) [C Function]
Close the directory stream *stream*. The return value is unspecified.

Here is an example showing how to display all the entries in a directory:

```
(define dir (opendir "/usr/lib"))
(do ((entry (readdir dir) (readdir dir)))
    ((eof-object? entry))
  (display entry)(newline))
(closedir dir)
```

`sync` [Scheme Procedure]
`scm_sync` () [C Function]
Flush the operating system disk buffers. The return value is unspecified.

`mknod` *path type perms dev* [Scheme Procedure]
`scm_mknod` (*path, type, perms, dev*) [C Function]
Creates a new special file, such as a file corresponding to a device. *path* specifies the name of the file. *type* should be one of the following symbols: '`regular`', '`directory`', '`symlink`', '`block-special`', '`char-special`', '`fifo`', or '`socket`'. *perms* (an integer) specifies the file permissions. *dev* (an integer) specifies which device the special file refers to. Its exact interpretation depends on the kind of special file being created.

E.g.,

```
(mknod "/dev/fd0" 'block-special #o660 (+ (* 2 256) 2))
```

The return value is unspecified.

`tmpnam` [Scheme Procedure]
`scm_tmpnam` () [C Function]
Return an auto-generated name of a temporary file, a file which doesn't already exist. The name includes a path, it's usually in `/tmp` but that's system dependent.

Care must be taken when using `tmpnam`. In between choosing the name and creating the file another program might use that name, or an attacker might even make it a symlink pointing at something important and causing you to overwrite that.

The safe way is to create the file using `open` with `O_EXCL` to avoid any overwriting. A loop can try again with another name if the file exists (error `EEXIST`). `mkstemp!` below does that.

`mkstemp!` *tmpl* [*mode*] [Scheme Procedure]
`scm_mkstemp` (*tmpl*) [C Function]

Create a new unique file in the file system and return a new buffered port open for reading and writing to the file.

tmpl is a string specifying where the file should be created: it must end with '`XXXXXX`' and those '`X`'s will be changed in the string to return the name of the file. (`port-filename` on the port also gives the name.)

POSIX doesn't specify the permissions mode of the file, on GNU and most systems it's `#o600`. An application can use `chmod` to relax that if desired. For example `#o666` less `umask`, which is usual for ordinary file creation,

```
(let ((port (mkstemp! (string-copy "/tmp/myfile-XXXXXX"))))
  (chmod port (logand #o666 (lognot (umask))))
  ...)
```

The optional *mode* argument specifies a mode with which to open the new file, as a string in the same format that `open-file` takes. It defaults to `"w+"`.

`tmpfile` [Scheme Procedure]
`scm_tmpfile` () [C Function]

Return an input/output port to a unique temporary file named using the path prefix `P_tmpdir` defined in `stdio.h`. The file is automatically deleted when the port is closed or the program terminates.

`dirname` *filename* [Scheme Procedure]
`scm_dirname` (*filename*) [C Function]

Return the directory name component of the file name *filename*. If *filename* does not contain a directory component, `.` is returned.

`basename` *filename* [*suffix*] [Scheme Procedure]
`scm_basename` (*filename*, *suffix*) [C Function]

Return the base name of the file name *filename*. The base name is the file name without any directory components. If *suffix* is provided, and is equal to the end of *basename*, it is removed also.

```
(basename "/tmp/test.xml" ".xml")
⇒ "test"
```

`canonicalize-path` *path* [Scheme Procedure]
`scm_canonicalize_path` (*path*) [C Function]

Return the canonical (absolute) path of *path*. A canonical path has no `.` or `..` components, nor any repeated path separators (`/`) nor symlinks.

Raises an error if any component of *path* does not exist.

```
(canonicalize-path "test.xml")
⇒ "/tmp/test.xml"
```

`file-exists?` *filename* [Scheme Procedure]
Return `#t` if the file named *filename* exists, `#f` if not.

Many operating systems, such as GNU, use `/` (forward slash) to separate the components of a file name; any file name starting with `/` is considered an *absolute file name*. These conventions are specified by the POSIX Base Definitions, which refer to conforming file names as "pathnames". Some operating systems use a different convention; in particular, Windows uses `\` (backslash) as the file name separator, and also has the notion of *volume names* like `C:\` for absolute file names. The following procedures and variables provide support for portable file name manipulations.

`system-file-name-convention` [Scheme Procedure]
Return either `posix` or `windows`, depending on what kind of system this Guile is running on.

`file-name-separator?` *c* [Scheme Procedure]
Return true if character *c* is a file name separator on the host platform.

`absolute-file-name?` *file-name* [Scheme Procedure]
Return true if *file-name* denotes an absolute file name on the host platform.

`file-name-separator-string` [Scheme Variable]
The preferred file name separator.

Note that on MinGW builds for Windows, both `/` and `\` are valid separators. Thus, programs should not assume that `file-name-separator-string` is the *only* file name separator—e.g., when extracting the components of a file name.

7.2.4 User Information

The facilities in this section provide an interface to the user and group database. They should be used with care since they are not reentrant.

The following functions accept an object representing user information and return a selected component:

`passwd:name` *pw* [Scheme Procedure]
The name of the userid.

`passwd:passwd` *pw* [Scheme Procedure]
The encrypted passwd.

`passwd:uid` *pw* [Scheme Procedure]
The user id number.

`passwd:gid` *pw* [Scheme Procedure]
The group id number.

`passwd:gecos` *pw* [Scheme Procedure]
The full name.

`passwd:dir` *pw* [Scheme Procedure]
The home directory.

`passwd:shell` *pw* [Scheme Procedure]
The login shell.

`getpwuid` *uid* [Scheme Procedure]
Look up an integer userid in the user database.

`getpwnam` *name* [Scheme Procedure]
Look up a user name string in the user database.

`setpwent` [Scheme Procedure]
Initializes a stream used by `getpwent` to read from the user database. The next use of `getpwent` will return the first entry. The return value is unspecified.

`getpwent` [Scheme Procedure]
Read the next entry in the user database stream. The return is a passwd user object as above, or `#f` when no more entries.

`endpwent` [Scheme Procedure]
Closes the stream used by `getpwent`. The return value is unspecified.

`setpw` [*arg*] [Scheme Procedure]
`scm_setpwent` (*arg*) [C Function]
If called with a true argument, initialize or reset the password data stream. Otherwise, close the stream. The `setpwent` and `endpwent` procedures are implemented on top of this.

`getpw` [*user*] [Scheme Procedure]
`scm_getpwuid` (*user*) [C Function]
Look up an entry in the user database. *user* can be an integer, a string, or omitted, giving the behaviour of getpwuid, getpwnam or getpwent respectively.

The following functions accept an object representing group information and return a selected component:

`group:name` *gr* [Scheme Procedure]
The group name.

`group:passwd` *gr* [Scheme Procedure]
The encrypted group password.

`group:gid` *gr* [Scheme Procedure]
The group id number.

`group:mem` *gr* [Scheme Procedure]
A list of userids which have this group as a supplementary group.

`getgrgid` *gid* [Scheme Procedure]
Look up an integer group id in the group database.

`getgrnam` *name* [Scheme Procedure]
Look up a group name in the group database.

`setgrent` [Scheme Procedure]
Initializes a stream used by `getgrent` to read from the group database. The next use of `getgrent` will return the first entry. The return value is unspecified.

`getgrent` [Scheme Procedure]
Return the next entry in the group database, using the stream set by `setgrent`.

`endgrent` [Scheme Procedure]
Closes the stream used by `getgrent`. The return value is unspecified.

`setgr` [*arg*] [Scheme Procedure]
`scm_setgrent` (*arg*) [C Function]
If called with a true argument, initialize or reset the group data stream. Otherwise, close the stream. The `setgrent` and `endgrent` procedures are implemented on top of this.

`getgr` [*group*] [Scheme Procedure]
`scm_getgrgid` (*group*) [C Function]
Look up an entry in the group database. *group* can be an integer, a string, or omitted, giving the behaviour of getgrgid, getgrnam or getgrent respectively.

In addition to the accessor procedures for the user database, the following shortcut procedure is also available.

`getlogin` [Scheme Procedure]
`scm_getlogin` () [C Function]
Return a string containing the name of the user logged in on the controlling terminal of the process, or `#f` if this information cannot be obtained.

7.2.5 Time

`current-time` [Scheme Procedure]
`scm_current_time` () [C Function]
Return the number of seconds since 1970-01-01 00:00:00 UTC, excluding leap seconds.

`gettimeofday` [Scheme Procedure]
`scm_gettimeofday` () [C Function]
Return a pair containing the number of seconds and microseconds since 1970-01-01 00:00:00 UTC, excluding leap seconds. Note: whether true microsecond resolution is available depends on the operating system.

The following procedures either accept an object representing a broken down time and return a selected component, or accept an object representing a broken down time and a value and set the component to the value. The numbers in parentheses give the usual range.

`tm:sec` *tm* [Scheme Procedure]
`set-tm:sec` *tm val* [Scheme Procedure]
Seconds (0-59).

`tm:min` *tm* [Scheme Procedure]
`set-tm:min` *tm val* [Scheme Procedure]
Minutes (0-59).

`tm:hour` *tm* [Scheme Procedure]
`set-tm:hour` *tm val* [Scheme Procedure]
Hours (0-23).

`tm:mday` *tm* [Scheme Procedure]
`set-tm:mday` *tm val* [Scheme Procedure]
Day of the month (1-31).

`tm:mon` *tm* [Scheme Procedure]
`set-tm:mon` *tm val* [Scheme Procedure]
Month (0-11).

`tm:year` *tm* [Scheme Procedure]
`set-tm:year` *tm val* [Scheme Procedure]
Year (70-), the year minus 1900.

`tm:wday` *tm* [Scheme Procedure]
`set-tm:wday` *tm val* [Scheme Procedure]
Day of the week (0-6) with Sunday represented as 0.

`tm:yday` *tm* [Scheme Procedure]
`set-tm:yday` *tm val* [Scheme Procedure]
Day of the year (0-364, 365 in leap years).

`tm:isdst` *tm* [Scheme Procedure]
`set-tm:isdst` *tm val* [Scheme Procedure]
Daylight saving indicator (0 for "no", greater than 0 for "yes", less than 0 for "unknown").

`tm:gmtoff` *tm* [Scheme Procedure]
`set-tm:gmtoff` *tm val* [Scheme Procedure]
Time zone offset in seconds west of UTC (-46800 to 43200). For example on East coast USA (zone '`EST+5`') this would be 18000 (ie. $5 \times 60 \times 60$) in winter, or 14400 (ie. $4 \times 60 \times 60$) during daylight savings.

Note `tm:gmtoff` is not the same as `tm_gmtoff` in the C `tm` structure. `tm_gmtoff` is seconds east and hence the negative of the value here.

`tm:zone` *tm* [Scheme Procedure]
`set-tm:zone` *tm val* [Scheme Procedure]
Time zone label (a string), not necessarily unique.

`localtime` *time* [*zone*] [Scheme Procedure]
`scm_localtime` (*time*, *zone*) [C Function]
Return an object representing the broken down components of *time*, an integer like the one returned by `current-time`. The time zone for the calculation is optionally specified by *zone* (a string), otherwise the `TZ` environment variable or the system default is used.

`gmtime` *time* [Scheme Procedure]
`scm_gmtime` (*time*) [C Function]
Return an object representing the broken down components of *time*, an integer like the one returned by `current-time`. The values are calculated for UTC.

`mktime` *sbd-time* [*zone*] [Scheme Procedure]
`scm_mktime` (*sbd_time*, *zone*) [C Function]
For a broken down time object *sbd-time*, return a pair the `car` of which is an integer time like `current-time`, and the `cdr` of which is a new broken down time with normalized fields.

zone is a timezone string, or the default is the `TZ` environment variable or the system default (see Section "Specifying the Time Zone with `TZ`" in *GNU C Library Reference Manual*). *sbd-time* is taken to be in that *zone*.

The following fields of *sbd-time* are used: `tm:year`, `tm:mon`, `tm:mday`, `tm:hour`, `tm:min`, `tm:sec`, `tm:isdst`. The values can be outside their usual ranges. For example `tm:hour` normally goes up to 23, but a value say 33 would mean 9 the following day.

`tm:isdst` in *sbd-time* says whether the time given is with daylight savings or not. This is ignored if *zone* doesn't have any daylight savings adjustment amount.

The broken down time in the return normalizes the values of *sbd-time* by bringing them into their usual ranges, and using the actual daylight savings rule for that time in *zone* (which may differ from what *sbd-time* had). The easiest way to think of this is that *sbd-time* plus *zone* converts to the integer UTC time, then a `localtime` is applied to get the normal presentation of that time, in *zone*.

`tzset` [Scheme Procedure]
`scm_tzset` () [C Function]
Initialize the timezone from the `TZ` environment variable or the system default. It's not usually necessary to call this procedure since it's done automatically by other procedures that depend on the timezone.

`strftime` *format tm* [Scheme Procedure]
`scm_strftime` (*format*, *tm*) [C Function]
Return a string which is broken-down time structure *tm* formatted according to the given *format* string.

format contains field specifications introduced by a '`%`' character. See Section "Formatting Calendar Time" in *The GNU C Library Reference Manual*, or '`man 3 strftime`', for the available formatting.

```
(strftime "%c" (localtime (current-time)))
⇒ "Mon Mar 11 20:17:43 2002"
```

If `setlocale` has been called (see Section 7.2.13 [Locales], page 547), month and day names are from the current locale and in the locale character set.

`strptime` *format string* [Scheme Procedure]
`scm_strptime` (*format*, *string*) [C Function]
Performs the reverse action to `strftime`, parsing *string* according to the specification supplied in *format*. The interpretation of month and day names is dependent on the current locale. The value returned is a pair. The CAR has an object with time components in the form returned by `localtime` or `gmtime`, but the time zone components are not usefully set. The CDR reports the number of characters from *string* which were used for the conversion.

`internal-time-units-per-second` [Variable]
The value of this variable is the number of time units per second reported by the following procedures.

`times` [Scheme Procedure]
`scm_times` () [C Function]
Return an object with information about real and processor time. The following procedures accept such an object as an argument and return a selected component:

`tms:clock` *tms* [Scheme Procedure]
The current real time, expressed as time units relative to an arbitrary base.

`tms:utime` *tms* [Scheme Procedure]
The CPU time units used by the calling process.

`tms:stime` *tms* [Scheme Procedure]
The CPU time units used by the system on behalf of the calling process.

`tms:cutime` *tms* [Scheme Procedure]
The CPU time units used by terminated child processes of the calling process, whose status has been collected (e.g., using `waitpid`).

`tms:cstime` *tms* [Scheme Procedure]
Similarly, the CPU times units used by the system on behalf of terminated child processes.

`get-internal-real-time` [Scheme Procedure]
`scm_get_internal_real_time` () [C Function]
Return the number of time units since the interpreter was started.

`get-internal-run-time` [Scheme Procedure]
`scm_get_internal_run_time` () [C Function]
Return the number of time units of processor time used by the interpreter. Both *system* and *user* time are included but subprocesses are not.

7.2.6 Runtime Environment

`program-arguments` [Scheme Procedure]
`command-line` [Scheme Procedure]
`set-program-arguments` [Scheme Procedure]
`scm_program_arguments` () [C Function]
`scm_set_program_arguments_scm` (*lst*) [C Function]

Get the command line arguments passed to Guile, or set new arguments.

The arguments are a list of strings, the first of which is the invoked program name. This is just `"guile"` (or the executable path) when run interactively, or it's the script name when running a script with `-s` (see Section 4.2 [Invoking Guile], page 35).

```
guile -L /my/extra/dir -s foo.scm abc def

(program-arguments) ⇒ ("foo.scm" "abc" "def")
```

`set-program-arguments` allows a library module or similar to modify the arguments, for example to strip options it recognises, leaving the rest for the mainline.

The argument list is held in a fluid, which means it's separate for each thread. Neither the list nor the strings within it are copied at any point and normally should not be mutated.

The two names `program-arguments` and `command-line` are an historical accident, they both do exactly the same thing. The name `scm_set_program_arguments_scm` has an extra `_scm` on the end to avoid clashing with the C function below.

`void scm_set_program_arguments` (*int argc, char **argv, char *first*) [C Function]

Set the list of command line arguments for `program-arguments` and `command-line` above.

argv is an array of null-terminated strings, as in a C `main` function. *argc* is the number of strings in *argv*, or if it's negative then a `NULL` in *argv* marks its end.

first is an extra string put at the start of the arguments, or `NULL` for no such extra. This is a convenient way to pass the program name after advancing *argv* to strip option arguments. Eg.

```
{
  char *progname = argv[0];
  for (argv++; argv[0] != NULL && argv[0][0] == '-'; argv++)
    {
      /* munch option ... */
    }
  /* remaining args for scheme level use */
  scm_set_program_arguments (-1, argv, progname);
}
```

This sort of thing is often done at startup under `scm_boot_guile` with options handled at the C level removed. The given strings are all copied, so the C data is not accessed again once `scm_set_program_arguments` returns.

`getenv` *name* [Scheme Procedure]
`scm_getenv` (*name*) [C Function]
Looks up the string *name* in the current environment. The return value is `#f` unless a string of the form `NAME=VALUE` is found, in which case the string `VALUE` is returned.

`setenv` *name value* [Scheme Procedure]
Modifies the environment of the current process, which is also the default environment inherited by child processes.

If *value* is `#f`, then *name* is removed from the environment. Otherwise, the string *name*=*value* is added to the environment, replacing any existing string with name matching *name*.

The return value is unspecified.

`unsetenv` *name* [Scheme Procedure]
Remove variable *name* from the environment. The name can not contain a '`=`' character.

`environ` [*env*] [Scheme Procedure]
`scm_environ` (*env*) [C Function]
If *env* is omitted, return the current environment (in the Unix sense) as a list of strings. Otherwise set the current environment, which is also the default environment for child processes, to the supplied list of strings. Each member of *env* should be of the form *name*=*value* and values of *name* should not be duplicated. If *env* is supplied then the return value is unspecified.

`putenv` *str* [Scheme Procedure]
`scm_putenv` (*str*) [C Function]
Modifies the environment of the current process, which is also the default environment inherited by child processes.

If *str* is of the form `NAME=VALUE` then it will be written directly into the environment, replacing any existing environment string with name matching `NAME`. If *str* does not contain an equal sign, then any existing string with name matching *str* will be removed.

The return value is unspecified.

7.2.7 Processes

`chdir` *str* [Scheme Procedure]
`scm_chdir` (*str*) [C Function]
Change the current working directory to *str*. The return value is unspecified.

`getcwd` [Scheme Procedure]
`scm_getcwd` () [C Function]
Return the name of the current working directory.

`umask` [*mode*] [Scheme Procedure]
`scm_umask` (*mode*) [C Function]
If *mode* is omitted, returns a decimal number representing the current file creation mask. Otherwise the file creation mask is set to *mode* and the previous value is

returned. See Section "Assigning File Permissions" in *The GNU C Library Reference Manual*, for more on how to use umasks.

E.g., `(umask #o022)` sets the mask to octal 22/decimal 18.

`chroot` *path* [Scheme Procedure]
`scm_chroot` (*path*) [C Function]
Change the root directory to that specified in *path*. This directory will be used for path names beginning with `/`. The root directory is inherited by all children of the current process. Only the superuser may change the root directory.

`getpid` [Scheme Procedure]
`scm_getpid` () [C Function]
Return an integer representing the current process ID.

`getgroups` [Scheme Procedure]
`scm_getgroups` () [C Function]
Return a vector of integers representing the current supplementary group IDs.

`getppid` [Scheme Procedure]
`scm_getppid` () [C Function]
Return an integer representing the process ID of the parent process.

`getuid` [Scheme Procedure]
`scm_getuid` () [C Function]
Return an integer representing the current real user ID.

`getgid` [Scheme Procedure]
`scm_getgid` () [C Function]
Return an integer representing the current real group ID.

`geteuid` [Scheme Procedure]
`scm_geteuid` () [C Function]
Return an integer representing the current effective user ID. If the system does not support effective IDs, then the real ID is returned. `(provided? 'EIDs)` reports whether the system supports effective IDs.

`getegid` [Scheme Procedure]
`scm_getegid` () [C Function]
Return an integer representing the current effective group ID. If the system does not support effective IDs, then the real ID is returned. `(provided? 'EIDs)` reports whether the system supports effective IDs.

`setgroups` *vec* [Scheme Procedure]
`scm_setgroups` (*vec*) [C Function]
Set the current set of supplementary group IDs to the integers in the given vector *vec*. The return value is unspecified.

Generally only the superuser can set the process group IDs (see Section "Setting Groups" in *The GNU C Library Reference Manual*).

`setuid` *id* [Scheme Procedure]
`scm_setuid` (*id*) [C Function]
Sets both the real and effective user IDs to the integer *id*, provided the process has appropriate privileges. The return value is unspecified.

`setgid` *id* [Scheme Procedure]
`scm_setgid` (*id*) [C Function]
Sets both the real and effective group IDs to the integer *id*, provided the process has appropriate privileges. The return value is unspecified.

`seteuid` *id* [Scheme Procedure]
`scm_seteuid` (*id*) [C Function]
Sets the effective user ID to the integer *id*, provided the process has appropriate privileges. If effective IDs are not supported, the real ID is set instead—`(provided? 'EIDs)` reports whether the system supports effective IDs. The return value is unspecified.

`setegid` *id* [Scheme Procedure]
`scm_setegid` (*id*) [C Function]
Sets the effective group ID to the integer *id*, provided the process has appropriate privileges. If effective IDs are not supported, the real ID is set instead—`(provided? 'EIDs)` reports whether the system supports effective IDs. The return value is unspecified.

`getpgrp` [Scheme Procedure]
`scm_getpgrp` () [C Function]
Return an integer representing the current process group ID. This is the POSIX definition, not BSD.

`setpgid` *pid pgid* [Scheme Procedure]
`scm_setpgid` (*pid*, *pgid*) [C Function]
Move the process *pid* into the process group *pgid*. *pid* or *pgid* must be integers: they can be zero to indicate the ID of the current process. Fails on systems that do not support job control. The return value is unspecified.

`setsid` [Scheme Procedure]
`scm_setsid` () [C Function]
Creates a new session. The current process becomes the session leader and is put in a new process group. The process will be detached from its controlling terminal if it has one. The return value is an integer representing the new process group ID.

`getsid` *pid* [Scheme Procedure]
`scm_getsid` (*pid*) [C Function]
Returns the session ID of process *pid*. (The session ID of a process is the process group ID of its session leader.)

`waitpid` *pid* [*options*] [Scheme Procedure]
`scm_waitpid` (*pid*, *options*) [C Function]
This procedure collects status information from a child process which has terminated or (optionally) stopped. Normally it will suspend the calling process until this can

be done. If more than one child process is eligible then one will be chosen by the operating system.

The value of *pid* determines the behaviour:

pid greater than 0
: Request status information from the specified child process.

pid equal to -1 or `WAIT_ANY`
: Request status information for any child process.

pid equal to 0 or `WAIT_MYPGRP`
: Request status information for any child process in the current process group.

pid less than -1
: Request status information for any child process whose process group ID is $-pid$.

The *options* argument, if supplied, should be the bitwise OR of the values of zero or more of the following variables:

`WNOHANG` [Variable]
: Return immediately even if there are no child processes to be collected.

`WUNTRACED` [Variable]
: Report status information for stopped processes as well as terminated processes.

The return value is a pair containing:

1. The process ID of the child process, or 0 if `WNOHANG` was specified and no process was collected.
2. The integer status value (see Section "Process Completion Status" in *The GNU C Library Reference Manual*).

The following three functions can be used to decode the integer status value returned by `waitpid`.

`status:exit-val` *status* [Scheme Procedure]
`scm_status_exit_val` (*status*) [C Function]
: Return the exit status value, as would be set if a process ended normally through a call to `exit` or `_exit`, if any, otherwise `#f`.

`status:term-sig` *status* [Scheme Procedure]
`scm_status_term_sig` (*status*) [C Function]
: Return the signal number which terminated the process, if any, otherwise `#f`.

`status:stop-sig` *status* [Scheme Procedure]
`scm_status_stop_sig` (*status*) [C Function]
: Return the signal number which stopped the process, if any, otherwise `#f`.

`system` [*cmd*] [Scheme Procedure]
`scm_system` (*cmd*) [C Function]

Execute *cmd* using the operating system's "command processor". Under Unix this is usually the default shell `sh`. The value returned is *cmd*'s exit status as returned by `waitpid`, which can be interpreted using the functions above.

If `system` is called without arguments, return a boolean indicating whether the command processor is available.

`system*` *arg1 arg2 ...* [Scheme Procedure]
`scm_system_star` (*args*) [C Function]

Execute the command indicated by *arg1 arg2 ...*. The first element must be a string indicating the command to be executed, and the remaining items must be strings representing each of the arguments to that command.

This function returns the exit status of the command as provided by `waitpid`. This value can be handled with `status:exit-val` and the related functions.

`system*` is similar to `system`, but accepts only one string per-argument, and performs no shell interpretation. The command is executed using fork and execlp. Accordingly this function may be safer than `system` in situations where shell interpretation is not required.

Example: (system* "echo" "foo" "bar")

`quit` [*status*] [Scheme Procedure]
`exit` [*status*] [Scheme Procedure]

Terminate the current process with proper unwinding of the Scheme stack. The exit status zero if *status* is not supplied. If *status* is supplied, and it is an integer, that integer is used as the exit status. If *status* is `#t` or `#f`, the exit status is *EXIT_SUCCESS* or *EXIT_FAILURE*, respectively.

The procedure `exit` is an alias of `quit`. They have the same functionality.

`EXIT_SUCCESS` [Scheme Variable]
`EXIT_FAILURE` [Scheme Variable]

These constants represent the standard exit codes for success (zero) or failure (one.)

`primitive-exit` [*status*] [Scheme Procedure]
`primitive-_exit` [*status*] [Scheme Procedure]
`scm_primitive_exit` (*status*) [C Function]
`scm_primitive__exit` (*status*) [C Function]

Terminate the current process without unwinding the Scheme stack. The exit status is *status* if supplied, otherwise zero.

`primitive-exit` uses the C `exit` function and hence runs usual C level cleanups (flush output streams, call `atexit` functions, etc, see Section "Normal Termination" in *The GNU C Library Reference Manual*)).

`primitive-_exit` is the `_exit` system call (see Section "Termination Internals" in *The GNU C Library Reference Manual*). This terminates the program immediately, with neither Scheme-level nor C-level cleanups.

The typical use for `primitive-_exit` is from a child process created with `primitive-fork`. For example in a Gdk program the child process inherits the X

server connection and a C-level `atexit` cleanup which will close that connection. But closing in the child would upset the protocol in the parent, so `primitive-_exit` should be used to exit without that.

`execl` *filename arg . . .* [Scheme Procedure]
`scm_execl` (*filename, args*) [C Function]

Executes the file named by *filename* as a new process image. The remaining arguments are supplied to the process; from a C program they are accessible as the `argv` argument to `main`. Conventionally the first *arg* is the same as *filename*. All arguments must be strings.

If *arg* is missing, *filename* is executed with a null argument list, which may have system-dependent side-effects.

This procedure is currently implemented using the `execv` system call, but we call it `execl` because of its Scheme calling interface.

`execlp` *filename arg . . .* [Scheme Procedure]
`scm_execlp` (*filename, args*) [C Function]

Similar to `execl`, however if *filename* does not contain a slash then the file to execute will be located by searching the directories listed in the `PATH` environment variable.

This procedure is currently implemented using the `execvp` system call, but we call it `execlp` because of its Scheme calling interface.

`execle` *filename env arg . . .* [Scheme Procedure]
`scm_execle` (*filename, env, args*) [C Function]

Similar to `execl`, but the environment of the new process is specified by *env*, which must be a list of strings as returned by the `environ` procedure.

This procedure is currently implemented using the `execve` system call, but we call it `execle` because of its Scheme calling interface.

`primitive-fork` [Scheme Procedure]
`scm_fork` () [C Function]

Creates a new "child" process by duplicating the current "parent" process. In the child the return value is 0. In the parent the return value is the integer process ID of the child.

Note that it is unsafe to fork a process that has multiple threads running, as only the thread that calls `primitive-fork` will persist in the child. Any resources that other threads held, such as locked mutexes or open file descriptors, are lost. Indeed, POSIX specifies that only async-signal-safe procedures are safe to call after a multithreaded fork, which is a very limited set. Guile issues a warning if it detects a fork from a multi-threaded program.

If you are going to `exec` soon after forking, the procedures in `(ice-9 popen)` may be useful to you, as they fork and exec within an async-signal-safe function carefully written to ensure robust program behavior, even in the presence of threads. See Section 7.2.10 [Pipes], page 529, for more.

This procedure has been renamed from `fork` to avoid a naming conflict with the scsh fork.

`nice` *incr* [Scheme Procedure]
`scm_nice` (*incr*) [C Function]
Increment the priority of the current process by *incr*. A higher priority value means that the process runs less often. The return value is unspecified.

`setpriority` *which who prio* [Scheme Procedure]
`scm_setpriority` (*which, who, prio*) [C Function]
Set the scheduling priority of the process, process group or user, as indicated by *which* and *who*. *which* is one of the variables `PRIO_PROCESS`, `PRIO_PGRP` or `PRIO_USER`, and *who* is interpreted relative to *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user identifier for `PRIO_USER`. A zero value of *who* denotes the current process, process group, or user. *prio* is a value in the range [−20,20]. The default priority is 0; lower priorities (in numerical terms) cause more favorable scheduling. Sets the priority of all of the specified processes. Only the super-user may lower priorities. The return value is not specified.

`getpriority` *which who* [Scheme Procedure]
`scm_getpriority` (*which, who*) [C Function]
Return the scheduling priority of the process, process group or user, as indicated by *which* and *who*. *which* is one of the variables `PRIO_PROCESS`, `PRIO_PGRP` or `PRIO_USER`, and *who* should be interpreted depending on *which* (a process identifier for `PRIO_PROCESS`, process group identifier for `PRIO_PGRP`, and a user identifier for `PRIO_USER`). A zero value of *who* denotes the current process, process group, or user. Return the highest priority (lowest numerical value) of any of the specified processes.

`getaffinity` *pid* [Scheme Procedure]
`scm_getaffinity` (*pid*) [C Function]
Return a bitvector representing the CPU affinity mask for process *pid*. Each CPU the process has affinity with has its corresponding bit set in the returned bitvector. The number of bits set is a good estimate of how many CPUs Guile can use without stepping on other processes' toes.

Currently this procedure is only defined on GNU variants (see Section "CPU Affinity" in *The GNU C Library Reference Manual*).

`setaffinity` *pid mask* [Scheme Procedure]
`scm_setaffinity` (*pid, mask*) [C Function]
Install the CPU affinity mask *mask*, a bitvector, for the process or thread with ID *pid*. The return value is unspecified.

Currently this procedure is only defined on GNU variants (see Section "CPU Affinity" in *The GNU C Library Reference Manual*).

See Section 6.22.1 [Threads], page 442, for information on how get the number of processors available on a system.

7.2.8 Signals

The following procedures raise, handle and wait for signals.

Scheme code signal handlers are run via an async (see Section 6.22.3 [Asyncs], page 445), so they're called in the handler's thread at the next safe opportunity. Generally this is

after any currently executing primitive procedure finishes (which could be a long time for primitives that wait for an external event).

`kill` *pid sig* [Scheme Procedure]
`scm_kill` (*pid, sig*) [C Function]

Sends a signal to the specified process or group of processes.

pid specifies the processes to which the signal is sent:

pid greater than 0
: The process whose identifier is *pid*.

pid equal to 0
: All processes in the current process group.

pid less than -1
: The process group whose identifier is *-pid*

pid equal to -1
: If the process is privileged, all processes except for some special system processes. Otherwise, all processes with the current effective user ID.

sig should be specified using a variable corresponding to the Unix symbolic name, e.g.,

`SIGHUP` [Variable]
: Hang-up signal.

`SIGINT` [Variable]
: Interrupt signal.

A full list of signals on the GNU system may be found in Section "Standard Signals" in *The GNU C Library Reference Manual*.

`raise` *sig* [Scheme Procedure]
`scm_raise` (*sig*) [C Function]

Sends a specified signal *sig* to the current process, where *sig* is as described for the `kill` procedure.

`sigaction` *signum* [*handler* [*flags* [*thread*]]] [Scheme Procedure]
`scm_sigaction` (*signum, handler, flags*) [C Function]
`scm_sigaction_for_thread` (*signum, handler, flags, thread*) [C Function]

Install or report the signal handler for a specified signal.

signum is the signal number, which can be specified using the value of variables such as `SIGINT`.

If *handler* is omitted, `sigaction` returns a pair: the CAR is the current signal hander, which will be either an integer with the value `SIG_DFL` (default action) or `SIG_IGN` (ignore), or the Scheme procedure which handles the signal, or `#f` if a non-Scheme procedure handles the signal. The CDR contains the current `sigaction` flags for the handler.

If *handler* is provided, it is installed as the new handler for *signum*. *handler* can be a Scheme procedure taking one argument, or the value of `SIG_DFL` (default action)

or `SIG_IGN` (ignore), or `#f` to restore whatever signal handler was installed before `sigaction` was first used. When a scheme procedure has been specified, that procedure will run in the given *thread*. When no thread has been given, the thread that made this call to `sigaction` is used.

flags is a `logior` (see Section 6.6.2.13 [Bitwise Operations], page 125) of the following (where provided by the system), or `0` for none.

`SA_NOCLDSTOP` [Variable]

By default, `SIGCHLD` is signalled when a child process stops (ie. receives `SIGSTOP`), and when a child process terminates. With the `SA_NOCLDSTOP` flag, `SIGCHLD` is only signalled for termination, not stopping.

`SA_NOCLDSTOP` has no effect on signals other than `SIGCHLD`.

`SA_RESTART` [Variable]

If a signal occurs while in a system call, deliver the signal then restart the system call (as opposed to returning an `EINTR` error from that call).

Guile handles signals asynchronously. When it receives a signal, the synchronous signal handler just records the fact that a signal was received and sets a flag to tell the relevant Guile thread that it has a pending signal. When the Guile thread checks the pending-interrupt flag, it will arrange to run the asynchronous part of the signal handler, which is the handler attached by `sigaction`.

This strategy has some perhaps-unexpected interactions with the `SA_RESTART` flag, though: because the synchronous handler doesn't do very much, and notably it doesn't run the Guile handler, it's impossible to interrupt a thread stuck in a long-running system call via a signal handler that is installed with `SA_RESTART`: the synchronous handler just records the pending interrupt, but then the system call resumes and Guile doesn't have a chance to actually check the flag and run the asynchronous handler. That's just how it is.

The return value is a pair with information about the old handler as described above.

This interface does not provide access to the "signal blocking" facility. Maybe this is not needed, since the thread support may provide solutions to the problem of consistent access to data structures.

`restore-signals` [Scheme Procedure]
`scm_restore_signals ()` [C Function]

Return all signal handlers to the values they had before any call to `sigaction` was made. The return value is unspecified.

`alarm` *i* [Scheme Procedure]
`scm_alarm (`*i*`)` [C Function]

Set a timer to raise a `SIGALRM` signal after the specified number of seconds (an integer). It's advisable to install a signal handler for `SIGALRM` beforehand, since the default action is to terminate the process.

The return value indicates the time remaining for the previous alarm, if any. The new value replaces the previous alarm. If there was no previous alarm, the return value is zero.

`pause` [Scheme Procedure]
`scm_pause ()` [C Function]
Pause the current process (thread?) until a signal arrives whose action is to either terminate the current process or invoke a handler procedure. The return value is unspecified.

`sleep` *secs* [Scheme Procedure]
`usleep` *usecs* [Scheme Procedure]
`scm_sleep` (*secs*) [C Function]
`scm_usleep` (*usecs*) [C Function]
Wait the given period *secs* seconds or *usecs* microseconds (both integers). If a signal arrives the wait stops and the return value is the time remaining, in seconds or microseconds respectively. If the period elapses with no signal the return is zero.

On most systems the process scheduler is not microsecond accurate and the actual period slept by `usleep` might be rounded to a system clock tick boundary, which might be 10 milliseconds for instance.

See `scm_std_sleep` and `scm_std_usleep` for equivalents at the C level (see Section 6.22.6 [Blocking], page 452).

`getitimer` *which_timer* [Scheme Procedure]
`setitimer` *which_timer interval_seconds* [Scheme Procedure]
interval_microseconds value_seconds value_microseconds
`scm_getitimer` (*which_timer*) [C Function]
`scm_setitimer` (*which_timer*, *interval_seconds*, [C Function]
interval_microseconds, *value_seconds*, *value_microseconds*)
Get or set the periods programmed in certain system timers.

These timers have two settings. The first setting, the interval, is the value at which the timer will be reset when the current timer expires. The second is the current value of the timer, indicating when the next expiry will be signalled.

which_timer is one of the following values:

`ITIMER_REAL` [Variable]
A real-time timer, counting down elapsed real time. At zero it raises `SIGALRM`. This is like `alarm` above, but with a higher resolution period.

`ITIMER_VIRTUAL` [Variable]
A virtual-time timer, counting down while the current process is actually using CPU. At zero it raises `SIGVTALRM`.

`ITIMER_PROF` [Variable]
A profiling timer, counting down while the process is running (like `ITIMER_VIRTUAL`) and also while system calls are running on the process's behalf. At zero it raises a `SIGPROF`.

This timer is intended for profiling where a program is spending its time (by looking where it is when the timer goes off).

`getitimer` returns the restart timer value and its current value, as a list containing two pairs. Each pair is a time in seconds and microseconds: `((`*`interval_secs`* `.` *`interval_usecs`*`) (`*`value_secs`* `.` *`value_usecs`*`))`.

`setitimer` sets the timer values similarly, in seconds and microseconds (which must be integers). The interval value can be zero to have the timer run down just once. The return value is the timer's previous setting, in the same form as `getitimer` returns.

```
(setitimer ITIMER_REAL
           5 500000     ;; Raise SIGALRM every 5.5 seconds
           2 0)         ;; with the first SIGALRM in 2 seconds
```

Although the timers are programmed in microseconds, the actual accuracy might not be that high.

Note that `ITIMER_PROF` and `ITIMER_VIRTUAL` are not functional on all platforms and may always error when called. `(provided? 'ITIMER_PROF)` and `(provided? 'ITIMER_VIRTUAL)` can be used to test if the those itimers are supported on the given host. `ITIMER_REAL` is supported on all platforms that support `setitimer`.

7.2.9 Terminals and Ptys

`isatty?` *port* [Scheme Procedure]
`scm_isatty_p` (*port*) [C Function]
Return `#t` if *port* is using a serial non–file device, otherwise `#f`.

`ttyname` *port* [Scheme Procedure]
`scm_ttyname` (*port*) [C Function]
Return a string with the name of the serial terminal device underlying *port*.

`ctermid` [Scheme Procedure]
`scm_ctermid` () [C Function]
Return a string containing the file name of the controlling terminal for the current process.

`tcgetpgrp` *port* [Scheme Procedure]
`scm_tcgetpgrp` (*port*) [C Function]
Return the process group ID of the foreground process group associated with the terminal open on the file descriptor underlying *port*.

If there is no foreground process group, the return value is a number greater than 1 that does not match the process group ID of any existing process group. This can happen if all of the processes in the job that was formerly the foreground job have terminated, and no other job has yet been moved into the foreground.

`tcsetpgrp` *port pgid* [Scheme Procedure]
`scm_tcsetpgrp` (*port, pgid*) [C Function]
Set the foreground process group ID for the terminal used by the file descriptor underlying *port* to the integer *pgid*. The calling process must be a member of the same session as *pgid* and must have the same controlling terminal. The return value is unspecified.

7.2.10 Pipes

The following procedures are similar to the `popen` and `pclose` system routines. The code is in a separate "popen" module[1]:

```
(use-modules (ice-9 popen))
```

`open-pipe` *command mode* [Scheme Procedure]
`open-pipe*` *mode prog* [*args...*] [Scheme Procedure]

Execute a command in a subprocess, with a pipe to it or from it, or with pipes in both directions.

`open-pipe` runs the shell *command* using '`/bin/sh -c`'. `open-pipe*` executes *prog* directly, with the optional *args* arguments (all strings).

mode should be one of the following values. `OPEN_READ` is an input pipe, ie. to read from the subprocess. `OPEN_WRITE` is an output pipe, ie. to write to it.

`OPEN_READ` [Variable]
`OPEN_WRITE` [Variable]
`OPEN_BOTH` [Variable]

For an input pipe, the child's standard output is the pipe and standard input is inherited from `current-input-port`. For an output pipe, the child's standard input is the pipe and standard output is inherited from `current-output-port`. In all cases the child's standard error is inherited from `current-error-port` (see Section 6.14.9 [Default Ports], page 343).

If those `current-X-ports` are not files of some kind, and hence don't have file descriptors for the child, then `/dev/null` is used instead.

Care should be taken with `OPEN_BOTH`, a deadlock will occur if both parent and child are writing, and waiting until the write completes before doing any reading. Each direction has `PIPE_BUF` bytes of buffering (see Section 6.14.6 [Buffering], page 339), which will be enough for small writes, but not for say putting a big file through a filter.

`open-input-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_READ`.

```
(let* ((port (open-input-pipe "date --utc"))
       (str  (read-line port))) ; from (ice-9 rdelim)
  (close-pipe port)
  str)
⇒ "Mon Mar 11 20:10:44 UTC 2002"
```

`open-output-pipe` *command* [Scheme Procedure]

Equivalent to `open-pipe` with mode `OPEN_WRITE`.

```
(let ((port (open-output-pipe "lpr")))
  (display "Something for the line printer.\n" port)
  (if (not (eqv? 0 (status:exit-val (close-pipe port))))
      (error "Cannot print")))
```

[1] This module is only available on systems where the `popen` feature is provided (see Section 6.23.2.2 [Common Feature Symbols], page 458).

`open-input-output-pipe` *command* [Scheme Procedure]
Equivalent to `open-pipe` with mode `OPEN_BOTH`.

`close-pipe` *port* [Scheme Procedure]
Close a pipe created by `open-pipe`, wait for the process to terminate, and return the wait status code. The status is as per `waitpid` and can be decoded with `status:exit-val` etc (see Section 7.2.7 [Processes], page 518)

`waitpid WAIT_ANY` should not be used when pipes are open, since it can reap a pipe's child process, causing an error from a subsequent `close-pipe`.

`close-port` (see Section 6.14.1 [Ports], page 331) can close a pipe, but it doesn't reap the child process.

The garbage collector will close a pipe no longer in use, and reap the child process with `waitpid`. If the child hasn't yet terminated the garbage collector doesn't block, but instead checks again in the next GC.

Many systems have per-user and system-wide limits on the number of processes, and a system-wide limit on the number of pipes, so pipes should be closed explicitly when no longer needed, rather than letting the garbage collector pick them up at some later time.

`pipeline` *commands* [Scheme Procedure]
Execute a pipeline of *commands*, where each command is a list of a program and its arguments as strings, returning an input port to the end of the pipeline, an output port to the beginning of the pipeline and a list of PIDs of the processes executing the *commands*.

```
(let ((commands '(("git" "ls-files")
                  ("tar" "-cf-" "-T-")
                  ("sha1sum" "-")))
      (success? (lambda (pid)
                  (zero?
                   (status:exit-val (cdr (waitpid pid)))))))
  (receive (from to pids) (pipeline commands)
    (let* ((sha1 (read-delimited " " from))
           (index (list-index (negate success?) (reverse pids))))
      (close to)
      (close from)
      (if (not index)
          sha1
          (string-append "pipeline failed in command: "
                         (string-join (list-ref commands index)))))))
⇒ "52f99d234503fca8c84ef94b1005a3a28d8b3bc1"
```

7.2.11 Networking

7.2.11.1 Network Address Conversion

This section describes procedures which convert internet addresses between numeric and string formats.

IPv4 Address Conversion

An IPv4 Internet address is a 4-byte value, represented in Guile as an integer in host byte order, so that say "0.0.0.1" is 1, or "1.0.0.0" is 16777216.

Some underlying C functions use network byte order for addresses, Guile converts as necessary so that at the Scheme level its host byte order everywhere.

`INADDR_ANY` [Variable]
: For a server, this can be used with `bind` (see Section 7.2.11.4 [Network Sockets and Communication], page 540) to allow connections from any interface on the machine.

`INADDR_BROADCAST` [Variable]
: The broadcast address on the local network.

`INADDR_LOOPBACK` [Variable]
: The address of the local host using the loopback device, ie. '`127.0.0.1`'.

`inet-netof` *address* [Scheme Procedure]
`scm_inet_netof` (*address*) [C Function]
: Return the network number part of the given IPv4 Internet address. E.g.,

```
(inet-netof 2130706433) ⇒ 127
```

`inet-lnaof` *address* [Scheme Procedure]
`scm_lnaof` (*address*) [C Function]
: Return the local-address-with-network part of the given IPv4 Internet address, using the obsolete class A/B/C system. E.g.,

```
(inet-lnaof 2130706433) ⇒ 1
```

`inet-makeaddr` *net lna* [Scheme Procedure]
`scm_inet_makeaddr` (*net*, *lna*) [C Function]
: Make an IPv4 Internet address by combining the network number *net* with the local-address-within-network number *lna*. E.g.,

```
(inet-makeaddr 127 1) ⇒ 2130706433
```

IPv6 Address Conversion

An IPv6 Internet address is a 16-byte value, represented in Guile as an integer in host byte order, so that say "::1" is 1.

`inet-ntop` *family address* [Scheme Procedure]
`scm_inet_ntop` (*family*, *address*) [C Function]
: Convert a network address from an integer to a printable string. *family* can be `AF_INET` or `AF_INET6`. E.g.,

```
(inet-ntop AF_INET 2130706433) ⇒ "127.0.0.1"
(inet-ntop AF_INET6 (- (expt 2 128) 1))
  ⇒ "ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff"
```

`inet-pton` *family address* [Scheme Procedure]
`scm_inet_pton` (*family*, *address*) [C Function]
Convert a string containing a printable network address to an integer address. *family* can be `AF_INET` or `AF_INET6`. E.g.,

```
(inet-pton AF_INET "127.0.0.1") ⇒ 2130706433
(inet-pton AF_INET6 "::1") ⇒ 1
```

7.2.11.2 Network Databases

This section describes procedures which query various network databases. Care should be taken when using the database routines since they are not reentrant.

getaddrinfo

The `getaddrinfo` procedure maps host and service names to socket addresses and associated information in a protocol-independent way.

`getaddrinfo` *name service* [*hint_flags* [*hint_family* [*hint_socktype* [*hint_protocol*]]]] [Scheme Procedure]
`scm_getaddrinfo` (*name*, *service*, *hint_flags*, *hint_family*, *hint_socktype*, *hint_protocol*) [C Function]
Return a list of `addrinfo` structures containing a socket address and associated information for host *name* and/or *service* to be used in creating a socket with which to address the specified service.

```
(let* ((ai (car (getaddrinfo "www.gnu.org" "http")))
       (s  (socket (addrinfo:fam ai) (addrinfo:socktype ai)
                   (addrinfo:protocol ai))))
  (connect s (addrinfo:addr ai))
  s)
```

When *service* is omitted or is `#f`, return network-level addresses for *name*. When *name* is `#f` *service* must be provided and service locations local to the caller are returned.

Additional hints can be provided. When specified, *hint_flags* should be a bitwise-or of zero or more constants among the following:

`AI_PASSIVE`
Socket address is intended for `bind`.

`AI_CANONNAME`
Request for canonical host name, available via `addrinfo:canonname`. This makes sense mainly when DNS lookups are involved.

`AI_NUMERICHOST`
Specifies that *name* is a numeric host address string (e.g., `"127.0.0.1"`), meaning that name resolution will not be used.

`AI_NUMERICSERV`
Likewise, specifies that *service* is a numeric port string (e.g., `"80"`).

`AI_ADDRCONFIG`
Return only addresses configured on the local system It is highly recommended to provide this flag when the returned socket addresses are to

be used to make connections; otherwise, some of the returned addresses could be unreachable or use a protocol that is not supported.

`AI_V4MAPPED`
: When looking up IPv6 addresses, return mapped IPv4 addresses if there is no IPv6 address available at all.

`AI_ALL`
: If this flag is set along with `AI_V4MAPPED` when looking up IPv6 addresses, return all IPv6 addresses as well as all IPv4 addresses, the latter mapped to IPv6 format.

When given, *hint_family* should specify the requested address family, e.g., `AF_INET6`. Similarly, *hint_socktype* should specify the requested socket type (e.g., `SOCK_DGRAM`), and *hint_protocol* should specify the requested protocol (its value is interpreted as in calls to `socket`).

On error, an exception with key `getaddrinfo-error` is thrown, with an error code (an integer) as its argument:

```
(catch 'getaddrinfo-error
  (lambda ()
    (getaddrinfo "www.gnu.org" "gopher"))
  (lambda (key errcode)
    (cond ((= errcode EAI_SERVICE)
   (display "doesn't know about Gopher!\n"))
  ((= errcode EAI_NONAME)
   (display "www.gnu.org not found\\n"))
  (else
   (format #t "something wrong: ~a\n"
   (gai-strerror errcode))))))
```

Error codes are:

`EAI_AGAIN`
: The name or service could not be resolved at this time. Future attempts may succeed.

`EAI_BADFLAGS`
: *hint_flags* contains an invalid value.

`EAI_FAIL`
: A non-recoverable error occurred when attempting to resolve the name.

`EAI_FAMILY`
: *hint_family* was not recognized.

`EAI_NONAME`
: Either *name* does not resolve for the supplied parameters, or neither *name* nor *service* were supplied.

`EAI_NODATA`
: This non-POSIX error code can be returned on some systems (GNU and Darwin, at least), for example when *name* is known but requests that were made turned out no data. Error handling code should be prepared to handle it when it is defined.

`EAI_SERVICE`
: *service* was not recognized for the specified socket type.

`EAI_SOCKTYPE`
: *hint_socktype* was not recognized.

`EAI_SYSTEM`
: A system error occurred. In C, the error code can be found in `errno`; this value is not accessible from Scheme, but in practice it provides little information about the actual error cause.

Users are encouraged to read the "POSIX specification (`http://www.opengroup.org/onlinepubs/9699919799/functions/getaddrinfo.html`) for more details.

The following procedures take an `addrinfo` object as returned by `getaddrinfo`:

`addrinfo:flags` *ai* [Scheme Procedure]
: Return flags for *ai* as a bitwise or of `AI_` values (see above).

`addrinfo:fam` *ai* [Scheme Procedure]
: Return the address family of *ai* (a `AF_` value).

`addrinfo:socktype` *ai* [Scheme Procedure]
: Return the socket type for *ai* (a `SOCK_` value).

`addrinfo:protocol` *ai* [Scheme Procedure]
: Return the protocol of *ai*.

`addrinfo:addr` *ai* [Scheme Procedure]
: Return the socket address associated with *ai* as a `sockaddr` object (see Section 7.2.11.3 [Network Socket Address], page 538).

`addrinfo:canonname` *ai* [Scheme Procedure]
: Return a string for the canonical name associated with *ai* if the `AI_CANONNAME` flag was supplied.

The Host Database

A *host object* is a structure that represents what is known about a network host, and is the usual way of representing a system's network identity inside software.

The following functions accept a host object and return a selected component:

`hostent:name` *host* [Scheme Procedure]
: The "official" hostname for *host*.

`hostent:aliases` *host* [Scheme Procedure]
: A list of aliases for *host*.

`hostent:addrtype` *host* [Scheme Procedure]
: The host address type, one of the `AF` constants, such as `AF_INET` or `AF_INET6`.

`hostent:length` *host* [Scheme Procedure]
: The length of each address for *host*, in bytes.

`hostent:addr-list` *host* [Scheme Procedure]
: The list of network addresses associated with *host*. For `AF_INET` these are integer IPv4 address (see Section 7.2.11.1 [Network Address Conversion], page 530).

The following procedures can be used to search the host database. However, `getaddrinfo` should be preferred over them since it's more generic and thread-safe.

`gethost` [*host*] [Scheme Procedure]
`gethostbyname` *hostname* [Scheme Procedure]
`gethostbyaddr` *address* [Scheme Procedure]
`scm_gethost` (*host*) [C Function]
: Look up a host by name or address, returning a host object. The `gethost` procedure will accept either a string name or an integer address; if given no arguments, it behaves like `gethostent` (see below). If a name or address is supplied but the address can not be found, an error will be thrown to one of the keys: `host-not-found`, `try-again`, `no-recovery` or `no-data`, corresponding to the equivalent `h_error` values. Unusual conditions may result in errors thrown to the `system-error` or `misc_error` keys.

```
(gethost "www.gnu.org")
⇒ #("www.gnu.org" () 2 4 (3353880842))

(gethostbyname "www.emacs.org")
⇒ #("emacs.org" ("www.emacs.org") 2 4 (1073448978))
```

The following procedures may be used to step through the host database from beginning to end.

`sethostent` [*stayopen*] [Scheme Procedure]
: Initialize an internal stream from which host objects may be read. This procedure must be called before any calls to `gethostent`, and may also be called afterward to reset the host entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `gethostbyname` or `gethostbyaddr` calls, possibly giving an efficiency gain.

`gethostent` [Scheme Procedure]
: Return the next host object from the host database, or `#f` if there are no more hosts to be found (or an error has been encountered). This procedure may not be used before `sethostent` has been called.

`endhostent` [Scheme Procedure]
: Close the stream used by `gethostent`. The return value is unspecified.

`sethost` [*stayopen*] [Scheme Procedure]
`scm_sethost` (*stayopen*) [C Function]
: If *stayopen* is omitted, this is equivalent to `endhostent`. Otherwise it is equivalent to `sethostent stayopen`.

The Network Database

The following functions accept an object representing a network and return a selected component:

`netent:name` *net* [Scheme Procedure]
The "official" network name.

`netent:aliases` *net* [Scheme Procedure]
A list of aliases for the network.

`netent:addrtype` *net* [Scheme Procedure]
The type of the network number. Currently, this returns only `AF_INET`.

`netent:net` *net* [Scheme Procedure]
The network number.

The following procedures are used to search the network database:

`getnet` [*net*] [Scheme Procedure]
`getnetbyname` *net-name* [Scheme Procedure]
`getnetbyaddr` *net-number* [Scheme Procedure]
`scm_getnet` (*net*) [C Function]
Look up a network by name or net number in the network database. The *net-name* argument must be a string, and the *net-number* argument must be an integer. `getnet` will accept either type of argument, behaving like `getnetent` (see below) if no arguments are given.

The following procedures may be used to step through the network database from beginning to end.

`setnetent` [*stayopen*] [Scheme Procedure]
Initialize an internal stream from which network objects may be read. This procedure must be called before any calls to `getnetent`, and may also be called afterward to reset the net entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getnetbyname` or `getnetbyaddr` calls, possibly giving an efficiency gain.

`getnetent` [Scheme Procedure]
Return the next entry from the network database.

`endnetent` [Scheme Procedure]
Close the stream used by `getnetent`. The return value is unspecified.

`setnet` [*stayopen*] [Scheme Procedure]
`scm_setnet` (*stayopen*) [C Function]
If *stayopen* is omitted, this is equivalent to `endnetent`. Otherwise it is equivalent to `setnetent stayopen`.

The Protocol Database

The following functions accept an object representing a protocol and return a selected component:

`protoent:name` *protocol* [Scheme Procedure]
The "official" protocol name.

`protoent:aliases` *protocol* [Scheme Procedure]
A list of aliases for the protocol.

`protoent:proto` *protocol* [Scheme Procedure]
The protocol number.

The following procedures are used to search the protocol database:

`getproto` [*protocol*] [Scheme Procedure]
`getprotobyname` *name* [Scheme Procedure]
`getprotobynumber` *number* [Scheme Procedure]
`scm_getproto` (*protocol*) [C Function]
Look up a network protocol by name or by number. `getprotobyname` takes a string argument, and `getprotobynumber` takes an integer argument. `getproto` will accept either type, behaving like `getprotoent` (see below) if no arguments are supplied.

The following procedures may be used to step through the protocol database from beginning to end.

`setprotoent` [*stayopen*] [Scheme Procedure]
Initialize an internal stream from which protocol objects may be read. This procedure must be called before any calls to `getprotoent`, and may also be called afterward to reset the protocol entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getprotobyname` or `getprotobynumber` calls, possibly giving an efficiency gain.

`getprotoent` [Scheme Procedure]
Return the next entry from the protocol database.

`endprotoent` [Scheme Procedure]
Close the stream used by `getprotoent`. The return value is unspecified.

`setproto` [*stayopen*] [Scheme Procedure]
`scm_setproto` (*stayopen*) [C Function]
If *stayopen* is omitted, this is equivalent to `endprotoent`. Otherwise it is equivalent to `setprotoent stayopen`.

The Service Database

The following functions accept an object representing a service and return a selected component:

`servent:name` *serv* [Scheme Procedure]
The "official" name of the network service.

`servent:aliases` *serv* [Scheme Procedure]
A list of aliases for the network service.

`servent:port` *serv* [Scheme Procedure]
The Internet port used by the service.

`servent:proto` *serv* [Scheme Procedure]
The protocol used by the service. A service may be listed many times in the database under different protocol names.

The following procedures are used to search the service database:

`getserv` [*name* [*protocol*]] [Scheme Procedure]
`getservbyname` *name protocol* [Scheme Procedure]
`getservbyport` *port protocol* [Scheme Procedure]
`scm_getserv` (*name, protocol*) [C Function]
Look up a network service by name or by service number, and return a network service object. The *protocol* argument specifies the name of the desired protocol; if the protocol found in the network service database does not match this name, a system error is signalled.

The `getserv` procedure will take either a service name or number as its first argument; if given no arguments, it behaves like `getservent` (see below).

```
(getserv "imap" "tcp")
⇒ #("imap2" ("imap") 143 "tcp")

(getservbyport 88 "udp")
⇒ #("kerberos" ("kerberos5" "krb5") 88 "udp")
```

The following procedures may be used to step through the service database from beginning to end.

`setservent` [*stayopen*] [Scheme Procedure]
Initialize an internal stream from which service objects may be read. This procedure must be called before any calls to `getservent`, and may also be called afterward to reset the service entry stream. If *stayopen* is supplied and is not `#f`, the database is not closed by subsequent `getservbyname` or `getservbyport` calls, possibly giving an efficiency gain.

`getservent` [Scheme Procedure]
Return the next entry from the services database.

`endservent` [Scheme Procedure]
Close the stream used by `getservent`. The return value is unspecified.

`setserv` [*stayopen*] [Scheme Procedure]
`scm_setserv` (*stayopen*) [C Function]
If *stayopen* is omitted, this is equivalent to `endservent`. Otherwise it is equivalent to `setservent stayopen`.

7.2.11.3 Network Socket Address

A *socket address* object identifies a socket endpoint for communication. In the case of `AF_INET` for instance, the socket address object comprises the host address (or interface on the host) and a port number which specifies a particular open socket in a running client or server process. A socket address object can be created with,

`make-socket-address` *AF_INET ipv4addr port* [Scheme Procedure]
`make-socket-address` *AF_INET6 ipv6addr port* [*flowinfo* [*scopeid*]] [Scheme Procedure]
`make-socket-address` *AF_UNIX path* [Scheme Procedure]
`scm_make_socket_address` (*family*, *address*, *arglist*) [C Function]

Return a new socket address object. The first argument is the address family, one of the `AF` constants, then the arguments vary according to the family.

For `AF_INET` the arguments are an IPv4 network address number (see Section 7.2.11.1 [Network Address Conversion], page 530), and a port number.

For `AF_INET6` the arguments are an IPv6 network address number and a port number. Optional *flowinfo* and *scopeid* arguments may be given (both integers, default 0).

For `AF_UNIX` the argument is a filename (a string).

The C function `scm_make_socket_address` takes the *family* and *address* arguments directly, then *arglist* is a list of further arguments, being the port for IPv4, port and optional flowinfo and scopeid for IPv6, or the empty list `SCM_EOL` for Unix domain.

The following functions access the fields of a socket address object,

`sockaddr:fam` *sa* [Scheme Procedure]
Return the address family from socket address object *sa*. This is one of the `AF` constants (e.g. `AF_INET`).

`sockaddr:path` *sa* [Scheme Procedure]
For an `AF_UNIX` socket address object *sa*, return the filename.

`sockaddr:addr` *sa* [Scheme Procedure]
For an `AF_INET` or `AF_INET6` socket address object *sa*, return the network address number.

`sockaddr:port` *sa* [Scheme Procedure]
For an `AF_INET` or `AF_INET6` socket address object *sa*, return the port number.

`sockaddr:flowinfo` *sa* [Scheme Procedure]
For an `AF_INET6` socket address object *sa*, return the flowinfo value.

`sockaddr:scopeid` *sa* [Scheme Procedure]
For an `AF_INET6` socket address object *sa*, return the scope ID value.

The functions below convert to and from the C `struct sockaddr` (see Section "Address Formats" in *The GNU C Library Reference Manual*). That structure is a generic type, an application can cast to or from `struct sockaddr_in`, `struct sockaddr_in6` or `struct sockaddr_un` according to the address family.

In a `struct sockaddr` taken or returned, the byte ordering in the fields follows the C conventions (see Section "Byte Order Conversion" in *The GNU C Library Reference Manual*). This means network byte order for `AF_INET` host address (`sin_addr.s_addr`) and port number (`sin_port`), and `AF_INET6` port number (`sin6_port`). But at the Scheme level these values are taken or returned in host byte order, so the port is an ordinary integer, and the host address likewise is an ordinary integer (as described in Section 7.2.11.1 [Network Address Conversion], page 530).

`struct sockaddr * scm_c_make_socket_address` (*SCM family, SCM address, SCM args, size_t *outsize*) [C Function]

Return a newly-`malloc`ed `struct sockaddr` created from arguments like those taken by `scm_make_socket_address` above.

The size (in bytes) of the `struct sockaddr` return is stored into `*outsize`. An application must call `free` to release the returned structure when no longer required.

`SCM scm_from_sockaddr` (*const struct sockaddr *address, unsigned address_size*) [C Function]

Return a Scheme socket address object from the C *address* structure. *address_size* is the size in bytes of *address*.

`struct sockaddr * scm_to_sockaddr` (*SCM address, size_t *address_size*) [C Function]

Return a newly-`malloc`ed `struct sockaddr` from a Scheme level socket address object.

The size (in bytes) of the `struct sockaddr` return is stored into `*outsize`. An application must call `free` to release the returned structure when no longer required.

7.2.11.4 Network Sockets and Communication

Socket ports can be created using `socket` and `socketpair`. The ports are initially unbuffered, to make reading and writing to the same port more reliable. A buffer can be added to the port using `setvbuf` (see Section 6.14.6 [Buffering], page 339).

Most systems have limits on how many files and sockets can be open, so it's strongly recommended that socket ports be closed explicitly when no longer required (see Section 6.14.1 [Ports], page 331).

Some of the underlying C functions take values in network byte order, but the convention in Guile is that at the Scheme level everything is ordinary host byte order and conversions are made automatically where necessary.

`socket` *family style proto* [Scheme Procedure]
`scm_socket` (*family, style, proto*) [C Function]

Return a new socket port of the type specified by *family*, *style* and *proto*. All three parameters are integers. The possible values for *family* are as follows, where supported by the system,

`PF_UNIX` [Variable]
`PF_INET` [Variable]
`PF_INET6` [Variable]

The possible values for *style* are as follows, again where supported by the system,

`SOCK_STREAM` [Variable]
`SOCK_DGRAM` [Variable]
`SOCK_RAW` [Variable]
`SOCK_RDM` [Variable]
`SOCK_SEQPACKET` [Variable]

proto can be obtained from a protocol name using `getprotobyname` (see Section 7.2.11.2 [Network Databases], page 532). A value of zero means the default protocol, which is usually right.

A socket cannot by used for communication until it has been connected somewhere, usually with either `connect` or `accept` below.

`socketpair` *family style proto* [Scheme Procedure]
`scm_socketpair` (*family, style, proto*) [C Function]
Return a pair, the `car` and `cdr` of which are two unnamed socket ports connected to each other. The connection is full-duplex, so data can be transferred in either direction between the two.

family, *style* and *proto* are as per `socket` above. But many systems only support socket pairs in the `PF_UNIX` family. Zero is likely to be the only meaningful value for *proto*.

`getsockopt` *sock level optname* [Scheme Procedure]
`setsockopt` *sock level optname value* [Scheme Procedure]
`scm_getsockopt` (*sock, level, optname*) [C Function]
`scm_setsockopt` (*sock, level, optname, value*) [C Function]
Get or set an option on socket port *sock*. `getsockopt` returns the current value. `setsockopt` sets a value and the return is unspecified.

level is an integer specifying a protocol layer, either `SOL_SOCKET` for socket level options, or a protocol number from the `IPPROTO` constants or `getprotoent` (see Section 7.2.11.2 [Network Databases], page 532).

`SOL_SOCKET` [Variable]
`IPPROTO_IP` [Variable]
`IPPROTO_TCP` [Variable]
`IPPROTO_UDP` [Variable]
optname is an integer specifying an option within the protocol layer.

For `SOL_SOCKET` level the following *optname*s are defined (when provided by the system). For their meaning see Section "Socket-Level Options" in *The GNU C Library Reference Manual*, or `man 7 socket`.

`SO_DEBUG` [Variable]
`SO_REUSEADDR` [Variable]
`SO_STYLE` [Variable]
`SO_TYPE` [Variable]
`SO_ERROR` [Variable]
`SO_DONTROUTE` [Variable]
`SO_BROADCAST` [Variable]
`SO_SNDBUF` [Variable]
`SO_RCVBUF` [Variable]
`SO_KEEPALIVE` [Variable]
`SO_OOBINLINE` [Variable]
`SO_NO_CHECK` [Variable]
`SO_PRIORITY` [Variable]
`SO_REUSEPORT` [Variable]
The *value* taken or returned is an integer.

`SO_LINGER` [Variable]
: The *value* taken or returned is a pair of integers (*ENABLE* . *TIMEOUT*). On old systems without timeout support (ie. without `struct linger`), only *ENABLE* has an effect but the value in Guile is always a pair.

For IP level (`IPPROTO_IP`) the following *optname*s are defined (when provided by the system). See `man ip` for what they mean.

`IP_MULTICAST_IF` [Variable]
: This sets the source interface used by multicast traffic.

`IP_MULTICAST_TTL` [Variable]
: This sets the default TTL for multicast traffic. This defaults to 1 and should be increased to allow traffic to pass beyond the local network.

`IP_ADD_MEMBERSHIP` [Variable]
`IP_DROP_MEMBERSHIP` [Variable]
: These can be used only with `setsockopt`, not `getsockopt`. *value* is a pair (*MULTIADDR* . *INTERFACEADDR*) of integer IPv4 addresses (see Section 7.2.11.1 [Network Address Conversion], page 530). *MULTIADDR* is a multicast address to be added to or dropped from the interface *INTERFACEADDR*. *INTERFACEADDR* can be `INADDR_ANY` to have the system select the interface. *INTERFACEADDR* can also be an interface index number, on systems supporting that.

For `IPPROTO_TCP` level the following *optname*s are defined (when provided by the system). For their meaning see `man 7 tcp`.

`TCP_NODELAY` [Variable]
`TCP_CORK` [Variable]
: The *value* taken or returned is an integer.

`shutdown` *sock how* [Scheme Procedure]
`scm_shutdown` (*sock*, *how*) [C Function]
: Sockets can be closed simply by using `close-port`. The `shutdown` procedure allows reception or transmission on a connection to be shut down individually, according to the parameter *how*:

  0 — Stop receiving data for this socket. If further data arrives, reject it.

  1 — Stop trying to transmit data from this socket. Discard any data waiting to be sent. Stop looking for acknowledgement of data already sent; don't retransmit it if it is lost.

  2 — Stop both reception and transmission.

  The return value is unspecified.

`connect` *sock sockaddr* [Scheme Procedure]
`connect` *sock AF_INET ipv4addr port* [Scheme Procedure]
`connect` *sock AF_INET6 ipv6addr port* [*flowinfo* [*scopeid*]] [Scheme Procedure]
`connect` *sock AF_UNIX path* [Scheme Procedure]

`scm_connect` (*sock, fam, address, args*) [C Function]
Initiate a connection on socket port *sock* to a given address. The destination is either a socket address object, or arguments the same as `make-socket-address` would take to make such an object (see Section 7.2.11.3 [Network Socket Address], page 538). Return true unless the socket was configured as non-blocking and the connection could not be made immediately.

```
(connect sock AF_INET INADDR_LOOPBACK 23)
(connect sock (make-socket-address AF_INET INADDR_LOOPBACK 23))
```

`bind` *sock sockaddr* [Scheme Procedure]
`bind` *sock AF_INET ipv4addr port* [Scheme Procedure]
`bind` *sock AF_INET6 ipv6addr port* [*flowinfo* [*scopeid*]] [Scheme Procedure]
`bind` *sock AF_UNIX path* [Scheme Procedure]
`scm_bind` (*sock, fam, address, args*) [C Function]
Bind socket port *sock* to the given address. The address is either a socket address object, or arguments the same as `make-socket-address` would take to make such an object (see Section 7.2.11.3 [Network Socket Address], page 538). The return value is unspecified.

Generally a socket is only explicitly bound to a particular address when making a server, i.e. to listen on a particular port. For an outgoing connection the system will assign a local address automatically, if not already bound.

```
(bind sock AF_INET INADDR_ANY 12345)
(bind sock (make-socket-address AF_INET INADDR_ANY 12345))
```

`listen` *sock backlog* [Scheme Procedure]
`scm_listen` (*sock, backlog*) [C Function]
Enable *sock* to accept connection requests. *backlog* is an integer specifying the maximum length of the queue for pending connections. If the queue fills, new clients will fail to connect until the server calls `accept` to accept a connection from the queue.

The return value is unspecified.

`accept` *sock* [*flags*] [Scheme Procedure]
`scm_accept` (*sock*) [C Function]
Accept a connection from socket port *sock* which has been enabled for listening with `listen` above.

If there are no incoming connections in the queue, there are two possible behaviors, depending on whether *sock* has been configured for non-blocking operation or not:

- If there is no connection waiting and the socket was set to non-blocking mode with the `O_NONBLOCK` port option (see Section 7.2.2 [Ports and File Descriptors], page 497), return `#f` directly.
- Otherwise wait until a connection is available.

The return value is a pair. The `car` is a new socket port, connected and ready to communicate. The `cdr` is a socket address object (see Section 7.2.11.3 [Network Socket Address], page 538) which is where the remote connection is from (like `getpeername` below).

flags, if given, may include `SOCK_CLOEXEC` or `SOCK_NONBLOCK`, which like `O_CLOEXEC` and `O_NONBLOCK` apply to the newly accepted socket.

All communication takes place using the new socket returned. The given *sock* remains bound and listening, and `accept` may be called on it again to get another incoming connection when desired.

`getsockname` *sock* [Scheme Procedure]
`scm_getsockname` (*sock*) [C Function]
Return a socket address object which is the where *sock* is bound locally. *sock* may have obtained its local address from `bind` (above), or if a `connect` is done with an otherwise unbound socket (which is usual) then the system will have assigned an address.

Note that on many systems the address of a socket in the `AF_UNIX` namespace cannot be read.

`getpeername` *sock* [Scheme Procedure]
`scm_getpeername` (*sock*) [C Function]
Return a socket address object which is where *sock* is connected to, i.e. the remote endpoint.

Note that on many systems the address of a socket in the `AF_UNIX` namespace cannot be read.

`recv!` *sock buf* [*flags*] [Scheme Procedure]
`scm_recv` (*sock, buf, flags*) [C Function]
Receive data from a socket port. *sock* must already be bound to the address from which data is to be received. *buf* is a bytevector into which the data will be written. The size of *buf* limits the amount of data which can be received: in the case of packet protocols, if a packet larger than this limit is encountered then some data will be irrevocably lost.

The optional *flags* argument is a value or bitwise OR of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

The value returned is the number of bytes read from the socket.

Note that the data is read directly from the socket file descriptor: any unread buffered port data is ignored.

`send` *sock message* [*flags*] [Scheme Procedure]
`scm_send` (*sock, message, flags*) [C Function]
Transmit bytevector *message* on socket port *sock*. *sock* must already be bound to a destination address. The value returned is the number of bytes transmitted—it's possible for this to be less than the length of *message* if the socket is set to be non-blocking. The optional *flags* argument is a value or bitwise OR of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

Note that the data is written directly to the socket file descriptor: any unflushed buffered port data is ignored.

`recvfrom!` *sock buf* [*flags* [*start* [*end*]]] [Scheme Procedure]
`scm_recvfrom` (*sock, buf, flags, start, end*) [C Function]

Receive data from socket port *sock*, returning the originating address as well as the data. This function is usually for datagram sockets, but can be used on stream-oriented sockets too.

The data received is stored in bytevector *buf*, using either the whole bytevector or just the region between the optional *start* and *end* positions. The size of *buf* limits the amount of data that can be received. For datagram protocols if a packet larger than this is received then excess bytes are irrevocably lost.

The return value is a pair. The `car` is the number of bytes read. The `cdr` is a socket address object (see Section 7.2.11.3 [Network Socket Address], page 538) which is where the data came from, or `#f` if the origin is unknown.

The optional *flags* argument is a or bitwise-OR (`logior`) of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

Data is read directly from the socket file descriptor, any buffered port data is ignored.

On a GNU/Linux system `recvfrom!` is not multi-threading, all threads stop while a `recvfrom!` call is in progress. An application may need to use `select`, `O_NONBLOCK` or `MSG_DONTWAIT` to avoid this.

`sendto` *sock message sockaddr* [*flags*] [Scheme Procedure]
`sendto` *sock message AF_INET ipv4addr port* [*flags*] [Scheme Procedure]
`sendto` *sock message AF_INET6 ipv6addr port* [*flowinfo* [*scopeid* [*flags*]]] [Scheme Procedure]
`sendto` *sock message AF_UNIX path* [*flags*] [Scheme Procedure]
`scm_sendto` (*sock, message, fam, address, args_and_flags*) [C Function]

Transmit bytevector *message* as a datagram socket port *sock*. The destination is specified either as a socket address object, or as arguments the same as would be taken by `make-socket-address` to create such an object (see Section 7.2.11.3 [Network Socket Address], page 538).

The destination address may be followed by an optional *flags* argument which is a `logior` (see Section 6.6.2.13 [Bitwise Operations], page 125) of `MSG_OOB`, `MSG_PEEK`, `MSG_DONTROUTE` etc.

The value returned is the number of bytes transmitted – it's possible for this to be less than the length of *message* if the socket is set to be non-blocking. Note that the data is written directly to the socket file descriptor: any unflushed buffered port data is ignored.

7.2.11.5 Network Socket Examples

The following give examples of how to use network sockets.

Internet Socket Client Example

The following example demonstrates an Internet socket client. It connects to the HTTP daemon running on the local machine and returns the contents of the root index URL.

```
(let ((s (socket PF_INET SOCK_STREAM 0)))
  (connect s AF_INET (inet-pton AF_INET "127.0.0.1") 80)
```

```
(display "GET / HTTP/1.0\r\n\r\n" s)

(do ((line (read-line s) (read-line s)))
    ((eof-object? line))
  (display line)
  (newline)))
```

Internet Socket Server Example

The following example shows a simple Internet server which listens on port 2904 for incoming connections and sends a greeting back to the client.

```
(let ((s (socket PF_INET SOCK_STREAM 0)))
  (setsockopt s SOL_SOCKET SO_REUSEADDR 1)
  ;; Specific address?
  ;; (bind s AF_INET (inet-pton AF_INET "127.0.0.1") 2904)
  (bind s AF_INET INADDR_ANY 2904)
  (listen s 5)

  (simple-format #t "Listening for clients in pid: ~S" (getpid))
  (newline)

  (while #t
    (let* ((client-connection (accept s))
           (client-details (cdr client-connection))
           (client (car client-connection)))
      (simple-format #t "Got new client connection: ~S"
                     client-details)
      (newline)
      (simple-format #t "Client address: ~S"
                     (gethostbyaddr
                      (sockaddr:addr client-details)))
      (newline)
      ;; Send back the greeting to the client port
      (display "Hello client\r\n" client)
      (close client))))
```

7.2.12 System Identification

This section lists the various procedures Guile provides for accessing information about the system it runs on.

`uname` [Scheme Procedure]
`scm_uname ()` [C Function]

Return an object with some information about the computer system the program is running on.

The following procedures accept an object as returned by `uname` and return a selected component (all of which are strings).

`utsname:sysname` *un* [Scheme Procedure]
The name of the operating system.

`utsname:nodename` *un* [Scheme Procedure]
The network name of the computer.

`utsname:release` *un* [Scheme Procedure]
The current release level of the operating system implementation.

`utsname:version` *un* [Scheme Procedure]
The current version level within the release of the operating system.

`utsname:machine` *un* [Scheme Procedure]
A description of the hardware.

`gethostname` [Scheme Procedure]
`scm_gethostname` () [C Function]
Return the host name of the current processor.

`sethostname` *name* [Scheme Procedure]
`scm_sethostname` (*name*) [C Function]
Set the host name of the current processor to *name*. May only be used by the superuser. The return value is not specified.

7.2.13 Locales

`setlocale` *category* [*locale*] [Scheme Procedure]
`scm_setlocale` (*category, locale*) [C Function]
Get or set the current locale, used for various internationalizations. Locales are strings, such as '`sv_SE`'.

If *locale* is given then the locale for the given *category* is set and the new value returned. If *locale* is not given then the current value is returned. *category* should be one of the following values (see Section "Locale Categories" in *The GNU C Library Reference Manual*):

`LC_ALL` [Variable]
`LC_COLLATE` [Variable]
`LC_CTYPE` [Variable]
`LC_MESSAGES` [Variable]
`LC_MONETARY` [Variable]
`LC_NUMERIC` [Variable]
`LC_TIME` [Variable]
A common usage is '`(setlocale LC_ALL "")`', which initializes all categories based on standard environment variables (`LANG` etc). For full details on categories and locale names see Section "Locales and Internationalization" in *The GNU C Library Reference Manual*.

Note that `setlocale` affects locale settings for the whole process. See Section 6.25.1 [i18n Introduction], page 465, for a thread-safe alternative.

7.2.14 Encryption

Please note that the procedures in this section are not suited for strong encryption, they are only interfaces to the well-known and common system library functions of the same name. They are just as good (or bad) as the underlying functions, so you should refer to your system documentation before using them (see Section "Encrypting Passwords" in *The GNU C Library Reference Manual*).

`crypt` *key salt* [Scheme Procedure]
`scm_crypt` (*key*, *salt*) [C Function]
Encrypt *key*, with the addition of *salt* (both strings), using the `crypt` C library call.

Although `getpass` is not an encryption procedure per se, it appears here because it is often used in combination with `crypt`:

`getpass` *prompt* [Scheme Procedure]
`scm_getpass` (*prompt*) [C Function]
Display *prompt* to the standard error output and read a password from `/dev/tty`. If this file is not accessible, it reads from standard input. The password may be up to 127 characters in length. Additional characters and the terminating newline character are discarded. While reading the password, echoing and the generation of signals by special characters is disabled.

7.3 HTTP, the Web, and All That

It has always been possible to connect computers together and share information between them, but the rise of the World Wide Web over the last couple of decades has made it much easier to do so. The result is a richly connected network of computation, in which Guile forms a part.

By "the web", we mean the HTTP protocol[2] as handled by servers, clients, proxies, caches, and the various kinds of messages and message components that can be sent and received by that protocol, notably HTML.

On one level, the web is text in motion: the protocols themselves are textual (though the payload may be binary), and it's possible to create a socket and speak text to the web. But such an approach is obviously primitive. This section details the higher-level data types and operations provided by Guile: URIs, HTTP request and response records, and a conventional web server implementation.

The material in this section is arranged in ascending order, in which later concepts build on previous ones. If you prefer to start with the highest-level perspective, see Section 7.3.10 [Web Examples], page 574, and work your way back.

7.3.1 Types and the Web

It is a truth universally acknowledged, that a program with good use of data types, will be free from many common bugs. Unfortunately, the common practice in web programming seems to ignore this maxim. This subsection makes the case for expressive data types in web programming.

[2] Yes, the P is for protocol, but this phrase appears repeatedly in RFC 2616.

By "expressive data types", we mean that the data types *say* something about how a program solves a problem. For example, if we choose to represent dates using SRFI 19 date records (see Section 7.5.16 [SRFI-19], page 614), this indicates that there is a part of the program that will always have valid dates. Error handling for a number of basic cases, like invalid dates, occurs on the boundary in which we produce a SRFI 19 date record from other types, like strings.

With regards to the web, data types are helpful in the two broad phases of HTTP messages: parsing and generation.

Consider a server, which has to parse a request, and produce a response. Guile will parse the request into an HTTP request object (see Section 7.3.6 [Requests], page 564), with each header parsed into an appropriate Scheme data type. This transition from an incoming stream of characters to typed data is a state change in a program—the strings might parse, or they might not, and something has to happen if they do not. (Guile throws an error in this case.) But after you have the parsed request, "client" code (code built on top of the Guile web framework) will not have to check for syntactic validity. The types already make this information manifest.

This state change on the parsing boundary makes programs more robust, as they themselves are freed from the need to do a number of common error checks, and they can use normal Scheme procedures to handle a request instead of ad-hoc string parsers.

The need for types on the response generation side (in a server) is more subtle, though not less important. Consider the example of a POST handler, which prints out the text that a user submits from a form. Such a handler might include a procedure like this:

```
;; First, a helper procedure
(define (para . contents)
  (string-append "<p>" (string-concatenate contents) "</p>"))

;; Now the meat of our simple web application
(define (you-said text)
  (para "You said: " text))

(display (you-said "Hi!"))
⊣ <p>You said: Hi!</p>
```

This is a perfectly valid implementation, provided that the incoming text does not contain the special HTML characters '`<`', '`>`', or '`&`'. But this provision of a restricted character set is not reflected anywhere in the program itself: we must *assume* that the programmer understands this, and performs the check elsewhere.

Unfortunately, the short history of the practice of programming does not bear out this assumption. A *cross-site scripting* (XSS) vulnerability is just such a common error in which unfiltered user input is allowed into the output. A user could submit a crafted comment to your web site which results in visitors running malicious Javascript, within the security context of your domain:

```
(display (you-said "<script src=\"http://bad.com/nasty.js\" />"))
⊣ <p>You said: <script src="http://bad.com/nasty.js" /></p>
```

The fundamental problem here is that both user data and the program template are represented using strings. This identity means that types can't help the programmer to make a distinction between these two, so they get confused.

There are a number of possible solutions, but perhaps the best is to treat HTML not as strings, but as native s-expressions: as SXML. The basic idea is that HTML is either text, represented by a string, or an element, represented as a tagged list. So '`foo`' becomes '`"foo"`', and '`<b>foo</b>`' becomes '`(b "foo")`'. Attributes, if present, go in a tagged list headed by '`@`', like '`(img (@ (src "http://example.com/foo.png")))`'. See Section 7.21 [SXML], page 748, for more information.

The good thing about SXML is that HTML elements cannot be confused with text. Let's make a new definition of `para`:

```
(define (para . contents)
  `(p ,@contents))

(use-modules (sxml simple))
(sxml->xml (you-said "Hi!"))
⊣ <p>You said: Hi!</p>

(sxml->xml (you-said "<i>Rats, foiled again!</i>"))
⊣ <p>You said: &lt;i&gt;Rats, foiled again!&lt;/i&gt;</p>
```

So we see in the second example that HTML elements cannot be unwittingly introduced into the output. However it is now perfectly acceptable to pass SXML to `you-said`; in fact, that is the big advantage of SXML over everything-as-a-string.

```
(sxml->xml (you-said (you-said "<Hi!>")))
⊣ <p>You said: <p>You said: &lt;Hi!&gt;</p></p>
```

The SXML types allow procedures to *compose*. The types make manifest which parts are HTML elements, and which are text. So you needn't worry about escaping user input; the type transition back to a string handles that for you. XSS vulnerabilities are a thing of the past.

Well. That's all very nice and opinionated and such, but how do I use the thing? Read on!

7.3.2 Universal Resource Identifiers

Guile provides a standard data type for Universal Resource Identifiers (URIs), as defined in RFC 3986.

The generic URI syntax is as follows:

```
URI-reference := [scheme ":"] ["//" [userinfo "@"] host [":" port]] path \
                 [ "?" query ] [ "#" fragment ]
```

For example, in the URI, '`http://www.gnu.org/help/`', the scheme is `http`, the host is `www.gnu.org`, the path is `/help/`, and there is no userinfo, port, query, or fragment.

Userinfo is something of an abstraction, as some legacy URI schemes allowed userinfo of the form *`username:passwd`*. But since passwords do not belong in URIs, the RFC does not want to condone this practice, so it calls anything before the `@` sign *userinfo*.

```
(use-modules (web uri))
```

The following procedures can be found in the `(web uri)` module. Load it into your Guile, using a form like the above, to have access to them.

The most common way to build a URI from Scheme is with the `build-uri` function.

`build-uri` *scheme* [*#:userinfo*=`#f`] [*#:host*=`#f`] [*#:port*=`#f`] [*#:path*=`""`] [*#:query*=`#f`] [*#:fragment*=`#f`] [*#:validate?*=`#t`] [Scheme Procedure]

Construct a URI. *scheme* should be a symbol, *port* either a positive, exact integer or `#f`, and the rest of the fields are either strings or `#f`. If *validate?* is true, also run some consistency checks to make sure that the constructed URI is valid.

`uri?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a URI.

Guile, URIs are represented as URI records, with a number of associated accessors.

`uri-scheme` *uri* [Scheme Procedure]
`uri-userinfo` *uri* [Scheme Procedure]
`uri-host` *uri* [Scheme Procedure]
`uri-port` *uri* [Scheme Procedure]
`uri-path` *uri* [Scheme Procedure]
`uri-query` *uri* [Scheme Procedure]
`uri-fragment` *uri* [Scheme Procedure]

Field accessors for the URI record type. The URI scheme will be a symbol, or `#f` if the object is a relative-ref (see below). The port will be either a positive, exact integer or `#f`, and the rest of the fields will be either strings or `#f` if not present.

`string->uri` *string* [Scheme Procedure]

Parse *string* into a URI object. Return `#f` if the string could not be parsed.

`uri->string` *uri* [*#:include-fragment?*=`#t`] [Scheme Procedure]

Serialize *uri* to a string. If the URI has a port that is the default port for its scheme, the port is not included in the serialization. If *include-fragment?* is given as false, the resulting string will omit the fragment (if any).

`declare-default-port!` *scheme port* [Scheme Procedure]

Declare a default port for the given URI scheme.

`uri-decode` *str* [*#:encoding*=`"utf-8"`] [*#:decode-plus-to-space? #t*] [Scheme Procedure]

Percent-decode the given *str*, according to *encoding*, which should be the name of a character encoding.

Note that this function should not generally be applied to a full URI string. For paths, use `split-and-decode-uri-path` instead. For query strings, split the query on `&` and `=` boundaries, and decode the components separately.

Note also that percent-encoded strings encode *bytes*, not characters. There is no guarantee that a given byte sequence is a valid string encoding. Therefore this routine may signal an error if the decoded bytes are not valid for the given encoding. Pass `#f` for *encoding* if you want decoded bytes as a bytevector directly. See Section 6.14.1 [Ports], page 331, for more information on character encodings.

If *decode-plus-to-space?* is true, which is the default, also replace instances of the plus character '`+`' with a space character. This is needed when parsing `application/x-www-form-urlencoded` data.

Returns a string of the decoded characters, or a bytevector if *encoding* was `#f`.

`uri-encode` *str* [*#:encoding*=`"utf-8"`] [*#:unescaped-chars*] [Scheme Procedure]
Percent-encode any character not in the character set, *unescaped-chars*.

The default character set includes alphanumerics from ASCII, as well as the special characters '`-`', '`.`', '`_`', and '`~`'. Any other character will be percent-encoded, by writing out the character to a bytevector within the given *encoding*, then encoding each byte as `%`*HH*, where *HH* is the hexadecimal representation of the byte.

`split-and-decode-uri-path` *path* [Scheme Procedure]
Split *path* into its components, and decode each component, removing empty components.

For example, `"/foo/bar%20baz/"` decodes to the two-element list, `("foo" "bar baz")`.

`encode-and-join-uri-path` *parts* [Scheme Procedure]
URI-encode each element of *parts*, which should be a list of strings, and join the parts together with `/` as a delimiter.

For example, the list `("scrambled eggs" "biscuits&gravy")` encodes as `"scrambled%20eggs/biscuits%26gravy"`.

Subtypes of URI

As we noted above, not all URI objects have a scheme. You might have noted in the "generic URI syntax" example that the left-hand side of that grammar definition was URI-reference, not URI. A *URI-reference* is a generalization of a URI where the scheme is optional. If no scheme is specified, it is taken to be relative to some other related URI. A common use of URI references is when you want to be vague regarding the choice of HTTP or HTTPS – serving a web page referring to `/foo.css` will use HTTPS if loaded over HTTPS, or HTTP otherwise.

`build-uri-reference` [*#:scheme*=`#f`] [*#:userinfo*=`#f`] [Scheme Procedure]
[*#:host*=`#f`] [*#:port*=`#f`] [*#:path*=`""`] [*#:query*=`#f`] [*#:fragment*=`#f`]
[*#:validate?*=`#t`]
Like `build-uri`, but with an optional scheme.

`uri-reference?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a URI-reference. This is the most general URI predicate, as it includes not only full URIs that have schemes (those that match `uri?`) but also URIs without schemes.

It's also possible to build a *relative-ref*: a URI-reference that explicitly lacks a scheme.

`build-relative-ref` [*#:userinfo*=`#f`] [*#:host*=`#f`] [Scheme Procedure]
[*#:port*=`#f`] [*#:path*=`""`] [*#:query*=`#f`] [*#:fragment*=`#f`]
[*#:validate?*=`#t`]
Like `build-uri`, but with no scheme.

`relative-ref?` *obj* [Scheme Procedure]
: Return `#t` if *obj* is a "relative-ref": a URI-reference that has no scheme. Every URI-reference will either match `uri?` or `relative-ref?` (but not both).

In case it's not clear from the above, the most general of these URI types is the URI-reference, with `build-uri-reference` as the most general constructor. `build-uri` and `build-relative-ref` enforce enforce specific restrictions on the URI-reference. The most generic URI parser is then `string->uri-reference`, and there is also a parser for when you know that you want a relative-ref.

Note that `uri?` will only return `#t` for URI objects that have schemes; that is, it rejects relative-refs.

`string->uri-reference` *string* [Scheme Procedure]
: Parse *string* into a URI object, while not requiring a scheme. Return `#f` if the string could not be parsed.

`string->relative-ref` *string* [Scheme Procedure]
: Parse *string* into a URI object, while asserting that no scheme is present. Return `#f` if the string could not be parsed.

7.3.3 The Hyper-Text Transfer Protocol

The initial motivation for including web functionality in Guile, rather than rely on an external package, was to establish a standard base on which people can share code. To that end, we continue the focus on data types by providing a number of low-level parsers and unparsers for elements of the HTTP protocol.

If you are want to skip the low-level details for now and move on to web pages, see Section 7.3.8 [Web Client], page 569, and see Section 7.3.9 [Web Server], page 571. Otherwise, load the HTTP module, and read on.

```
(use-modules (web http))
```

The focus of the `(web http)` module is to parse and unparse standard HTTP headers, representing them to Guile as native data structures. For example, a `Date:` header will be represented as a SRFI-19 date record (see Section 7.5.16 [SRFI-19], page 614), rather than as a string.

Guile tries to follow RFCs fairly strictly—the road to perdition being paved with compatibility hacks—though some allowances are made for not-too-divergent texts.

Header names are represented as lower-case symbols.

`string->header` *name* [Scheme Procedure]
: Parse *name* to a symbolic header name.

`header->string` *sym* [Scheme Procedure]
: Return the string form for the header named *sym*.

For example:

```
(string->header "Content-Length")
⇒ content-length
(header->string 'content-length)
```

```
⇒ "Content-Length"

(string->header "FOO")
⇒ foo
(header->string 'foo)
⇒ "Foo"
```

Guile keeps a registry of known headers, their string names, and some parsing and serialization procedures. If a header is unknown, its string name is simply its symbol name in title-case.

`known-header?` *sym* [Scheme Procedure]
: Return `#t` if *sym* is a known header, with associated parsers and serialization procedures, or `#f` otherwise.

`header-parser` *sym* [Scheme Procedure]
: Return the value parser for headers named *sym*. The result is a procedure that takes one argument, a string, and returns the parsed value. If the header isn't known to Guile, a default parser is returned that passes through the string unchanged.

`header-validator` *sym* [Scheme Procedure]
: Return a predicate which returns `#t` if the given value is valid for headers named *sym*. The default validator for unknown headers is `string?`.

`header-writer` *sym* [Scheme Procedure]
: Return a procedure that writes values for headers named *sym* to a port. The resulting procedure takes two arguments: a value and a port. The default writer is `display`.

For more on the set of headers that Guile knows about out of the box, see Section 7.3.4 [HTTP Headers], page 556. To add your own, use the `declare-header!` procedure:

`declare-header!` *name parser validator writer* [Scheme Procedure]
[#:multiple?=#f]
: Declare a parser, validator, and writer for a given header.

For example, let's say you are running a web server behind some sort of proxy, and your proxy adds an `X-Client-Address` header, indicating the IPv4 address of the original client. You would like for the HTTP request record to parse out this header to a Scheme value, instead of leaving it as a string. You could register this header with Guile's HTTP stack like this:

```
(declare-header! "X-Client-Address"
  (lambda (str)
    (inet-pton AF_INET str))
  (lambda (ip)
    (and (integer? ip) (exact? ip) (<= 0 ip #xffffffff)))
  (lambda (ip port)
    (display (inet-ntop AF_INET ip) port)))
```

`declare-opaque-header!` *name* [Scheme Procedure]
: A specialised version of `declare-header!` for the case in which you want a header's value to be returned/written "as-is".

`valid-header?` *sym val* [Scheme Procedure]
Return a true value if *val* is a valid Scheme value for the header with name *sym*, or `#f` otherwise.

Now that we have a generic interface for reading and writing headers, we do just that.

`read-header` *port* [Scheme Procedure]
Read one HTTP header from *port*. Return two values: the header name and the parsed Scheme value. May raise an exception if the header was known but the value was invalid.

Returns the end-of-file object for both values if the end of the message body was reached (i.e., a blank line).

`parse-header` *name val* [Scheme Procedure]
Parse *val*, a string, with the parser for the header named *name*. Returns the parsed value.

`write-header` *name val port* [Scheme Procedure]
Write the given header name and value to *port*, using the writer from `header-writer`.

`read-headers` *port* [Scheme Procedure]
Read the headers of an HTTP message from *port*, returning them as an ordered alist.

`write-headers` *headers port* [Scheme Procedure]
Write the given header alist to *port*. Doesn't write the final '`\r\n`', as the user might want to add another header.

The `(web http)` module also has some utility procedures to read and write request and response lines.

`parse-http-method` *str* [*start*] [*end*] [Scheme Procedure]
Parse an HTTP method from *str*. The result is an upper-case symbol, like `GET`.

`parse-http-version` *str* [*start*] [*end*] [Scheme Procedure]
Parse an HTTP version from *str*, returning it as a major–minor pair. For example, `HTTP/1.1` parses as the pair of integers, `(1 . 1)`.

`parse-request-uri` *str* [*start*] [*end*] [Scheme Procedure]
Parse a URI from an HTTP request line. Note that URIs in requests do not have to have a scheme or host name. The result is a URI object.

`read-request-line` *port* [Scheme Procedure]
Read the first line of an HTTP request from *port*, returning three values: the method, the URI, and the version.

`write-request-line` *method uri version port* [Scheme Procedure]
Write the first line of an HTTP request to *port*.

`read-response-line` *port* [Scheme Procedure]
Read the first line of an HTTP response from *port*, returning three values: the HTTP version, the response code, and the "reason phrase".

`write-response-line` *version code reason-phrase port* [Scheme Procedure]
Write the first line of an HTTP response to *port*.

7.3.4 HTTP Headers

In addition to defining the infrastructure to parse headers, the `(web http)` module defines specific parsers and unparsers for all headers defined in the HTTP/1.1 standard.

For example, if you receive a header named '`Accept-Language`' with a value '`en, es;q=0.8`', Guile parses it as a quality list (defined below):

```
(parse-header 'accept-language "en, es;q=0.8")
⇒ ((1000 . "en") (800 . "es"))
```

The format of the value for '`Accept-Language`' headers is defined below, along with all other headers defined in the HTTP standard. (If the header were unknown, the value would have been returned as a string.)

For brevity, the header definitions below are given in the form, *Type* ***name***, indicating that values for the header ***name*** will be of the given *Type*. Since Guile internally treats header names in lower case, in this document we give types title-cased names. A short description of the each header's purpose and an example follow.

For full details on the meanings of all of these headers, see the HTTP 1.1 standard, RFC 2616.

7.3.4.1 HTTP Header Types

Here we define the types that are used below, when defining headers.

`Date` [HTTP Header Type]

A SRFI-19 date.

`KVList` [HTTP Header Type]

A list whose elements are keys or key-value pairs. Keys are parsed to symbols. Values are strings by default. Non-string values are the exception, and are mentioned explicitly below, as appropriate.

`SList` [HTTP Header Type]

A list of strings.

`Quality` [HTTP Header Type]

An exact integer between 0 and 1000. Qualities are used to express preference, given multiple options. An option with a quality of 870, for example, is preferred over an option with quality 500.

(Qualities are written out over the wire as numbers between 0.0 and 1.0, but since the standard only allows three digits after the decimal, it's equivalent to integers between 0 and 1000, so that's what Guile uses.)

`QList` [HTTP Header Type]

A quality list: a list of pairs, the car of which is a quality, and the cdr a string. Used to express a list of options, along with their qualities.

`ETag` [HTTP Header Type]

An entity tag, represented as a pair. The car of the pair is an opaque string, and the cdr is `#t` if the entity tag is a "strong" entity tag, and `#f` otherwise.

7.3.4.2 General Headers

General HTTP headers may be present in any HTTP message.

`KVList cache-control` [HTTP Header]

A key-value list of cache-control directives. See RFC 2616, for more details.

If present, parameters to `max-age`, `max-stale`, `min-fresh`, and `s-maxage` are all parsed as non-negative integers.

If present, parameters to `private` and `no-cache` are parsed as lists of header names, as symbols.

```
(parse-header 'cache-control "no-cache,no-store"
⇒ (no-cache no-store)
(parse-header 'cache-control "no-cache=\"Authorization,Date\",no-store"
⇒ ((no-cache . (authorization date)) no-store)
(parse-header 'cache-control "no-cache=\"Authorization,Date\",max-age=10"
⇒ ((no-cache . (authorization date)) (max-age . 10))
```

`List connection` [HTTP Header]

A list of header names that apply only to this HTTP connection, as symbols. Additionally, the symbol '`close`' may be present, to indicate that the server should close the connection after responding to the request.

```
(parse-header 'connection "close")
⇒ (close)
```

`Date date` [HTTP Header]

The date that a given HTTP message was originated.

```
(parse-header 'date "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`KVList pragma` [HTTP Header]

A key-value list of implementation-specific directives.

```
(parse-header 'pragma "no-cache, broccoli=tasty")
⇒ (no-cache (broccoli . "tasty"))
```

`List trailer` [HTTP Header]

A list of header names which will appear after the message body, instead of with the message headers.

```
(parse-header 'trailer "ETag")
⇒ (etag)
```

`List transfer-encoding` [HTTP Header]

A list of transfer codings, expressed as key-value lists. The only transfer coding defined by the specification is `chunked`.

```
(parse-header 'transfer-encoding "chunked")
⇒ ((chunked))
```

`List upgrade` [HTTP Header]
A list of strings, indicating additional protocols that a server could use in response to a request.

```
(parse-header 'upgrade "WebSocket")
⇒ ("WebSocket")
```

FIXME: parse out more fully?

`List via` [HTTP Header]
A list of strings, indicating the protocol versions and hosts of intermediate servers and proxies. There may be multiple `via` headers in one message.

```
(parse-header 'via "1.0 venus, 1.1 mars")
⇒ ("1.0 venus" "1.1 mars")
```

`List warning` [HTTP Header]
A list of warnings given by a server or intermediate proxy. Each warning is a itself a list of four elements: a code, as an exact integer between 0 and 1000, a host as a string, the warning text as a string, and either `#f` or a SRFI-19 date.

There may be multiple `warning` headers in one message.

```
(parse-header 'warning "123 foo \"core breach imminent\"")
⇒ ((123 "foo" "core-breach imminent" #f))
```

7.3.4.3 Entity Headers

Entity headers may be present in any HTTP message, and refer to the resource referenced in the HTTP request or response.

`List allow` [HTTP Header]
A list of allowed methods on a given resource, as symbols.

```
(parse-header 'allow "GET, HEAD")
⇒ (GET HEAD)
```

`List content-encoding` [HTTP Header]
A list of content codings, as symbols.

```
(parse-header 'content-encoding "gzip")
⇒ (gzip)
```

`List content-language` [HTTP Header]
The languages that a resource is in, as strings.

```
(parse-header 'content-language "en")
⇒ ("en")
```

`UInt content-length` [HTTP Header]
The number of bytes in a resource, as an exact, non-negative integer.

```
(parse-header 'content-length "300")
⇒ 300
```

`URI content-location` [HTTP Header]

The canonical URI for a resource, in the case that it is also accessible from a different URI.

```
(parse-header 'content-location "http://example.com/foo")
⇒ #<<uri> ...>
```

`String content-md5` [HTTP Header]

The MD5 digest of a resource.

```
(parse-header 'content-md5 "ffaea1a79810785575e29e2bd45e2fa5")
⇒ "ffaea1a79810785575e29e2bd45e2fa5"
```

`List content-range` [HTTP Header]

Range specification as a list of three elements: the symbol `bytes`, either the symbol `*` or a pair of integers indicating the byte range, and either `*` or an integer indicating the instance length. Used to indicate that a response only includes part of a resource.

```
(parse-header 'content-range "bytes 10-20/*")
⇒ (bytes (10 . 20) *)
```

`List content-type` [HTTP Header]

The MIME type of a resource, as a symbol, along with any parameters.

```
(parse-header 'content-type "text/plain")
⇒ (text/plain)
(parse-header 'content-type "text/plain;charset=utf-8")
⇒ (text/plain (charset . "utf-8"))
```

Note that the `charset` parameter is something of a misnomer, and the HTTP specification admits this. It specifies the *encoding* of the characters, not the character set.

`Date expires` [HTTP Header]

The date/time after which the resource given in a response is considered stale.

```
(parse-header 'expires "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`Date last-modified` [HTTP Header]

The date/time on which the resource given in a response was last modified.

```
(parse-header 'expires "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

7.3.4.4 Request Headers

Request headers may only appear in an HTTP request, not in a response.

`List accept` [HTTP Header]

A list of preferred media types for a response. Each element of the list is itself a list, in the same format as `content-type`.

```
(parse-header 'accept "text/html,text/plain;charset=utf-8")
⇒ ((text/html) (text/plain (charset . "utf-8")))
```

Preference is expressed with quality values:

```
(parse-header 'accept "text/html;q=0.8,text/plain;q=0.6")
⇒ ((text/html (q . 800)) (text/plain (q . 600)))
```

`QList accept-charset` [HTTP Header]

A quality list of acceptable charsets. Note again that what HTTP calls a "charset" is what Guile calls a "character encoding".

```
(parse-header 'accept-charset "iso-8859-5, unicode-1-1;q=0.8")
⇒ ((1000 . "iso-8859-5") (800 . "unicode-1-1"))
```

`QList accept-encoding` [HTTP Header]

A quality list of acceptable content codings.

```
(parse-header 'accept-encoding "gzip,identity=0.8")
⇒ ((1000 . "gzip") (800 . "identity"))
```

`QList accept-language` [HTTP Header]

A quality list of acceptable languages.

```
(parse-header 'accept-language "cn,en=0.75")
⇒ ((1000 . "cn") (750 . "en"))
```

`Pair authorization` [HTTP Header]

Authorization credentials. The car of the pair indicates the authentication scheme, like `basic`. For basic authentication, the cdr of the pair will be the base64-encoded '*user*:*pass*' string. For other authentication schemes, like `digest`, the cdr will be a key-value list of credentials.

```
(parse-header 'authorization "Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ=="
⇒ (basic . "QWxhZGRpbjpvcGVuIHNlc2FtZQ==")
```

`List expect` [HTTP Header]

A list of expectations that a client has of a server. The expectations are key-value lists.

```
(parse-header 'expect "100-continue")
⇒ ((100-continue))
```

`String from` [HTTP Header]

The email address of a user making an HTTP request.

```
(parse-header 'from "bob@example.com")
⇒ "bob@example.com"
```

`Pair host` [HTTP Header]

The host for the resource being requested, as a hostname-port pair. If no port is given, the port is `#f`.

```
(parse-header 'host "gnu.org:80")
⇒ ("gnu.org" . 80)
(parse-header 'host "gnu.org")
⇒ ("gnu.org" . #f)
```

`*|List if-match` [HTTP Header]

A set of etags, indicating that the request should proceed if and only if the etag of the resource is in that set. Either the symbol `*`, indicating any etag, or a list of entity tags.

```
(parse-header 'if-match "*")
⇒ *
(parse-header 'if-match "asdfadf")
⇒ (("asdfadf" . #t))
(parse-header 'if-match W/"asdfadf")
⇒ (("asdfadf" . #f))
```

`Date if-modified-since` [HTTP Header]

Indicates that a response should proceed if and only if the resource has been modified since the given date.

```
(parse-header 'if-modified-since "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`*|List if-none-match` [HTTP Header]

A set of etags, indicating that the request should proceed if and only if the etag of the resource is not in the set. Either the symbol `*`, indicating any etag, or a list of entity tags.

```
(parse-header 'if-none-match "*")
⇒ *
```

`ETag|Date if-range` [HTTP Header]

Indicates that the range request should proceed if and only if the resource matches a modification date or an etag. Either an entity tag, or a SRFI-19 date.

```
(parse-header 'if-range "\"original-etag\"")
⇒ ("original-etag" . #t)
```

`Date if-unmodified-since` [HTTP Header]

Indicates that a response should proceed if and only if the resource has not been modified since the given date.

```
(parse-header 'if-not-modified-since "Tue, 15 Nov 1994 08:12:31 GMT")
⇒ #<date ...>
```

`UInt max-forwards` [HTTP Header]

The maximum number of proxy or gateway hops that a request should be subject to.

```
(parse-header 'max-forwards "10")
⇒ 10
```

`Pair proxy-authorization` [HTTP Header]

Authorization credentials for a proxy connection. See the documentation for `authorization` above for more information on the format.

```
(parse-header 'proxy-authorization "Digest foo=bar,baz=qux"
⇒ (digest (foo . "bar") (baz . "qux"))
```

`Pair range` [HTTP Header]

A range request, indicating that the client wants only part of a resource. The car of the pair is the symbol `bytes`, and the cdr is a list of pairs. Each element of the cdr indicates a range; the car is the first byte position and the cdr is the last byte position, as integers, or `#f` if not given.

```
(parse-header 'range "bytes=10-30,50-")
⇒ (bytes (10 . 30) (50 . #f))
```

`URI referer` [HTTP Header]

The URI of the resource that referred the user to this resource. The name of the header is a misspelling, but we are stuck with it.

```
(parse-header 'referer "http://www.gnu.org/")
⇒ #<uri ...>
```

`List te` [HTTP Header]

A list of transfer codings, expressed as key-value lists. A common transfer coding is `trailers`.

```
(parse-header 'te "trailers")
⇒ ((trailers))
```

`String user-agent` [HTTP Header]

A string indicating the user agent making the request. The specification defines a structured format for this header, but it is widely disregarded, so Guile does not attempt to parse strictly.

```
(parse-header 'user-agent "Mozilla/5.0")
⇒ "Mozilla/5.0"
```

7.3.4.5 Response Headers

`List accept-ranges` [HTTP Header]

A list of range units that the server supports, as symbols.

```
(parse-header 'accept-ranges "bytes")
⇒ (bytes)
```

`UInt age` [HTTP Header]

The age of a cached response, in seconds.

```
(parse-header 'age "3600")
⇒ 3600
```

`ETag etag` [HTTP Header]

The entity-tag of the resource.

```
(parse-header 'etag "\"foo\"")
⇒ ("foo" . #t)
```

`URI-reference location` [HTTP Header]

A URI reference on which a request may be completed. Used in combination with a redirecting status code to perform client-side redirection.

```
(parse-header 'location "http://example.com/other")
⇒ #<uri ...>
```

`List proxy-authenticate` [HTTP Header]
A list of challenges to a proxy, indicating the need for authentication.

```
(parse-header 'proxy-authenticate "Basic realm=\"foo\"")
⇒ ((basic (realm . "foo")))
```

`UInt|Date retry-after` [HTTP Header]
Used in combination with a server-busy status code, like 503, to indicate that a client should retry later. Either a number of seconds, or a date.

```
(parse-header 'retry-after "60")
⇒ 60
```

`String server` [HTTP Header]
A string identifying the server.

```
(parse-header 'server "My first web server")
⇒ "My first web server"
```

`*|List vary` [HTTP Header]
A set of request headers that were used in computing this response. Used to indicate that server-side content negotiation was performed, for example in response to the `accept-language` header. Can also be the symbol `*`, indicating that all headers were considered.

```
(parse-header 'vary "Accept-Language, Accept")
⇒ (accept-language accept)
```

`List www-authenticate` [HTTP Header]
A list of challenges to a user, indicating the need for authentication.

```
(parse-header 'www-authenticate "Basic realm=\"foo\"")
⇒ ((basic (realm . "foo")))
```

7.3.5 Transfer Codings

HTTP 1.1 allows for various transfer codings to be applied to message bodies. These include various types of compression, and HTTP chunked encoding. Currently, only chunked encoding is supported by guile.

Chunked coding is an optional coding that may be applied to message bodies, to allow messages whose length is not known beforehand to be returned. Such messages can be split into chunks, terminated by a final zero length chunk.

In order to make dealing with encodings more simple, guile provides procedures to create ports that "wrap" existing ports, applying transformations transparently under the hood.

These procedures are in the `(web http)` module.

```
(use-modules (web http))
```

`make-chunked-input-port` *port* [*#:keep-alive?=#f*] [Scheme Procedure]
Returns a new port, that transparently reads and decodes chunk-encoded data from *port*. If no more chunk-encoded data is available, it returns the end-of-file object. When the port is closed, *port* will also be closed, unless *keep-alive?* is true.

```
(use-modules (ice-9 rdelim))

(define s "5\r\nFirst\r\nA\r\n line\n Sec\r\n8\r\nond line\r\n0\r\n")
(define p (make-chunked-input-port (open-input-string s)))
(read-line s)
⇒ "First line"
(read-line s)
⇒ "Second line"
```

`make-chunked-output-port` *port* [*#:keep-alive?=#f*] [Scheme Procedure]

Returns a new port, which transparently encodes data as chunk-encoded before writing it to *port*. Whenever a write occurs on this port, it buffers it, until the port is flushed, at which point it writes a chunk containing all the data written so far. When the port is closed, the data remaining is written to *port*, as is the terminating zero chunk. It also causes *port* to be closed, unless *keep-alive?* is true.

Note. Forcing a chunked output port when there is no data is buffered does not write a zero chunk, as this would cause the data to be interpreted incorrectly by the client.

```
(call-with-output-string
  (lambda (out)
    (define out* (make-chunked-output-port out #:keep-alive? #t))
    (display "first chunk" out*)
    (force-output out*)
    (force-output out*) ; note this does not write a zero chunk
    (display "second chunk" out*)
    (close-port out*)))
⇒ "b\r\nfirst chunk\r\nc\r\nsecond chunk\r\n0\r\n"
```

7.3.6 HTTP Requests

```
(use-modules (web request))
```

The request module contains a data type for HTTP requests.

7.3.6.1 An Important Note on Character Sets

HTTP requests consist of two parts: the request proper, consisting of a request line and a set of headers, and (optionally) a body. The body might have a binary content-type, and even in the textual case its length is specified in bytes, not characters.

Therefore, HTTP is a fundamentally binary protocol. However the request line and headers are specified to be in a subset of ASCII, so they can be treated as text, provided that the port's encoding is set to an ASCII-compatible one-byte-per-character encoding. ISO-8859-1 (latin-1) is just such an encoding, and happens to be very efficient for Guile.

So what Guile does when reading requests from the wire, or writing them out, is to set the port's encoding to latin-1, and treating the request headers as text.

The request body is another issue. For binary data, the data is probably in a bytevector, so we use the R6RS binary output procedures to write out the binary payload. Textual data usually has to be written out to some character encoding, usually UTF-8, and then the resulting bytevector is written out to the port.

In summary, Guile reads and writes HTTP over latin-1 sockets, without any loss of generality.

7.3.6.2 Request API

`request?` *obj* [Scheme Procedure]
`request-method` *request* [Scheme Procedure]
`request-uri` *request* [Scheme Procedure]
`request-version` *request* [Scheme Procedure]
`request-headers` *request* [Scheme Procedure]
`request-meta` *request* [Scheme Procedure]
`request-port` *request* [Scheme Procedure]

A predicate and field accessors for the request type. The fields are as follows:

`method` The HTTP method, for example, `GET`.

`uri` The URI as a URI record.

`version` The HTTP version pair, like `(1 . 1)`.

`headers` The request headers, as an alist of parsed values.

`meta` An arbitrary alist of other data, for example information returned in the `sockaddr` from `accept` (see Section 7.2.11.4 [Network Sockets and Communication], page 540).

`port` The port on which to read or write a request body, if any.

`read-request` *port* [*meta='()*] [Scheme Procedure]

Read an HTTP request from *port*, optionally attaching the given metadata, *meta*.

As a side effect, sets the encoding on *port* to ISO-8859-1 (latin-1), so that reading one character reads one byte. See the discussion of character sets above, for more information.

Note that the body is not part of the request. Once you have read a request, you may read the body separately, and likewise for writing requests.

`build-request` *uri* [*#:method='GET*] [*#:version='(1 . 1)*] [*#:headers='()*] [*#:port=#f*] [*#:meta='()*] [*#:validate-headers?=#t*] [Scheme Procedure]

Construct an HTTP request object. If *validate-headers?* is true, the headers are each run through their respective validators.

`write-request` *r port* [Scheme Procedure]

Write the given HTTP request to *port*.

Return a new request, whose `request-port` will continue writing on *port*, perhaps using some transfer encoding.

`read-request-body` *r* [Scheme Procedure]

Reads the request body from *r*, as a bytevector. Return `#f` if there was no request body.

`write-request-body` *r bv* [Scheme Procedure]

Write *bv*, a bytevector, to the port corresponding to the HTTP request *r*.

The various headers that are typically associated with HTTP requests may be accessed with these dedicated accessors. See Section 7.3.4 [HTTP Headers], page 556, for more information on the format of parsed headers.

`request-accept` *request* [*default='()*] [Scheme Procedure]
`request-accept-charset` *request* [*default='()*] [Scheme Procedure]
`request-accept-encoding` *request* [*default='()*] [Scheme Procedure]
`request-accept-language` *request* [*default='()*] [Scheme Procedure]
`request-allow` *request* [*default='()*] [Scheme Procedure]
`request-authorization` *request* [*default=#f*] [Scheme Procedure]
`request-cache-control` *request* [*default='()*] [Scheme Procedure]
`request-connection` *request* [*default='()*] [Scheme Procedure]
`request-content-encoding` *request* [*default='()*] [Scheme Procedure]
`request-content-language` *request* [*default='()*] [Scheme Procedure]
`request-content-length` *request* [*default=#f*] [Scheme Procedure]
`request-content-location` *request* [*default=#f*] [Scheme Procedure]
`request-content-md5` *request* [*default=#f*] [Scheme Procedure]
`request-content-range` *request* [*default=#f*] [Scheme Procedure]
`request-content-type` *request* [*default=#f*] [Scheme Procedure]
`request-date` *request* [*default=#f*] [Scheme Procedure]
`request-expect` *request* [*default='()*] [Scheme Procedure]
`request-expires` *request* [*default=#f*] [Scheme Procedure]
`request-from` *request* [*default=#f*] [Scheme Procedure]
`request-host` *request* [*default=#f*] [Scheme Procedure]
`request-if-match` *request* [*default=#f*] [Scheme Procedure]
`request-if-modified-since` *request* [*default=#f*] [Scheme Procedure]
`request-if-none-match` *request* [*default=#f*] [Scheme Procedure]
`request-if-range` *request* [*default=#f*] [Scheme Procedure]
`request-if-unmodified-since` *request* [*default=#f*] [Scheme Procedure]
`request-last-modified` *request* [*default=#f*] [Scheme Procedure]
`request-max-forwards` *request* [*default=#f*] [Scheme Procedure]
`request-pragma` *request* [*default='()*] [Scheme Procedure]
`request-proxy-authorization` *request* [*default=#f*] [Scheme Procedure]
`request-range` *request* [*default=#f*] [Scheme Procedure]
`request-referer` *request* [*default=#f*] [Scheme Procedure]
`request-te` *request* [*default=#f*] [Scheme Procedure]
`request-trailer` *request* [*default='()*] [Scheme Procedure]
`request-transfer-encoding` *request* [*default='()*] [Scheme Procedure]
`request-upgrade` *request* [*default='()*] [Scheme Procedure]
`request-user-agent` *request* [*default=#f*] [Scheme Procedure]
`request-via` *request* [*default='()*] [Scheme Procedure]
`request-warning` *request* [*default='()*] [Scheme Procedure]

Return the given request header, or *default* if none was present.

`request-absolute-uri` *r* [*default-host=#f*] [Scheme Procedure]
[*default-port=#f*] [*default-scheme=#f*]
A helper routine to determine the absolute URI of a request, using the `host` header and the default scheme, host and port. If there is no default scheme and the URI is not itself absolute, an error is signalled.

7.3.7 HTTP Responses

```
(use-modules (web response))
```

As with requests (see Section 7.3.6 [Requests], page 564), Guile offers a data type for HTTP responses. Again, the body is represented separately from the request.

`response?` *obj* [Scheme Procedure]
`response-version` *response* [Scheme Procedure]
`response-code` *response* [Scheme Procedure]
`response-reason-phrase` *response* [Scheme Procedure]
`response-headers` *response* [Scheme Procedure]
`response-port` *response* [Scheme Procedure]
A predicate and field accessors for the response type. The fields are as follows:

`version` The HTTP version pair, like `(1 . 1)`.

`code` The HTTP response code, like `200`.

`reason-phrase`
The reason phrase, or the standard reason phrase for the response's code.

`headers` The response headers, as an alist of parsed values.

`port` The port on which to read or write a response body, if any.

`read-response` *port* [Scheme Procedure]
Read an HTTP response from *port*.

As a side effect, sets the encoding on *port* to ISO-8859-1 (latin-1), so that reading one character reads one byte. See the discussion of character sets in Section 7.3.7 [Responses], page 567, for more information.

`build-response` [*#:version='(1 . 1)*] [*#:code=200*] [Scheme Procedure]
[*#:reason-phrase=#f*] [*#:headers='()*] [*#:port=#f*]
[*#:validate-headers?=#t*]
Construct an HTTP response object. If *validate-headers?* is true, the headers are each run through their respective validators.

`adapt-response-version` *response version* [Scheme Procedure]
Adapt the given response to a different HTTP version. Return a new HTTP response.

The idea is that many applications might just build a response for the default HTTP version, and this method could handle a number of programmatic transformations to respond to older HTTP versions (0.9 and 1.0). But currently this function is a bit heavy-handed, just updating the version field.

`write-response` *r port* [Scheme Procedure]
Write the given HTTP response to *port*.

Return a new response, whose `response-port` will continue writing on *port*, perhaps using some transfer encoding.

`response-must-not-include-body?` *r* [Scheme Procedure]
Some responses, like those with status code 304, are specified as never having bodies. This predicate returns `#t` for those responses.

Note also, though, that responses to `HEAD` requests must also not have a body.

`response-body-port` *r* [*#:decode?=#t*] [*#:keep-alive?=#t*] [Scheme Procedure]
Return an input port from which the body of *r* can be read. The encoding of the returned port is set according to *r*'s `content-type` header, when it's textual, except if *decode?* is `#f`. Return `#f` when no body is available.

When *keep-alive?* is `#f`, closing the returned port also closes *r*'s response port.

`read-response-body` *r* [Scheme Procedure]
Read the response body from *r*, as a bytevector. Returns `#f` if there was no response body.

`write-response-body` *r bv* [Scheme Procedure]
Write *bv*, a bytevector, to the port corresponding to the HTTP response *r*.

As with requests, the various headers that are typically associated with HTTP responses may be accessed with these dedicated accessors. See Section 7.3.4 [HTTP Headers], page 556, for more information on the format of parsed headers.

`response-accept-ranges` *response* [*default=#f*] [Scheme Procedure]
`response-age` *response* [*default='()*] [Scheme Procedure]
`response-allow` *response* [*default='()*] [Scheme Procedure]
`response-cache-control` *response* [*default='()*] [Scheme Procedure]
`response-connection` *response* [*default='()*] [Scheme Procedure]
`response-content-encoding` *response* [*default='()*] [Scheme Procedure]
`response-content-language` *response* [*default='()*] [Scheme Procedure]
`response-content-length` *response* [*default=#f*] [Scheme Procedure]
`response-content-location` *response* [*default=#f*] [Scheme Procedure]
`response-content-md5` *response* [*default=#f*] [Scheme Procedure]
`response-content-range` *response* [*default=#f*] [Scheme Procedure]
`response-content-type` *response* [*default=#f*] [Scheme Procedure]
`response-date` *response* [*default=#f*] [Scheme Procedure]
`response-etag` *response* [*default=#f*] [Scheme Procedure]
`response-expires` *response* [*default=#f*] [Scheme Procedure]
`response-last-modified` *response* [*default=#f*] [Scheme Procedure]
`response-location` *response* [*default=#f*] [Scheme Procedure]
`response-pragma` *response* [*default='()*] [Scheme Procedure]
`response-proxy-authenticate` *response* [*default=#f*] [Scheme Procedure]
`response-retry-after` *response* [*default=#f*] [Scheme Procedure]
`response-server` *response* [*default=#f*] [Scheme Procedure]

`response-trailer` *response* [*default='()*] [Scheme Procedure]
`response-transfer-encoding` *response* [*default='()*] [Scheme Procedure]
`response-upgrade` *response* [*default='()*] [Scheme Procedure]
`response-vary` *response* [*default='()*] [Scheme Procedure]
`response-via` *response* [*default='()*] [Scheme Procedure]
`response-warning` *response* [*default='()*] [Scheme Procedure]
`response-www-authenticate` *response* [*default=#f*] [Scheme Procedure]
Return the given response header, or *default* if none was present.

`text-content-type?` *type* [Scheme Procedure]
Return `#t` if *type*, a symbol as returned by `response-content-type`, represents a textual type such as `text/plain`.

7.3.8 Web Client

`(web client)` provides a simple, synchronous HTTP client, built on the lower-level HTTP, request, and response modules.

```
(use-modules (web client))
```

`open-socket-for-uri` *uri* [*#:verify-certificate? #t*] [Scheme Procedure]
Return an open input/output port for a connection to URI. Guile dynamically loads GnuTLS for HTTPS support. See Section "Guile Preparations" in *GnuTLS-Guile*, for more information.

When *verify-certificate?* is true, verify the server's X.509 certificates against those read from `x509-certificate-directory`. When an error occurs—e.g., the server's certificate has expired, or its host name does not match—raise a `tls-certificate-error` exception. The arguments to the `tls-certificate-error` exception are:

1. a symbol indicating the failure cause, `host-mismatch` if the certificate's host name does not match the server's host name, and `invalid-certificate` for other causes;
2. the server's X.509 certificate (see Section "Guile Reference" in *GnuTLS-Guile*);
3. the server's host name (a string);
4. in the case of `invalid-certificate` errors, a list of GnuTLS certificate status values—one of the `certificate-status/` constants, such as `certificate-status/signer-not-found` or `certificate-status/revoked`.

`http-request` *uri arg...* [Scheme Procedure]
Connect to the server corresponding to *uri* and make a request over HTTP, using *method* (`GET`, `HEAD`, `POST`, etc.).

The following keyword arguments allow you to modify the requests in various ways, for example attaching a body to the request, or setting specific headers. The following table lists the keyword arguments and their default values.

```
#:method 'GET
#:body #f
```

```
#:verify-certificate? #t
#:port (open-socket-for-uri uri #:verify-certificate?
verify-certificate?)
#:version '(1 . 1)
#:keep-alive? #f
#:headers '()
#:decode-body? #t
#:streaming? #f
```

If you already have a port open, pass it as *port*. Otherwise, a connection will be opened to the server corresponding to *uri*. Any extra headers in the alist *headers* will be added to the request.

If *body* is not `#f`, a message body will also be sent with the HTTP request. If *body* is a string, it is encoded according to the content-type in *headers*, defaulting to UTF-8. Otherwise *body* should be a bytevector, or `#f` for no body. Although a message body may be sent with any request, usually only `POST` and `PUT` requests have bodies.

If *decode-body?* is true, as is the default, the body of the response will be decoded to string, if it is a textual content-type. Otherwise it will be returned as a bytevector.

However, if *streaming?* is true, instead of eagerly reading the response body from the server, this function only reads off the headers. The response body will be returned as a port on which the data may be read.

Unless *keep-alive?* is true, the port will be closed after the full response body has been read.

If *port* is false, *uri* denotes an HTTPS URL, and *verify-certificate?* is true, verify X.509 certificates against those available in `x509-certificate-directory`.

Returns two values: the response read from the server, and the response body as a string, bytevector, #f value, or as a port (if *streaming?* is true).

`http-get` *uri arg...* [Scheme Procedure]
`http-head` *uri arg...* [Scheme Procedure]
`http-post` *uri arg...* [Scheme Procedure]
`http-put` *uri arg...* [Scheme Procedure]
`http-delete` *uri arg...* [Scheme Procedure]
`http-trace` *uri arg...* [Scheme Procedure]
`http-options` *uri arg...* [Scheme Procedure]

Connect to the server corresponding to *uri* and make a request over HTTP, using the appropriate method (`GET`, `HEAD`, `POST`, etc.).

These procedures are variants of `http-request` specialized with a specific *method* argument, and have the same prototype: a URI followed by an optional sequence of keyword arguments. See [http-request], page 569, for full documentation on the various keyword arguments.

`x509-certificate-directory` [Scheme Parameter]

This parameter gives the name of the directory where X.509 certificates for HTTPS connections should be looked for.

Its default value is one of:

- the value of the `GUILE_TLS_CERTIFICATE_DIRECTORY` environment variable;

- or the value of the `SSL_CERT_DIR` environment variable (also honored by the OpenSSL library);
- or, as a last resort, `"/etc/ssl/certs"`.

X.509 certificates are used when authenticating the identity of a remote site, when the `#:verify-certificate?` argument to `open-socket-for-uri`, to `http-request`, or to related procedures is true.

`http-get` is useful for making one-off requests to web sites. If you are writing a web spider or some other client that needs to handle a number of requests in parallel, it's better to build an event-driven URL fetcher, similar in structure to the web server (see Section 7.3.9 [Web Server], page 571).

Another option, good but not as performant, would be to use threads, possibly via par-map or futures.

`current-http-proxy` [Scheme Parameter]
`current-https-proxy` [Scheme Parameter]

Either `#f` or a non-empty string containing the URL of the HTTP or HTTPS proxy server to be used by the procedures in the `(web client)` module, including `open-socket-for-uri`. Its initial value is based on the `http_proxy` and `https_proxy` environment variables.

```
(current-http-proxy) ⇒ "http://localhost:8123/"
(parameterize ((current-http-proxy #f))
  (http-get "http://example.com/"))  ; temporarily bypass proxy
(current-http-proxy) ⇒ "http://localhost:8123/"
```

7.3.9 Web Server

`(web server)` is a generic web server interface, along with a main loop implementation for web servers controlled by Guile.

```
(use-modules (web server))
```

The lowest layer is the `<server-impl>` object, which defines a set of hooks to open a server, read a request from a client, write a response to a client, and close a server. These hooks – `open`, `read`, `write`, and `close`, respectively – are bound together in a `<server-impl>` object. Procedures in this module take a `<server-impl>` object, if needed.

A `<server-impl>` may also be looked up by name. If you pass the `http` symbol to `run-server`, Guile looks for a variable named `http` in the `(web server http)` module, which should be bound to a `<server-impl>` object. Such a binding is made by instantiation of the `define-server-impl` syntax. In this way the run-server loop can automatically load other backends if available.

The life cycle of a server goes as follows:

1. The `open` hook is called, to open the server. `open` takes zero or more arguments, depending on the backend, and returns an opaque server socket object, or signals an error.
2. The `read` hook is called, to read a request from a new client. The `read` hook takes one argument, the server socket. It should return three values: an opaque client socket, the

request, and the request body. The request should be a `<request>` object, from `(web request)`. The body should be a string or a bytevector, or `#f` if there is no body.

If the read failed, the `read` hook may return #f for the client socket, request, and body.

3. A user-provided handler procedure is called, with the request and body as its arguments. The handler should return two values: the response, as a `<response>` record from `(web response)`, and the response body as bytevector, or `#f` if not present.

   The respose and response body are run through `sanitize-response`, documented below. This allows the handler writer to take some convenient shortcuts: for example, instead of a `<response>`, the handler can simply return an alist of headers, in which case a default response object is constructed with those headers. Instead of a bytevector for the body, the handler can return a string, which will be serialized into an appropriate encoding; or it can return a procedure, which will be called on a port to write out the data. See the `sanitize-response` documentation, for more.
4. The `write` hook is called with three arguments: the client socket, the response, and the body. The `write` hook returns no values.
5. At this point the request handling is complete. For a loop, we loop back and try to read a new request.
6. If the user interrupts the loop, the `close` hook is called on the server socket.

A user may define a server implementation with the following form:

`define-server-impl` *name open read write close* [Scheme Syntax]
Make a `<server-impl>` object with the hooks *open*, *read*, *write*, and *close*, and bind it to the symbol *name* in the current module.

`lookup-server-impl` *impl* [Scheme Procedure]
Look up a server implementation. If *impl* is a server implementation already, it is returned directly. If it is a symbol, the binding named *impl* in the `(web server impl)` module is looked up. Otherwise an error is signaled.

Currently a server implementation is a somewhat opaque type, useful only for passing to other procedures in this module, like `read-client`.

The `(web server)` module defines a number of routines that use `<server-impl>` objects to implement parts of a web server. Given that we don't expose the accessors for the various fields of a `<server-impl>`, indeed these routines are the only procedures with any access to the impl objects.

`open-server` *impl open-params* [Scheme Procedure]
Open a server for the given implementation. Return one value, the new server object. The implementation's `open` procedure is applied to *open-params*, which should be a list.

`read-client` *impl server* [Scheme Procedure]
Read a new client from *server*, by applying the implementation's `read` procedure to the server. If successful, return three values: an object corresponding to the client, a request object, and the request body. If any exception occurs, return `#f` for all three values.

`handle-request` *handler request body state* [Scheme Procedure]
Handle a given request, returning the response and body.

The response and response body are produced by calling the given *handler* with *request* and *body* as arguments.

The elements of *state* are also passed to *handler* as arguments, and may be returned as additional values. The new *state*, collected from the *handler*'s return values, is then returned as a list. The idea is that a server loop receives a handler from the user, along with whatever state values the user is interested in, allowing the user's handler to explicitly manage its state.

`sanitize-response` *request response body* [Scheme Procedure]
"Sanitize" the given response and body, making them appropriate for the given request.

As a convenience to web handler authors, *response* may be given as an alist of headers, in which case it is used to construct a default response. Ensures that the response version corresponds to the request version. If *body* is a string, encodes the string to a bytevector, in an encoding appropriate for *response*. Adds a `content-length` and `content-type` header, as necessary.

If *body* is a procedure, it is called with a port as an argument, and the output collected as a bytevector. In the future we might try to instead use a compressing, chunk-encoded port, and call this procedure later, in the write-client procedure. Authors are advised not to rely on the procedure being called at any particular time.

`write-client` *impl server client response body* [Scheme Procedure]
Write an HTTP response and body to *client*. If the server and client support persistent connections, it is the implementation's responsibility to keep track of the client thereafter, presumably by attaching it to the *server* argument somehow.

`close-server` *impl server* [Scheme Procedure]
Release resources allocated by a previous invocation of `open-server`.

Given the procedures above, it is a small matter to make a web server:

`serve-one-client` *handler impl server state* [Scheme Procedure]
Read one request from *server*, call *handler* on the request and body, and write the response to the client. Return the new state produced by the handler procedure.

`run-server` *handler* [*impl='http*] [*open-params='()*] *arg ...* [Scheme Procedure]
Run Guile's built-in web server.

handler should be a procedure that takes two or more arguments, the HTTP request and request body, and returns two or more values, the response and response body.

For examples, skip ahead to the next section, Section 7.3.10 [Web Examples], page 574.

The response and body will be run through `sanitize-response` before sending back to the client.

Additional arguments to *handler* are taken from *arg ...*. These arguments comprise a *state*. Additional return values are accumulated into a new state, which will be used for subsequent requests. In this way a handler can explicitly manage its state.

The default web server implementation is `http`, which binds to a socket, listening for request on that port.

`http` [*#:host=#f*] [*#:family=AF_INET*] [*#:addr=INADDR_LOOPBACK*] [*#:port 8080*] [*#:socket*] [HTTP Implementation]

The default HTTP implementation. We document it as a function with keyword arguments, because that is precisely the way that it is – all of the *open-params* to `run-server` get passed to the implementation's open function.

```
;; The defaults: localhost:8080
(run-server handler)
;; Same thing
(run-server handler 'http '())
;; On a different port
(run-server handler 'http '(#:port 8081))
;; IPv6
(run-server handler 'http '(#:family AF_INET6 #:port 8081))
;; Custom socket
(run-server handler 'http `(#:socket ,(sudo-make-me-a-socket)))
```

7.3.10 Web Examples

Well, enough about the tedious internals. Let's make a web application!

7.3.10.1 Hello, World!

The first program we have to write, of course, is "Hello, World!". This means that we have to implement a web handler that does what we want.

Now we define a handler, a function of two arguments and two return values:

```
(define (handler request request-body)
  (values response response-body))
```

In this first example, we take advantage of a short-cut, returning an alist of headers instead of a proper response object. The response body is our payload:

```
(define (hello-world-handler request request-body)
  (values '((content-type . (text/plain)))
          "Hello World!"))
```

Now let's test it, by running a server with this handler. Load up the web server module if you haven't yet done so, and run a server with this handler:

```
(use-modules (web server))
(run-server hello-world-handler)
```

By default, the web server listens for requests on `localhost:8080`. Visit that address in your web browser to test. If you see the string, `Hello World!`, sweet!

7.3.10.2 Inspecting the Request

The Hello World program above is a general greeter, responding to all URIs. To make a more exclusive greeter, we need to inspect the request object, and conditionally produce different results. So let's load up the request, response, and URI modules, and do just that.

```
(use-modules (web server)) ; you probably did this already
```

```
(use-modules (web request)
             (web response)
             (web uri))

(define (request-path-components request)
  (split-and-decode-uri-path (uri-path (request-uri request))))

(define (hello-hacker-handler request body)
  (if (equal? (request-path-components request)
              '("hacker"))
      (values '((content-type . (text/plain)))
              "Hello hacker!")
      (not-found request)))

(run-server hello-hacker-handler)
```

Here we see that we have defined a helper to return the components of the URI path as a list of strings, and used that to check for a request to `/hacker/`. Then the success case is just as before – visit `http://localhost:8080/hacker/` in your browser to check.

You should always match against URI path components as decoded by `split-and-decode-uri-path`. The above example will work for `/hacker/`, `//hacker///`, and `/h%61ck%65r`.

But we forgot to define `not-found`! If you are pasting these examples into a REPL, accessing any other URI in your web browser will drop your Guile console into the debugger:

```
<unnamed port>:38:7: In procedure module-lookup:
<unnamed port>:38:7: Unbound variable: not-found

Entering a new prompt.  Type ‘,bt’ for a backtrace or ‘,q’ to continue.
scheme@(guile-user) [1]>
```

So let's define the function, right there in the debugger. As you probably know, we'll want to return a 404 response.

```
;; Paste this in your REPL
(define (not-found request)
  (values (build-response #:code 404)
          (string-append "Resource not found: "
                         (uri->string (request-uri request)))))

;; Now paste this to let the web server keep going:
,continue
```

Now if you access `http://localhost/foo/`, you get this error message. (Note that some popular web browsers won't show server-generated 404 messages, showing their own instead, unless the 404 message body is long enough.)

7.3.10.3 Higher-Level Interfaces

The web handler interface is a common baseline that all kinds of Guile web applications can use. You will usually want to build something on top of it, however, especially when

producing HTML. Here is a simple example that builds up HTML output using SXML (see Section 7.21 [SXML], page 748).

First, load up the modules:

```
(use-modules (web server)
             (web request)
             (web response)
             (sxml simple))
```

Now we define a simple templating function that takes a list of HTML body elements, as SXML, and puts them in our super template:

```
(define (templatize title body)
  `(html (head (title ,title))
         (body ,@body)))
```

For example, the simplest Hello HTML can be produced like this:

```
(sxml->xml (templatize "Hello!" '((b "Hi!"))))
⊣
<html><head><title>Hello!</title></head><body><b>Hi!</b></body></html>
```

Much better to work with Scheme data types than to work with HTML as strings. Now we define a little response helper:

```
(define* (respond #:optional body #:key
                  (status 200)
                  (title "Hello hello!")
                  (doctype "<!DOCTYPE html>\n")
                  (content-type-params '((charset . "utf-8")))
                  (content-type 'text/html)
                  (extra-headers '())
                  (sxml (and body (templatize title body))))
  (values (build-response
           #:code status
           #:headers `((content-type
                        . (,content-type ,@content-type-params))
                       ,@extra-headers))
          (lambda (port)
            (if sxml
                (begin
                  (if doctype (display doctype port))
                  (sxml->xml sxml port))))))
```

Here we see the power of keyword arguments with default initializers. By the time the arguments are fully parsed, the `sxml` local variable will hold the templated SXML, ready for sending out to the client.

Also, instead of returning the body as a string, `respond` gives a procedure, which will be called by the web server to write out the response to the client.

Now, a simple example using this responder, which lays out the incoming headers in an HTML table.

```
(define (debug-page request body)
```

```
      (respond
       `((h1 "hello world!")
         (table
          (tr (th "header") (th "value"))
          ,@(map (lambda (pair)
                   `(tr (td (tt ,(with-output-to-string
                                   (lambda () (display (car pair))))))
                        (td (tt ,(with-output-to-string
                                   (lambda ()
                                     (write (cdr pair)))))))))
                 (request-headers request))))))

  (run-server debug-page)
```

Now if you visit any local address in your web browser, we actually see some HTML, finally.

7.3.10.4 Conclusion

Well, this is about as far as Guile's built-in web support goes, for now. There are many ways to make a web application, but hopefully by standardizing the most fundamental data types, users will be able to choose the approach that suits them best, while also being able to switch between implementations of the server. This is a relatively new part of Guile, so if you have feedback, let us know, and we can take it into account. Happy hacking on the web!

7.4 The (ice-9 getopt-long) Module

The `(ice-9 getopt-long)` facility is designed to help parse arguments that are passed to Guile programs on the command line, and is modelled after the C library's facility of the same name (see Section "Getopt" in *The GNU C Library Reference Manual*). For a more low-level interface to command-line argument parsing, See Section 7.5.25 [SRFI-37], page 628.

The `(ice-9 getopt-long)` module exports two procedures: `getopt-long` and `option-ref`.

- `getopt-long` takes a list of strings — the command line arguments — an *option specification*, and some optional keyword parameters. It parses the command line arguments according to the option specification and keyword parameters, and returns a data structure that encapsulates the results of the parsing.
- `option-ref` then takes the parsed data structure and a specific option's name, and returns information about that option in particular.

To make these procedures available to your Guile script, include the expression `(use-modules (ice-9 getopt-long))` somewhere near the top, before the first usage of `getopt-long` or `option-ref`.

7.4.1 A Short getopt-long Example

This section illustrates how `getopt-long` is used by presenting and dissecting a simple example. The first thing that we need is an *option specification* that tells `getopt-long`

how to parse the command line. This specification is an association list with the long option name as the key. Here is how such a specification might look:

```
(define option-spec
  '((version (single-char #\v) (value #f))
    (help    (single-char #\h) (value #f))))
```

This alist tells `getopt-long` that it should accept two long options, called *version* and *help*, and that these options can also be selected by the single-letter abbreviations *v* and *h*, respectively. The `(value #f)` clauses indicate that neither of the options accepts a value.

With this specification we can use `getopt-long` to parse a given command line:

```
(define options (getopt-long (command-line) option-spec))
```

After this call, `options` contains the parsed command line and is ready to be examined by `option-ref`. `option-ref` is called like this:

```
(option-ref options 'help #f)
```

It expects the parsed command line, a symbol indicating the option to examine, and a default value. The default value is returned if the option was not present in the command line, or if the option was present but without a value; otherwise the value from the command line is returned. Usually `option-ref` is called once for each possible option that a script supports.

The following example shows a main program which puts all this together to parse its command line and figure out what the user wanted.

```
(define (main args)
  (let* ((option-spec '((version (single-char #\v) (value #f))
                        (help    (single-char #\h) (value #f))))
         (options (getopt-long args option-spec))
         (help-wanted (option-ref options 'help #f))
         (version-wanted (option-ref options 'version #f)))
    (if (or version-wanted help-wanted)
        (begin
          (if version-wanted
              (display "getopt-long-example version 0.3\n"))
          (if help-wanted
              (display "\
getopt-long-example [options]
  -v, --version    Display version
  -h, --help       Display this help
")))
        (begin
          (display "Hello, World!") (newline)))))
```

7.4.2 How to Write an Option Specification

An option specification is an association list (see Section 6.6.20 [Association Lists], page 230) with one list element for each supported option. The key of each list element is a symbol that names the option, while the value is a list of option properties:

```
OPTION-SPEC ::=  '( (OPT-NAME1 (PROP-NAME PROP-VALUE) ...)
```

```
  (OPT-NAME2 (PROP-NAME PROP-VALUE) ...)
  (OPT-NAME3 (PROP-NAME PROP-VALUE) ...)
  ...
)
```

Each *opt-name* specifies the long option name for that option. For example, a list element with *opt-name* `background` specifies an option that can be specified on the command line using the long option `--background`. Further information about the option — whether it takes a value, whether it is required to be present in the command line, and so on — is specified by the option properties.

In the example of the preceding section, we already saw that a long option name can have a equivalent *short option* character. The equivalent short option character can be set for an option by specifying a `single-char` property in that option's property list. For example, a list element like `'(output (single-char #\o) ...)` specifies an option with long name `--output` that can also be specified by the equivalent short name `-o`.

The `value` property specifies whether an option requires or accepts a value. If the `value` property is set to `#t`, the option requires a value: `getopt-long` will signal an error if the option name is present without a corresponding value. If set to `#f`, the option does not take a value; in this case, a non-option word that follows the option name in the command line will be treated as a non-option argument. If set to the symbol `optional`, the option accepts a value but does not require one: a non-option word that follows the option name in the command line will be interpreted as that option's value. If the option name for an option with `'(value optional)` is immediately followed in the command line by *another* option name, the value for the first option is implicitly `#t`.

The `required?` property indicates whether an option is required to be present in the command line. If the `required?` property is set to `#t`, `getopt-long` will signal an error if the option is not specified.

Finally, the `predicate` property can be used to constrain the possible values of an option. If used, the `predicate` property should be set to a procedure that takes one argument — the proposed option value as a string — and returns either `#t` or `#f` according as the proposed value is or is not acceptable. If the predicate procedure returns `#f`, `getopt-long` will signal an error.

By default, options do not have single-character equivalents, are not required, and do not take values. Where the list element for an option includes a `value` property but no `predicate` property, the option values are unconstrained.

7.4.3 Expected Command Line Format

In order for `getopt-long` to correctly parse a command line, that command line must conform to a standard set of rules for how command line options are specified. This section explains what those rules are.

`getopt-long` splits a given command line into several pieces. All elements of the argument list are classified to be either options or normal arguments. Options consist of two dashes and an option name (so-called *long* options), or of one dash followed by a single letter (*short* options).

Options can behave as switches, when they are given without a value, or they can be used to pass a value to the program. The value for an option may be specified using an equals

sign, or else is simply the next word in the command line, so the following two invocations are equivalent:

```
$ ./foo.scm --output=bar.txt
$ ./foo.scm --output bar.txt
```

Short options can be used instead of their long equivalents and can be grouped together after a single dash. For example, the following commands are equivalent.

```
$ ./foo.scm --version --help
$ ./foo.scm -v --help
$ ./foo.scm -vh
```

If an option requires a value, it can only be grouped together with other short options if it is the last option in the group; the value is the next argument. So, for example, with the following option specification —

```
((apples    (single-char #\a))
 (blimps    (single-char #\b) (value #t))
 (catalexis (single-char #\c) (value #t)))
```

— the following command lines would all be acceptable:

```
$ ./foo.scm -a -b bang -c couth
$ ./foo.scm -ab bang -c couth
$ ./foo.scm -ac couth -b bang
```

But the next command line is an error, because `-b` is not the last option in its combination, and because a group of short options cannot include two options that both require values:

```
$ ./foo.scm -abc couth bang
```

If an option's value is optional, `getopt-long` decides whether the option has a value by looking at what follows it in the argument list. If the next element is a string, and it does not appear to be an option itself, then that string is the option's value.

If the option `--` appears in the argument list, argument parsing stops there and subsequent arguments are returned as ordinary arguments, even if they resemble options. So, with the command line

```
$ ./foo.scm --apples "Granny Smith" -- --blimp Goodyear
```

`getopt-long` will recognize the `--apples` option as having the value `"Granny Smith"`, but will not treat `--blimp` as an option. The strings `--blimp` and `Goodyear` will be returned as ordinary argument strings.

7.4.4 Reference Documentation for `getopt-long`

`getopt-long` *args grammar* [*#:stop-at-first-non-option #t*] [Scheme Procedure]

Parse the command line given in *args* (which must be a list of strings) according to the option specification *grammar*.

The *grammar* argument is expected to be a list of this form:

`((`*option* `(`*property value*`) ...) ...)`

where each *option* is a symbol denoting the long option, but without the two leading dashes (e.g. `version` if the option is called `--version`).

For each option, there may be list of arbitrarily many property/value pairs. The order of the pairs is not important, but every property may only appear once in the property list. The following table lists the possible properties:

`(single-char` *`char`*`)`
: Accept `-`*`char`* as a single-character equivalent to `--`*`option`*. This is how to specify traditional Unix-style flags.

`(required?` *`bool`*`)`
: If *bool* is true, the option is required. `getopt-long` will raise an error if it is not found in *args*.

`(value` *`bool`*`)`
: If *bool* is `#t`, the option accepts a value; if it is `#f`, it does not; and if it is the symbol `optional`, the option may appear in *args* with or without a value.

`(predicate` *`func`*`)`
: If the option accepts a value (i.e. you specified `(value #t)` for this option), then `getopt-long` will apply *func* to the value, and throw an exception if it returns `#f`. *func* should be a procedure which accepts a string and returns a boolean value; you may need to use quasiquotes to get it into *grammar*.

The `#:stop-at-first-non-option` keyword, if specified with any true value, tells `getopt-long` to stop when it gets to the first non-option in the command line. That is, at the first word which is neither an option itself, nor the value of an option. Everything in the command line from that word onwards will be returned as non-option arguments.

`getopt-long`'s *args* parameter is expected to be a list of strings like the one returned by `command-line`, with the first element being the name of the command. Therefore `getopt-long` ignores the first element in *args* and starts argument interpretation with the second element.

`getopt-long` signals an error if any of the following conditions hold.

- The option grammar has an invalid syntax.
- One of the options in the argument list was not specified by the grammar.
- A required option is omitted.
- An option which requires an argument did not get one.
- An option that doesn't accept an argument does get one (this can only happen using the long option `--opt=`*`value`* syntax).
- An option predicate fails.

`#:stop-at-first-non-option` is useful for command line invocations like `guild [--help | --version] [script [script-options]]` and `cvs [general-options] command [command-options]`, where there are options at two levels: some generic and understood by the outer command, and some that are specific to the particular script or command being invoked. To use `getopt-long` in such cases, you would call it twice: firstly with `#:stop-at-first-non-option #t`, so as to parse any generic options and identify

the wanted script or sub-command; secondly, and after trimming off the initial generic command words, with a script- or sub-command-specific option grammar, so as to process those specific options.

7.4.5 Reference Documentation for `option-ref`

`option-ref` *options key default* [Scheme Procedure]
: Search *options* for a command line option named *key* and return its value, if found. If the option has no value, but was given, return `#t`. If the option was not given, return *default*. *options* must be the result of a call to `getopt-long`.

`option-ref` always succeeds, either by returning the requested option value from the command line, or the default value.

The special key `'()` can be used to get a list of all non-option arguments.

7.5 SRFI Support Modules

SRFI is an acronym for Scheme Request For Implementation. The SRFI documents define a lot of syntactic and procedure extensions to standard Scheme as defined in R5RS.

Guile has support for a number of SRFIs. This chapter gives an overview over the available SRFIs and some usage hints. For complete documentation, design rationales and further examples, we advise you to get the relevant SRFI documents from the SRFI home page `http://srfi.schemers.org/`.

7.5.1 About SRFI Usage

SRFI support in Guile is currently implemented partly in the core library, and partly as add-on modules. That means that some SRFIs are automatically available when the interpreter is started, whereas the other SRFIs require you to use the appropriate support module explicitly.

There are several reasons for this inconsistency. First, the feature checking syntactic form `cond-expand` (see Section 7.5.2 [SRFI-0], page 583) must be available immediately, because it must be there when the user wants to check for the Scheme implementation, that is, before she can know that it is safe to use `use-modules` to load SRFI support modules. The second reason is that some features defined in SRFIs had been implemented in Guile before the developers started to add SRFI implementations as modules (for example SRFI-13 (see Section 7.5.11 [SRFI-13], page 609)). In the future, it is possible that SRFIs in the core library might be factored out into separate modules, requiring explicit module loading when they are needed. So you should be prepared to have to use `use-modules` someday in the future to access SRFI-13 bindings. If you want, you can do that already. We have included the module `(srfi srfi-13)` in the distribution, which currently does nothing, but ensures that you can write future-safe code.

Generally, support for a specific SRFI is made available by using modules named `(srfi srfi-`*number*`)`, where *number* is the number of the SRFI needed. Another possibility is to use the command line option `--use-srfi`, which will load the necessary modules automatically (see Section 4.2 [Invoking Guile], page 35).

7.5.2 SRFI-0 - cond-expand

This SRFI lets a portable Scheme program test for the presence of certain features, and adapt itself by using different blocks of code, or fail if the necessary features are not available. There's no module to load, this is in the Guile core.

A program designed only for Guile will generally not need this mechanism, such a program can of course directly use the various documented parts of Guile.

`cond-expand` (*feature body...*) ... [syntax]

Expand to the *body* of the first clause whose *feature* specification is satisfied. It is an error if no *feature* is satisfied.

Features are symbols such as `srfi-1`, and a feature specification can use `and`, `or` and `not` forms to test combinations. The last clause can be an `else`, to be used if no other passes.

For example, define a private version of `alist-cons` if SRFI-1 is not available.

```
(cond-expand (srfi-1
              )
             (else
              (define (alist-cons key val alist)
                (cons (cons key val) alist))))
```

Or demand a certain set of SRFIs (list operations, string ports, `receive` and string operations), failing if they're not available.

```
(cond-expand ((and srfi-1 srfi-6 srfi-8 srfi-13)
              ))
```

The Guile core has the following features,

```
guile
guile-2   ;; starting from Guile 2.x
guile-2.2 ;; starting from Guile 2.2
guile-3   ;; starting from Guile 3.x
guile-3.0 ;; starting from Guile 3.0
r5rs
r6rs
r7rs
exact-closed ieee-float full-unicode ratios ;; R7RS features
srfi-0
srfi-4
srfi-6
srfi-13
srfi-14
srfi-16
srfi-23
srfi-30
srfi-39
srfi-46
srfi-55
srfi-61
```

```
srfi-62
srfi-87
srfi-105
```

Other SRFI feature symbols are defined once their code has been loaded with `use-modules`, since only then are their bindings available.

The '`--use-srfi`' command line option (see Section 4.2 [Invoking Guile], page 35) is a good way to load SRFIs to satisfy `cond-expand` when running a portable program.

Testing the `guile` feature allows a program to adapt itself to the Guile module system, but still run on other Scheme systems. For example the following demands SRFI-8 (`receive`), but also knows how to load it with the Guile mechanism.

```
(cond-expand (srfi-8
              )
             (guile
              (use-modules (srfi srfi-8))))
```

Likewise, testing the `guile-2` feature allows code to be portable between Guile 2.*x* and previous versions of Guile. For instance, it makes it possible to write code that accounts for Guile 2.*x*'s compiler, yet be correctly interpreted on 1.8 and earlier versions:

```
(cond-expand (guile-2 (eval-when (compile)
                        ;; This must be evaluated at compile time.
                        (fluid-set! current-reader my-reader)))
             (guile
                      ;; Earlier versions of Guile do not have a
                      ;; separate compilation phase.
                      (fluid-set! current-reader my-reader)))
```

It should be noted that `cond-expand` is separate from the `*features*` mechanism (see Section 6.23.2 [Feature Tracking], page 458), feature symbols in one are unrelated to those in the other.

7.5.3 SRFI-1 - List library

The list library defined in SRFI-1 contains a lot of useful list processing procedures for construction, examining, destructuring and manipulating lists and pairs.

Since SRFI-1 also defines some procedures which are already contained in R5RS and thus are supported by the Guile core library, some list and pair procedures which appear in the SRFI-1 document may not appear in this section. So when looking for a particular list/pair processing procedure, you should also have a look at the sections Section 6.6.9 [Lists], page 181, and Section 6.6.8 [Pairs], page 178.

7.5.3.1 Constructors

New lists can be constructed by calling one of the following procedures.

`xcons` *d a* [Scheme Procedure]

Like `cons`, but with interchanged arguments. Useful mostly when passed to higher-order procedures.

`list-tabulate` *n init-proc* [Scheme Procedure]
Return an *n*-element list, where each list element is produced by applying the procedure *init-proc* to the corresponding list index. The order in which *init-proc* is applied to the indices is not specified.

`list-copy` *lst* [Scheme Procedure]
Return a new list containing the elements of the list *lst*.

This function differs from the core `list-copy` (see Section 6.6.9.3 [List Constructors], page 182) in accepting improper lists too. And if *lst* is not a pair at all then it's treated as the final tail of an improper list and simply returned.

`circular-list` *elt1 elt2 ...* [Scheme Procedure]
Return a circular list containing the given arguments *elt1 elt2 ...*.

`iota` *count* [*start step*] [Scheme Procedure]
Return a list containing *count* numbers, starting from *start* and adding *step* each time. The default *start* is 0, the default *step* is 1. For example,

```
(iota 6)        ⇒ (0 1 2 3 4 5)
(iota 4 2.5 -2) ⇒ (2.5 0.5 -1.5 -3.5)
```

This function takes its name from the corresponding primitive in the APL language.

7.5.3.2 Predicates

The procedures in this section test specific properties of lists.

`proper-list?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a proper list, or `#f` otherwise. This is the same as the core `list?` (see Section 6.6.9.2 [List Predicates], page 181).

A proper list is a list which ends with the empty list `()` in the usual way. The empty list `()` itself is a proper list too.

```
(proper-list? '(1 2 3))  ⇒ #t
(proper-list? '())       ⇒ #t
```

`circular-list?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a circular list, or `#f` otherwise.

A circular list is a list where at some point the `cdr` refers back to a previous pair in the list (either the start or some later point), so that following the `cdr`s takes you around in a circle, with no end.

```
(define x (list 1 2 3 4))
(set-cdr! (last-pair x) (cddr x))
x ⇒ (1 2 3 4 3 4 3 4 ...)
(circular-list? x)  ⇒ #t
```

`dotted-list?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a dotted list, or `#f` otherwise.

A dotted list is a list where the `cdr` of the last pair is not the empty list `()`. Any non-pair *obj* is also considered a dotted list, with length zero.

```
(dotted-list? '(1 2 . 3))  ⇒ #t
(dotted-list? 99)          ⇒ #t
```

It will be noted that any Scheme object passes exactly one of the above three tests `proper-list?`, `circular-list?` and `dotted-list?`. Non-lists are `dotted-list?`, finite lists are either `proper-list?` or `dotted-list?`, and infinite lists are `circular-list?`.

`null-list?` *lst* [Scheme Procedure]
: Return `#t` if *lst* is the empty list `()`, `#f` otherwise. If something else than a proper or circular list is passed as *lst*, an error is signalled. This procedure is recommended for checking for the end of a list in contexts where dotted lists are not allowed.

`not-pair?` *obj* [Scheme Procedure]
: Return `#t` is *obj* is not a pair, `#f` otherwise. This is shorthand notation `(not (pair? obj))` and is supposed to be used for end-of-list checking in contexts where dotted lists are allowed.

`list=` *elt= list1 ...* [Scheme Procedure]
: Return `#t` if all argument lists are equal, `#f` otherwise. List equality is determined by testing whether all lists have the same length and the corresponding elements are equal in the sense of the equality predicate *elt=*. If no or only one list is given, `#t` is returned.

7.5.3.3 Selectors

`first` *pair* [Scheme Procedure]
`second` *pair* [Scheme Procedure]
`third` *pair* [Scheme Procedure]
`fourth` *pair* [Scheme Procedure]
`fifth` *pair* [Scheme Procedure]
`sixth` *pair* [Scheme Procedure]
`seventh` *pair* [Scheme Procedure]
`eighth` *pair* [Scheme Procedure]
`ninth` *pair* [Scheme Procedure]
`tenth` *pair* [Scheme Procedure]
: These are synonyms for `car`, `cadr`, `caddr`, ...

`car+cdr` *pair* [Scheme Procedure]
: Return two values, the CAR and the CDR of *pair*.

`take` *lst i* [Scheme Procedure]
`take!` *lst i* [Scheme Procedure]
: Return a list containing the first *i* elements of *lst*.

  `take!` may modify the structure of the argument list *lst* in order to produce the result.

`drop` *lst i* [Scheme Procedure]
: Return a list containing all but the first *i* elements of *lst*.

`take-right` *lst i* [Scheme Procedure]
: Return a list containing the *i* last elements of *lst*. The return shares a common tail with *lst*.

`drop-right` *lst i* [Scheme Procedure]
`drop-right!` *lst i* [Scheme Procedure]

Return a list containing all but the *i* last elements of *lst*.

`drop-right` always returns a new list, even when *i* is zero. `drop-right!` may modify the structure of the argument list *lst* in order to produce the result.

`split-at` *lst i* [Scheme Procedure]
`split-at!` *lst i* [Scheme Procedure]

Return two values, a list containing the first *i* elements of the list *lst* and a list containing the remaining elements.

`split-at!` may modify the structure of the argument list *lst* in order to produce the result.

`last` *lst* [Scheme Procedure]

Return the last element of the non-empty, finite list *lst*.

7.5.3.4 Length, Append, Concatenate, etc.

`length+` *lst* [Scheme Procedure]

Return the length of the argument list *lst*. When *lst* is a circular list, `#f` is returned.

`concatenate` *list-of-lists* [Scheme Procedure]
`concatenate!` *list-of-lists* [Scheme Procedure]

Construct a list by appending all lists in *list-of-lists*.

`concatenate!` may modify the structure of the given lists in order to produce the result.

`concatenate` is the same as `(apply append` *`list-of-lists`*`)`. It exists because some Scheme implementations have a limit on the number of arguments a function takes, which the `apply` might exceed. In Guile there is no such limit.

`append-reverse` *rev-head tail* [Scheme Procedure]
`append-reverse!` *rev-head tail* [Scheme Procedure]

Reverse *rev-head*, append *tail* to it, and return the result. This is equivalent to `(append (reverse` *`rev-head`*`)` *`tail`*`)`, but its implementation is more efficient.

```
(append-reverse '(1 2 3) '(4 5 6)) ⇒ (3 2 1 4 5 6)
```

`append-reverse!` may modify *rev-head* in order to produce the result.

`zip` *lst1 lst2 ...* [Scheme Procedure]

Return a list as long as the shortest of the argument lists, where each element is a list. The first list contains the first elements of the argument lists, the second list contains the second elements, and so on.

`unzip1` *lst* [Scheme Procedure]
`unzip2` *lst* [Scheme Procedure]
`unzip3` *lst* [Scheme Procedure]
`unzip4` *lst* [Scheme Procedure]
`unzip5` *lst* [Scheme Procedure]

`unzip1` takes a list of lists, and returns a list containing the first elements of each list, `unzip2` returns two lists, the first containing the first elements of each lists and the second containing the second elements of each lists, and so on.

`count` *pred lst1 lst2 ...* [Scheme Procedure]

Return a count of the number of times *pred* returns true when called on elements from the given lists.

pred is called with *N* parameters (*pred elem1 ... elemN*), each element being from the corresponding list. The first call is with the first element of each list, the second with the second element from each, and so on.

Counting stops when the end of the shortest list is reached. At least one list must be non-circular.

7.5.3.5 Fold, Unfold & Map

`fold` *proc init lst1 lst2 ...* [Scheme Procedure]
`fold-right` *proc init lst1 lst2 ...* [Scheme Procedure]

Apply *proc* to the elements of *lst1 lst2 ...* to build a result, and return that result.

Each *proc* call is (*proc elem1 elem2 ... previous*), where *elem1* is from *lst1*, *elem2* is from *lst2*, and so on. *previous* is the return from the previous call to *proc*, or the given *init* for the first call. If any list is empty, just *init* is returned.

`fold` works through the list elements from first to last. The following shows a list reversal and the calls it makes,

```
(fold cons '() '(1 2 3))

(cons 1 '())
(cons 2 '(1))
(cons 3 '(2 1)
⇒ (3 2 1)
```

`fold-right` works through the list elements from last to first, ie. from the right. So for example the following finds the longest string, and the last among equal longest,

```
(fold-right (lambda (str prev)
              (if (> (string-length str) (string-length prev))
                  str
                  prev))
            ""
            '("x" "abc" "xyz" "jk"))
⇒ "xyz"
```

If *lst1 lst2 ...* have different lengths, `fold` stops when the end of the shortest is reached; `fold-right` commences at the last element of the shortest. Ie. elements past the length of the shortest are ignored in the other *lst*s. At least one *lst* must be non-circular.

`fold` should be preferred over `fold-right` if the order of processing doesn't matter, or can be arranged either way, since `fold` is a little more efficient.

The way `fold` builds a result from iterating is quite general, it can do more than other iterations like say `map` or `filter`. The following for example removes adjacent duplicate elements from a list,

```
(define (delete-adjacent-duplicates lst)
  (fold-right (lambda (elem ret)
```

```
                    (if (equal? elem (first ret))
                        ret
                        (cons elem ret)))
                (list (last lst))
                lst))
(delete-adjacent-duplicates '(1 2 3 3 4 4 4 5))
⇒ (1 2 3 4 5)
```

Clearly the same sort of thing can be done with a `for-each` and a variable in which to build the result, but a self-contained *proc* can be re-used in multiple contexts, where a `for-each` would have to be written out each time.

`pair-fold` *proc init lst1 lst2 . . .* [Scheme Procedure]
`pair-fold-right` *proc init lst1 lst2 . . .* [Scheme Procedure]

The same as `fold` and `fold-right`, but apply *proc* to the pairs of the lists instead of the list elements.

`reduce` *proc default lst* [Scheme Procedure]
`reduce-right` *proc default lst* [Scheme Procedure]

`reduce` is a variant of `fold`, where the first call to *proc* is on two elements from *lst*, rather than one element and a given initial value.

If *lst* is empty, `reduce` returns *default* (this is the only use for *default*). If *lst* has just one element then that's the return value. Otherwise *proc* is called on the elements of *lst*.

Each *proc* call is `(`*proc elem previous*`)`, where *elem* is from *lst* (the second and subsequent elements of *lst*), and *previous* is the return from the previous call to *proc*. The first element of *lst* is the *previous* for the first call to *proc*.

For example, the following adds a list of numbers, the calls made to `+` are shown. (Of course `+` accepts multiple arguments and can add a list directly, with `apply`.)

```
(reduce + 0 '(5 6 7)) ⇒ 18

(+ 6 5)  ⇒ 11
(+ 7 11) ⇒ 18
```

`reduce` can be used instead of `fold` where the *init* value is an "identity", meaning a value which under *proc* doesn't change the result, in this case 0 is an identity since `(+ 5 0)` is just 5. `reduce` avoids that unnecessary call.

`reduce-right` is a similar variation on `fold-right`, working from the end (ie. the right) of *lst*. The last element of *lst* is the *previous* for the first call to *proc*, and the *elem* values go from the second last.

`reduce` should be preferred over `reduce-right` if the order of processing doesn't matter, or can be arranged either way, since `reduce` is a little more efficient.

`unfold` *p f g seed* [*tail-gen*] [Scheme Procedure]

`unfold` is defined as follows:

```
(unfold p f g seed) =
   (if (p seed) (tail-gen seed)
       (cons (f seed)
```

```
(unfold p f g (g seed))))
```

p Determines when to stop unfolding.

f Maps each seed value to the corresponding list element.

g Maps each seed value to next seed value.

seed The state value for the unfold.

tail-gen Creates the tail of the list; defaults to `(lambda (x) '())`.

g produces a series of seed values, which are mapped to list elements by *f*. These elements are put into a list in left-to-right order, and *p* tells when to stop unfolding.

`unfold-right` *p f g seed* [*tail*] [Scheme Procedure]

Construct a list with the following loop.

```
(let lp ((seed seed) (lis tail))
   (if (p seed) lis
       (lp (g seed)
           (cons (f seed) lis))))
```

p Determines when to stop unfolding.

f Maps each seed value to the corresponding list element.

g Maps each seed value to next seed value.

seed The state value for the unfold.

tail The tail of the list; defaults to `'()`.

`map` *f lst1 lst2 ...* [Scheme Procedure]

Map the procedure over the list(s) *lst1*, *lst2*, ... and return a list containing the results of the procedure applications. This procedure is extended with respect to R5RS, because the argument lists may have different lengths. The result list will have the same length as the shortest argument lists. The order in which *f* will be applied to the list element(s) is not specified.

`for-each` *f lst1 lst2 ...* [Scheme Procedure]

Apply the procedure *f* to each pair of corresponding elements of the list(s) *lst1*, *lst2*, ... The return value is not specified. This procedure is extended with respect to R5RS, because the argument lists may have different lengths. The shortest argument list determines the number of times *f* is called. *f* will be applied to the list elements in left-to-right order.

`append-map` *f lst1 lst2 ...* [Scheme Procedure]
`append-map!` *f lst1 lst2 ...* [Scheme Procedure]

Equivalent to

```
(apply append (map f clist1 clist2 ...))
```

and

```
(apply append! (map f clist1 clist2 ...))
```

Map *f* over the elements of the lists, just as in the `map` function. However, the results of the applications are appended together to make the final result. `append-map` uses `append` to append the results together; `append-map!` uses `append!`.

The dynamic order in which the various applications of *f* are made is not specified.

`map!` *f lst1 lst2 ...* [Scheme Procedure]
Linear-update variant of `map` – `map!` is allowed, but not required, to alter the cons cells of *lst1* to construct the result list.

The dynamic order in which the various applications of *f* are made is not specified. In the n-ary case, *lst2*, *lst3*, ... must have at least as many elements as *lst1*.

`pair-for-each` *f lst1 lst2 ...* [Scheme Procedure]
Like `for-each`, but applies the procedure *f* to the pairs from which the argument lists are constructed, instead of the list elements. The return value is not specified.

`filter-map` *f lst1 lst2 ...* [Scheme Procedure]
Like `map`, but only results from the applications of *f* which are true are saved in the result list.

7.5.3.6 Filtering and Partitioning

Filtering means to collect all elements from a list which satisfy a specific condition. Partitioning a list means to make two groups of list elements, one which contains the elements satisfying a condition, and the other for the elements which don't.

The `filter` and `filter!` functions are implemented in the Guile core, See Section 6.6.9.6 [List Modification], page 184.

`partition` *pred lst* [Scheme Procedure]
`partition!` *pred lst* [Scheme Procedure]
Split *lst* into those elements which do and don't satisfy the predicate *pred*.

The return is two values (see Section 6.13.7 [Multiple Values], page 309), the first being a list of all elements from *lst* which satisfy *pred*, the second a list of those which do not.

The elements in the result lists are in the same order as in *lst* but the order in which the calls `(`*pred* `elem)` are made on the list elements is unspecified.

`partition` does not change *lst*, but one of the returned lists may share a tail with it. `partition!` may modify *lst* to construct its return.

`remove` *pred lst* [Scheme Procedure]
`remove!` *pred lst* [Scheme Procedure]
Return a list containing all elements from *lst* which do not satisfy the predicate *pred*. The elements in the result list have the same order as in *lst*. The order in which *pred* is applied to the list elements is not specified.

`remove!` is allowed, but not required to modify the structure of the input list.

7.5.3.7 Searching

The procedures for searching elements in lists either accept a predicate or a comparison object for determining which elements are to be searched.

`find` *pred lst* [Scheme Procedure]
Return the first element of *lst* that satisfies the predicate *pred* and `#f` if no such element is found.

`find-tail` *pred lst* [Scheme Procedure]
Return the first pair of *lst* whose CAR satisfies the predicate *pred* and `#f` if no such element is found.

`take-while` *pred lst* [Scheme Procedure]
`take-while!` *pred lst* [Scheme Procedure]
Return the longest initial prefix of *lst* whose elements all satisfy the predicate *pred*.

`take-while!` is allowed, but not required to modify the input list while producing the result.

`drop-while` *pred lst* [Scheme Procedure]
Drop the longest initial prefix of *lst* whose elements all satisfy the predicate *pred*.

`span` *pred lst* [Scheme Procedure]
`span!` *pred lst* [Scheme Procedure]
`break` *pred lst* [Scheme Procedure]
`break!` *pred lst* [Scheme Procedure]
`span` splits the list *lst* into the longest initial prefix whose elements all satisfy the predicate *pred*, and the remaining tail. `break` inverts the sense of the predicate.

`span!` and `break!` are allowed, but not required to modify the structure of the input list *lst* in order to produce the result.

Note that the name `break` conflicts with the `break` binding established by `while` (see Section 6.13.4 [while do], page 301). Applications wanting to use `break` from within a `while` loop will need to make a new define under a different name.

`any` *pred lst1 lst2 ...* [Scheme Procedure]
Test whether any set of elements from *lst1 lst2 ...* satisfies *pred*. If so, the return value is the return value from the successful *pred* call, or if not, the return value is `#f`.

If there are n list arguments, then *pred* must be a predicate taking n arguments. Each *pred* call is `(`*`pred elem1 elem2 ...`*` )` taking an element from each *lst*. The calls are made successively for the first, second, etc. elements of the lists, stopping when *pred* returns non-`#f`, or when the end of the shortest list is reached.

The *pred* call on the last set of elements (i.e., when the end of the shortest list has been reached), if that point is reached, is a tail call.

`every` *pred lst1 lst2 ...* [Scheme Procedure]
Test whether every set of elements from *lst1 lst2 ...* satisfies *pred*. If so, the return value is the return from the final *pred* call, or if not, the return value is `#f`.

If there are n list arguments, then *pred* must be a predicate taking n arguments. Each *pred* call is `(pred elem1 elem2 ...)` taking an element from each *lst*. The calls are made successively for the first, second, etc. elements of the lists, stopping if *pred* returns `#f`, or when the end of any of the lists is reached.

The *pred* call on the last set of elements (i.e., when the end of the shortest list has been reached) is a tail call.

If one of *lst1 lst2* ...is empty then no calls to *pred* are made, and the return value is `#t`.

`list-index` *pred lst1 lst2 ...* [Scheme Procedure]

Return the index of the first set of elements, one from each of *lst1 lst2* ..., which satisfies *pred*.

pred is called as `(elem1 elem2 ...)`. Searching stops when the end of the shortest *lst* is reached. The return index starts from 0 for the first set of elements. If no set of elements pass, then the return value is `#f`.

```
(list-index odd? '(2 4 6 9))      ⇒ 3
(list-index = '(1 2 3) '(3 1 2))  ⇒ #f
```

`member` *x lst* [=] [Scheme Procedure]

Return the first sublist of *lst* whose CAR is equal to *x*. If *x* does not appear in *lst*, return `#f`.

Equality is determined by `equal?`, or by the equality predicate = if given. = is called `(= x elem)`, ie. with the given *x* first, so for example to find the first element greater than 5,

```
(member 5 '(3 5 1 7 2 9) <) ⇒ (7 2 9)
```

This version of `member` extends the core `member` (see Section 6.6.9.7 [List Searching], page 185) by accepting an equality predicate.

7.5.3.8 Deleting

`delete` *x lst* [=] [Scheme Procedure]
`delete!` *x lst* [=] [Scheme Procedure]

Return a list containing the elements of *lst* but with those equal to *x* deleted. The returned elements will be in the same order as they were in *lst*.

Equality is determined by the = predicate, or `equal?` if not given. An equality call is made just once for each element, but the order in which the calls are made on the elements is unspecified.

The equality calls are always `(= x elem)`, ie. the given *x* is first. This means for instance elements greater than 5 can be deleted with `(delete 5 lst <)`.

`delete` does not modify *lst*, but the return might share a common tail with *lst*. `delete!` may modify the structure of *lst* to construct its return.

These functions extend the core `delete` and `delete!` (see Section 6.6.9.6 [List Modification], page 184) in accepting an equality predicate. See also `lset-difference` (see Section 7.5.3.10 [SRFI-1 Set Operations], page 595) for deleting multiple elements from a list.

`delete-duplicates` *lst* [=] [Scheme Procedure]
`delete-duplicates!` *lst* [=] [Scheme Procedure]

Return a list containing the elements of *lst* but without duplicates.

When elements are equal, only the first in *lst* is retained. Equal elements can be anywhere in *lst*, they don't have to be adjacent. The returned list will have the retained elements in the same order as they were in *lst*.

Equality is determined by the = predicate, or `equal?` if not given. Calls `(= x y)` are made with element *x* being before *y* in *lst*. A call is made at most once for each combination, but the sequence of the calls across the elements is unspecified.

`delete-duplicates` does not modify *lst*, but the return might share a common tail with *lst*. `delete-duplicates!` may modify the structure of *lst* to construct its return.

In the worst case, this is an $O(N^2)$ algorithm because it must check each element against all those preceding it. For long lists it is more efficient to sort and then compare only adjacent elements.

7.5.3.9 Association Lists

Association lists are described in detail in section Section 6.6.20 [Association Lists], page 230. The present section only documents the additional procedures for dealing with association lists defined by SRFI-1.

`assoc` *key alist* [=] [Scheme Procedure]

Return the pair from *alist* which matches *key*. This extends the core `assoc` (see Section 6.6.20.3 [Retrieving Alist Entries], page 232) by taking an optional = comparison procedure.

The default comparison is `equal?`. If an = parameter is given it's called `(= key alistcar)`, i.e. the given target *key* is the first argument, and a `car` from *alist* is second.

For example a case-insensitive string lookup,

```
(assoc "yy" '(("XX" . 1) ("YY" . 2)) string-ci=?)
⇒ ("YY" . 2)
```

`alist-cons` *key datum alist* [Scheme Procedure]

Cons a new association *key* and *datum* onto *alist* and return the result. This is equivalent to

```
(cons (cons key datum) alist)
```

`acons` (see Section 6.6.20.2 [Adding or Setting Alist Entries], page 230) in the Guile core does the same thing.

`alist-copy` *alist* [Scheme Procedure]

Return a newly allocated copy of *alist*, that means that the spine of the list as well as the pairs are copied.

`alist-delete` *key alist* [=] [Scheme Procedure]
`alist-delete!` *key alist* [=] [Scheme Procedure]

Return a list containing the elements of *alist* but with those elements whose keys are equal to *key* deleted. The returned elements will be in the same order as they were in *alist*.

Equality is determined by the = predicate, or `equal?` if not given. The order in which elements are tested is unspecified, but each equality call is made `(= key alistkey)`, i.e. the given *key* parameter is first and the key from *alist* second. This means for instance all associations with a key greater than 5 can be removed with `(alist-delete 5 alist <)`.

`alist-delete` does not modify *alist*, but the return might share a common tail with *alist*. `alist-delete!` may modify the list structure of *alist* to construct its return.

7.5.3.10 Set Operations on Lists

Lists can be used to represent sets of objects. The procedures in this section operate on such lists as sets.

Note that lists are not an efficient way to implement large sets. The procedures here typically take time $m \times n$ when operating on m and n element lists. Other data structures like trees, bitsets (see Section 6.6.11 [Bit Vectors], page 191) or hash tables (see Section 6.6.22 [Hash Tables], page 238) are faster.

All these procedures take an equality predicate as the first argument. This predicate is used for testing the objects in the list sets for sameness. This predicate must be consistent with `eq?` (see Section 6.11.1 [Equality], page 283) in the sense that if two list elements are `eq?` then they must also be equal under the predicate. This simply means a given object must be equal to itself.

`lset<=` = *list* ... [Scheme Procedure]

Return `#t` if each list is a subset of the one following it. I.e., *list1* is a subset of *list2*, *list2* is a subset of *list3*, etc., for as many lists as given. If only one list or no lists are given, the return value is `#t`.

A list x is a subset of y if each element of x is equal to some element in y. Elements are compared using the given = procedure, called as `(= xelem yelem)`.

```
(lset<= eq?)                        ⇒ #t
(lset<= eqv? '(1 2 3) '(1))         ⇒ #f
(lset<= eqv? '(1 3 2) '(4 3 1 2)) ⇒ #t
```

`lset=` = *list* ... [Scheme Procedure]

Return `#t` if all argument lists are set-equal. *list1* is compared to *list2*, *list2* to *list3*, etc., for as many lists as given. If only one list or no lists are given, the return value is `#t`.

Two lists x and y are set-equal if each element of x is equal to some element of y and conversely each element of y is equal to some element of x. The order of the elements in the lists doesn't matter. Element equality is determined with the given = procedure, called as `(= xelem yelem)`, but exactly which calls are made is unspecified.

```
(lset= eq?)                          ⇒ #t
(lset= eqv? '(1 2 3) '(3 2 1))     ⇒ #t
(lset= string-ci=? '("a" "A" "b") '("B" "b" "a")) ⇒ #t
```

`lset-adjoin` = *list elem* ... [Scheme Procedure]

Add to *list* any of the given *elem*s not already in the list. *elem*s are `cons`ed onto the start of *list* (so the return value shares a common tail with *list*), but the order that the *elem*s are added is unspecified.

The given = procedure is used for comparing elements, called as `(= listelem elem)`, i.e., the second argument is one of the given *elem* parameters.

```
(lset-adjoin eqv? '(1 2 3) 4 1 5) ⇒ (5 4 1 2 3)
```

`lset-union` = *list ...* [Scheme Procedure]
`lset-union!` = *list ...* [Scheme Procedure]

Return the union of the argument list sets. The result is built by taking the union of *list1* and *list2*, then the union of that with *list3*, etc., for as many lists as given. For one list argument that list itself is the result, for no list arguments the result is the empty list.

The union of two lists x and y is formed as follows. If x is empty then the result is y. Otherwise start with x as the result and consider each y element (from first to last). A y element not equal to something already in the result is `cons`ed onto the result.

The given = procedure is used for comparing elements, called as `(= relem yelem)`. The first argument is from the result accumulated so far, and the second is from the list being union-ed in. But exactly which calls are made is otherwise unspecified.

Notice that duplicate elements in *list1* (or the first non-empty list) are preserved, but that repeated elements in subsequent lists are only added once.

```
(lset-union eqv?)                          ⇒ ()
(lset-union eqv? '(1 2 3))                 ⇒ (1 2 3)
(lset-union eqv? '(1 2 1 3) '(2 4 5) '(5)) ⇒ (5 4 1 2 1 3)
```

`lset-union` doesn't change the given lists but the result may share a tail with the first non-empty list. `lset-union!` can modify all of the given lists to form the result.

`lset-intersection` = *list1 list2 ...* [Scheme Procedure]
`lset-intersection!` = *list1 list2 ...* [Scheme Procedure]

Return the intersection of *list1* with the other argument lists, meaning those elements of *list1* which are also in all of *list2* etc. For one list argument, just that list is returned.

The test for an element of *list1* to be in the return is simply that it's equal to some element in each of *list2* etc. Notice this means an element appearing twice in *list1* but only once in each of *list2* etc will go into the return twice. The return has its elements in the same order as they were in *list1*.

The given = procedure is used for comparing elements, called as `(= elem1 elemN)`. The first argument is from *list1* and the second is from one of the subsequent lists. But exactly which calls are made and in what order is unspecified.

```
(lset-intersection eqv? '(x y))                       ⇒ (x y)
(lset-intersection eqv? '(1 2 3) '(4 3 2))            ⇒ (2 3)
(lset-intersection eqv? '(1 1 2 2) '(1 2) '(2 1) '(2)) ⇒ (2 2)
```

The return from `lset-intersection` may share a tail with *list1*. `lset-intersection!` may modify *list1* to form its result.

`lset-difference` = *list1 list2 ...* [Scheme Procedure]
`lset-difference!` = *list1 list2 ...* [Scheme Procedure]

Return *list1* with any elements in *list2*, *list3* etc removed (ie. subtracted). For one list argument, just that list is returned.

The given = procedure is used for comparing elements, called as `(= elem1 elemN)`. The first argument is from *list1* and the second from one of the subsequent lists. But exactly which calls are made and in what order is unspecified.

```
(lset-difference eqv? '(x y))             ⇒ (x y)
(lset-difference eqv? '(1 2 3) '(3 1))    ⇒ (2)
(lset-difference eqv? '(1 2 3) '(3) '(2)) ⇒ (1)
```

The return from `lset-difference` may share a tail with *list1*. `lset-difference!` may modify *list1* to form its result.

`lset-diff+intersection` = *list1 list2 ...* [Scheme Procedure]
`lset-diff+intersection!` = *list1 list2 ...* [Scheme Procedure]

Return two values (see Section 6.13.7 [Multiple Values], page 309), the difference and intersection of the argument lists as per `lset-difference` and `lset-intersection` above.

For two list arguments this partitions *list1* into those elements of *list1* which are in *list2* and not in *list2*. (But for more than two arguments there can be elements of *list1* which are neither part of the difference nor the intersection.)

One of the return values from `lset-diff+intersection` may share a tail with *list1*. `lset-diff+intersection!` may modify *list1* to form its results.

`lset-xor` = *list ...* [Scheme Procedure]
`lset-xor!` = *list ...* [Scheme Procedure]

Return an XOR of the argument lists. For two lists this means those elements which are in exactly one of the lists. For more than two lists it means those elements which appear in an odd number of the lists.

To be precise, the XOR of two lists x and y is formed by taking those elements of x not equal to any element of y, plus those elements of y not equal to any element of x. Equality is determined with the given = procedure, called as `(= e1 e2)`. One argument is from x and the other from y, but which way around is unspecified. Exactly which calls are made is also unspecified, as is the order of the elements in the result.

```
(lset-xor eqv? '(x y))             ⇒ (x y)
(lset-xor eqv? '(1 2 3) '(4 3 2))  ⇒ (4 1)
```

The return from `lset-xor` may share a tail with one of the list arguments. `lset-xor!` may modify *list1* to form its result.

7.5.4 SRFI-2 - and-let*

The following syntax can be obtained with

```
(use-modules (srfi srfi-2))
```

or alternatively

```
(use-modules (ice-9 and-let-star))
```

`and-let*` (*clause ...*) *body ...* [library syntax]

A combination of `and` and `let*`.

Each *clause* is evaluated in turn, and if `#f` is obtained then evaluation stops and `#f` is returned. If all are non-`#f` then *body* is evaluated and the last form gives the return

value, or if *body* is empty then the result is `#t`. Each *clause* should be one of the following,

`(symbol expr)`
: Evaluate *expr*, check for `#f`, and bind it to *symbol*. Like `let*`, that binding is available to subsequent clauses.

`(expr)`
: Evaluate *expr* and check for `#f`.

`symbol`
: Get the value bound to *symbol* and check for `#f`.

Notice that `(expr)` has an "extra" pair of parentheses, for instance `((eq? x y))`. One way to remember this is to imagine the `symbol` in `(symbol expr)` is omitted.

`and-let*` is good for calculations where a `#f` value means termination, but where a non-`#f` value is going to be needed in subsequent expressions.

The following illustrates this, it returns text between brackets '`[...]`' in a string, or `#f` if there are no such brackets (ie. either `string-index` gives `#f`).

```
(define (extract-brackets str)
  (and-let* ((start (string-index str #\[))
             (end   (string-index str #\] start)))
    (substring str (1+ start) end)))
```

The following shows plain variables and expressions tested too. `diagnostic-levels` is taken to be an alist associating a diagnostic type with a level. `str` is printed only if the type is known and its level is high enough.

```
(define (show-diagnostic type str)
  (and-let* (want-diagnostics
             (level (assq-ref diagnostic-levels type))
             ((>= level current-diagnostic-level)))
    (display str)))
```

The advantage of `and-let*` is that an extended sequence of expressions and tests doesn't require lots of nesting as would arise from separate `and` and `let*`, or from `cond` with `=>`.

7.5.5 SRFI-4 - Homogeneous numeric vector datatypes

SRFI-4 provides an interface to uniform numeric vectors: vectors whose elements are all of a single numeric type. Guile offers uniform numeric vectors for signed and unsigned 8-bit, 16-bit, 32-bit, and 64-bit integers, two sizes of floating point values, and, as an extension to SRFI-4, complex floating-point numbers of these two sizes.

The standard SRFI-4 procedures and data types may be included via loading the appropriate module:

```
(use-modules (srfi srfi-4))
```

This module is currently a part of the default Guile environment, but it is a good practice to explicitly import the module. In the future, using SRFI-4 procedures without importing the SRFI-4 module will cause a deprecation message to be printed. (Of course, one may call the C functions at any time. Would that C had modules!)

7.5.5.1 SRFI-4 - Overview

Uniform numeric vectors can be useful since they consume less memory than the non-uniform, general vectors. Also, since the types they can store correspond directly to C types, it is easier to work with them efficiently on a low level. Consider image processing as an example, where you want to apply a filter to some image. While you could store the pixels of an image in a general vector and write a general convolution function, things are much more efficient with uniform vectors: the convolution function knows that all pixels are unsigned 8-bit values (say), and can use a very tight inner loop.

This is implemented in Scheme by having the compiler notice calls to the SRFI-4 accessors, and inline them to appropriate compiled code. From C you have access to the raw array; functions for efficiently working with uniform numeric vectors from C are listed at the end of this section.

Uniform numeric vectors are the special case of one dimensional uniform numeric arrays.

There are 12 standard kinds of uniform numeric vectors, and they all have their own complement of constructors, accessors, and so on. Procedures that operate on a specific kind of uniform numeric vector have a "tag" in their name, indicating the element type.

`u8` unsigned 8-bit integers

`s8` signed 8-bit integers

`u16` unsigned 16-bit integers

`s16` signed 16-bit integers

`u32` unsigned 32-bit integers

`s32` signed 32-bit integers

`u64` unsigned 64-bit integers

`s64` signed 64-bit integers

`f32` the C type `float`

`f64` the C type `double`

In addition, Guile supports uniform arrays of complex numbers, with the nonstandard tags:

`c32` complex numbers in rectangular form with the real and imaginary part being a `float`

`c64` complex numbers in rectangular form with the real and imaginary part being a `double`

The external representation (ie. read syntax) for these vectors is similar to normal Scheme vectors, but with an additional tag from the tables above indicating the vector's type. For example,

```
#u16(1 2 3)
#f64(3.1415 2.71)
```

Note that the read syntax for floating-point here conflicts with `#f` for false. In Standard Scheme one can write `(1 #f3)` for a three element list `(1 #f 3)`, but for Guile `(1 #f3)` is invalid. `(1 #f 3)` is almost certainly what one should write anyway to make the intention clear, so this is rarely a problem.

7.5.5.2 SRFI-4 - API

Note that the `c32` and `c64` functions are only available from `(srfi srfi-4 gnu)`.

`u8vector?` *obj* [Scheme Procedure]
`s8vector?` *obj* [Scheme Procedure]
`u16vector?` *obj* [Scheme Procedure]
`s16vector?` *obj* [Scheme Procedure]
`u32vector?` *obj* [Scheme Procedure]
`s32vector?` *obj* [Scheme Procedure]
`u64vector?` *obj* [Scheme Procedure]
`s64vector?` *obj* [Scheme Procedure]
`f32vector?` *obj* [Scheme Procedure]
`f64vector?` *obj* [Scheme Procedure]
`c32vector?` *obj* [Scheme Procedure]
`c64vector?` *obj* [Scheme Procedure]
`scm_u8vector_p` (*obj*) [C Function]
`scm_s8vector_p` (*obj*) [C Function]
`scm_u16vector_p` (*obj*) [C Function]
`scm_s16vector_p` (*obj*) [C Function]
`scm_u32vector_p` (*obj*) [C Function]
`scm_s32vector_p` (*obj*) [C Function]
`scm_u64vector_p` (*obj*) [C Function]
`scm_s64vector_p` (*obj*) [C Function]
`scm_f32vector_p` (*obj*) [C Function]
`scm_f64vector_p` (*obj*) [C Function]
`scm_c32vector_p` (*obj*) [C Function]
`scm_c64vector_p` (*obj*) [C Function]

Return `#t` if *obj* is a homogeneous numeric vector of the indicated type.

`make-u8vector` *n* [*value*] [Scheme Procedure]
`make-s8vector` *n* [*value*] [Scheme Procedure]
`make-u16vector` *n* [*value*] [Scheme Procedure]
`make-s16vector` *n* [*value*] [Scheme Procedure]
`make-u32vector` *n* [*value*] [Scheme Procedure]
`make-s32vector` *n* [*value*] [Scheme Procedure]
`make-u64vector` *n* [*value*] [Scheme Procedure]
`make-s64vector` *n* [*value*] [Scheme Procedure]
`make-f32vector` *n* [*value*] [Scheme Procedure]
`make-f64vector` *n* [*value*] [Scheme Procedure]
`make-c32vector` *n* [*value*] [Scheme Procedure]
`make-c64vector` *n* [*value*] [Scheme Procedure]
`scm_make_u8vector` (*n*, *value*) [C Function]
`scm_make_s8vector` (*n*, *value*) [C Function]
`scm_make_u16vector` (*n*, *value*) [C Function]
`scm_make_s16vector` (*n*, *value*) [C Function]
`scm_make_u32vector` (*n*, *value*) [C Function]
`scm_make_s32vector` (*n*, *value*) [C Function]

`scm_make_u64vector` (*n*, *value*) [C Function]
`scm_make_s64vector` (*n*, *value*) [C Function]
`scm_make_f32vector` (*n*, *value*) [C Function]
`scm_make_f64vector` (*n*, *value*) [C Function]
`scm_make_c32vector` (*n*, *value*) [C Function]
`scm_make_c64vector` (*n*, *value*) [C Function]

Return a newly allocated homogeneous numeric vector holding *n* elements of the indicated type. If *value* is given, the vector is initialized with that value, otherwise the contents are unspecified.

`u8vector` *value ...* [Scheme Procedure]
`s8vector` *value ...* [Scheme Procedure]
`u16vector` *value ...* [Scheme Procedure]
`s16vector` *value ...* [Scheme Procedure]
`u32vector` *value ...* [Scheme Procedure]
`s32vector` *value ...* [Scheme Procedure]
`u64vector` *value ...* [Scheme Procedure]
`s64vector` *value ...* [Scheme Procedure]
`f32vector` *value ...* [Scheme Procedure]
`f64vector` *value ...* [Scheme Procedure]
`c32vector` *value ...* [Scheme Procedure]
`c64vector` *value ...* [Scheme Procedure]
`scm_u8vector` (*values*) [C Function]
`scm_s8vector` (*values*) [C Function]
`scm_u16vector` (*values*) [C Function]
`scm_s16vector` (*values*) [C Function]
`scm_u32vector` (*values*) [C Function]
`scm_s32vector` (*values*) [C Function]
`scm_u64vector` (*values*) [C Function]
`scm_s64vector` (*values*) [C Function]
`scm_f32vector` (*values*) [C Function]
`scm_f64vector` (*values*) [C Function]
`scm_c32vector` (*values*) [C Function]
`scm_c64vector` (*values*) [C Function]

Return a newly allocated homogeneous numeric vector of the indicated type, holding the given parameter *value*s. The vector length is the number of parameters given.

`u8vector-length` *vec* [Scheme Procedure]
`s8vector-length` *vec* [Scheme Procedure]
`u16vector-length` *vec* [Scheme Procedure]
`s16vector-length` *vec* [Scheme Procedure]
`u32vector-length` *vec* [Scheme Procedure]
`s32vector-length` *vec* [Scheme Procedure]
`u64vector-length` *vec* [Scheme Procedure]
`s64vector-length` *vec* [Scheme Procedure]
`f32vector-length` *vec* [Scheme Procedure]
`f64vector-length` *vec* [Scheme Procedure]
`c32vector-length` *vec* [Scheme Procedure]

`c64vector-length` *vec* [Scheme Procedure]
`scm_u8vector_length` (*vec*) [C Function]
`scm_s8vector_length` (*vec*) [C Function]
`scm_u16vector_length` (*vec*) [C Function]
`scm_s16vector_length` (*vec*) [C Function]
`scm_u32vector_length` (*vec*) [C Function]
`scm_s32vector_length` (*vec*) [C Function]
`scm_u64vector_length` (*vec*) [C Function]
`scm_s64vector_length` (*vec*) [C Function]
`scm_f32vector_length` (*vec*) [C Function]
`scm_f64vector_length` (*vec*) [C Function]
`scm_c32vector_length` (*vec*) [C Function]
`scm_c64vector_length` (*vec*) [C Function]

Return the number of elements in *vec*.

`u8vector-ref` *vec i* [Scheme Procedure]
`s8vector-ref` *vec i* [Scheme Procedure]
`u16vector-ref` *vec i* [Scheme Procedure]
`s16vector-ref` *vec i* [Scheme Procedure]
`u32vector-ref` *vec i* [Scheme Procedure]
`s32vector-ref` *vec i* [Scheme Procedure]
`u64vector-ref` *vec i* [Scheme Procedure]
`s64vector-ref` *vec i* [Scheme Procedure]
`f32vector-ref` *vec i* [Scheme Procedure]
`f64vector-ref` *vec i* [Scheme Procedure]
`c32vector-ref` *vec i* [Scheme Procedure]
`c64vector-ref` *vec i* [Scheme Procedure]
`scm_u8vector_ref` (*vec*, *i*) [C Function]
`scm_s8vector_ref` (*vec*, *i*) [C Function]
`scm_u16vector_ref` (*vec*, *i*) [C Function]
`scm_s16vector_ref` (*vec*, *i*) [C Function]
`scm_u32vector_ref` (*vec*, *i*) [C Function]
`scm_s32vector_ref` (*vec*, *i*) [C Function]
`scm_u64vector_ref` (*vec*, *i*) [C Function]
`scm_s64vector_ref` (*vec*, *i*) [C Function]
`scm_f32vector_ref` (*vec*, *i*) [C Function]
`scm_f64vector_ref` (*vec*, *i*) [C Function]
`scm_c32vector_ref` (*vec*, *i*) [C Function]
`scm_c64vector_ref` (*vec*, *i*) [C Function]

Return the element at index *i* in *vec*. The first element in *vec* is index 0.

`u8vector-set!` *vec i value* [Scheme Procedure]
`s8vector-set!` *vec i value* [Scheme Procedure]
`u16vector-set!` *vec i value* [Scheme Procedure]
`s16vector-set!` *vec i value* [Scheme Procedure]
`u32vector-set!` *vec i value* [Scheme Procedure]
`s32vector-set!` *vec i value* [Scheme Procedure]
`u64vector-set!` *vec i value* [Scheme Procedure]

`s64vector-set!` *vec i value* [Scheme Procedure]
`f32vector-set!` *vec i value* [Scheme Procedure]
`f64vector-set!` *vec i value* [Scheme Procedure]
`c32vector-set!` *vec i value* [Scheme Procedure]
`c64vector-set!` *vec i value* [Scheme Procedure]
`scm_u8vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s8vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u16vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s16vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_u64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_s64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_f32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_f64vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_c32vector_set_x` (*vec*, *i*, *value*) [C Function]
`scm_c64vector_set_x` (*vec*, *i*, *value*) [C Function]

Set the element at index *i* in *vec* to *value*. The first element in *vec* is index 0. The return value is unspecified.

`u8vector->list` *vec* [Scheme Procedure]
`s8vector->list` *vec* [Scheme Procedure]
`u16vector->list` *vec* [Scheme Procedure]
`s16vector->list` *vec* [Scheme Procedure]
`u32vector->list` *vec* [Scheme Procedure]
`s32vector->list` *vec* [Scheme Procedure]
`u64vector->list` *vec* [Scheme Procedure]
`s64vector->list` *vec* [Scheme Procedure]
`f32vector->list` *vec* [Scheme Procedure]
`f64vector->list` *vec* [Scheme Procedure]
`c32vector->list` *vec* [Scheme Procedure]
`c64vector->list` *vec* [Scheme Procedure]
`scm_u8vector_to_list` (*vec*) [C Function]
`scm_s8vector_to_list` (*vec*) [C Function]
`scm_u16vector_to_list` (*vec*) [C Function]
`scm_s16vector_to_list` (*vec*) [C Function]
`scm_u32vector_to_list` (*vec*) [C Function]
`scm_s32vector_to_list` (*vec*) [C Function]
`scm_u64vector_to_list` (*vec*) [C Function]
`scm_s64vector_to_list` (*vec*) [C Function]
`scm_f32vector_to_list` (*vec*) [C Function]
`scm_f64vector_to_list` (*vec*) [C Function]
`scm_c32vector_to_list` (*vec*) [C Function]
`scm_c64vector_to_list` (*vec*) [C Function]

Return a newly allocated list holding all elements of *vec*.

`list->u8vector` *lst* [Scheme Procedure]
`list->s8vector` *lst* [Scheme Procedure]

`list->u16vector` *lst* [Scheme Procedure]
`list->s16vector` *lst* [Scheme Procedure]
`list->u32vector` *lst* [Scheme Procedure]
`list->s32vector` *lst* [Scheme Procedure]
`list->u64vector` *lst* [Scheme Procedure]
`list->s64vector` *lst* [Scheme Procedure]
`list->f32vector` *lst* [Scheme Procedure]
`list->f64vector` *lst* [Scheme Procedure]
`list->c32vector` *lst* [Scheme Procedure]
`list->c64vector` *lst* [Scheme Procedure]
`scm_list_to_u8vector` (*lst*) [C Function]
`scm_list_to_s8vector` (*lst*) [C Function]
`scm_list_to_u16vector` (*lst*) [C Function]
`scm_list_to_s16vector` (*lst*) [C Function]
`scm_list_to_u32vector` (*lst*) [C Function]
`scm_list_to_s32vector` (*lst*) [C Function]
`scm_list_to_u64vector` (*lst*) [C Function]
`scm_list_to_s64vector` (*lst*) [C Function]
`scm_list_to_f32vector` (*lst*) [C Function]
`scm_list_to_f64vector` (*lst*) [C Function]
`scm_list_to_c32vector` (*lst*) [C Function]
`scm_list_to_c64vector` (*lst*) [C Function]

Return a newly allocated homogeneous numeric vector of the indicated type, initialized with the elements of the list *lst*.

`SCM scm_take_u8vector` (*const scm_t_uint8 *data, size_t len*) [C Function]
`SCM scm_take_s8vector` (*const scm_t_int8 *data, size_t len*) [C Function]
`SCM scm_take_u16vector` (*const scm_t_uint16 *data, size_t len*) [C Function]
`SCM scm_take_s16vector` (*const scm_t_int16 *data, size_t len*) [C Function]
`SCM scm_take_u32vector` (*const scm_t_uint32 *data, size_t len*) [C Function]
`SCM scm_take_s32vector` (*const scm_t_int32 *data, size_t len*) [C Function]
`SCM scm_take_u64vector` (*const scm_t_uint64 *data, size_t len*) [C Function]
`SCM scm_take_s64vector` (*const scm_t_int64 *data, size_t len*) [C Function]
`SCM scm_take_f32vector` (*const float *data, size_t len*) [C Function]
`SCM scm_take_f64vector` (*const double *data, size_t len*) [C Function]
`SCM scm_take_c32vector` (*const float *data, size_t len*) [C Function]
`SCM scm_take_c64vector` (*const double *data, size_t len*) [C Function]

Return a new uniform numeric vector of the indicated type and length that uses the memory pointed to by *data* to store its elements. This memory will eventually be freed with `free`. The argument *len* specifies the number of elements in *data*, not its size in bytes.

The `c32` and `c64` variants take a pointer to a C array of `float`s or `double`s. The real parts of the complex numbers are at even indices in that array, the corresponding imaginary parts are at the following odd index.

`const scm_t_uint8 * scm_u8vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const scm_t_int8 * scm_s8vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const scm_t_uint16 * scm_u16vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const scm_t_int16 * scm_s16vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const scm_t_uint32 * scm_u32vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const scm_t_int32 * scm_s32vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const scm_t_uint64 * scm_u64vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const scm_t_int64 * scm_s64vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const float * scm_f32vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const double * scm_f64vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const float * scm_c32vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`const double * scm_c64vector_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

Like `scm_vector_elements` (see Section 6.6.10.4 [Vector Accessing from C], page 189), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

`scm_t_uint8 * scm_u8vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`scm_t_int8 * scm_s8vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`scm_t_uint16 * scm_u16vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`scm_t_int16 * scm_s16vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`scm_t_uint32 * scm_u32vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`scm_t_int32 * scm_s32vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`scm_t_uint64 * scm_u64vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`scm_t_int64 * scm_s64vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]
`float * scm_f32vector_writable_elements` (*SCM vec*, *scm_t_array_handle *handle*, *size_t *lenp*, *ssize_t *incp*) [C Function]

`double * scm_f64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`float * scm_c32vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

`double * scm_c64vector_writable_elements` (*SCM vec, scm_t_array_handle *handle, size_t *lenp, ssize_t *incp*) [C Function]

Like `scm_vector_writable_elements` (see Section 6.6.10.4 [Vector Accessing from C], page 189), but returns a pointer to the elements of a uniform numeric vector of the indicated kind.

7.5.5.3 SRFI-4 - Relation to bytevectors

Guile implements SRFI-4 vectors using bytevectors (see Section 6.6.12 [Bytevectors], page 193). Often when you have a numeric vector, you end up wanting to write its bytes somewhere, or have access to the underlying bytes, or read in bytes from somewhere else. Bytevectors are very good at this sort of thing. But the SRFI-4 APIs are nicer to use when doing number-crunching, because they are addressed by element and not by byte.

So as a compromise, Guile allows all bytevector functions to operate on numeric vectors. They address the underlying bytes in the native endianness, as one would expect.

Following the same reasoning, that it's just bytes underneath, Guile also allows uniform vectors of a given type to be accessed as if they were of any type. One can fill a `u32vector`, and access its elements with `u8vector-ref`. One can use `f64vector-ref` on bytevectors. It's all the same to Guile.

In this way, uniform numeric vectors may be written to and read from input/output ports using the procedures that operate on bytevectors.

See Section 6.6.12 [Bytevectors], page 193, for more information.

7.5.5.4 SRFI-4 - Guile extensions

Guile defines some useful extensions to SRFI-4, which are not available in the default Guile environment. They may be imported by loading the extensions module:

```
(use-modules (srfi srfi-4 gnu))
```

`any->u8vector` *obj* [Scheme Procedure]

`any->s8vector` *obj* [Scheme Procedure]

`any->u16vector` *obj* [Scheme Procedure]

`any->s16vector` *obj* [Scheme Procedure]

`any->u32vector` *obj* [Scheme Procedure]

`any->s32vector` *obj* [Scheme Procedure]

`any->u64vector` *obj* [Scheme Procedure]

`any->s64vector` *obj* [Scheme Procedure]

`any->f32vector` *obj* [Scheme Procedure]

`any->f64vector` *obj* [Scheme Procedure]

`any->c32vector` *obj* [Scheme Procedure]

`any->c64vector` *obj* [Scheme Procedure]

`scm_any_to_u8vector` (*obj*) [C Function]

`scm_any_to_s8vector` (*obj*) [C Function]

`scm_any_to_u16vector` (*obj*) [C Function]

`scm_any_to_s16vector` (*obj*) [C Function]
`scm_any_to_u32vector` (*obj*) [C Function]
`scm_any_to_s32vector` (*obj*) [C Function]
`scm_any_to_u64vector` (*obj*) [C Function]
`scm_any_to_s64vector` (*obj*) [C Function]
`scm_any_to_f32vector` (*obj*) [C Function]
`scm_any_to_f64vector` (*obj*) [C Function]
`scm_any_to_c32vector` (*obj*) [C Function]
`scm_any_to_c64vector` (*obj*) [C Function]

Return a (maybe newly allocated) uniform numeric vector of the indicated type, initialized with the elements of *obj*, which must be a list, a vector, or a uniform vector. When *obj* is already a suitable uniform numeric vector, it is returned unchanged.

7.5.6 SRFI-6 - Basic String Ports

SRFI-6 defines the procedures `open-input-string`, `open-output-string` and `get-output-string`. These procedures are included in the Guile core, so using this module does not make any difference at the moment. But it is possible that support for SRFI-6 will be factored out of the core library in the future, so using this module does not hurt, after all.

7.5.7 SRFI-8 - receive

`receive` is a syntax for making the handling of multiple-value procedures easier. It is documented in See Section 6.13.7 [Multiple Values], page 309.

7.5.8 SRFI-9 - define-record-type

This SRFI is a syntax for defining new record types and creating predicate, constructor, and field getter and setter functions. It is documented in the "Data Types" section of the manual (see Section 6.6.16 [SRFI-9 Records], page 218).

7.5.9 SRFI-10 - Hash-Comma Reader Extension

This SRFI implements a reader extension `#,()` called hash-comma. It allows the reader to give new kinds of objects, for use both in data and as constants or literals in source code. This feature is available with

```
(use-modules (srfi srfi-10))
```

The new read syntax is of the form

```
#,(tag arg...)
```

where *tag* is a symbol and the *arg*s are objects taken as parameters. *tag*s are registered with the following procedure.

`define-reader-ctor` *tag proc* [Scheme Procedure]

Register *proc* as the constructor for a hash-comma read syntax starting with symbol *tag*, i.e. `#,(`*tag* `arg...)`. *proc* is called with the given arguments `(`*proc* `arg...)` and the object it returns is the result of the read.

For example, a syntax giving a list of N copies of an object.

```
(define-reader-ctor 'repeat
```

```
  (lambda (obj reps)
    (make-list reps obj)))

(display '#,(repeat 99 3))
⊣ (99 99 99)
```

Notice the quote ' when the `#,( )` is used. The `repeat` handler returns a list and the program must quote to use it literally, the same as any other list. Ie.

```
(display '#,(repeat 99 3))
⇒
(display '(99 99 99))
```

When a handler returns an object which is self-evaluating, like a number or a string, then there's no need for quoting, just as there's no need when giving those directly as literals. For example an addition,

```
(define-reader-ctor 'sum
  (lambda (x y)
    (+ x y)))
(display #,(sum 123 456)) ⊣ 579
```

Once `(srfi srfi-10)` has loaded, `#,()` is available globally, there's no need to use `(srfi srfi-10)` in later modules. Similarly the tags registered are global and can be used anywhere once registered.

We do not recommend `#,()` reader extensions, however, and for three reasons.

First of all, this SRFI is not modular: the tag is matched by name, not as an identifier within a scope. Defining a reader extension in one part of a program can thus affect unrelated parts of a program because the tag is not scoped.

Secondly, reader extensions can be hard to manage from a time perspective: when does the reader extension take effect? See Section 6.10.8 [Eval When], page 279, for more discussion.

Finally, reader extensions can easily produce objects that can't be reified to an object file by the compiler. For example if you define a reader extension that makes a hash table (see Section 6.6.22 [Hash Tables], page 238), then it will work fine when run with the interpreter, and you think you have a neat hack. But then if you try to compile your program, after wrangling with the `eval-when` concerns mentioned above, the compiler will carp that it doesn't know how to serialize a hash table to disk.

In the specific case of hash tables, it would be possible for Guile to know how to pack hash tables into compiled files, but this doesn't work in general. What if the object you produce is an instance of a record type? Guile would then have to serialize the record type to disk too, and then what happens if the program independently loads the code that defines the record type? Does it define the same type or a different type? Guile's record types are nominal, not structural, so the answer is not clear at all.

For all of these reasons we recommend macros over reader extensions. Macros fulfill many of the same needs while preserving modular composition, and their interaction with `eval-when` is well-known. If you need brevity, instead use `read-hash-extend` and make your reader extension expand to a macro invocation. In that way we preserve scoping as much as possible. See Section 6.18.1.6 [Reader Extensions], page 385.

7.5.10 SRFI-11 - let-values

This module implements the binding forms for multiple values `let-values` and `let*-values`. These forms are similar to `let` and `let*` (see Section 6.12.2 [Local Bindings], page 294), but they support binding of the values returned by multiple-valued expressions.

Write `(use-modules (srfi srfi-11))` to make the bindings available.

```
(let-values (((x y) (values 1 2))
             ((z f) (values 3 4)))
   (+ x y z f))
⇒
10
```

`let-values` performs all bindings simultaneously, which means that no expression in the binding clauses may refer to variables bound in the same clause list. `let*-values`, on the other hand, performs the bindings sequentially, just like `let*` does for single-valued expressions.

7.5.11 SRFI-13 - String Library

The SRFI-13 procedures are always available, See Section 6.6.5 [Strings], page 141.

7.5.12 SRFI-14 - Character-set Library

The SRFI-14 data type and procedures are always available, See Section 6.6.4 [Character Sets], page 134.

7.5.13 SRFI-16 - case-lambda

SRFI-16 defines a variable-arity `lambda` form, `case-lambda`. This form is available in the default Guile environment. See Section 6.9.5 [Case-lambda], page 256, for more information.

7.5.14 SRFI-17 - Generalized set!

This SRFI implements a generalized `set!`, allowing some "referencing" functions to be used as the target location of a `set!`. This feature is available from

```
(use-modules (srfi srfi-17))
```

For example `vector-ref` is extended so that

```
(set! (vector-ref vec idx) new-value)
```

is equivalent to

```
(vector-set! vec idx new-value)
```

The idea is that a `vector-ref` expression identifies a location, which may be either fetched or stored. The same form is used for the location in both cases, encouraging visual clarity. This is similar to the idea of an "lvalue" in C.

The mechanism for this kind of `set!` is in the Guile core (see Section 6.9.8 [Procedures with Setters], page 259). This module adds definitions of the following functions as procedures with setters, allowing them to be targets of a `set!`,

`car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `caaaar`, `caaadr`, `caadar`, `caaddr`, `cadaar`, `cadadr`, `caddar`, `cadddr`, `cdaaar`, `cdaadr`, `cdadar`, `cdaddr`, `cddaar`, `cddadr`, `cdddar`, `cddddr`

`string-ref`, `vector-ref`

The SRFI specifies `setter` (see Section 6.9.8 [Procedures with Setters], page 259) as a procedure with setter, allowing the setter for a procedure to be changed, eg. `(set! (setter foo) my-new-setter-handler)`. Currently Guile does not implement this, a setter can only be specified on creation (`getter-with-setter` below).

`getter-with-setter` [Function]
: The same as the Guile core `make-procedure-with-setter` (see Section 6.9.8 [Procedures with Setters], page 259).

7.5.15 SRFI-18 - Multithreading support

This is an implementation of the SRFI-18 threading and synchronization library. The functions and variables described here are provided by

```
(use-modules (srfi srfi-18))
```

SRFI-18 defines facilities for threads, mutexes, condition variables, time, and exception handling. Because these facilities are at a higher level than Guile's primitives, they are implemented as a layer on top of what Guile provides. In particular this means that a Guile mutex is not a SRFI-18 mutex, and a Guile thread is not a SRFI-18 thread, and so on. Guile provides a set of primitives and SRFI-18 is one of the systems built in terms of those primitives.

7.5.15.1 SRFI-18 Threads

Threads created by SRFI-18 differ in two ways from threads created by Guile's built-in thread functions. First, a thread created by SRFI-18 `make-thread` begins in a blocked state and will not start execution until `thread-start!` is called on it. Second, SRFI-18 threads are constructed with a top-level exception handler that captures any exceptions that are thrown on thread exit.

SRFI-18 threads are disjoint from Guile's primitive threads. See Section 6.22.1 [Threads], page 442, for more on Guile's primitive facility.

`current-thread` [Function]
: Returns the thread that called this function. This is the same procedure as the same-named built-in procedure `current-thread` (see Section 6.22.1 [Threads], page 442).

`thread?` *obj* [Function]
: Returns `#t` if *obj* is a thread, `#f` otherwise. This is the same procedure as the same-named built-in procedure `thread?` (see Section 6.22.1 [Threads], page 442).

`make-thread` *thunk* [*name*] [Function]
: Call `thunk` in a new thread and with a new dynamic state, returning the new thread and optionally assigning it the object name *name*, which may be any Scheme object.

  Note that the name `make-thread` conflicts with the `(ice-9 threads)` function `make-thread`. Applications wanting to use both of these functions will need to refer to them by different names.

`thread-name` *thread* [Function]
: Returns the name assigned to *thread* at the time of its creation, or `#f` if it was not given a name.

`thread-specific` *thread* [Function]
`thread-specific-set!` *thread obj* [Function]
Get or set the "object-specific" property of *thread*. In Guile's implementation of SRFI-18, this value is stored as an object property, and will be `#f` if not set.

`thread-start!` *thread* [Function]
Unblocks *thread* and allows it to begin execution if it has not done so already.

`thread-yield!` [Function]
If one or more threads are waiting to execute, calling `thread-yield!` forces an immediate context switch to one of them. Otherwise, `thread-yield!` has no effect. `thread-yield!` behaves identically to the Guile built-in function `yield`.

`thread-sleep!` *timeout* [Function]
The current thread waits until the point specified by the time object *timeout* is reached (see Section 7.5.15.4 [SRFI-18 Time], page 613). This blocks the thread only if *timeout* represents a point in the future. it is an error for *timeout* to be `#f`.

`thread-terminate!` *thread* [Function]
Causes an abnormal termination of *thread*. If *thread* is not already terminated, all mutexes owned by *thread* become unlocked/abandoned. If *thread* is the current thread, `thread-terminate!` does not return. Otherwise `thread-terminate!` returns an unspecified value; the termination of *thread* will occur before `thread-terminate!` returns. Subsequent attempts to join on *thread* will cause a "terminated thread exception" to be raised.

`thread-terminate!` is compatible with the thread cancellation procedures in the core threads API (see Section 6.22.1 [Threads], page 442) in that if a cleanup handler has been installed for the target thread, it will be called before the thread exits and its return value (or exception, if any) will be stored for later retrieval via a call to `thread-join!`.

`thread-join!` *thread* [*timeout* [*timeout-val*]] [Function]
Wait for *thread* to terminate and return its exit value. When a time value *timeout* is given, it specifies a point in time where the waiting should be aborted. When the waiting is aborted, *timeout-val* is returned if it is specified; otherwise, a `join-timeout-exception` exception is raised (see Section 7.5.15.5 [SRFI-18 Exceptions], page 613). Exceptions may also be raised if the thread was terminated by a call to `thread-terminate!` (`terminated-thread-exception` will be raised) or if the thread exited by raising an exception that was handled by the top-level exception handler (`uncaught-exception` will be raised; the original exception can be retrieved using `uncaught-exception-reason`).

7.5.15.2 SRFI-18 Mutexes

SRFI-18 mutexes are disjoint from Guile's primitive mutexes. See Section 6.22.5 [Mutexes and Condition Variables], page 448, for more on Guile's primitive facility.

`make-mutex` [*name*] [Function]
Returns a new mutex, optionally assigning it the object name *name*, which may be any Scheme object. The returned mutex will be created with the configuration described above.

`mutex-name` *mutex* [Function]
Returns the name assigned to *mutex* at the time of its creation, or `#f` if it was not given a name.

`mutex-specific` *mutex* [Function]
Return the "object-specific" property of *mutex*, or `#f` if none is set.

`mutex-specific-set!` *mutex obj* [Function]
Set the "object-specific" property of *mutex*.

`mutex-state` *mutex* [Function]
Returns information about the state of *mutex*. Possible values are:

- thread *t*: the mutex is in the locked/owned state and thread *t* is the owner of the mutex
- symbol `not-owned`: the mutex is in the locked/not-owned state
- symbol `abandoned`: the mutex is in the unlocked/abandoned state
- symbol `not-abandoned`: the mutex is in the unlocked/not-abandoned state

`mutex-lock!` *mutex* [*timeout* [*thread*]] [Function]
Lock *mutex*, optionally specifying a time object *timeout* after which to abort the lock attempt and a thread *thread* giving a new owner for *mutex* different than the current thread.

`mutex-unlock!` *mutex* [*condition-variable* [*timeout*]] [Function]
Unlock *mutex*, optionally specifying a condition variable *condition-variable* on which to wait, either indefinitely or, optionally, until the time object *timeout* has passed, to be signalled.

7.5.15.3 SRFI-18 Condition variables

SRFI-18 does not specify a "wait" function for condition variables. Waiting on a condition variable can be simulated using the SRFI-18 `mutex-unlock!` function described in the previous section.

SRFI-18 condition variables are disjoint from Guile's primitive condition variables. See Section 6.22.5 [Mutexes and Condition Variables], page 448, for more on Guile's primitive facility.

`condition-variable?` *obj* [Function]
Returns `#t` if *obj* is a condition variable, `#f` otherwise.

`make-condition-variable` [*name*] [Function]
Returns a new condition variable, optionally assigning it the object name *name*, which may be any Scheme object.

`condition-variable-name` *condition-variable* [Function]
Returns the name assigned to *condition-variable* at the time of its creation, or `#f` if it was not given a name.

`condition-variable-specific` *condition-variable* [Function]
Return the "object-specific" property of *condition-variable*, or `#f` if none is set.

`condition-variable-specific-set!` *condition-variable obj* [Function]
Set the "object-specific" property of *condition-variable*.

`condition-variable-signal!` *condition-variable* [Function]
`condition-variable-broadcast!` *condition-variable* [Function]
Wake up one thread that is waiting for *condition-variable*, in the case of `condition-variable-signal!`, or all threads waiting for it, in the case of `condition-variable-broadcast!`.

7.5.15.4 SRFI-18 Time

The SRFI-18 time functions manipulate time in two formats: a "time object" type that represents an absolute point in time in some implementation-specific way; and the number of seconds since some unspecified "epoch". In Guile's implementation, the epoch is the Unix epoch, 00:00:00 UTC, January 1, 1970.

`current-time` [Function]
Return the current time as a time object. This procedure replaces the procedure of the same name in the core library, which returns the current time in seconds since the epoch.

`time?` *obj* [Function]
Returns `#t` if *obj* is a time object, `#f` otherwise.

`time->seconds` *time* [Function]
`seconds->time` *seconds* [Function]
Convert between time objects and numerical values representing the number of seconds since the epoch. When converting from a time object to seconds, the return value is the number of seconds between *time* and the epoch. When converting from seconds to a time object, the return value is a time object that represents a time *seconds* seconds after the epoch.

7.5.15.5 SRFI-18 Exceptions

SRFI-18 exceptions are identical to the exceptions provided by Guile's implementation of SRFI-34. The behavior of exception handlers invoked to handle exceptions thrown from SRFI-18 functions, however, differs from the conventional behavior of SRFI-34 in that the continuation of the handler is the same as that of the call to the function. Handlers are called in a tail-recursive manner; the exceptions do not "bubble up".

`current-exception-handler` [Function]
Returns the current exception handler.

`with-exception-handler` *handler thunk* [Function]
Installs *handler* as the current exception handler and calls the procedure *thunk* with no arguments, returning its value as the value of the exception. *handler* must be a procedure that accepts a single argument. The current exception handler at the time this procedure is called will be restored after the call returns.

`raise` *obj* [Function]
Raise *obj* as an exception. This is the same procedure as the same-named procedure defined in SRFI 34.

`join-timeout-exception?` *obj* [Function]
: Returns `#t` if *obj* is an exception raised as the result of performing a timed join on a thread that does not exit within the specified timeout, `#f` otherwise.

`abandoned-mutex-exception?` *obj* [Function]
: Returns `#t` if *obj* is an exception raised as the result of attempting to lock a mutex that has been abandoned by its owner thread, `#f` otherwise.

`terminated-thread-exception?` *obj* [Function]
: Returns `#t` if *obj* is an exception raised as the result of joining on a thread that exited as the result of a call to `thread-terminate!`.

`uncaught-exception?` *obj* [Function]
`uncaught-exception-reason` *exc* [Function]
: `uncaught-exception?` returns `#t` if *obj* is an exception thrown as the result of joining a thread that exited by raising an exception that was handled by the top-level exception handler installed by `make-thread`. When this occurs, the original exception is preserved as part of the exception thrown by `thread-join!` and can be accessed by calling `uncaught-exception-reason` on that exception. Note that because this exception-preservation mechanism is a side-effect of `make-thread`, joining on threads that exited as described above but were created by other means will not raise this `uncaught-exception` error.

7.5.16 SRFI-19 - Time/Date Library

This is an implementation of the SRFI-19 time/date library. The functions and variables described here are provided by

```
(use-modules (srfi srfi-19))
```

7.5.16.1 SRFI-19 Introduction

This module implements time and date representations and calculations, in various time systems, including Coordinated Universal Time (UTC) and International Atomic Time (TAI).

For those not familiar with these time systems, TAI is based on a fixed length second derived from oscillations of certain atoms. UTC differs from TAI by an integral number of seconds, which is increased or decreased at announced times to keep UTC aligned to a mean solar day (the orbit and rotation of the earth are not quite constant).

So far, only increases in the TAI $\leftrightarrow$ UTC difference have been needed. Such an increase is a "leap second", an extra second of TAI introduced at the end of a UTC day. When working entirely within UTC this is never seen, every day simply has 86400 seconds. But when converting from TAI to a UTC date, an extra 23:59:60 is present, where normally a day would end at 23:59:59. Effectively the UTC second from 23:59:59 to 00:00:00 has taken two TAI seconds.

In the current implementation, the system clock is assumed to be UTC, and a table of leap seconds in the code converts to TAI. See comments in `srfi-19.scm` for how to update this table.

Also, for those not familiar with the terminology, a *Julian Day* represents a point in time as a real number of days since -4713-11-24T12:00:00Z, i.e. midday UT on 24 November 4714 BC in the proleptic Gregorian calendar (1 January 4713 BC in the proleptic Julian calendar).

A *Modified Julian Day* represents a point in time as a real number of days since 1858-11-17T00:00:00Z, i.e. midnight UT on Wednesday 17 November AD 1858. That time is julian day 2400000.5.

7.5.16.2 SRFI-19 Time

A *time* object has type, seconds and nanoseconds fields representing a point in time starting from some epoch. This is an arbitrary point in time, not just a time of day. Although times are represented in nanoseconds, the actual resolution may be lower.

The following variables hold the possible time types. For instance `(current-time time-process)` would give the current CPU process time.

`time-utc` [Variable]
: Universal Coordinated Time (UTC).

`time-tai` [Variable]
: International Atomic Time (TAI).

`time-monotonic` [Variable]
: Monotonic time, meaning a monotonically increasing time starting from an unspecified epoch.

  Note that in the current implementation `time-monotonic` is the same as `time-tai`, and unfortunately is therefore affected by adjustments to the system clock. Perhaps this will change in the future.

`time-duration` [Variable]
: A duration, meaning simply a difference between two times.

`time-process` [Variable]
: CPU time spent in the current process, starting from when the process began.

`time-thread` [Variable]
: CPU time spent in the current thread. Not currently implemented.

`time?` *obj* [Function]
: Return `#t` if *obj* is a time object, or `#f` if not.

`make-time` *type nanoseconds seconds* [Function]
: Create a time object with the given *type*, *seconds* and *nanoseconds*.

`time-type` *time* [Function]
`time-nanosecond` *time* [Function]
`time-second` *time* [Function]
`set-time-type!` *time type* [Function]
`set-time-nanosecond!` *time nsec* [Function]

`set-time-second!` *time sec* [Function]
`set-time-type!` *time type* [Function]
Get or set the type, seconds or nanoseconds fields of a time object.

`set-time-type!` merely changes the field, it doesn't convert the time value. For conversions, see Section 7.5.16.4 [SRFI-19 Time/Date conversions], page 618.

`copy-time` *time* [Function]
Return a new time object, which is a copy of the given *time*.

`current-time` [*type*] [Function]
Return the current time of the given *type*. The default *type* is `time-utc`.

Note that the name `current-time` conflicts with the Guile core `current-time` function (see Section 7.2.5 [Time], page 513) as well as the SRFI-18 `current-time` function (see Section 7.5.15.4 [SRFI-18 Time], page 613). Applications wanting to use more than one of these functions will need to refer to them by different names.

`time-resolution` [*type*] [Function]
Return the resolution, in nanoseconds, of the given time *type*. The default *type* is `time-utc`.

`time<=?` *t1 t2* [Function]
`time<?` *t1 t2* [Function]
`time=?` *t1 t2* [Function]
`time>=?` *t1 t2* [Function]
`time>?` *t1 t2* [Function]
Return `#t` or `#f` according to the respective relation between time objects *t1* and *t2*. *t1* and *t2* must be the same time type.

`time-difference` *t1 t2* [Function]
`time-difference!` *t1 t2* [Function]
Return a time object of type `time-duration` representing the period between *t1* and *t2*. *t1* and *t2* must be the same time type.

`time-difference` returns a new time object, `time-difference!` may modify *t1* to form its return.

`add-duration` *time duration* [Function]
`add-duration!` *time duration* [Function]
`subtract-duration` *time duration* [Function]
`subtract-duration!` *time duration* [Function]
Return a time object which is *time* with the given *duration* added or subtracted. *duration* must be a time object of type `time-duration`.

`add-duration` and `subtract-duration` return a new time object. `add-duration!` and `subtract-duration!` may modify the given *time* to form their return.

7.5.16.3 SRFI-19 Date

A *date* object represents a date in the Gregorian calendar and a time of day on that date in some timezone.

The fields are year, month, day, hour, minute, second, nanoseconds and timezone. A date object is immutable, its fields can be read but they cannot be modified once the object is created.

Historically, the Gregorian calendar was only used from the latter part of the year 1582 onwards, and not until even later in many countries. Prior to that most countries used the Julian calendar. SRFI-19 does not deal with the Julian calendar at all, and so does not reflect this historical calendar reform. Instead it projects the Gregorian calendar back proleptically as far as necessary. When dealing with historical data, especially prior to the British Empire's adoption of the Gregorian calendar in 1752, one should be mindful of which calendar is used in each context, and apply non-SRFI-19 facilities to convert where necessary.

`date?` *obj* [Function]
Return `#t` if *obj* is a date object, or `#f` if not.

`make-date` *nsecs seconds minutes hours date month year zone-offset* [Function]
Create a new date object.

`date-nanosecond` *date* [Function]
Nanoseconds, 0 to 999999999.

`date-second` *date* [Function]
Seconds, 0 to 59, or 60 for a leap second. 60 is never seen when working entirely within UTC, it's only when converting to or from TAI.

`date-minute` *date* [Function]
Minutes, 0 to 59.

`date-hour` *date* [Function]
Hour, 0 to 23.

`date-day` *date* [Function]
Day of the month, 1 to 31 (or less, according to the month).

`date-month` *date* [Function]
Month, 1 to 12.

`date-year` *date* [Function]
Year, eg. 2003. Dates B.C. are negative, eg. −46 is 46 B.C. There is no year 0, year −1 is followed by year 1.

`date-zone-offset` *date* [Function]
Time zone, an integer number of seconds east of Greenwich.

`date-year-day` *date* [Function]
Day of the year, starting from 1 for 1st January.

`date-week-day` *date* [Function]
Day of the week, starting from 0 for Sunday.

`date-week-number` *date dstartw* [Function]
Week of the year, ignoring a first partial week. *dstartw* is the day of the week which is taken to start a week, 0 for Sunday, 1 for Monday, etc.

`current-date` [*tz-offset*] [Function]
Return a date object representing the current date/time, in UTC offset by *tz-offset*. *tz-offset* is seconds east of Greenwich and defaults to the local timezone.

`current-julian-day` [Function]
Return the current Julian Day.

`current-modified-julian-day` [Function]
Return the current Modified Julian Day.

7.5.16.4 SRFI-19 Time/Date conversions

`date->julian-day` *date* [Function]
`date->modified-julian-day` *date* [Function]
`date->time-monotonic` *date* [Function]
`date->time-tai` *date* [Function]
`date->time-utc` *date* [Function]

`julian-day->date` *jdn* [*tz-offset*] [Function]
`julian-day->time-monotonic` *jdn* [Function]
`julian-day->time-tai` *jdn* [Function]
`julian-day->time-utc` *jdn* [Function]

`modified-julian-day->date` *jdn* [*tz-offset*] [Function]
`modified-julian-day->time-monotonic` *jdn* [Function]
`modified-julian-day->time-tai` *jdn* [Function]
`modified-julian-day->time-utc` *jdn* [Function]

`time-monotonic->date` *time* [*tz-offset*] [Function]
`time-monotonic->time-tai` *time* [Function]
`time-monotonic->time-tai!` *time* [Function]
`time-monotonic->time-utc` *time* [Function]
`time-monotonic->time-utc!` *time* [Function]

`time-tai->date` *time* [*tz-offset*] [Function]
`time-tai->julian-day` *time* [Function]
`time-tai->modified-julian-day` *time* [Function]
`time-tai->time-monotonic` *time* [Function]
`time-tai->time-monotonic!` *time* [Function]
`time-tai->time-utc` *time* [Function]
`time-tai->time-utc!` *time* [Function]

`time-utc->date` *time* [*tz-offset*] [Function]
`time-utc->julian-day` *time* [Function]
`time-utc->modified-julian-day` *time* [Function]
`time-utc->time-monotonic` *time* [Function]
`time-utc->time-monotonic!` *time* [Function]

`time-utc->time-tai` *time* [Function]
`time-utc->time-tai!` *time* [Function]

Convert between dates, times and days of the respective types. For instance `time-tai->time-utc` accepts a *time* object of type `time-tai` and returns an object of type `time-utc`.

The `!` variants may modify their *time* argument to form their return. The plain functions create a new object.

For conversions to dates, *tz-offset* is seconds east of Greenwich. The default is the local timezone, at the given time, as provided by the system, using `localtime` (see Section 7.2.5 [Time], page 513).

On 32-bit systems, `localtime` is limited to a 32-bit `time_t`, so a default *tz-offset* is only available for times between Dec 1901 and Jan 2038. For prior dates an application might like to use the value in 1902, though some locations have zone changes prior to that. For future dates an application might like to assume today's rules extend indefinitely. But for correct daylight savings transitions it will be necessary to take an offset for the same day and time but a year in range and which has the same starting weekday and same leap/non-leap (to support rules like last Sunday in October).

7.5.16.5 SRFI-19 Date to string

`date->string` *date* [*format*] [Function]

Convert a date to a string under the control of a format. *format* should be a string containing '`~`' escapes, which will be expanded as per the following conversion table. The default *format* is '`~c`', a locale-dependent date and time.

Many of these conversion characters are the same as POSIX `strftime` (see Section 7.2.5 [Time], page 513), but there are some extras and some variations.

`~~`	literal `~`
`~a`	locale abbreviated weekday, eg. '`Sun`'
`~A`	locale full weekday, eg. '`Sunday`'
`~b`	locale abbreviated month, eg. '`Jan`'
`~B`	locale full month, eg. '`January`'
`~c`	locale date and time, eg. '`Fri Jul 14 20:28:42-0400 2000`'
`~d`	day of month, zero padded, '`01`' to '`31`'
`~e`	day of month, blank padded, '` 1`' to '`31`'
`~f`	seconds and fractional seconds, with locale decimal point, eg. '`5.2`'
`~h`	same as `~b`
`~H`	hour, 24-hour clock, zero padded, '`00`' to '`23`'
`~I`	hour, 12-hour clock, zero padded, '`01`' to '`12`'
`~j`	day of year, zero padded, '`001`' to '`366`'
`~k`	hour, 24-hour clock, blank padded, '` 0`' to '`23`'
`~l`	hour, 12-hour clock, blank padded, '` 1`' to '`12`'
`~m`	month, zero padded, '`01`' to '`12`'
`~M`	minute, zero padded, '`00`' to '`59`'

`~n`	newline
`~N`	nanosecond, zero padded, '`000000000`' to '`999999999`'
`~p`	locale AM or PM
`~r`	time, 12 hour clock, '`~I:~M:~S ~p`'
`~s`	number of full seconds since "the epoch" in UTC
`~S`	second, zero padded '`00`' to '`60`' (usual limit is 59, 60 is a leap second)
`~t`	horizontal tab character
`~T`	time, 24 hour clock, '`~H:~M:~S`'
`~U`	week of year, Sunday first day of week, '`00`' to '`52`'
`~V`	week of year, Monday first day of week, '`01`' to '`53`'
`~w`	day of week, 0 for Sunday, '`0`' to '`6`'
`~W`	week of year, Monday first day of week, '`00`' to '`52`'
`~y`	year, two digits, '`00`' to '`99`'
`~Y`	year, full, eg. '`2003`'
`~z`	time zone, RFC-822 style
`~Z`	time zone symbol (not currently implemented)
`~1`	ISO-8601 date, '`~Y-~m-~d`'
`~2`	ISO-8601 time+zone, '`~H:~M:~S~z`'
`~3`	ISO-8601 time, '`~H:~M:~S`'
`~4`	ISO-8601 date/time+zone, '`~Y-~m-~dT~H:~M:~S~z`'
`~5`	ISO-8601 date/time, '`~Y-~m-~dT~H:~M:~S`'

Conversions '`~D`', '`~x`' and '`~X`' are not currently described here, since the specification and reference implementation differ.

Conversion is locale-dependent on systems that support it (see Section 6.25.5 [Accessing Locale Information], page 468). See Section 7.2.13 [Locales], page 547, for information on how to change the current locale.

7.5.16.6 SRFI-19 String to date

`string->date` *input template* [Function]

Convert an *input* string to a date under the control of a *template* string. Return a newly created date object.

Literal characters in *template* must match characters in *input* and '`~`' escapes must match the input forms described in the table below. "Skip to" means characters up to one of the given type are ignored, or "no skip" for no skipping. "Read" is what's then read, and "Set" is the field affected in the date object.

For example '`~Y`' skips input characters until a digit is reached, at which point it expects a year and stores that to the year field of the date.

	Skip to	Read	Set
`~~`	no skip	literal ~	nothing
`~a`	`char-alphabetic?`	locale abbreviated weekday name	nothing

~A	char-alphabetic?	locale full weekday name	nothing
~b	char-alphabetic?	locale abbreviated month name	date-month
~B	char-alphabetic?	locale full month name	date-month
~d	char-numeric?	day of month	date-day
~e	no skip	day of month, blank padded	date-day
~h	same as '~b'		
~H	char-numeric?	hour	date-hour
~k	no skip	hour, blank padded	date-hour
~m	char-numeric?	month	date-month
~M	char-numeric?	minute	date-minute
~N	char-numeric?	nanosecond	date-nanose
~S	char-numeric?	second	date-second
~y	no skip	2-digit year	date-year 50 years
~Y	char-numeric?	year	date-year
~z	no skip	time zone	date-zone-offs

Notice that the weekday matching forms don't affect the date object returned, instead the weekday will be derived from the day, month and year.

Conversion is locale-dependent on systems that support it (see Section 6.25.5 [Accessing Locale Information], page 468). See Section 7.2.13 [Locales], page 547, for information on how to change the current locale.

7.5.17 SRFI-23 - Error Reporting

The SRFI-23 `error` procedure is always available.

7.5.18 SRFI-26 - specializing parameters

This SRFI provides a syntax for conveniently specializing selected parameters of a function. It can be used with,

```
(use-modules (srfi srfi-26))
```

`cut` *slot1 slot2 ...* [library syntax]
`cute` *slot1 slot2 ...* [library syntax]

Return a new procedure which will make a call (*slot1 slot2 ...*) but with selected parameters specialized to given expressions.

An example will illustrate the idea. The following is a specialization of `write`, sending output to `my-output-port`,

```
(cut write <> my-output-port)
⇒
(lambda (obj) (write obj my-output-port))
```

The special symbol `<>` indicates a slot to be filled by an argument to the new procedure. `my-output-port` on the other hand is an expression to be evaluated and passed, ie. it specializes the behaviour of `write`.

`<>` A slot to be filled by an argument from the created procedure. Arguments are assigned to `<>` slots in the order they appear in the `cut` form, there's no way to re-arrange arguments.

The first argument to `cut` is usually a procedure (or expression giving a procedure), but `<>` is allowed there too. For example,

```
(cut <> 1 2 3)
⇒
(lambda (proc) (proc 1 2 3))
```

`<...>` A slot to be filled by all remaining arguments from the new procedure. This can only occur at the end of a `cut` form.

For example, a procedure taking a variable number of arguments like `max` but in addition enforcing a lower bound,

```
(define my-lower-bound 123)

(cut max my-lower-bound <...>)
⇒
(lambda arglist (apply max my-lower-bound arglist))
```

For `cut` the specializing expressions are evaluated each time the new procedure is called. For `cute` they're evaluated just once, when the new procedure is created. The name `cute` stands for "`cut` with evaluated arguments". In all cases the evaluations take place in an unspecified order.

The following illustrates the difference between `cut` and `cute`,

```
(cut format <> "the time is ~s" (current-time))
⇒
(lambda (port) (format port "the time is ~s" (current-time)))

(cute format <> "the time is ~s" (current-time))
⇒
(let ((val (current-time)))
  (lambda (port) (format port "the time is ~s" val))
```

(There's no provision for a mixture of `cut` and `cute` where some expressions would be evaluated every time but others evaluated only once.)

`cut` is really just a shorthand for the sort of `lambda` forms shown in the above examples. But notice `cut` avoids the need to name unspecialized parameters, and is more compact. Use in functional programming style or just with `map`, `for-each` or similar is typical.

```
(map (cut * 2 <>) '(1 2 3 4))

(for-each (cut write <> my-port) my-list)
```

7.5.19 SRFI-27 - Sources of Random Bits

This subsection is based on the specification of SRFI-27 (`http://srfi.schemers.org/srfi-27/srfi-27.html`) written by Sebastian Egner.

This SRFI provides access to a (pseudo) random number generator; for Guile's built-in random number facilities, which SRFI-27 is implemented upon, See Section 6.6.2.14 [Random], page 127. With SRFI-27, random numbers are obtained from a *random source*, which encapsulates a random number generation algorithm and its state.

7.5.19.1 The Default Random Source

`random-integer` *n* [Function]
: Return a random number between zero (inclusive) and *n* (exclusive), using the default random source. The numbers returned have a uniform distribution.

`random-real` [Function]
: Return a random number in (0,1), using the default random source. The numbers returned have a uniform distribution.

`default-random-source` [Function]
: A random source from which `random-integer` and `random-real` have been derived using `random-source-make-integers` and `random-source-make-reals` (see Section 7.5.19.3 [SRFI-27 Random Number Generators], page 624, for those procedures). Note that an assignment to `default-random-source` does not change `random-integer` or `random-real`; it is also strongly recommended not to assign a new value.

7.5.19.2 Random Sources

`make-random-source` [Function]
: Create a new random source. The stream of random numbers obtained from each random source created by this procedure will be identical, unless its state is changed by one of the procedures below.

`random-source?` *object* [Function]
: Tests whether *object* is a random source. Random sources are a disjoint type.

`random-source-randomize!` *source* [Function]
: Attempt to set the state of the random source to a truly random value. The current implementation uses a seed based on the current system time.

`random-source-pseudo-randomize!` *source i j* [Function]
: Changes the state of the random source s into the initial state of the (*i*, *j*)-th independent random source, where *i* and *j* are non-negative integers. This procedure provides a mechanism to obtain a large number of independent random sources (usually all derived from the same backbone generator), indexed by two integers. In contrast to `random-source-randomize!`, this procedure is entirely deterministic.

The state associated with a random state can be obtained an reinstated with the following procedures:

`random-source-state-ref` *source* [Function]
`random-source-state-set!` *source state* [Function]
: Get and set the state of a random source. No assumptions should be made about the nature of the state object, besides it having an external representation (i.e. it can be passed to `write` and subsequently `read` back).

7.5.19.3 Obtaining random number generator procedures

`random-source-make-integers` *source* [Function]
: Obtains a procedure to generate random integers using the random source *source*. The returned procedure takes a single argument *n*, which must be a positive integer, and returns the next uniformly distributed random integer from the interval {0, ..., *n*-1} by advancing the state of *source*.

  If an application obtains and uses several generators for the same random source *source*, a call to any of these generators advances the state of *source*. Hence, the generators do not produce the same sequence of random integers each but rather share a state. This also holds for all other types of generators derived from a fixed random sources.

  While the SRFI text specifies that "Implementations that support concurrency make sure that the state of a generator is properly advanced", this is currently not the case in Guile's implementation of SRFI-27, as it would cause a severe performance penalty. So in multi-threaded programs, you either must perform locking on random sources shared between threads yourself, or use different random sources for multiple threads.

`random-source-make-reals` *source* [Function]
`random-source-make-reals` *source unit* [Function]
: Obtains a procedure to generate random real numbers $0 < x < 1$ using the random source *source*. The procedure rand is called without arguments.

  The optional parameter *unit* determines the type of numbers being produced by the returned procedure and the quantization of the output. *unit* must be a number such that $0 < unit < 1$. The numbers created by the returned procedure are of the same numerical type as *unit* and the potential output values are spaced by at most *unit*. One can imagine rand to create numbers as $x * unit$ where x is a random integer in {1, ..., floor(1/unit)-1}. Note, however, that this need not be the way the values are actually created and that the actual resolution of rand can be much higher than unit. In case *unit* is absent it defaults to a reasonably small value (related to the width of the mantissa of an efficient number format).

7.5.20 SRFI-28 - Basic Format Strings

SRFI-28 provides a basic `format` procedure that provides only the `~a`, `~s`, `~%`, and `~~` format specifiers. You can import this procedure by using:

```
(use-modules (srfi srfi-28))
```

`format` *message arg ...* [Scheme Procedure]

Returns a formatted message, using *message* as the format string, which can contain the following format specifiers:

`~a` Insert the textual representation of the next *arg*, as if printed by `display`.

`~s` Insert the textual representation of the next *arg*, as if printed by `write`.

`~%` Insert a newline.

`~~` Insert a tilde.

This procedure is the same as calling `simple-format` (see Section 6.14.5 [Simple Output], page 338) with `#f` as the destination.

7.5.21 SRFI-30 - Nested Multi-line Comments

Starting from version 2.0, Guile's `read` supports SRFI-30/R6RS nested multi-line comments by default, Section 6.18.1.3 [Block Comments], page 384.

7.5.22 SRFI-31 - A special form 'rec' for recursive evaluation

SRFI-31 defines a special form that can be used to create self-referential expressions more conveniently. The syntax is as follows:

```
<rec expression> --> (rec <variable> <expression>)
<rec expression> --> (rec (<variable>+) <body>)
```

The first syntax can be used to create self-referential expressions, for example:

```
guile> (define tmp (rec ones (cons 1 (delay ones))))
```

The second syntax can be used to create anonymous recursive functions:

```
guile> (define tmp (rec (display-n item n)
                        (if (positive? n)
                            (begin (display n) (display-n (- n 1))))))
guile> (tmp 42 3)
424242
guile>
```

7.5.23 SRFI-34 - Exception handling for programs

Guile provides an implementation of SRFI-34's exception handling mechanisms (`http://srfi.schemers.org/srfi-34/srfi-34.html`) as an alternative to its own built-in mechanisms (see Section 6.13.8 [Exceptions], page 311). It can be made available as follows:

```
(use-modules (srfi srfi-34))
```

See Section 6.13.8.2 [Raising and Handling Exceptions], page 315, for more on `with-exception-handler` and `raise` (known as `raise-exception` in core Guile).

SRFI-34's `guard` form is syntactic sugar over `with-exception-handler`:

`guard` (*var clause ...*) *body ...* [Syntax]
Evaluate *body* with an exception handler that binds the raised object to *var* and within the scope of that binding evaluates *clause*... as if they were the clauses of a cond expression. That implicit cond expression is evaluated with the continuation and dynamic environment of the guard expression.

If every *clause*'s test evaluates to false and there is no `else` clause, then `raise` is re-invoked on the raised object within the dynamic environment of the original call to `raise` except that the current exception handler is that of the `guard` expression.

7.5.24 SRFI-35 - Conditions

SRFI-35 (`http://srfi.schemers.org/srfi-35/srfi-35.html`) defines *conditions*, a data structure akin to records designed to convey information about exceptional conditions between parts of a program. It is normally used in conjunction with SRFI-34's `raise`:

```
(raise (condition (&message
                   (message "An error occurred"))))
```

Users can define *condition types* containing arbitrary information. Condition types may inherit from one another. This allows the part of the program that handles (or "catches") conditions to get accurate information about the exceptional condition that arose.

SRFI-35 conditions are made available using:

```
(use-modules (srfi srfi-35))
```

The procedures available to manipulate condition types are the following:

`make-condition-type` *id parent field-names* [Scheme Procedure]
Return a new condition type named *id*, inheriting from *parent*, and with the fields whose names are listed in *field-names*. *field-names* must be a list of symbols and must not contain names already used by *parent* or one of its supertypes.

`condition-type?` *obj* [Scheme Procedure]
Return true if *obj* is a condition type.

Conditions can be created and accessed with the following procedures:

`make-condition` *type . field+value* [Scheme Procedure]
Return a new condition of type *type* with fields initialized as specified by *field+value*, a sequence of field names (symbols) and values as in the following example:

```
(let ((&ct (make-condition-type 'foo &condition '(a b c))))
  (make-condition &ct 'a 1 'b 2 'c 3))
```

Note that all fields of *type* and its supertypes must be specified.

`make-compound-condition` *condition1 condition2 ...* [Scheme Procedure]
Return a new compound condition composed of *condition1 condition2 ...*. The returned condition has the type of each condition of condition1 condition2 ... (per `condition-has-type?`).

`condition-has-type?` *c type* [Scheme Procedure]
Return true if condition *c* has type *type*.

`condition-ref` *c field-name* [Scheme Procedure]
Return the value of the field named *field-name* from condition *c*.

If *c* is a compound condition and several underlying condition types contain a field named *field-name*, then the value of the first such field is returned, using the order in which conditions were passed to `make-compound-condition`.

`extract-condition` *c type* [Scheme Procedure]
Return a condition of condition type *type* with the field values specified by *c*.

If *c* is a compound condition, extract the field values from the subcondition belonging to *type* that appeared first in the call to `make-compound-condition` that created the condition.

Convenience macros are also available to create condition types and conditions.

`define-condition-type` *type supertype predicate field-spec...* [library syntax]
Define a new condition type named *type* that inherits from *supertype*. In addition, bind *predicate* to a type predicate that returns true when passed a condition of type *type* or any of its subtypes. *field-spec* must have the form `(field accessor)` where *field* is the name of field of *type* and *accessor* is the name of a procedure to access field *field* in conditions of type *type*.

The example below defines condition type `&foo`, inheriting from `&condition` with fields `a`, `b` and `c`:

```
(define-condition-type &foo &condition
  foo-condition?
  (a  foo-a)
  (b  foo-b)
  (c  foo-c))
```

`condition` *type-field-binding1 type-field-binding2 . . .* [library syntax]
Return a new condition or compound condition, initialized according to *type-field-binding1 type-field-binding2 . . .*. Each *type-field-binding* must have the form `(type field-specs...)`, where *type* is the name of a variable bound to a condition type; each *field-spec* must have the form `(field-name value)` where *field-name* is a symbol denoting the field being initialized to *value*. As for `make-condition`, all fields must be specified.

The following example returns a simple condition:

```
(condition (&message (message "An error occurred")))
```

The one below returns a compound condition:

```
(condition (&message (message "An error occurred"))
           (&serious))
```

Finally, SRFI-35 defines a several standard condition types.

`&condition` [Variable]
This condition type is the root of all condition types. It has no fields.

`&message` [Variable]
A condition type that carries a message describing the nature of the condition to humans.

`message-condition?` *c* [Scheme Procedure]
Return true if *c* is of type `&message` or one of its subtypes.

`condition-message` *c* [Scheme Procedure]
Return the message associated with message condition *c*.

`&serious` [Variable]
This type describes conditions serious enough that they cannot safely be ignored. It has no fields.

`serious-condition?` *c* [Scheme Procedure]
Return true if *c* is of type `&serious` or one of its subtypes.

`&error` [Variable]
This condition describes errors, typically caused by something that has gone wrong in the interaction of the program with the external world or the user.

`error?` *c* [Scheme Procedure]
Return true if *c* is of type `&error` or one of its subtypes.

As an implementation note, condition objects in Guile are the same as "exception objects". See Section 6.13.8.1 [Exception Objects], page 311. The `&condition`, `&serious`, and `&error` condition types are known in core Guile as `&exception`, `&error`, and `&external-error`, respectively.

7.5.25 SRFI-37 - args-fold

This is a processor for GNU `getopt_long`-style program arguments. It provides an alternative, less declarative interface than `getopt-long` in `(ice-9 getopt-long)` (see Section 7.4 [The (ice-9 getopt-long) Module], page 577). Unlike `getopt-long`, it supports repeated options and any number of short and long names per option. Access it with:

```
(use-modules (srfi srfi-37))
```

SRFI-37 principally provides an `option` type and the `args-fold` function. To use the library, create a set of options with `option` and use it as a specification for invoking `args-fold`.

Here is an example of a simple argument processor for the typical '`--version`' and '`--help`' options, which returns a backwards list of files given on the command line:

```
(args-fold (cdr (program-arguments))
           (let ((display-and-exit-proc
                  (lambda (msg)
                    (lambda (opt name arg loads)
                      (display msg) (quit)))))
             (list (option '(#\v "version") #f #f
                           (display-and-exit-proc "Foo version 42.0\n"))
                   (option '(#\h "help") #f #f
```

```
                    (display-and-exit-proc
                     "Usage: foo scheme-file ...")))
          (lambda (opt name arg loads)
            (error "Unrecognized option `~A'" name))
          (lambda (op loads) (cons op loads))
          '())
```

`option` *names required-arg? optional-arg? processor* [Scheme Procedure]
Return an object that specifies a single kind of program option.

names is a list of command-line option names, and should consist of characters for traditional `getopt` short options and strings for `getopt_long`-style long options.

required-arg? and *optional-arg?* are mutually exclusive; one or both must be `#f`. If *required-arg?*, the option must be followed by an argument on the command line, such as '`--opt=value`' for long options, or an error will be signalled. If *optional-arg?*, an argument will be taken if available.

processor is a procedure that takes at least 3 arguments, called when `args-fold` encounters the option: the containing option object, the name used on the command line, and the argument given for the option (or `#f` if none). The rest of the arguments are `args-fold` "seeds", and the *processor* should return seeds as well.

`option-names` *opt* [Scheme Procedure]
`option-required-arg?` *opt* [Scheme Procedure]
`option-optional-arg?` *opt* [Scheme Procedure]
`option-processor` *opt* [Scheme Procedure]
Return the specified field of *opt*, an option object, as described above for `option`.

`args-fold` *args options unrecognized-option-proc operand-proc seed ...* [Scheme Procedure]
Process *args*, a list of program arguments such as that returned by `(cdr (program-arguments))`, in order against *options*, a list of option objects as described above. All functions called take the "seeds", or the last multiple-values as multiple arguments, starting with *seed ...*, and must return the new seeds. Return the final seeds.

Call `unrecognized-option-proc`, which is like an option object's processor, for any options not found in *options*.

Call `operand-proc` with any items on the command line that are not named options. This includes arguments after '`--`'. It is called with the argument in question, as well as the seeds.

7.5.26 SRFI-38 - External Representation for Data With Shared Structure

This subsection is based on the specification of SRFI-38 (`http://srfi.schemers.org/srfi-38/srfi-38.html`) written by Ray Dillinger.

This SRFI creates an alternative external representation for data written and read using `write-with-shared-structure` and `read-with-shared-structure`. It is identical to the

grammar for external representation for data written and read with `write` and `read` given in section 7 of R5RS, except that the single production

```
<datum> --> <simple datum> | <compound datum>
```

is replaced by the following five productions:

```
<datum> --> <defining datum> | <nondefining datum> | <defined datum>
<defining datum> -->  #<indexnum>=<nondefining datum>
<defined datum> --> #<indexnum>#
<nondefining datum> --> <simple datum> | <compound datum>
<indexnum> --> <digit 10>+
```

`write-with-shared-structure` *obj* [Scheme procedure]
`write-with-shared-structure` *obj port* [Scheme procedure]
`write-with-shared-structure` *obj port optarg* [Scheme procedure]

Writes an external representation of *obj* to the given port. Strings that appear in the written representation are enclosed in doublequotes, and within those strings backslash and doublequote characters are escaped by backslashes. Character objects are written using the `#\` notation.

Objects which denote locations rather than values (cons cells, vectors, and non-zero-length strings in R5RS scheme; also Guile's structs, bytevectors and ports and hash-tables), if they appear at more than one point in the data being written, are preceded by '`#N=`' the first time they are written and replaced by '`#N#`' all subsequent times they are written, where N is a natural number used to identify that particular object. If objects which denote locations occur only once in the structure, then `write-with-shared-structure` must produce the same external representation for those objects as `write`.

`write-with-shared-structure` terminates in finite time and produces a finite representation when writing finite data.

`write-with-shared-structure` returns an unspecified value. The *port* argument may be omitted, in which case it defaults to the value returned by `(current-output-port)`. The *optarg* argument may also be omitted. If present, its effects on the output and return value are unspecified but `write-with-shared-structure` must still write a representation that can be read by `read-with-shared-structure`. Some implementations may wish to use *optarg* to specify formatting conventions, numeric radixes, or return values. Guile's implementation ignores *optarg*.

For example, the code

```
(begin (define a (cons 'val1 'val2))
       (set-cdr! a a)
       (write-with-shared-structure a))
```

should produce the output `#1=(val1 . #1#)`. This shows a cons cell whose `cdr` contains itself.

`read-with-shared-structure` [Scheme procedure]
`read-with-shared-structure` *port* [Scheme procedure]

`read-with-shared-structure` converts the external representations of Scheme objects produced by `write-with-shared-structure` into Scheme objects. That is,

it is a parser for the nonterminal '<datum>' in the augmented external representation grammar defined above. `read-with-shared-structure` returns the next object parsable from the given input port, updating *port* to point to the first character past the end of the external representation of the object.

If an end-of-file is encountered in the input before any characters are found that can begin an object, then an end-of-file object is returned. The port remains open, and further attempts to read it (by `read-with-shared-structure` or `read` will also return an end-of-file object. If an end of file is encountered after the beginning of an object's external representation, but the external representation is incomplete and therefore not parsable, an error is signalled.

The *port* argument may be omitted, in which case it defaults to the value returned by `(current-input-port)`. It is an error to read from a closed port.

7.5.27 SRFI-39 - Parameters

This SRFI adds support for dynamically-scoped parameters. SRFI 39 is implemented in the Guile core; there's no module needed to get SRFI-39 itself. Parameters are documented in Section 6.13.12 [Parameters], page 326.

This module does export one extra function: `with-parameters*`. This is a Guile-specific addition to the SRFI, similar to the core `with-fluids*` (see Section 6.13.11 [Fluids and Dynamic States], page 323).

`with-parameters*` *param-list value-list thunk* [Function]

Establish a new dynamic scope, as per `parameterize` above, taking parameters from *param-list* and corresponding values from *value-list*. A call `(thunk)` is made in the new scope and the result from that *thunk* is the return from `with-parameters*`.

7.5.28 SRFI-41 - Streams

This subsection is based on the specification of SRFI-41 (`http://srfi.schemers.org/srfi-41/srfi-41.html`) by Philip L. Bewig.

This SRFI implements streams, sometimes called lazy lists, a sequential data structure containing elements computed only on demand. A stream is either null or is a pair with a stream in its cdr. Since elements of a stream are computed only when accessed, streams can be infinite. Once computed, the value of a stream element is cached in case it is needed again. SRFI-41 can be made available with:

```
(use-modules (srfi srfi-41))
```

7.5.28.1 SRFI-41 Stream Fundamentals

SRFI-41 Streams are based on two mutually-recursive abstract data types: An object of the `stream` abstract data type is a promise that, when forced, is either `stream-null` or is an object of type `stream-pair`. An object of the `stream-pair` abstract data type contains a `stream-car` and a `stream-cdr`, which must be a `stream`. The essential feature of streams is the systematic suspensions of the recursive promises between the two data types.

The object stored in the `stream-car` of a `stream-pair` is a promise that is forced the first time the `stream-car` is accessed; its value is cached in case it is needed again. The object may have any type, and different stream elements may have different types. If the

`stream-car` is never accessed, the object stored there is never evaluated. Likewise, the `stream-cdr` is a promise to return a stream, and is only forced on demand.

7.5.28.2 SRFI-41 Stream Primitives

This library provides eight operators: constructors for `stream-null` and `stream-pair`s, type predicates for streams and the two kinds of streams, accessors for both fields of a `stream-pair`, and a lambda that creates procedures that return streams.

`stream-null` [Scheme Variable]
: A promise that, when forced, is a single object, distinguishable from all other objects, that represents the null stream. `stream-null` is immutable and unique.

`stream-cons` *object-expr stream-expr* [Scheme Syntax]
: Creates a newly-allocated stream containing a promise that, when forced, is a `stream-pair` with *object-expr* in its `stream-car` and *stream-expr* in its `stream-cdr`. Neither *object-expr* nor *stream-expr* is evaluated when `stream-cons` is called.

  Once created, a `stream-pair` is immutable; there is no `stream-set-car!` or `stream-set-cdr!` that modifies an existing stream-pair. There is no dotted-pair or improper stream as with lists.

`stream?` *object* [Scheme Procedure]
: Returns true if *object* is a stream, otherwise returns false. If *object* is a stream, its promise will not be forced. If `(stream? obj)` returns true, then one of `(stream-null? obj)` or `(stream-pair? obj)` will return true and the other will return false.

`stream-null?` *object* [Scheme Procedure]
: Returns true if *object* is the distinguished null stream, otherwise returns false. If *object* is a stream, its promise will be forced.

`stream-pair?` *object* [Scheme Procedure]
: Returns true if *object* is a `stream-pair` constructed by `stream-cons`, otherwise returns false. If *object* is a stream, its promise will be forced.

`stream-car` *stream* [Scheme Procedure]
: Returns the object stored in the `stream-car` of *stream*. An error is signalled if the argument is not a `stream-pair`. This causes the *object-expr* passed to `stream-cons` to be evaluated if it had not yet been; the value is cached in case it is needed again.

`stream-cdr` *stream* [Scheme Procedure]
: Returns the stream stored in the `stream-cdr` of *stream*. An error is signalled if the argument is not a `stream-pair`.

`stream-lambda` *formals body . . .* [Scheme Syntax]
: Creates a procedure that returns a promise to evaluate the *body* of the procedure. The last *body* expression to be evaluated must yield a stream. As with normal `lambda`, *formals* may be a single variable name, in which case all the formal arguments are collected into a single list, or a list of variable names, which may be null if there are no arguments, proper if there are an exact number of arguments, or dotted if a fixed number of arguments is to be followed by zero or more arguments collected into a list. *Body* must contain at least one expression, and may contain internal definitions preceding any expressions to be evaluated.

```
(define strm123
  (stream-cons 1
    (stream-cons 2
      (stream-cons 3
        stream-null))))

(stream-car strm123) ⇒ 1
(stream-car (stream-cdr strm123) ⇒ 2

(stream-pair?
  (stream-cdr
    (stream-cons (/ 1 0) stream-null))) ⇒ #f

(stream? (list 1 2 3)) ⇒ #f

(define iter
  (stream-lambda (f x)
    (stream-cons x (iter f (f x)))))

(define nats (iter (lambda (x) (+ x 1)) 0))

(stream-car (stream-cdr nats)) ⇒ 1

(define stream-add
  (stream-lambda (s1 s2)
    (stream-cons
      (+ (stream-car s1) (stream-car s2))
      (stream-add (stream-cdr s1)
                  (stream-cdr s2)))))

(define evens (stream-add nats nats))

(stream-car evens) ⇒ 0
(stream-car (stream-cdr evens)) ⇒ 2
(stream-car (stream-cdr (stream-cdr evens))) ⇒ 4
```

7.5.28.3 SRFI-41 Stream Library

`define-stream` (*name args ...*) *body ...* [Scheme Syntax]

Creates a procedure that returns a stream, and may appear anywhere a normal `define` may appear, including as an internal definition. It may contain internal definitions of its own. The defined procedure takes arguments in the same way as `stream-lambda`. `define-stream` is syntactic sugar on `stream-lambda`; see also `stream-let`, which is also a sugaring of `stream-lambda`.

A simple version of `stream-map` that takes only a single input stream calls itself recursively:

```
(define-stream (stream-map proc strm)
```

```
(if (stream-null? strm)
    stream-null
    (stream-cons
      (proc (stream-car strm))
      (stream-map proc (stream-cdr strm))))))
```

`list->stream` *list* [Scheme Procedure]

Returns a newly-allocated stream containing the elements from *list*.

`port->stream` [*port*] [Scheme Procedure]

Returns a newly-allocated stream containing in its elements the characters on the port. If *port* is not given it defaults to the current input port. The returned stream has finite length and is terminated by `stream-null`.

It looks like one use of `port->stream` would be this:

```
(define s ;wrong!
  (with-input-from-file filename
    (lambda () (port->stream))))
```

But that fails, because `with-input-from-file` is eager, and closes the input port prematurely, before the first character is read. To read a file into a stream, say:

```
(define-stream (file->stream filename)
  (let ((p (open-input-file filename)))
    (stream-let loop ((c (read-char p)))
      (if (eof-object? c)
          (begin (close-input-port p)
                 stream-null)
          (stream-cons c
            (loop (read-char p)))))))
```

`stream` *object-expr ...* [Scheme Syntax]

Creates a newly-allocated stream containing in its elements the objects, in order. The *object-expr*s are evaluated when they are accessed, not when the stream is created. If no objects are given, as in (stream), the null stream is returned. See also `list->stream`.

```
(define strm123 (stream 1 2 3))

; (/ 1 0) not evaluated when stream is created
(define s (stream 1 (/ 1 0) -1))
```

`stream->list` [*n*] *stream* [Scheme Procedure]

Returns a newly-allocated list containing in its elements the first *n* items in *stream*. If *stream* has less than *n* items, all the items in the stream will be included in the returned list. If *n* is not given it defaults to infinity, which means that unless *stream* is finite `stream->list` will never return.

```
(stream->list 10
  (stream-map (lambda (x) (* x x))
    (stream-from 0)))
  ⇒ (0 1 4 9 16 25 36 49 64 81)
```

`stream-append` *stream ...* [Scheme Procedure]
Returns a newly-allocated stream containing in its elements those elements contained in its input *stream*s, in order of input. If any of the input streams is infinite, no elements of any of the succeeding input streams will appear in the output stream. See also `stream-concat`.

`stream-concat` *stream* [Scheme Procedure]
Takes a *stream* consisting of one or more streams and returns a newly-allocated stream containing all the elements of the input streams. If any of the streams in the input *stream* is infinite, any remaining streams in the input stream will never appear in the output stream. See also `stream-append`.

`stream-constant` *object ...* [Scheme Procedure]
Returns a newly-allocated stream containing in its elements the *object*s, repeating in succession forever.

```
(stream-constant 1) ⇒ 1 1 1 ...
(stream-constant #t #f) ⇒ #t #f #t #f #t #f ...
```

`stream-drop` *n stream* [Scheme Procedure]
Returns the suffix of the input *stream* that starts at the next element after the first *n* elements. The output stream shares structure with the input *stream*; thus, promises forced in one instance of the stream are also forced in the other instance of the stream. If the input *stream* has less than *n* elements, `stream-drop` returns the null stream. See also `stream-take`.

`stream-drop-while` *pred stream* [Scheme Procedure]
Returns the suffix of the input *stream* that starts at the first element *x* for which `(pred x)` returns false. The output stream shares structure with the input *stream*. See also `stream-take-while`.

`stream-filter` *pred stream* [Scheme Procedure]
Returns a newly-allocated stream that contains only those elements *x* of the input *stream* which satisfy the predicate `pred`.

```
(stream-filter odd? (stream-from 0))
   ⇒ 1 3 5 7 9 ...
```

`stream-fold` *proc base stream* [Scheme Procedure]
Applies a binary procedure *proc* to *base* and the first element of *stream* to compute a new *base*, then applies the procedure to the new *base* and the next element of *stream* to compute a succeeding *base*, and so on, accumulating a value that is finally returned as the value of `stream-fold` when the end of the stream is reached. *stream* must be finite, or `stream-fold` will enter an infinite loop. See also `stream-scan`, which is similar to `stream-fold`, but useful for infinite streams. For readers familiar with other functional languages, this is a left-fold; there is no corresponding right-fold, since right-fold relies on finite streams that are fully-evaluated, in which case they may as well be converted to a list.

`stream-for-each` *proc stream . . .* [Scheme Procedure]
Applies *proc* element-wise to corresponding elements of the input *stream*s for side-effects; it returns nothing. `stream-for-each` stops as soon as any of its input streams is exhausted.

`stream-from` *first* [*step*] [Scheme Procedure]
Creates a newly-allocated stream that contains *first* as its first element and increments each succeeding element by *step*. If *step* is not given it defaults to 1. *first* and *step* may be of any numeric type. `stream-from` is frequently useful as a generator in `stream-of` expressions. See also `stream-range` for a similar procedure that creates finite streams.

`stream-iterate` *proc base* [Scheme Procedure]
Creates a newly-allocated stream containing *base* in its first element and applies *proc* to each element in turn to determine the succeeding element. See also `stream-unfold` and `stream-unfolds`.

`stream-length` *stream* [Scheme Procedure]
Returns the number of elements in the *stream*; it does not evaluate its elements. `stream-length` may only be used on finite streams; it enters an infinite loop with infinite streams.

`stream-let` *tag* ((*var expr*) . . .) *body . . .* [Scheme Syntax]
Creates a local scope that binds each variable to the value of its corresponding expression. It additionally binds *tag* to a procedure which takes the bound variables as arguments and *body* as its defining expressions, binding the *tag* with `stream-lambda`. *tag* is in scope within body, and may be called recursively. When the expanded expression defined by the `stream-let` is evaluated, `stream-let` evaluates the expressions in its *body* in an environment containing the newly-bound variables, returning the value of the last expression evaluated, which must yield a stream.

`stream-let` provides syntactic sugar on `stream-lambda`, in the same manner as normal `let` provides syntactic sugar on normal `lambda`. However, unlike normal `let`, the *tag* is required, not optional, because unnamed `stream-let` is meaningless.

For example, `stream-member` returns the first `stream-pair` of the input *strm* with a `stream-car` *x* that satisfies `(eql? obj x)`, or the null stream if *x* is not present in *strm*.

```
(define-stream (stream-member eql? obj strm)
  (stream-let loop ((strm strm))
    (cond ((stream-null? strm) strm)
          ((eql? obj (stream-car strm)) strm)
          (else (loop (stream-cdr strm))))))
```

`stream-map` *proc stream . . .* [Scheme Procedure]
Applies *proc* element-wise to corresponding elements of the input *stream*s, returning a newly-allocated stream containing elements that are the results of those procedure applications. The output stream has as many elements as the minimum-length input stream, and may be infinite.

`stream-match` *stream clause ...* [Scheme Syntax]

Provides pattern-matching for streams. The input *stream* is an expression that evaluates to a stream. Clauses are of the form `(pattern [fender] expression)`, consisting of a *pattern* that matches a stream of a particular shape, an optional *fender* that must succeed if the pattern is to match, and an *expression* that is evaluated if the pattern matches. There are four types of patterns:

- () matches the null stream.
- (*pat0 pat1 ...*) matches a finite stream with length exactly equal to the number of pattern elements.
- (*pat0 pat1 ... . pat-rest*) matches an infinite stream, or a finite stream with length at least as great as the number of pattern elements before the literal dot.
- *pat* matches an entire stream. Should always appear last in the list of clauses; it's not an error to appear elsewhere, but subsequent clauses could never match.

Each pattern element may be either:

- An identifier, which matches any stream element. Additionally, the value of the stream element is bound to the variable named by the identifier, which is in scope in the *fender* and *expression* of the corresponding *clause*. Each identifier in a single pattern must be unique.
- A literal underscore (`_`), which matches any stream element but creates no bindings.

The *pattern*s are tested in order, left-to-right, until a matching pattern is found; if *fender* is present, it must evaluate to a true value for the match to be successful. Pattern variables are bound in the corresponding *fender* and *expression*. Once the matching *pattern* is found, the corresponding *expression* is evaluated and returned as the result of the match. An error is signaled if no pattern matches the input *stream*.

`stream-match` is often used to distinguish null streams from non-null streams, binding *head* and *tail*:

```
(define (len strm)
  (stream-match strm
    (() 0)
    ((head . tail) (+ 1 (len tail)))))
```

Fenders can test the common case where two stream elements must be identical; the `else` pattern is an identifier bound to the entire stream, not a keyword as in `cond`.

```
(stream-match strm
  ((x y . _) (equal? x y) 'ok)
  (else 'error))
```

A more complex example uses two nested matchers to match two different stream arguments; `(stream-merge lt? . strms)` stably merges two or more streams ordered by the `lt?` predicate:

```
(define-stream (stream-merge lt? . strms)
  (define-stream (merge xx yy)
    (stream-match xx (() yy) ((x . xs)
      (stream-match yy (() xx) ((y . ys)
```

```
            (if (lt? y x)
                (stream-cons y (merge xx ys))
                (stream-cons x (merge xs yy))))))))
      (stream-let loop ((strms strms))
        (cond ((null? strms) stream-null)
              ((null? (cdr strms)) (car strms))
              (else (merge (car strms)
                           (apply stream-merge lt?
                             (cdr strms)))))))
```

`stream-of` *expr clause ...* [Scheme Syntax]

Provides the syntax of stream comprehensions, which generate streams by means of looping expressions. The result is a stream of objects of the type returned by *expr*. There are four types of clauses:

- (*var* `in` *stream-expr*) loops over the elements of *stream-expr*, in order from the start of the stream, binding each element of the stream in turn to *var*. `stream-from` and `stream-range` are frequently useful as generators for *stream-expr*.
- (*var* `is` *expr*) binds *var* to the value obtained by evaluating *expr*.
- (*pred expr*) includes in the output stream only those elements *x* which satisfy the predicate *pred*.

The scope of variables bound in the stream comprehension is the clauses to the right of the binding clause (but not the binding clause itself) plus the result expression.

When two or more generators are present, the loops are processed as if they are nested from left to right; that is, the rightmost generator varies fastest. A consequence of this is that only the first generator may be infinite and all subsequent generators must be finite. If no generators are present, the result of a stream comprehension is a stream containing the result expression; thus, '`(stream-of 1)`' produces a finite stream containing only the element 1.

```
(stream-of (* x x)
  (x in (stream-range 0 10))
  (even? x))
  ⇒ 0 4 16 36 64

(stream-of (list a b)
  (a in (stream-range 1 4))
  (b in (stream-range 1 3)))
  ⇒ (1 1) (1 2) (2 1) (2 2) (3 1) (3 2)

(stream-of (list i j)
  (i in (stream-range 1 5))
  (j in (stream-range (+ i 1) 5)))
  ⇒ (1 2) (1 3) (1 4) (2 3) (2 4) (3 4)
```

`stream-range` *first past* [*step*] [Scheme Procedure]

Creates a newly-allocated stream that contains *first* as its first element and increments each succeeding element by *step*. The stream is finite and ends before *past*, which

is not an element of the stream. If *step* is not given it defaults to 1 if *first* is less than past and -1 otherwise. *first*, *past* and *step* may be of any real numeric type. `stream-range` is frequently useful as a generator in `stream-of` expressions. See also `stream-from` for a similar procedure that creates infinite streams.

```
(stream-range 0 10) ⇒ 0 1 2 3 4 5 6 7 8 9
(stream-range 0 10 2) ⇒ 0 2 4 6 8
```

Successive elements of the stream are calculated by adding *step* to *first*, so if any of *first*, *past* or *step* are inexact, the length of the output stream may differ from `(ceiling (- (/ (- past first) step) 1)`.

`stream-ref` *stream n* [Scheme Procedure]

Returns the *n*th element of stream, counting from zero. An error is signaled if *n* is greater than or equal to the length of stream.

```
(define (fact n)
  (stream-ref
    (stream-scan * 1 (stream-from 1))
    n))
```

`stream-reverse` *stream* [Scheme Procedure]

Returns a newly-allocated stream containing the elements of the input *stream* but in reverse order. `stream-reverse` may only be used with finite streams; it enters an infinite loop with infinite streams. `stream-reverse` does not force evaluation of the elements of the stream.

`stream-scan` *proc base stream* [Scheme Procedure]

Accumulates the partial folds of an input *stream* into a newly-allocated output stream. The output stream is the *base* followed by `(stream-fold proc base (stream-take i stream))` for each of the first *i* elements of *stream*.

```
(stream-scan + 0 (stream-from 1))
  ⇒ (stream 0 1 3 6 10 15 ...)

(stream-scan * 1 (stream-from 1))
  ⇒ (stream 1 1 2 6 24 120 ...)
```

`stream-take` *n stream* [Scheme Procedure]

Returns a newly-allocated stream containing the first *n* elements of the input *stream*. If the input *stream* has less than *n* elements, so does the output stream. See also `stream-drop`.

`stream-take-while` *pred stream* [Scheme Procedure]

Takes a predicate and a `stream` and returns a newly-allocated stream containing those elements `x` that form the maximal prefix of the input stream which satisfy *pred*. See also `stream-drop-while`.

`stream-unfold` *map pred gen base* [Scheme Procedure]

The fundamental recursive stream constructor. It constructs a stream by repeatedly applying *gen* to successive values of *base*, in the manner of `stream-iterate`, then applying *map* to each of the values so generated, appending each of the mapped

values to the output stream as long as `(pred? base)` returns a true value. See also `stream-iterate` and `stream-unfolds`.

The expression below creates the finite stream '`0 1 4 9 16 25 36 49 64 81`'. Initially the *base* is 0, which is less than 10, so *map* squares the *base* and the mapped value becomes the first element of the output stream. Then *gen* increments the *base* by 1, so it becomes 1; this is less than 10, so *map* squares the new *base* and 1 becomes the second element of the output stream. And so on, until the base becomes 10, when *pred* stops the recursion and stream-null ends the output stream.

```
(stream-unfold
  (lambda (x) (expt x 2)) ; map
  (lambda (x) (< x 10))   ; pred?
  (lambda (x) (+ x 1))    ; gen
  0)                      ; base
```

`stream-unfolds` *proc seed* [Scheme Procedure]

Returns *n* newly-allocated streams containing those elements produced by successive calls to the generator *proc*, which takes the current *seed* as its argument and returns *n*+1 values

(*proc seed*) ⇒ *seed result_0 ... result_n-1*

where the returned *seed* is the input *seed* to the next call to the generator and *result_i* indicates how to produce the next element of the *i*th result stream:

- (*value*): *value* is the next car of the result stream.
- `#f`: no value produced by this iteration of the generator *proc* for the result stream.
- (): the end of the result stream.

It may require multiple calls of *proc* to produce the next element of any particular result stream. See also `stream-iterate` and `stream-unfold`.

```
(define (stream-partition pred? strm)
  (stream-unfolds
    (lambda (s)
      (if (stream-null? s)
          (values s '() '())
          (let ((a (stream-car s))
                (d (stream-cdr s)))
            (if (pred? a)
                (values d (list a) #f)
                (values d #f (list a))))))
    strm))

(call-with-values
  (lambda ()
    (stream-partition odd?
      (stream-range 1 6)))
  (lambda (odds evens)
    (list (stream->list odds)
```

```
        (stream->list evens))))
⇒ ((1 3 5) (2 4))
```

`stream-zip` *stream ...* [Scheme Procedure]

Returns a newly-allocated stream in which each element is a list (not a stream) of the corresponding elements of the input *stream*s. The output stream is as long as the shortest input *stream*, if any of the input *stream*s is finite, or is infinite if all the input *stream*s are infinite.

7.5.29 SRFI-42 - Eager Comprehensions

See the specification of SRFI-42 (`http://srfi.schemers.org/srfi-42/srfi-42.html`).

7.5.30 SRFI-43 - Vector Library

This subsection is based on the specification of SRFI-43 (`http://srfi.schemers.org/srfi-43/srfi-43.html`) by Taylor Campbell.

SRFI-43 implements a comprehensive library of vector operations. It can be made available with:

```
(use-modules (srfi srfi-43))
```

7.5.30.1 SRFI-43 Constructors

`make-vector` *size* [*fill*] [Scheme Procedure]

Create and return a vector of size *size*, optionally filling it with *fill*. The default value of *fill* is unspecified.

```
(make-vector 5 3) ⇒ #(3 3 3 3 3)
```

`vector` *x ...* [Scheme Procedure]

Create and return a vector whose elements are *x ...*.

```
(vector 0 1 2 3 4) ⇒ #(0 1 2 3 4)
```

`vector-unfold` *f length initial-seed ...* [Scheme Procedure]

The fundamental vector constructor. Create a vector whose length is *length* and iterates across each index k from 0 up to *length* - 1, applying *f* at each iteration to the current index and current seeds, in that order, to receive n + 1 values: the element to put in the kth slot of the new vector, and n new seeds for the next iteration. It is an error for the number of seeds to vary between iterations.

```
(vector-unfold (lambda (i x) (values x (- x 1)))
               10 0)
⇒ #(0 -1 -2 -3 -4 -5 -6 -7 -8 -9)

(vector-unfold values 10)
⇒ #(0 1 2 3 4 5 6 7 8 9)
```

`vector-unfold-right` *f length initial-seed ...* [Scheme Procedure]

Like `vector-unfold`, but it uses *f* to generate elements from right-to-left, rather than left-to-right.

```
(vector-unfold-right (lambda (i x) (values x (+ x 1)))
```

```
                          10 0)
⇒ #(9 8 7 6 5 4 3 2 1 0)
```

`vector-copy` *vec* [*start* [*end* [*fill*]]] [Scheme Procedure]
Allocate a new vector whose length is *end* - *start* and fills it with elements from *vec*, taking elements from *vec* starting at index *start* and stopping at index *end*. *start* defaults to 0 and *end* defaults to the value of `(vector-length vec)`. If *end* extends beyond the length of *vec*, the slots in the new vector that obviously cannot be filled by elements from *vec* are filled with *fill*, whose default value is unspecified.

```
(vector-copy '#(a b c d e f g h i))
⇒ #(a b c d e f g h i)

(vector-copy '#(a b c d e f g h i) 6)
⇒ #(g h i)

(vector-copy '#(a b c d e f g h i) 3 6)
⇒ #(d e f)

(vector-copy '#(a b c d e f g h i) 6 12 'x)
⇒ #(g h i x x x)
```

`vector-reverse-copy` *vec* [*start* [*end*]] [Scheme Procedure]
Like `vector-copy`, but it copies the elements in the reverse order from *vec*.

```
(vector-reverse-copy '#(5 4 3 2 1 0) 1 5)
⇒ #(1 2 3 4)
```

`vector-append` *vec* ... [Scheme Procedure]
Return a newly allocated vector that contains all elements in order from the subsequent locations in *vec* ....

```
(vector-append '#(a) '#(b c d))
⇒ #(a b c d)
```

`vector-concatenate` *list-of-vectors* [Scheme Procedure]
Append each vector in *list-of-vectors*. Equivalent to `(apply vector-append list-of-vectors)`.

```
(vector-concatenate '(#(a b) #(c d)))
⇒ #(a b c d)
```

7.5.30.2 SRFI-43 Predicates

`vector?` *obj* [Scheme Procedure]
Return true if *obj* is a vector, else return false.

`vector-empty?` *vec* [Scheme Procedure]
Return true if *vec* is empty, i.e. its length is 0, else return false.

`vector=` *elt=? vec* ... [Scheme Procedure]
Return true if the vectors *vec* ... have equal lengths and equal elements according to *elt=?*. *elt=?* is always applied to two arguments. Element comparison must be

consistent with `eq?` in the following sense: if `(eq? a b)` returns true, then `(elt=? a b)` must also return true. The order in which comparisons are performed is unspecified.

7.5.30.3 SRFI-43 Selectors

`vector-ref` *vec i* [Scheme Procedure]
Return the element at index *i* in *vec*. Indexing is based on zero.

`vector-length` *vec* [Scheme Procedure]
Return the length of *vec*.

7.5.30.4 SRFI-43 Iteration

`vector-fold` *kons knil vec1 vec2 . . .* [Scheme Procedure]
The fundamental vector iterator. *kons* is iterated over each index in all of the vectors, stopping at the end of the shortest; *kons* is applied as

```
(kons i state (vector-ref vec1 i) (vector-ref vec2 i) ...)
```

where *state* is the current state value, and *i* is the current index. The current state value begins with *knil*, and becomes whatever *kons* returned at the respective iteration. The iteration is strictly left-to-right.

`vector-fold-right` *kons knil vec1 vec2 . . .* [Scheme Procedure]
Similar to `vector-fold`, but it iterates right-to-left instead of left-to-right.

`vector-map` *f vec1 vec2 . . .* [Scheme Procedure]
Return a new vector of the shortest size of the vector arguments. Each element at index i of the new vector is mapped from the old vectors by

```
(f i (vector-ref vec1 i) (vector-ref vec2 i) ...)
```

The dynamic order of application of *f* is unspecified.

`vector-map!` *f vec1 vec2 . . .* [Scheme Procedure]
Similar to `vector-map`, but rather than mapping the new elements into a new vector, the new mapped elements are destructively inserted into *vec1*. The dynamic order of application of *f* is unspecified.

`vector-for-each` *f vec1 vec2 . . .* [Scheme Procedure]
Call `(f i (vector-ref vec1 i) (vector-ref vec2 i) ...)` for each index i less than the length of the shortest vector passed. The iteration is strictly left-to-right.

`vector-count` *pred? vec1 vec2 . . .* [Scheme Procedure]
Count the number of parallel elements in the vectors that satisfy *pred?*, which is applied, for each index i less than the length of the smallest vector, to i and each parallel element in the vectors at that index, in order.

```
(vector-count (lambda (i elt) (even? elt))
              '#(3 1 4 1 5 9 2 5 6))
⇒ 3
(vector-count (lambda (i x y) (< x y))
              '#(1 3 6 9) '#(2 4 6 8 10 12))
⇒ 2
```

7.5.30.5 SRFI-43 Searching

`vector-index` *pred? vec1 vec2 ...* [Scheme Procedure]
Find and return the index of the first elements in *vec1 vec2 ...* that satisfy *pred?*. If no matching element is found by the end of the shortest vector, return `#f`.

```
(vector-index even? '#(3 1 4 1 5 9))
⇒ 2
(vector-index < '#(3 1 4 1 5 9 2 5 6) '#(2 7 1 8 2))
⇒ 1
(vector-index = '#(3 1 4 1 5 9 2 5 6) '#(2 7 1 8 2))
⇒ #f
```

`vector-index-right` *pred? vec1 vec2 ...* [Scheme Procedure]
Like `vector-index`, but it searches right-to-left, rather than left-to-right. Note that the SRFI 43 specification requires that all the vectors must have the same length, but both the SRFI 43 reference implementation and Guile's implementation allow vectors with unequal lengths, and start searching from the last index of the shortest vector.

`vector-skip` *pred? vec1 vec2 ...* [Scheme Procedure]
Find and return the index of the first elements in *vec1 vec2 ...* that do not satisfy *pred?*. If no matching element is found by the end of the shortest vector, return `#f`. Equivalent to `vector-index` but with the predicate inverted.

```
(vector-skip number? '#(1 2 a b 3 4 c d)) ⇒ 2
```

`vector-skip-right` *pred? vec1 vec2 ...* [Scheme Procedure]
Like `vector-skip`, but it searches for a non-matching element right-to-left, rather than left-to-right. Note that the SRFI 43 specification requires that all the vectors must have the same length, but both the SRFI 43 reference implementation and Guile's implementation allow vectors with unequal lengths, and start searching from the last index of the shortest vector.

`vector-binary-search` *vec value cmp* [*start* [*end*]] [Scheme Procedure]
Find and return an index of *vec* between *start* and *end* whose value is *value* using a binary search. If no matching element is found, return `#f`. The default *start* is 0 and the default *end* is the length of *vec*.

cmp must be a procedure of two arguments such that `(cmp a b)` returns a negative integer if $a < b$, a positive integer if $a > b$, or zero if $a = b$. The elements of *vec* must be sorted in non-decreasing order according to *cmp*.

Note that SRFI 43 does not document the *start* and *end* arguments, but both its reference implementation and Guile's implementation support them.

```
(define (char-cmp c1 c2)
  (cond ((char<? c1 c2) -1)
        ((char>? c1 c2) 1)
        (else 0)))

(vector-binary-search '#(#\a #\b #\c #\d #\e #\f #\g #\h)
                      #\g
```

```
                                char-cmp)
      ⇒ 6
```

`vector-any` *pred? vec1 vec2 ...* [Scheme Procedure]
Find the first parallel set of elements from *vec1 vec2 ...* for which *pred?* returns a true value. If such a parallel set of elements exists, `vector-any` returns the value that *pred?* returned for that set of elements. The iteration is strictly left-to-right.

`vector-every` *pred? vec1 vec2 ...* [Scheme Procedure]
If, for every index i between 0 and the length of the shortest vector argument, the set of elements `(vector-ref vec1 i) (vector-ref vec2 i) ...` satisfies *pred?*, `vector-every` returns the value that *pred?* returned for the last set of elements, at the last index of the shortest vector. Otherwise it returns `#f`. The iteration is strictly left-to-right.

7.5.30.6 SRFI-43 Mutators

`vector-set!` *vec i value* [Scheme Procedure]
Assign the contents of the location at *i* in *vec* to *value*.

`vector-swap!` *vec i j* [Scheme Procedure]
Swap the values of the locations in *vec* at *i* and *j*.

`vector-fill!` *vec fill* [*start* [*end*]] [Scheme Procedure]
Assign the value of every location in *vec* between *start* and *end* to *fill*. *start* defaults to 0 and *end* defaults to the length of *vec*.

`vector-reverse!` *vec* [*start* [*end*]] [Scheme Procedure]
Destructively reverse the contents of *vec* between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *vec*.

`vector-copy!` *target tstart source* [*sstart* [*send*]] [Scheme Procedure]
Copy a block of elements from *source* to *target*, both of which must be vectors, starting in *target* at *tstart* and starting in *source* at *sstart*, ending when (*send* - *sstart*) elements have been copied. It is an error for *target* to have a length less than (*tstart* + *send* - *sstart*). *sstart* defaults to 0 and *send* defaults to the length of *source*.

`vector-reverse-copy!` *target tstart source* [*sstart* [*send*]] [Scheme Procedure]
Like `vector-copy!`, but this copies the elements in the reverse order. It is an error if *target* and *source* are identical vectors and the *target* and *source* ranges overlap; however, if *tstart* = *sstart*, `vector-reverse-copy!` behaves as `(vector-reverse! target tstart send)` would.

7.5.30.7 SRFI-43 Conversion

`vector->list` *vec* [*start* [*end*]] [Scheme Procedure]
Return a newly allocated list containing the elements in *vec* between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *vec*.

`reverse-vector->list` *vec* [*start* [*end*]] [Scheme Procedure]
Like `vector->list`, but the resulting list contains the specified range of elements of *vec* in reverse order.

`list->vector` *proper-list* [*start* [*end*]] [Scheme Procedure]
Return a newly allocated vector of the elements from *proper-list* with indices between *start* and *end*. *start* defaults to 0 and *end* defaults to the length of *proper-list*. Note that SRFI 43 does not document the *start* and *end* arguments, but both its reference implementation and Guile's implementation support them.

`reverse-list->vector` *proper-list* [*start* [*end*]] [Scheme Procedure]
Like `list->vector`, but the resulting vector contains the specified range of elements of *proper-list* in reverse order. Note that SRFI 43 does not document the *start* and *end* arguments, but both its reference implementation and Guile's implementation support them.

7.5.31 SRFI-45 - Primitives for Expressing Iterative Lazy Algorithms

This subsection is based on the specification of SRFI-45 (`http://srfi.schemers.org/srfi-45/srfi-45.html`) written by André van Tonder.

Lazy evaluation is traditionally simulated in Scheme using `delay` and `force`. However, these primitives are not powerful enough to express a large class of lazy algorithms that are iterative. Indeed, it is folklore in the Scheme community that typical iterative lazy algorithms written using delay and force will often require unbounded memory.

This SRFI provides set of three operations: {`lazy`, `delay`, `force`}, which allow the programmer to succinctly express lazy algorithms while retaining bounded space behavior in cases that are properly tail-recursive. A general recipe for using these primitives is provided. An additional procedure `eager` is provided for the construction of eager promises in cases where efficiency is a concern.

Although this SRFI redefines `delay` and `force`, the extension is conservative in the sense that the semantics of the subset {`delay`, `force`} in isolation (i.e., as long as the program does not use `lazy`) agrees with that in R5RS. In other words, no program that uses the R5RS definitions of delay and force will break if those definition are replaced by the SRFI-45 definitions of delay and force.

Guile also adds `promise?` to the list of exports, which is not part of the official SRFI-45.

`promise?` *obj* [Scheme Procedure]
Return true if *obj* is an SRFI-45 promise, otherwise return false.

`delay` *expression* [Scheme Syntax]
Takes an expression of arbitrary type *a* and returns a promise of type `(Promise a)` which at some point in the future may be asked (by the `force` procedure) to evaluate the expression and deliver the resulting value.

`lazy` *expression* [Scheme Syntax]
Takes an expression of type `(Promise a)` and returns a promise of type `(Promise a)` which at some point in the future may be asked (by the `force` procedure) to evaluate the expression and deliver the resulting promise.

`force` *expression* [Scheme Procedure]
Takes an argument of type `(Promise a)` and returns a value of type *a* as follows: If a value of type *a* has been computed for the promise, this value is returned. Otherwise,

the promise is first evaluated, then overwritten by the obtained promise or value, and then force is again applied (iteratively) to the promise.

`eager` *expression* [Scheme Procedure]

Takes an argument of type *a* and returns a value of type `(Promise a)`. As opposed to `delay`, the argument is evaluated eagerly. Semantically, writing `(eager expression)` is equivalent to writing

```
(let ((value expression)) (delay value)).
```

However, the former is more efficient since it does not require unnecessary creation and evaluation of thunks. We also have the equivalence

```
(delay expression) = (lazy (eager expression))
```

The following reduction rules may be helpful for reasoning about these primitives. However, they do not express the memoization and memory usage semantics specified above:

```
(force (delay expression)) -> expression
(force (lazy  expression)) -> (force expression)
(force (eager value))      -> value
```

Correct usage

We now provide a general recipe for using the primitives {`lazy`, `delay`, `force`} to express lazy algorithms in Scheme. The transformation is best described by way of an example: Consider the stream-filter algorithm, expressed in a hypothetical lazy language as

```
(define (stream-filter p? s)
  (if (null? s) '()
      (let ((h (car s))
            (t (cdr s)))
        (if (p? h)
            (cons h (stream-filter p? t))
            (stream-filter p? t)))))
```

This algorithm can be expressed as follows in Scheme:

```
(define (stream-filter p? s)
  (lazy
     (if (null? (force s)) (delay '())
         (let ((h (car (force s)))
               (t (cdr (force s))))
           (if (p? h)
               (delay (cons h (stream-filter p? t)))
               (stream-filter p? t))))))
```

In other words, we

- wrap all constructors (e.g., `'()`, `cons`) with `delay`,
- apply `force` to arguments of deconstructors (e.g., `car`, `cdr` and `null?`),
- wrap procedure bodies with `(lazy ...)`.

7.5.32 SRFI-46 Basic syntax-rules Extensions

Guile's core `syntax-rules` supports the extensions specified by SRFI-46/R7RS. Tail patterns have been supported since at least Guile 2.0, and custom ellipsis identifiers have been supported since Guile 2.0.10. See Section 6.10.2 [Syntax Rules], page 263.

7.5.33 SRFI-55 - Requiring Features

SRFI-55 provides `require-extension` which is a portable mechanism to load selected SRFI modules. This is implemented in the Guile core, there's no module needed to get SRFI-55 itself.

`require-extension` *clause1 clause2 ...* [library syntax]

Require the features of *clause1 clause2 ...* , throwing an error if any are unavailable.

A *clause* is of the form `(identifier arg...)`. The only *identifier* currently supported is `srfi` and the arguments are SRFI numbers. For example to get SRFI-1 and SRFI-6,

```
(require-extension (srfi 1 6))
```

`require-extension` can only be used at the top-level.

A Guile-specific program can simply `use-modules` to load SRFIs not already in the core, `require-extension` is for programs designed to be portable to other Scheme implementations.

7.5.34 SRFI-60 - Integers as Bits

This SRFI provides various functions for treating integers as bits and for bitwise manipulations. These functions can be obtained with,

```
(use-modules (srfi srfi-60))
```

Integers are treated as infinite precision twos-complement, the same as in the core logical functions (see Section 6.6.2.13 [Bitwise Operations], page 125). And likewise bit indexes start from 0 for the least significant bit. The following functions in this SRFI are already in the Guile core,

`logand`, `logior`, `logxor`, `lognot`, `logtest`, `logcount`, `integer-length`, `logbit?`, `ash`

`bitwise-and` *n1 ...* [Function]
`bitwise-ior` *n1 ...* [Function]
`bitwise-xor` *n1 ...* [Function]
`bitwise-not` *n* [Function]
`any-bits-set?` *j k* [Function]
`bit-set?` *index n* [Function]
`arithmetic-shift` *n count* [Function]
`bit-field` *n start end* [Function]
`bit-count` *n* [Function]

Aliases for `logand`, `logior`, `logxor`, `lognot`, `logtest`, `logbit?`, `ash`, `bit-extract` and `logcount` respectively.

Note that the name `bit-count` conflicts with `bit-count` in the core (see Section 6.6.11 [Bit Vectors], page 191).

`bitwise-if` *mask n1 n0* [Function]
`bitwise-merge` *mask n1 n0* [Function]
Return an integer with bits selected from *n1* and *n0* according to *mask*. Those bits where *mask* has 1s are taken from *n1*, and those where *mask* has 0s are taken from *n0*.

```
(bitwise-if 3 #b0101 #b1010) ⇒ 9
```

`log2-binary-factors` *n* [Function]
`first-set-bit` *n* [Function]
Return a count of how many factors of 2 are present in *n*. This is also the bit index of the lowest 1 bit in *n*. If *n* is 0, the return is −1.

```
(log2-binary-factors 6) ⇒ 1
(log2-binary-factors -8) ⇒ 3
```

`copy-bit` *index n newbit* [Function]
Return *n* with the bit at *index* set according to *newbit*. *newbit* should be `#t` to set the bit to 1, or `#f` to set it to 0. Bits other than at *index* are unchanged in the return.

```
(copy-bit 1 #b0101 #t) ⇒ 7
```

`copy-bit-field` *n newbits start end* [Function]
Return *n* with the bits from *start* (inclusive) to *end* (exclusive) changed to the value *newbits*.

The least significant bit in *newbits* goes to *start*, the next to *start* + 1, etc. Anything in *newbits* past the *end* given is ignored.

```
(copy-bit-field #b10000 #b11 1 3) ⇒ #b10110
```

`rotate-bit-field` *n count start end* [Function]
Return *n* with the bit field from *start* (inclusive) to *end* (exclusive) rotated upwards by *count* bits.

count can be positive or negative, and it can be more than the field width (it'll be reduced modulo the width).

```
(rotate-bit-field #b0110 2 1 4) ⇒ #b1010
```

`reverse-bit-field` *n start end* [Function]
Return *n* with the bits from *start* (inclusive) to *end* (exclusive) reversed.

```
(reverse-bit-field #b101001 2 4) ⇒ #b100101
```

`integer->list` *n* [*len*] [Function]
Return bits from *n* in the form of a list of `#t` for 1 and `#f` for 0. The least significant *len* bits are returned, and the first list element is the most significant of those bits. If *len* is not given, the default is `(integer-length n)` (see Section 6.6.2.13 [Bitwise Operations], page 125).

```
(integer->list 6)   ⇒ (#t #t #f)
(integer->list 1 4) ⇒ (#f #f #f #t)
```

`list->integer` *lst* [Function]
`booleans->integer` *bool...* [Function]

Return an integer formed bitwise from the given *lst* list of booleans, or for `booleans->integer` from the *bool* arguments.

Each boolean is `#t` for a 1 and `#f` for a 0. The first element becomes the most significant bit in the return.

```
(list->integer '(#t #f #t #f)) ⇒ 10
```

7.5.35 SRFI-61 - A more general `cond` clause

This SRFI extends RnRS `cond` to support test expressions that return multiple values, as well as arbitrary definitions of test success. SRFI 61 is implemented in the Guile core; there's no module needed to get SRFI-61 itself. Extended `cond` is documented in Section 6.13.2 [Simple Conditional Evaluation], page 299.

7.5.36 SRFI-62 - S-expression comments.

Starting from version 2.0, Guile's `read` supports SRFI-62/R7RS S-expression comments by default.

7.5.37 SRFI-64 - A Scheme API for test suites.

See the specification of SRFI-64 (`http://srfi.schemers.org/srfi-64/srfi-64.html`).

7.5.38 SRFI-67 - Compare procedures

See the specification of SRFI-67 (`http://srfi.schemers.org/srfi-67/srfi-67.html`).

7.5.39 SRFI-69 - Basic hash tables

This is a portable wrapper around Guile's built-in hash table and weak table support. See Section 6.6.22 [Hash Tables], page 238, for information on that built-in support. Above that, this hash-table interface provides association of equality and hash functions with tables at creation time, so variants of each function are not required, as well as a procedure that takes care of most uses for Guile hash table handles, which this SRFI does not provide as such.

Access it with:

```
(use-modules (srfi srfi-69))
```

7.5.39.1 Creating hash tables

`make-hash-table` [*equal-proc hash-proc #:weak weakness start-size*] [Scheme Procedure]

Create and answer a new hash table with *equal-proc* as the equality function and *hash-proc* as the hashing function.

By default, *equal-proc* is `equal?`. It can be any two-argument procedure, and should answer whether two keys are the same for this table's purposes.

My default *hash-proc* assumes that `equal-proc` is no coarser than `equal?` unless it is literally `string-ci=?`. If provided, *hash-proc* should be a two-argument procedure that takes a key and the current table size, and answers a reasonably good hash integer between 0 (inclusive) and the size (exclusive).

weakness should be `#f` or a symbol indicating how "weak" the hash table is:

`#f` An ordinary non-weak hash table. This is the default.

`key` When the key has no more non-weak references at GC, remove that entry.

`value` When the value has no more non-weak references at GC, remove that entry.

`key-or-value`
When either has no more non-weak references at GC, remove the association.

As a legacy of the time when Guile couldn't grow hash tables, *start-size* is an optional integer argument that specifies the approximate starting size for the hash table, which will be rounded to an algorithmically-sounder number.

By *coarser* than `equal?`, we mean that for all *x* and *y* values where `(`*equal-proc x y*`)`, `(equal?` *x y*`)` as well. If that does not hold for your *equal-proc*, you must provide a *hash-proc*.

In the case of weak tables, remember that *references* above always refers to `eq?`-wise references. Just because you have a reference to some string `"foo"` doesn't mean that an association with key `"foo"` in a weak-key table *won't* be collected; it only counts as a reference if the two `"foo"`s are `eq?`, regardless of *equal-proc*. As such, it is usually only sensible to use `eq?` and `hashq` as the equivalence and hash functions for a weak table. See Section 6.19.3 [Weak References], page 407, for more information on Guile's built-in weak table support.

`alist->hash-table` *alist* [*equal-proc hash-proc #:weak weakness start-size*] [Scheme Procedure]
As with `make-hash-table`, but initialize it with the associations in *alist*. Where keys are repeated in *alist*, the leftmost association takes precedence.

7.5.39.2 Accessing table items

`hash-table-ref` *table key* [*default-thunk*] [Scheme Procedure]
`hash-table-ref/default` *table key default* [Scheme Procedure]
Answer the value associated with *key* in *table*. If *key* is not present, answer the result of invoking the thunk *default-thunk*, which signals an error instead by default.

`hash-table-ref/default` is a variant that requires a third argument, *default*, and answers *default* itself instead of invoking it.

`hash-table-set!` *table key new-value* [Scheme Procedure]
Set *key* to *new-value* in *table*.

`hash-table-delete!` *table key* [Scheme Procedure]
Remove the association of *key* in *table*, if present. If absent, do nothing.

`hash-table-exists?` *table key* [Scheme Procedure]
Answer whether *key* has an association in *table*.

`hash-table-update!` *table key modifier* [*default-thunk*] [Scheme Procedure]
`hash-table-update!/default` *table key modifier default* [Scheme Procedure]
Replace *key*'s associated value in *table* by invoking *modifier* with one argument, the old value.

If *key* is not present, and *default-thunk* is provided, invoke it with no arguments to get the "old value" to be passed to *modifier* as above. If *default-thunk* is not provided in such a case, signal an error.

`hash-table-update!/default` is a variant that requires the fourth argument, which is used directly as the "old value" rather than as a thunk to be invoked to retrieve the "old value".

7.5.39.3 Table properties

`hash-table-size` *table* [Scheme Procedure]
Answer the number of associations in *table*. This is guaranteed to run in constant time for non-weak tables.

`hash-table-keys` *table* [Scheme Procedure]
Answer an unordered list of the keys in *table*.

`hash-table-values` *table* [Scheme Procedure]
Answer an unordered list of the values in *table*.

`hash-table-walk` *table proc* [Scheme Procedure]
Invoke *proc* once for each association in *table*, passing the key and value as arguments.

`hash-table-fold` *table proc init* [Scheme Procedure]
Invoke (*proc key value previous*) for each *key* and *value* in *table*, where *previous* is the result of the previous invocation, using *init* as the first *previous* value. Answer the final *proc* result.

`hash-table->alist` *table* [Scheme Procedure]
Answer an alist where each association in *table* is an association in the result.

7.5.39.4 Hash table algorithms

Each hash table carries an *equivalence function* and a *hash function*, used to implement key lookups. Beginning users should follow the rules for consistency of the default *hash-proc* specified above. Advanced users can use these to implement their own equivalence and hash functions for specialized lookup semantics.

`hash-table-equivalence-function` *hash-table* [Scheme Procedure]
`hash-table-hash-function` *hash-table* [Scheme Procedure]
Answer the equivalence and hash function of *hash-table*, respectively.

`hash` *obj* [*size*] [Scheme Procedure]
`string-hash` *obj* [*size*] [Scheme Procedure]
`string-ci-hash` *obj* [*size*] [Scheme Procedure]
`hash-by-identity` *obj* [*size*] [Scheme Procedure]
Answer a hash value appropriate for equality predicate `equal?`, `string=?`, `string-ci=?`, and `eq?`, respectively.

`hash` is a backwards-compatible replacement for Guile's built-in `hash`.

7.5.40 SRFI-71 - Extended let-syntax for multiple values

This SRFI shadows the forms for `let`, `let*`, and `letrec` so that they may accept multiple values. For example:

```
(use-modules (srfi srfi-71))

(let* ((x y (values 1 2))
       (z (+ x y)))
  (* z 2))
⇒ 6
```

See the specification of SRFI-71 (`http://srfi.schemers.org/srfi-71/srfi-71.html`).

7.5.41 SRFI-87 => in case clauses

Starting from version 2.0.6, Guile's core `case` syntax supports `=>` in clauses, as specified by SRFI-87/R7RS. See Section 6.13.2 [Conditionals], page 299.

7.5.42 SRFI-88 Keyword Objects

SRFI-88 (`http://srfi.schemers.org/srfi-88/srfi-88.html`) provides *keyword objects*, which are equivalent to Guile's keywords (see Section 6.6.7 [Keywords], page 174). SRFI-88 keywords can be entered using the *postfix keyword syntax*, which consists of an identifier followed by `:` (see Section 6.18.2 [Scheme Read], page 385). SRFI-88 can be made available with:

```
(use-modules (srfi srfi-88))
```

Doing so installs the right reader option for keyword syntax, using `(read-set! keywords 'postfix)`. It also provides the procedures described below.

`keyword?` *obj* [Scheme Procedure]

Return `#t` if *obj* is a keyword. This is the same procedure as the same-named built-in procedure (see Section 6.6.7.4 [Keyword Procedures], page 176).

```
(keyword? foo:)         ⇒ #t
(keyword? 'foo:)        ⇒ #t
(keyword? "foo")        ⇒ #f
```

`keyword->string` *kw* [Scheme Procedure]

Return the name of *kw* as a string, i.e., without the trailing colon. The returned string may not be modified, e.g., with `string-set!`.

```
(keyword->string foo:)  ⇒ "foo"
```

`string->keyword` *str* [Scheme Procedure]

Return the keyword object whose name is *str*.

```
(keyword->string (string->keyword "a b c"))     ⇒ "a b c"
```

7.5.43 SRFI-98 Accessing environment variables.

This is a portable wrapper around Guile's built-in support for interacting with the current environment, See Section 7.2.6 [Runtime Environment], page 517.

`get-environment-variable` *name* [Scheme Procedure]
: Returns a string containing the value of the environment variable given by the string `name`, or `#f` if the named environment variable is not found. This is equivalent to `(getenv name)`.

`get-environment-variables` [Scheme Procedure]
: Returns the names and values of all the environment variables as an association list in which both the keys and the values are strings.

7.5.44 SRFI-105 Curly-infix expressions.

Guile's built-in reader includes support for SRFI-105 curly-infix expressions. See the specification of SRFI-105 (`http://srfi.schemers.org/srfi-105/srfi-105.html`). Some examples:

```
{n <= 5}                ⇒  (<= n 5)
{a + b + c}             ⇒  (+ a b c)
{a * {b + c}}           ⇒  (* a (+ b c))
{(- a) / b}             ⇒  (/ (- a) b)
{-(a) / b}              ⇒  (/ (- a) b) as well
{(f a b) + (g h)}       ⇒  (+ (f a b) (g h))
{f(a b) + g(h)}         ⇒  (+ (f a b) (g h)) as well
{f[a b] + g(h)}         ⇒  (+ ($bracket-apply$ f a b) (g h))
'{a + f(b) + x}         ⇒  '(+ a (f b) x)
{length(x) >= 6}        ⇒  (>= (length x) 6)
{n-1 + n-2}             ⇒  (+ n-1 n-2)
{n * factorial{n - 1}}  ⇒  (* n (factorial (- n 1)))
{{a > 0} and {b >= 1}}  ⇒  (and (> a 0) (>= b 1))
{f{n - 1}(x)}           ⇒  ((f (- n 1)) x)
{a . z}                 ⇒  ($nfx$ a . z)
{a + b - c}             ⇒  ($nfx$ a + b - c)
```

To enable curly-infix expressions within a file, place the reader directive `#!curly-infix` before the first use of curly-infix notation. To globally enable curly-infix expressions in Guile's reader, set the `curly-infix` read option.

Guile also implements the following non-standard extension to SRFI-105: if `curly-infix` is enabled and there is no other meaning assigned to square brackets (i.e. the `square-brackets` read option is turned off), then lists within square brackets are read as normal lists but with the special symbol `$bracket-list$` added to the front. To enable this combination of read options within a file, use the reader directive `#!curly-infix-and-bracket-lists`. For example:

```
[a b]    ⇒  ($bracket-list$ a b)
[a . b]  ⇒  ($bracket-list$ a . b)
```

For more information on reader options, See Section 6.18.2 [Scheme Read], page 385.

7.5.45 SRFI-111 Boxes.

SRFI-111 (`http://srfi.schemers.org/srfi-111/srfi-111.html`) provides boxes: objects with a single mutable cell.

`box` *value* [Scheme Procedure]
Return a newly allocated box whose contents is initialized to *value*.

`box?` *obj* [Scheme Procedure]
Return true if *obj* is a box, otherwise return false.

`unbox` *box* [Scheme Procedure]
Return the current contents of *box*.

`set-box!` *box value* [Scheme Procedure]
Set the contents of *box* to *value*.

7.5.46 Transducers

Some of the most common operations used in the Scheme language are those transforming lists: map, filter, take and so on. They work well, are well understood, and are used daily by most Scheme programmers. They are however not general because they only work on lists, and they do not compose very well since combining N of them builds `(- N 1)` intermediate lists.

Transducers are oblivious to what kind of process they are used in, and are composable without building intermediate collections. This means we can create a transducer that squares all even numbers:

```
(compose (tfilter odd?) (tmap (lambda (x) (* x x))))
```

and reuse it with lists, vectors, or in just about any context where data flows in one direction. We could use it as a processing step for asynchronous channels, with an event framework as a pre-processing step, or even in lazy contexts where you pass a lazy collection and a transducer to a function and get a new lazy collection back.

The traditional Scheme approach of having collection-specific procedures is not changed. We instead specify a general form of transformations that complement these procedures. The benefits are obvious: a clear, well-understood way of describing common transformations in a way that is faster than just chaining the collection-specific counterparts. For guile in particular this means a lot better GC performance.

Notice however that `(compose ...)` composes transducers left-to-right, due to how transducers are initiated.

7.5.46.1 SRFI-171 General Discussion

The concept of reducers

The central part of transducers are 3-arity reducing procedures.

- no arguments: Produces the identity of the reducer.
- (result-so-far): completion. Returns `result-so-far` either with or without transforming it first.

- (result-so-far input) combines `result-so-far` and `input` to produce a new `result-so-far`.

In the case of a summing `+` reducer, the reducer would produce, in arity order: `0`, `result-so-far`, `(+ result-so-far input)`. This happens to be exactly what the regular `+` does.

The concept of transducers

A transducer is a one-arity procedure that takes a reducer and produces a reducing function that behaves as follows:

- no arguments: calls reducer with no arguments (producing its identity)
- (result-so-far): Maybe transform the result-so-far and call reducer with it.
- (result-so-far input) Maybe do something to input and maybe call the reducer with result-so-far and the maybe-transformed input.

A simple example is as following:

```
(list-transduce (tfilter odd?)+ '(1 2 3 4 5)).
```

This first returns a transducer filtering all odd elements, then it runs `+` without arguments to retrieve its identity. It then starts the transduction by passing `+` to the transducer returned by `(tfilter odd?)` which returns a reducing function. It works not unlike reduce from SRFI 1, but also checks whether one of the intermediate transducers returns a "reduced" value (implemented as a SRFI 9 record), which means the reduction finished early.

Because transducers compose and the final reduction is only executed in the last step, composed transducers will not build any intermediate result or collections. Although the normal way of thinking about application of composed functions is right to left, due to how the transduction is built it is applied left to right. `(compose (tfilter odd?) (tmap sqrt))` will create a transducer that first filters out any odd values and then computes the square root of the rest.

State

Even though transducers appear to be somewhat of a generalisation of `map` and friends, this is not really true. Since transducers don't know in which context they are being used, some transducers must keep state where their collection-specific counterparts do not. The transducers that keep state do so using hidden mutable state, and as such all the caveats of mutation, parallelism, and multi-shot continuations apply. Each transducer keeping state is clearly described as doing so in the documentation.

Naming

Reducers exported from the transducers module are named as in their SRFI-1 counterpart, but prepended with an r. Transducers also follow that naming, but are prepended with a t.

7.5.46.2 Applying Transducers

`list-transduce` *xform f lst* [Scheme Procedure]

`list-transduce` *xform f identity lst* [Scheme Procedure]
Initialize the transducer *xform* by passing the reducer *f* to it. If no identity is provided, *f* runs without arguments to return the reducer identity. It then reduces over *lst* using the identity as the seed.

If one of the transducers finishes early (such as `ttake` or `tdrop`), it communicates this by returning a reduced value, which in the guile implementation is just a value wrapped in a SRFI 9 record type named "reduced". If such a value is returned by the transducer, `list-transduce` must stop execution and return an unreduced value immediately.

`vector-transduce` *xform f vec* [Scheme Procedure]
`vector-transduce` *xform f identity vec* [Scheme Procedure]
`string-transduce` *xform f str* [Scheme Procedure]
`string-transduce` *xform f identity str* [Scheme Procedure]
`bytevector-u8-transduce` *xform f bv* [Scheme Procedure]
`bytevector-u8-transduce` *xform f identity bv* [Scheme Procedure]
`generator-transduce` *xform f gen* [Scheme Procedure]
`generator-transduce` *xform f identity gen* [Scheme Procedure]
Same as `list-transduce`, but for vectors, strings, u8-bytevectors and SRFI-158-styled generators respectively.

`port-transduce` *xform f reader* [Scheme Procedure]
`port-transduce` *xform f reader port* [Scheme Procedure]
`port-transduce` *xform f identity reader port* [Scheme Procedure]
Same as `list-reduce` but for ports. Called without a port, it reduces over the results of applying *reader* until the EOF-object is returned, presumably to read from `current-input-port`. With a port *reader* is applied to *port* instead of without any arguments. If *identity* is provided, that is used as the initial identity in the reduction.

7.5.46.3 Reducers

`rcons` [Scheme Procedure]
a simple consing reducer. When called without values, it returns its identity, `'()`. With one value, which will be a list, it reverses the list (using `reverse!`). When called with two values, it conses the second value to the first.

```
(list-transduce (tmap (lambda (x) (+ x 1)) rcons (list 0 1 2 3))
⇒ (1 2 3 4)
```

`reverse-rcons` [Scheme Procedure]
same as rcons, but leaves the values in their reversed order.

```
(list-transduce (tmap (lambda (x) (+ x 1))) reverse-rcons (list 0 1 2 3))
⇒ (4 3 2 1)
```

`rany` *pred?* [Scheme Procedure]
The reducer version of any. Returns `(reduced (pred? value))` if any `(pred? value)` returns non-#f. The identity is #f.

```
(list-transduce (tmap (lambda (x) (+ x 1))) (rany odd?) (list 1 3 5))
⇒ #f
```

```
(list-transduce (tmap (lambda (x) (+ x 1))) (rany odd?) (list 1 3 4 5))
⇒ #t
```

`revery` *pred?* [Scheme Procedure]
The reducer version of every. Stops the transduction and returns `(reduced #f)` if any `(pred? value)` returns #f. If every `(pred? value)` returns true, it returns the result of the last invocation of `(pred? value)`. The identity is #t.

```
(list-transduce
  (tmap (lambda (x) (+ x 1)))
  (revery (lambda (v) (if (odd? v) v #f)))
  (list 2 4 6))
  ⇒ 7

(list-transduce (tmap (lambda (x) (+ x 1)) (revery odd?) (list 2 4 5 6))
⇒ #f
```

`rcount` [Scheme Procedure]
A simple counting reducer. Counts the values that pass through the transduction.

```
(list-transduce (tfilter odd?) rcount (list 1 2 3 4)) ⇒ 2.
```

7.5.46.4 Transducers

`tmap` *proc* [Scheme Procedure]
Returns a transducer that applies *proc* to all values. Stateless.

`pred?` [tfilter]
Returns a transducer that removes values for which *pred?* returns #f.

Stateless.

`tremove` *pred?* [Scheme Procedure]
Returns a transducer that removes values for which *pred?* returns non-#f.

Stateless

`tfilter-map` *proc* [Scheme Procedure]
The same as `(compose (tmap proc) (tfilter values))`. Stateless.

`treplace` *mapping* [Scheme Procedure]
The argument *mapping* is an association list (using `equal?` to compare keys), a hash-table, a one-argument procedure taking one argument and either producing that same argument or a replacement value.

Returns a transducer which checks for the presence of any value passed through it in mapping. If a mapping is found, the value of that mapping is returned, otherwise it just returns the original value.

Does not keep internal state, but modifying the mapping while it's in use by treplace is an error.

`tdrop` *n* [Scheme Procedure]

Returns a transducer that discards the first *n* values.

Stateful.

`ttake` *n* [Scheme Procedure]

Returns a transducer that discards all values and stops the transduction after the first *n* values have been let through. Any subsequent values are ignored.

Stateful.

`tdrop-while` *pred?* [Scheme Procedure]

Returns a transducer that discards the the first values for which *pred?* returns true.

Stateful.

`ttake-while` *pred?* [Scheme Procedure]
`ttake-while` *pred? retf* [Scheme Procedure]

Returns a transducer that stops the transduction after *pred?* has returned #f. Any subsequent values are ignored and the last successful value is returned. *retf* is a function that gets called whenever *pred?* returns false. The arguments passed are the result so far and the input for which pred? returns `#f`. The default function is `(lambda (result input) result)`.

Stateful.

`tconcatenate` [Scheme Procedure]

tconcatenate *is* a transducer that concatenates the content of each value (that must be a list) into the reduction.

```
(list-transduce tconcatenate rcons '((1 2) (3 4 5) (6 (7 8) 9)))
⇒ (1 2 3 4 5 6 (7 8) 9)
```

`tappend-map` *proc* [Scheme Procedure]

The same as `(compose (tmap proc) tconcatenate)`.

`tflatten` [Scheme Procedure]

tflatten *is* a transducer that flattens an input consisting of lists.

```
(list-transduce tflatten rcons '((1 2) 3 (4 (5 6) 7 8) 9)
⇒ (1 2 3 4 5 6 7 8 9)
```

`tdelete-neighbor-duplicates` [Scheme Procedure]
`tdelete-neighbor-duplicates` *equality-predicate* [Scheme Procedure]

Returns a transducer that removes any directly following duplicate elements. The default *equality-predicate* is `equal?`.

Stateful.

`tdelete-duplicates` [Scheme Procedure]
`tdelete-duplicates` *equality-predicate* [Scheme Procedure]

Returns a transducer that removes any subsequent duplicate elements compared using *equality-predicate*. The default *equality-predicate* is `equal?`.

Stateful.

`tsegment` *n* [Scheme Procedure]

Returns a transducer that groups *n* inputs in lists of *n* elements. When the transduction stops, it flushes any remaining collection, even if it contains fewer than *n* elements.

Stateful.

`tpartition` *pred?* [Scheme Procedure]

Returns a transducer that groups inputs in lists by whenever `(pred? input)` changes value.

Stateful.

`tadd-between` *value* [Scheme Procedure]

Returns a transducer which interposes *value* between each value and the next. This does not compose gracefully with transducers like `ttake`, as you might end up ending the transduction on `value`.

Stateful.

`tenumerate` [Scheme Procedure]
`tenumerate` *start* [Scheme Procedure]

Returns a transducer that indexes values passed through it, starting at *start*, which defaults to 0. The indexing is done through cons pairs like `(index . input)`.

```
(list-transduce (tenumerate 1) rcons (list 'first 'second 'third))
⇒ ((1 . first) (2 . second) (3 . third))
```

Stateful.

`tlog` [Scheme Procedure]
`tlog` *logger* [Scheme Procedure]

Returns a transducer that can be used to log or print values and results. The result of the *logger* procedure is discarded. The default *logger* is `(lambda (result input) (write input) (newline))`.

Stateless.

Guile-specific transducers

These transducers are available in the `(srfi srfi-171 gnu)` library, and are provided outside the standard described by the SRFI-171 document.

`tbatch` *reducer* [Scheme Procedure]
`tbatch` *transducer reducer* [Scheme Procedure]

A batching transducer that accumulates results using *reducer* or `((transducer) reducer)` until it returns a reduced value. This can be used to generalize something like `tsegment`:

```
;; This behaves exactly like (tsegment 4).
(list-transduce (tbatch (ttake 4) rcons) rcons (iota 10))
⇒ ((0 1 2 3) (4 5 6 7) (8 9))
```

`tfold` *reducer* [Scheme Procedure]
`tfold` *reducer seed* [Scheme Procedure]
A folding transducer that yields the result of `(reducer seed value)`, saving it's result between iterations.

```
(list-transduce (tfold +) rcons (iota 10))
⇒ (0 1 3 6 10 15 21 28 36 45)
```

7.5.46.5 Helper functions for writing transducers

These functions are in the `(srfi srfi-171 meta)` module and are only usable when you want to write your own transducers.

`reduced` *value* [Scheme Procedure]
Wraps a value in a `<reduced>` container, signalling that the reduction should stop.

`reduced?` *value* [Scheme Procedure]
Returns #t if value is a `<reduced>` record.

`unreduce` *reduced-container* [Scheme Procedure]
Returns the value in reduced-container.

`ensure-reduced` *value* [Scheme Procedure]
Wraps value in a `<reduced>` container if it is not already reduced.

`preserving-reduced` *reducer* [Scheme Procedure]
Wraps `reducer` in another reducer that encapsulates any returned reduced value in another reduced container. This is useful in places where you re-use a reducer with [collection]-reduce. If the reducer returns a reduced value, [collection]-reduce unwraps it. Unless handled, this leads to the reduction continuing.

`list-reduce` *f identity lst* [Scheme Procedure]
The reducing function used internally by `list-transduce`. *f* is a reducer as returned by a transducer. *identity* is the identity (sometimes called "seed") of the reduction. *lst* is a list. If *f* returns a reduced value, the reduction stops immediately and the unreduced value is returned.

`vector-reduce` *f identity vec* [Scheme Procedure]
The vector version of list-reduce.

`string-reduce` *f identity str* [Scheme Procedure]
The string version of list-reduce.

`bytevector-u8-reduce` *f identity bv* [Scheme Procedure]
The bytevector-u8 version of list-reduce.

`port-reduce` *f identity reader port* [Scheme Procedure]
The port version of list-reducer. It reduces over port using reader until reader returns the EOF object.

`generator-reduce` *f identity gen* [Scheme Procedure]
The port version of list-reduce. It reduces over `gen` until it returns the EOF object

7.6 R6RS Support

See Section 6.20.6 [R6RS Libraries], page 417, for more information on how to define R6RS libraries, and their integration with Guile modules.

7.6.1 Incompatibilities with the R6RS

There are some incompatibilities between Guile and the R6RS. Some of them are intentional, some of them are bugs, and some are simply unimplemented features. Please let the Guile developers know if you find one that is not on this list.

- The R6RS specifies many situations in which a conforming implementation must signal a specific error. Guile doesn't really care about that too much—if a correct R6RS program would not hit that error, we don't bother checking for it.
- Multiple `library` forms in one file are not yet supported. This is because the expansion of `library` sets the current module, but does not restore it. This is a bug.
- R6RS unicode escapes within strings are disabled by default, because they conflict with Guile's already-existing escapes. The same is the case for R6RS treatment of escaped newlines in strings.

  R6RS behavior can be turned on via a reader option. See Section 6.6.5.1 [String Syntax], page 141, for more information.
- Guile does not yet support Unicode escapes in symbols, such as `H\x65;llo` (the same as `Hello`), or `\x3BB;` (the same as λ).
- A `set!` to a variable transformer may only expand to an expression, not a definition—even if the original `set!` expression was in definition context.
- Instead of using the algorithm detailed in chapter 10 of the R6RS, expansion of toplevel forms happens sequentially.

  For example, while the expansion of the following set of toplevel definitions does the correct thing:

```
(begin
 (define even?
   (lambda (x)
     (or (= x 0) (odd? (- x 1)))))
 (define-syntax odd?
   (syntax-rules ()
     ((odd? x) (not (even? x)))))
 (even? 10))
⇒ #t
```

  The same definitions outside of the `begin` wrapper do not:

```
(define even?
  (lambda (x)
    (or (= x 0) (odd? (- x 1)))))
(define-syntax odd?
  (syntax-rules ()
    ((odd? x) (not (even? x)))))
(even? 10)
<unnamed port>:4:18: In procedure even?:
```

```
<unnamed port>:4:18: Wrong type to apply: #<syntax-transformer odd?>
```

This is because when expanding the right-hand-side of `even?`, the reference to `odd?` is not yet marked as a syntax transformer, so it is assumed to be a function.

This bug will only affect top-level programs, not code in `library` forms. Fixing it for toplevel forms seems doable, but tricky to implement in a backward-compatible way. Suggestions and/or patches would be appreciated.

- The `(rnrs io ports)` module is incomplete. Work is ongoing to fix this.
- Guile does not prevent use of textual I/O procedures on binary ports, or vice versa. All ports in Guile support both binary and textual I/O. See Section 6.14.3 [Encoding], page 334, for full details.
- Guile's implementation of `equal?` may fail to terminate when applied to arguments containing cycles.

Guile exposes a procedure in the root module to choose R6RS defaults over Guile's historical defaults.

`install-r6rs!` [Scheme Procedure]

Alter Guile's default settings to better conform to the R6RS.

While Guile's defaults may evolve over time, the current changes that this procedure imposes are to add `.sls` and `.guile.sls` to the set of supported `%load-extensions`, to better support R6RS conventions. See Section 6.18.7 [Load Paths], page 393. Also, enable R6RS unicode escapes in strings; see the discussion above.

Finally, note that the `--r6rs` command-line argument will call `install-r6rs!` before calling user code. R6RS users probably want to pass this argument to their Guile.

7.6.2 R6RS Standard Libraries

In contrast with earlier versions of the Revised Report, the R6RS organizes the procedures and syntactic forms required of conforming implementations into a set of "standard libraries" which can be imported as necessary by user programs and libraries. Here we briefly list the libraries that have been implemented for Guile.

We do not attempt to document these libraries fully here, as most of their functionality is already available in Guile itself. The expectation is that most Guile users will use the well-known and well-documented Guile modules. These R6RS libraries are mostly useful to users who want to port their code to other R6RS systems.

The documentation in the following sections reproduces some of the content of the library section of the Report, but is mostly intended to provide supplementary information about Guile's implementation of the R6RS standard libraries. For complete documentation, design rationales and further examples, we advise you to consult the "Standard Libraries" section of the Report (see Section "Standard Libraries" in *The Revised^6 Report on the Algorithmic Language Scheme*).

7.6.2.1 Library Usage

Guile implements the R6RS 'library' form as a transformation to a native Guile module definition. As a consequence of this, all of the libraries described in the following subsections, in addition to being available for use by R6RS libraries and top-level programs, can also be

imported as if they were normal Guile modules—via a `use-modules` form, say. For example, the R6RS "composite" library can be imported by:

```
(import (rnrs (6)))
(use-modules ((rnrs) :version (6)))
```

For more information on Guile's library implementation, see (see Section 6.20.6 [R6RS Libraries], page 417).

7.6.2.2 rnrs base

The `(rnrs base (6))` library exports the procedures and syntactic forms described in the main section of the Report (see Section "Base library" in *The Revised^6 Report on the Algorithmic Language Scheme*). They are grouped below by the existing manual sections to which they correspond.

`boolean?` *obj* [Scheme Procedure]
`not` *x* [Scheme Procedure]

See Section 6.6.1 [Booleans], page 104, for documentation.

`symbol?` *obj* [Scheme Procedure]
`symbol->string` *sym* [Scheme Procedure]
`string->symbol` *str* [Scheme Procedure]

See Section 6.6.6.4 [Symbol Primitives], page 167, for documentation.

`char?` *obj* [Scheme Procedure]
`char=?` [Scheme Procedure]
`char<?` [Scheme Procedure]
`char>?` [Scheme Procedure]
`char<=?` [Scheme Procedure]
`char>=?` [Scheme Procedure]
`integer->char` *n* [Scheme Procedure]
`char->integer` *chr* [Scheme Procedure]

See Section 6.6.3 [Characters], page 129, for documentation.

`list?` *x* [Scheme Procedure]
`null?` *x* [Scheme Procedure]

See Section 6.6.9.2 [List Predicates], page 181, for documentation.

`pair?` *x* [Scheme Procedure]
`cons` *x y* [Scheme Procedure]
`car` *pair* [Scheme Procedure]
`cdr` *pair* [Scheme Procedure]
`caar` *pair* [Scheme Procedure]
`cadr` *pair* [Scheme Procedure]
`cdar` *pair* [Scheme Procedure]
`cddr` *pair* [Scheme Procedure]
`caaar` *pair* [Scheme Procedure]
`caadr` *pair* [Scheme Procedure]
`cadar` *pair* [Scheme Procedure]
`cdaar` *pair* [Scheme Procedure]

`caddr` *pair* [Scheme Procedure]
`cdadr` *pair* [Scheme Procedure]
`cddar` *pair* [Scheme Procedure]
`cdddr` *pair* [Scheme Procedure]
`caaaar` *pair* [Scheme Procedure]
`caaadr` *pair* [Scheme Procedure]
`caadar` *pair* [Scheme Procedure]
`cadaar` *pair* [Scheme Procedure]
`cdaaar` *pair* [Scheme Procedure]
`cddaar` *pair* [Scheme Procedure]
`cdadar` *pair* [Scheme Procedure]
`cdaadr` *pair* [Scheme Procedure]
`cadadr` *pair* [Scheme Procedure]
`caaddr` *pair* [Scheme Procedure]
`caddar` *pair* [Scheme Procedure]
`cadddr` *pair* [Scheme Procedure]
`cdaddr` *pair* [Scheme Procedure]
`cddadr` *pair* [Scheme Procedure]
`cdddar` *pair* [Scheme Procedure]
`cddddr` *pair* [Scheme Procedure]

See Section 6.6.8 [Pairs], page 178, for documentation.

`number?` *obj* [Scheme Procedure]

See Section 6.6.2.1 [Numerical Tower], page 105, for documentation.

`string?` *obj* [Scheme Procedure]

See Section 6.6.5.2 [String Predicates], page 143, for documentation.

`procedure?` *obj* [Scheme Procedure]

See Section 6.9.7 [Procedure Properties], page 258, for documentation.

`define` *name value* [Scheme Syntax]
`set!` *variable-name value* [Scheme Syntax]

See Section 3.1.3 [Definition], page 16, for documentation.

`define-syntax` *keyword expression* [Scheme Syntax]
`let-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Scheme Syntax]
`letrec-syntax` ((*keyword transformer*) ...) *exp1 exp2 ...* [Scheme Syntax]

See Section 6.10.1 [Defining Macros], page 261, for documentation.

`identifier-syntax` *exp* [Scheme Syntax]

See Section 6.10.6 [Identifier Macros], page 276, for documentation.

`syntax-rules` *literals* (*pattern template*) ... [Scheme Syntax]

See Section 6.10.2 [Syntax Rules], page 263, for documentation.

`lambda` *formals body* [Scheme Syntax]

See Section 6.9.1 [Lambda], page 248, for documentation.

`let` *bindings body* [Scheme Syntax]
`let*` *bindings body* [Scheme Syntax]
`letrec` *bindings body* [Scheme Syntax]
`letrec*` *bindings body* [Scheme Syntax]
See Section 6.12.2 [Local Bindings], page 294, for documentation.

`let-values` *bindings body* [Scheme Syntax]
`let*-values` *bindings body* [Scheme Syntax]
See Section 7.5.10 [SRFI-11], page 609, for documentation.

`begin` *expr1 expr2 ...* [Scheme Syntax]
See Section 6.13.1 [begin], page 298, for documentation.

`quote` *expr* [Scheme Syntax]
`quasiquote` *expr* [Scheme Syntax]
`unquote` *expr* [Scheme Syntax]
`unquote-splicing` *expr* [Scheme Syntax]
See Section 6.18.1.1 [Expression Syntax], page 382, for documentation.

`if` *test consequence* [*alternate*] [Scheme Syntax]
`cond` *clause1 clause2 ...* [Scheme Syntax]
`case` *key clause1 clause2 ...* [Scheme Syntax]
See Section 6.13.2 [Conditionals], page 299, for documentation.

`and` *expr ...* [Scheme Syntax]
`or` *expr ...* [Scheme Syntax]
See Section 6.13.3 [and or], page 301, for documentation.

`eq?` *x y* [Scheme Procedure]
`eqv?` *x y* [Scheme Procedure]
`equal?` *x y* [Scheme Procedure]
`symbol=?` *symbol1 symbol2 ...* [Scheme Procedure]
See Section 6.11.1 [Equality], page 283, for documentation.

`symbol=?` is identical to `eq?`.

`complex?` *z* [Scheme Procedure]
See Section 6.6.2.4 [Complex Numbers], page 113, for documentation.

`real-part` *z* [Scheme Procedure]
`imag-part` *z* [Scheme Procedure]
`make-rectangular` *real_part imaginary_part* [Scheme Procedure]
`make-polar` *x y* [Scheme Procedure]
`magnitude` *z* [Scheme Procedure]
`angle` *z* [Scheme Procedure]
See Section 6.6.2.10 [Complex], page 118, for documentation.

`sqrt` *z* [Scheme Procedure]
`exp` *z* [Scheme Procedure]
`expt` *z1 z2* [Scheme Procedure]
`log` *z* [Scheme Procedure]

`sin` *z* [Scheme Procedure]
`cos` *z* [Scheme Procedure]
`tan` *z* [Scheme Procedure]
`asin` *z* [Scheme Procedure]
`acos` *z* [Scheme Procedure]
`atan` *z* [Scheme Procedure]
See Section 6.6.2.12 [Scientific], page 123, for documentation.

`real?` *x* [Scheme Procedure]
`rational?` *x* [Scheme Procedure]
`numerator` *x* [Scheme Procedure]
`denominator` *x* [Scheme Procedure]
`rationalize` *x eps* [Scheme Procedure]
See Section 6.6.2.3 [Reals and Rationals], page 110, for documentation.

`exact?` *x* [Scheme Procedure]
`inexact?` *x* [Scheme Procedure]
`exact` *z* [Scheme Procedure]
`inexact` *z* [Scheme Procedure]
See Section 6.6.2.5 [Exactness], page 113, for documentation. The `exact` and `inexact` procedures are identical to the `inexact->exact` and `exact->inexact` procedures provided by Guile's code library.

`integer?` *x* [Scheme Procedure]
See Section 6.6.2.2 [Integers], page 106, for documentation.

`odd?` *n* [Scheme Procedure]
`even?` *n* [Scheme Procedure]
`gcd` *x ...* [Scheme Procedure]
`lcm` *x ...* [Scheme Procedure]
`exact-integer-sqrt` *k* [Scheme Procedure]
See Section 6.6.2.7 [Integer Operations], page 116, for documentation.

`=` [Scheme Procedure]
`<` [Scheme Procedure]
`>` [Scheme Procedure]
`<=` [Scheme Procedure]
`>=` [Scheme Procedure]
`zero?` *x* [Scheme Procedure]
`positive?` *x* [Scheme Procedure]
`negative?` *x* [Scheme Procedure]
See Section 6.6.2.8 [Comparison], page 117, for documentation.

`for-each` *f lst1 lst2 ...* [Scheme Procedure]
See Section 7.5.3.5 [SRFI-1 Fold and Map], page 588, for documentation.

`list` *elem ...* [Scheme Procedure]
See Section 6.6.9.3 [List Constructors], page 182, for documentation.

`length` *lst* [Scheme Procedure]
`list-ref` *lst k* [Scheme Procedure]
`list-tail` *lst k* [Scheme Procedure]
See Section 6.6.9.4 [List Selection], page 183, for documentation.

`append` *lst ... obj* [Scheme Procedure]
`append` [Scheme Procedure]
`reverse` *lst* [Scheme Procedure]
See Section 6.6.9.5 [Append/Reverse], page 183, for documentation.

`number->string` *n* [*radix*] [Scheme Procedure]
`string->number` *str* [*radix*] [Scheme Procedure]
See Section 6.6.2.9 [Conversion], page 118, for documentation.

`string` *char ...* [Scheme Procedure]
`make-string` *k* [*chr*] [Scheme Procedure]
`list->string` *lst* [Scheme Procedure]
See Section 6.6.5.3 [String Constructors], page 144, for documentation.

`string->list` *str* [*start* [*end*]] [Scheme Procedure]
See Section 6.6.5.4 [List/String Conversion], page 145, for documentation.

`string-length` *str* [Scheme Procedure]
`string-ref` *str k* [Scheme Procedure]
`string-copy` *str* [*start* [*end*]] [Scheme Procedure]
`substring` *str start* [*end*] [Scheme Procedure]
See Section 6.6.5.5 [String Selection], page 145, for documentation.

`string=?` *s1 s2 s3 ...* [Scheme Procedure]
`string<?` *s1 s2 s3 ...* [Scheme Procedure]
`string>?` *s1 s2 s3 ...* [Scheme Procedure]
`string<=?` *s1 s2 s3 ...* [Scheme Procedure]
`string>=?` *s1 s2 s3 ...* [Scheme Procedure]
See Section 6.6.5.7 [String Comparison], page 148, for documentation.

`string-append` *arg ...* [Scheme Procedure]
See Section 6.6.5.10 [Reversing and Appending Strings], page 155, for documentation.

`string-for-each` *proc s* [*start* [*end*]] [Scheme Procedure]
See Section 6.6.5.11 [Mapping Folding and Unfolding], page 156, for documentation.

`+` *z1 ...* [Scheme Procedure]
`-` *z1 z2 ...* [Scheme Procedure]
`*` *z1 ...* [Scheme Procedure]
`/` *z1 z2 ...* [Scheme Procedure]
`max` *x1 x2 ...* [Scheme Procedure]
`min` *x1 x2 ...* [Scheme Procedure]
`abs` *x* [Scheme Procedure]
`truncate` *x* [Scheme Procedure]
`floor` *x* [Scheme Procedure]

`ceiling` *x* [Scheme Procedure]
`round` *x* [Scheme Procedure]

See Section 6.6.2.11 [Arithmetic], page 119, for documentation.

`div` *x y* [Scheme Procedure]
`mod` *x y* [Scheme Procedure]
`div-and-mod` *x y* [Scheme Procedure]

These procedures accept two real numbers *x* and *y*, where the divisor *y* must be non-zero. `div` returns the integer *q* and `mod` returns the real number *r* such that $x = q * y + r$ and $0 <= r < abs(y)$. `div-and-mod` returns both *q* and *r*, and is more efficient than computing each separately. Note that when $y > 0$, `div` returns $floor(x/y)$, otherwise it returns $ceiling(x/y)$.

```
(div 123 10) ⇒ 12
(mod 123 10) ⇒ 3
(div-and-mod 123 10) ⇒ 12 and 3
(div-and-mod 123 -10) ⇒ -12 and 3
(div-and-mod -123 10) ⇒ -13 and 7
(div-and-mod -123 -10) ⇒ 13 and 7
(div-and-mod -123.2 -63.5) ⇒ 2.0 and 3.8
(div-and-mod 16/3 -10/7) ⇒ -3 and 22/21
```

`div0` *x y* [Scheme Procedure]
`mod0` *x y* [Scheme Procedure]
`div0-and-mod0` *x y* [Scheme Procedure]

These procedures accept two real numbers *x* and *y*, where the divisor *y* must be non-zero. `div0` returns the integer *q* and `mod0` returns the real number *r* such that $x = q * y + r$ and $-abs(y/2) <= r < abs(y/2)$. `div0-and-mod0` returns both *q* and *r*, and is more efficient than computing each separately.

Note that `div0` returns x/y rounded to the nearest integer. When x/y lies exactly half-way between two integers, the tie is broken according to the sign of *y*. If $y > 0$, ties are rounded toward positive infinity, otherwise they are rounded toward negative infinity. This is a consequence of the requirement that $-abs(y/2) <= r < abs(y/2)$.

```
(div0 123 10) ⇒ 12
(mod0 123 10) ⇒ 3
(div0-and-mod0 123 10) ⇒ 12 and 3
(div0-and-mod0 123 -10) ⇒ -12 and 3
(div0-and-mod0 -123 10) ⇒ -12 and -3
(div0-and-mod0 -123 -10) ⇒ 12 and -3
(div0-and-mod0 -123.2 -63.5) ⇒ 2.0 and 3.8
(div0-and-mod0 16/3 -10/7) ⇒ -4 and -8/21
```

`real-valued?` *obj* [Scheme Procedure]
`rational-valued?` *obj* [Scheme Procedure]
`integer-valued?` *obj* [Scheme Procedure]

These procedures return `#t` if and only if their arguments can, respectively, be coerced to a real, rational, or integer value without a loss of numerical precision.

`real-valued?` will return `#t` for complex numbers whose imaginary parts are zero.

`nan?` *x* [Scheme Procedure]
`infinite?` *x* [Scheme Procedure]
`finite?` *x* [Scheme Procedure]
`nan?` returns `#t` if *x* is a NaN value, `#f` otherwise. `infinite?` returns `#t` if *x* is an infinite value, `#f` otherwise. `finite?` returns `#t` if *x* is neither infinite nor a NaN value, otherwise it returns `#f`. Every real number satisfies exactly one of these predicates. An exception is raised if *x* is not real.

`assert` *expr* [Scheme Syntax]
Raises an `&assertion` condition if *expr* evaluates to `#f`; otherwise evaluates to the value of *expr*.

`error` *who message irritant1 ...* [Scheme Procedure]
`assertion-violation` *who message irritant1 ...* [Scheme Procedure]
These procedures raise compound conditions based on their arguments: If *who* is not `#f`, the condition will include a `&who` condition whose `who` field is set to *who*; a `&message` condition will be included with a `message` field equal to *message*; an `&irritants` condition will be included with its `irritants` list given by `irritant1 ...`.

`error` produces a compound condition with the simple conditions described above, as well as an `&error` condition; `assertion-violation` produces one that includes an `&assertion` condition.

`vector-map` *proc v* [Scheme Procedure]
`vector-for-each` *proc v* [Scheme Procedure]
These procedures implement the `map` and `for-each` contracts over vectors.

`vector` *arg ...* [Scheme Procedure]
`vector?` *obj* [Scheme Procedure]
`make-vector` *len* [Scheme Procedure]
`make-vector` *len fill* [Scheme Procedure]
`list->vector` *l* [Scheme Procedure]
`vector->list` *v* [Scheme Procedure]
See Section 6.6.10.2 [Vector Creation], page 187, for documentation.

`vector-length` *vector* [Scheme Procedure]
`vector-ref` *vector k* [Scheme Procedure]
`vector-set!` *vector k obj* [Scheme Procedure]
`vector-fill!` *v fill* [Scheme Procedure]
See Section 6.6.10.3 [Vector Accessors], page 188, for documentation.

`call-with-current-continuation` *proc* [Scheme Procedure]
`call/cc` *proc* [Scheme Procedure]
See Section 6.13.6 [Continuations], page 307, for documentation.

`values` *arg ...* [Scheme Procedure]
`call-with-values` *producer consumer* [Scheme Procedure]
See Section 6.13.7 [Multiple Values], page 309, for documentation.

`dynamic-wind` *in_guard thunk out_guard* [Scheme Procedure]
See Section 6.13.10 [Dynamic Wind], page 320, for documentation.

`apply` *proc arg . . . arglst* [Scheme Procedure]
See Section 6.18.4 [Fly Evaluation], page 387, for documentation.

7.6.2.3 rnrs unicode

The `(rnrs unicode (6))` library provides procedures for manipulating Unicode characters and strings.

`char-upcase` *char* [Scheme Procedure]
`char-downcase` *char* [Scheme Procedure]
`char-titlecase` *char* [Scheme Procedure]
`char-foldcase` *char* [Scheme Procedure]
These procedures translate their arguments from one Unicode character set to another. `char-upcase`, `char-downcase`, and `char-titlecase` are identical to their counterparts in the Guile core library; See Section 6.6.3 [Characters], page 129, for documentation.

`char-foldcase` returns the result of applying `char-upcase` to its argument, followed by `char-downcase`—except in the case of the Turkic characters `U+0130` and `U+0131`, for which the procedure acts as the identity function.

`char-ci=?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci<?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci>?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci<=?` *char1 char2 char3 ...* [Scheme Procedure]
`char-ci>=?` *char1 char2 char3 ...* [Scheme Procedure]
These procedures facilitate case-insensitive comparison of Unicode characters. They are identical to the procedures provided by Guile's core library. See Section 6.6.3 [Characters], page 129, for documentation.

`char-alphabetic?` *char* [Scheme Procedure]
`char-numeric?` *char* [Scheme Procedure]
`char-whitespace?` *char* [Scheme Procedure]
`char-upper-case?` *char* [Scheme Procedure]
`char-lower-case?` *char* [Scheme Procedure]
`char-title-case?` *char* [Scheme Procedure]
These procedures implement various Unicode character set predicates. They are identical to the procedures provided by Guile's core library. See Section 6.6.3 [Characters], page 129, for documentation.

`char-general-category` *char* [Scheme Procedure]
See Section 6.6.3 [Characters], page 129, for documentation.

`string-upcase` *string* [Scheme Procedure]
`string-downcase` *string* [Scheme Procedure]
`string-titlecase` *string* [Scheme Procedure]
`string-foldcase` *string* [Scheme Procedure]
These procedures perform Unicode case folding operations on their input. See Section 6.6.5.9 [Alphabetic Case Mapping], page 154, for documentation.

`string-ci=?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci<?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci>?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci<=?` *string1 string2 string3 ...* [Scheme Procedure]
`string-ci>=?` *string1 string2 string3 ...* [Scheme Procedure]
These procedures perform case-insensitive comparison on their input. See Section 6.6.5.7 [String Comparison], page 148, for documentation.

`string-normalize-nfd` *string* [Scheme Procedure]
`string-normalize-nfkd` *string* [Scheme Procedure]
`string-normalize-nfc` *string* [Scheme Procedure]
`string-normalize-nfkc` *string* [Scheme Procedure]
These procedures perform Unicode string normalization operations on their input. See Section 6.6.5.7 [String Comparison], page 148, for documentation.

7.6.2.4 rnrs bytevectors

The `(rnrs bytevectors (6))` library provides procedures for working with blocks of binary data. This functionality is documented in its own section of the manual; See Section 6.6.12 [Bytevectors], page 193.

7.6.2.5 rnrs lists

The `(rnrs lists (6))` library provides procedures additional procedures for working with lists.

`find` *proc list* [Scheme Procedure]
This procedure is identical to the one defined in Guile's SRFI-1 implementation. See Section 7.5.3.7 [SRFI-1 Searching], page 592, for documentation.

`for-all` *proc list1 list2 ...* [Scheme Procedure]
`exists` *proc list1 list2 ...* [Scheme Procedure]
The `for-all` procedure is identical to the `every` procedure defined by SRFI-1; the `exists` procedure is identical to SRFI-1's `any`. See Section 7.5.3.7 [SRFI-1 Searching], page 592, for documentation.

`filter` *proc list* [Scheme Procedure]
`partition` *proc list* [Scheme Procedure]
These procedures are identical to the ones provided by SRFI-1. See Section 6.6.9.6 [List Modification], page 184, for a description of `filter`; See Section 7.5.3.6 [SRFI-1 Filtering and Partitioning], page 591, for `partition`.

`fold-right` *combine nil list1 list2 . . .* [Scheme Procedure]
This procedure is identical the `fold-right` procedure provided by SRFI-1. See Section 7.5.3.5 [SRFI-1 Fold and Map], page 588, for documentation.

`fold-left` *combine nil list1 list2 . . .* [Scheme Procedure]
This procedure is like `fold` from SRFI-1, but *combine* is called with the seed as the first argument. See Section 7.5.3.5 [SRFI-1 Fold and Map], page 588, for documentation.

`remp` *proc list* [Scheme Procedure]
`remove` *obj list* [Scheme Procedure]
`remv` *obj list* [Scheme Procedure]
`remq` *obj list* [Scheme Procedure]

`remove`, `remv`, and `remq` are identical to the `delete`, `delv`, and `delq` procedures provided by Guile's core library, (see Section 6.6.9.6 [List Modification], page 184). `remp` is identical to the alternate `remove` procedure provided by SRFI-1; See Section 7.5.3.8 [SRFI-1 Deleting], page 593.

`memp` *proc list* [Scheme Procedure]
`member` *obj list* [Scheme Procedure]
`memv` *obj list* [Scheme Procedure]
`memq` *obj list* [Scheme Procedure]

`member`, `memv`, and `memq` are identical to the procedures provided by Guile's core library; See Section 6.6.9.7 [List Searching], page 185, for their documentation. `memp` uses the specified predicate function `proc` to test elements of the list *list*—it behaves similarly to `find`, except that it returns the first sublist of *list* whose `car` satisfies *proc*.

`assp` *proc alist* [Scheme Procedure]
`assoc` *obj alist* [Scheme Procedure]
`assv` *obj alist* [Scheme Procedure]
`assq` *obj alist* [Scheme Procedure]

`assoc`, `assv`, and `assq` are identical to the procedures provided by Guile's core library; See Section 6.6.20.1 [Alist Key Equality], page 230, for their documentation. `assp` uses the specified predicate function `proc` to test keys in the association list *alist*.

`cons*` *obj1 ... obj* [Scheme Procedure]
`cons*` *obj* [Scheme Procedure]

This procedure is identical to the one exported by Guile's core library. See Section 6.6.9.3 [List Constructors], page 182, for documentation.

7.6.2.6 rnrs sorting

The `(rnrs sorting (6))` library provides procedures for sorting lists and vectors.

`list-sort` *proc list* [Scheme Procedure]
`vector-sort` *proc vector* [Scheme Procedure]

These procedures return their input sorted in ascending order, without modifying the original data. *proc* must be a procedure that takes two elements from the input list or vector as arguments, and returns a true value if the first is "less" than the second, `#f` otherwise. `list-sort` returns a list; `vector-sort` returns a vector.

Both `list-sort` and `vector-sort` are implemented in terms of the `stable-sort` procedure from Guile's core library. See Section 6.11.3 [Sorting], page 286, for a discussion of the behavior of that procedure.

`vector-sort!` *proc vector* [Scheme Procedure]

Performs a destructive, "in-place" sort of *vector*, using *proc* as described above to determine an ascending ordering of elements. `vector-sort!` returns an unspecified value.

This procedure is implemented in terms of the `sort!` procedure from Guile's core library. See Section 6.11.3 [Sorting], page 286, for more information.

7.6.2.7 rnrs control

The `(rnrs control (6))` library provides syntactic forms useful for constructing conditional expressions and controlling the flow of execution.

`when` *test expression1 expression2 ...* [Scheme Syntax]
`unless` *test expression1 expression2 ...* [Scheme Syntax]

The `when` form is evaluated by evaluating the specified *test* expression; if the result is a true value, the *expression*s that follow it are evaluated in order, and the value of the final *expression* becomes the value of the entire `when` expression.

The `unless` form behaves similarly, with the exception that the specified *expression*s are only evaluated if the value of *test* is false.

`do` ((*variable init step*) ...) (*test expression ...*) *command ...* [Scheme Syntax]

This form is identical to the one provided by Guile's core library. See Section 6.13.4 [while do], page 301, for documentation.

`case-lambda` *clause ...* [Scheme Syntax]

This form is identical to the one provided by Guile's core library. See Section 6.9.5 [Case-lambda], page 256, for documentation.

7.6.2.8 R6RS Records

The manual sections below describe Guile's implementation of R6RS records, which provide support for user-defined data types. The R6RS records API provides a superset of the features provided by Guile's "native" records, as well as those of the SRFI-9 records API; See Section 6.6.17 [Records], page 221, and Section 6.6.16 [SRFI-9 Records], page 218, for a description of those interfaces.

As with SRFI-9 and Guile's native records, R6RS records are constructed using a record-type descriptor that specifies attributes like the record's name, its fields, and the mutability of those fields.

R6RS records extend this framework to support single inheritance via the specification of a "parent" type for a record type at definition time. Accessors and mutator procedures for the fields of a parent type may be applied to records of a subtype of this parent. A record type may be *sealed*, in which case it cannot be used as the parent of another record type.

The inheritance mechanism for record types also informs the process of initializing the fields of a record and its parents. Constructor procedures that generate new instances of a record type are obtained from a record constructor descriptor, which encapsulates the record-type descriptor of the record to be constructed along with a *protocol* procedure that defines how constructors for record subtypes delegate to the constructors of their parent types.

A protocol is a procedure used by the record system at construction time to bind arguments to the fields of the record being constructed. The protocol procedure is passed a procedure *n* that accepts the arguments required to construct the record's parent type; this

procedure, when invoked, will return a procedure *p* that accepts the arguments required to construct a new instance of the record type itself and returns a new instance of the record type.

The protocol should in turn return a procedure that uses *n* and *p* to initialize the fields of the record type and its parent type(s). This procedure will be the constructor returned by

As a trivial example, consider the hypothetical record type `pixel`, which encapsulates an x-y location on a screen, and `voxel`, which has `pixel` as its parent type and stores an additional coordinate. The following protocol produces a constructor procedure that accepts all three coordinates, uses the first two to initialize the fields of `pixel`, and binds the third to the single field of `voxel`.

```
(lambda (n)
  (lambda (x y z)
    (let ((p (n x y)))
      (p z))))
```

It may be helpful to think of protocols as "constructor factories" that produce chains of delegating constructors glued together by the helper procedure *n*.

An R6RS record type may be declared to be *nongenerative* via the use of a unique generated or user-supplied symbol—or *uid*—such that subsequent record type declarations with the same uid and attributes will return the previously-declared record-type descriptor.

R6RS record types may also be declared to be *opaque*, in which case the various predicates and introspection procedures defined in `(rnrs records introspection)` will behave as if records of this type are not records at all.

Note that while the R6RS records API shares much of its namespace with both the SRFI-9 and native Guile records APIs, it is not currently compatible with either.

7.6.2.9 rnrs records syntactic

The `(rnrs records syntactic (6))` library exports the syntactic API for working with R6RS records.

`define-record-type` *name-spec record-clause . . .* [Scheme Syntax]

Defines a new record type, introducing bindings for a record-type descriptor, a record constructor descriptor, a constructor procedure, a record predicate, and accessor and mutator procedures for the new record type's fields.

name-spec must either be an identifier or must take the form `(record-name constructor-name predicate-name)`, where *record-name*, *constructor-name*, and *predicate-name* are all identifiers and specify the names to which, respectively, the record-type descriptor, constructor, and predicate procedures will be bound. If *name-spec* is only an identifier, it specifies the name to which the generated record-type descriptor will be bound.

Each *record-clause* must be one of the following:

- `(fields field-spec*)`, where each *field-spec* specifies a field of the new record type and takes one of the following forms:

- `(immutable field-name accessor-name)`, which specifies an immutable field with the name *field-name* and binds an accessor procedure for it to the name given by *accessor-name*
- `(mutable field-name accessor-name mutator-name)`, which specifies a mutable field with the name *field-name* and binds accessor and mutator procedures to *accessor-name* and *mutator-name*, respectively
- `(immutable field-name)`, which specifies an immutable field with the name *field-name*; an accessor procedure for it will be created and named by appending record name and *field-name* with a hyphen separator
- `(mutable field-name)`, which specifies a mutable field with the name *field-name*; an accessor procedure for it will be created and named as described above; a mutator procedure will also be created and named by appending `-set!` to the accessor name
- `field-name`, which specifies an immutable field with the name *field-name*; an access procedure for it will be created and named as described above

- `(parent parent-name)`, where *parent-name* is a symbol giving the name of the record type to be used as the parent of the new record type
- `(protocol expression)`, where *expression* evaluates to a protocol procedure which behaves as described above, and is used to create a record constructor descriptor for the new record type
- `(sealed sealed?)`, where *sealed?* is a boolean value that specifies whether or not the new record type is sealed
- `(opaque opaque?)`, where *opaque?* is a boolean value that specifies whether or not the new record type is opaque
- `(nongenerative [uid])`, which specifies that the record type is nongenerative via the optional uid *uid*. If *uid* is not specified, a unique uid will be generated at expansion time
- `(parent-rtd parent-rtd parent-cd)`, a more explicit form of the `parent` form above; *parent-rtd* and *parent-cd* should evaluate to a record-type descriptor and a record constructor descriptor, respectively

`record-type-descriptor` *record-name* [Scheme Syntax]

Evaluates to the record-type descriptor associated with the type specified by *record-name*.

`record-constructor-descriptor` *record-name* [Scheme Syntax]

Evaluates to the record-constructor descriptor associated with the type specified by *record-name*.

7.6.2.10 rnrs records procedural

The `(rnrs records procedural (6))` library exports the procedural API for working with R6RS records.

`make-record-type-descriptor` *name parent uid sealed? opaque? fields* [Scheme Procedure]

Returns a new record-type descriptor with the specified characteristics: *name* must be a symbol giving the name of the new record type; *parent* must be either `#f` or a

non-sealed record-type descriptor for the returned record type to extend; *uid* must be either `#f`, indicating that the record type is generative, or a symbol giving the type's nongenerative uid; *sealed?* and *opaque?* must be boolean values that specify the sealedness and opaqueness of the record type; *fields* must be a vector of zero or more field specifiers of the form `(mutable name)` or `(immutable name)`, where name is a symbol giving a name for the field.

If *uid* is not `#f`, it must be a symbol

`record-type-descriptor?` *obj* [Scheme Procedure]
Returns `#t` if *obj* is a record-type descriptor, `#f` otherwise.

`make-record-constructor-descriptor` *rtd* [Scheme Procedure]
parent-constructor-descriptor protocol
Returns a new record constructor descriptor that can be used to produce constructors for the record type specified by the record-type descriptor *rtd* and whose delegation and binding behavior are specified by the protocol procedure *protocol*.

parent-constructor-descriptor specifies a record constructor descriptor for the parent type of *rtd*, if one exists. If *rtd* represents a base type, then *parent-constructor-descriptor* must be `#f`. If *rtd* is an extension of another type, *parent-constructor-descriptor* may still be `#f`, but protocol must also be `#f` in this case.

`record-constructor` *rcd* [Scheme Procedure]
Returns a record constructor procedure by invoking the protocol defined by the record-constructor descriptor *rcd*.

`record-predicate` *rtd* [Scheme Procedure]
Returns the record predicate procedure for the record-type descriptor *rtd*.

`record-accessor` *rtd k* [Scheme Procedure]
Returns the record field accessor procedure for the *k*th field of the record-type descriptor *rtd*.

`record-mutator` *rtd k* [Scheme Procedure]
Returns the record field mutator procedure for the *k*th field of the record-type descriptor *rtd*. An `&assertion` condition will be raised if this field is not mutable.

7.6.2.11 rnrs records inspection

The `(rnrs records inspection (6))` library provides procedures useful for accessing metadata about R6RS records.

`record?` *obj* [Scheme Procedure]
Return `#t` if the specified object is a non-opaque R6RS record, `#f` otherwise.

`record-rtd` *record* [Scheme Procedure]
Returns the record-type descriptor for *record*. An `&assertion` is raised if *record* is opaque.

`record-type-name` *rtd* [Scheme Procedure]
Returns the name of the record-type descriptor *rtd*.

`record-type-parent` *rtd* [Scheme Procedure]
Returns the parent of the record-type descriptor *rtd*, or `#f` if it has none.

`record-type-uid` *rtd* [Scheme Procedure]
Returns the uid of the record-type descriptor *rtd*, or `#f` if it has none.

`record-type-generative?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is generative, `#f` otherwise.

`record-type-sealed?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is sealed, `#f` otherwise.

`record-type-opaque?` *rtd* [Scheme Procedure]
Returns `#t` if the record-type descriptor *rtd* is opaque, `#f` otherwise.

`record-type-field-names` *rtd* [Scheme Procedure]
Returns a vector of symbols giving the names of the fields defined by the record-type descriptor *rtd* (and not any of its sub- or supertypes).

`record-field-mutable?` *rtd k* [Scheme Procedure]
Returns `#t` if the field at index *k* of the record-type descriptor *rtd* (and not any of its sub- or supertypes) is mutable.

7.6.2.12 rnrs exceptions

The `(rnrs exceptions (6))` library provides functionality related to signaling and handling exceptional situations. This functionality re-exports Guile's core exception-handling primitives. See Section 6.13.8 [Exceptions], page 311, for a full discussion. See Section 7.5.23 [SRFI-34], page 625, for a similar pre-R6RS facility. In Guile, SRFI-34, SRFI-35, and R6RS exception handling are all built on the same core facilities, and so are interoperable.

`with-exception-handler` *handler thunk* [Scheme Procedure]
See Section 6.13.8.2 [Raising and Handling Exceptions], page 315, for more information on `with-exception-handler`.

`guard` (*variable clause1 clause2 ...*) *body* [Scheme Syntax]
Evaluates the expression given by *body*, first creating an ad hoc exception handler that binds a raised exception to *variable* and then evaluates the specified *clause*s as if they were part of a `cond` expression, with the value of the first matching clause becoming the value of the `guard` expression (see Section 6.13.2 [Conditionals], page 299). If none of the clause's test expressions evaluates to `#t`, the exception is re-raised, with the exception handler that was current before the evaluation of the `guard` form.

For example, the expression

```
(guard (ex ((eq? ex 'foo) 'bar) ((eq? ex 'bar) 'baz))
  (raise 'bar))
```

evaluates to `baz`.

`raise` *obj* [Scheme Procedure]
Equivalent to core Guile `(raise-exception obj)`. See Section 6.13.8.2 [Raising and Handling Exceptions], page 315. (Unfortunately, `raise` is already bound to a different function in core Guile. See Section 7.2.8 [Signals], page 524.)

`raise-continuable` *obj* [Scheme Procedure]
Equivalent to core Guile `(raise-exception` *obj* `#:continuable? #t)`. See Section 6.13.8.2 [Raising and Handling Exceptions], page 315.

7.6.2.13 rnrs conditions

The `(rnrs condition (6))` library provides forms and procedures for constructing new condition types, as well as a library of pre-defined condition types that represent a variety of common exceptional situations. Conditions are records of a subtype of the `&condition` record type, which is neither sealed nor opaque. See Section 7.6.2.8 [R6RS Records], page 674.

Conditions may be manipulated singly, as *simple conditions*, or when composed with other conditions to form *compound conditions*. Compound conditions do not "nest"—constructing a new compound condition out of existing compound conditions will "flatten" them into their component simple conditions. For example, making a new condition out of a `&message` condition and a compound condition that contains an `&assertion` condition and another `&message` condition will produce a compound condition that contains two `&message` conditions and one `&assertion` condition.

The record type predicates and field accessors described below can operate on either simple or compound conditions. In the latter case, the predicate returns `#t` if the compound condition contains a component simple condition of the appropriate type; the field accessors return the requisite fields from the first component simple condition found to be of the appropriate type.

Guile's R6RS layer uses core exception types from the `(ice-9 exceptions)` module as the basis for its R6RS condition system. Guile prefers to use the term "exception object" and "exception type" rather than "condition" or "condition type", but that's just a naming difference. Guile also has different names for the types in the condition hierarchy. See Section 6.13.8.1 [Exception Objects], page 311, for full details.

This library is quite similar to the SRFI-35 conditions module (see Section 7.5.24 [SRFI-35], page 626). Among other minor differences, the `(rnrs conditions)` library features slightly different semantics around condition field accessors, and comes with a larger number of pre-defined condition types. The two APIs are compatible; the `condition?` predicate from one API will return `#t` when applied to a condition object created in the other. of the condition types are the same, also.

`&condition` [Condition Type]
`condition?` *obj* [Scheme Procedure]
The base record type for conditions. Known as `&exception` in core Guile.

`condition` *condition1 ...* [Scheme Procedure]
`simple-conditions` *condition* [Scheme Procedure]
The `condition` procedure creates a new compound condition out of its condition arguments, flattening any specified compound conditions into their component simple conditions as described above.

`simple-conditions` returns a list of the component simple conditions of the compound condition `condition`, in the order in which they were specified at construction time.

`condition-predicate` *rtd* [Scheme Procedure]
`condition-accessor` *rtd proc* [Scheme Procedure]
These procedures return condition predicate and accessor procedures for the specified condition record type *rtd*.

`define-condition-type` *condition-type supertype constructor predicate field-spec ...* [Scheme Syntax]
Evaluates to a new record type definition for a condition type with the name *condition-type* that has the condition type *supertype* as its parent. A default constructor, which binds its arguments to the fields of this type and its parent types, will be bound to the identifier *constructor*; a condition predicate will be bound to *predicate*. The fields of the new type, which are immutable, are specified by the *field-spec*s, each of which must be of the form:

```
(field accessor)
```

where *field* gives the name of the field and *accessor* gives the name for a binding to an accessor procedure created for this field.

`&message` [Condition Type]
`make-message-condition` *message* [Scheme Procedure]
`message-condition?` *obj* [Scheme Procedure]
`condition-message` *condition* [Scheme Procedure]
A type that includes a message describing the condition that occurred.

`&warning` [Condition Type]
`make-warning` [Scheme Procedure]
`warning?` *obj* [Scheme Procedure]
A base type for representing non-fatal conditions during execution.

`&serious` [Condition Type]
`make-serious-condition` [Scheme Procedure]
`serious-condition?` *obj* [Scheme Procedure]
A base type for conditions representing errors serious enough that cannot be ignored. Known as `&error` in core Guile.

`&error` [Condition Type]
`make-error` [Scheme Procedure]
`error?` *obj* [Scheme Procedure]
A base type for conditions representing errors. Known as `&external-error` in core Guile.

`&violation` [Condition Type]
`make-violation` [Scheme Procedure]
`violation?` [Scheme Procedure]
A subtype of `&serious` that can be used to represent violations of a language or library standard. Known as `&programming-error` in core Guile.

`&assertion` [Condition Type]
`make-assertion-violation` [Scheme Procedure]

`assertion-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` that indicates an invalid call to a procedure. Known as `&assertion-failure` in core Guile.

`&irritants` [Condition Type]
`make-irritants-condition` *irritants* [Scheme Procedure]
`irritants-condition?` *obj* [Scheme Procedure]
`condition-irritants` *condition* [Scheme Procedure]
A base type used for storing information about the causes of another condition in a compound condition.

`&who` [Condition Type]
`make-who-condition` *who* [Scheme Procedure]
`who-condition?` *obj* [Scheme Procedure]
`condition-who` *condition* [Scheme Procedure]
A base type used for storing the identity, a string or symbol, of the entity responsible for another condition in a compound condition.

`&non-continuable` [Condition Type]
`make-non-continuable-violation` [Scheme Procedure]
`non-continuable-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` used to indicate that an exception handler invoked by `raise` has returned locally.

`&implementation-restriction` [Condition Type]
`make-implementation-restriction-violation` [Scheme Procedure]
`implementation-restriction-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` used to indicate a violation of an implementation restriction.

`&lexical` [Condition Type]
`make-lexical-violation` [Scheme Procedure]
`lexical-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` used to indicate a syntax violation at the level of the datum syntax.

`&syntax` [Condition Type]
`make-syntax-violation` *form subform* [Scheme Procedure]
`syntax-violation?` *obj* [Scheme Procedure]
`syntax-violation-form` *condition* [Scheme Procedure]
`syntax-violation-subform` *condition* [Scheme Procedure]
A subtype of `&violation` that indicates a syntax violation. The *form* and *subform* fields, which must be datum values, indicate the syntactic form responsible for the condition.

`&undefined` [Condition Type]
`make-undefined-violation` [Scheme Procedure]
`undefined-violation?` *obj* [Scheme Procedure]
A subtype of `&violation` that indicates a reference to an unbound identifier. Known as `&undefined-variable` in core Guile.

7.6.2.14 I/O Conditions

These condition types are exported by both the `(rnrs io ports (6))` and `(rnrs io simple (6))` libraries.

`&i/o` [Condition Type]
`make-i/o-error` [Scheme Procedure]
`i/o-error?` *obj* [Scheme Procedure]
A condition supertype for more specific I/O errors.

`&i/o-read` [Condition Type]
`make-i/o-read-error` [Scheme Procedure]
`i/o-read-error?` *obj* [Scheme Procedure]
A subtype of `&i/o`; represents read-related I/O errors.

`&i/o-write` [Condition Type]
`make-i/o-write-error` [Scheme Procedure]
`i/o-write-error?` *obj* [Scheme Procedure]
A subtype of `&i/o`; represents write-related I/O errors.

`&i/o-invalid-position` [Condition Type]
`make-i/o-invalid-position-error` *position* [Scheme Procedure]
`i/o-invalid-position-error?` *obj* [Scheme Procedure]
`i/o-error-position` *condition* [Scheme Procedure]
A subtype of `&i/o`; represents an error related to an attempt to set the file position to an invalid position.

`&i/o-filename` [Condition Type]
`make-io-filename-error` *filename* [Scheme Procedure]
`i/o-filename-error?` *obj* [Scheme Procedure]
`i/o-error-filename` *condition* [Scheme Procedure]
A subtype of `&i/o`; represents an error related to an operation on a named file.

`&i/o-file-protection` [Condition Type]
`make-i/o-file-protection-error` *filename* [Scheme Procedure]
`i/o-file-protection-error?` *obj* [Scheme Procedure]
A subtype of `&i/o-filename`; represents an error resulting from an attempt to access a named file for which the caller had insufficient permissions.

`&i/o-file-is-read-only` [Condition Type]
`make-i/o-file-is-read-only-error` *filename* [Scheme Procedure]
`i/o-file-is-read-only-error?` *obj* [Scheme Procedure]
A subtype of `&i/o-file-protection`; represents an error related to an attempt to write to a read-only file.

`&i/o-file-already-exists` [Condition Type]
`make-i/o-file-already-exists-error` *filename* [Scheme Procedure]
`i/o-file-already-exists-error?` *obj* [Scheme Procedure]
A subtype of `&i/o-filename`; represents an error related to an operation on an existing file that was assumed not to exist.

`&i/o-file-does-not-exist` [Condition Type]
`make-i/o-file-does-not-exist-error` [Scheme Procedure]
`i/o-file-does-not-exist-error?` *obj* [Scheme Procedure]

A subtype of `&i/o-filename`; represents an error related to an operation on a non-existent file that was assumed to exist.

`&i/o-port` [Condition Type]
`make-i/o-port-error` *port* [Scheme Procedure]
`i/o-port-error?` *obj* [Scheme Procedure]
`i/o-error-port` *condition* [Scheme Procedure]

A subtype of `&i/o`; represents an error related to an operation on the port *port*.

7.6.2.15 Transcoders

The transcoder facilities are exported by `(rnrs io ports)`.

Several different Unicode encoding schemes describe standard ways to encode characters and strings as byte sequences and to decode those sequences. Within this document, a *codec* is an immutable Scheme object that represents a Unicode or similar encoding scheme.

An *end-of-line style* is a symbol that, if it is not `none`, describes how a textual port transcodes representations of line endings.

A *transcoder* is an immutable Scheme object that combines a codec with an end-of-line style and a method for handling decoding errors. Each transcoder represents some specific bidirectional (but not necessarily lossless), possibly stateful translation between byte sequences and Unicode characters and strings. Every transcoder can operate in the input direction (bytes to characters) or in the output direction (characters to bytes). A *transcoder* parameter name means that the corresponding argument must be a transcoder.

A *binary port* is a port that supports binary I/O, does not have an associated transcoder and does not support textual I/O. A *textual port* is a port that supports textual I/O, and does not support binary I/O. A textual port may or may not have an associated transcoder.

`latin-1-codec` [Scheme Procedure]
`utf-8-codec` [Scheme Procedure]
`utf-16-codec` [Scheme Procedure]

These are predefined codecs for the ISO 8859-1, UTF-8, and UTF-16 encoding schemes.

A call to any of these procedures returns a value that is equal in the sense of `eqv?` to the result of any other call to the same procedure.

`eol-style` *eol-style-symbol* [Scheme Syntax]

eol-style-symbol should be a symbol whose name is one of `lf`, `cr`, `crlf`, `nel`, `crnel`, `ls`, and `none`.

The form evaluates to the corresponding symbol. If the name of *eol-style-symbol* is not one of these symbols, the effect and result are implementation-dependent; in particular, the result may be an eol-style symbol acceptable as an *eol-style* argument to `make-transcoder`. Otherwise, an exception is raised.

All eol-style symbols except `none` describe a specific line-ending encoding:

`lf` linefeed

`cr`	carriage return
`crlf`	carriage return, linefeed
`nel`	next line
`crnel`	carriage return, next line
`ls`	line separator

For a textual port with a transcoder, and whose transcoder has an eol-style symbol `none`, no conversion occurs. For a textual input port, any eol-style symbol other than `none` means that all of the above line-ending encodings are recognized and are translated into a single linefeed. For a textual output port, `none` and `lf` are equivalent. Linefeed characters are encoded according to the specified eol-style symbol, and all other characters that participate in possible line endings are encoded as is.

Note: Only the name of *eol-style-symbol* is significant.

`native-eol-style` [Scheme Procedure]

Returns the default end-of-line style of the underlying platform, e.g., `lf` on Unix and `crlf` on Windows.

`&i/o-decoding` [Condition Type]
`make-i/o-decoding-error` *port* [Scheme Procedure]
`i/o-decoding-error?` *obj* [Scheme Procedure]

This condition type could be defined by

```
(define-condition-type &i/o-decoding &i/o-port
  make-i/o-decoding-error i/o-decoding-error?)
```

An exception with this type is raised when one of the operations for textual input from a port encounters a sequence of bytes that cannot be translated into a character or string by the input direction of the port's transcoder.

When such an exception is raised, the port's position is past the invalid encoding.

`&i/o-encoding` [Condition Type]
`make-i/o-encoding-error` *port char* [Scheme Procedure]
`i/o-encoding-error?` *obj* [Scheme Procedure]
`i/o-encoding-error-char` *condition* [Scheme Procedure]

This condition type could be defined by

```
(define-condition-type &i/o-encoding &i/o-port
  make-i/o-encoding-error i/o-encoding-error?
  (char i/o-encoding-error-char))
```

An exception with this type is raised when one of the operations for textual output to a port encounters a character that cannot be translated into bytes by the output direction of the port's transcoder. *char* is the character that could not be encoded.

`error-handling-mode` *error-handling-mode-symbol* [Scheme Syntax]

error-handling-mode-symbol should be a symbol whose name is one of `ignore`, `raise`, and `replace`. The form evaluates to the corresponding symbol. If *error-handling-mode-symbol* is not one of these identifiers, effect and result are implementation-dependent: The result may be an error-handling-mode symbol

acceptable as a *handling-mode* argument to `make-transcoder`. If it is not acceptable as a *handling-mode* argument to `make-transcoder`, an exception is raised.

Note: Only the name of *error-handling-mode-symbol* is significant.

The error-handling mode of a transcoder specifies the behavior of textual I/O operations in the presence of encoding or decoding errors.

If a textual input operation encounters an invalid or incomplete character encoding, and the error-handling mode is `ignore`, an appropriate number of bytes of the invalid encoding are ignored and decoding continues with the following bytes.

If the error-handling mode is `replace`, the replacement character U+FFFD is injected into the data stream, an appropriate number of bytes are ignored, and decoding continues with the following bytes.

If the error-handling mode is `raise`, an exception with condition type `&i/o-decoding` is raised.

If a textual output operation encounters a character it cannot encode, and the error-handling mode is `ignore`, the character is ignored and encoding continues with the next character. If the error-handling mode is `replace`, a codec-specific replacement character is emitted by the transcoder, and encoding continues with the next character. The replacement character is U+FFFD for transcoders whose codec is one of the Unicode encodings, but is the `?` character for the Latin-1 encoding. If the error-handling mode is `raise`, an exception with condition type `&i/o-encoding` is raised.

`make-transcoder` *codec* [Scheme Procedure]
`make-transcoder` *codec eol-style* [Scheme Procedure]
`make-transcoder` *codec eol-style handling-mode* [Scheme Procedure]

codec must be a codec; *eol-style*, if present, an eol-style symbol; and *handling-mode*, if present, an error-handling-mode symbol.

eol-style may be omitted, in which case it defaults to the native end-of-line style of the underlying platform. *handling-mode* may be omitted, in which case it defaults to `replace`. The result is a transcoder with the behavior specified by its arguments.

`native-transcoder` [Scheme procedure]

Returns an implementation-dependent transcoder that represents a possibly locale-dependent "native" transcoding.

`transcoder-codec` *transcoder* [Scheme Procedure]
`transcoder-eol-style` *transcoder* [Scheme Procedure]
`transcoder-error-handling-mode` *transcoder* [Scheme Procedure]

These are accessors for transcoder objects; when applied to a transcoder returned by `make-transcoder`, they return the *codec*, *eol-style*, and *handling-mode* arguments, respectively.

`bytevector->string` *bytevector transcoder* [Scheme Procedure]

Returns the string that results from transcoding the *bytevector* according to the input direction of the transcoder.

`string->bytevector` *string transcoder* [Scheme Procedure]
Returns the bytevector that results from transcoding the *string* according to the output direction of the transcoder.

7.6.2.16 rnrs io ports

Guile's binary and textual port interface was heavily inspired by R6RS, so many R6RS port interfaces are documented elsewhere. Note that R6RS ports are not disjoint from Guile's native ports, so Guile-specific procedures will work on ports created using the R6RS API, and vice versa. Also note that in Guile, all ports are both textual and binary. See Section 6.14 [Input and Output], page 331, for more on Guile's core port API. The R6RS ports module wraps Guile's I/O routines in a helper that will translate native Guile exceptions to R6RS conditions; See Section 7.6.2.14 [R6RS I/O Conditions], page 682, for more. See Section 7.6.2.17 [R6RS File Ports], page 689, for documentation on the R6RS file port interface.

Note: The implementation of this R6RS API is not complete yet.

`eof-object?` *obj* [Scheme Procedure]
See Section 6.14.2 [Binary I/O], page 332, for documentation.

`eof-object` [Scheme Procedure]
Return the end-of-file (EOF) object.

```
(eof-object? (eof-object))
⇒ #t
```

`port?` *obj* [Scheme Procedure]
`input-port?` *obj* [Scheme Procedure]
`output-port?` *obj* [Scheme Procedure]
See Section 6.14.1 [Ports], page 331, for documentation.

`port-transcoder` *port* [Scheme Procedure]
Return a transcoder associated with the encoding of *port*. See Section 6.14.3 [Encoding], page 334, and See Section 7.6.2.15 [R6RS Transcoders], page 683.

`binary-port?` *port* [Scheme Procedure]
Return `#t` if *port* appears to be a binary port, else return `#f`. Note that Guile does not currently distinguish between binary and textual ports, so this predicate is not a reliable indicator of whether the port was created as a binary port. Currently, it returns `#t` if and only if the port encoding is "ISO-8859-1", because Guile uses this encoding when creating a binary port. See Section 6.14.3 [Encoding], page 334, for more details.

`textual-port?` *port* [Scheme Procedure]
Return `#t` if *port* appears to be a textual port, else return `#f`. Note that Guile does not currently distinguish between binary and textual ports, so this predicate is not a reliable indicator of whether the port was created as a textual port. Currently, it always returns `#t`, because all ports can be used for textual I/O in Guile. See Section 6.14.3 [Encoding], page 334, for more details.

`transcoded-port` *binary-port transcoder* [Scheme Procedure]
The `transcoded-port` procedure returns a new textual port with the specified *transcoder*. Otherwise the new textual port's state is largely the same as that of *binary-port*. If *binary-port* is an input port, the new textual port will be an input port and will transcode the bytes that have not yet been read from *binary-port*. If *binary-port* is an output port, the new textual port will be an output port and will transcode output characters into bytes that are written to the byte sink represented by *binary-port*.

As a side effect, however, `transcoded-port` closes *binary-port* in a special way that allows the new textual port to continue to use the byte source or sink represented by *binary-port*, even though *binary-port* itself is closed and cannot be used by the input and output operations described in this chapter.

`port-position` *port* [Scheme Procedure]
Equivalent to `(seek` *port* `0 SEEK_CUR)`. See Section 6.14.7 [Random Access], page 340.

`port-has-port-position?` *port* [Scheme Procedure]
Return `#t` is *port* supports `port-position`.

`set-port-position!` *port offset* [Scheme Procedure]
Equivalent to `(seek` *port offset* `SEEK_SET)`. See Section 6.14.7 [Random Access], page 340.

`port-has-set-port-position!?` *port* [Scheme Procedure]
Return `#t` is *port* supports `set-port-position!`.

`call-with-port` *port proc* [Scheme Procedure]
Call *proc*, passing it *port* and closing *port* upon exit of *proc*. Return the return values of *proc*.

`port-eof?` *input-port* [Scheme Procedure]
Equivalent to `(eof-object? (lookahead-u8` *input-port*`))`.

`standard-input-port` [Scheme Procedure]
`standard-output-port` [Scheme Procedure]
`standard-error-port` [Scheme Procedure]
Returns a fresh binary input port connected to standard input, or a binary output port connected to the standard output or standard error, respectively. Whether the port supports the `port-position` and `set-port-position!` operations is implementation-dependent.

`current-input-port` [Scheme Procedure]
`current-output-port` [Scheme Procedure]
`current-error-port` [Scheme Procedure]
See Section 6.14.9 [Default Ports], page 343.

`open-bytevector-input-port` *bv* [*transcoder*] [Scheme Procedure]
`open-bytevector-output-port` [*transcoder*] [Scheme Procedure]
See Section 6.14.10.2 [Bytevector Ports], page 347.

`make-custom-binary-input-port` *id read! get-position set-position! close* [Scheme Procedure]
`make-custom-binary-output-port` *id write! get-position set-position! close* [Scheme Procedure]
`make-custom-binary-input/output-port` *id read! write! get-position set-position! close* [Scheme Procedure]
See Section 6.14.10.4 [Custom Ports], page 348.

`get-u8` *port* [Scheme Procedure]
`lookahead-u8` *port* [Scheme Procedure]
`get-bytevector-n` *port count* [Scheme Procedure]
`get-bytevector-n!` *port bv start count* [Scheme Procedure]
`get-bytevector-some` *port* [Scheme Procedure]
`get-bytevector-all` *port* [Scheme Procedure]
`put-u8` *port octet* [Scheme Procedure]
`put-bytevector` *port bv* [*start* [*count*]] [Scheme Procedure]
See Section 6.14.2 [Binary I/O], page 332.

`get-char` *textual-input-port* [Scheme Procedure]
`lookahead-char` *textual-input-port* [Scheme Procedure]
`get-string-n` *textual-input-port count* [Scheme Procedure]
`get-string-n!` *textual-input-port string start count* [Scheme Procedure]
`get-string-all` *textual-input-port* [Scheme Procedure]
`get-line` *textual-input-port* [Scheme Procedure]
`put-char` *port char* [Scheme Procedure]
`put-string` *port string* [*start* [*count*]] [Scheme Procedure]
See Section 6.14.4 [Textual I/O], page 336.

`get-datum` *textual-input-port count* [Scheme Procedure]
Reads an external representation from *textual-input-port* and returns the datum it represents. The `get-datum` procedure returns the next datum that can be parsed from the given *textual-input-port*, updating *textual-input-port* to point exactly past the end of the external representation of the object.

Any *interlexeme space* (comment or whitespace, see Section 6.18.1 [Scheme Syntax], page 382) in the input is first skipped. If an end of file occurs after the interlexeme space, the end-of-file object is returned.

If a character inconsistent with an external representation is encountered in the input, an exception with condition types `&lexical` and `&i/o-read` is raised. Also, if the end of file is encountered after the beginning of an external representation, but the external representation is incomplete and therefore cannot be parsed, an exception with condition types `&lexical` and `&i/o-read` is raised.

`put-datum` *textual-output-port datum* [Scheme Procedure]
datum should be a datum value. The `put-datum` procedure writes an external representation of *datum* to *textual-output-port*. The specific external representation is implementation-dependent. However, whenever possible, an implementation should produce a representation for which `get-datum`, when reading the representation, will return an object equal (in the sense of `equal?`) to *datum*.

Note: Not all datums may allow producing an external representation for which `get-datum` will produce an object that is equal to the original. Specifically, NaNs contained in *datum* may make this impossible.

Note: The `put-datum` procedure merely writes the external representation, but no trailing delimiter. If `put-datum` is used to write several subsequent external representations to an output port, care should be taken to delimit them properly so they can be read back in by subsequent calls to `get-datum`.

`flush-output-port` *port* [Scheme Procedure]

See Section 6.14.6 [Buffering], page 339, for documentation on `force-output`.

7.6.2.17 R6RS File Ports

The facilities described in this section are exported by the `(rnrs io ports)` module.

`buffer-mode` *buffer-mode-symbol* [Scheme Syntax]

buffer-mode-symbol must be a symbol whose name is one of `none`, `line`, and `block`. The result is the corresponding symbol, and specifies the associated buffer mode. See Section 6.14.6 [Buffering], page 339, for a discussion of these different buffer modes. To control the amount of buffering, use `setvbuf` instead. Note that only the name of *buffer-mode-symbol* is significant.

See Section 6.14.6 [Buffering], page 339, for a discussion of port buffering.

`buffer-mode?` *obj* [Scheme Procedure]

Returns `#t` if the argument is a valid buffer-mode symbol, and returns `#f` otherwise.

When opening a file, the various procedures accept a `file-options` object that encapsulates flags to specify how the file is to be opened. A `file-options` object is an enum-set (see Section 7.6.2.26 [rnrs enums], page 702) over the symbols constituting valid file options.

A *file-options* parameter name means that the corresponding argument must be a file-options object.

`file-options` *file-options-symbol* ... [Scheme Syntax]

Each *file-options-symbol* must be a symbol.

The `file-options` syntax returns a file-options object that encapsulates the specified options.

When supplied to an operation that opens a file for output, the file-options object returned by `(file-options)` specifies that the file is created if it does not exist and an exception with condition type `&i/o-file-already-exists` is raised if it does exist. The following standard options can be included to modify the default behavior.

`no-create`

If the file does not already exist, it is not created; instead, an exception with condition type `&i/o-file-does-not-exist` is raised. If the file already exists, the exception with condition type `&i/o-file-already-exists` is not raised and the file is truncated to zero length.

`no-fail` If the file already exists, the exception with condition type `&i/o-file-already-exists` is not raised, even if `no-create` is not included, and the file is truncated to zero length.

`no-truncate`
If the file already exists and the exception with condition type `&i/o-file-already-exists` has been inhibited by inclusion of `no-create` or `no-fail`, the file is not truncated, but the port's current position is still set to the beginning of the file.

These options have no effect when a file is opened only for input. Symbols other than those listed above may be used as *file-options-symbol*s; they have implementation-specific meaning, if any.

Note: Only the name of *file-options-symbol* is significant.

`open-file-input-port` *filename* [Scheme Procedure]
`open-file-input-port` *filename file-options* [Scheme Procedure]
`open-file-input-port` *filename file-options buffer-mode* [Scheme Procedure]
`open-file-input-port` *filename file-options buffer-mode maybe-transcoder* [Scheme Procedure]

maybe-transcoder must be either a transcoder or `#f`.

The `open-file-input-port` procedure returns an input port for the named file. The *file-options* and *maybe-transcoder* arguments are optional.

The *file-options* argument, which may determine various aspects of the returned port, defaults to the value of `(file-options)`.

The *buffer-mode* argument, if supplied, must be one of the symbols that name a buffer mode. The *buffer-mode* argument defaults to `block`.

If *maybe-transcoder* is a transcoder, it becomes the transcoder associated with the returned port.

If *maybe-transcoder* is `#f` or absent, the port will be a binary port and will support the `port-position` and `set-port-position!` operations. Otherwise the port will be a textual port, and whether it supports the `port-position` and `set-port-position!` operations is implementation-dependent (and possibly transcoder-dependent).

`open-file-output-port` *filename* [Scheme Procedure]
`open-file-output-port` *filename file-options* [Scheme Procedure]
`open-file-output-port` *filename file-options buffer-mode* [Scheme Procedure]
`open-file-output-port` *filename file-options buffer-mode maybe-transcoder* [Scheme Procedure]

maybe-transcoder must be either a transcoder or `#f`.

The `open-file-output-port` procedure returns an output port for the named file.

The *file-options* argument, which may determine various aspects of the returned port, defaults to the value of `(file-options)`.

The *buffer-mode* argument, if supplied, must be one of the symbols that name a buffer mode. The *buffer-mode* argument defaults to `block`.

If *maybe-transcoder* is a transcoder, it becomes the transcoder associated with the port.

If *maybe-transcoder* is `#f` or absent, the port will be a binary port and will support the `port-position` and `set-port-position!` operations. Otherwise the port will be a textual port, and whether it supports the `port-position` and `set-port-position!` operations is implementation-dependent (and possibly transcoder-dependent).

7.6.2.18 rnrs io simple

The `(rnrs io simple (6))` library provides convenience functions for performing textual I/O on ports. This library also exports all of the condition types and associated procedures described in (see Section 7.6.2.14 [R6RS I/O Conditions], page 682). In the context of this section, when stating that a procedure behaves "identically" to the corresponding procedure in Guile's core library, this is modulo the behavior wrt. conditions: such procedures raise the appropriate R6RS conditions in case of error, but otherwise behave identically.

> **Note:** There are still known issues regarding condition-correctness; some errors may still be thrown as native Guile exceptions instead of the appropriate R6RS conditions.

`eof-object` [Scheme Procedure]
`eof-object?` *obj* [Scheme Procedure]

These procedures are identical to the ones provided by the `(rnrs io ports (6))` library. See Section 7.6.2.16 [rnrs io ports], page 686, for documentation.

`input-port?` *obj* [Scheme Procedure]
`output-port?` *obj* [Scheme Procedure]

These procedures are identical to the ones provided by Guile's core library. See Section 6.14.1 [Ports], page 331, for documentation.

`call-with-input-file` *filename proc* [Scheme Procedure]
`call-with-output-file` *filename proc* [Scheme Procedure]
`open-input-file` *filename* [Scheme Procedure]
`open-output-file` *filename* [Scheme Procedure]
`with-input-from-file` *filename thunk* [Scheme Procedure]
`with-output-to-file` *filename thunk* [Scheme Procedure]

These procedures are identical to the ones provided by Guile's core library. See Section 6.14.10.1 [File Ports], page 344, for documentation.

`close-input-port` *input-port* [Scheme Procedure]
`close-output-port` *output-port* [Scheme Procedure]

Closes the given *input-port* or *output-port*. These are legacy interfaces; just use `close-port`.

`peek-char` [Scheme Procedure]
`peek-char` *textual-input-port* [Scheme Procedure]
`read-char` [Scheme Procedure]
`read-char` *textual-input-port* [Scheme Procedure]

These procedures are identical to the ones provided by Guile's core library. See Section 6.14.11 [Venerable Port Interfaces], page 350, for documentation.

`read` [Scheme Procedure]
`read` *textual-input-port* [Scheme Procedure]
This procedure is identical to the one provided by Guile's core library. See Section 6.18.2 [Scheme Read], page 385, for documentation.

`display` *obj* [Scheme Procedure]
`display` *obj textual-output-port* [Scheme Procedure]
`newline` [Scheme Procedure]
`newline` *textual-output-port* [Scheme Procedure]
`write` *obj* [Scheme Procedure]
`write` *obj textual-output-port* [Scheme Procedure]
`write-char` *char* [Scheme Procedure]
`write-char` *char textual-output-port* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. See Section 6.14.11 [Venerable Port Interfaces], page 350, and See Section 6.18.3 [Scheme Write], page 386, for documentation.

7.6.2.19 rnrs files

The `(rnrs files (6))` library provides the `file-exists?` and `delete-file` procedures, which test for the existence of a file and allow the deletion of files from the file system, respectively.

These procedures are identical to the ones provided by Guile's core library. See Section 7.2.3 [File System], page 504, for documentation.

7.6.2.20 rnrs programs

The `(rnrs programs (6))` library provides procedures for process management and introspection.

`command-line` [Scheme Procedure]
This procedure is identical to the one provided by Guile's core library. See Section 7.2.6 [Runtime Environment], page 517, for documentation.

`exit` [*status*] [Scheme Procedure]
This procedure is identical to the one provided by Guile's core library. See Section 7.2.7 [Processes], page 518, for documentation.

7.6.2.21 rnrs arithmetic fixnums

The `(rnrs arithmetic fixnums (6))` library provides procedures for performing arithmetic operations on an implementation-dependent range of exact integer values, which R6RS refers to as *fixnums*. In Guile, the size of a fixnum is determined by the size of the `SCM` type; a single SCM struct is guaranteed to be able to hold an entire fixnum, making fixnum computations particularly efficient—(see Section 6.3 [The SCM Type], page 100). On 32-bit systems, the most negative and most positive fixnum values are, respectively, -536870912 and 536870911.

Unless otherwise specified, all of the procedures below take fixnums as arguments, and will raise an `&assertion` condition if passed a non-fixnum argument or an `&implementation-restriction` condition if their result is not itself a fixnum.

`fixnum?` *obj* [Scheme Procedure]
Returns `#t` if *obj* is a fixnum, `#f` otherwise.

`fixnum-width` [Scheme Procedure]
`least-fixnum` [Scheme Procedure]
`greatest-fixnum` [Scheme Procedure]
These procedures return, respectively, the maximum number of bits necessary to represent a fixnum value in Guile, the minimum fixnum value, and the maximum fixnum value.

`fx=?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx>?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx<?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx>=?` *fx1 fx2 fx3 ...* [Scheme Procedure]
`fx<=?` *fx1 fx2 fx3 ...* [Scheme Procedure]
These procedures return `#t` if their fixnum arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; `#f` otherwise.

`fxzero?` *fx* [Scheme Procedure]
`fxpositive?` *fx* [Scheme Procedure]
`fxnegative?` *fx* [Scheme Procedure]
`fxodd?` *fx* [Scheme Procedure]
`fxeven?` *fx* [Scheme Procedure]
These numerical predicates return `#t` if *fx* is, respectively, zero, greater than zero, less than zero, odd, or even; `#f` otherwise.

`fxmax` *fx1 fx2 ...* [Scheme Procedure]
`fxmin` *fx1 fx2 ...* [Scheme Procedure]
These procedures return the maximum or minimum of their arguments.

`fx+` *fx1 fx2* [Scheme Procedure]
`fx*` *fx1 fx2* [Scheme Procedure]
These procedures return the sum or product of their arguments.

`fx-` *fx1 fx2* [Scheme Procedure]
`fx-` *fx* [Scheme Procedure]
Returns the difference of *fx1* and *fx2*, or the negation of *fx*, if called with a single argument.

An `&assertion` condition is raised if the result is not itself a fixnum.

`fxdiv-and-mod` *fx1 fx2* [Scheme Procedure]
`fxdiv` *fx1 fx2* [Scheme Procedure]
`fxmod` *fx1 fx2* [Scheme Procedure]
`fxdiv0-and-mod0` *fx1 fx2* [Scheme Procedure]
`fxdiv0` *fx1 fx2* [Scheme Procedure]
`fxmod0` *fx1 fx2* [Scheme Procedure]
These procedures implement number-theoretic division on fixnums; See ⟨undefined⟩ [(rnrs base)], page ⟨undefined⟩, for a description of their semantics.

`fx+/carry` *fx1 fx2 fx3* [Scheme Procedure]

Returns the two fixnum results of the following computation:

```
(let* ((s (+ fx1 fx2 fx3))
       (s0 (mod0 s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))
```

`fx-/carry` *fx1 fx2 fx3* [Scheme Procedure]

Returns the two fixnum results of the following computation:

```
(let* ((d (- fx1 fx2 fx3))
       (d0 (mod0 d (expt 2 (fixnum-width))))
       (d1 (div0 d (expt 2 (fixnum-width)))))
  (values d0 d1))
```

`fx*/carry` *fx1 fx2 fx3* [Scheme Procedure]

```
Returns the two fixnum results of the following computation:
(let* ((s (+ (* fx1 fx2) fx3))
       (s0 (mod0 s (expt 2 (fixnum-width))))
       (s1 (div0 s (expt 2 (fixnum-width)))))
  (values s0 s1))
```

`fxnot` *fx* [Scheme Procedure]
`fxand` *fx1 ...* [Scheme Procedure]
`fxior` *fx1 ...* [Scheme Procedure]
`fxxor` *fx1 ...* [Scheme Procedure]

These procedures are identical to the `lognot`, `logand`, `logior`, and `logxor` procedures provided by Guile's core library. See Section 6.6.2.13 [Bitwise Operations], page 125, for documentation.

`fxif` *fx1 fx2 fx3* [Scheme Procedure]

Returns the bitwise "if" of its fixnum arguments. The bit at position `i` in the return value will be the `i`th bit from *fx2* if the `i`th bit of *fx1* is 1, the `i`th bit from *fx3*.

`fxbit-count` *fx* [Scheme Procedure]

Returns the number of 1 bits in the two's complement representation of *fx*.

`fxlength` *fx* [Scheme Procedure]

Returns the number of bits necessary to represent *fx*.

`fxfirst-bit-set` *fx* [Scheme Procedure]

Returns the index of the least significant 1 bit in the two's complement representation of *fx*.

`fxbit-set?` *fx1 fx2* [Scheme Procedure]

Returns `#t` if the *fx2*th bit in the two's complement representation of *fx1* is 1, `#f` otherwise.

`fxcopy-bit` *fx1 fx2 fx3* [Scheme Procedure]

Returns the result of setting the *fx2*th bit of *fx1* to the *fx2*th bit of *fx3*.

`fxbit-field` *fx1 fx2 fx3* [Scheme Procedure]
Returns the integer representation of the contiguous sequence of bits in *fx1* that starts at position *fx2* (inclusive) and ends at position *fx3* (exclusive).

`fxcopy-bit-field` *fx1 fx2 fx3 fx4* [Scheme Procedure]
Returns the result of replacing the bit field in *fx1* with start and end positions *fx2* and *fx3* with the corresponding bit field from *fx4*.

`fxarithmetic-shift` *fx1 fx2* [Scheme Procedure]
`fxarithmetic-shift-left` *fx1 fx2* [Scheme Procedure]
`fxarithmetic-shift-right` *fx1 fx2* [Scheme Procedure]
Returns the result of shifting the bits of *fx1* right or left by the *fx2* positions. `fxarithmetic-shift` is identical to `fxarithmetic-shift-left`.

`fxrotate-bit-field` *fx1 fx2 fx3 fx4* [Scheme Procedure]
Returns the result of cyclically permuting the bit field in *fx1* with start and end positions *fx2* and *fx3* by *fx4* bits in the direction of more significant bits.

`fxreverse-bit-field` *fx1 fx2 fx3* [Scheme Procedure]
Returns the result of reversing the order of the bits of *fx1* between position *fx2* (inclusive) and position *fx3* (exclusive).

7.6.2.22 rnrs arithmetic flonums

The `(rnrs arithmetic flonums (6))` library provides procedures for performing arithmetic operations on inexact representations of real numbers, which R6RS refers to as *flonums*.

Unless otherwise specified, all of the procedures below take flonums as arguments, and will raise an `&assertion` condition if passed a non-flonum argument.

`flonum?` *obj* [Scheme Procedure]
Returns `#t` if *obj* is a flonum, `#f` otherwise.

`real->flonum` *x* [Scheme Procedure]
Returns the flonum that is numerically closest to the real number *x*.

`fl=?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl<?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl<=?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl>?` *fl1 fl2 fl3 ...* [Scheme Procedure]
`fl>=?` *fl1 fl2 fl3 ...* [Scheme Procedure]
These procedures return `#t` if their flonum arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing; `#f` otherwise.

`flinteger?` *fl* [Scheme Procedure]
`flzero?` *fl* [Scheme Procedure]
`flpositive?` *fl* [Scheme Procedure]
`flnegative?` *fl* [Scheme Procedure]
`flodd?` *fl* [Scheme Procedure]

`fleven?` *fl* [Scheme Procedure]

These numerical predicates return `#t` if *fl* is, respectively, an integer, zero, greater than zero, less than zero, odd, even, `#f` otherwise. In the case of `flodd?` and `fleven?`, *fl* must be an integer-valued flonum.

`flfinite?` *fl* [Scheme Procedure]
`flinfinite?` *fl* [Scheme Procedure]
`flnan?` *fl* [Scheme Procedure]

These numerical predicates return `#t` if *fl* is, respectively, not infinite, infinite, or a `NaN` value.

`flmax` *fl1 fl2 ...* [Scheme Procedure]
`flmin` *fl1 fl2 ...* [Scheme Procedure]

These procedures return the maximum or minimum of their arguments.

`fl+` *fl1 ...* [Scheme Procedure]
`fl*` *fl ...* [Scheme Procedure]

These procedures return the sum or product of their arguments.

`fl-` *fl1 fl2 ...* [Scheme Procedure]
`fl-` *fl* [Scheme Procedure]
`fl/` *fl1 fl2 ...* [Scheme Procedure]
`fl/` *fl* [Scheme Procedure]

These procedures return, respectively, the difference or quotient of their arguments when called with two arguments; when called with a single argument, they return the additive or multiplicative inverse of *fl*.

`flabs` *fl* [Scheme Procedure]

Returns the absolute value of *fl*.

`fldiv-and-mod` *fl1 fl2* [Scheme Procedure]
`fldiv` *fl1 fl2* [Scheme Procedure]
`fldmod` *fl1 fl2* [Scheme Procedure]
`fldiv0-and-mod0` *fl1 fl2* [Scheme Procedure]
`fldiv0` *fl1 fl2* [Scheme Procedure]
`flmod0` *fl1 fl2* [Scheme Procedure]

These procedures implement number-theoretic division on flonums; See ⟨undefined⟩ [(rnrs base)], page ⟨undefined⟩, for a description for their semantics.

`flnumerator` *fl* [Scheme Procedure]
`fldenominator` *fl* [Scheme Procedure]

These procedures return the numerator or denominator of *fl* as a flonum.

`flfloor` *fl1* [Scheme Procedure]
`flceiling` *fl* [Scheme Procedure]
`fltruncate` *fl* [Scheme Procedure]
`flround` *fl* [Scheme Procedure]

These procedures are identical to the `floor`, `ceiling`, `truncate`, and `round` procedures provided by Guile's core library. See Section 6.6.2.11 [Arithmetic], page 119, for documentation.

`flexp` *fl* [Scheme Procedure]
`fllog` *fl* [Scheme Procedure]
`fllog` *fl1 fl2* [Scheme Procedure]
`flsin` *fl* [Scheme Procedure]
`flcos` *fl* [Scheme Procedure]
`fltan` *fl* [Scheme Procedure]
`flasin` *fl* [Scheme Procedure]
`flacos` *fl* [Scheme Procedure]
`flatan` *fl* [Scheme Procedure]
`flatan` *fl1 fl2* [Scheme Procedure]

These procedures, which compute the usual transcendental functions, are the flonum variants of the procedures provided by the R6RS base library (see ⟨undefined⟩ [(rnrs base)], page ⟨undefined⟩).

`flsqrt` *fl* [Scheme Procedure]

Returns the square root of *fl*. If *fl* is `-0.0`, *-0.0* is returned; for other negative values, a `NaN` value is returned.

`flexpt` *fl1 fl2* [Scheme Procedure]

Returns the value of *fl1* raised to the power of *fl2*.

The following condition types are provided to allow Scheme implementations that do not support infinities or `NaN` values to indicate that a computation resulted in such a value. Guile supports both of these, so these conditions will never be raised by Guile's standard libraries implementation.

`&no-infinities` [Condition Type]
`make-no-infinities-violation` *obj* [Scheme Procedure]
`no-infinities-violation?` [Scheme Procedure]

A condition type indicating that a computation resulted in an infinite value on a Scheme implementation incapable of representing infinities.

`&no-nans` [Condition Type]
`make-no-nans-violation` *obj* [Scheme Procedure]
`no-nans-violation?` *obj* [Scheme Procedure]

A condition type indicating that a computation resulted in a `NaN` value on a Scheme implementation incapable of representing `NaN`s.

`fixnum->flonum` *fx* [Scheme Procedure]

Returns the flonum that is numerically closest to the fixnum *fx*.

7.6.2.23 rnrs arithmetic bitwise

The `(rnrs arithmetic bitwise (6))` library provides procedures for performing bitwise arithmetic operations on the two's complement representations of fixnums.

This library and the procedures it exports share functionality with SRFI-60, which provides support for bitwise manipulation of integers (see Section 7.5.34 [SRFI-60], page 648).

`bitwise-not` *ei* [Scheme Procedure]
`bitwise-and` *ei1 ...* [Scheme Procedure]

`bitwise-ior` *ei1 ...* [Scheme Procedure]
`bitwise-xor` *ei1 ...* [Scheme Procedure]
These procedures are identical to the `lognot`, `logand`, `logior`, and `logxor` procedures provided by Guile's core library. See Section 6.6.2.13 [Bitwise Operations], page 125, for documentation.

`bitwise-if` *ei1 ei2 ei3* [Scheme Procedure]
Returns the bitwise "if" of its arguments. The bit at position `i` in the return value will be the `i`th bit from *ei2* if the `i`th bit of *ei1* is 1, the `i`th bit from *ei3*.

`bitwise-bit-count` *ei* [Scheme Procedure]
Returns the number of 1 bits in the two's complement representation of *ei*.

`bitwise-length` *ei* [Scheme Procedure]
Returns the number of bits necessary to represent *ei*.

`bitwise-first-bit-set` *ei* [Scheme Procedure]
Returns the index of the least significant 1 bit in the two's complement representation of *ei*.

`bitwise-bit-set?` *ei1 ei2* [Scheme Procedure]
Returns `#t` if the *ei2*th bit in the two's complement representation of *ei1* is 1, `#f` otherwise.

`bitwise-copy-bit` *ei1 ei2 ei3* [Scheme Procedure]
Returns the result of setting the *ei2*th bit of *ei1* to the *ei2*th bit of *ei3*.

`bitwise-bit-field` *ei1 ei2 ei3* [Scheme Procedure]
Returns the integer representation of the contiguous sequence of bits in *ei1* that starts at position *ei2* (inclusive) and ends at position *ei3* (exclusive).

`bitwise-copy-bit-field` *ei1 ei2 ei3 ei4* [Scheme Procedure]
Returns the result of replacing the bit field in *ei1* with start and end positions *ei2* and *ei3* with the corresponding bit field from *ei4*.

`bitwise-arithmetic-shift` *ei1 ei2* [Scheme Procedure]
`bitwise-arithmetic-shift-left` *ei1 ei2* [Scheme Procedure]
`bitwise-arithmetic-shift-right` *ei1 ei2* [Scheme Procedure]
Returns the result of shifting the bits of *ei1* right or left by the *ei2* positions. `bitwise-arithmetic-shift` is identical to `bitwise-arithmetic-shift-left`.

`bitwise-rotate-bit-field` *ei1 ei2 ei3 ei4* [Scheme Procedure]
Returns the result of cyclically permuting the bit field in *ei1* with start and end positions *ei2* and *ei3* by *ei4* bits in the direction of more significant bits.

`bitwise-reverse-bit-field` *ei1 ei2 ei3* [Scheme Procedure]
Returns the result of reversing the order of the bits of *ei1* between position *ei2* (inclusive) and position *ei3* (exclusive).

7.6.2.24 rnrs syntax-case

The `(rnrs syntax-case (6))` library provides access to the `syntax-case` system for writing hygienic macros. With one exception, all of the forms and procedures exported by this library are "re-exports" of Guile's native support for `syntax-case`; See Section 6.10.3 [Syntax Case], page 268, for documentation, examples, and rationale.

`make-variable-transformer` *proc* [Scheme Procedure]

Creates a new variable transformer out of *proc*, a procedure that takes a syntax object as input and returns a syntax object. If an identifier to which the result of this procedure is bound appears on the left-hand side of a `set!` expression, *proc* will be called with a syntax object representing the entire `set!` expression, and its return value will replace that `set!` expression.

`syntax-case` *expression* (*literal ...*) *clause ...* [Scheme Syntax]

The `syntax-case` pattern matching form.

`syntax` *template* [Scheme Syntax]
`quasisyntax` *template* [Scheme Syntax]
`unsyntax` *template* [Scheme Syntax]
`unsyntax-splicing` *template* [Scheme Syntax]

These forms allow references to be made in the body of a syntax-case output expression subform to datum and non-datum values. They are identical to the forms provided by Guile's core library; See Section 6.10.3 [Syntax Case], page 268, for documentation.

`identifier?` *obj* [Scheme Procedure]
`bound-identifier=?` *id1 id2* [Scheme Procedure]
`free-identifier=?` *id1 id2* [Scheme Procedure]

These predicate procedures operate on syntax objects representing Scheme identifiers. `identifier?` returns `#t` if *obj* represents an identifier, `#f` otherwise. `bound-identifier=?` returns `#t` if and only if a binding for *id1* would capture a reference to *id2* in the transformer's output, or vice-versa. `free-identifier=?` returns `#t` if and only *id1* and *id2* would refer to the same binding in the output of the transformer, independent of any bindings introduced by the transformer.

`generate-temporaries` *l* [Scheme Procedure]

Returns a list, of the same length as *l*, which must be a list or a syntax object representing a list, of globally unique symbols.

`syntax->datum` *syntax-object* [Scheme Procedure]
`datum->syntax` *template-id datum* [Scheme Procedure]

These procedures convert wrapped syntax objects to and from Scheme datum values. The syntax object returned by `datum->syntax` shares contextual information with the syntax object *template-id*.

`syntax-violation` *whom message form* [Scheme Procedure]
`syntax-violation` *whom message form subform* [Scheme Procedure]

Constructs a new compound condition that includes the following simple conditions:

- If *whom* is not `#f`, a `&who` condition with the *whom* as its field

- A `&message` condition with the specified *message*
- A `&syntax` condition with the specified *form* and optional *subform* fields

7.6.2.25 rnrs hashtables

The `(rnrs hashtables (6))` library provides structures and procedures for creating and accessing hash tables. The hash tables API defined by R6RS is substantially similar to both Guile's native hash tables implementation as well as the one provided by SRFI-69; See Section 6.6.22 [Hash Tables], page 238, and Section 7.5.39 [SRFI-69], page 650, respectively. Note that you can write portable R6RS library code that manipulates SRFI-69 hash tables (by importing the `(srfi :69)` library); however, hash tables created by one API cannot be used by another.

Like SRFI-69 hash tables—and unlike Guile's native ones—R6RS hash tables associate hash and equality functions with a hash table at the time of its creation. Additionally, R6RS allows for the creation (via `hashtable-copy`; see below) of immutable hash tables.

`make-eq-hashtable` [Scheme Procedure]
`make-eq-hashtable` *k* [Scheme Procedure]
Returns a new hash table that uses `eq?` to compare keys and Guile's `hashq` procedure as a hash function. If *k* is given, it specifies the initial capacity of the hash table.

`make-eqv-hashtable` [Scheme Procedure]
`make-eqv-hashtable` *k* [Scheme Procedure]
Returns a new hash table that uses `eqv?` to compare keys and Guile's `hashv` procedure as a hash function. If *k* is given, it specifies the initial capacity of the hash table.

`make-hashtable` *hash-function equiv* [Scheme Procedure]
`make-hashtable` *hash-function equiv k* [Scheme Procedure]
Returns a new hash table that uses *equiv* to compare keys and *hash-function* as a hash function. *equiv* must be a procedure that accepts two arguments and returns a true value if they are equivalent, `#f` otherwise; *hash-function* must be a procedure that accepts one argument and returns a non-negative integer.

If *k* is given, it specifies the initial capacity of the hash table.

`hashtable?` *obj* [Scheme Procedure]
Returns `#t` if *obj* is an R6RS hash table, `#f` otherwise.

`hashtable-size` *hashtable* [Scheme Procedure]
Returns the number of keys currently in the hash table *hashtable*.

`hashtable-ref` *hashtable key default* [Scheme Procedure]
Returns the value associated with *key* in the hash table *hashtable*, or *default* if none is found.

`hashtable-set!` *hashtable key obj* [Scheme Procedure]
Associates the key *key* with the value *obj* in the hash table *hashtable*, and returns an unspecified value. An `&assertion` condition is raised if *hashtable* is immutable.

`hashtable-delete!` *hashtable key* [Scheme Procedure]
Removes any association found for the key *key* in the hash table *hashtable*, and returns an unspecified value. An `&assertion` condition is raised if *hashtable* is immutable.

`hashtable-contains?` *hashtable key* [Scheme Procedure]
Returns `#t` if the hash table *hashtable* contains an association for the key *key*, `#f` otherwise.

`hashtable-update!` *hashtable key proc default* [Scheme Procedure]
Associates with *key* in the hash table *hashtable* the result of calling *proc*, which must be a procedure that takes one argument, on the value currently associated *key* in *hashtable*—or on *default* if no such association exists. An `&assertion` condition is raised if *hashtable* is immutable.

`hashtable-copy` *hashtable* [Scheme Procedure]
`hashtable-copy` *hashtable mutable* [Scheme Procedure]
Returns a copy of the hash table *hashtable*. If the optional argument *mutable* is provided and is a true value, the new hash table will be mutable.

`hashtable-clear!` *hashtable* [Scheme Procedure]
`hashtable-clear!` *hashtable k* [Scheme Procedure]
Removes all of the associations from the hash table *hashtable*. The optional argument *k*, which specifies a new capacity for the hash table, is accepted by Guile's `(rnrs hashtables)` implementation, but is ignored.

`hashtable-keys` *hashtable* [Scheme Procedure]
Returns a vector of the keys with associations in the hash table *hashtable*, in an unspecified order.

`hashtable-entries` *hashtable* [Scheme Procedure]
Return two values—a vector of the keys with associations in the hash table *hashtable*, and a vector of the values to which these keys are mapped, in corresponding but unspecified order.

`hashtable-equivalence-function` *hashtable* [Scheme Procedure]
Returns the equivalence predicated use by *hashtable*. This procedure returns `eq?` and `eqv?`, respectively, for hash tables created by `make-eq-hashtable` and `make-eqv-hashtable`.

`hashtable-hash-function` *hashtable* [Scheme Procedure]
Returns the hash function used by *hashtable*. For hash tables created by `make-eq-hashtable` or `make-eqv-hashtable`, `#f` is returned.

`hashtable-mutable?` *hashtable* [Scheme Procedure]
Returns `#t` if *hashtable* is mutable, `#f` otherwise.

A number of hash functions are provided for convenience:

`equal-hash` *obj* [Scheme Procedure]
Returns an integer hash value for *obj*, based on its structure and current contents. This hash function is suitable for use with `equal?` as an equivalence function.

`string-hash` *string* [Scheme Procedure]
`symbol-hash` *symbol* [Scheme Procedure]
These procedures are identical to the ones provided by Guile's core library. See Section 6.6.22.2 [Hash Table Reference], page 239, for documentation.

`string-ci-hash` *string* [Scheme Procedure]
Returns an integer hash value for *string* based on its contents, ignoring case. This hash function is suitable for use with `string-ci=?` as an equivalence function.

7.6.2.26 rnrs enums

The `(rnrs enums (6))` library provides structures and procedures for working with enumerable sets of symbols. Guile's implementation defines an *enum-set* record type that encapsulates a finite set of distinct symbols, the *universe*, and a subset of these symbols, which define the enumeration set.

The SRFI-1 list library provides a number of procedures for performing set operations on lists; Guile's `(rnrs enums)` implementation makes use of several of them. See Section 7.5.3.10 [SRFI-1 Set Operations], page 595, for more information.

`make-enumeration` *symbol-list* [Scheme Procedure]
Returns a new enum-set whose universe and enumeration set are both equal to *symbol-list*, a list of symbols.

`enum-set-universe` *enum-set* [Scheme Procedure]
Returns an enum-set representing the universe of *enum-set*, an enum-set.

`enum-set-indexer` *enum-set* [Scheme Procedure]
Returns a procedure that takes a single argument and returns the zero-indexed position of that argument in the universe of *enum-set*, or `#f` if its argument is not a member of that universe.

`enum-set-constructor` *enum-set* [Scheme Procedure]
Returns a procedure that takes a single argument, a list of symbols from the universe of *enum-set*, an enum-set, and returns a new enum-set with the same universe that represents a subset containing the specified symbols.

`enum-set->list` *enum-set* [Scheme Procedure]
Returns a list containing the symbols of the set represented by *enum-set*, an enum-set, in the order that they appear in the universe of *enum-set*.

`enum-set-member?` *symbol enum-set* [Scheme Procedure]
`enum-set-subset?` *enum-set1 enum-set2* [Scheme Procedure]
`enum-set=?` *enum-set1 enum-set2* [Scheme Procedure]
These procedures test for membership of symbols and enum-sets in other enum-sets. `enum-set-member?` returns `#t` if and only if *symbol* is a member of the subset specified by *enum-set*. `enum-set-subset?` returns `#t` if and only if the universe of *enum-set1* is a subset of the universe of *enum-set2* and every symbol in *enum-set1* is present in *enum-set2*. `enum-set=?` returns `#t` if and only if *enum-set1* is a subset, as per `enum-set-subset?` of *enum-set2* and vice versa.

`enum-set-union` *enum-set1 enum-set2* [Scheme Procedure]
`enum-set-intersection` *enum-set1 enum-set2* [Scheme Procedure]
`enum-set-difference` *enum-set1 enum-set2* [Scheme Procedure]
These procedures return, respectively, the union, intersection, and difference of their enum-set arguments.

`enum-set-complement` *enum-set* [Scheme Procedure]
Returns *enum-set*'s complement (an enum-set), with regard to its universe.

`enum-set-projection` *enum-set1 enum-set2* [Scheme Procedure]
Returns the projection of the enum-set *enum-set1* onto the universe of the enum-set *enum-set2*.

`define-enumeration` *type-name* (*symbol ...*) [Scheme Syntax]
constructor-syntax
Evaluates to two new definitions: A constructor bound to *constructor-syntax* that behaves similarly to constructors created by `enum-set-constructor`, above, and creates new *enum-set*s in the universe specified by `(symbol ...)`; and a "predicate macro" bound to *type-name*, which has the following form:

```
(type-name sym)
```

If *sym* is a member of the universe specified by the *symbol*s above, this form evaluates to *sym*. Otherwise, a `&syntax` condition is raised.

7.6.2.27 rnrs

The `(rnrs (6))` library is a composite of all of the other R6RS standard libraries—it imports and re-exports all of their exported procedures and syntactic forms—with the exception of the following libraries:

- `(rnrs eval (6))`
- `(rnrs mutable-pairs (6))`
- `(rnrs mutable-strings (6))`
- `(rnrs r5rs (6))`

7.6.2.28 rnrs eval

The `(rnrs eval (6)` library provides procedures for performing "on-the-fly" evaluation of expressions.

`eval` *expression environment* [Scheme Procedure]
Evaluates *expression*, which must be a datum representation of a valid Scheme expression, in the environment specified by *environment*. This procedure is identical to the one provided by Guile's code library; See Section 6.18.4 [Fly Evaluation], page 387, for documentation.

`environment` *import-spec ...* [Scheme Procedure]
Constructs and returns a new environment based on the specified *import-spec*s, which must be datum representations of the import specifications used with the `import` form. See Section 6.20.6 [R6RS Libraries], page 417, for documentation.

7.6.2.29 rnrs mutable-pairs

The `(rnrs mutable-pairs (6))` library provides the `set-car!` and `set-cdr!` procedures, which allow the `car` and `cdr` fields of a pair to be modified.

These procedures are identical to the ones provide by Guile's core library. See Section 6.6.8 [Pairs], page 178, for documentation. All pairs in Guile are mutable; consequently, these procedures will never throw the `&assertion` condition described in the R6RS libraries specification.

7.6.2.30 rnrs mutable-strings

The `(rnrs mutable-strings (6))` library provides the `string-set!` and `string-fill!` procedures, which allow the content of strings to be modified "in-place."

These procedures are identical to the ones provided by Guile's core library. See Section 6.6.5.6 [String Modification], page 147, for documentation. All strings in Guile are mutable; consequently, these procedures will never throw the `&assertion` condition described in the R6RS libraries specification.

7.6.2.31 rnrs r5rs

The `(rnrs r5rs (6))` library exports bindings for some procedures present in R5RS but omitted from the R6RS base library specification.

`exact->inexact` *z* [Scheme Procedure]
`inexact->exact` *z* [Scheme Procedure]

These procedures are identical to the ones provided by Guile's core library. See Section 6.6.2.5 [Exactness], page 113, for documentation.

`quotient` *n1 n2* [Scheme Procedure]
`remainder` *n1 n2* [Scheme Procedure]
`modulo` *n1 n2* [Scheme Procedure]

These procedures are identical to the ones provided by Guile's core library. See Section 6.6.2.7 [Integer Operations], page 116, for documentation.

`delay` *expr* [Scheme Syntax]
`force` *promise* [Scheme Procedure]

The `delay` form and the `force` procedure are identical to their counterparts in Guile's core library. See Section 6.18.9 [Delayed Evaluation], page 396, for documentation.

`null-environment` *n* [Scheme Procedure]
`scheme-report-environment` *n* [Scheme Procedure]

These procedures are identical to the ones provided by the `(ice-9 r5rs)` Guile module. See Section 6.20.12 [Environments], page 427, for documentation.

7.7 R7RS Support

The R7RS standard is essentially R5RS (directly supported by Guile), plus a module facility, plus an organization of bindings into a standard set of modules.

Happily, the syntax for R7RS modules was chosen to be compatible with R6RS, and so Guile's documentation there applies. See Section 6.20.6 [R6RS Libraries], page 417, for more information on how to define R6RS libraries, and their integration with Guile modules. See Section 7.6.2.1 [Library Usage], page 663, also.

7.7.1 Incompatibilities with the R7RS

As the R7RS is a much less ambitious standard than the R6RS (see Section 1.1 [Guile and Scheme], page 3), it is very easy for Guile to support. As such, Guile is a fully conforming implementation of R7RS, with the exception of the occasional bug and a couple of unimplemented features:

- The R7RS specifies a syntax for reading circular data structures using *datum labels*, such as `#0=(1 2 3 . #0#)`. Guile's reader does not support this syntax currently; `https://bugs.gnu.org/38236`.
- As with R6RS, a number of lexical features of R7RS conflict with Guile's historical syntax. In addition to `r6rs-hex-escapes` and `hungry-eol-escapes` (see Section 7.6.1 [R6RS Incompatibilities], page 662), the `r7rs-symbols` reader feature needs to be explicitly enabled.

Guile exposes a procedure in the root module to choose R7RS defaults over Guile's historical defaults.

`install-r7rs!` [Scheme Procedure]

Alter Guile's default settings to better conform to the R7RS.

While Guile's defaults may evolve over time, the current changes that this procedure imposes are to add `.sls` and `.guile.sls` to the set of supported `%load-extensions`, to better support R7RS conventions. See Section 6.18.7 [Load Paths], page 393. `install-r7rs!` will also enable the reader options mentioned above.

Finally, note that the `--r7rs` command-line argument will call `install-r7rs!` before calling user code. R7RS users probably want to pass this argument to their Guile.

7.7.2 R7RS Standard Libraries

The R7RS organizes the definitions from R5RS into modules, and also adds a few new definitions.

We do not attempt to document these libraries fully here, as unlike R6RS, there are few new definitions in R7RS relative to R5RS. Most of their functionality is already in Guile's standard environment. Again, the expectation is that most Guile users will use the well-known and well-documented Guile modules; these R7RS libraries are mostly useful to users who want to port their code to other R7RS systems.

As a brief overview, we note that the libraries defined by the R7RS are as follows:

`(scheme base)`

The core functions, mostly corresponding to R5RS minus the elements listed separately below, but plus SRFI-34 error handling (see Section 7.5.23 [SRFI-34], page 625), bytevectors and bytevector ports (see Section 6.6.12 [Bytevectors], page 193), and some miscellaneous other new procedures.

`(scheme case-lambda)`

`case-lambda`.

`(scheme char)`

Converting strings and characters to upper or lower case, predicates for if a characer is numeric, and so on.

`(scheme complex)`

Constructors and accessors for complex numbers.

`(scheme cxr)`

`cddr`, `cadadr`, and all that.

`(scheme eval)`
: `eval`, but also an `environment` routine allowing a user to specify an environment using a module import set.

`(scheme file)`
: `call-with-input-file` and so on.

`(scheme inexact)`
: Routines that operate on inexact numbers: `sin`, `finite?`, and so on.

`(scheme lazy)`
: Promises.

`(scheme load)`
: The `load` procedure.

`(scheme process-context)`
: Environment variables. See Section 7.5.43 [SRFI-98], page 654. Also, `commmand-line`, `emergency-exit` (like Guile's `primitive-_exit`), and `exit`.

`(scheme r5rs)`
: The precise set of bindings exported by `r5rs`, but without `transcript-off` / `transcript-on`, and also with the auxiliary syntax definitions like `_` or `else`. See Section 6.10.2 [Syntax Rules], page 263, for more on auxiliary syntax.

`(scheme read)`
: The `read` procedure.

`(scheme repl)`
: The `interaction-environment` procedure.

`(scheme time)`
: `current-second`, as well as `current-jiffy` and `jiffies-per-second`. Guile uses the term "internal time unit" for what R7RS calls "jiffies".

`(scheme write)`
: `display`, `write`, as well as `write-shared` and `write-simple`.

For complete documentation, we advise the interested user to consult the R7RS directly (see Section "R7RS" in *The Revised^7 Report on the Algorithmic Language Scheme*).

7.8 Pattern Matching

The `(ice-9 match)` module provides a *pattern matcher*, written by Alex Shinn, and compatible with Andrew K. Wright's pattern matcher found in many Scheme implementations.

A pattern matcher can match an object against several patterns and extract the elements that make it up. Patterns can represent any Scheme object: lists, strings, symbols, records, etc. They can optionally contain *pattern variables*. When a matching pattern is found, an expression associated with the pattern is evaluated, optionally with all pattern variables bound to the corresponding elements of the object:

```
(let ((l '(hello (world))))
  (match l           ;; <- the input object
    (('hello (who))  ;; <- the pattern
```

```
      who)))          ;; <- the expression evaluated upon matching
⇒ world
```

In this example, list *l* matches the pattern `('hello (who))`, because it is a two-element list whose first element is the symbol `hello` and whose second element is a one-element list. Here *who* is a pattern variable. `match`, the pattern matcher, locally binds *who* to the value contained in this one-element list—i.e., the symbol `world`. An error would be raised if *l* did not match the pattern.

The same object can be matched against a simpler pattern:

```
(let ((l '(hello (world))))
  (match l
    ((x y)
     (values x y))))
⇒ hello
⇒ (world)
```

Here pattern `(x y)` matches any two-element list, regardless of the types of these elements. Pattern variables *x* and *y* are bound to, respectively, the first and second element of *l*.

Patterns can be composed, and nested. For instance, `...` (ellipsis) means that the previous pattern may be matched zero or more times in a list:

```
(match lst
  (((heads tails ...) ...)
   heads))
```

This expression returns the first element of each list within *lst*. For proper lists of proper lists, it is equivalent to `(map car lst)`. However, it performs additional checks to make sure that *lst* and the lists therein are proper lists, as prescribed by the pattern, raising an error if they are not.

Compared to hand-written code, pattern matching noticeably improves clarity and conciseness—no need to resort to series of `car` and `cdr` calls when matching lists, for instance. It also improves robustness, by making sure the input *completely* matches the pattern—conversely, hand-written code often trades robustness for conciseness. And of course, `match` is a macro, and the code it expands to is just as efficient as equivalent hand-written code.

The pattern matcher is defined as follows:

`match` *exp clause1 clause2 ...* [Scheme Syntax]

Match object *exp* against the patterns in *clause1 clause2 ...* in the order in which they appear. Return the value produced by the first matching clause. If no clause matches, throw an exception with key `match-error`.

Each clause has the form `(pattern body1 body2 ...)`. Each *pattern* must follow the syntax described below. Each body is an arbitrary Scheme expression, possibly referring to pattern variables of *pattern*.

The syntax and interpretation of patterns is as follows:

`patterns:`	`matches:`

```
pat ::= identifier                      anything, and binds identifier
      | _                               anything
      | ()                              the empty list
      | #t                              #t
      | #f                              #f
      | string                          a string
      | number                          a number
      | character                       a character
      | 'sexp                           an s-expression
      | 'symbol                         a symbol (special case of s-expr)
      | (pat_1 ... pat_n)               list of n elements
      | (pat_1 ... pat_n . pat_{n+1})   list of n or more
      | (pat_1 ... pat_n pat_n+1 ooo)   list of n or more, each element
                                          of remainder must match pat_n+1
      | #(pat_1 ... pat_n)              vector of n elements
      | #(pat_1 ... pat_n pat_n+1 ooo)  vector of n or more, each element
                                          of remainder must match pat_n+1
      | #&pat                           box
      | ($ record-name pat_1 ... pat_n) a record
      | (= field pat)                   a ``field'' of an object
      | (and pat_1 ... pat_n)           if all of pat_1 thru pat_n match
      | (or pat_1 ... pat_n)            if any of pat_1 thru pat_n match
      | (not pat_1 ... pat_n)           if all pat_1 thru pat_n don't match
      | (? predicate pat_1 ... pat_n)   if predicate true and all of
                                          pat_1 thru pat_n match
      | (set! identifier)               anything, and binds setter
      | (get! identifier)               anything, and binds getter
      | `qp                             a quasi-pattern
      | (identifier *** pat)            matches pat in a tree and binds
                                        identifier to the path leading
                                        to the object that matches pat

ooo ::= ...                             zero or more
      | ___                             zero or more
      | ..1                             1 or more

        quasi-patterns:                 matches:

qp  ::= ()                              the empty list
      | #t                              #t
      | #f                              #f
      | string                          a string
      | number                          a number
      | character                       a character
      | identifier                      a symbol
      | (qp_1 ... qp_n)                 list of n elements
      | (qp_1 ... qp_n . qp_{n+1})      list of n or more
```

```
| (qp_1 ... qp_n qp_n+1 ooo)      list of n or more, each element
                                    of remainder must match qp_n+1
| #(qp_1 ... qp_n)                vector of n elements
| #(qp_1 ... qp_n qp_n+1 ooo)     vector of n or more, each element
                                    of remainder must match qp_n+1
| #&qp                            box
| ,pat                            a pattern
| ,@pat                           a pattern
```

The names `quote`, `quasiquote`, `unquote`, `unquote-splicing`, `?`, `_`, `$`, `and`, `or`, `not`, `set!`, `get!`, `...`, and `___` cannot be used as pattern variables.

Here is a more complex example:

```
(use-modules (srfi srfi-9))

(let ()
  (define-record-type person
    (make-person name friends)
    person?
    (name    person-name)
    (friends person-friends))

  (letrec ((alice (make-person "Alice" (delay (list bob))))
           (bob   (make-person "Bob" (delay (list alice)))))
    (match alice
      (($ person name (= force (($ person "Bob"))))
       (list 'friend-of-bob name))
      (_ #f))))

⇒ (friend-of-bob "Alice")
```

Here the `$` pattern is used to match a SRFI-9 record of type *person* containing two or more slots. The value of the first slot is bound to *name*. The `=` pattern is used to apply `force` on the second slot, and then checking that the result matches the given pattern. In other words, the complete pattern matches any *person* whose second slot is a promise that evaluates to a one-element list containing a *person* whose first slot is `"Bob"`.

The `(ice-9 match)` module also provides the following convenient syntactic sugar macros wrapping around `match`.

`match-lambda` *clause1 clause2 ...* [Scheme Syntax]

Create a procedure of one argument that matches its argument against each clause, and returns the result of evaluating the corresponding expressions.

```
(match-lambda clause1 clause2 ...)
≡
(lambda (arg) (match arg clause1 clause2 ...))
```

```
((match-lambda
   (('hello (who))
    who))
```

```
 '(hello (world)))
⇒ world
```

`match-lambda*` *clause1 clause2 ...* [Scheme Syntax]

Create a procedure of any number of arguments that matches its argument list against each clause, and returns the result of evaluating the corresponding expressions.

```
(match-lambda* clause1 clause2 ...)
≡
(lambda args (match args clause1 clause2 ...))
```

```
((match-lambda*
   (('hello (who))
    who))
 'hello '(world))
⇒ world
```

`match-let` ((*pattern expression*) ...) *body* [Scheme Syntax]

Match each pattern to the corresponding expression, and evaluate the body with all matched variables in scope. Raise an error if any of the expressions fail to match. `match-let` is analogous to named let and can also be used for recursive functions which match on their arguments as in `match-lambda*`.

```
(match-let (((x y) (list 1 2))
            ((a b) (list 3 4)))
  (list a b x y))
⇒
(3 4 1 2)
```

`match-let` *variable* ((*pattern init*) ...) *body* [Scheme Syntax]

Similar to `match-let`, but analogously to *named let*, locally bind VARIABLE to a new procedure which accepts as many arguments as there are INIT expressions. The procedure is initially applied to the results of evaluating the INIT expressions. When called, the procedure matches each argument against the corresponding PATTERN, and returns the result(s) of evaluating the BODY expressions. See Section 6.13.4 [while do], page 301, for more on *named let*.

`match-let*` ((*variable expression*) ...) *body* [Scheme Syntax]

Similar to `match-let`, but analogously to `let*`, match and bind the variables in sequence, with preceding match variables in scope.

```
(match-let* (((x y) (list 1 2))
             ((a b) (list x 4)))
  (list a b x y))
≡
(match-let (((x y) (list 1 2)))
  (match-let (((a b) (list x 4)))
    (list a b x y)))
⇒
(1 4 1 2)
```

`match-letrec` ((*variable expression*) ...) *body* [Scheme Syntax]
Similar to `match-let`, but analogously to `letrec`, match and bind the variables with all match variables in scope.

Guile also comes with a pattern matcher specifically tailored to SXML trees, See Section 7.17 [sxml-match], page 738.

7.9 Readline Support

Guile comes with an interface module to the readline library (see *GNU Readline Library*). This makes interactive use much more convenient, because of the command-line editing features of readline. Using `(ice-9 readline)`, you can navigate through the current input line with the cursor keys, retrieve older command lines from the input history and even search through the history entries.

7.9.1 Loading Readline Support

The module is not loaded by default and so has to be loaded and activated explicitly. This is done with two simple lines of code:

```
(use-modules (ice-9 readline))
(activate-readline)
```

The first line will load the necessary code, and the second will activate readline's features for the REPL. If you plan to use this module often, you should save these to lines to your `.guile` personal startup file.

You will notice that the REPL's behaviour changes a bit when you have loaded the readline module. For example, when you press Enter before typing in the closing parentheses of a list, you will see the *continuation* prompt, three dots: `...` This gives you a nice visual feedback when trying to match parentheses. To make this even easier, *bouncing parentheses* are implemented. That means that when you type in a closing parentheses, the cursor will jump to the corresponding opening parenthesis for a short time, making it trivial to make them match.

Once the readline module is activated, all lines entered interactively will be stored in a history and can be recalled later using the cursor-up and -down keys. Readline also understands the Emacs keys for navigating through the command line and history.

When you quit your Guile session by evaluating `(quit)` or pressing Ctrl-D, the history will be saved to the file `.guile_history` and read in when you start Guile for the next time. Thus you can start a new Guile session and still have the (probably long-winded) definition expressions available.

You can specify a different history file by setting the environment variable `GUILE_HISTORY`. And you can make Guile specific customizations to your `.inputrc` by testing for application '`Guile`' (see Section "Conditional Init Constructs" in *GNU Readline Library*). For instance to define a key inserting a matched pair of parentheses,

```
$if Guile
  "\C-o": "()\C-b"
$endif
```

7.9.2 Readline Options

The readline interface module can be tweaked in a few ways to better suit the user's needs. Configuration is done via the readline module's options interface, in a similar way to the evaluator and debugging options (see Section 6.23.3 [Runtime Options], page 460).

`readline-options` [Scheme Procedure]
`readline-enable` *option-name* [Scheme Procedure]
`readline-disable` *option-name* [Scheme Procedure]
`readline-set!` *option-name value* [Scheme Syntax]

Accessors for the readline options. Note that unlike the enable/disable procedures, `readline-set!` is syntax, which expects an unquoted option name.

Here is the list of readline options generated by typing `(readline-options 'help)` in Guile. You can also see the default values.

```
history-file    yes     Use history file.
history-length  200     History length.
bounce-parens   500     Time (ms) to show matching opening parenthesis
                        (0 = off).
bracketed-paste yes     Disable interpretation of control characters
                        in pastes.
```

The readline options interface can only be used *after* loading the readline module, because it is defined in that module.

7.9.3 Readline Functions

The following functions are provided by

```
(use-modules (ice-9 readline))
```

There are two ways to use readline from Scheme code, either make calls to `readline` directly to get line by line input, or use the readline port below with all the usual reading functions.

`readline` [*prompt*] [Function]

Read a line of input from the user and return it as a string (without a newline at the end). *prompt* is the prompt to show, or the default is the string set in `set-readline-prompt!` below.

```
(readline "Type something: ") ⇒ "hello"
```

`set-readline-input-port!` *port* [Function]
`set-readline-output-port!` *port* [Function]

Set the input and output port the readline function should read from and write to. *port* must be a file port (see Section 6.14.10.1 [File Ports], page 344), and should usually be a terminal.

The default is the `current-input-port` and `current-output-port` (see Section 6.14.9 [Default Ports], page 343) when `(ice-9 readline)` loads, which in an interactive user session means the Unix "standard input" and "standard output".

7.9.3.1 Readline Port

`readline-port` [Function]

Return a buffered input port (see Section 7.15 [Buffered Input], page 735) which calls the `readline` function above to get input. This port can be used with all the usual reading functions (`read`, `read-char`, etc), and the user gets the interactive editing features of readline.

There's only a single readline port created. `readline-port` creates it when first called, and on subsequent calls just returns what it previously made.

`activate-readline` [Function]

If the `current-input-port` is a terminal (see Section 7.2.9 [`isatty?`], page 528) then enable readline for all reading from `current-input-port` (see Section 6.14.9 [Default Ports], page 343) and enable readline features in the interactive REPL (see Section 3.3.3 [The REPL], page 25).

```
(activate-readline)
(read-char)
```

`activate-readline` enables readline on `current-input-port` simply by a `set-current-input-port` to the `readline-port` above. An application can do that directly if the extra REPL features that `activate-readline` adds are not wanted.

`set-readline-prompt!` *prompt1* [*prompt2*] [Function]

Set the prompt string to print when reading input. This is used when reading through `readline-port`, and is also the default prompt for the `readline` function above.

prompt1 is the initial prompt shown. If a user might enter an expression across multiple lines, then *prompt2* is a different prompt to show further input required. In the Guile REPL for instance this is an ellipsis ('`...`').

See `set-buffered-input-continuation?!` (see Section 7.15 [Buffered Input], page 735) for an application to indicate the boundaries of logical expressions (assuming of course an application has such a notion).

7.9.3.2 Completion

`with-readline-completion-function` *completer thunk* [Function]

Call (*`thunk`*) with *completer* as the readline tab completion function to be used in any readline calls within that *thunk*. *completer* can be `#f` for no completion.

completer will be called as (*`completer`* `text state`), as described in (see Section "How Completing Works" in *GNU Readline Library*). *text* is a partial word to be completed, and each *completer* call should return a possible completion string or `#f` when no more. *state* is `#f` for the first call asking about a new *text* then `#t` while getting further completions of that *text*.

Here's an example *completer* for user login names from the password file (see Section 7.2.4 [User Information], page 511), much like readline's own `rl_username_completion_function`,

```
(define (username-completer-function text state)
  (if (not state)
```

```
      (setpwent))  ;; new, go to start of database
  (let more ((pw (getpwent)))
    (if pw
        (if (string-prefix? text (passwd:name pw))
            (passwd:name pw)     ;; this name matches, return it
            (more (getpwent)))   ;; doesn't match, look at next
        (begin
          ;; end of database, close it and return #f
          (endpwent)
          #f))))
```

`apropos-completion-function` *text state* [Function]
A completion function offering completions for Guile functions and variables (all `define`s). This is the default completion function.

`filename-completion-function` *text state* [Function]
A completion function offering filename completions. This is readline's `rl_filename_completion_function` (see Section "Completion Functions" in *GNU Readline Library*).

`make-completion-function` *string-list* [Function]
Return a completion function which offers completions from the possibilities in *string-list*. Matching is case-sensitive.

7.10 Pretty Printing

The module `(ice-9 pretty-print)` provides the procedure `pretty-print`, which provides nicely formatted output of Scheme objects. This is especially useful for deeply nested or complex data structures, such as lists and vectors.

The module is loaded by entering the following:

```
(use-modules (ice-9 pretty-print))
```

This makes the procedure `pretty-print` available. As an example how `pretty-print` will format the output, see the following:

```
(pretty-print '(define (foo) (lambda (x)
(cond ((zero? x) #t) ((negative? x) -x) (else
(if (= x 1) 2 (* x x x)))))))
⊣
(define (foo)
  (lambda (x)
    (cond ((zero? x) #t)
          ((negative? x) -x)
          (else (if (= x 1) 2 (* x x x))))))
```

`pretty-print` *obj* [*port*] [*keyword-options*] [Scheme Procedure]
Print the textual representation of the Scheme object *obj* to *port*. *port* defaults to the current output port, if not given.

The further *keyword-options* are keywords and parameters as follows,

`#:display?` *flag*
: If *flag* is true then print using `display`. The default is `#f` which means use `write` style. See Section 6.18.3 [Scheme Write], page 386.

`#:per-line-prefix` *string*
: Print the given *string* as a prefix on each line. The default is no prefix.

`#:width` *columns*
: Print within the given *columns*. The default is 79.

`#:max-expr-width` *columns*
: The maximum width of an expression. The default is 50.

Also exported by the `(ice-9 pretty-print)` module is `truncated-print`, a procedure to print Scheme datums, truncating the output to a certain number of characters. This is useful when you need to present an arbitrary datum to the user, but you only have one line in which to do so.

```
(define exp '(a b #(c d e) f . g))
(truncated-print exp #:width 10) (newline)
⊣ (a b . #)
(truncated-print exp #:width 15) (newline)
⊣ (a b # f . g)
(truncated-print exp #:width 18) (newline)
⊣ (a b #(c ...) . #)
(truncated-print exp #:width 20) (newline)
⊣ (a b #(c d e) f . g)
(truncated-print "The quick brown fox" #:width 20) (newline)
⊣ "The quick brown..."
(truncated-print (current-module) #:width 20) (newline)
⊣ #<directory (gui...>
```

`truncated-print` will not output a trailing newline. If an expression does not fit in the given width, it will be truncated – possibly ellipsized[3], or in the worst case, displayed as `#`.

`truncated-print` *obj* [*port*] [*keyword-options*] [Scheme Procedure]
: Print *obj*, truncating the output, if necessary, to make it fit into *width* characters. By default, *obj* will be printed using `write`, though that behavior can be overridden via the *display?* keyword argument.

  The default behaviour is to print depth-first, meaning that the entire remaining width will be available to each sub-expression of *obj* – e.g., if *obj* is a vector, each member of *obj*. One can attempt to "ration" the available width, trying to allocate it equally to each sub-expression, via the *breadth-first?* keyword argument.

  The further *keyword-options* are keywords and parameters as follows,

  `#:display?` *flag*
  : If *flag* is true then print using `display`. The default is `#f` which means use `write` style. see Section 6.18.3 [Scheme Write], page 386.

[3] On Unicode-capable ports, the ellipsis is represented by character 'HORIZONTAL ELLIPSIS' (U+2026), otherwise it is represented by three dots.

`#:width` *columns*
: Print within the given *columns*. The default is 79.

`#:breadth-first?` *flag*
: If *flag* is true, then allocate the available width breadth-first among elements of a compound data structure (list, vector, pair, etc.). The default is `#f` which means that any element is allowed to consume all of the available width.

7.11 Formatted Output

The `format` function is a powerful way to print numbers, strings and other objects together with literal text under the control of a format string. This function is available from

```
(use-modules (ice-9 format))
```

A format string is generally more compact and easier than using just the standard procedures like `display`, `write` and `newline`. Parameters in the output string allow various output styles, and parameters can be taken from the arguments for runtime flexibility.

`format` is similar to the Common Lisp procedure of the same name, but it's not identical and doesn't have quite all the features found in Common Lisp.

C programmers will note the similarity between `format` and `printf`, though escape sequences are marked with `~` instead of `%`, and are more powerful.

`format` *dest fmt arg ...* [Scheme Procedure]
: Write output specified by the *fmt* string to *dest*. *dest* can be an output port, `#t` for `current-output-port` (see Section 6.14.9 [Default Ports], page 343), or `#f` to return the output as a string.

fmt can contain literal text to be output, and `~` escapes. Each escape has the form

```
~ [param [, param...] [:] [@] code
```

`code` is a character determining the escape sequence. The `:` and `@` characters are optional modifiers, one or both of which change the way various codes operate. Optional parameters are accepted by some codes too. Parameters have the following forms,

`[+/-]number`
: An integer, with optional `+` or `-`.

`'` (apostrophe)
: The following character in the format string, for instance `'z` for `z`.

`v`
: The next function argument as the parameter. `v` stands for "variable", a parameter can be calculated at runtime and included in the arguments. Upper case `V` can be used too.

`#`
: The number of arguments remaining. (See `~*` below for some usages.)

Parameters are separated by commas (`,`). A parameter can be left empty to keep its default value when supplying later parameters.

The following escapes are available. The code letters are not case-sensitive, upper and lower case are the same.

`~a`
`~s` Object output. Parameters: *minwidth*, *padinc*, *minpad*, *padchar*.

`~a` outputs an argument like `display`, `~s` outputs an argument like `write` (see Section 6.18.3 [Scheme Write], page 386).

```
(format #t "~a" "foo") ⊣ foo
(format #t "~s" "foo") ⊣ "foo"
```

`~:a` and `~:s` put objects that don't have an external representation in quotes like a string.

```
(format #t "~:a" car) ⊣ "#<primitive-procedure car>"
```

If the output is less than *minwidth* characters (default 0), it's padded on the right with *padchar* (default space). `~@a` and `~@s` put the padding on the left instead.

```
(format #f "~5a" 'abc)       ⇒ "abc  "
(format #f "~5,,,'-@a" 'abc) ⇒ "--abc"
```

minpad is a minimum for the padding then plus a multiple of *padinc*. Ie. the padding is $minpad+N*padinc$, where n is the smallest integer making the total object plus padding greater than or equal to *minwidth*. The default *minpad* is 0 and the default *padinc* is 1 (imposing no minimum or multiple).

```
(format #f "~5,1,4a" 'abc) ⇒ "abc    "
```

`~c` Character. Parameter: *charnum*.

Output a character. The default is to simply output, as per `write-char` (see Section 6.14.11 [Venerable Port Interfaces], page 350). `~@c` prints in `write` style. `~:c` prints control characters (ASCII 0 to 31) in `^X` form.

```
(format #t "~c" #\z)        ⊣ z
(format #t "~@c" #\z)       ⊣ #\z
(format #t "~:c" #\newline) ⊣ ^J
```

If the *charnum* parameter is given then an argument is not taken but instead the character is `(integer->char charnum)` (see Section 6.6.3 [Characters], page 129). This can be used for instance to output characters given by their ASCII code.

```
(format #t "~65c")  ⊣ A
```

`~d`
`~x`
`~o`
`~b` Integer. Parameters: *minwidth*, *padchar*, *commachar*, *commawidth*.

Output an integer argument as a decimal, hexadecimal, octal or binary integer (respectively), in a locale-independent way.

```
(format #t "~d" 123) ⊣ 123
```

`~@d` etc shows a `+` sign is shown on positive numbers.

```
(format #t "~@b" 12) ⊣ +1100
```

If the output is less than the *minwidth* parameter (default no minimum), it's padded on the left with the *padchar* parameter (default space).

```
(format #t "~5,'*d" 12)    ⊣ ***12
(format #t "~5,'0d" 12)    ⊣ 00012
(format #t "~3d"    1234)  ⊣ 1234
```

`~:d` adds commas (or the *commachar* parameter) every three digits (or the *commawidth* parameter many). However, when your intent is to write numbers in a way that follows typographical conventions, using `~h` is recommended.

```
(format #t "~:d" 1234567)          ⊣ 1,234,567
(format #t "~10,'*,'/,2:d" 12345)  ⊣ ***1/23/45
```

Hexadecimal `~x` output is in lower case, but the `~(` and `~)` case conversion directives described below can be used to get upper case.

```
(format #t "~x"       65261)  ⊣ feed
(format #t "~:@(~x~)" 65261)  ⊣ FEED
```

`~r` Integer in words, roman numerals, or a specified radix. Parameters: *radix*, *minwidth*, *padchar*, *commachar*, *commawidth*.

With no parameters output is in words as a cardinal like "ten", or `~:r` prints an ordinal like "tenth".

```
(format #t "~r" 9)   ⊣ nine        ;; cardinal
(format #t "~r" -9)  ⊣ minus nine  ;; cardinal
(format #t "~:r" 9)  ⊣ ninth       ;; ordinal
```

And also with no parameters, `~@r` gives roman numerals and `~:@r` gives old roman numerals. In old roman numerals there's no "subtraction", so 9 is `VIIII` instead of `IX`. In both cases only positive numbers can be output.

```
(format #t "~@r" 89)   ⊣ LXXXIX     ;; roman
(format #t "~:@r" 89)  ⊣ LXXXVIIII  ;; old roman
```

When a parameter is given it means numeric output in the specified *radix*. The modifiers and parameters following the radix are the same as described for `~d` etc above.

```
(format #f "~3r" 27)   ⇒ "1000"    ;; base 3
(format #f "~3,5r" 26) ⇒ "  222"   ;; base 3 width 5
```

`~f` Fixed-point float. Parameters: *width*, *decimals*, *scale*, *overflowchar*, *padchar*.

Output a number or number string in fixed-point format, ie. with a decimal point.

```
(format #t "~f" 5)       ⊣ 5.0
(format #t "~f" "123")   ⊣ 123.0
(format #t "~f" "1e-1")  ⊣ 0.1
```

`~@f` prints a `+` sign on positive numbers (including zero).

```
(format #t "~@f" 0)  ⊣ +0.0
```

If the output is less than *width* characters it's padded on the left with *padchar* (space by default). If the output equals or exceeds *width* then there's no padding. The default for *width* is no padding.

```
(format #f "~6f" -1.5)      ⇒ "  -1.5"
(format #f "~6,,,,'*f" 23)  ⇒ "**23.0"
(format #f "~6f" 1234567.0) ⇒ "1234567.0"
```

decimals is how many digits to print after the decimal point, with the value rounded or padded with zeros as necessary. (The default is to output as many decimals as required.)

```
(format #t "~1,2f" 3.125) ⊣ 3.13
(format #t "~1,2f" 1.5)   ⊣ 1.50
```

scale is a power of 10 applied to the value, moving the decimal point that many places. A positive *scale* increases the value shown, a negative decreases it.

```
(format #t "~,,2f" 1234)  ⊣ 123400.0
(format #t "~,,-2f" 1234) ⊣ 12.34
```

If *overflowchar* and *width* are both given and if the output would exceed *width*, then that many *overflowchar*s are printed instead of the value.

```
(format #t "~6,,,'xf" 12345) ⊣ 12345.
(format #t "~5,,,'xf" 12345) ⊣ xxxxx
```

`~h` Localized number[4]. Parameters: *width*, *decimals*, *padchar*.

Like `~f`, output an exact or floating point number, but do so according to the current locale, or according to the given locale object when the `:` modifier is used (see Section 6.25.4 [Number Input and Output], page 468).

```
(format #t "~h" 12345.5678)  ; with "C" as the current locale
⊣ 12345.5678

(format #t "~14,,'*:h" 12345.5678
        (make-locale LC_ALL "en_US"))
⊣ ***12,345.5678

(format #t "~,2:h" 12345.5678
        (make-locale LC_NUMERIC "fr_FR"))
⊣ 12 345,56
```

`~e` Exponential float. Parameters: *width*, *mantdigits*, *expdigits*, *intdigits*, *overflowchar*, *padchar*, *expchar*.

Output a number or number string in exponential notation.

```
(format #t "~e" 5000.25) ⊣ 5.00025E+3
(format #t "~e" "123.4") ⊣ 1.234E+2
(format #t "~e" "1e4")   ⊣ 1.0E+4
```

[4] The `~h` format specifier first appeared in Guile version 2.0.6.

`~@e` prints a `+` sign on positive numbers (including zero). (This is for the mantissa, a `+` or `-` sign is always shown on the exponent.)

```
(format #t "~@e" 5000.0) ⊣ +5.0E+3
```

If the output is less than *width* characters it's padded on the left with *padchar* (space by default). The default for *width* is to output with no padding.

```
(format #f "~10e" 1234.0)     ⇒ "  1.234E+3"
(format #f "~10,,,,,'*e" 0.5) ⇒ "****5.0E-1"
```

mantdigits is the number of digits shown in the mantissa after the decimal point. The value is rounded or trailing zeros are added as necessary. The default *mantdigits* is to show as much as needed by the value.

```
(format #f "~,3e" 11111.0) ⇒ "1.111E+4"
(format #f "~,8e" 123.0)   ⇒ "1.23000000E+2"
```

expdigits is the minimum number of digits shown for the exponent, with leading zeros added if necessary. The default for *expdigits* is to show only as many digits as required. At least 1 digit is always shown.

```
(format #f "~,,1e" 1.0e99) ⇒ "1.0E+99"
(format #f "~,,6e" 1.0e99) ⇒ "1.0E+000099"
```

intdigits (default 1) is the number of digits to show before the decimal point in the mantissa. *intdigits* can be zero, in which case the integer part is a single `0`, or it can be negative, in which case leading zeros are shown after the decimal point.

```
(format #t "~,,,3e" 12345.0)  ⊣ 123.45E+2
(format #t "~,,,0e" 12345.0)  ⊣ 0.12345E+5
(format #t "~,,,-3e" 12345.0) ⊣ 0.00012345E+8
```

If *overflowchar* is given then *width* is a hard limit. If the output would exceed *width* then instead that many *overflowchar*s are printed.

```
(format #f "~6,,,,'xe" 100.0) ⇒ "1.0E+2"
(format #f "~3,,,,'xe" 100.0) ⇒ "xxx"
```

expchar is the exponent marker character (default `E`).

```
(format #t "~,,,,,,'ee" 100.0) ⊣ 1.0e+2
```

`~g` General float. Parameters: *width*, *mantdigits*, *expdigits*, *intdigits*, *overflowchar*, *padchar*, *expchar*.

Output a number or number string in either exponential format the same as `~e`, or fixed-point format like `~f` but aligned where the mantissa would have been and followed by padding where the exponent would have been.

Fixed-point is used when the absolute value is 0.1 or more and it takes no more space than the mantissa in exponential format, ie. basically up to *mantdigits* digits.

```
(format #f "~12,4,2g" 999.0)    ⇒ "   999.0    "
(format #f "~12,4,2g" "100000") ⇒ "  1.0000E+05"
```

The parameters are interpreted as per `~e` above. When fixed-point is used, the *decimals* parameter to `~f` is established from *mantdigits*, so as to give a total *mantdigits* + 1 figures.

`~$` Monetary style fixed-point float. Parameters: *decimals*, *intdigits*, *width*, *padchar*.

Output a number or number string in fixed-point format, ie. with a decimal point. *decimals* is the number of decimal places to show, default 2.

```
(format #t "~$" 5)       ⊣ 5.00
(format #t "~4$" "2.25") ⊣ 2.2500
(format #t "~4$" "1e-2") ⊣ 0.0100
```

`~@$` prints a `+` sign on positive numbers (including zero).

```
(format #t "~@$" 0) ⊣ +0.00
```

intdigits is a minimum number of digits to show in the integer part of the value (default 1).

```
(format #t "~,3$" 9.5)   ⊣ 009.50
(format #t "~,0$" 0.125) ⊣ .13
```

If the output is less than *width* characters (default 0), it's padded on the left with *padchar* (default space). `~:$` puts the padding after the sign.

```
(format #f "~,,8$" -1.5)   ⇒ "   -1.50"
(format #f "~,,8:$" -1.5)  ⇒ "-   1.50"
(format #f "~,,8,'.:@$" 3) ⇒ "+...3.00"
```

Note that floating point for dollar amounts is generally not a good idea, because a cent 0.01 cannot be represented exactly in the binary floating point Guile uses, which leads to slowly accumulating rounding errors. Keeping values as cents (or fractions of a cent) in integers then printing with the scale option in `~f` may be a better approach.

`~i` Complex fixed-point float. Parameters: *width*, *decimals*, *scale*, *overflowchar*, *padchar*.

Output the argument as a complex number, with both real and imaginary part shown (even if one or both are zero).

The parameters and modifiers are the same as for fixed-point `~f` described above. The real and imaginary parts are both output with the same given parameters and modifiers, except that for the imaginary part the `@` modifier is always enabled, so as to print a `+` sign between the real and imaginary parts.

```
(format #t "~i" 1)  ⊣ 1.0+0.0i
```

`~p` Plural. No parameters.

Output nothing if the argument is 1, or '`s`' for any other value.

```
(format #t "enter name~p" 1) ⊣ enter name
(format #t "enter name~p" 2) ⊣ enter names
```

`~@p` prints '`y`' for 1 or '`ies`' otherwise.

```
(format #t "pupp~@p" 1) ⊣ puppy
(format #t "pupp~@p" 2) ⊣ puppies
```

~:p re-uses the preceding argument instead of taking a new one, which can be convenient when printing some sort of count.

```
(format #t "~d cat~:p" 9)    ⊣ 9 cats
(format #t "~d pupp~:@p" 5) ⊣ 5 puppies
```

~p is designed for English plurals and there's no attempt to support other languages. ~[conditionals (below) may be able to help. When using gettext to translate messages ngettext is probably best though (see Section 6.25 [Internationalization], page 465).

~y Structured printing. Parameters: *width*.

~y outputs an argument using pretty-print (see Section 7.10 [Pretty Printing], page 714). The result will be formatted to fit within *width* columns (79 by default), consuming multiple lines if necessary.

~@y outputs an argument using truncated-print (see Section 7.10 [Pretty Printing], page 714). The resulting code will be formatted to fit within *width* columns (79 by default), on a single line. The output will be truncated if necessary.

~:@y is like ~@y, except the *width* parameter is interpreted to be the maximum column to which to output. That is to say, if you are at column 10, and ~60:@y is seen, the datum will be truncated to 50 columns.

~?

~k Sub-format. No parameters.

Take a format string argument and a second argument which is a list of arguments for that string, and output the result.

```
(format #t "~?" "~d ~d" '(1 2))     ⊣ 1 2
```

~@? takes arguments for the sub-format directly rather than in a list.

```
(format #t "~@? ~s" "~d ~d" 1 2 "foo") ⊣ 1 2 "foo"
```

~? and ~k are the same, ~k is provided for T-Scheme compatibility.

~* Argument jumping. Parameter: *N*.

Move forward *N* arguments (default 1) in the argument list. ~:* moves backwards. (*N* cannot be negative.)

```
(format #f "~d ~2*~d" 1 2 3 4) ⇒ "1 4"
(format #f "~d ~:*~d" 6)       ⇒ "6 6"
```

~@* moves to argument number *N*. The first argument is number 0 (and that's the default for *N*).

```
(format #f "~d~d again ~@*~d~d" 1 2) ⇒ "12 again 12"
(format #f "~d~d~d ~1@*~d~d" 1 2 3)  ⇒ "123 23"
```

A # move to the end followed by a : modifier move back can be used for an absolute position relative to the end of the argument list, a reverse of what the @ modifier does.

```
(format #t "~#*~2:*~a" 'a 'b 'c 'd)   ⊣ c
```

At the end of the format string the current argument position doesn't matter, any further arguments are ignored.

`~t` Advance to a column position. Parameters: *colnum*, *colinc*, *padchar*.

Output *padchar* (space by default) to move to the given *colnum* column. The start of the line is column 0, the default for *colnum* is 1.

```
(format #f "~tX")  ⇒ " X"
(format #f "~3tX") ⇒ "   X"
```

If the current column is already past *colnum*, then the move is to there plus a multiple of *colinc*, ie. column $colnum + N * colinc$ for the smallest N which makes that value greater than or equal to the current column. The default *colinc* is 1 (which means no further move).

```
(format #f "abcd~2,5,'.tx") ⇒ "abcd...x"
```

`~@t` takes *colnum* as an offset from the current column. *colnum* many pad characters are output, then further padding to make the current column a multiple of *colinc*, if it isn't already so.

```
(format #f "a~3,5'*@tx") ⇒ "a****x"
```

`~t` is implemented using `port-column` (see Section 6.14.4 [Textual I/O], page 336), so it works even there has been other output before `format`.

`~~` Tilde character. Parameter: *n*.

Output a tilde character `~`, or *n* many if a parameter is given. Normally `~` introduces an escape sequence, `~~` is the way to output a literal tilde.

`~%` Newline. Parameter: *n*.

Output a newline character, or *n* many if a parameter is given. A newline (or a few newlines) can of course be output just by including them in the format string.

`~&` Start a new line. Parameter: *n*.

Output a newline if not already at the start of a line. With a parameter, output that many newlines, but with the first only if not already at the start of a line. So for instance 3 would be a newline if not already at the start of a line, and 2 further newlines.

`~_` Space character. Parameter: *n*.

Output a space character, or *n* many if a parameter is given.

With a variable parameter this is one way to insert runtime calculated padding (`~t` or the various field widths can do similar things).

```
(format #f "~v_foo" 4) ⇒ "    foo"
```

`~/` Tab character. Parameter: *n*.

Output a tab character, or *n* many if a parameter is given.

`~|` Formfeed character. Parameter: *n*.

Output a formfeed character, or *n* many if a parameter is given.

`~!` Force output. No parameters.

At the end of output, call `force-output` to flush any buffers on the destination (see Section 6.14.6 [Buffering], page 339). `~!` can occur anywhere in the format string, but the force is done at the end of output.

When output is to a string (destination `#f`), `~!` does nothing.

`~newline` (ie. newline character)
: Continuation line. No parameters.

  Skip this newline and any following whitespace in the format string, ie. don't send it to the output. This can be used to break up a long format string for readability, but not print the extra whitespace.

```
(format #f "abc~
            ~d def~
            ~d" 1 2) ⇒ "abc1 def2"
```

  `~:newline` skips the newline but leaves any further whitespace to be printed normally.

  `~@newline` prints the newline then skips following whitespace.

`~( ~)`
: Case conversion. No parameters.

  Between `~(` and `~)` the case of all output is changed. The modifiers on `~(` control the conversion.

  `~(` — lower case.

  `~:@(` — upper case.

  For example,

```
(format #t "~(Hello~)")    ⊣ hello
(format #t "~:@(Hello~)") ⊣ HELLO
```

  In the future it's intended the modifiers `:` and `@` alone will capitalize the first letters of words, as per Common Lisp `format`, but the current implementation of this is flawed and not recommended for use.

  Case conversions do not nest, currently. This might change in the future, but if it does then it will be to Common Lisp style where the outermost conversion has priority, overriding inner ones (making those fairly pointless).

`~{ ~}`
: Iteration. Parameter: *maxreps* (for `~{`).

  The format between `~{` and `~}` is iterated. The modifiers to `~{` determine how arguments are taken. The default is a list argument with each iteration successively consuming elements from it. This is a convenient way to output a whole list.

```
(format #t "~{~d~}"     '(1 2 3))        ⊣ 123
(format #t "~{~s=~d ~}" '("x" 1 "y" 2)) ⊣ "x"=1 "y"=2
```

  `~:{` takes a single argument which is a list of lists, each of those contained lists gives the arguments for the iterated format.

```
(format #t "~:{~dx~d ~}" '((1 2) (3 4) (5 6)))
⊣ 1x2 3x4 5x6
```

  `~@{` takes arguments directly, with each iteration successively consuming arguments.

```
(format #t "~@{~d~}"     1 2 3)        ⊣ 123
(format #t "~@{~s=~d ~}" "x" 1 "y" 2) ⊣ "x"=1 "y"=2
```

`~:@{` takes list arguments, one argument for each iteration, using that list for the format.

```
(format #t "~:@{~dx~d ~}" '(1 2) '(3 4) '(5 6))
⊣ 1x2 3x4 5x6
```

Iterating stops when there are no more arguments or when the *maxreps* parameter to `~{` is reached (default no maximum).

```
(format #t "~2{~d~}" '(1 2 3 4)) ⊣ 12
```

If the format between `~{` and `~}` is empty, then a format string argument is taken (before iteration argument(s)) and used instead. This allows a sub-format (like `~?` above) to be iterated.

```
(format #t "~{~}" "~d" '(1 2 3)) ⊣ 123
```

Iterations can be nested, an inner iteration operates in the same way as described, but of course on the arguments the outer iteration provides it. This can be used to work into nested list structures. For example in the following the inner `~{~d~}x` is applied to `(1 2)` then `(3 4 5)` etc.

```
(format #t "~{~{~d~}x~}" '((1 2) (3 4 5))) ⊣ 12x345x
```

See also `~^` below for escaping from iteration.

`~[ ~; ~]` Conditional. Parameter: *selector*.

A conditional block is delimited by `~[` and `~]`, and `~;` separates clauses within the block. `~[` takes an integer argument and that number clause is used. The first clause is number 0.

```
(format #f "~[peach~;banana~;mango~]" 1)  ⇒ "banana"
```

The *selector* parameter can be used for the clause number, instead of taking an argument.

```
(format #f "~2[peach~;banana~;mango~]") ⇒ "mango"
```

If the clause number is out of range then nothing is output. Or the last clause can be `~:;` to use that for a number out of range.

```
(format #f "~[banana~;mango~]"         99) ⇒ ""
(format #f "~[banana~;mango~:;fruit~]" 99) ⇒ "fruit"
```

`~:[` treats the argument as a flag, and expects two clauses. The first is used if the argument is `#f` or the second otherwise.

```
(format #f "~:[false~;not false~]" #f)   ⇒ "false"
(format #f "~:[false~;not false~]" 'abc) ⇒ "not false"

(let ((n 3))
  (format #t "~d gnu~:[s are~; is~] here" n (= 1 n)))
⊣ 3 gnus are here
```

`~@[` also treats the argument as a flag, and expects one clause. If the argument is `#f` then no output is produced and the argument is consumed, otherwise the clause is used and the argument is not consumed, it's left for the clause. This can be used for instance to suppress output if `#f` means something not available.

```
(format #f "~@[temperature=~d~]" 27) ⇒ "temperature=27"
```

```
(format #f "~@[temperature=~d~]" #f) ⇒ ""
```

`~^` Escape. Parameters: *val1*, *val2*, *val3*.

Stop formatting if there are no more arguments. This can be used for instance to have a format string adapt to a variable number of arguments.

```
(format #t "~d~^ ~d" 1)    ⊣ 1
(format #t "~d~^ ~d" 1 2)  ⊣ 1 2
```

Within a `~{ ~}` iteration, `~^` stops the current iteration step if there are no more arguments to that step, but continuing with possible further steps and the rest of the format. This can be used for instance to avoid a separator on the last iteration, or to adapt to variable length argument lists.

```
(format #f "~{~d~^/~} go"    '(1 2 3))     ⇒ "1/2/3 go"
(format #f "~:{ ~d~^~d~} go" '((1) (2 3))) ⇒ " 1 23 go"
```

Within a `~?` sub-format, `~^` operates just on that sub-format. If it terminates the sub-format then the originating format will still continue.

```
(format #t "~? items" "~d~^ ~d" '(1))    ⊣ 1 items
(format #t "~? items" "~d~^ ~d" '(1 2)) ⊣ 1 2 items
```

The parameters to `~^` (which are numbers) change the condition used to terminate. For a single parameter, termination is when that value is zero (notice this makes plain `~^` equivalent to `~#^`). For two parameters, termination is when those two are equal. For three parameters, termination is when *val1* $\leq$ *val2* and *val2* $\leq$ *val3*.

`~q` Inquiry message. Insert a copyright message into the output.

`~:q` inserts the format implementation version.

It's an error if there are not enough arguments for the escapes in the format string, but any excess arguments are ignored.

Iterations `~{ ~}` and conditionals `~[ ~; ~]` can be nested, but must be properly nested, meaning the inner form must be entirely within the outer form. So it's not possible, for instance, to try to conditionalize the endpoint of an iteration.

```
(format #t "~{ ~[ ... ~] ~}" ...)       ;; good
(format #t "~{ ~[ ... ~} ... ~]" ...)   ;; bad
```

The same applies to case conversions `~( ~)`, they must properly nest with respect to iterations and conditionals (though currently a case conversion cannot nest within another case conversion).

When a sub-format (`~?`) is used, that sub-format string must be self-contained. It cannot for instance give a `~{` to begin an iteration form and have the `~}` up in the originating format, or similar.

Guile contains a `format` procedure even when the module `(ice-9 format)` is not loaded. The default `format` is `simple-format` (see Section 6.14.5 [Simple Output], page 338), it doesn't support all escape sequences documented in this section, and will signal an error

if you try to use one of them. The reason for two versions is that the full `format` is fairly large and requires some time to load. `simple-format` is often adequate too.

7.12 File Tree Walk

The functions in this section traverse a tree of files and directories. They come in two flavors: the first one is a high-level functional interface, and the second one is similar to the C `ftw` and `nftw` routines (see Section "Working with Directory Trees" in *GNU C Library Reference Manual*).

```
(use-modules (ice-9 ftw))
```

`file-system-tree` *file-name* [*enter?* [*stat*]] [Scheme Procedure]

Return a tree of the form `(file-name stat children ...)` where *stat* is the result of `(stat file-name)` and *children* are similar structures for each file contained in *file-name* when it designates a directory.

The optional *enter?* predicate is invoked as `(enter? name stat)` and should return true to allow recursion into directory *name*; the default value is a procedure that always returns `#t`. When a directory does not match *enter?*, it nonetheless appears in the resulting tree, only with zero children.

The *stat* argument is optional and defaults to `lstat`, as for `file-system-fold` (see below.)

The example below shows how to obtain a hierarchical listing of the files under the `module/language` directory in the Guile source tree, discarding their `stat` info:

```
(use-modules (ice-9 match))

(define remove-stat
  ;; Remove the 'stat' object the 'file-system-tree' provides
  ;; for each file in the tree.
  (match-lambda
    ((name stat)              ; flat file
     name)
    ((name stat children ...) ; directory
     (list name (map remove-stat children)))))

(let ((dir (string-append (assq-ref %guile-build-info 'top_srcdir)
                          "/module/language")))
  (remove-stat (file-system-tree dir)))

⇒
("language"
 (("value" ("spec.go" "spec.scm"))
  ("scheme"
   ("spec.go"
    "spec.scm"
    "compile-tree-il.scm"
    "decompile-tree-il.scm"
```

```
    "decompile-tree-il.go"
    "compile-tree-il.go"))
  ("tree-il"
   ("spec.go"
    "fix-letrec.go"
    "inline.go"
    "fix-letrec.scm"
    "compile-glil.go"
    "spec.scm"
    "optimize.scm"
    "primitives.scm"
    ...))
  ...))
```

It is often desirable to process directories entries directly, rather than building up a tree of entries in memory, like `file-system-tree` does. The following procedure, a *combinator*, is designed to allow directory entries to be processed directly as a directory tree is traversed; in fact, `file-system-tree` is implemented in terms of it.

`file-system-fold` *enter? leaf down up skip error init file-name* [*stat*] [Scheme Procedure]

Traverse the directory at *file-name*, recursively, and return the result of the successive applications of the *leaf*, *down*, *up*, and *skip* procedures as described below.

Enter sub-directories only when (*enter? path stat result*) returns true. When a sub-directory is entered, call (*down path stat result*), where *path* is the path of the sub-directory and *stat* the result of (`false-if-exception` (*stat path*)); when it is left, call (*up path stat result*).

For each file in a directory, call (*leaf path stat result*).

When *enter?* returns `#f`, or when an unreadable directory is encountered, call (*skip path stat result*).

When *file-name* names a flat file, (*leaf path stat init*) is returned.

When an `opendir` or *stat* call fails, call (*error path stat errno result*), with *errno* being the operating system error number that was raised—e.g., `EACCES`—and *stat* either `#f` or the result of the *stat* call for that entry, when available.

The special `.` and `..` entries are not passed to these procedures. The *path* argument to the procedures is a full file name—e.g., `"../foo/bar/gnu"`; if *file-name* is an absolute file name, then *path* is also an absolute file name. Files and directories, as identified by their device/inode number pair, are traversed only once.

The optional *stat* argument defaults to `lstat`, which means that symbolic links are not followed; the `stat` procedure can be used instead when symbolic links are to be followed (see Section 7.2.3 [File System], page 504).

The example below illustrates the use of `file-system-fold`:

```
(define (total-file-size file-name)
  "Return the size in bytes of the files under FILE-NAME (similar
to ‘du --apparent-size’ with GNU Coreutils.)"
```

```
(define (enter? name stat result)
  ;; Skip version control directories.
  (not (member (basename name) '(".git" ".svn" "CVS"))))
(define (leaf name stat result)
  ;; Return RESULT plus the size of the file at NAME.
  (+ result (stat:size stat)))

;; Count zero bytes for directories.
(define (down name stat result) result)
(define (up name stat result) result)

;; Likewise for skipped directories.
(define (skip name stat result) result)

;; Ignore unreadable files/directories but warn the user.
(define (error name stat errno result)
  (format (current-error-port) "warning: ~a: ~a~%"
          name (strerror errno))
  result)

(file-system-fold enter? leaf down up skip error
                  0  ; initial counter is zero bytes
                  file-name))

(total-file-size ".")
⇒ 8217554

(total-file-size "/dev/null")
⇒ 0
```

The alternative C-like functions are described below.

`scandir` *name* [*select?* [*entry<?*]] [Scheme Procedure]

Return the list of the names of files contained in directory *name* that match predicate *select?* (by default, all files). The returned list of file names is sorted according to *entry<?*, which defaults to `string-locale<?` such that file names are sorted in the locale's alphabetical order (see Section 6.25.2 [Text Collation], page 466). Return `#f` when *name* is unreadable or is not a directory.

This procedure is modeled after the C library function of the same name (see Section "Scanning Directory Content" in *GNU C Library Reference Manual*).

`ftw` *startname proc* ['*hash-size n*] [Scheme Procedure]

Walk the file system tree descending from *startname*, calling *proc* for each file and directory.

Hard links and symbolic links are followed. A file or directory is reported to *proc* only once, and skipped if seen again in another place. One consequence of this is that `ftw` is safe against circularly linked directory structures.

Each *proc* call is `(`*proc* `filename statinfo flag)` and it should return `#t` to continue, or any other value to stop.

filename is the item visited, being *startname* plus a further path and the name of the item. *statinfo* is the return from `stat` (see Section 7.2.3 [File System], page 504) on *filename*. *flag* is one of the following symbols,

`regular` *filename* is a file, this includes special files like devices, named pipes, etc.

`directory`
: *filename* is a directory.

`invalid-stat`
: An error occurred when calling `stat`, so nothing is known. *statinfo* is `#f` in this case.

`directory-not-readable`
: *filename* is a directory, but one which cannot be read and hence won't be recursed into.

`symlink` *filename* is a dangling symbolic link. Symbolic links are normally followed and their target reported, the link itself is reported if the target does not exist.

The return value from `ftw` is `#t` if it ran to completion, or otherwise the non-`#t` value from *proc* which caused the stop.

Optional argument symbol `hash-size` and an integer can be given to set the size of the hash table used to track items already visited. (see Section 6.6.22.2 [Hash Table Reference], page 239)

In the current implementation, returning non-`#t` from *proc* is the only valid way to terminate `ftw`. *proc* must not use `throw` or similar to escape.

`nftw` *startname proc* [*'chdir*] [*'depth*] [*'hash-size n*] [*'mount*] [*'physical*] [Scheme Procedure]

Walk the file system tree starting at *startname*, calling *proc* for each file and directory. `nftw` has extra features over the basic `ftw` described above.

Like `ftw`, hard links and symbolic links are followed. A file or directory is reported to *proc* only once, and skipped if seen again in another place. One consequence of this is that `nftw` is safe against circular linked directory structures.

Each *proc* call is `(`*proc* `filename statinfo flag base level)` and it should return `#t` to continue, or any other value to stop.

filename is the item visited, being *startname* plus a further path and the name of the item. *statinfo* is the return from `stat` on *filename* (see Section 7.2.3 [File System], page 504). *base* is an integer offset into *filename* which is where the basename for this item begins. *level* is an integer giving the directory nesting level, starting from 0 for the contents of *startname* (or that item itself if it's a file). *flag* is one of the following symbols,

`regular` *filename* is a file, including special files like devices, named pipes, etc.

`directory`
: *filename* is a directory.

`directory-processed`
: *filename* is a directory, and its contents have all been visited. This flag is given instead of `directory` when the `depth` option below is used.

`invalid-stat`
: An error occurred when applying `stat` to *filename*, so nothing is known about it. *statinfo* is `#f` in this case.

`directory-not-readable`
: *filename* is a directory, but one which cannot be read and hence won't be recursed into.

`stale-symlink`
: *filename* is a dangling symbolic link. Links are normally followed and their target reported, the link itself is reported if its target does not exist.

`symlink`
: When the `physical` option described below is used, this indicates *filename* is a symbolic link whose target exists (and is not being followed).

The following optional arguments can be given to modify the way `nftw` works. Each is passed as a symbol (and `hash-size` takes a following integer value).

`chdir`
: Change to the directory containing the item before calling *proc*. When `nftw` returns the original current directory is restored.

  Under this option, generally the *base* parameter to each *proc* call should be used to pick out the base part of the *filename*. The *filename* is still a path but with a changed directory it won't be valid (unless the *startname* directory was absolute).

`depth`
: Visit files "depth first", meaning *proc* is called for the contents of each directory before it's called for the directory itself. Normally a directory is reported first, then its contents.

  Under this option, the *flag* to *proc* for a directory is `directory-processed` instead of `directory`.

`hash-size` *n*
: Set the size of the hash table used to track items already visited. (see Section 6.6.22.2 [Hash Table Reference], page 239)

`mount`
: Don't cross a mount point, meaning only visit items on the same file system as *startname* (ie. the same `stat:dev`).

`physical`
: Don't follow symbolic links, instead report them to *proc* as `symlink`. Dangling links (those whose target doesn't exist) are still reported as `stale-symlink`.

The return value from `nftw` is `#t` if it ran to completion, or otherwise the non-`#t` value from *proc* which caused the stop.

In the current implementation, returning non-`#t` from *proc* is the only valid way to terminate `ftw`. *proc* must not use `throw` or similar to escape.

7.13 Queues

The functions in this section are provided by

```
(use-modules (ice-9 q))
```

This module implements queues holding arbitrary scheme objects and designed for efficient first-in / first-out operations.

`make-q` creates a queue, and objects are entered and removed with `enq!` and `deq!`. `q-push!` and `q-pop!` can be used too, treating the front of the queue like a stack.

`make-q` [Scheme Procedure]
Return a new queue.

`q?` *obj* [Scheme Procedure]
Return `#t` if *obj* is a queue, or `#f` if not.

Note that queues are not a distinct class of objects but are implemented with cons cells. For that reason certain list structures can get `#t` from `q?`.

`enq!` *q obj* [Scheme Procedure]
Add *obj* to the rear of *q*, and return *q*.

`deq!` *q* [Scheme Procedure]
`q-pop!` *q* [Scheme Procedure]
Remove and return the front element from *q*. If *q* is empty, a `q-empty` exception is thrown.

`deq!` and `q-pop!` are the same operation, the two names just let an application match `enq!` with `deq!`, or `q-push!` with `q-pop!`.

`q-push!` *q obj* [Scheme Procedure]
Add *obj* to the front of *q*, and return *q*.

`q-length` *q* [Scheme Procedure]
Return the number of elements in *q*.

`q-empty?` *q* [Scheme Procedure]
Return true if *q* is empty.

`q-empty-check` *q* [Scheme Procedure]
Throw a `q-empty` exception if *q* is empty.

`q-front` *q* [Scheme Procedure]
Return the first element of *q* (without removing it). If *q* is empty, a `q-empty` exception is thrown.

`q-rear` *q* [Scheme Procedure]
Return the last element of *q* (without removing it). If *q* is empty, a `q-empty` exception is thrown.

`q-remove!` *q obj* [Scheme Procedure]
Remove all occurrences of *obj* from *q*, and return *q*. *obj* is compared to queue elements using `eq?`.

The `q-empty` exceptions described above are thrown just as `(throw 'q-empty)`, there's no message etc like an error throw.

A queue is implemented as a cons cell, the `car` containing a list of queued elements, and the `cdr` being the last cell in that list (for ease of enqueuing).

```
(list . last-cell)
```

If the queue is empty, *list* is the empty list and *last-cell* is `#f`.

An application can directly access the queue list if desired, for instance to search the elements or to insert at a specific point.

`sync-q!` *q* [Scheme Procedure]

Recompute the *last-cell* field in *q*.

All the operations above maintain *last-cell* as described, so normally there's no need for `sync-q!`. But if an application modifies the queue *list* then it must either maintain *last-cell* similarly, or call `sync-q!` to recompute it.

7.14 Streams

This section documents Guile's legacy stream module. For a more complete and portable stream library, see Section 7.5.28 [SRFI-41], page 631.

A stream represents a sequence of values, each of which is calculated only when required. This allows large or even infinite sequences to be represented and manipulated with familiar operations like "car", "cdr", "map" or "fold". In such manipulations only as much as needed is actually held in memory at any one time. The functions in this section are available from

```
(use-modules (ice-9 streams))
```

Streams are implemented using promises (see Section 6.18.9 [Delayed Evaluation], page 396), which is how the underlying calculation of values is made only when needed, and the values then retained so the calculation is not repeated.

Here is a simple example producing a stream of all odd numbers,

```
(define odds (make-stream (lambda (state)
                            (cons state (+ state 2)))
                          1))
(stream-car odds)              ⇒ 1
(stream-car (stream-cdr odds)) ⇒ 3
```

`stream-map` could be used to derive a stream of odd squares,

```
(define (square n) (* n n))
(define oddsquares (stream-map square odds))
```

These are infinite sequences, so it's not possible to convert them to a list, but they could be printed (infinitely) with for example

```
(stream-for-each (lambda (n sq)
                   (format #t "~a squared is ~a\n" n sq))
                 odds oddsquares)
⊣
1 squared is 1
3 squared is 9
```

```
5 squared is 25
7 squared is 49
...
```

`make-stream` *proc initial-state* [Scheme Procedure]
Return a new stream, formed by calling *proc* successively.

Each call is `(proc state)`, it should return a pair, the `car` being the value for the stream, and the `cdr` being the new *state* for the next call. For the first call *state* is the given *initial-state*. At the end of the stream, *proc* should return some non-pair object.

`stream-car` *stream* [Scheme Procedure]
Return the first element from *stream*. *stream* must not be empty.

`stream-cdr` *stream* [Scheme Procedure]
Return a stream which is the second and subsequent elements of *stream*. *stream* must not be empty.

`stream-null?` *stream* [Scheme Procedure]
Return true if *stream* is empty.

`list->stream` *list* [Scheme Procedure]
`vector->stream` *vector* [Scheme Procedure]
Return a stream with the contents of *list* or *vector*.

list or *vector* should not be modified subsequently, since it's unspecified whether changes there will be reflected in the stream returned.

`port->stream` *port readproc* [Scheme Procedure]
Return a stream which is the values obtained by reading from *port* using *readproc*. Each read call is `(readproc port)`, and it should return an EOF object (see Section 6.14.2 [Binary I/O], page 332) at the end of input.

For example a stream of characters from a file,

```
(port->stream (open-input-file "/foo/bar.txt") read-char)
```

`stream->list` *stream* [Scheme Procedure]
Return a list which is the entire contents of *stream*.

`stream->reversed-list` *stream* [Scheme Procedure]
Return a list which is the entire contents of *stream*, but in reverse order.

`stream->list&length` *stream* [Scheme Procedure]
Return two values (see Section 6.13.7 [Multiple Values], page 309), being firstly a list which is the entire contents of *stream*, and secondly the number of elements in that list.

`stream->reversed-list&length` *stream* [Scheme Procedure]
Return two values (see Section 6.13.7 [Multiple Values], page 309) being firstly a list which is the entire contents of *stream*, but in reverse order, and secondly the number of elements in that list.

`stream->vector` *stream* [Scheme Procedure]
Return a vector which is the entire contents of *stream*.

`stream-fold` *proc init stream1 stream2 . . .* [Function]
Apply *proc* successively over the elements of the given streams, from first to last until the end of the shortest stream is reached. Return the result from the last *proc* call.

Each call is `(`*proc* `elem1 elem2 ... prev)`, where each *elem* is from the corresponding *stream*. *prev* is the return from the previous *proc* call, or the given *init* for the first call.

`stream-for-each` *proc stream1 stream2 . . .* [Function]
Call *proc* on the elements from the given *stream*s. The return value is unspecified.

Each call is `(`*proc* `elem1 elem2 ...)`, where each *elem* is from the corresponding *stream*. `stream-for-each` stops when it reaches the end of the shortest *stream*.

`stream-map` *proc stream1 stream2 . . .* [Function]
Return a new stream which is the results of applying *proc* to the elements of the given *stream*s.

Each call is `(`*proc* `elem1 elem2 ...)`, where each *elem* is from the corresponding *stream*. The new stream ends when the end of the shortest given *stream* is reached.

7.15 Buffered Input

The following functions are provided by

```
(use-modules (ice-9 buffered-input))
```

A buffered input port allows a reader function to return chunks of characters which are to be handed out on reading the port. A notion of further input for an application level logical expression is maintained too, and passed through to the reader.

`make-buffered-input-port` *reader* [Scheme Procedure]
Create an input port which returns characters obtained from the given *reader* function. *reader* is called (*reader* cont), and should return a string or an EOF object.

The new port gives precisely the characters returned by *reader*, nothing is added, so if any newline characters or other separators are desired they must come from the reader function.

The *cont* parameter to *reader* is `#f` for initial input, or `#t` when continuing an expression. This is an application level notion, set with `set-buffered-input-continuation?!` below. If the user has entered a partial expression then it allows *reader* for instance to give a different prompt to show more is required.

`make-line-buffered-input-port` *reader* [Scheme Procedure]
Create an input port which returns characters obtained from the specified *reader* function, similar to `make-buffered-input-port` above, but where *reader* is expected to be a line-oriented.

reader is called (*reader* cont), and should return a string or an EOF object as above. Each string is a line of input without a newline character, the port code inserts a newline after each string.

`set-buffered-input-continuation?!` *port cont* [Scheme Procedure]

Set the input continuation flag for a given buffered input *port*.

An application uses this by calling with a *cont* flag of `#f` when beginning to read a new logical expression. For example with the Scheme `read` function (see Section 6.18.2 [Scheme Read], page 385),

```
(define my-port (make-buffered-input-port my-reader))

(set-buffered-input-continuation?! my-port #f)
(let ((obj (read my-port)))
  ...
```

7.16 Expect

The macros in this section are made available with:

```
(use-modules (ice-9 expect))
```

`expect` is a macro for selecting actions based on the output from a port. The name comes from a tool of similar functionality by Don Libes. Actions can be taken when a particular string is matched, when a timeout occurs, or when end-of-file is seen on the port. The `expect` macro is described below; `expect-strings` is a front-end to `expect` based on regexec (see the regular expression documentation).

`expect-strings` *clause ...* [Macro]

By default, `expect-strings` will read from the current input port. The first term in each clause consists of an expression evaluating to a string pattern (regular expression). As characters are read one-by-one from the port, they are accumulated in a buffer string which is matched against each of the patterns. When a pattern matches, the remaining expression(s) in the clause are evaluated and the value of the last is returned. For example:

```
(with-input-from-file "/etc/passwd"
  (lambda ()
    (expect-strings
      ("^nobody" (display "Got a nobody user.\n")
                 (display "That's no problem.\n"))
      ("^daemon" (display "Got a daemon user.\n")))))
```

The regular expression is compiled with the `REG_NEWLINE` flag, so that the `^` and $ anchors will match at any newline, not just at the start and end of the string.

There are two other ways to write a clause:

The expression(s) to evaluate can be omitted, in which case the result of the regular expression match (converted to strings, as obtained from regexec with match-pick set to `""`) will be returned if the pattern matches.

The symbol `=>` can be used to indicate that the expression is a procedure which will accept the result of a successful regular expression match. E.g.,

```
("^daemon" => write)
("^d(aemon)" => (lambda args (for-each write args)))
("^da(em)on" => (lambda (all sub)
```

```
                    (write all) (newline)
                    (write sub) (newline)))
```

The order of the substrings corresponds to the order in which the opening brackets occur.

A number of variables can be used to control the behaviour of `expect` (and `expect-strings`). Most have default top-level bindings to the value `#f`, which produces the default behaviour. They can be redefined at the top level or locally bound in a form enclosing the expect expression.

`expect-port`
: A port to read characters from, instead of the current input port.

`expect-timeout`
: `expect` will terminate after this number of seconds, returning `#f` or the value returned by expect-timeout-proc.

`expect-timeout-proc`
: A procedure called if timeout occurs. The procedure takes a single argument: the accumulated string.

`expect-eof-proc`
: A procedure called if end-of-file is detected on the input port. The procedure takes a single argument: the accumulated string.

`expect-char-proc`
: A procedure to be called every time a character is read from the port. The procedure takes a single argument: the character which was read.

`expect-strings-compile-flags`
: Flags to be used when compiling a regular expression, which are passed to `make-regexp` See Section 6.15.1 [Regexp Functions], page 359. The default value is `regexp/newline`.

`expect-strings-exec-flags`
: Flags to be used when executing a regular expression, which are passed to regexp-exec See Section 6.15.1 [Regexp Functions], page 359. The default value is `regexp/noteol`, which prevents `$` from matching the end of the string while it is still accumulating, but still allows it to match after a line break or at the end of file.

Here's an example using all of the variables:

```
(let ((expect-port (open-input-file "/etc/passwd"))
      (expect-timeout 1)
      (expect-timeout-proc
        (lambda (s) (display "Times up!\n")))
      (expect-eof-proc
        (lambda (s) (display "Reached the end of the file!\n")))
      (expect-char-proc display)
      (expect-strings-compile-flags (logior regexp/newline regexp/icase))
      (expect-strings-exec-flags 0))
  (expect-strings
    ("^nobody"  (display "Got a nobody user\n"))))
```

`expect` *clause ...* [Macro]

`expect` is used in the same way as `expect-strings`, but tests are specified not as patterns, but as procedures. The procedures are called in turn after each character is read from the port, with two arguments: the value of the accumulated string and a flag to indicate whether end-of-file has been reached. The flag will usually be `#f`, but if end-of-file is reached, the procedures are called an additional time with the final accumulated string and `#t`.

The test is successful if the procedure returns a non-false value.

If the `=>` syntax is used, then if the test succeeds it must return a list containing the arguments to be provided to the corresponding expression.

In the following example, a string will only be matched at the beginning of the file:

```
(let ((expect-port (open-input-file "/etc/passwd")))
  (expect
     ((lambda (s eof?) (string=? s "fnord!"))
        (display "Got a nobody user!\n"))))
```

The control variables described for `expect-strings` also influence the behaviour of `expect`, with the exception of variables whose names begin with `expect-strings-`.

7.17 `sxml-match`: Pattern Matching of SXML

The `(sxml match)` module provides syntactic forms for pattern matching of SXML trees, in a "by example" style reminiscent of the pattern matching of the `syntax-rules` and `syntax-case` macro systems. See Section 7.21 [SXML], page 748, for more information on SXML.

The following example[5] provides a brief illustration, transforming a music album catalog language into HTML.

```
(define (album->html x)
  (sxml-match x
    ((album (@ (title ,t)) (catalog (num ,n) (fmt ,f)) ...)
     `(ul (li ,t)
          (li (b ,n) (i ,f)) ...))))
```

Three macros are provided: `sxml-match`, `sxml-match-let`, and `sxml-match-let*`.

Compared to a standard s-expression pattern matcher (see Section 7.8 [Pattern Matching], page 706), `sxml-match` provides the following benefits:

- matching of SXML elements does not depend on any degree of normalization of the SXML;
- matching of SXML attributes (within an element) is under-ordered; the order of the attributes specified within the pattern need not match the ordering with the element being matched;
- all attributes specified in the pattern must be present in the element being matched; in the spirit that XML is 'extensible', the element being matched may include additional attributes not specified in the pattern.

[5] This example is taken from a paper by Krishnamurthi et al. Their paper was the first to show the usefulness of the `syntax-rules` style of pattern matching for transformation of XML, though the language described, XT3D, is an XML language.

The present module is a descendant of WebIt!, and was inspired by an s-expression pattern matcher developed by Erik Hilsdale, Dan Friedman, and Kent Dybvig at Indiana University.

Syntax

`sxml-match` provides `case`-like form for pattern matching of XML nodes.

`sxml-match` *input-expression clause1 clause2 . . .* [Scheme Syntax]
Match *input-expression*, an SXML tree, according to the given *clause*s (one or more), each consisting of a pattern and one or more expressions to be evaluated if the pattern match succeeds. Optionally, each *clause* within `sxml-match` may include a *guard expression*.

The pattern notation is based on that of Scheme's `syntax-rules` and `syntax-case` macro systems. The grammar for the `sxml-match` syntax is given below:

```
match-form ::= (sxml-match input-expression
                 clause+)

clause ::= [node-pattern action-expression+]
         | [node-pattern (guard expression*) action-expression+]

node-pattern ::= literal-pattern
               | pat-var-or-cata
               | element-pattern
               | list-pattern

literal-pattern ::= string
                  | character
                  | number
                  | #t
                  | #f

attr-list-pattern ::= (@ attribute-pattern*)
                    | (@ attribute-pattern* . pat-var-or-cata)

attribute-pattern ::= (tag-symbol attr-val-pattern)

attr-val-pattern ::= literal-pattern
                   | pat-var-or-cata
                   | (pat-var-or-cata default-value-expr)

element-pattern ::= (tag-symbol attr-list-pattern?)
                  | (tag-symbol attr-list-pattern? nodeset-pattern)
                  | (tag-symbol attr-list-pattern?
                                nodeset-pattern? . pat-var-or-cata)

list-pattern ::= (list nodeset-pattern)
```

```
               | (list nodeset-pattern? . pat-var-or-cata)
               | (list)

nodeset-pattern ::= node-pattern
                  | node-pattern ...
                  | node-pattern nodeset-pattern
                  | node-pattern ... nodeset-pattern

pat-var-or-cata ::= (unquote var-symbol)
                  | (unquote [var-symbol*])
                  | (unquote [cata-expression -> var-symbol*])
```

Within a list or element body pattern, ellipses may appear only once, but may be followed by zero or more node patterns.

Guard expressions cannot refer to the return values of catamorphisms.

Ellipses in the output expressions must appear only in an expression context; ellipses are not allowed in a syntactic form.

The sections below illustrate specific aspects of the `sxml-match` pattern matcher.

Matching XML Elements

The example below illustrates the pattern matching of an XML element:

```
(sxml-match '(e (@ (i 1)) 3 4 5)
  ((e (@ (i ,d)) ,a ,b ,c) (list d a b c))
  (,otherwise #f))
```

Each clause in `sxml-match` contains two parts: a pattern and one or more expressions which are evaluated if the pattern is successfully match. The example above matches an element `e` with an attribute `i` and three children.

Pattern variables must be "unquoted" in the pattern. The above expression binds *d* to `1`, *a* to `3`, *b* to `4`, and *c* to `5`.

Ellipses in Patterns

As in `syntax-rules`, ellipses may be used to specify a repeated pattern. Note that the pattern `item ...` specifies zero-or-more matches of the pattern `item`.

The use of ellipses in a pattern is illustrated in the code fragment below, where nested ellipses are used to match the children of repeated instances of an `a` element, within an element `d`.

```
(define x '(d (a 1 2 3) (a 4 5) (a 6 7 8) (a 9 10)))

(sxml-match x
  ((d (a ,b ...) ...)
   (list (list b ...) ...)))
```

The above expression returns a value of `((1 2 3) (4 5) (6 7 8) (9 10))`.

Ellipses in Quasiquote'd Output

Within the body of an `sxml-match` form, a slightly extended version of quasiquote is provided, which allows the use of ellipses. This is illustrated in the example below.

```
(sxml-match '(e 3 4 5 6 7)
  ((e ,i ... 6 7) `("start" ,(list 'wrap i) ... "end"))
  (,otherwise #f))
```

The general pattern is that `` `(something ,i ...) `` is rewritten as `` `(something ,@i) ``.

Matching Nodesets

A nodeset pattern is designated by a list in the pattern, beginning the identifier list. The example below illustrates matching a nodeset.

```
(sxml-match '("i" "j" "k" "l" "m")
  ((list ,a ,b ,c ,d ,e)
   `((p ,a) (p ,b) (p ,c) (p ,d) (p ,e))))
```

This example wraps each nodeset item in an HTML paragraph element. This example can be rewritten and simplified through using ellipsis:

```
(sxml-match '("i" "j" "k" "l" "m")
  ((list ,i ...)
   `((p ,i) ...)))
```

This version will match nodesets of any length, and wrap each item in the nodeset in an HTML paragraph element.

Matching the "Rest" of a Nodeset

Matching the "rest" of a nodeset is achieved by using a `. rest)` pattern at the end of an element or nodeset pattern.

This is illustrated in the example below:

```
(sxml-match '(e 3 (f 4 5 6) 7)
  ((e ,a (f . ,y) ,d)
   (list a y d)))
```

The above expression returns `(3 (4 5 6) 7)`.

Matching the Unmatched Attributes

Sometimes it is useful to bind a list of attributes present in the element being matched, but which do not appear in the pattern. This is achieved by using a `. rest)` pattern at the end of the attribute list pattern. This is illustrated in the example below:

```
(sxml-match '(a (@ (z 1) (y 2) (x 3)) 4 5 6)
  ((a (@ (y ,www) . ,qqq) ,t ,u ,v)
   (list www qqq t u v)))
```

The above expression matches the attribute `y` and binds a list of the remaining attributes to the variable *qqq*. The result of the above expression is `(2 ((z 1) (x 3)) 4 5 6)`.

This type of pattern also allows the binding of all attributes:

```
(sxml-match '(a (@ (z 1) (y 2) (x 3)))
  ((a (@ . ,qqq))
   qqq))
```

Default Values in Attribute Patterns

It is possible to specify a default value for an attribute which is used if the attribute is not present in the element being matched. This is illustrated in the following example:

```
(sxml-match '(e 3 4 5)
  ((e (@ (z (,d 1))) ,a ,b ,c) (list d a b c)))
```

The value `1` is used when the attribute `z` is absent from the element `e`.

Guards in Patterns

Guards may be added to a pattern clause via the `guard` keyword. A guard expression may include zero or more expressions which are evaluated only if the pattern is matched. The body of the clause is only evaluated if the guard expressions evaluate to `#t`.

The use of guard expressions is illustrated below:

```
(sxml-match '(a 2 3)
  ((a ,n) (guard (number? n)) n)
  ((a ,m ,n) (guard (number? m) (number? n)) (+ m n)))
```

Catamorphisms

The example below illustrates the use of explicit recursion within an `sxml-match` form. This example implements a simple calculator for the basic arithmetic operations, which are represented by the XML elements `plus`, `minus`, `times`, and `div`.

```
(define simple-eval
  (lambda (x)
    (sxml-match x
      (,i (guard (integer? i)) i)
      ((plus ,x ,y) (+ (simple-eval x) (simple-eval y)))
      ((times ,x ,y) (* (simple-eval x) (simple-eval y)))
      ((minus ,x ,y) (- (simple-eval x) (simple-eval y)))
      ((div ,x ,y) (/ (simple-eval x) (simple-eval y)))
      (,otherwise (error "simple-eval: invalid expression" x)))))
```

Using the catamorphism feature of `sxml-match`, a more concise version of `simple-eval` can be written. The pattern `,(x)` recursively invokes the pattern matcher on the value bound in this position.

```
(define simple-eval
  (lambda (x)
    (sxml-match x
      (,i (guard (integer? i)) i)
      ((plus ,(x) ,(y)) (+ x y))
      ((times ,(x) ,(y)) (* x y))
      ((minus ,(x) ,(y)) (- x y))
      ((div ,(x) ,(y)) (/ x y))
      (,otherwise (error "simple-eval: invalid expression" x)))))
```

Named-Catamorphisms

It is also possible to explicitly name the operator in the "cata" position. Where `,(id*)` recurs to the top of the current `sxml-match`, `,(cata -> id*)` recurs to `cata`. `cata` must

evaluate to a procedure which takes one argument, and returns as many values as there are identifiers following `->`.

Named catamorphism patterns allow processing to be split into multiple, mutually recursive procedures. This is illustrated in the example below: a transformation that formats a "TV Guide" into HTML.

```
(define (tv-guide->html g)
  (define (cast-list cl)
    (sxml-match cl
      ((CastList (CastMember (Character (Name ,ch)) (Actor (Name ,a))) ...)
       `(div (ul (li ,ch ": " ,a) ...)))))
  (define (prog p)
    (sxml-match p
      ((Program (Start ,start-time) (Duration ,dur) (Series ,series-title)
                (Description ,desc ...))
       `(div (p ,start-time
                (br) ,series-title
                (br) ,desc ...)))
      ((Program (Start ,start-time) (Duration ,dur) (Series ,series-title)
                (Description ,desc ...)
                ,(cast-list -> cl))
       `(div (p ,start-time
                (br) ,series-title
                (br) ,desc ...)
             ,cl))))
  (sxml-match g
    ((TVGuide (@ (start ,start-date)
                 (end ,end-date))
              (Channel (Name ,nm) ,(prog -> p) ...) ...)
     `(html (head (title "TV Guide"))
            (body (h1 "TV Guide")
                  (div (h2 ,nm) ,p ...) ...)))))
```

`sxml-match-let` and `sxml-match-let*`

`sxml-match-let` ((*pat expr*) ...) *expression0 expression ...* [Scheme Syntax]
`sxml-match-let*` ((*pat expr*) ...) *expression0 expression ...* [Scheme Syntax]

These forms generalize the `let` and `let*` forms of Scheme to allow an XML pattern in the binding position, rather than a simple variable.

For example, the expression below:

```
(sxml-match-let (((a ,i ,j) '(a 1 2)))
  (+ i j))
```

binds the variables *i* and *j* to `1` and `2` in the XML value given.

7.18 The Scheme shell (scsh)

An incomplete port of the Scheme shell (scsh) was once available for Guile as a separate package. However this code has bitrotten somewhat. The pieces are available in Guile's

legacy CVS repository, which may be browsed at `http://cvs.savannah.gnu.org/viewvc/guile/guile-scsh/?root=guile`.

For information about scsh see `http://www.scsh.net/`.

This bitrotting is a bit of a shame, as there is a good deal of well-written Scheme code in scsh. Adopting this code and porting it to current Guile should be an educational experience, in addition to providing something of value to Guile folks.

7.19 Curried Definitions

The macros in this section are provided by

```
(use-modules (ice-9 curried-definitions))
```

and replace those provided by default.

Prior to Guile 2.0, Guile provided a type of definition known colloquially as a "curried definition". The idea is to extend the syntax of `define` so that you can conveniently define procedures that return procedures, up to any desired depth.

For example,

```
(define ((foo x) y)
  (list x y))
```

is a convenience form of

```
(define foo
  (lambda (x)
    (lambda (y)
      (list x y))))
```

`define` (. . . (*name args* . . .) . . .) *body* . . . [Scheme Syntax]
`define*` (. . . (*name args* . . .) . . .) *body* . . . [Scheme Syntax]
`define-public` (. . . (*name args* . . .) . . .) *body* . . . [Scheme Syntax]

Create a top level variable *name* bound to the procedure with parameter list *args*. If *name* is itself a formal parameter list, then a higher order procedure is created using that formal-parameter list, and returning a procedure that has parameter list *args*. This nesting may occur to arbitrary depth.

`define*` is similar but the formal parameter lists take additional options as described in Section 6.9.4.1 [lambda* and define*], page 253. For example,

```
(define* ((foo #:keys (bar 'baz) (quux 'zot)) frotz #:rest rest)
  (list bar quux frotz rest))


((foo #:quux 'foo) 1 2 3 4 5)
⇒ (baz foo 1 (2 3 4 5))
```

`define-public` is similar to `define` but it also adds *name* to the list of exported bindings of the current module.

7.20 Statprof

Statprof is a statistical profiler for Guile.

A simple use of statprof would look like this:

```
(use-modules (statprof))
(statprof (lambda ()
            (map 1+ (iota 1000000))
            #f))
```

This would run the thunk with statistical profiling, finally displaying a flat table of statistics which could look something like this:

```
%     cumulative   self
time   seconds     seconds  procedure
 57.14  39769.73      0.07  ice-9/boot-9.scm:249:5:map1
 28.57      0.04      0.04  ice-9/boot-9.scm:1165:0:iota
 14.29      0.02      0.02  1+
  0.00      0.12      0.00  <current input>:2:10
---
Sample count: 7
Total time: 0.123490713 seconds (0.201983993 seconds in GC)
```

All of the numerical data with the exception of the calls column is statistically approximate. In the following column descriptions, and in all of statprof, "time" refers to execution time (both user and system), not wall clock time.

The `% time` column indicates the percentage of the run-time time spent inside the procedure itself (not counting children). It is calculated as `self seconds`, measuring the amount of time spent in the procedure, divided by the total run-time.

`cumulative seconds` also counts time spent in children of a function. For recursive functions, this can exceed the total time, as in our example above, because each activation on the stack adds to the cumulative time.

Finally, the GC time measures the time spent in the garbage collector. On systems with multiple cores, this time can be larger than the run time, because it counts time spent in all threads, and will run the "marking" phase of GC in parallel. If GC time is a significant fraction of the run time, that means that most time in your program is spent allocating objects and cleaning up after those allocations. To speed up your program, one good place to start would be to look at how to reduce the allocation rate.

Statprof's main mode of operation is as a statistical profiler. However statprof can also run in a "precise" mode as well. Pass the `#:count-calls? #t` keyword argument to `statprof` to record all calls:

```
(use-modules (statprof))
(statprof (lambda ()
            (map 1+ (iota 1000000))
            #f)
          #:count-calls? #t)
```

The result has an additional `calls` column:

```
%     cumulative   self
time   seconds    seconds   calls   procedure
```

```
 82.26      0.73      0.73 1000000  1+
 11.29 420925.80      0.10 1000001  ice-9/boot-9.scm:249:5:map1
  4.84      0.06      0.04       1  ice-9/boot-9.scm:1165:0:iota
[...]
---
Sample count: 62
Total time: 0.893098065 seconds (1.222796536 seconds in GC)
```

As you can see, the profile is perturbed: `1+` ends up on top, whereas it was not marked as hot in the earlier profile. This is because the overhead of call-counting unfairly penalizes calls. Still, this precise mode can be useful at times to do algorithmic optimizations based on the precise call counts.

Implementation notes

The profiler works by setting the unix profiling signal `ITIMER_PROF` to go off after the interval you define in the call to `statprof-reset`. When the signal fires, a sampling routine runs which crawls up the stack, recording all instruction pointers into a buffer. After the sample is complete, the profiler resets profiling timer to fire again after the appropriate interval.

Later, when profiling stops, that log buffer is analyzed to produce the "self seconds" and "cumulative seconds" statistics. A procedure at the top of the stack counts toward "self" samples, and everything on the stack counts towards "cumulative" samples.

While the profiler is running it measures how much CPU time (system and user – which is also what `ITIMER_PROF` tracks) has elapsed while code has been executing within the profiler. Only run time counts towards the profile, not wall-clock time. For example, sleeping and waiting for input or output do not cause the timer clock to advance.

Usage

`statprof` *thunk* [*#:loop loop=1*] [*#:hz hz=100*] [*#:port port=(current-output-port)*] [*#:count-calls? count-calls?=#f*] [*#:display-style display-style='flat*] [Scheme Procedure]

Profile the execution of *thunk*, and return its return values.

The stack will be sampled *hz* times per second, and the thunk itself will be called *loop* times.

If *count-calls?* is true, all procedure calls will be recorded. This operation is somewhat expensive.

After the *thunk* has been profiled, print out a profile to *port*. If *display-style* is `flat`, the results will be printed as a flat profile. Otherwise if *display-style* is `tree`, print the results as a tree profile.

Note that `statprof` requires a working profiling timer. Some platforms do not support profiling timers. `(provided? 'ITIMER_PROF)` can be used to check for support of profiling timers.

Profiling can also be enabled and disabled manually.

`statprof-active?` [Scheme Procedure]
: Returns `#t` if `statprof-start` has been called more times than `statprof-stop`, `#f` otherwise.

`statprof-start` [Scheme Procedure]
`statprof-stop` [Scheme Procedure]
: Start or stop the profiler.

`statprof-reset` *sample-seconds sample-microseconds count-calls?* [Scheme Procedure]
: Reset the profiling sample interval to *sample-seconds* and *sample-microseconds*. If *count-calls?* is true, arrange to instrument procedure calls as well as collecting statistical profiling data.

If you use the manual `statprof-start`/`statprof-stop` interface, an implicit statprof state will persist starting from the last call to `statprof-reset`, or the first call to `statprof-start`. There are a number of accessors to fetch statistics from this implicit state.

`statprof-accumulated-time` [Scheme Procedure]
: Returns the time accumulated during the last statprof run.

`statprof-sample-count` [Scheme Procedure]
: Returns the number of samples taken during the last statprof run.

`statprof-fold-call-data` *proc init* [Scheme Procedure]
: Fold *proc* over the call-data accumulated by statprof. This procedure cannot be called while statprof is active.

  proc will be called with arguments, *call-data* and *prior-result*.

`statprof-proc-call-data` *proc* [Scheme Procedure]
: Returns the call-data associated with *proc*, or `#f` if none is available.

`statprof-call-data-name` *cd* [Scheme Procedure]
`statprof-call-data-calls` *cd* [Scheme Procedure]
`statprof-call-data-cum-samples` *cd* [Scheme Procedure]
`statprof-call-data-self-samples` *cd* [Scheme Procedure]
: Accessors for the fields in a statprof call-data object.

`statprof-call-data->stats` *call-data* [Scheme Procedure]
: Returns an object of type `statprof-stats`.

`statprof-stats-proc-name` *stats* [Scheme Procedure]
`statprof-stats-%-time-in-proc` *stats* [Scheme Procedure]
`statprof-stats-cum-secs-in-proc` *stats* [Scheme Procedure]
`statprof-stats-self-secs-in-proc` *stats* [Scheme Procedure]
`statprof-stats-calls` *stats* [Scheme Procedure]
`statprof-stats-self-secs-per-call` *stats* [Scheme Procedure]
`statprof-stats-cum-secs-per-call` *stats* [Scheme Procedure]
: Accessors for the fields in a `statprof-stats` object.

`statprof-display` [*port=(current-output-port)*] [*#:style style=flat*] [Scheme Procedure]

Displays a summary of the statistics collected. Possible values for *style* include:

`flat` Display a traditional gprof-style flat profile.

`anomalies`
Find statistical anomalies in the data.

`tree` Display a tree profile.

`statprof-fetch-stacks` [Scheme Procedure]

Returns a list of stacks, as they were captured since the last call to `statprof-reset`.

`statprof-fetch-call-tree` [*#:precise precise?=#f*] [Scheme Procedure]

```
Return a call tree for the previous statprof run.

The return value is a list of nodes.  A node is a list of the form:
@code
 node ::= (@var{proc} @var{count} . @var{nodes})
@end code

The @var{proc} is a printable representation of a procedure, as a
string.  If @var{precise?} is false, which is the default, then a node
corresponds to a procedure invocation.  If it is true, then a node
corresponds to a return point in a procedure.  Passing @code{#:precise?
#t} allows a user to distinguish different source lines in a procedure,
but usually it is too much detail, so it is off by default.
```

`gcprof` *thunk* [*#:loop*] [Scheme Procedure]

Like the `statprof` procedure, but instead of profiling CPU time, we profile garbage collection.

The stack will be sampled soon after every garbage collection during the evaluation of *thunk*, yielding an approximate idea of what is causing allocation in your program.

Since GC does not occur very frequently, you may need to use the *loop* parameter, to cause *thunk* to be called *loop* times.

7.21 SXML

SXML is a native representation of XML in terms of standard Scheme data types: lists, symbols, and strings. For example, the simple XML fragment:

```
<parrot type="African Grey"><name>Alfie</name></parrot>
```

may be represented with the following SXML:

```
(parrot (@ (type "African Grey")) (name "Alfie"))
```

SXML is very general, and is capable of representing all of XML. Formally, this means that SXML is a conforming implementation of the http://www.w3.org/TR/xml-infoset/ (`XML Information Set`) standard.

Guile includes several facilities for working with XML and SXML: parsers, serializers, and transformers.

7.21.1 SXML Overview

(This section needs to be written; volunteers welcome.)

7.21.2 Reading and Writing XML

The `(sxml simple)` module presents a basic interface for parsing XML from a port into the Scheme SXML format, and for serializing it back to text.

```
(use-modules (sxml simple))
```

`xml->sxml` [*string-or-port*] [*#:namespaces='()*] [Scheme Procedure]
[*#:declare-namespaces?=#t*] [*#:trim-whitespace?=#f*] [*#:entities='()*]
[*#:default-entity-handler=#f*] [*#:doctype-handler=#f*]

Use SSAX to parse an XML document into SXML. Takes one optional argument, *string-or-port*, which defaults to the current input port. Returns the resulting SXML document. If *string-or-port* is a port, it will be left pointing at the next available character in the port.

As is normal in SXML, XML elements parse as tagged lists. Attributes, if any, are placed after the tag, within an `@` element. The root of the resulting XML will be contained in a special tag, `*TOP*`. This tag will contain the root element of the XML, but also any prior processing instructions.

```
(xml->sxml "<foo/>")
⇒ (*TOP* (foo))
(xml->sxml "<foo>text</foo>")
⇒ (*TOP* (foo "text"))
(xml->sxml "<foo kind=\"bar\">text</foo>")
⇒ (*TOP* (foo (@ (kind "bar")) "text"))
(xml->sxml "<?xml version=\"1.0\"?><foo/>")
⇒ (*TOP* (*PI* xml "version=\"1.0\"") (foo))
```

All namespaces in the XML document must be declared, via `xmlns` attributes. SXML elements built from non-default namespaces will have their tags prefixed with their URI. Users can specify custom prefixes for certain namespaces with the `#:namespaces` keyword argument to `xml->sxml`.

```
(xml->sxml "<foo xmlns=\"http://example.org/ns1\">text</foo>")
⇒ (*TOP* (http://example.org/ns1:foo "text"))
(xml->sxml "<foo xmlns=\"http://example.org/ns1\">text</foo>"
           #:namespaces '((ns1 . "http://example.org/ns1")))
⇒ (*TOP* (ns1:foo "text"))
(xml->sxml "<foo xmlns:bar=\"http://example.org/ns2\"><bar:baz/></foo>"
           #:namespaces '((ns2 . "http://example.org/ns2")))
⇒ (*TOP* (foo (ns2:baz)))
```

By default, namespaces passed to `xml->sxml` are treated as if they were declared on the root element. Passing a false `#:declare-namespaces?` argument will disable this behavior, requiring in-document declarations of namespaces before use..

```
(xml->sxml "<foo><ns2:baz/></foo>"
           #:namespaces '((ns2 . "http://example.org/ns2")))
⇒ (*TOP* (foo (ns2:baz)))
```

```
(xml->sxml "<foo><ns2:baz/></foo>"
           #:namespaces '((ns2 . "http://example.org/ns2"))
           #:declare-namespaces? #f)
⇒ error: undeclared namespace: ‘bar’
```

By default, all whitespace in XML is significant. Passing the `#:trim-whitespace?` keyword argument to `xml->sxml` will trim whitespace in front, behind and between elements, treating it as "unsignificant". Whitespace in text fragments is left alone.

```
(xml->sxml "<foo>\n<bar> Alfie the parrot! </bar>\n</foo>")
⇒ (*TOP* (foo "\n" (bar " Alfie the parrot! ") "\n"))
(xml->sxml "<foo>\n<bar> Alfie the parrot! </bar>\n</foo>"
           #:trim-whitespace? #t)
⇒ (*TOP* (foo (bar " Alfie the parrot! ")))
```

Parsed entities may be declared with the `#:entities` keyword argument, or handled with the `#:default-entity-handler`. By default, only the standard `<`, `>`, `&`, `&apos;` and `&quot;` entities are defined, as well as the `&#`*N*`;` and `&#x`*N*`;` (decimal and hexadecimal) numeric character entities.

```
(xml->sxml "<foo>&amp;</foo>")
⇒ (*TOP* (foo "&"))
(xml->sxml "<foo> </foo>")
⇒ error: undefined entity: nbsp
(xml->sxml "<foo> </foo>")
⇒ (*TOP* (foo "\xa0"))
(xml->sxml "<foo> </foo>"
           #:entities '((nbsp . "\xa0")))
⇒ (*TOP* (foo "\xa0"))
(xml->sxml "<foo>  &foo;</foo>"
           #:default-entity-handler
           (lambda (port name)
             (case name
               ((nbsp) "\xa0")
               (else
                (format (current-warning-port)
                        "~a:~a:~a: undefined entitity: ~a\n"
                        (or (port-filename port) "<unknown file>")
                        (port-line port) (port-column port)
                        name)
                (symbol->string name)))))
⊣ <unknown file>:0:17: undefined entitity: foo
⇒ (*TOP* (foo "\xa0 foo"))
```

By default, `xml->sxml` skips over the `<!DOCTYPE>` declaration, if any. This behavior can be overridden with the `#:doctype-handler` argument, which should be a procedure of three arguments: the *docname* (a symbol), *systemid* (a string), and the internal doctype subset (as a string or `#f` if not present).

The handler should return keyword arguments as multiple values, as if it were calling its continuation with keyword arguments. The continuation accepts the `#:entities` and

`#:namespaces` keyword arguments, in the same format that `xml->sxml` itself takes. These entities and namespaces will be prepended to those given to the `xml->sxml` invocation.

```
(define (handle-foo docname systemid internal-subset)
  (case docname
    ((foo)
     (values #:entities '((greets . "<i>Hello, world!</i>"))))
    (else
     (values))))

(xml->sxml "<!DOCTYPE foo><p>&greets;</p>"
           #:doctype-handler handle-foo)
⇒ (*TOP* (p (i "Hello, world!")))
```

If the document has no doctype declaration, the *doctype-handler* is invoked with `#f` for the three arguments.

In the future, the continuation may accept other keyword arguments, for example to validate the parsed SXML against the doctype.

`sxml->xml` *tree* [*port*] [Scheme Procedure]
Serialize the SXML tree *tree* as XML. The output will be written to the current output port, unless the optional argument *port* is present.

`sxml->string` *sxml* [Scheme Procedure]
Detag an sxml tree *sxml* into a string. Does not perform any formatting.

7.21.3 SSAX: A Functional XML Parsing Toolkit

Guile's XML parser is based on Oleg Kiselyov's powerful XML parsing toolkit, SSAX.

7.21.3.1 History

Back in the 1990s, when the world was young again and XML was the solution to all of its problems, there were basically two kinds of XML parsers out there: DOM parsers and SAX parsers.

A DOM parser reads through an entire XML document, building up a tree of "DOM objects" representing the document structure. They are very easy to use, but sometimes you don't actually want all of the information in a document; building an object tree is not necessary if all you want to do is to count word frequencies in a document, for example.

SAX parsers were created to give the programmer more control on the parsing process. A programmer gives the SAX parser a number of "callbacks": functions that will be called on various features of the XML stream as they are encountered. SAX parsers are more efficient, but much harder to use, as users typically have to manually maintain a stack of open elements.

Kiselyov realized that the SAX programming model could be made much simpler if the callbacks were formulated not as a linear fold across the features of the XML stream, but as a *tree fold* over the structure implicit in the XML. In this way, the user has a very convenient, functional-style interface that can still generate optimal parsers.

The `xml->sxml` interface from the `(sxml simple)` module is a DOM-style parser built using SSAX, though it returns SXML instead of DOM objects.

7.21.3.2 Implementation

`(sxml ssax)` is a package of low-to-high level lexing and parsing procedures that can be combined to yield a SAX, a DOM, a validating parser, or a parser intended for a particular document type. The procedures in the package can be used separately to tokenize or parse various pieces of XML documents. The package supports XML Namespaces, internal and external parsed entities, user-controlled handling of whitespace, and validation. This module therefore is intended to be a framework, a set of "Lego blocks" you can use to build a parser following any discipline and performing validation to any degree. As an example of the parser construction, the source file includes a semi-validating SXML parser.

SSAX has a "sequential" feel of SAX yet a "functional style" of DOM. Like a SAX parser, the framework scans the document only once and permits incremental processing. An application that handles document elements in order can run as efficiently as possible. *Unlike* a SAX parser, the framework does not require an application register stateful call-backs and surrender control to the parser. Rather, it is the application that can drive the framework – calling its functions to get the current lexical or syntax element. These functions do not maintain or mutate any state save the input port. Therefore, the framework permits parsing of XML in a pure functional style, with the input port being a monad (or a linear, read-once parameter).

Besides the *port*, there is another monad – *seed*. Most of the middle- and high-level parsers are single-threaded through the *seed*. The functions of this framework do not process or affect the *seed* in any way: they simply pass it around as an instance of an opaque datatype. User functions, on the other hand, can use the seed to maintain user's state, to accumulate parsing results, etc. A user can freely mix their own functions with those of the framework. On the other hand, the user may wish to instantiate a high-level parser: `SSAX:make-elem-parser` or `SSAX:make-parser`. In the latter case, the user must provide functions of specific signatures, which are called at predictable moments during the parsing: to handle character data, element data, or processing instructions (PI). The functions are always given the *seed*, among other parameters, and must return the new *seed*.

From a functional point of view, XML parsing is a combined pre-post-order traversal of a "tree" that is the XML document itself. This down-and-up traversal tells the user about an element when its start tag is encountered. The user is notified about the element once more, after all element's children have been handled. The process of XML parsing therefore is a fold over the raw XML document. Unlike a fold over trees defined in [1], the parser is necessarily single-threaded – obviously as elements in a text XML document are laid down sequentially. The parser therefore is a tree fold that has been transformed to accept an accumulating parameter [1,2].

Formally, the denotational semantics of the parser can be expressed as

```
parser:: (Start-tag -> Seed -> Seed) ->
  (Start-tag -> Seed -> Seed -> Seed) ->
  (Char-Data -> Seed -> Seed) ->
  XML-text-fragment -> Seed -> Seed
parser fdown fup fchar "<elem attrs> content </elem>" seed
 = fup "<elem attrs>" seed
(parser fdown fup fchar "content" (fdown "<elem attrs>" seed))

parser fdown fup fchar "char-data content" seed
```

```
 = parser fdown fup fchar "content" (fchar "char-data" seed)

parser fdown fup fchar "elem-content content" seed
 = parser fdown fup fchar "content" (
parser fdown fup fchar "elem-content" seed)
```

Compare the last two equations with the left fold

```
fold-left kons elem:list seed = fold-left kons list (kons elem seed)
```

The real parser created by `SSAX:make-parser` is slightly more complicated, to account for processing instructions, entity references, namespaces, processing of document type declaration, etc.

The XML standard document referred to in this module is `http://www.w3.org/TR/1998/REC-xml-19980210.html`

The present file also defines a procedure that parses the text of an XML document or of a separate element into SXML, an S-expression-based model of an XML Information Set. SXML is also an Abstract Syntax Tree of an XML document. SXML is similar but not identical to DOM; SXML is particularly suitable for Scheme-based XML/HTML authoring, SXPath queries, and tree transformations. See SXML.html for more details. SXML is a term implementation of evaluation of the XML document [3]. The other implementation is context-passing.

The present frameworks fully supports the XML Namespaces Recommendation: `http://www.w3.org/TR/REC-xml-names/`.

Other links:

[1] Jeremy Gibbons, Geraint Jones, "The Under-appreciated Unfold," Proc. ICFP'98, 1998, pp. 273-279.

[2] Richard S. Bird, The promotion and accumulation strategies in transformational programming, ACM Trans. Progr. Lang. Systems, 6(4):487-504, October 1984.

[3] Ralf Hinze, "Deriving Backtracking Monad Transformers," Functional Pearl. Proc ICFP'00, pp. 186-197.

7.21.3.3 Usage

`current-ssax-error-port` [Scheme Procedure]

`with-ssax-error-to-port` *port thunk* [Scheme Procedure]

`xml-token? _` [Scheme Procedure]

```
-- Scheme Procedure: pair? x
    Return '#t' if X is a pair; otherwise return '#f'.
```

`xml-token-kind` *token* [Scheme Syntax]

`xml-token-head` *token* [Scheme Syntax]

`make-empty-attlist` [Scheme Procedure]

`attlist-add` *attlist name-value* [Scheme Procedure]

`attlist-null?` *x* [Scheme Procedure]
Return `#t` if *x* is the empty list, else `#f`.

`attlist-remove-top` *attlist* [Scheme Procedure]

`attlist->alist` *attlist* [Scheme Procedure]

`attlist-fold` *kons knil lis1* [Scheme Procedure]

`define-parsed-entity!` *entity str* [Scheme Procedure]
Define a new parsed entity. *entity* should be a symbol.

Instances of *&entity;* in XML text will be replaced with the string *str*, which will then be parsed.

`reset-parsed-entity-definitions!` [Scheme Procedure]
Restore the set of parsed entity definitions to its initial state.

`ssax:uri-string->symbol` *uri-str* [Scheme Procedure]

`ssax:skip-internal-dtd` *port* [Scheme Procedure]

`ssax:read-pi-body-as-string` *port* [Scheme Procedure]

`ssax:reverse-collect-str-drop-ws` *fragments* [Scheme Procedure]

`ssax:read-markup-token` *port* [Scheme Procedure]

`ssax:read-cdata-body` *port str-handler seed* [Scheme Procedure]

`ssax:read-char-ref` *port* [Scheme Procedure]

`ssax:read-attributes` *port entities* [Scheme Procedure]

`ssax:complete-start-tag` *tag-head port elems entities namespaces* [Scheme Procedure]

`ssax:read-external-id` *port* [Scheme Procedure]

`ssax:read-char-data` *port expect-eof? str-handler seed* [Scheme Procedure]

`ssax:xml->sxml` *port namespace-prefix-assig* [Scheme Procedure]

`ssax:make-parser` *. kw-val-pairs* [Scheme Syntax]

`ssax:make-pi-parser` *orig-handlers* [Scheme Syntax]

`ssax:make-elem-parser` *my-new-level-seed my-finish-element my-char-data-handler my-pi-handlers* [Scheme Syntax]

7.21.4 Transforming SXML

7.21.4.1 Overview

SXML expression tree transformers

Pre-Post-order traversal of a tree and creation of a new tree

```
pre-post-order:: <tree> x <bindings> -> <new-tree>
```

where

```
 <bindings> ::= (<binding> ...)
 <binding> ::= (<trigger-symbol> *preorder* . <handler>) |
               (<trigger-symbol> *macro* . <handler>) |
(<trigger-symbol> <new-bindings> . <handler>) |
(<trigger-symbol> . <handler>)
 <trigger-symbol> ::= XMLname | *text* | *default*
 <handler> :: <trigger-symbol> x [<tree>] -> <new-tree>
```

The pre-post-order function visits the nodes and nodelists pre-post-order (depth-first). For each `<Node>` of the form (*name* `<Node> ...`), it looks up an association with the given *name* among its `<`*bindings*`>`. If failed, `pre-post-order` tries to locate a `*default*` binding. It's an error if the latter attempt fails as well. Having found a binding, the `pre-post-order` function first checks to see if the binding is of the form

```
(<trigger-symbol> *preorder* . <handler>)
```

If it is, the handler is 'applied' to the current node. Otherwise, the pre-post-order function first calls itself recursively for each child of the current node, with `<`*new-bindings*`>` prepended to the `<`*bindings*`>` in effect. The result of these calls is passed to the `<`*handler*`>` (along with the head of the current `<`*Node*`>`). To be more precise, the handler is _applied_ to the head of the current node and its processed children. The result of the handler, which should also be a `<tree>`, replaces the current `<`*Node*`>`. If the current `<`*Node*`>` is a text string or other atom, a special binding with a symbol `*text*` is looked up.

A binding can also be of a form

```
(<trigger-symbol> *macro* . <handler>)
```

This is equivalent to `*preorder*` described above. However, the result is re-processed again, with the current stylesheet.

7.21.4.2 Usage

`SRV:send-reply` . *fragments* [Scheme Procedure]

Output the *fragments* to the current output port.

The fragments are a list of strings, characters, numbers, thunks, `#f`, `#t` – and other fragments. The function traverses the tree depth-first, writes out strings and characters, executes thunks, and ignores `#f` and `'()`. The function returns `#t` if anything was written at all; otherwise the result is `#f` If `#t` occurs among the fragments, it is not written out but causes the result of `SRV:send-reply` to be `#t`.

`foldts` *fdown fup fhere seed tree* [Scheme Procedure]

`post-order` *tree bindings* [Scheme Procedure]

`pre-post-order` *tree bindings* [Scheme Procedure]

`replace-range` *beg-pred end-pred forest* [Scheme Procedure]

7.21.5 SXML Tree Fold

7.21.5.1 Overview

`(sxml fold)` defines a number of variants of the *fold* algorithm for use in transforming SXML trees. Additionally it defines the layout operator, `fold-layout`, which might be described as a context-passing variant of SSAX's `pre-post-order`.

7.21.5.2 Usage

`foldt` *fup fhere tree* [Scheme Procedure]

The standard multithreaded tree fold.

fup is of type [a] -> a. *fhere* is of type object -> a.

`foldts` *fdown fup fhere seed tree* [Scheme Procedure]

The single-threaded tree fold originally defined in SSAX. See Section 7.21.3 [SSAX], page 751, for more information.

`foldts*` *fdown fup fhere seed tree* [Scheme Procedure]

A variant of `foldts` that allows pre-order tree rewrites. Originally defined in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

`fold-values` *proc list . seeds* [Scheme Procedure]

A variant of `fold` that allows multi-valued seeds. Note that the order of the arguments differs from that of `fold`. See Section 7.5.3.5 [SRFI-1 Fold and Map], page 588.

`foldts*-values` *fdown fup fhere tree . seeds* [Scheme Procedure]

A variant of `foldts*` that allows multi-valued seeds. Originally defined in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

`fold-layout` *tree bindings params layout stylesheet* [Scheme Procedure]

A traversal combinator in the spirit of `pre-post-order`. See Section 7.21.4 [Transforming SXML], page 754.

`fold-layout` was originally presented in Andy Wingo's 2007 paper, *Applications of fold to XML transformation*.

```
bindings := (<binding>...)
binding  := (<tag> <handler-pair>...)
          | (*default* . <post-handler>)
          | (*text* . <text-handler>)
tag      := <symbol>
handler-pair := (pre-layout . <pre-layout-handler>)
          | (post . <post-handler>)
          | (bindings . <bindings>)
          | (pre . <pre-handler>)
          | (macro . <macro-handler>)
```

pre-layout-handler

A function of three arguments:

kids the kids of the current node, before traversal

params the params of the current node

layout the layout coming into this node

pre-layout-handler is expected to use this information to return a layout to pass to the kids. The default implementation returns the layout given in the arguments.

post-handler

A function of five arguments:

tag the current tag being processed

params the params of the current node

layout the layout coming into the current node, before any kids were processed

klayout the layout after processing all of the children

kids the already-processed child nodes

post-handler should return two values, the layout to pass to the next node and the final tree.

text-handler

text-handler is a function of three arguments:

text the string

params the current params

layout the current layout

text-handler should return two values, the layout to pass to the next node and the value to which the string should transform.

7.21.6 SXPath

7.21.6.1 Overview

SXPath: SXML Query Language

SXPath is a query language for SXML, an instance of XML Information set (Infoset) in the form of s-expressions. See `(sxml ssax)` for the definition of SXML and more details. SXPath is also a translation into Scheme of an XML Path Language, XPath (`http://www.w3.org/TR/xpath`). XPath and SXPath describe means of selecting a set of Infoset's items or their properties.

To facilitate queries, XPath maps the XML Infoset into an explicit tree, and introduces important notions of a location path and a current, context node. A location path denotes a selection of a set of nodes relative to a context node. Any XPath tree has a distinguished, root node – which serves as the context node for absolute location paths. Location path is recursively defined as a location step joined with a location path. A location step is a simple query of the database relative to a context node. A step may include expressions that further filter the selected set. Each node in the resulting set is used as a context node for the adjoining location path. The result of the step is a union of the sets returned by the latter location paths.

The SXML representation of the XML Infoset (see SSAX.scm) is rather suitable for querying as it is. Bowing to the XPath specification, we will refer to SXML information items as 'Nodes':

```
<Node> ::= <Element> | <attributes-coll> | <attrib>
   | "text string" | <PI>
```

This production can also be described as

```
<Node> ::= (name . <Nodeset>) | "text string"
```

An (ordered) set of nodes is just a list of the constituent nodes:

```
<Nodeset> ::= (<Node> ...)
```

Nodesets, and Nodes other than text strings are both lists. A <Nodeset> however is either an empty list, or a list whose head is not a symbol. A symbol at the head of a node is either an XML name (in which case it's a tag of an XML element), or an administrative name such as '@'. This uniform list representation makes processing rather simple and elegant, while avoiding confusion. The multi-branch tree structure formed by the mutually-recursive datatypes <Node> and <Nodeset> lends itself well to processing by functional languages.

A location path is in fact a composite query over an XPath tree or its branch. A singe step is a combination of a projection, selection or a transitive closure. Multiple steps are combined via join and union operations. This insight allows us to *elegantly* implement XPath as a sequence of projection and filtering primitives – converters – joined by *combinators*. Each converter takes a node and returns a nodeset which is the result of the corresponding query relative to that node. A converter can also be called on a set of nodes. In that case it returns a union of the corresponding queries over each node in the set. The union is easily implemented as a list append operation as all nodes in a SXML tree are considered distinct, by XPath conventions. We also preserve the order of the members in the union. Query combinators are high-order functions: they take converter(s) (which is a Node|Nodeset -> Nodeset function) and compose or otherwise combine them. We will be concerned with only relative location paths [XPath]: an absolute location path is a relative path applied to the root node.

Similarly to XPath, SXPath defines full and abbreviated notations for location paths. In both cases, the abbreviated notation can be mechanically expanded into the full form by simple rewriting rules. In the case of SXPath the corresponding rules are given in the documentation of the `sxpath` procedure. See [SXPath procedure documentation], page 762.

The regression test suite at the end of the file `SXPATH-old.scm` shows a representative sample of SXPaths in both notations, juxtaposed with the corresponding XPath expressions. Most of the samples are borrowed literally from the XPath specification.

Much of the following material is taken from the SXPath sources by Oleg Kiselyov et al.

7.21.6.2 Basic Converters and Applicators

A converter is a function mapping a nodeset (or a single node) to another nodeset. Its type can be represented like this:

```
type Converter = Node|Nodeset -> Nodeset
```

A converter can also play the role of a predicate: in that case, if a converter, applied to a node or a nodeset, yields a non-empty nodeset, the converter-predicate is deemed satisfied. Likewise, an empty nodeset is equivalent to `#f` in denoting failure.

`nodeset?` *x* [Scheme Procedure]

Return `#t` if *x* is a nodeset.

`node-typeof?` *crit* [Scheme Procedure]

This function implements a 'Node test' as defined in Sec. 2.3 of the XPath document. A node test is one of the components of a location step. It is also a converter-predicate in SXPath.

The function `node-typeof?` takes a type criterion and returns a function, which, when applied to a node, will tell if the node satisfies the test.

The criterion *crit* is a symbol, one of the following:

`id`	tests if the node has the right name (id)
`@`	tests if the node is an <attributes-coll>
`*`	tests if the node is an <Element>
`*text*`	tests if the node is a text node
`*PI*`	tests if the node is a PI (processing instruction) node
`*any*`	`#t` for any type of node

`node-eq?` *other* [Scheme Procedure]

A curried equivalence converter predicate that takes a node *other* and returns a function that takes another node. The two nodes are compared using `eq?`.

`node-equal?` *other* [Scheme Procedure]

A curried equivalence converter predicate that takes a node *other* and returns a function that takes another node. The two nodes are compared using `equal?`.

`node-pos` *n* [Scheme Procedure]

Select the *n*'th element of a nodeset and return as a singular nodeset. If the *n*'th element does not exist, return an empty nodeset. If *n* is a negative number the node is picked from the tail of the list.

```
((node-pos 1) nodeset)  ; return the the head of the nodeset (if exists)
((node-pos 2) nodeset)  ; return the node after that (if exists)
((node-pos -1) nodeset) ; selects the last node of a non-empty nodeset
((node-pos -2) nodeset) ; selects the last but one node, if exists.
```

`filter` *pred?* [Scheme Procedure]

A filter applicator, which introduces a filtering context. The argument converter *pred?* is considered a predicate, with either `#f` or `nil` meaning failure.

`take-until` *pred?* [Scheme Procedure]

```
take-until:: Converter -> Converter, or
take-until:: Pred -> Node|Nodeset -> Nodeset
```

Given a converter-predicate *pred?* and a nodeset, apply the predicate to each element of the nodeset, until the predicate yields anything but `#f` or `nil`. Return the elements of the input nodeset that have been processed until that moment (that is, which fail the predicate).

`take-until` is a variation of the `filter` above: `take-until` passes elements of an ordered input set up to (but not including) the first element that satisfies the predicate. The nodeset returned by `((take-until (not pred)) nset)` is a subset – to be more precise, a prefix – of the nodeset returned by `((filter pred) nset)`.

`take-after` *pred?* [Scheme Procedure]

```
take-after:: Converter -> Converter, or
take-after:: Pred -> Node|Nodeset -> Nodeset
```

Given a converter-predicate *pred?* and a nodeset, apply the predicate to each element of the nodeset, until the predicate yields anything but `#f` or `nil`. Return the elements of the input nodeset that have not been processed: that is, return the elements of the input nodeset that follow the first element that satisfied the predicate.

`take-after` along with `take-until` partition an input nodeset into three parts: the first element that satisfies a predicate, all preceding elements and all following elements.

`map-union` *proc lst* [Scheme Procedure]

Apply *proc* to each element of *lst* and return the list of results. If *proc* returns a nodeset, splice it into the result

From another point of view, `map-union` is a function `Converter->Converter`, which places an argument-converter in a joining context.

`node-reverse` *node-or-nodeset* [Scheme Procedure]

```
node-reverse :: Converter, or
node-reverse:: Node|Nodeset -> Nodeset
```

Reverses the order of nodes in the nodeset. This basic converter is needed to implement a reverse document order (see the XPath Recommendation).

`node-trace` *title* [Scheme Procedure]

```
node-trace:: String -> Converter
```

`(node-trace title)` is an identity converter. In addition it prints out the node or nodeset it is applied to, prefixed with the *title*. This converter is very useful for debugging.

7.21.6.3 Converter Combinators

Combinators are higher-order functions that transmogrify a converter or glue a sequence of converters into a single, non-trivial converter. The goal is to arrive at converters that correspond to XPath location paths.

From a different point of view, a combinator is a fixed, named *pattern* of applying converters. Given below is a complete set of such patterns that together implement XPath location path specification. As it turns out, all these combinators can be built from a small number of basic blocks: regular functional composition, `map-union` and `filter` applicators, and the nodeset union.

`select-kids` *test-pred?* [Scheme Procedure]

`select-kids` takes a converter (or a predicate) as an argument and returns another converter. The resulting converter applied to a nodeset returns an ordered subset of its children that satisfy the predicate *test-pred?*.

`node-self` *pred?* [Scheme Procedure]

Similar to `select-kids` except that the predicate *pred?* is applied to the node itself rather than to its children. The resulting nodeset will contain either one component, or will be empty if the node failed the predicate.

`node-join` . *selectors* [Scheme Procedure]

```
node-join:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-join:: [Converter] -> Converter
```

Join the sequence of location steps or paths as described above.

`node-reduce` . *converters* [Scheme Procedure]

```
node-reduce:: [LocPath] -> Node|Nodeset -> Nodeset, or
node-reduce:: [Converter] -> Converter
```

A regular functional composition of converters. From a different point of view, `((apply node-reduce converters) nodeset)` is equivalent to `(foldl apply nodeset converters)`, i.e., folding, or reducing, a list of converters with the nodeset as a seed.

`node-or` . *converters* [Scheme Procedure]

```
node-or:: [Converter] -> Converter
```

This combinator applies all converters to a given node and produces the union of their results. This combinator corresponds to a union (`|` operation) for XPath location paths.

`node-closure` *test-pred?* [Scheme Procedure]

```
node-closure:: Converter -> Converter
```

Select all *descendants* of a node that satisfy a converter-predicate *test-pred?*. This combinator is similar to `select-kids` but applies to grand... children as well. This combinator implements the `descendant::` XPath axis. Conceptually, this combinator can be expressed as

```
(define (node-closure f)
  (node-or
    (select-kids f)
    (node-reduce (select-kids (node-typeof? '*)) (node-closure f))))
```

This definition, as written, looks somewhat like a fixpoint, and it will run forever. It is obvious however that sooner or later `(select-kids (node-typeof? '*))` will return an empty nodeset. At this point further iterations will no longer affect the result and can be stopped.

`node-parent` *rootnode* [Scheme Procedure]

```
node-parent:: RootNode -> Converter
```

`(node-parent rootnode)` yields a converter that returns a parent of a node it is applied to. If applied to a nodeset, it returns the list of parents of nodes in the nodeset. The *rootnode* does not have to be the root node of the whole SXML tree – it may be a root node of a branch of interest.

Given the notation of Philip Wadler's paper on semantics of XSLT,

```
parent(x) = { y | y=subnode*(root), x=subnode(y) }
```

Therefore, `node-parent` is not the fundamental converter: it can be expressed through the existing ones. Yet `node-parent` is a rather convenient converter. It corresponds to a `parent::` axis of SXPath. Note that the `parent::` axis can be used with an attribute node as well.

`sxpath` *path* [Scheme Procedure]

Evaluate an abbreviated SXPath.

```
sxpath:: AbbrPath -> Converter, or
sxpath:: AbbrPath -> Node|Nodeset -> Nodeset
```

path is a list. It is translated to the full SXPath according to the following rewriting rules:

```
(sxpath '())
⇒ (node-join)

(sxpath '(path-component ...))
⇒ (node-join (sxpath1 path-component) (sxpath '(...)))

(sxpath1 '//)
⇒ (node-or
   (node-self (node-typeof? '*any*))
   (node-closure (node-typeof? '*any*)))

(sxpath1 '(equal? x))
⇒ (select-kids (node-equal? x))

(sxpath1 '(eq? x))
⇒ (select-kids (node-eq? x))

(sxpath1 ?symbol)
⇒ (select-kids (node-typeof? ?symbol)

(sxpath1 procedure)
⇒ procedure

(sxpath1 '(?symbol ...))
⇒ (sxpath1 '((?symbol) ...))

(sxpath1 '(path reducer ...))
⇒ (node-reduce (sxpath path) (sxpathr reducer) ...)

(sxpathr number)
⇒ (node-pos number)

(sxpathr path-filter)
⇒ (filter (sxpath path-filter))
```

7.21.7 (sxml ssax input-parse)

7.21.7.1 Overview

A simple lexer.

The procedures in this module surprisingly often suffice to parse an input stream. They either skip, or build and return tokens, according to inclusion or delimiting semantics. The list of characters to expect, include, or to break at may vary from one invocation of a function to another. This allows the functions to easily parse even context-sensitive languages.

EOF is generally frowned on, and thrown up upon if encountered. Exceptions are mentioned specifically. The list of expected characters (characters to skip until, or break-characters) may include an EOF "character", which is to be coded as the symbol, `*eof*`.

The input stream to parse is specified as a *port*, which is usually the last (and optional) argument. It defaults to the current input port if omitted.

If the parser encounters an error, it will throw an exception to the key `parser-error`. The arguments will be of the form (*port message specialising-msg**).

The first argument is a port, which typically points to the offending character or its neighborhood. You can then use `port-column` and `port-line` to query the current position. *message* is the description of the error. Other arguments supply more details about the problem.

7.21.7.2 Usage

`peek-next-char` [*port*] [Scheme Procedure]

`assert-curr-char` *expected-chars comment* [*port*] [Scheme Procedure]

`skip-until` *arg* [*port*] [Scheme Procedure]

`skip-while` *skip-chars* [*port*] [Scheme Procedure]

`next-token` *prefix-skipped-chars break-chars* [*comment*] [*port*] [Scheme Procedure]

`next-token-of` *incl-list/pred* [*port*] [Scheme Procedure]

`read-text-line` [*port*] [Scheme Procedure]

`read-string` *n* [*port*] [Scheme Procedure]

`find-string-from-port? _ _ . _` [Scheme Procedure]

Looks for *str* in <*input-port*>, optionally within the first *max-no-char* characters.

7.21.8 (sxml apply-templates)

7.21.8.1 Overview

Pre-order traversal of a tree and creation of a new tree:

```
apply-templates:: tree x <templates> -> <new-tree>
```

where

```
<templates> ::= (<template> ...)
<template>  ::= (<node-test> <node-test> ... <node-test> . <handler>)
```

```
<node-test> ::= an argument to node-typeof? above
<handler>   ::= <tree> -> <new-tree>
```

This procedure does a *normal*, pre-order traversal of an SXML tree. It walks the tree, checking at each node against the list of matching templates.

If the match is found (which must be unique, i.e., unambiguous), the corresponding handler is invoked and given the current node as an argument. The result from the handler, which must be a `<tree>`, takes place of the current node in the resulting tree. The name of the function is not accidental: it resembles rather closely an `apply-templates` function of XSLT.

7.21.8.2 Usage

`apply-templates` *tree templates* [Scheme Procedure]

7.22 Texinfo Processing

7.22.1 (texinfo)

7.22.1.1 Overview

Texinfo processing in scheme

This module parses texinfo into SXML. TeX will always be the processor of choice for print output, of course. However, although `makeinfo` works well for info, its output in other formats is not very customizable, and the program is not extensible as a whole. This module aims to provide an extensible framework for texinfo processing that integrates texinfo into the constellation of SXML processing tools.

Notes on the SXML vocabulary

Consider the following texinfo fragment:

```
@deffn Primitive set-car! pair value
This function...
@end deffn
```

Logically, the category (Primitive), name (set-car!), and arguments (pair value) are "attributes" of the deffn, with the description as the content. However, texinfo allows for @-commands within the arguments to an environment, like `@deffn`, which means that texinfo "attributes" are PCDATA. XML attributes, on the other hand, are CDATA. For this reason, "attributes" of texinfo @-commands are called "arguments", and are grouped under the special element, '%'.

Because '%' is not a valid NCName, stexinfo is a superset of SXML. In the interests of interoperability, this module provides a conversion function to replace the '%' with 'texinfo-arguments'.

7.22.1.2 Usage

`call-with-file-and-dir` *filename proc* [Function]

Call the one-argument procedure *proc* with an input port that reads from *filename*. During the dynamic extent of *proc*'s execution, the current directory will be `(dirname`

filename). This is useful for parsing documents that can include files by relative path name.

`texi-command-specs` [Variable]

`texi-command-depth` *command max-depth* [Function]
Given the texinfo command *command*, return its nesting level, or `#f` if it nests too deep for *max-depth*.

Examples:

```
(texi-command-depth 'chapter 4)        ⇒ 1
(texi-command-depth 'top 4)            ⇒ 0
(texi-command-depth 'subsection 4)     ⇒ 3
(texi-command-depth 'appendixsubsec 4) ⇒ 3
(texi-command-depth 'subsection 2)     ⇒ #f
```

`texi-fragment->stexi` *string-or-port* [Function]
Parse the texinfo commands in *string-or-port*, and return the resultant stexi tree. The head of the tree will be the special command, `*fragment*`.

`texi->stexi` *port* [Function]
Read a full texinfo document from *port* and return the parsed stexi tree. The parsing will start at the `@settitle` and end at `@bye` or EOF.

`stexi->sxml` *tree* [Function]
Transform the stexi tree *tree* into sxml. This involves replacing the `%` element that keeps the texinfo arguments with an element for each argument.

FIXME: right now it just changes % to `texinfo-arguments` – that doesn't hang with the idea of making a dtd at some point

7.22.2 (texinfo docbook)

7.22.2.1 Overview

This module exports procedures for transforming a limited subset of the SXML representation of docbook into stexi. It is not complete by any means. The intention is to gather a number of routines and stylesheets so that external modules can parse specific subsets of docbook, for example that set generated by certain tools.

7.22.2.2 Usage

`*sdocbook->stexi-rules*` [Variable]

`*sdocbook-block-commands*` [Variable]

`sdocbook-flatten` *sdocbook* [Function]
"Flatten" a fragment of sdocbook so that block elements do not nest inside each other.

Docbook is a nested format, where e.g. a `refsect2` normally appears inside a `refsect1`. Logical divisions in the document are represented via the tree topology; a `refsect2` element *contains* all of the elements in its section.

On the contrary, texinfo is a flat format, in which sections are marked off by standalone section headers like `@subsection`, and block elements do not nest inside each other.

This function takes a nested sdocbook fragment *sdocbook* and flattens all of the sections, such that e.g.

```
(refsect1 (refsect2 (para "Hello")))
```

becomes

```
((refsect1) (refsect2) (para "Hello"))
```

Oftentimes (always?) sectioning elements have `<title>` as their first element child; users interested in processing the `refsect*` elements into proper sectioning elements like `chapter` might be interested in `replace-titles` and `filter-empty-elements`. See [replace-titles], page 766, and [filter-empty-elements], page 766.

Returns a nodeset; that is to say, an untagged list of stexi elements. See Section 7.21.6 [SXPath], page 757, for the definition of a nodeset.

`filter-empty-elements` *sdocbook* [Function]

Filters out empty elements in an sdocbook nodeset. Mostly useful after running `sdocbook-flatten`.

`replace-titles` *sdocbook-fragment* [Function]

Iterate over the sdocbook nodeset *sdocbook-fragment*, transforming contiguous `refsect` and `title` elements into the appropriate texinfo sectioning command. Most useful after having run `sdocbook-flatten`.

For example:

```
(replace-titles '((refsect1) (title "Foo") (para "Bar.")))
   ⇒ '((chapter "Foo") (para "Bar."))
```

7.22.3 (texinfo html)

7.22.3.1 Overview

This module implements transformation from `stexi` to HTML. Note that the output of `stexi->shtml` is actually SXML with the HTML vocabulary. This means that the output can be further processed, and that it must eventually be serialized by `sxml->xml`. See Section 7.21.2 [Reading and Writing XML], page 749.

References (i.e., the `@ref` family of commands) are resolved by a *ref-resolver*. See [texinfo html add-ref-resolver!], page 766.

7.22.3.2 Usage

`add-ref-resolver!` *proc* [Function]

Add *proc* to the head of the list of ref-resolvers. *proc* will be expected to take the name of a node and the name of a manual and return the URL of the referent, or `#f` to pass control to the next ref-resolver in the list.

The default ref-resolver will return the concatenation of the manual name, `#`, and the node name.

`stexi->shtml` *tree* [Function]
Transform the stexi *tree* into shtml, resolving references via ref-resolvers. See the module commentary for more details.

`urlify` *str* [Function]

7.22.4 (texinfo indexing)

7.22.4.1 Overview

Given a piece of stexi, return an index of a specified variety.

Note that currently, `stexi-extract-index` doesn't differentiate between different kinds of index entries. That's a bug ;)

7.22.4.2 Usage

`stexi-extract-index` *tree manual-name kind* [Function]
Given an stexi tree *tree*, index all of the entries of type *kind*. *kind* can be one of the predefined texinfo indices (`concept`, `variable`, `function`, `key`, `program`, `type`) or one of the special symbols `auto` or `all`. `auto` will scan the stext for a `(printindex)` statement, and `all` will generate an index from all entries, regardless of type.

The returned index is a list of pairs, the CAR of which is the entry (a string) and the CDR of which is a node name (a string).

7.22.5 (texinfo string-utils)

7.22.5.1 Overview

Module '`(texinfo string-utils)`' provides various string-related functions useful to Guile's texinfo support.

7.22.5.2 Usage

`escape-special-chars` *str special-chars escape-char* [Function]
Returns a copy of *str* with all given special characters preceded by the given *escape-char*.

special-chars can either be a single character, or a string consisting of all the special characters.

```
;; make a string regexp-safe...
 (escape-special-chars "***(Example String)***"
                       "[]()/*."
                       #\\)
=> "\\*\\*\\*\\(Example String\\)\\*\\*\\*"

;; also can escape a singe char...
 (escape-special-chars "richardt@vzavenue.net"
                       #\@
                       #\@)
=> "richardt@@vzavenue.net"
```

`transform-string` *str match? replace* [*start*] [*end*] [Function]

Uses *match?* against each character in *str*, and performs a replacement on each character for which matches are found.

match? may either be a function, a character, a string, or `#t`. If *match?* is a function, then it takes a single character as input, and should return '`#t`' for matches. *match?* is a character, it is compared to each string character using `char=?`. If *match?* is a string, then any character in that string will be considered a match. `#t` will cause every character to be a match.

If *replace* is a function, it is called with the matched character as an argument, and the returned value is sent to the output string via '`display`'. If *replace* is anything else, it is sent through the output string via '`display`'.

Note that the replacement for the matched characters does not need to be a single character. That is what differentiates this function from '`string-map`', and what makes it useful for applications such as converting '`#\&`' to '`"&"`' in web page text. Some other functions in this module are just wrappers around common uses of '`transform-string`'. Transformations not possible with this function should probably be done with regular expressions.

If *start* and *end* are given, they control which portion of the string undergoes transformation. The entire input string is still output, though. So, if *start* is '`5`', then the first five characters of *str* will still appear in the returned string.

```
; these two are equivalent...
 (transform-string str #\space #\-) ; change all spaces to -'s
 (transform-string str (lambda (c) (char=? #\space c)) #\-)
```

`expand-tabs` *str* [*tab-size*] [Function]

Returns a copy of *str* with all tabs expanded to spaces. *tab-size* defaults to 8.

Assuming tab size of 8, this is equivalent to:

```
 (transform-string str #\tab "        ")
```

`center-string` *str* [*width*] [*chr*] [*rchr*] [Function]

Returns a copy of *str* centered in a field of *width* characters. Any needed padding is done by character *chr*, which defaults to '`#\space`'. If *rchr* is provided, then the padding to the right will use it instead. See the examples below. left and *rchr* on the right. The default *width* is 80. The default *chr* and *rchr* is '`#\space`'. The string is never truncated.

```
 (center-string "Richard Todd" 24)
=> "      Richard Todd      "

 (center-string " Richard Todd " 24 #\=)
=> "===== Richard Todd ====="

 (center-string " Richard Todd " 24 #\< #\>)
=> "<<<<< Richard Todd >>>>>"
```

`left-justify-string` *str* [*width*] [*chr*] [Function]

`left-justify-string str [width chr]`. Returns a copy of *str* padded with *chr* such that it is left justified in a field of *width* characters. The default *width* is 80. Unlike '`string-pad`' from srfi-13, the string is never truncated.

`right-justify-string` *str* [*width*] [*chr*] [Function]

Returns a copy of *str* padded with *chr* such that it is right justified in a field of *width* characters. The default *width* is 80. The default *chr* is '`#\space`'. Unlike '`string-pad`' from srfi-13, the string is never truncated.

`collapse-repeated-chars` *str* [*chr*] [*num*] [Function]

Returns a copy of *str* with all repeated instances of *chr* collapsed down to at most *num* instances. The default value for *chr* is '`#\space`', and the default value for *num* is 1.

```
 (collapse-repeated-chars "H  e  l  l  o")
=> "H e l l o"
 (collapse-repeated-chars "H--e--l--l--o" #\-)
=> "H-e-l-l-o"
 (collapse-repeated-chars "H-e--l---l----o" #\- 2)
=> "H-e--l--l--o"
```

`make-text-wrapper` [*#:line-width*] [*#:expand-tabs?*] [*#:tab-width*] [*#:collapse-whitespace?*] [*#:subsequent-indent*] [*#:initial-indent*] [*#:break-long-words?*] [Function]

Returns a procedure that will split a string into lines according to the given parameters.

`#:line-width`

This is the target length used when deciding where to wrap lines. Default is 80.

`#:expand-tabs?`

Boolean describing whether tabs in the input should be expanded. Default is #t.

`#:tab-width`

If tabs are expanded, this will be the number of spaces to which they expand. Default is 8.

`#:collapse-whitespace?`

Boolean describing whether the whitespace inside the existing text should be removed or not. Default is #t.

If text is already well-formatted, and is just being wrapped to fit in a different width, then set this to '`#f`'. This way, many common text conventions (such as two spaces between sentences) can be preserved if in the original text. If the input text spacing cannot be trusted, then leave this setting at the default, and all repeated whitespace will be collapsed down to a single space.

`#:initial-indent`
: Defines a string that will be put in front of the first line of wrapped text. Default is the empty string, "".

`#:subsequent-indent`
: Defines a string that will be put in front of all lines of wrapped text, except the first one. Default is the empty string, "".

`#:break-long-words?`
: If a single word is too big to fit on a line, this setting tells the wrapper what to do. Defaults to #t, which will break up long words. When set to #f, the line will be allowed, even though it is longer than the defined `#:line-width`.

The return value is a procedure of one argument, the input string, which returns a list of strings, where each element of the list is one line.

`fill-string` *str . kwargs* [Function]
: Wraps the text given in string *str* according to the parameters provided in *kwargs*, or the default setting if they are not given. Returns a single string with the wrapped text. Valid keyword arguments are discussed in `make-text-wrapper`.

`string->wrapped-lines` *str . kwargs* [Function]
: `string->wrapped-lines str keywds ...`. Wraps the text given in string *str* according to the parameters provided in *keywds*, or the default setting if they are not given. Returns a list of strings representing the formatted lines. Valid keyword arguments are discussed in `make-text-wrapper`.

7.22.6 (texinfo plain-text)

7.22.6.1 Overview

Transformation from stexi to plain-text. Strives to re-create the output from `info`; comes pretty damn close.

7.22.6.2 Usage

`stexi->plain-text` *tree* [Function]
: Transform *tree* into plain text. Returns a string.

`*line-width*` [Scheme Variable]
: This fluid (see Section 6.13.11 [Fluids and Dynamic States], page 323) specifies the length of line for the purposes of line wrapping in the `stexi->plain-text` conversion.

7.22.7 (texinfo serialize)

7.22.7.1 Overview

Serialization of `stexi` to plain texinfo.

7.22.7.2 Usage

`stexi->texi` *tree* [Function]
: Serialize the stexi *tree* into plain texinfo.

7.22.8 (texinfo reflection)

7.22.8.1 Overview

Routines to generare `stexi` documentation for objects and modules.

Note that in this context, an *object* is just a value associated with a location. It has nothing to do with GOOPS.

7.22.8.2 Usage

`module-stexi-documentation` *sym-name* [*%docs-resolver*] [*#:docs-resolver*] [Function]

Return documentation for the module named *sym-name*. The documentation will be formatted as `stexi` (see Section 7.22.1 [texinfo], page 764).

`script-stexi-documentation` *scriptpath* [Function]

Return documentation for given script. The documentation will be taken from the script's commentary, and will be returned in the `stexi` format (see Section 7.22.1 [texinfo], page 764).

`object-stexi-documentation` _ [_] [*#:force*] [Function]

`package-stexi-standard-copying` *name version updated years copyright-holder permissions* [Function]

Create a standard texinfo `copying` section.

years is a list of years (as integers) in which the modules being documented were released. All other arguments are strings.

`package-stexi-standard-titlepage` *name version updated authors* [Function]

Create a standard GNU title page.

authors is a list of `(name . email)` pairs. All other arguments are strings.

Here is an example of the usage of this procedure:

```
(package-stexi-standard-titlepage
 "Foolib"
 "3.2"
 "26 September 2006"
 '(("Alyssa P Hacker" . "alyssa@example.com"))
 '(2004 2005 2006)
 "Free Software Foundation, Inc."
 "Standard GPL permissions blurb goes here")
```

`package-stexi-generic-menu` *name entries* [Function]

Create a menu from a generic alist of entries, the car of which should be the node name, and the cdr the description. As an exception, an entry of `#f` will produce a separator.

`package-stexi-standard-menu` *name modules module-descriptions extra-entries* [Function]

Create a standard top node and menu, suitable for processing by makeinfo.

`package-stexi-extended-menu` *name module-pairs script-pairs extra-entries* [Function]

Create an "extended" menu, like the standard menu but with a section for scripts.

`package-stexi-standard-prologue` *name filename category description copying titlepage menu* [Function]

Create a standard prologue, suitable for later serialization to texinfo and .info creation with makeinfo.

Returns a list of stexinfo forms suitable for passing to `package-stexi-documentation` as the prologue. See [texinfo reflection package-stexi-documentation], page 772, [texinfo reflection package-stexi-standard-titlepage], page 771, [texinfo reflection package-stexi-standard-copying], page 771, and [texinfo reflection package-stexi-standard-menu], page 771.

`package-stexi-documentation` *modules name filename prologue epilogue* [*#:module-stexi-documentation-args*] [*#:scripts*] [Function]

Create stexi documentation for a *package*, where a package is a set of modules that is released together.

modules is expected to be a list of module names, where a module name is a list of symbols. The stexi that is returned will be titled *name* and a texinfo filename of *filename*.

prologue and *epilogue* are lists of stexi forms that will be spliced into the output document before and after the generated modules documentation, respectively. See [texinfo reflection package-stexi-standard-prologue], page 772, to create a conventional GNU texinfo prologue.

module-stexi-documentation-args is an optional argument that, if given, will be added to the argument list when `module-texi-documentation` is called. For example, it might be useful to define a `#:docs-resolver` argument.

`package-stexi-documentation-for-include` *modules module-descriptions* [*#:module-stexi-documentation-args*] [Function]

Create stexi documentation for a *package*, where a package is a set of modules that is released together.

modules is expected to be a list of module names, where a module name is a list of symbols. Returns an stexinfo fragment.

Unlike `package-stexi-documentation`, this function simply produces a menu and the module documentations instead of producing a full texinfo document. This can be useful if you write part of your manual by hand, and just use `@include` to pull in the automatically generated parts.

module-stexi-documentation-args is an optional argument that, if given, will be added to the argument list when `module-texi-documentation` is called. For example, it might be useful to define a `#:docs-resolver` argument.

8 GOOPS

GOOPS is the object oriented extension to Guile. Its implementation is derived from STk-3.99.3 by Erick Gallesio and version 1.3 of Gregor Kiczales' *Tiny-Clos*. It is very close in spirit to CLOS, the Common Lisp Object System, but is adapted for the Scheme language.

GOOPS is a full object oriented system, with classes, objects, multiple inheritance, and generic functions with multi-method dispatch. Furthermore its implementation relies on a meta object protocol — which means that GOOPS's core operations are themselves defined as methods on relevant classes, and can be customised by overriding or redefining those methods.

To start using GOOPS you first need to import the `(oop goops)` module. You can do this at the Guile REPL by evaluating:

```
(use-modules (oop goops))
```

8.1 Copyright Notice

The material in this chapter is partly derived from the STk Reference Manual written by Erick Gallesio, whose copyright notice is as follows.

The material has been adapted for use in Guile, with the author's permission.

8.2 Class Definition

A new class is defined with the `define-class` syntax:

```
(define-class class (superclass ...)
   slot-description ...
   class-option ...)
```

class is the class being defined. The list of *superclass*es specifies which existing classes, if any, to inherit slots and properties from. *Slots* hold per-instance[1] data, for instances of that class — like "fields" or "member variables" in other object oriented systems. Each *slot-description* gives the name of a slot and optionally some "properties" of this slot; for example its initial value, the name of a function which will access its value, and so on. Class options, slot descriptions and inheritance are discussed more below.

`define-class` *name* (*super ...*) *slot-definition ... class-option ...* [syntax]

Define a class called *name* that inherits from *super*s, with direct slots defined by *slot-definition*s and *class-option*s. The newly created class is bound to the variable name *name* in the current environment.

[1] Usually — but see also the `#:allocation` slot option.

Each *slot-definition* is either a symbol that names the slot or a list,

```
(slot-name-symbol . slot-options)
```

where *slot-name-symbol* is a symbol and *slot-options* is a list with an even number of elements. The even-numbered elements of *slot-options* (counting from zero) are slot option keywords; the odd-numbered elements are the corresponding values for those keywords.

Each *class-option* is an option keyword and corresponding value.

As an example, let us define a type for representing a complex number in terms of two real numbers.[2] This can be done with the following class definition:

```
(define-class <my-complex> (<number>)
   r i)
```

This binds the variable `<my-complex>` to a new class whose instances will contain two slots. These slots are called `r` and `i` and will hold the real and imaginary parts of a complex number. Note that this class inherits from `<number>`, which is a predefined class.[3]

Slot options are described in the next section. The possible class options are as follows.

`#:metaclass` *metaclass* [class option]

The `#:metaclass` class option specifies the metaclass of the class being defined. *metaclass* must be a class that inherits from `<class>`. For the use of metaclasses, see Section 8.11.1 [Metaobjects and the Metaobject Protocol], page 797, and Section 8.11.2 [Metaclasses], page 799.

If the `#:metaclass` option is absent, GOOPS reuses or constructs a metaclass for the new class by calling `ensure-metaclass` (see Section 8.11.5 [ensure-metaclass], page 801).

`#:name` *name* [class option]

The `#:name` class option specifies the new class's name. This name is used to identify the class whenever related objects - the class itself, its instances and its subclasses - are printed.

If the `#:name` option is absent, GOOPS uses the first argument to `define-class` as the class name.

8.3 Instance Creation and Slot Access

An instance (or object) of a defined class can be created with `make`. `make` takes one mandatory parameter, which is the class of the instance to create, and a list of optional arguments that will be used to initialize the slots of the new instance. For instance the following form

```
(define c (make <my-complex>))
```

creates a new `<my-complex>` object and binds it to the Scheme variable `c`.

---

[2] Of course Guile already provides complex numbers, and `<complex>` is in fact a predefined class in GOOPS; but the definition here is still useful as an example.

[3] `<number>` is the direct superclass of the predefined class `<complex>`; `<complex>` is the superclass of `<real>`, and `<real>` is the superclass of `<integer>`.

`make` [generic]
`make` (*class* <*class*>) *initarg* ... [method]

Create and return a new instance of class *class*, initialized using *initarg* ....

In theory, *initarg* ... can have any structure that is understood by whatever methods get applied when the `initialize` generic function is applied to the newly allocated instance.

In practice, specialized `initialize` methods would normally call `(next-method)`, and so eventually the standard GOOPS `initialize` methods are applied. These methods expect *initargs* to be a list with an even number of elements, where even-numbered elements (counting from zero) are keywords and odd-numbered elements are the corresponding values.

GOOPS processes initialization argument keywords automatically for slots whose definition includes the `#:init-keyword` option (see Section 8.4 [init-keyword], page 775). Other keyword value pairs can only be processed by an `initialize` method that is specialized for the new instance's class. Any unprocessed keyword value pairs are ignored.

`make-instance` [generic]
`make-instance` (*class* <*class*>) *initarg* ... [method]

`make-instance` is an alias for `make`.

The slots of the new complex number can be accessed using `slot-ref` and `slot-set!`. `slot-set!` sets the value of an object slot and `slot-ref` retrieves it.

```
(slot-set! c 'r 10)
(slot-set! c 'i 3)
(slot-ref c 'r) ⇒ 10
(slot-ref c 'i) ⇒ 3
```

The `(oop goops describe)` module provides a `describe` function that is useful for seeing all the slots of an object; it prints the slots and their values to standard output.

```
(describe c)
⊣
#<<my-complex> 401d8638> is an instance of class <my-complex>
Slots are:
     r = 10
     i = 3
```

8.4 Slot Options

When specifying a slot (in a `(define-class ...)` form), various options can be specified in addition to the slot's name. Each option is specified by a keyword. The list of possible keywords is as follows.

`#:init-value` *init-value* [slot option]
`#:init-form` *init-form* [slot option]
`#:init-thunk` *init-thunk* [slot option]
`#:init-keyword` *init-keyword* [slot option]

These options provide various ways to specify how to initialize the slot's value at instance creation time.

init-value specifies a fixed initial slot value (shared across all new instances of the class).

init-thunk specifies a thunk that will provide a default value for the slot. The thunk is called when a new instance is created and should return the desired initial slot value.

init-form specifies a form that, when evaluated, will return an initial value for the slot. The form is evaluated each time that an instance of the class is created, in the lexical environment of the containing `define-class` expression.

init-keyword specifies a keyword that can be used to pass an initial slot value to `make` when creating a new instance.

Note that, since an `init-value` value is shared across all instances of a class, you should only use it when the initial value is an immutable value, like a constant. If you want to initialize a slot with a fresh, independently mutable value, you should use `init-thunk` or `init-form` instead. Consider the following example.

```
(define-class <chbouib> ()
  (hashtab #:init-value (make-hash-table)))
```

Here only one hash table is created and all instances of `<chbouib>` have their `hashtab` slot refer to it. In order to have each instance of `<chbouib>` refer to a new hash table, you should instead write:

```
(define-class <chbouib> ()
  (hashtab #:init-thunk make-hash-table))
```

or:

```
(define-class <chbouib> ()
  (hashtab #:init-form (make-hash-table)))
```

If more than one of these options is specified for the same slot, the order of precedence, highest first is

- `#:init-keyword`, if *init-keyword* is present in the options passed to `make`
- `#:init-thunk`, `#:init-form` or `#:init-value`.

If the slot definition contains more than one initialization option of the same precedence, the later ones are ignored. If a slot is not initialized at all, its value is unbound.

In general, slots that are shared between more than one instance are only initialized at new instance creation time if the slot value is unbound at that time. However, if the new instance creation specifies a valid init keyword and value for a shared slot, the slot is re-initialized regardless of its previous value.

Note, however, that the power of GOOPS' metaobject protocol means that everything written here may be customized or overridden for particular classes! The slot initializations described here are performed by the least specialized method of the generic function `initialize`, whose signature is

```
(define-method (initialize (object <object>) initargs) ...)
```

The initialization of instances of any given class can be customized by defining a `initialize` method that is specialized for that class, and the author of the specialized method may decide to call `next-method` - which will result in a call to the next less specialized `initialize` method - at any point within the specialized code, or maybe not at all. In general, therefore, the initialization mechanisms described here may be

modified or overridden by more specialized code, or may not be supported at all for particular classes.

`#:getter` *getter* [slot option]
`#:setter` *setter* [slot option]
`#:accessor` *accessor* [slot option]

Given an object *obj* with slots named `foo` and `bar`, it is always possible to read and write those slots by calling `slot-ref` and `slot-set!` with the relevant slot name; for example:

```
(slot-ref obj 'foo)
(slot-set! obj 'bar 25)
```

The `#:getter`, `#:setter` and `#:accessor` options, if present, tell GOOPS to create generic function and method definitions that can be used to get and set the slot value more conveniently. *getter* specifies a generic function to which GOOPS will add a method for getting the slot value. *setter* specifies a generic function to which GOOPS will add a method for setting the slot value. *accessor* specifies an accessor to which GOOPS will add methods for both getting and setting the slot value.

So if a class includes a slot definition like this:

```
(c #:getter get-count #:setter set-count #:accessor count)
```

GOOPS defines generic function methods such that the slot value can be referenced using either the getter or the accessor -

```
(let ((current-count (get-count obj))) ...)
(let ((current-count (count obj))) ...)
```

- and set using either the setter or the accessor -

```
(set-count obj (+ 1 current-count))
(set! (count obj) (+ 1 current-count))
```

Note that

- with an accessor, the slot value is set using the generalized `set!` syntax
- in practice, it is unusual for a slot to use all three of these options: read-only, write-only and read-write slots would typically use only `#:getter`, `#:setter` and `#:accessor` options respectively.

The binding of the specified names is done in the environment of the `define-class` expression. If the names are already bound (in that environment) to values that cannot be upgraded to generic functions, those values are overwritten when the `define-class` expression is evaluated. For more detail, see Section 8.11.9 [ensure-generic], page 807.

`#:allocation` *allocation* [slot option]

The `#:allocation` option tells GOOPS how to allocate storage for the slot. Possible values for *allocation* are

- `#:instance`

  Indicates that GOOPS should create separate storage for this slot in each new instance of the containing class (and its subclasses). This is the default.

- `#:class`

  Indicates that GOOPS should create storage for this slot that is shared by all instances of the containing class (and its subclasses). In other words, a slot in class *C* with allocation `#:class` is shared by all *instance*s for which `(is-a? instance c)`. This permits defining a kind of global variable which can be accessed only by (in)direct instances of the class which defines the slot.

- `#:each-subclass`

  Indicates that GOOPS should create storage for this slot that is shared by all *direct* instances of the containing class, and that whenever a subclass of the containing class is defined, GOOPS should create a new storage for the slot that is shared by all *direct* instances of the subclass. In other words, a slot with allocation `#:each-subclass` is shared by all instances with the same `class-of`.

- `#:virtual`

  Indicates that GOOPS should not allocate storage for this slot. The slot definition must also include the `#:slot-ref` and `#:slot-set!` options to specify how to reference and set the value for this slot. See the example below.

Slot allocation options are processed when defining a new class by the generic function `compute-get-n-set`, which is specialized by the class's metaclass. Hence new types of slot allocation can be implemented by defining a new metaclass and a method for `compute-get-n-set` that is specialized for the new metaclass. For an example of how to do this, see Section 8.11.6 [Customizing Class Definition], page 804.

`#:slot-ref` *getter* [slot option]

`#:slot-set!` *setter* [slot option]

The `#:slot-ref` and `#:slot-set!` options must be specified if the slot allocation is `#:virtual`, and are ignored otherwise.

getter should be a closure taking a single *instance* parameter that returns the current slot value. *setter* should be a closure taking two parameters - *instance* and *new-val* - that sets the slot value to *new-val*.

8.5 Illustrating Slot Description

To illustrate slot description, we can redefine the `<my-complex>` class seen before. A definition could be:

```
(define-class <my-complex> (<number>)
   (r #:init-value 0 #:getter get-r #:setter set-r! #:init-keyword #:r)
   (i #:init-value 0 #:getter get-i #:setter set-i! #:init-keyword #:i))
```

With this definition, the `r` and `i` slots are set to 0 by default, and can be initialised to other values by calling `make` with the `#:r` and `#:i` keywords. Also the generic functions `get-r`, `set-r!`, `get-i` and `set-i!` are automatically defined to read and write the slots.

```
(define c1 (make <my-complex> #:r 1 #:i 2))
(get-r c1) ⇒ 1
(set-r! c1 12)
(get-r c1) ⇒ 12
(define c2 (make <my-complex> #:r 2))
```

```
(get-r c2) ⇒ 2
(get-i c2) ⇒ 0
```

Accessors can both read and write a slot. So, another definition of the `<my-complex>` class, using the `#:accessor` option, could be:

```
(define-class <my-complex> (<number>)
   (r #:init-value 0 #:accessor real-part #:init-keyword #:r)
   (i #:init-value 0 #:accessor imag-part #:init-keyword #:i))
```

With this definition, the `r` slot can be read with:

```
(real-part c)
```

and set with:

```
(set! (real-part c) new-value)
```

Suppose now that we want to manipulate complex numbers with both rectangular and polar coordinates. One solution could be to have a definition of complex numbers which uses one particular representation and some conversion functions to pass from one representation to the other. A better solution is to use virtual slots, like this:

```
(define-class <my-complex> (<number>)
   ;; True slots use rectangular coordinates
   (r #:init-value 0 #:accessor real-part #:init-keyword #:r)
   (i #:init-value 0 #:accessor imag-part #:init-keyword #:i)
   ;; Virtual slots access do the conversion
   (m #:accessor magnitude #:init-keyword #:magn
      #:allocation #:virtual
      #:slot-ref (lambda (o)
                   (let ((r (slot-ref o 'r)) (i (slot-ref o 'i)))
                     (sqrt (+ (* r r) (* i i)))))
      #:slot-set! (lambda (o m)
                    (let ((a (slot-ref o 'a)))
                      (slot-set! o 'r (* m (cos a)))
                      (slot-set! o 'i (* m (sin a))))))
   (a #:accessor angle #:init-keyword #:angle
      #:allocation #:virtual
      #:slot-ref (lambda (o)
                   (atan (slot-ref o 'i) (slot-ref o 'r)))
      #:slot-set! (lambda(o a)
                    (let ((m (slot-ref o 'm)))
                      (slot-set! o 'r (* m (cos a)))
                      (slot-set! o 'i (* m (sin a)))))))
```

In this class definition, the magnitude `m` and angle `a` slots are virtual, and are calculated, when referenced, from the normal (i.e. `#:allocation #:instance`) slots `r` and `i`, by calling the function defined in the relevant `#:slot-ref` option. Correspondingly, writing `m` or `a` leads to calling the function defined in the `#:slot-set!` option. Thus the following expression

```
(slot-set! c 'a 3)
```

permits to set the angle of the `c` complex number.

```
(define c (make <my-complex> #:r 12 #:i 20))
(real-part c) ⇒ 12
(angle c) ⇒ 1.03037682652431
(slot-set! c 'i 10)
(set! (real-part c) 1)
(describe c)
⊣
#<<my-complex> 401e9b58> is an instance of class <my-complex>
Slots are:
     r = 1
     i = 10
     m = 10.0498756211209
     a = 1.47112767430373
```

Since initialization keywords have been defined for the four slots, we can now define the standard Scheme primitives `make-rectangular` and `make-polar`.

```
(define make-rectangular
   (lambda (x y) (make <my-complex> #:r x #:i y)))

(define make-polar
   (lambda (x y) (make <my-complex> #:magn x #:angle y)))
```

8.6 Methods and Generic Functions

A GOOPS method is like a Scheme procedure except that it is specialized for a particular set of argument classes, and will only be used when the actual arguments in a call match the classes in the method definition.

```
(define-method (+ (x <string>) (y <string>))
  (string-append x y))

(+ "abc" "de") ⇒ "abcde"
```

A method is not formally associated with any single class (as it is in many other object oriented languages), because a method can be specialized for a combination of several classes. If you've studied object orientation in non-Lispy languages, you may remember discussions such as whether a method to stretch a graphical image around a surface should be a method of the image class, with a surface as a parameter, or a method of the surface class, with an image as a parameter. In GOOPS you'd just write

```
(define-method (stretch (im <image>) (sf <surface>))
  ...)
```

and the question of which class the method is more associated with does not need answering.

There can simultaneously be several methods with the same name but different sets of specializing argument classes; for example:

```
(define-method (+ (x <string>) (y <string)) ...)
(define-method (+ (x <matrix>) (y <matrix>)) ...)
(define-method (+ (f <fish>) (b <bicycle>)) ...)
```

```
(define-method (+ (a <foo>) (b <bar>) (c <baz>)) ...)
```

A generic function is a container for the set of such methods that a program intends to use.

If you look at a program's source code, and see `(+ x y)` somewhere in it, conceptually what is happening is that the program at that point calls a generic function (in this case, the generic function bound to the identifier `+`). When that happens, Guile works out which of the generic function's methods is the most appropriate for the arguments that the function is being called with; then it evaluates the method's code with the arguments as formal parameters. This happens every time that a generic function call is evaluated — it isn't assumed that a given source code call will end up invoking the same method every time.

Defining an identifier as a generic function is done with the `define-generic` macro. Definition of a new method is done with the `define-method` macro. Note that `define-method` automatically does a `define-generic` if the identifier concerned is not already a generic function, so often an explicit `define-generic` call is not needed.

`define-generic` *symbol* [syntax]

Create a generic function with name *symbol* and bind it to the variable *symbol*. If *symbol* was previously bound to a Scheme procedure (or procedure-with-setter), the old procedure (and setter) is incorporated into the new generic function as its default procedure (and setter). Any other previous value, including an existing generic function, is discarded and replaced by a new, empty generic function.

`define-method` (*generic parameter ...*) *body ...* [syntax]

Define a method for the generic function or accessor *generic* with parameters *parameters* and body *body ...*.

generic is a generic function. If *generic* is a variable which is not yet bound to a generic function object, the expansion of `define-method` will include a call to `define-generic`. If *generic* is `(setter` *generic-with-setter*`)`, where *generic-with-setter* is a variable which is not yet bound to a generic-with-setter object, the expansion will include a call to `define-accessor`.

Each *parameter* must be either a symbol or a two-element list `(`*symbol class*`)`. The symbols refer to variables in the body forms that will be bound to the parameters supplied by the caller when calling this method. The *class*es, if present, specify the possible combinations of parameters to which this method can be applied.

body ... are the bodies of the method definition.

`define-method` expressions look a little like Scheme procedure definitions of the form

```
(define (name formals ...) . body)
```

The important difference is that each formal parameter, apart from the possible "rest" argument, can be qualified by a class name: *formal* becomes `(`*formal class*`)`. The meaning of this qualification is that the method being defined will only be applicable in a particular generic function invocation if the corresponding argument is an instance of *class* (or one of its subclasses). If more than one of the formal parameters is qualified in this way, then the method will only be applicable if each of the corresponding arguments is an instance of its respective qualifying class.

Note that unqualified formal parameters act as though they are qualified by the class `<top>`, which GOOPS uses to mean the superclass of all valid Scheme types, including both primitive types and GOOPS classes.

For example, if a generic function method is defined with *parameter*s `(s1 <square>)` and `(n <number>)`, that method is only applicable to invocations of its generic function that have two parameters where the first parameter is an instance of the `<square>` class and the second parameter is a number.

8.6.1 Accessors

An accessor is a generic function that can also be used with the generalized `set!` syntax (see Section 6.9.8 [Procedures with Setters], page 259). Guile will handle a call like

```
(set! (accessor args...) value)
```

by calling the most specialized method of `accessor` that matches the classes of `args` and `value`. `define-accessor` is used to bind an identifier to an accessor.

`define-accessor` *symbol* [syntax]

Create an accessor with name *symbol* and bind it to the variable *symbol*. If *symbol* was previously bound to a Scheme procedure (or procedure-with-setter), the old procedure (and setter) is incorporated into the new accessor as its default procedure (and setter). Any other previous value, including an existing generic function or accessor, is discarded and replaced by a new, empty accessor.

8.6.2 Extending Primitives

Many of Guile's primitive procedures can be extended by giving them a generic function definition that operates in conjunction with their normal C-coded implementation. When a primitive is extended in this way, it behaves like a generic function with the C-coded implementation as its default method.

This extension happens automatically if a method is defined (by a `define-method` call) for a variable whose current value is a primitive. But it can also be forced by calling `enable-primitive-generic!`.

`enable-primitive-generic!` *primitive* [primitive procedure]

Force the creation of a generic function definition for *primitive*.

Once the generic function definition for a primitive has been created, it can be retrieved using `primitive-generic-generic`.

`primitive-generic-generic` *primitive* [primitive procedure]

Return the generic function definition of *primitive*.

`primitive-generic-generic` raises an error if *primitive* is not a primitive with generic capability.

8.6.3 Merging Generics

GOOPS generic functions and accessors often have short, generic names. For example, if a vector package provides an accessor for the X coordinate of a vector, that accessor may just be called `x`. It doesn't need to be called, for example, `vector:x`, because GOOPS will work out, when it sees code like `(x obj)`, that the vector-specific method of `x` should be called if *obj* is a vector.

That raises the question, though, of what happens when different packages define a generic function with the same name. Suppose we work with a graphical package which

needs to use two independent vector packages for 2D and 3D vectors respectively. If both packages export `x`, what does the code using those packages end up with?

Section 6.20.3 [duplicate binding handlers], page 413, explains how this is resolved for conflicting bindings in general. For generics, there is a special duplicates handler, `merge-generics`, which tells the module system to merge generic functions with the same name. Here is an example:

```
(define-module (math 2D-vectors)
  #:use-module (oop goops)
  #:export (x y ...))

(define-module (math 3D-vectors)
  #:use-module (oop goops)
  #:export (x y z ...))

(define-module (my-module)
  #:use-module (oop goops)
  #:use-module (math 2D-vectors)
  #:use-module (math 3D-vectors)
  #:duplicates (merge-generics))
```

The generic function `x` in `(my-module)` will now incorporate all of the methods of `x` from both imported modules.

To be precise, there will now be three distinct generic functions named `x`: `x` in `(math 2D-vectors)`, `x` in `(math 3D-vectors)`, and `x` in `(my-module)`; and these functions share their methods in an interesting and dynamic way.

To explain, let's call the imported generic functions (in `(math 2D-vectors)` and `(math 3D-vectors)`) the *ancestors*, and the merged generic function (in `(my-module)`), the *descendant*. The general rule is that for any generic function G, the applicable methods are selected from the union of the methods of G's descendant functions, the methods of G itself and the methods of G's ancestor functions.

Thus ancestor functions effectively share methods with their descendants, and vice versa. In the example above, `x` in `(math 2D-vectors)` will share the methods of `x` in `(my-module)` and vice versa.[4] Sharing is dynamic, so adding another new method to a descendant implies adding it to that descendant's ancestors too.

8.6.4 Next-method

When you call a generic function, with a particular set of arguments, GOOPS builds a list of all the methods that are applicable to those arguments and orders them by how closely the method definitions match the actual argument types. It then calls the method at the top of this list. If the selected method's code wants to call on to the next method in this list, it can do so by using `next-method`.

```
(define-method (Test (a <integer>)) (cons 'integer (next-method)))
(define-method (Test (a <number>))  (cons 'number  (next-method)))
(define-method (Test a)             (list 'top))
```

[4] But note that `x` in `(math 2D-vectors)` doesn't share methods with `x` in `(math 3D-vectors)`, so modularity is still preserved.

With these definitions,

```
(Test 1)   ⇒ (integer number top)
(Test 1.0) ⇒ (number top)
(Test #t)  ⇒ (top)
```

`next-method` is always called as just `(next-method)`. The arguments for the next method call are always implicit, and always the same as for the original method call.

If you want to call on to a method with the same name but with a different set of arguments (as you might with overloaded methods in C++, for example), you do not use `next-method`, but instead simply write the new call as usual:

```
(define-method (Test (a <number>) min max)
  (if (and (>= a min) (<= a max))
      (display "Number is in range\n"))
  (Test a))

(Test 2 1 10)
⊣
Number is in range
⇒
(integer number top)
```

(You should be careful in this case that the `Test` calls do not lead to an infinite recursion, but this consideration is just the same as in Scheme code in general.)

8.6.5 Generic Function and Method Examples

Consider the following definitions:

```
(define-generic G)
(define-method (G (a <integer>) b) 'integer)
(define-method (G (a <real>) b) 'real)
(define-method (G a b) 'top)
```

The `define-generic` call defines *G* as a generic function. The three next lines define methods for *G*. Each method uses a sequence of *parameter specializers* that specify when the given method is applicable. A specializer permits to indicate the class a parameter must belong to (directly or indirectly) to be applicable. If no specializer is given, the system defaults it to `<top>`. Thus, the first method definition is equivalent to

```
(define-method (G (a <integer>) (b <top>)) 'integer)
```

Now, let's look at some possible calls to the generic function *G*:

```
(G 2 3)    ⇒ integer
(G 2 #t)   ⇒ integer
(G 1.2 'a) ⇒ real
(G #t #f)  ⇒ top
(G 1 2 3)  ⇒ error (since no method exists for 3 parameters)
```

The methods above use only one specializer per parameter list. But in general, any or all of a method's parameters may be specialized. Suppose we define now:

```
(define-method (G (a <integer>) (b <number>))  'integer-number)
(define-method (G (a <integer>) (b <real>))    'integer-real)
```

```
(define-method (G (a <integer>) (b <integer>)) 'integer-integer)
(define-method (G a (b <number>))              'top-number)
```

With these definitions:

```
(G 1 2)   ⇒ integer-integer
(G 1 1.0) ⇒ integer-real
(G 1 #t)  ⇒ integer
(G 'a 1)  ⇒ top-number
```

As a further example we shall continue to define operations on the `<my-complex>` class. Suppose that we want to use it to implement complex numbers completely. For instance a definition for the addition of two complex numbers could be

```
(define-method (new-+ (a <my-complex>) (b <my-complex>))
  (make-rectangular (+ (real-part a) (real-part b))
                    (+ (imag-part a) (imag-part b))))
```

To be sure that the `+` used in the method `new-+` is the standard addition we can do:

```
(define-generic new-+)

(let ((+ +))
  (define-method (new-+ (a <my-complex>) (b <my-complex>))
    (make-rectangular (+ (real-part a) (real-part b))
                      (+ (imag-part a) (imag-part b)))))
```

The `define-generic` ensures here that `new-+` will be defined in the global environment. Once this is done, we can add methods to the generic function `new-+` which make a closure on the `+` symbol. A complete writing of the `new-+` methods is shown in Figure 8.1.

```
(define-generic new-+)

(let ((+ +))

  (define-method (new-+ (a <real>) (b <real>)) (+ a b))

  (define-method (new-+ (a <real>) (b <my-complex>))
    (make-rectangular (+ a (real-part b)) (imag-part b)))

  (define-method (new-+ (a <my-complex>) (b <real>))
    (make-rectangular (+ (real-part a) b) (imag-part a)))

  (define-method (new-+ (a <my-complex>) (b <my-complex>))
    (make-rectangular (+ (real-part a) (real-part b))
                      (+ (imag-part a) (imag-part b))))

  (define-method (new-+ (a <number>))  a)

  (define-method (new-+) 0)

  (define-method (new-+ . args)
    (new-+ (car args)
      (apply new-+ (cdr args)))))

(set! + new-+)
```

Figure 8.1: Extending `+` to handle complex numbers

We take advantage here of the fact that generic function are not obliged to have a fixed number of parameters. The four first methods implement dyadic addition. The fifth method says that the addition of a single element is this element itself. The sixth method says that using the addition with no parameter always return 0 (as is also true for the primitive `+`). The last method takes an arbitrary number of parameters[5]. This method acts as a kind of `reduce`: it calls the dyadic addition on the *car* of the list and on the result of applying it on its rest. To finish, the `set!` permits to redefine the `+` symbol to our extended addition.

To conclude our implementation (integration?) of complex numbers, we could redefine standard Scheme predicates in the following manner:

```
(define-method (complex? c <my-complex>) #t)
(define-method (complex? c)           #f)

(define-method (number? n <number>) #t)
(define-method (number? n)          #f)
...
```

[5] The parameter list for a `define-method` follows the conventions used for Scheme procedures. In particular it can use the dot notation or a symbol to denote an arbitrary number of parameters

Standard primitives in which complex numbers are involved could also be redefined in the same manner.

8.6.6 Handling Invocation Errors

If a generic function is invoked with a combination of parameters for which there is no applicable method, GOOPS raises an error.

`no-method` [generic]
`no-method` (*gf* <*generic*>) *args* [method]
When an application invokes a generic function, and no methods at all have been defined for that generic function, GOOPS calls the `no-method` generic function. The default method calls `goops-error` with an appropriate message.

`no-applicable-method` [generic]
`no-applicable-method` (*gf* <*generic*>) *args* [method]
When an application applies a generic function to a set of arguments, and no methods have been defined for those argument types, GOOPS calls the `no-applicable-method` generic function. The default method calls `goops-error` with an appropriate message.

`no-next-method` [generic]
`no-next-method` (*gf* <*generic*>) *args* [method]
When a generic function method calls `(next-method)` to invoke the next less specialized method for that generic function, and no less specialized methods have been defined for the current generic function arguments, GOOPS calls the `no-next-method` generic function. The default method calls `goops-error` with an appropriate message.

8.7 Inheritance

Here are some class definitions to help illustrate inheritance:

```
(define-class A () a)
(define-class B () b)
(define-class C () c)
(define-class D (A B) d a)
(define-class E (A C) e c)
(define-class F (D E) f)
```

`A`, `B`, `C` have a null list of superclasses. In this case, the system will replace the null list by a list which only contains `<object>`, the root of all the classes defined by `define-class`. `D`, `E`, `F` use multiple inheritance: each class inherits from two previously defined classes. Those class definitions define a hierarchy which is shown in Figure 8.2. In this figure, the class `<top>` is also shown; this class is the superclass of all Scheme objects. In particular, `<top>` is the superclass of all standard Scheme types.

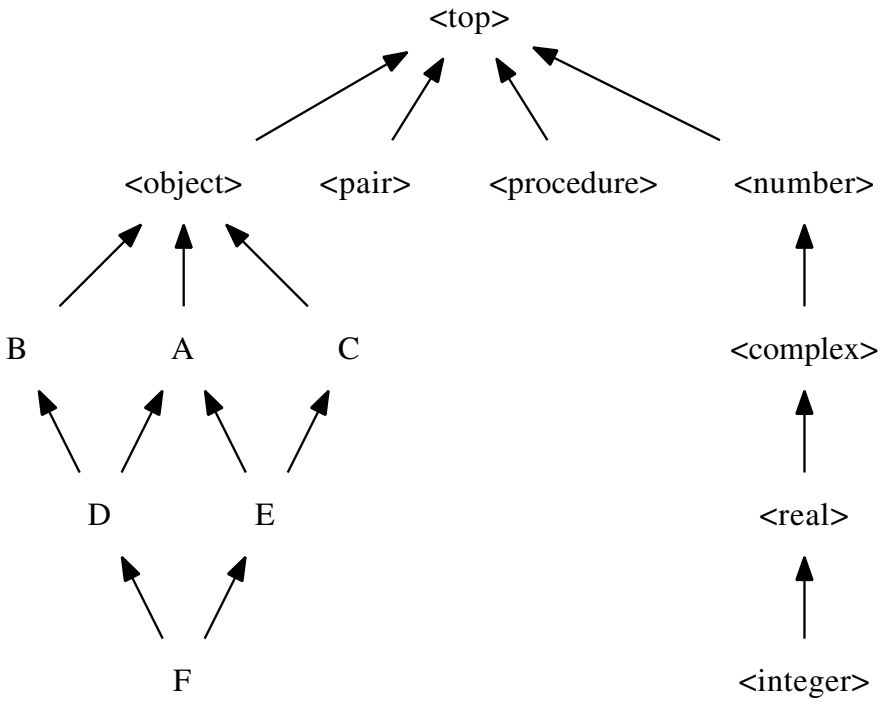

Figure 8.2: A class hierarchy.

When a class has superclasses, its set of slots is calculated by taking the union of its own slots and those of all its superclasses. Thus each instance of D will have three slots, `a`, `b` and `d`). The slots of a class can be discovered using the `class-slots` primitive. For instance,

```
(class-slots A) ⇒ ((a))
(class-slots E) ⇒ ((a) (e) (c))
(class-slots F) ⇒ ((e) (c) (b) (d) (a) (f))
```

The ordering of the returned slots is not significant.

8.7.1 Class Precedence List

What happens when a class inherits from two or more superclasses that have a slot with the same name but incompatible definitions — for example, different init values or slot allocations? We need a rule for deciding which slot definition the derived class ends up with, and this rule is provided by the class's *Class Precedence List*.[6]

Another problem arises when invoking a generic function, and there is more than one method that could apply to the call arguments. Here we need a way of ordering the applicable methods, so that Guile knows which method to use first, which to use next if that method calls `next-method`, and so on. One of the ingredients for this ordering

[6] This section is an adaptation of material from Jeff Dalton's (J.Dalton@ed.ac.uk) *Brief introduction to CLOS*

is determining, for each given call argument, which of the specializing classes, from each applicable method's definition, is the most specific for that argument; and here again the class precedence list helps.

If inheritance was restricted such that each class could only have one superclass — which is known as *single* inheritance — class ordering would be easy. The rule would be simply that a subclass is considered more specific than its superclass.

With multiple inheritance, ordering is less obvious, and we have to impose an arbitrary rule to determine precedence. Suppose we have

```
(define-class X ()
   (x #:init-value 1))

(define-class Y ()
   (x #:init-value 2))

(define-class Z (X Y)
   (...))
```

Clearly the `Z` class is more specific than `X` or `Y`, for instances of `Z`. But which is more specific out of `X` and `Y` — and hence, for the definitions above, which `#:init-value` will take effect when creating an instance of `Z`? The rule in GOOPS is that the superclasses listed earlier are more specific than those listed later. Hence `X` is more specific than `Y`, and the `#:init-value` for slot `x` in instances of `Z` will be 1.

Hence there is a linear ordering for a class and all its superclasses, from most specific to least specific, and this ordering is called the Class Precedence List of the class.

In fact the rules above are not quite enough to always determine a unique order, but they give an idea of how things work. For example, for the `F` class shown in Figure 8.2, the class precedence list is

```
(f d e a c b <object> <top>)
```

In cases where there is any ambiguity (like this one), it is a bad idea for programmers to rely on exactly what the order is. If the order for some superclasses is important, it can be expressed directly in the class definition.

The precedence list of a class can be obtained by calling `class-precedence-list`. This function returns a ordered list whose first element is the most specific class. For instance:

```
(class-precedence-list B) ⇒ (#<<class> B 401b97c8>
                             #<<class> <object> 401e4a10>
                             #<<class> <top> 4026a9d8>)
```

Or for a more immediately readable result:

```
(map class-name (class-precedence-list B)) ⇒ (B <object> <top>)
```

8.7.2 Sorting Methods

Now, with the idea of the class precedence list, we can state precisely how the possible methods are sorted when more than one of the methods of a generic function are applicable to the call arguments.

The rules are that

- the applicable methods are sorted in order of specificity, and the most specific method is used first, then the next if that method calls `next-method`, and so on
- a method M1 is more specific than another method M2 if the first specializing class that differs, between the definitions of M1 and M2, is more specific, in M1's definition, for the corresponding actual call argument, than the specializing class in M2's definition
- a class C1 is more specific than another class C2, for an object of actual class C, if C1 comes before C2 in C's class precedence list.

8.8 Introspection

Introspection, or *reflection*, means being able to obtain information dynamically about GOOPS objects. It is perhaps best illustrated by considering an object oriented language that does not provide any introspection, namely C++.

Nothing in C++ allows a running program to obtain answers to the following types of question:

- What are the data members of this object or class?
- What classes does this class inherit from?
- Is this method call virtual or non-virtual?
- If I invoke `Employee::adjustHoliday()`, what class contains the `adjustHoliday()` method that will be applied?

In C++, answers to such questions can only be determined by looking at the source code, if you have access to it. GOOPS, on the other hand, includes procedures that allow answers to these questions — or their GOOPS equivalents — to be obtained dynamically, at run time.

8.8.1 Classes

A GOOPS class is itself an instance of the `<class>` class, or of a subclass of `<class>`. The definition of the `<class>` class has slots that are used to describe the properties of a class, including the following.

`class-name` *class* [primitive procedure]
Return the name of class *class*. This is the value of *class*'s `name` slot.

`class-direct-supers` *class* [primitive procedure]
Return a list containing the direct superclasses of *class*. This is the value of *class*'s `direct-supers` slot.

`class-direct-slots` *class* [primitive procedure]
Return a list containing the slot definitions of the direct slots of *class*. This is the value of *class*'s `direct-slots` slot.

`class-direct-subclasses` *class* [primitive procedure]
Return a list containing the direct subclasses of *class*. This is the value of *class*'s `direct-subclasses` slot.

`class-direct-methods` *class* [primitive procedure]
Return a list of all the generic function methods that use *class* as a formal parameter specializer. This is the value of *class*'s `direct-methods` slot.

`class-precedence-list` *class* [primitive procedure]
Return the class precedence list for class *class* (see Section 8.7.1 [Class Precedence List], page 788). This is the value of *class*'s `cpl` slot.

`class-slots` *class* [primitive procedure]
Return a list containing the slot definitions for all *class*'s slots, including any slots that are inherited from superclasses. This is the value of *class*'s `slots` slot.

`class-subclasses` *class* [procedure]
Return a list of all subclasses of *class*.

`class-methods` *class* [procedure]
Return a list of all methods that use *class* or a subclass of *class* as one of its formal parameter specializers.

8.8.2 Instances

`class-of` *value* [primitive procedure]
Return the GOOPS class of any Scheme *value*.

`instance?` *object* [primitive procedure]
Return `#t` if *object* is any GOOPS instance, otherwise `#f`.

`is-a?` *object class* [procedure]
Return `#t` if *object* is an instance of *class* or one of its subclasses.

You can use the `is-a?` predicate to ask whether any given value belongs to a given class, or `class-of` to discover the class of a given value. Note that when GOOPS is loaded (by code using the `(oop goops)` module) built-in classes like `<string>`, `<list>` and `<number>` are automatically set up, corresponding to all Guile Scheme types.

```
(is-a? 2.3 <number>) ⇒ #t
(is-a? 2.3 <real>) ⇒ #t
(is-a? 2.3 <string>) ⇒ #f
(is-a? '("a" "b") <string>) ⇒ #f
(is-a? '("a" "b") <list>) ⇒ #t
(is-a? (car '("a" "b")) <string>) ⇒ #t
(is-a? <string> <class>) ⇒ #t
(is-a? <class> <string>) ⇒ #f

(class-of 2.3) ⇒ #<<class> <real> 908c708>
(class-of #(1 2 3)) ⇒ #<<class> <vector> 908cd20>
(class-of <string>) ⇒ #<<class> <class> 8bd3e10>
(class-of <class>) ⇒ #<<class> <class> 8bd3e10>
```

8.8.3 Slots

`class-slot-definition` *class slot-name* [procedure]
Return the slot definition for the slot named *slot-name* in class *class*. *slot-name* should be a symbol.

`slot-definition-name` *slot-def* [procedure]
Extract and return the slot name from *slot-def*.

`slot-definition-options` *slot-def* [procedure]
Extract and return the slot options from *slot-def*.

`slot-definition-allocation` *slot-def* [procedure]
Extract and return the slot allocation option from *slot-def*. This is the value of the `#:allocation` keyword (see Section 8.4 [allocation], page 775), or `#:instance` if the `#:allocation` keyword is absent.

`slot-definition-getter` *slot-def* [procedure]
Extract and return the slot getter option from *slot-def*. This is the value of the `#:getter` keyword (see Section 8.4 [getter], page 775), or `#f` if the `#:getter` keyword is absent.

`slot-definition-setter` *slot-def* [procedure]
Extract and return the slot setter option from *slot-def*. This is the value of the `#:setter` keyword (see Section 8.4 [setter], page 775), or `#f` if the `#:setter` keyword is absent.

`slot-definition-accessor` *slot-def* [procedure]
Extract and return the slot accessor option from *slot-def*. This is the value of the `#:accessor` keyword (see Section 8.4 [accessor], page 775), or `#f` if the `#:accessor` keyword is absent.

`slot-definition-init-value` *slot-def* [procedure]
Extract and return the slot init-value option from *slot-def*. This is the value of the `#:init-value` keyword (see Section 8.4 [init-value], page 775), or the unbound value if the `#:init-value` keyword is absent.

`slot-definition-init-form` *slot-def* [procedure]
Extract and return the slot init-form option from *slot-def*. This is the value of the `#:init-form` keyword (see Section 8.4 [init-form], page 775), or the unbound value if the `#:init-form` keyword is absent.

`slot-definition-init-thunk` *slot-def* [procedure]
Extract and return the slot init-thunk option from *slot-def*. This is the value of the `#:init-thunk` keyword (see Section 8.4 [init-thunk], page 775), or `#f` if the `#:init-thunk` keyword is absent.

`slot-definition-init-keyword` *slot-def* [procedure]
Extract and return the slot init-keyword option from *slot-def*. This is the value of the `#:init-keyword` keyword (see Section 8.4 [init-keyword], page 775), or `#f` if the `#:init-keyword` keyword is absent.

`slot-init-function` *class slot-name* [procedure]
Return the initialization function for the slot named *slot-name* in class *class*. *slot-name* should be a symbol.

The returned initialization function incorporates the effects of the standard `#:init-thunk`, `#:init-form` and `#:init-value` slot options. These initializations can be overridden by the `#:init-keyword` slot option or by a specialized `initialize` method, so, in general, the function returned by `slot-init-function` may be irrelevant. For a fuller discussion, see Section 8.4 [init-value], page 775.

8.8.4 Generic Functions

A generic function is an instance of the `<generic>` class, or of a subclass of `<generic>`. The definition of the `<generic>` class has slots that are used to describe the properties of a generic function.

`generic-function-name` *gf* [primitive procedure]
Return the name of generic function *gf*.

`generic-function-methods` *gf* [primitive procedure]
Return a list of the methods of generic function *gf*. This is the value of *gf*'s `methods` slot.

Similarly, a method is an instance of the `<method>` class, or of a subclass of `<method>`; and the definition of the `<method>` class has slots that are used to describe the properties of a method.

`method-generic-function` *method* [primitive procedure]
Return the generic function that *method* belongs to. This is the value of *method*'s `generic-function` slot.

`method-specializers` *method* [primitive procedure]
Return a list of *method*'s formal parameter specializers . This is the value of *method*'s `specializers` slot.

`method-procedure` *method* [primitive procedure]
Return the procedure that implements *method*. This is the value of *method*'s `procedure` slot.

`method-source` [generic]
`method-source` (*m* `<`*method*`>`) [method]
Return an expression that prints to show the definition of method *m*.

```
(define-generic cube)

(define-method (cube (n <number>))
  (* n n n))

(map method-source (generic-function-methods cube))
⇒
((method ((n <number>)) (* n n n)))
```

8.8.5 Accessing Slots

Any slot, regardless of its allocation, can be queried, referenced and set using the following four primitive procedures.

`slot-exists?` *obj slot-name* [primitive procedure]
Return `#t` if *obj* has a slot with name *slot-name*, otherwise `#f`.

`slot-bound?` *obj slot-name* [primitive procedure]
Return `#t` if the slot named *slot-name* in *obj* has a value, otherwise `#f`.

`slot-bound?` calls the generic function `slot-missing` if *obj* does not have a slot called *slot-name* (see Section 8.8.5 [Accessing Slots], page 794).

`slot-ref` *obj slot-name* [primitive procedure]
Return the value of the slot named *slot-name* in *obj*.

`slot-ref` calls the generic function `slot-missing` if *obj* does not have a slot called *slot-name* (see Section 8.8.5 [Accessing Slots], page 794).

`slot-ref` calls the generic function `slot-unbound` if the named slot in *obj* does not have a value (see Section 8.8.5 [Accessing Slots], page 794).

`slot-set!` *obj slot-name value* [primitive procedure]
Set the value of the slot named *slot-name* in *obj* to *value*.

`slot-set!` calls the generic function `slot-missing` if *obj* does not have a slot called *slot-name* (see Section 8.8.5 [Accessing Slots], page 794).

GOOPS stores information about slots in classes. Internally, all of these procedures work by looking up the slot definition for the slot named *slot-name* in the class `(class-of` *obj*`)`, and then using the slot definition's "getter" and "setter" closures to get and set the slot value.

The next four procedures differ from the previous ones in that they take the class as an explicit argument, rather than assuming `(class-of` *obj*`)`. Therefore they allow you to apply the "getter" and "setter" closures of a slot definition in one class to an instance of a different class.

`slot-exists-using-class?` *class obj slot-name* [primitive procedure]
Return `#t` if *class* has a slot definition for a slot with name *slot-name*, otherwise `#f`.

`slot-bound-using-class?` *class obj slot-name* [primitive procedure]
Return `#t` if applying `slot-ref-using-class` to the same arguments would call the generic function `slot-unbound`, otherwise `#f`.

`slot-bound-using-class?` calls the generic function `slot-missing` if *class* does not have a slot definition for a slot called *slot-name* (see Section 8.8.5 [Accessing Slots], page 794).

`slot-ref-using-class` *class obj slot-name* [primitive procedure]
Apply the "getter" closure for the slot named *slot-name* in *class* to *obj*, and return its result.

`slot-ref-using-class` calls the generic function `slot-missing` if *class* does not have a slot definition for a slot called *slot-name* (see Section 8.8.5 [Accessing Slots], page 794).

`slot-ref-using-class` calls the generic function `slot-unbound` if the application of the "getter" closure to *obj* returns an unbound value (see Section 8.8.5 [Accessing Slots], page 794).

`slot-set-using-class!` *class obj slot-name value* [primitive procedure]
Apply the "setter" closure for the slot named *slot-name* in *class* to *obj* and *value*.

`slot-set-using-class!` calls the generic function `slot-missing` if *class* does not have a slot definition for a slot called *slot-name* (see Section 8.8.5 [Accessing Slots], page 794).

Slots whose allocation is per-class rather than per-instance can be referenced and set without needing to specify any particular instance.

`class-slot-ref` *class slot-name* [procedure]
Return the value of the slot named *slot-name* in class *class*. The named slot must have `#:class` or `#:each-subclass` allocation (see Section 8.4 [allocation], page 775).

If there is no such slot with `#:class` or `#:each-subclass` allocation, `class-slot-ref` calls the `slot-missing` generic function with arguments *class* and *slot-name*. Otherwise, if the slot value is unbound, `class-slot-ref` calls the `slot-unbound` generic function, with the same arguments.

`class-slot-set!` *class slot-name value* [procedure]
Set the value of the slot named *slot-name* in class *class* to *value*. The named slot must have `#:class` or `#:each-subclass` allocation (see Section 8.4 [allocation], page 775).

If there is no such slot with `#:class` or `#:each-subclass` allocation, `class-slot-ref` calls the `slot-missing` generic function with arguments *class* and *slot-name*.

When a `slot-ref` or `slot-set!` call specifies a non-existent slot name, or tries to reference a slot whose value is unbound, GOOPS calls one of the following generic functions.

`slot-missing` [generic]
`slot-missing` (*class* <*class*>) *slot-name* [method]
`slot-missing` (*class* <*class*>) (*object* <*object*>) *slot-name* [method]
`slot-missing` (*class* <*class*>) (*object* <*object*>) *slot-name value* [method]
When an application attempts to reference or set a class or instance slot by name, and the slot name is invalid for the specified *class* or *object*, GOOPS calls the `slot-missing` generic function.

The default methods all call `goops-error` with an appropriate message.

`slot-unbound` [generic]
`slot-unbound` (*object* <*object*>) [method]
`slot-unbound` (*class* <*class*>) *slot-name* [method]
`slot-unbound` (*class* <*class*>) (*object* <*object*>) *slot-name* [method]
When an application attempts to reference a class or instance slot, and the slot's value is unbound, GOOPS calls the `slot-unbound` generic function.

The default methods all call `goops-error` with an appropriate message.

8.9 Error Handling

The procedure `goops-error` is called to raise an appropriate error by the default methods of the following generic functions:

- `slot-missing` (see Section 8.8.5 [slot-missing], page 794)
- `slot-unbound` (see Section 8.8.5 [slot-unbound], page 794)
- `no-method` (see Section 8.6.6 [no-method], page 787)
- `no-applicable-method` (see Section 8.6.6 [no-applicable-method], page 787)
- `no-next-method` (see Section 8.6.6 [no-next-method], page 787)

If you customize these functions for particular classes or metaclasses, you may still want to use `goops-error` to signal any error conditions that you detect.

`goops-error` *format-string arg ...* [procedure]

Raise an error with key `goops-error` and error message constructed from *format-string* and *arg ...*. Error message formatting is as done by `scm-error`.

8.10 GOOPS Object Miscellany

Here we cover some points about GOOPS objects that aren't substantial enough to merit sections on their own.

Object Equality

When GOOPS is loaded, `eqv?`, `equal?` and `=` become generic functions, and you can define methods for them, specialized for your own classes, so as to control what the various kinds of equality mean for your classes.

For example, the `assoc` procedure, for looking up an entry in an alist, is specified as using `equal?` to determine when the car of an entry in the alist is the same as the key parameter that `assoc` is called with. Hence, if you had defined a new class, and wanted to use instances of that class as the keys in an alist, you could define a method for `equal?`, for your class, to control `assoc`'s lookup precisely.

Cloning Objects

`shallow-clone` [generic]
`shallow-clone` (*self* <*object*>) [method]

Return a "shallow" clone of *self*. The default method makes a shallow clone by allocating a new instance and copying slot values from self to the new instance. Each slot value is copied either as an immediate value or by reference.

`deep-clone` [generic]
`deep-clone` (*self* <*object*>) [method]

Return a "deep" clone of *self*. The default method makes a deep clone by allocating a new instance and copying or cloning slot values from self to the new instance. If a slot value is an instance (satisfies `instance?`), it is cloned by calling `deep-clone` on that value. Other slot values are copied either as immediate values or by reference.

Write and Display

`write` *object port* [primitive generic]
`display` *object port* [primitive generic]

When GOOPS is loaded, `write` and `display` become generic functions with special methods for printing

- objects - instances of the class `<object>`
- foreign objects - instances of the class `<foreign-object>`
- classes - instances of the class `<class>`
- generic functions - instances of the class `<generic>`
- methods - instances of the class `<method>`.

`write` and `display` print non-GOOPS values in the same way as the Guile primitive `write` and `display` functions.

In addition to the cases mentioned, you can of course define `write` and `display` methods for your own classes, to customize how instances of those classes are printed.

8.11 The Metaobject Protocol

At this point, we've said about as much as can be said about GOOPS without having to confront the idea of the metaobject protocol. There are a couple more topics that could be discussed in isolation first — class redefinition, and changing the class of existing instances — but in practice developers using them will be advanced enough to want to understand the metaobject protocol too, and will probably be using the protocol to customize exactly what happens during these events.

So let's plunge in. GOOPS is based on a "metaobject protocol" (aka "MOP") derived from the ones used in CLOS (the Common Lisp Object System), tiny-clos (a small Scheme implementation of a subset of CLOS functionality) and STKlos.

The MOP underlies many possible GOOPS customizations — such as defining an `initialize` method to customize the initialization of instances of an application-defined class — and an understanding of the MOP makes it much easier to explain such customizations in a precise way. And at a deeper level, understanding the MOP is a key part of understanding GOOPS, and of taking full advantage of GOOPS' power, by customizing the behaviour of GOOPS itself.

8.11.1 Metaobjects and the Metaobject Protocol

The building blocks of GOOPS are classes, slot definitions, instances, generic functions and methods. A class is a grouping of inheritance relations and slot definitions. An instance is an object with slots that are allocated following the rules implied by its class's superclasses and slot definitions. A generic function is a collection of methods and rules for determining which of those methods to apply when the generic function is invoked. A method is a procedure and a set of specializers that specify the type of arguments to which the procedure is applicable.

Of these entities, GOOPS represents classes, generic functions and methods as "metaobjects". In other words, the values in a GOOPS program that describe classes, generic

functions and methods, are themselves instances (or "objects") of special GOOPS classes that encapsulate the behaviour, respectively, of classes, generic functions, and methods.

(The other two entities are slot definitions and instances. Slot definitions are not strictly instances, but every slot definition is associated with a GOOPS class that specifies the behaviour of the slot as regards accessibility and protection from garbage collection. Instances are of course objects in the usual sense, and there is no benefit from thinking of them as metaobjects.)

The "metaobject protocol" (or "MOP") is the specification of the generic functions which determine the behaviour of these metaobjects and the circumstances in which these generic functions are invoked.

For a concrete example of what this means, consider how GOOPS calculates the set of slots for a class that is being defined using `define-class`. The desired set of slots is the union of the new class's direct slots and the slots of all its superclasses. But `define-class` itself does not perform this calculation. Instead, there is a method of the `initialize` generic function that is specialized for instances of type `<class>`, and it is this method that performs the slot calculation.

`initialize` is a generic function which GOOPS calls whenever a new instance is created, immediately after allocating memory for a new instance, in order to initialize the new instance's slots. The sequence of steps is as follows.

- `define-class` uses `make` to make a new instance of the `<class>` class, passing as initialization arguments the superclasses, slot definitions and class options that were specified in the `define-class` form.
- `make` allocates memory for the new instance, and invokes the `initialize` generic function to initialize the new instance's slots.
- The `initialize` generic function applies the method that is specialized for instances of type `<class>`, and this method performs the slot calculation.

In other words, rather than being hardcoded in `define-class`, the default behaviour of class definition is encapsulated by generic function methods that are specialized for the class `<class>`.

It is possible to create a new class that inherits from `<class>`, which is called a "metaclass", and to write a new `initialize` method that is specialized for instances of the new metaclass. Then, if the `define-class` form includes a `#:metaclass` class option whose value is the new metaclass, the class that is defined by the `define-class` form will be an instance of the new metaclass rather than of the default `<class>`, and will be defined in accordance with the new `initialize` method. Thus the default slot calculation, as well as any other aspect of the new class's relationship with its superclasses, can be modified or overridden.

In a similar way, the behaviour of generic functions can be modified or overridden by creating a new class that inherits from the standard generic function class `<generic>`, writing appropriate methods that are specialized to the new class, and creating new generic functions that are instances of the new class.

The same is true for method metaobjects. And the same basic mechanism allows the application class author to write an `initialize` method that is specialized to their application class, to initialize instances of that class.

Such is the power of the MOP. Note that `initialize` is just one of a large number of generic functions that can be customized to modify the behaviour of application objects and classes and of GOOPS itself. Each following section covers a particular area of GOOPS functionality, and describes the generic functions that are relevant for customization of that area.

8.11.2 Metaclasses

A *metaclass* is the class of an object which represents a GOOPS class. Put more succinctly, a metaclass is a class's class.

Most GOOPS classes have the metaclass `<class>` and, by default, any new class that is created using `define-class` has the metaclass `<class>`.

But what does this really mean? To find out, let's look in more detail at what happens when a new class is created using `define-class`:

```
(define-class <my-class> (<object>) . slots)
```

Guile expands this to something like:

```
(define <my-class> (class (<object>) . slots))
```

which in turn expands to:

```
(define <my-class>
  (make <class> #:dsupers (list <object>) #:slots slots))
```

As this expansion makes clear, the resulting value of `<my-class>` is an instance of the class `<class>` with slot values specifying the superclasses and slot definitions for the class `<my-class>`. (`#:dsupers` and `#:slots` are initialization keywords for the `dsupers` and `dslots` slots of the `<class>` class.)

Now suppose that you want to define a new class with a metaclass other than the default `<class>`. This is done by writing:

```
(define-class <my-class2> (<object>)
   slot ...
   #:metaclass <my-metaclass>)
```

and Guile expands *this* to something like:

```
(define <my-class2>
  (make <my-metaclass> #:dsupers (list <object>) #:slots slots))
```

In this case, the value of `<my-class2>` is an instance of the more specialized class `<my-metaclass>`. Note that `<my-metaclass>` itself must previously have been defined as a subclass of `<class>`. For a full discussion of when and how it is useful to define new metaclasses, see Section 8.11.3 [MOP Specification], page 800.

Now let's make an instance of `<my-class2>`:

```
(define my-object (make <my-class2> ...))
```

All of the following statements are correct expressions of the relationships between `my-object`, `<my-class2>`, `<my-metaclass>` and `<class>`.

- `my-object` is an instance of the class `<my-class2>`.
- `<my-class2>` is an instance of the class `<my-metaclass>`.
- `<my-metaclass>` is an instance of the class `<class>`.

- The class of `my-object` is `<my-class2>`.
- The class of `<my-class2>` is `<my-metaclass>`.
- The class of `<my-metaclass>` is `<class>`.

8.11.3 MOP Specification

The aim of the MOP specification in this chapter is to specify all the customizable generic function invocations that can be made by the standard GOOPS syntax, procedures and methods, and to explain the protocol for customizing such invocations.

A generic function invocation is customizable if the types of the arguments to which it is applied are not completely determined by the lexical context in which the invocation appears. For example, the `(initialize` *`instance initargs`*`)` invocation in the default `make-instance` method is customizable, because the type of the *`instance`* argument is determined by the class that was passed to `make-instance`.

(Whereas — to give a counter-example — the `(make <generic> #:name ',name)` invocation in `define-generic` is not customizable, because all of its arguments have lexically determined types.)

When using this rule to decide whether a given generic function invocation is customizable, we ignore arguments that are expected to be handled in method definitions as a single "rest" list argument.

For each customizable generic function invocation, the *invocation protocol* is explained by specifying

- what, conceptually, the applied method is intended to do
- what assumptions, if any, the caller makes about the applied method's side effects
- what the caller expects to get as the applied method's return value.

8.11.4 Instance Creation Protocol

`make <class> .` *`initargs`* (method)

- `allocate-instance` *`class initargs`* (generic)

  The applied `allocate-instance` method should allocate storage for a new instance of class *class* and return the uninitialized instance.

- `initialize` *`instance initargs`* (generic)

  instance is the uninitialized instance returned by `allocate-instance`. The applied method should initialize the new instance in whatever sense is appropriate for its class. The method's return value is ignored.

`make` itself is a generic function. Hence the `make` invocation itself can be customized in the case where the new instance's metaclass is more specialized than the default `<class>`, by defining a `make` method that is specialized to that metaclass.

Normally, however, the method for classes with metaclass `<class>` will be applied. This method calls two generic functions:

- (allocate-instance *class* . *initargs*)
- (initialize *instance* . *initargs*)

`allocate-instance` allocates storage for and returns the new instance, uninitialized. You might customize `allocate-instance`, for example, if you wanted to provide a GOOPS wrapper around some other object programming system.

To do this, you would create a specialized metaclass, which would act as the metaclass for all classes and instances from the other system. Then define an `allocate-instance` method, specialized to that metaclass, which calls a Guile primitive C function (or FFI code), which in turn allocates the new instance using the interface of the other object system.

In this case, for a complete system, you would also need to customize a number of other generic functions like `make` and `initialize`, so that GOOPS knows how to make classes from the other system, access instance slots, and so on.

`initialize` initializes the instance that is returned by `allocate-instance`. The standard GOOPS methods perform initializations appropriate to the instance class.

- At the least specialized level, the method for instances of type `<object>` performs internal GOOPS instance initialization, and initializes the instance's slots according to the slot definitions and any slot initialization keywords that appear in *initargs*.
- The method for instances of type `<class>` calls `(next-method)`, then performs the class initializations described in Section 8.11.5 [Class Definition Protocol], page 801.
- and so on for generic functions, methods, operator classes ...

Similarly, you can customize the initialization of instances of any application-defined class by defining an `initialize` method specialized to that class.

Imagine a class whose instances' slots need to be initialized at instance creation time by querying a database. Although it might be possible to achieve this a combination of `#:init-thunk` keywords and closures in the slot definitions, it may be neater to write an `initialize` method for the class that queries the database once and initializes all the dependent slot values according to the results.

8.11.5 Class Definition Protocol

Here is a summary diagram of the syntax, procedures and generic functions that may be involved in class definition.

`define-class` (syntax)

- `class` (syntax)
  - `make-class` (procedure)
    - `ensure-metaclass` (procedure)
    - `make` *`metaclass`* `...` (generic)
      - `allocate-instance` (generic)
      - `initialize` (generic)
        - `compute-cpl` (generic)
          - `compute-std-cpl` (procedure)
        - `compute-slots` (generic)
        - `compute-get-n-set` (generic)
        - `compute-getter-method` (generic)

    - `compute-setter-method` (generic)
- `class-redefinition` (generic)
    - `remove-class-accessors` (generic)
    - `update-direct-method!` (generic)
    - `update-direct-subclass!` (generic)

Wherever a step above is marked as "generic", it can be customized, and the detail shown below it is only "correct" insofar as it describes what the default method of that generic function does. For example, if you write an `initialize` method, for some metaclass, that does not call `next-method` and does not call `compute-cpl`, then `compute-cpl` will not be called when a class is defined with that metaclass.

A `(define-class ...)` form (see Section 8.2 [Class Definition], page 773) expands to an expression which

- checks that it is being evaluated only at top level
- defines any accessors that are implied by the *slot-definition*s
- uses `class` to create the new class
- checks for a previous class definition for *name* and, if found, handles the redefinition by invoking `class-redefinition` (see Section 8.12 [Redefining a Class], page 809).

`class` *name* (*super ...*) *slot-definition ... class-option ...* [syntax]
: Return a newly created class that inherits from *super*s, with direct slots defined by *slot-definition*s and *class-option*s. For the format of *slot-definition*s and *class-option*s, see Section 8.2 [define-class], page 773.

`class` expands to an expression which

- processes the class and slot definition options to check that they are well-formed, to convert the `#:init-form` option to an `#:init-thunk` option, to supply a default environment parameter (the current top-level environment) and to evaluate all the bits that need to be evaluated
- calls `make-class` to create the class with the processed and evaluated parameters.

`make-class` *supers slots class-option ...* [procedure]
: Return a newly created class that inherits from *supers*, with direct slots defined by *slots* and *class-option*s. For the format of *slots* and *class-option*s, see Section 8.2 [define-class], page 773, except note that for `make-class`, *slots* is a separate list of slot definitions.

`make-class`

- adds `<object>` to the *supers* list if *supers* is empty or if none of the classes in *supers* have `<object>` in their class precedence list
- defaults the `#:environment`, `#:name` and `#:metaclass` options, if they are not specified by *options*, to the current top-level environment, the unbound value, and `(ensure-metaclass` *supers*`)` respectively
- checks for duplicate classes in *supers* and duplicate slot names in *slots*, and signals an error if there are any duplicates
- calls `make`, passing the metaclass as the first parameter and all other parameters as option keywords with values.

`ensure-metaclass` *supers env* [procedure]

Return a metaclass suitable for a class that inherits from the list of classes in *supers*. The returned metaclass is the union by inheritance of the metaclasses of the classes in *supers*.

In the simplest case, where all the *supers* are straightforward classes with metaclass `<class>`, the returned metaclass is just `<class>`.

For a more complex example, suppose that *supers* contained one class with metaclass `<operator-class>` and one with metaclass `<foreign-object-class>`. Then the returned metaclass would be a class that inherits from both `<operator-class>` and `<foreign-object-class>`.

If *supers* is the empty list, `ensure-metaclass` returns the default GOOPS metaclass `<class>`.

GOOPS keeps a list of the metaclasses created by `ensure-metaclass`, so that each required type of metaclass only has to be created once.

The `env` parameter is ignored.

`make` *metaclass initarg . . .* [generic]

metaclass is the metaclass of the class being defined, either taken from the `#:metaclass` class option or computed by `ensure-metaclass`. The applied method must create and return the fully initialized class metaobject for the new class definition.

The `(make` *`metaclass initarg ...`*`)` invocation is a particular case of the instance creation protocol covered in the previous section. It will create an class metaobject with metaclass *metaclass*. By default, this metaobject will be initialized by the `initialize` method that is specialized for instances of type `<class>`.

The `initialize` method for classes (signature `(initialize <class> initargs)`) calls the following generic functions.

- `compute-cpl` *`class`* (generic)

  The applied method should compute and return the class precedence list for *class* as a list of class metaobjects. When `compute-cpl` is called, the following *class* metaobject slots have all been initialized: `name`, `direct-supers`, `direct-slots`, `direct-subclasses` (empty), `direct-methods`. The value returned by `compute-cpl` will be stored in the `cpl` slot.

- `compute-slots` *`class`* (generic)

  The applied method should compute and return the slots (union of direct and inherited) for *class* as a list of slot definitions. When `compute-slots` is called, all the *class* metaobject slots mentioned for `compute-cpl` have been initialized, plus the following: `cpl`, `redefined` (`#f`), `environment`. The value returned by `compute-slots` will be stored in the `slots` slot.

- `compute-get-n-set` *`class slot-def`* (generic)

  `initialize` calls `compute-get-n-set` for each slot computed by `compute-slots`. The applied method should compute and return a pair of closures that, respectively, get and set the value of the specified slot. The get closure should have arity 1 and expect a single argument that is the instance whose slot value is to be retrieved. The set closure

should have arity 2 and expect two arguments, where the first argument is the instance whose slot value is to be set and the second argument is the new value for that slot. The closures should be returned in a two element list: `(list` *`get set`*`)`.

The closures returned by `compute-get-n-set` are stored as part of the value of the *class* metaobject's `getters-n-setters` slot. Specifically, the value of this slot is a list with the same number of elements as there are slots in the class, and each element looks either like

`(`*`slot-name-symbol init-function . index`*`)`

or like

`(`*`slot-name-symbol init-function get set`*`)`

Where the get and set closures are replaced by *index*, the slot is an instance slot and *index* is the slot's index in the underlying structure: GOOPS knows how to get and set the value of such slots and so does not need specially constructed get and set closures. Otherwise, *get* and *set* are the closures returned by `compute-get-n-set`.

The structure of the `getters-n-setters` slot value is important when understanding the next customizable generic functions that `initialize` calls...

- `compute-getter-method` *`class gns`* (generic)

  `initialize` calls `compute-getter-method` for each of the class's slots (as determined by `compute-slots`) that includes a `#:getter` or `#:accessor` slot option. *gns* is the element of the *class* metaobject's `getters-n-setters` slot that specifies how the slot in question is referenced and set, as described above under `compute-get-n-set`. The applied method should create and return a method that is specialized for instances of type *class* and uses the get closure to retrieve the slot's value. `initialize` uses `add-method!` to add the returned method to the generic function named by the slot definition's `#:getter` or `#:accessor` option.

- `compute-setter-method` *`class gns`* (generic)

  `compute-setter-method` is invoked with the same arguments as `compute-getter-method`, for each of the class's slots that includes a `#:setter` or `#:accessor` slot option. The applied method should create and return a method that is specialized for instances of type *class* and uses the set closure to set the slot's value. `initialize` then uses `add-method!` to add the returned method to the generic function named by the slot definition's `#:setter` or `#:accessor` option.

8.11.6 Customizing Class Definition

If the metaclass of the new class is something more specialized than the default `<class>`, then the type of *class* in the calls above is more specialized than `<class>`, and hence it becomes possible to define generic function methods, specialized for the new class's metaclass, that can modify or override the default behaviour of `initialize`, `compute-cpl` or `compute-get-n-set`.

`compute-cpl` computes the class precedence list ("CPL") for the new class (see Section 8.7.1 [Class Precedence List], page 788), and returns it as a list of class objects. The CPL is important because it defines a superclass ordering that is used, when a generic function is invoked upon an instance of the class, to decide which of the available generic function methods is the most specific. Hence `compute-cpl` could be customized in order to modify the CPL ordering algorithm for all classes with a special metaclass.

The default CPL algorithm is encapsulated by the `compute-std-cpl` procedure, which is called by the default `compute-cpl` method.

`compute-std-cpl` *class* [procedure]
Compute and return the class precedence list for *class* according to the algorithm described in Section 8.7.1 [Class Precedence List], page 788.

`compute-slots` computes and returns a list of all slot definitions for the new class. By default, this list includes the direct slot definitions from the `define-class` form, plus the slot definitions that are inherited from the new class's superclasses. The default `compute-slots` method uses the CPL computed by `compute-cpl` to calculate this union of slot definitions, with the rule that slots inherited from superclasses are shadowed by direct slots with the same name. One possible reason for customizing `compute-slots` would be to implement an alternative resolution strategy for slot name conflicts.

`compute-get-n-set` computes the low-level closures that will be used to get and set the value of a particular slot, and returns them in a list with two elements.

The closures returned depend on how storage for that slot is allocated. The standard `compute-get-n-set` method, specialized for classes of type `<class>`, handles the standard GOOPS values for the `#:allocation` slot option (see Section 8.4 [allocation], page 775). By defining a new `compute-get-n-set` method for a more specialized metaclass, it is possible to support new types of slot allocation.

Suppose you wanted to create a large number of instances of some class with a slot that should be shared between some but not all instances of that class - say every 10 instances should share the same slot storage. The following example shows how to implement and use a new type of slot allocation to do this.

```
(define-class <batched-allocation-metaclass> (<class>))

(let ((batch-allocation-count 0)
      (batch-get-n-set #f))
  (define-method (compute-get-n-set
                  (class <batched-allocation-metaclass>) s)
    (case (slot-definition-allocation s)
      ((#:batched)
       ;; If we've already used the same slot storage for 10 instances,
       ;; reset variables.
       (if (= batch-allocation-count 10)
           (begin
             (set! batch-allocation-count 0)
             (set! batch-get-n-set #f)))
       ;; If we don't have a current pair of get and set closures,
       ;; create one.  make-closure-variable returns a pair of closures
       ;; around a single Scheme variable - see goops.scm for details.
       (or batch-get-n-set
           (set! batch-get-n-set (make-closure-variable)))
       ;; Increment the batch allocation count.
       (set! batch-allocation-count (+ batch-allocation-count 1))
       batch-get-n-set)
```

```
        ;; Call next-method to handle standard allocation types.
        (else (next-method)))))

(define-class <class-using-batched-slot> ()
  ...
  (c #:allocation #:batched)
  ...
  #:metaclass <batched-allocation-metaclass>)
```

The usage of `compute-getter-method` and `compute-setter-method` is described in Section 8.11.5 [Class Definition Protocol], page 801.

`compute-cpl` and `compute-get-n-set` are called by the standard `initialize` method for classes whose metaclass is `<class>`. But `initialize` itself can also be modified, by defining an `initialize` method specialized to the new class's metaclass. Such a method could complete override the standard behaviour, by not calling `(next-method)` at all, but more typically it would perform additional class initialization steps before and/or after calling `(next-method)` for the standard behaviour.

8.11.7 Method Definition

`define-method` (syntax)

- `add-method!` *`target method`* (generic)

`define-method` invokes the `add-method!` generic function to handle adding the new method to a variety of possible targets. GOOPS includes methods to handle *target* as

- a generic function (the most common case)
- a procedure
- a primitive generic (see Section 8.6.2 [Extending Primitives], page 782)

By defining further methods for `add-method!`, you can theoretically handle adding methods to further types of target.

8.11.8 Method Definition Internals

`define-method`:

- checks the form of the first parameter, and applies the following steps to the accessor's setter if it has the `(setter ...)` form
- interpolates a call to `define-generic` or `define-accessor` if a generic function is not already defined with the supplied name
- calls `method` with the *parameter*s and *body*, to make a new method instance
- calls `add-method!` to add this method to the relevant generic function.

`method` (*parameter ...*) *body ...* [syntax]

Make a method whose specializers are defined by the classes in *parameter*s and whose procedure definition is constructed from the *parameter* symbols and *body* forms.

The *parameter* and *body* parameters should be as for `define-method` (see Section 8.6 [define-method], page 780).

`method`:

- extracts formals and specializing classes from the *parameter*s, defaulting the class for unspecialized parameters to `<top>`
- creates a closure using the formals and the *body* forms
- calls `make` with metaclass `<method>` and the specializers and closure using the `#:specializers` and `#:procedure` keywords.

`make-method` *specializers procedure* [procedure]
Make a method using *specializers* and *procedure*.

specializers should be a list of classes that specifies the parameter combinations to which this method will be applicable.

procedure should be the closure that will applied to the generic function parameters when this method is invoked.

`make-method` is a simple wrapper around `make` with metaclass `<method>`.

`add-method!` *target method* [generic]
Generic function for adding method *method* to *target*.

`add-method!` (*generic* `<`*generic*`>`) (*method* `<`*method*`>`) [method]
Add method *method* to the generic function *generic*.

`add-method!` (*proc* `<`*procedure*`>`) (*method* `<`*method*`>`) [method]
If *proc* is a procedure with generic capability (see Section 8.6.2 [generic-capability?], page 782), upgrade it to a primitive generic and add *method* to its generic function definition.

`add-method!` (*pg* `<`*primitive-generic*`>`) (*method* `<`*method*`>`) [method]
Add method *method* to the generic function definition of *pg*.

Implementation: `(add-method! (primitive-generic-generic pg) method)`.

`add-method!` (*whatever* `<`*top*`>`) (*method* `<`*method*`>`) [method]
Raise an error indicating that *whatever* is not a valid generic function.

8.11.9 Generic Function Internals

`define-generic` calls `ensure-generic` to upgrade a pre-existing procedure value, or `make` with metaclass `<generic>` to create a new generic function.

`define-accessor` calls `ensure-accessor` to upgrade a pre-existing procedure value, or `make-accessor` to create a new accessor.

`ensure-generic` *old-definition* [*name*] [procedure]
Return a generic function with name *name*, if possible by using or upgrading *old-definition*. If unspecified, *name* defaults to `#f`.

If *old-definition* is already a generic function, it is returned unchanged.

If *old-definition* is a Scheme procedure or procedure-with-setter, `ensure-generic` returns a new generic function that uses *old-definition* for its default procedure and setter.

Otherwise `ensure-generic` returns a new generic function with no defaults and no methods.

`make-generic` [*name*] [procedure]
: Return a new generic function with name `(car name)`. If unspecified, *name* defaults to `#f`.

`ensure-generic` calls `make` with metaclasses `<generic>` and `<generic-with-setter>`, depending on the previous value of the variable that it is trying to upgrade.

`make-generic` is a simple wrapper for `make` with metaclass `<generic>`.

`ensure-accessor` *proc* [*name*] [procedure]
: Return an accessor with name *name*, if possible by using or upgrading *proc*. If unspecified, *name* defaults to `#f`.

  If *proc* is already an accessor, it is returned unchanged.

  If *proc* is a Scheme procedure, procedure-with-setter or generic function, `ensure-accessor` returns an accessor that reuses the reusable elements of *proc*.

  Otherwise `ensure-accessor` returns a new accessor with no defaults and no methods.

`make-accessor` [*name*] [procedure]
: Return a new accessor with name `(car name)`. If unspecified, *name* defaults to `#f`.

`ensure-accessor` calls `make` with metaclass `<generic-with-setter>`, as well as calls to `ensure-generic`, `make-accessor` and (tail recursively) `ensure-accessor`.

`make-accessor` calls `make` twice, first with metaclass `<generic>` to create a generic function for the setter, then with metaclass `<generic-with-setter>` to create the accessor, passing the setter generic function as the value of the `#:setter` keyword.

8.11.10 Generic Function Invocation

There is a detailed and customizable protocol involved in the process of invoking a generic function — i.e., in the process of deciding which of the generic function's methods are applicable to the current arguments, and which one of those to apply. Here is a summary diagram of the generic functions involved.

`apply-generic` (generic)

- `no-method` (generic)
- `compute-applicable-methods` (generic)
- `sort-applicable-methods` (generic)
  - `method-more-specific?` (generic)
- `apply-methods` (generic)
  - `apply-method` (generic)
  - `no-next-method` (generic)
- `no-applicable-method`

We do not yet have full documentation for these. Please refer to the code (`oop/goops.scm`) for details.

8.12 Redefining a Class

Suppose that a class `<my-class>` is defined using `define-class` (see Section 8.2 [define-class], page 773), with slots that have accessor functions, and that an application has created several instances of `<my-class>` using `make` (see Section 8.3 [make], page 774). What then happens if `<my-class>` is redefined by calling `define-class` again?

8.12.1 Redefinable Classes

The ability for a class to be redefined is a choice for a class author to make. By default, classes in GOOPS are *not* redefinable. A redefinable class is an instance of `<redefinable-class>`; that is to say, a class with `<redefinable-class>` as its metaclass. Accordingly, to define a redefinable class, add `#:metaclass <redefinable-class>` to its class definition:

```
(define-class <foo> ()
  #:metaclass <redefinable-class>)
```

Note that any subclass of `<foo>` is also redefinable, without the need to explicitly pass the `#:metaclass` argument, so you only need to specify `#:metaclass` for the roots of your application's class hierarchy.

```
(define-class <bar> (<foo>))
(class-of <bar>) ⇒ <redefinable-class>
```

Note that prior to Guile 3.0, all GOOPS classes were redefinable in theory. In practice, attempting to, for example, redefine `<class>` itself would almost certainly not do what you want. Still, redefinition is an interesting capability when building long-lived resilient systems, so GOOPS does offer this facility.

8.12.2 Default Class Redefinition Behaviour

When a class is defined using `define-class` and the class name was previously defined, by default the new binding just replaces the old binding. This is the normal behavior for `define`. However if both the old and new bindings are redefinable classes (instances of `<redefinable-class>`), then the class will be updated in place, and its instances lazily migrated over.

The way that the class is updated and the way that the instances migrate over are of course part of the meta-object protocol. However the default behavior usually suffices, and it goes as follows.

- All existing direct instances of `<my-class>` are converted to be instances of the new class. This is achieved by preserving the values of slots that exist in both the old and new definitions, and initializing the values of new slots in the usual way (see Section 8.3 [make], page 774).
- All existing subclasses of `<my-class>` are redefined, as though the `define-class` expressions that defined them were re-evaluated following the redefinition of `<my-class>`, and the class redefinition process described here is applied recursively to the redefined subclasses.
- Once all of its instances and subclasses have been updated, the class metaobject previously bound to the variable `<my-class>` is no longer needed and so can be allowed to be garbage collected.

To keep things tidy, GOOPS also needs to do a little housekeeping on methods that are associated with the redefined class.

- Slot accessor methods for slots in the old definition should be removed from their generic functions. They will be replaced by accessor methods for the slots of the new class definition.
- Any generic function method that uses the old `<my-class>` metaobject as one of its formal parameter specializers must be updated to refer to the new `<my-class>` metaobject. (Whenever a new generic function method is defined, `define-method` adds the method to a list stored in the class metaobject for each class used as a formal parameter specializer, so it is easy to identify all the methods that must be updated when a class is redefined.)

If this class redefinition strategy strikes you as rather counter-intuitive, bear in mind that it is derived from similar behaviour in other object systems such as CLOS, and that experience in those systems has shown it to be very useful in practice.

Also bear in mind that, like most of GOOPS' default behaviour, it can be customized. . .

8.12.3 Customizing Class Redefinition

When `define-class` notices that a class is being redefined, it constructs the new class metaobject as usual, then invokes the `class-redefinition` generic function with the old and new classes as arguments. Therefore, if the old or new classes have metaclasses other than the default `<redefinable-class>`, class redefinition behaviour can be customized by defining a `class-redefinition` method that is specialized for the relevant metaclasses.

`class-redefinition` [generic]
: Handle the class redefinition from *old* to *new*, and return the new class metaobject that should be bound to the variable specified by `define-class`'s first argument.

`class-redefinition` (*old* `<`*top*`>`) (*new* `<`*class*`>`) [method]
: Not all classes are redefinable, and not all previous bindings are classes. See Section 8.12.1 [Redefinable Classes], page 809. This default method just returns *new*.

`class-redefinition` (*old* `<`*redefinable-class*`>`) (*new* `<`*redefinable-class*`>`) [method]
: This method implements GOOPS' default class redefinition behaviour, as described in Section 8.12.2 [Default Class Redefinition Behaviour], page 809. Returns the metaobject for the new class definition.

The `class-redefinition` method for classes with metaclass `<redefinable-class>` calls the following generic functions, which could of course be individually customized.

`remove-class-accessors!` *old* [generic]
: The default `remove-class-accessors!` method removes the accessor methods of the old class from all classes which they specialize.

`update-direct-method!` *method old new* [generic]
: The default `update-direct-method!` method substitutes the new class for the old in all methods specialized to the old class.

`update-direct-subclass!` *subclass old new* [generic]

The default `update-direct-subclass!` method invokes `class-redefinition` recursively to handle the redefinition of subclasses.

An alternative class redefinition strategy could be to leave all existing instances as instances of the old class, but accepting that the old class is now "nameless", since its name has been taken over by the new definition. In this strategy, any existing subclasses could also be left as they are, on the understanding that they inherit from a nameless superclass.

This strategy is easily implemented in GOOPS, by defining a new metaclass, that will be used as the metaclass for all classes to which the strategy should apply, and then defining a `class-redefinition` method that is specialized for this metaclass:

```
(define-class <can-be-nameless> (<redefinable-class>))

(define-method (class-redefinition (old <can-be-nameless>)
                                   (new <class>))
  new)
```

When customization can be as easy as this, aren't you glad that GOOPS implements the far more difficult strategy as its default!

8.13 Changing the Class of an Instance

When a redefinable class is redefined, any existing instance of the redefined class will be modified for the new class definition before the next time that any of the instance's slots is referenced or set. GOOPS modifies each instance by calling the generic function `change-class`.

More generally, you can change the class of an existing instance at any time by invoking the generic function `change-class` with two arguments: the instance and the new class.

The default method for `change-class` decides how to implement the change of class by looking at the slot definitions for the instance's existing class and for the new class. If the new class has slots with the same name as slots in the existing class, the values for those slots are preserved. Slots that are present only in the existing class are discarded. Slots that are present only in the new class are initialized using the corresponding slot definition's init function (see Section 8.8.1 [slot-init-function], page 790).

`change-class` *instance new-class* [generic]

`change-class` (*obj* <*object*>) (*new* <*redefinable-class*>) [method]

Modify instance *obj* to make it an instance of class *new*. *obj* itself must already be an instance of a redefinable class.

The value of each of *obj*'s slots is preserved only if a similarly named slot exists in *new*; any other slot values are discarded.

The slots in *new* that do not correspond to any of *obj*'s pre-existing slots are initialized according to *new*'s slot definitions' init functions.

The default `change-class` method also invokes another generic function, `update-instance-for-different-class`, as the last thing that it does before returning. The applied `update-instance-for-different-class` method can make any further

adjustments to *new-instance* that are required to complete or modify the change of class. The return value from the applied method is ignored.

`update-instance-for-different-class` *old-instance new-instance* [generic]
: A generic function that can be customized to put finishing touches to an instance whose class has just been changed. The default `update-instance-for-different-class` method does nothing.

Customized change of class behaviour can be implemented by defining `change-class` methods that are specialized either by the class of the instances to be modified or by the metaclass of the new class.

9 Guile Implementation

At some point, after one has been programming in Scheme for some time, another level of Scheme comes into view: its implementation. Knowledge of how Scheme can be implemented turns out to be necessary to become an expert hacker. As Peter Norvig notes in his retrospective on PAIP[1], "The expert Lisp programmer eventually develops a good 'efficiency model'."

By this Norvig means that over time, the Lisp hacker eventually develops an understanding of how much her code "costs" in terms of space and time.

This chapter describes Guile as an implementation of Scheme: its history, how it represents and evaluates its data, and its compiler. This knowledge can help you to make that step from being one who is merely familiar with Scheme to being a real hacker.

9.1 A Brief History of Guile

Guile is an artifact of historical processes, both as code and as a community of hackers. It is sometimes useful to know this history when hacking the source code, to know about past decisions and future directions.

Of course, the real history of Guile is written by the hackers hacking and not the writers writing, so we round up the section with a note on current status and future directions.

9.1.1 The Emacs Thesis

The story of Guile is the story of bringing the development experience of Emacs to the mass of programs on a GNU system.

Emacs, when it was first created in its GNU form in 1984, was a new take on the problem of "how to make a program". The Emacs thesis is that it is delightful to create composite programs based on an orthogonal kernel written in a low-level language together with a powerful, high-level extension language.

Extension languages foster extensible programs, programs which adapt readily to different users and to changing times. Proof of this can be seen in Emacs' current and continued existence, spanning more than a quarter-century.

Besides providing for modification of a program by others, extension languages are good for *intension* as well. Programs built in "the Emacs way" are pleasurable and easy for their authors to flesh out with the features that they need.

After the Emacs experience was appreciated more widely, a number of hackers started to consider how to spread this experience to the rest of the GNU system. It was clear that the easiest way to Emacsify a program would be to embed a shared language implementation into it.

9.1.2 Early Days

Tom Lord was the first to fully concentrate his efforts on an embeddable language runtime, which he named "GEL", the GNU Extension Language.

[1] PAIP is the common abbreviation for *Paradigms of Artificial Intelligence Programming*, an old but still useful text on Lisp. Norvig's retrospective sums up the lessons of PAIP, and can be found at `http://norvig.com/Lisp-retro.html`.

GEL was the product of converting SCM, Aubrey Jaffer's implementation of Scheme, into something more appropriate to embedding as a library. (SCM was itself based on an implementation by George Carrette, SIOD.)

Lord managed to convince Richard Stallman to dub GEL the official extension language for the GNU project. It was a natural fit, given that Scheme was a cleaner, more modern Lisp than Emacs Lisp. Part of the argument was that eventually when GEL became more capable, it could gain the ability to execute other languages, especially Emacs Lisp.

Due to a naming conflict with another programming language, Jim Blandy suggested a new name for GEL: "Guile". Besides being a recursive acronym, "Guile" craftily follows the naming of its ancestors, "Planner", "Conniver", and "Schemer". (The latter was truncated to "Scheme" due to a 6-character file name limit on an old operating system.) Finally, "Guile" suggests "guy-ell", or "Guy L. Steele", who, together with Gerald Sussman, originally discovered Scheme.

Around the same time that Guile (then GEL) was readying itself for public release, another extension language was gaining in popularity, Tcl. Many developers found advantages in Tcl because of its shell-like syntax and its well-developed graphical widgets library, Tk. Also, at the time there was a large marketing push promoting Tcl as a "universal extension language".

Richard Stallman, as the primary author of GNU Emacs, had a particular vision of what extension languages should be, and Tcl did not seem to him to be as capable as Emacs Lisp. He posted a criticism to the comp.lang.tcl newsgroup, sparking one of the internet's legendary flamewars. As part of these discussions, retrospectively dubbed the "Tcl Wars", he announced the Free Software Foundation's intent to promote Guile as the extension language for the GNU project.

It is a common misconception that Guile was created as a reaction to Tcl. While it is true that the public announcement of Guile happened at the same time as the "Tcl wars", Guile was created out of a condition that existed outside the polemic. Indeed, the need for a powerful language to bridge the gap between extension of existing applications and a more fully dynamic programming environment is still with us today.

9.1.3 A Scheme of Many Maintainers

Surveying the field, it seems that Scheme implementations correspond with their maintainers on an N-to-1 relationship. That is to say, that those people that implement Schemes might do so on a number of occasions, but that the lifetime of a given Scheme is tied to the maintainership of one individual.

Guile is atypical in this regard.

Tom Lord maintained Guile for its first year and a half or so, corresponding to the end of 1994 through the middle of 1996. The releases made in this time constitute an arc from SCM as a standalone program to Guile as a reusable, embeddable library, but passing through a explosion of features: embedded Tcl and Tk, a toolchain for compiling and disassembling Java, addition of a C-like syntax, creation of a module system, and a start at a rich POSIX interface.

Only some of those features remain in Guile. There were ongoing tensions between providing a small, embeddable language, and one which had all of the features (e.g. a graphical toolkit) that a modern Emacs might need. In the end, as Guile gained in uptake,

the development team decided to focus on depth, documentation and orthogonality rather than on breadth. This has been the focus of Guile ever since, although there is a wide range of third-party libraries for Guile.

Jim Blandy presided over that period of stabilization, in the three years until the end of 1999, when he too moved on to other projects. Since then, Guile has had a group maintainership. The first group was Maciej Stachowiak, Mikael Djurfeldt, and Marius Vollmer, with Vollmer staying on the longest. By late 2007, Marius had mostly moved on to other things, so Neil Jerram and Ludovic Courtès stepped up to take on the primary maintenance responsibility. Neil and Ludovic were joined by Andy Wingo in late 2009, allowing Neil to step away, and Mark Weaver joined shortly thereafter. After spending more than 5 years in the role, Mark stepped down as well, leaving Ludovic and Andy as the current co-maintainers of Guile as of January 2020.

Of course, a large part of the actual work on Guile has come from other contributors too numerous to mention, but without whom the world would be a poorer place.

9.1.4 A Timeline of Selected Guile Releases

guile-i — 4 February 1995
: SCM, turned into a library.

guile-ii — 6 April 1995
: A low-level module system was added. Tcl/Tk support was added, allowing extension of Scheme by Tcl or vice versa. POSIX support was improved, and there was an experimental stab at Java integration.

guile-iii — 18 August 1995
: The C-like syntax, ctax, was improved, but mostly this release featured a start at the task of breaking Guile into pieces.

1.0 — 5 January 1997
: `#f` was distinguished from `'()`. User-level, cooperative multi-threading was added. Source-level debugging became more useful, and programmer's and user's manuals were begun. The module system gained a high-level interface, which is still used today in more or less the same form.

1.1 — 16 May 1997
1.2 — 24 June 1997
: Support for Tcl/Tk and ctax were split off as separate packages, and have remained there since. Guile became more compatible with SCSH, and more useful as a UNIX scripting language. Libguile could now be built as a shared library, and third-party extensions written in C became loadable via dynamic linking.

1.3.0 — 19 October 1998
: Command-line editing became much more pleasant through the use of the readline library. The initial support for internationalization via multi-byte strings was removed; 10 years were to pass before proper internationalization would land again. Initial Emacs Lisp support landed, ports gained better support for file descriptors, and fluids were added.

1.3.2 — 20 August 1999
1.3.4 — 25 September 1999
1.4 — 21 June 2000

A long list of lispy features were added: hooks, Common Lisp's `format`, optional and keyword procedure arguments, `getopt-long`, sorting, random numbers, and many other fixes and enhancements. Guile also gained an interactive debugger, interactive help, and better backtraces.

1.6 — 6 September 2002

Guile gained support for the R5RS standard, and added a number of SRFI modules. The module system was expanded with programmatic support for identifier selection and renaming. The GOOPS object system was merged into Guile core.

1.8 — 20 February 2006

Guile's arbitrary-precision arithmetic switched to use the GMP library, and added support for exact rationals. Guile's embedded user-space threading was removed in favor of POSIX pre-emptive threads, providing true multiprocessing. Gettext support was added, and Guile's C API was cleaned up and orthogonalized in a massive way.

2.0 — 16 February 2010

A virtual machine was added to Guile, along with the associated compiler and toolchain. Support for internationalization was finally reimplemented, in terms of unicode, locales, and libunistring. Running Guile instances became controllable and debuggable from within Emacs, via Geiser. Guile caught up to features found in a number of other Schemes: SRFI-18 threads, module-hygienic macros, a profiler, tracer, and debugger, SSAX XML integration, bytevectors, a dynamic FFI, delimited continuations, module versions, and partial support for R6RS.

2.2 — 15 March 2017

The virtual machine and introduced in 2.0 was completely rewritten, along with much of the compiler and toolchain. This speeds up many Guile programs as well as reducing startup time and memory usage. Guile's POSIX multithreading was improved, stacks became dynamically expandable, the ports facility gained support for non-blocking I/O.

3.0 – January 2020

Guile gained support for native code generation via a simple just-in-time (JIT) compiler, further improving the speed of its virtual machine. The compiler itself gained a number of new optimizations: inlining of top-level bindings, better closure optimization, and better unboxing of integer and floating-point values. R7RS support was added, and R6RS support improved. The exception facility (throw and catch) was rewritten in terms of SRFI-34 exception handlers.

9.1.5 Status, or: Your Help Needed

Guile has achieved much of what it set out to achieve, but there is much remaining to do.

There is still the old problem of bringing existing applications into a more Emacs-like experience. Guile has had some successes in this respect, but still most applications in the GNU system are without Guile integration.

Getting Guile to those applications takes an investment, the "hacktivation energy" needed to wire Guile into a program that only pays off once it is good enough to enable new kinds of behavior. This would be a great way for new hackers to contribute: take an application that you use and that you know well, think of something that it can't yet do, and figure out a way to integrate Guile and implement that task in Guile.

With time, perhaps this exposure can reverse itself, whereby programs can run under Guile instead of vice versa, eventually resulting in the Emacsification of the entire GNU system. Indeed, this is the reason for the naming of the many Guile modules that live in the `ice-9` namespace, a nod to the fictional substance in Kurt Vonnegut's novel, Cat's Cradle, capable of acting as a seed crystal to crystallize the mass of software.

Implicit to this whole discussion is the idea that dynamic languages are somehow better than languages like C. While languages like C have their place, Guile's take on this question is that yes, Scheme is more expressive than C, and more fun to write. This realization carries an imperative with it to write as much code in Scheme as possible rather than in other languages.

These days it is possible to write extensible applications almost entirely from high-level languages, through byte-code and native compilation, speed gains in the underlying hardware, and foreign call interfaces in the high-level language. Smalltalk systems are like this, as are Common Lisp-based systems. While there already are a number of pure-Guile applications out there, in the past users have still needed to drop down to C for some tasks: interfacing to system libraries that don't have prebuilt Guile interfaces, and for some tasks requiring high performance. With the arrival of native code generation via a JIT compiler in Guile 3.0, most of these older applications can now be updated to move more C code to Scheme.

Still, even with an all-Guile application, sometimes you want to provide an opportunity for users to extend your program from a language with a syntax that is closer to C, or to Python. Another interesting idea to consider is compiling e.g. Python to Guile. It's not that far-fetched of an idea: see for example IronPython or JRuby.

Also, there's Emacs itself. Guile's Emacs Lisp support has reached an excellent level of correctness, robustness, and speed. However there is still work to do to finish its integration into Emacs itself. This will give lots of exciting things to Emacs: native threads, a real object system, more sophisticated types, cleaner syntax, and access to all of the Guile extensions.

Finally, so much of the world's computation is performed in web browsers that it makes sense to ask ourselves what the Guile-on-the-web-client story is. With the advent of WebAssembly, there may finally be a reasonable compilation target that's present on almost all user-exposed devices. Especially with the upcoming proposals to allow for tail calls, delimited continuations, and GC-managed objects, Scheme might once again have a place in the web browser. Get to it!

9.2 Data Representation

Scheme is a latently-typed language; this means that the system cannot, in general, determine the type of a given expression at compile time. Types only become apparent at run time. Variables do not have fixed types; a variable may hold a pair at one point, an integer at the next, and a thousand-element vector later. Instead, values, not variables, have fixed types.

In order to implement standard Scheme functions like `pair?` and `string?` and provide garbage collection, the representation of every value must contain enough information to accurately determine its type at run time. Often, Scheme systems also use this information to determine whether a program has attempted to apply an operation to an inappropriately typed value (such as taking the `car` of a string).

Because variables, pairs, and vectors may hold values of any type, Scheme implementations use a uniform representation for values — a single type large enough to hold either a complete value or a pointer to a complete value, along with the necessary typing information.

The following sections will present a simple typing system, and then make some refinements to correct its major weaknesses. We then conclude with a discussion of specific choices that Guile has made regarding garbage collection and data representation.

9.2.1 A Simple Representation

The simplest way to represent Scheme values in C would be to represent each value as a pointer to a structure containing a type indicator, followed by a union carrying the real value. Assuming that `SCM` is the name of our universal type, we can write:

```
enum type { integer, pair, string, vector, ... };

typedef struct value *SCM;

struct value {
  enum type type;
  union {
    int integer;
    struct { SCM car, cdr; } pair;
    struct { int length; char *elts; } string;
    struct { int length; SCM  *elts; } vector;
    ...
  } value;
};
```

with the ellipses replaced with code for the remaining Scheme types.

This representation is sufficient to implement all of Scheme's semantics. If x is an `SCM` value:

- To test if x is an integer, we can write *x*`->type == integer`.
- To find its value, we can write *x*`->value.integer`.
- To test if x is a vector, we can write *x*`->type == vector`.
- If we know x is a vector, we can write *x*`->value.vector.elts[0]` to refer to its first element.

- If we know *x* is a pair, we can write `x->value.pair.car` to extract its car.

9.2.2 Faster Integers

Unfortunately, the above representation has a serious disadvantage. In order to return an integer, an expression must allocate a `struct value`, initialize it to represent that integer, and return a pointer to it. Furthermore, fetching an integer's value requires a memory reference, which is much slower than a register reference on most processors. Since integers are extremely common, this representation is too costly, in both time and space. Integers should be very cheap to create and manipulate.

One possible solution comes from the observation that, on many architectures, heap-allocated data (i.e., what you get when you call `malloc`) must be aligned on an eight-byte boundary. (Whether or not the machine actually requires it, we can write our own allocator for `struct value` objects that assures this is true.) In this case, the lower three bits of the structure's address are known to be zero.

This gives us the room we need to provide an improved representation for integers. We make the following rules:

- If the lower three bits of an `SCM` value are zero, then the SCM value is a pointer to a `struct value`, and everything proceeds as before.
- Otherwise, the `SCM` value represents an integer, whose value appears in its upper bits.

Here is C code implementing this convention:

```
enum type { pair, string, vector, ... };

typedef struct value *SCM;

struct value {
  enum type type;
  union {
    struct { SCM car, cdr; } pair;
    struct { int length; char *elts; } string;
    struct { int length; SCM  *elts; } vector;
    ...
  } value;
};

#define POINTER_P(x) (((int) (x) & 7) == 0)
#define INTEGER_P(x) (! POINTER_P (x))

#define GET_INTEGER(x)  ((int) (x) >> 3)
#define MAKE_INTEGER(x) ((SCM) (((x) << 3) | 1))
```

Notice that `integer` no longer appears as an element of `enum type`, and the union has lost its `integer` member. Instead, we use the `POINTER_P` and `INTEGER_P` macros to make a coarse classification of values into integers and non-integers, and do further type testing as before.

Here's how we would answer the questions posed above (again, assume *x* is an `SCM` value):

- To test if *x* is an integer, we can write `INTEGER_P (x)`.

- To find its value, we can write `GET_INTEGER (x)`.
- To test if *x* is a vector, we can write:

  ```
  POINTER_P (x) && x->type == vector
  ```

  Given the new representation, we must make sure *x* is truly a pointer before we dereference it to determine its complete type.
- If we know *x* is a vector, we can write `x->value.vector.elts[0]` to refer to its first element, as before.
- If we know *x* is a pair, we can write `x->value.pair.car` to extract its car, just as before.

This representation allows us to operate more efficiently on integers than the first. For example, if *x* and *y* are known to be integers, we can compute their sum as follows:

```
MAKE_INTEGER (GET_INTEGER (x) + GET_INTEGER (y))
```

Now, integer math requires no allocation or memory references. Most real Scheme systems actually implement addition and other operations using an even more efficient algorithm, but this essay isn't about bit-twiddling. (Hint: how do you decide when to overflow to a bignum? How would you do it in assembly?)

9.2.3 Cheaper Pairs

However, there is yet another issue to confront. Most Scheme heaps contain more pairs than any other type of object; Jonathan Rees said at one point that pairs occupy 45% of the heap in his Scheme implementation, Scheme 48. However, our representation above spends three `SCM`-sized words per pair — one for the type, and two for the CAR and CDR. Is there any way to represent pairs using only two words?

Let us refine the convention we established earlier. Let us assert that:

- If the bottom three bits of an `SCM` value are `#b000`, then it is a pointer, as before.
- If the bottom three bits are `#b001`, then the upper bits are an integer. This is a bit more restrictive than before.
- If the bottom two bits are `#b010`, then the value, with the bottom three bits masked out, is the address of a pair.

Here is the new C code:

```
enum type { string, vector, ... };

typedef struct value *SCM;

struct value {
  enum type type;
  union {
    struct { int length; char *elts; } string;
    struct { int length; SCM  *elts; } vector;
    ...
  } value;
};
```

```
struct pair {
  SCM car, cdr;
};

#define POINTER_P(x) (((int) (x) & 7) == 0)

#define INTEGER_P(x)  (((int) (x) & 7) == 1)
#define GET_INTEGER(x)  ((int) (x) >> 3)
#define MAKE_INTEGER(x) ((SCM) (((x) << 3) | 1))

#define PAIR_P(x) (((int) (x) & 7) == 2)
#define GET_PAIR(x) ((struct pair *) ((int) (x) & ~7))
```

Notice that `enum type` and `struct value` now only contain provisions for vectors and strings; both integers and pairs have become special cases. The code above also assumes that an `int` is large enough to hold a pointer, which isn't generally true.

Our list of examples is now as follows:

- To test if x is an integer, we can write `INTEGER_P (x)`; this is as before.
- To find its value, we can write `GET_INTEGER (x)`, as before.
- To test if x is a vector, we can write:

  ```
  POINTER_P (x) && x->type == vector
  ```

  We must still make sure that x is a pointer to a `struct value` before dereferencing it to find its type.
- If we know x is a vector, we can write `x->value.vector.elts[0]` to refer to its first element, as before.
- We can write `PAIR_P (x)` to determine if x is a pair, and then write `GET_PAIR (x)->car` to refer to its car.

This change in representation reduces our heap size by 15%. It also makes it cheaper to decide if a value is a pair, because no memory references are necessary; it suffices to check the bottom two bits of the `SCM` value. This may be significant when traversing lists, a common activity in a Scheme system.

Again, most real Scheme systems use a slightly different implementation; for example, if GET_PAIR subtracts off the low bits of `x`, instead of masking them off, the optimizer will often be able to combine that subtraction with the addition of the offset of the structure member we are referencing, making a modified pointer as fast to use as an unmodified pointer.

9.2.4 Conservative Garbage Collection

Aside from the latent typing, the major source of constraints on a Scheme implementation's data representation is the garbage collector. The collector must be able to traverse every live object in the heap, to determine which objects are not live, and thus collectable.

There are many ways to implement this. Guile's garbage collection is built on a library, the Boehm-Demers-Weiser conservative garbage collector (BDW-GC). The BDW-GC "just works", for the most part. But since it is interesting to know how these things work, we include here a high-level description of what the BDW-GC does.

Garbage collection has two logical phases: a *mark* phase, in which the set of live objects is enumerated, and a *sweep* phase, in which objects not traversed in the mark phase are collected. Correct functioning of the collector depends on being able to traverse the entire set of live objects.

In the mark phase, the collector scans the system's global variables and the local variables on the stack to determine which objects are immediately accessible by the C code. It then scans those objects to find the objects they point to, and so on. The collector logically sets a *mark bit* on each object it finds, so each object is traversed only once.

When the collector can find no unmarked objects pointed to by marked objects, it assumes that any objects that are still unmarked will never be used by the program (since there is no path of dereferences from any global or local variable that reaches them) and deallocates them.

In the above paragraphs, we did not specify how the garbage collector finds the global and local variables; as usual, there are many different approaches. Frequently, the programmer must maintain a list of pointers to all global variables that refer to the heap, and another list (adjusted upon entry to and exit from each function) of local variables, for the collector's benefit.

The list of global variables is usually not too difficult to maintain, since global variables are relatively rare. However, an explicitly maintained list of local variables (in the author's personal experience) is a nightmare to maintain. Thus, the BDW-GC uses a technique called *conservative garbage collection*, to make the local variable list unnecessary.

The trick to conservative collection is to treat the C stack as an ordinary range of memory, and assume that *every* word on the C stack is a pointer into the heap. Thus, the collector marks all objects whose addresses appear anywhere in the C stack, without knowing for sure how that word is meant to be interpreted.

In addition to the stack, the BDW-GC will also scan static data sections. This means that global variables are also scanned when looking for live Scheme objects.

Obviously, such a system will occasionally retain objects that are actually garbage, and should be freed. In practice, this is not a problem, as the set of conservatively-scanned locations is fixed; the Scheme stack is maintained apart from the C stack, and is scanned precisely (as opposed to conservatively). The GC-managed heap is also partitioned into parts that can contain pointers (such as vectors) and parts that can't (such as bytevectors), limiting the potential for confusing a raw integer with a pointer to a live object.

Interested readers should see the BDW-GC web page at `http://www.hboehm.info/gc/`, for more information on conservative GC in general and the BDW-GC implementation in particular.

9.2.5 The SCM Type in Guile

Guile classifies Scheme objects into two kinds: those that fit entirely within an `SCM`, and those that require heap storage.

The former class are called *immediates*. The class of immediates includes small integers, characters, boolean values, the empty list, the mysterious end-of-file object, and some others.

The remaining types are called, not surprisingly, *non-immediates*. They include pairs, procedures, strings, vectors, and all other data types in Guile. For non-immediates, the `SCM`

word contains a pointer to data on the heap, with further information about the object in question is stored in that data.

This section describes how the SCM type is actually represented and used at the C level. Interested readers should see libguile/scm.h for an exposition of how Guile stores type information.

In fact, there are two basic C data types to represent objects in Guile: SCM and scm_t_bits.

9.2.5.1 Relationship Between SCM and scm_t_bits

A variable of type SCM is guaranteed to hold a valid Scheme object. A variable of type scm_t_bits, on the other hand, may hold a representation of a SCM value as a C integral type, but may also hold any C value, even if it does not correspond to a valid Scheme object.

For a variable x of type SCM, the Scheme object's type information is stored in a form that is not directly usable. To be able to work on the type encoding of the scheme value, the SCM variable has to be transformed into the corresponding representation as a scm_t_bits variable y by using the SCM_UNPACK macro. Once this has been done, the type of the scheme object x can be derived from the content of the bits of the scm_t_bits value y, in the way illustrated by the example earlier in this chapter (see Section 9.2.3 [Cheaper Pairs], page 820). Conversely, a valid bit encoding of a Scheme value as a scm_t_bits variable can be transformed into the corresponding SCM value using the SCM_PACK macro.

9.2.5.2 Immediate Objects

A Scheme object may either be an immediate, i.e. carrying all necessary information by itself, or it may contain a reference to a *heap object* which is, as the name implies, data on the heap. Although in general it should be irrelevant for user code whether an object is an immediate or not, within Guile's own code the distinction is sometimes of importance. Thus, the following low level macro is provided:

int SCM_IMP (*SCM x*) [Macro]

A Scheme object is an immediate if it fulfills the SCM_IMP predicate, otherwise it holds an encoded reference to a heap object. The result of the predicate is delivered as a C style boolean value. User code and code that extends Guile should normally not be required to use this macro.

Summary:

- Given a Scheme object x of unknown type, check first with SCM_IMP (x) if it is an immediate object.
- If so, all of the type and value information can be determined from the scm_t_bits value that is delivered by SCM_UNPACK (x).

There are a number of special values in Scheme, most of them documented elsewhere in this manual. It's not quite the right place to put them, but for now, here's a list of the C names given to some of these values:

SCM SCM_EOL [Macro]

The Scheme empty list object, or "End Of List" object, usually written in Scheme as '().

`SCM SCM_EOF_VAL` [Macro]
The Scheme end-of-file value. It has no standard written representation, for obvious reasons.

`SCM SCM_UNSPECIFIED` [Macro]
The value returned by some (but not all) expressions that the Scheme standard says return an "unspecified" value.

This is sort of a weirdly literal way to take things, but the standard read-eval-print loop prints nothing when the expression returns this value, so it's not a bad idea to return this when you can't think of anything else helpful.

`SCM SCM_UNDEFINED` [Macro]
The "undefined" value. Its most important property is that is not equal to any valid Scheme value. This is put to various internal uses by C code interacting with Guile.

For example, when you write a C function that is callable from Scheme and which takes optional arguments, the interpreter passes `SCM_UNDEFINED` for any arguments you did not receive.

We also use this to mark unbound variables.

`int SCM_UNBNDP (`*SCM x*`)` [Macro]
Return true if *x* is `SCM_UNDEFINED`. Note that this is not a check to see if *x* is `SCM_UNBOUND`. History will not be kind to us.

9.2.5.3 Non-Immediate Objects

A Scheme object of type `SCM` that does not fulfill the `SCM_IMP` predicate holds an encoded reference to a heap object. This reference can be decoded to a C pointer to a heap object using the `SCM_UNPACK_POINTER` macro. The encoding of a pointer to a heap object into a `SCM` value is done using the `SCM_PACK_POINTER` macro.

Before Guile 2.0, Guile had a custom garbage collector that allocated heap objects in units of 2-word *cells*. With the move to the BDW-GC collector in Guile 2.0, Guile can allocate heap objects of any size, and the concept of a cell is now obsolete. Still, we mention it here as the name still appears in various low-level interfaces.

`scm_t_bits * SCM_UNPACK_POINTER (`*SCM x*`)` [Macro]
`scm_t_cell * SCM2PTR (`*SCM x*`)` [Macro]
Extract and return the heap object pointer from a non-immediate `SCM` object *x*. The name `SCM2PTR` is deprecated but still common.

`SCM_PACK_POINTER (scm_t_bits *` *x*`)` [Macro]
`SCM PTR2SCM (`*scm_t_cell ** `x`*`)` [Macro]
Return a `SCM` value that encodes a reference to the heap object pointer *x*. The name `PTR2SCM` is deprecated but still common.

Note that it is also possible to transform a non-immediate `SCM` value by using `SCM_UNPACK` into a `scm_t_bits` variable. However, the result of `SCM_UNPACK` may not be used as a pointer to a heap object: only `SCM_UNPACK_POINTER` is guaranteed to transform a `SCM` object into a valid pointer to a heap object. Also, it is not allowed to apply `SCM_PACK_POINTER` to anything that is not a valid pointer to a heap object.

Summary:

- Only use `SCM_UNPACK_POINTER` on `SCM` values for which `SCM_IMP` is false!
- Don't use `(scm_t_cell *) SCM_UNPACK (x)`! Use `SCM_UNPACK_POINTER (x)` instead!
- Don't use `SCM_PACK_POINTER` for anything but a heap object pointer!

9.2.5.4 Allocating Heap Objects

Heap objects are heap-allocated data pointed to by non-immediate `SCM` value. The first word of the heap object should contain a type code. The object may be any number of words in length, and is generally scanned by the garbage collector for additional unless the object was allocated using a "pointerless" allocation function.

You should generally not need these functions, unless you are implementing a new data type, and thoroughly understand the code in `<libguile/scm.h>`.

If you just want to allocate pairs, use `scm_cons`.

`SCM scm_words` (*scm_t_bits word_0, uint32_t n_words*) [Function]
Allocate a new heap object containing *n_words*, and initialize the first slot to *word_0*, and return a non-immediate `SCM` value encoding a pointer to the object. Typically *word_0* will contain the type tag.

There are also deprecated but common variants of `scm_words` that use the term "cell" to indicate 2-word objects.

`SCM scm_cell` (*scm_t_bits word_0, scm_t_bits word_1*) [Function]
Allocate a new 2-word heap object, initialize the two slots with *word_0* and *word_1*, and return it. Just like calling `scm_words (word_0, 2)`, then initializing the second slot to *word_1*.

Note that *word_0* and *word_1* are of type `scm_t_bits`. If you want to pass a `SCM` object, you need to use `SCM_UNPACK`.

`SCM scm_double_cell` (*scm_t_bits word_0, scm_t_bits word_1, scm_t_bits word_2, scm_t_bits word_3*) [Function]
Like `scm_cell`, but allocates a 4-word heap object.

9.2.5.5 Heap Object Type Information

Heap objects contain a type tag and are followed by a number of word-sized slots. The interpretation of the object contents depends on the type of the object.

`scm_t_bits SCM_CELL_TYPE` (*SCM x*) [Macro]
Extract the first word of the heap object pointed to by *x*. This value holds the information about the cell type.

`void SCM_SET_CELL_TYPE` (*SCM x, scm_t_bits t*) [Macro]
For a non-immediate Scheme object *x*, write the value *t* into the first word of the heap object referenced by *x*. The value *t* must hold a valid cell type.

9.2.5.6 Accessing Heap Object Fields

For a non-immediate Scheme object *x*, the object type can be determined by using the `SCM_CELL_TYPE` macro described in the previous section. For each different type of heap object it is known which fields hold tagged Scheme objects and which fields hold untagged raw data. To access the different fields appropriately, the following macros are provided.

`scm_t_bits SCM_CELL_WORD (`*SCM* `x`, *unsigned int* `n``)` [Macro]
`scm_t_bits SCM_CELL_WORD_0 (x)` [Macro]
`scm_t_bits SCM_CELL_WORD_1 (x)` [Macro]
`scm_t_bits SCM_CELL_WORD_2 (x)` [Macro]
`scm_t_bits SCM_CELL_WORD_3 (x)` [Macro]

Deliver the field *n* of the heap object referenced by the non-immediate Scheme object *x* as raw untagged data. Only use this macro for fields containing untagged data; don't use it for fields containing tagged `SCM` objects.

`SCM SCM_CELL_OBJECT (`*SCM* `x`, *unsigned int* `n``)` [Macro]
`SCM SCM_CELL_OBJECT_0 (`*SCM* `x`)` [Macro]
`SCM SCM_CELL_OBJECT_1 (`*SCM* `x`)` [Macro]
`SCM SCM_CELL_OBJECT_2 (`*SCM* `x`)` [Macro]
`SCM SCM_CELL_OBJECT_3 (`*SCM* `x`)` [Macro]

Deliver the field *n* of the heap object referenced by the non-immediate Scheme object *x* as a Scheme object. Only use this macro for fields containing tagged `SCM` objects; don't use it for fields containing untagged data.

`void SCM_SET_CELL_WORD (`*SCM* `x`, *unsigned int* `n`, *scm_t_bits* `w``)` [Macro]
`void SCM_SET_CELL_WORD_0 (x, w)` [Macro]
`void SCM_SET_CELL_WORD_1 (x, w)` [Macro]
`void SCM_SET_CELL_WORD_2 (x, w)` [Macro]
`void SCM_SET_CELL_WORD_3 (x, w)` [Macro]

Write the raw value *w* into field number *n* of the heap object referenced by the non-immediate Scheme value *x*. Values that are written into heap objects as raw values should only be read later using the `SCM_CELL_WORD` macros.

`void SCM_SET_CELL_OBJECT (`*SCM* `x`, *unsigned int* `n`, *SCM* `o`)` [Macro]
`void SCM_SET_CELL_OBJECT_0 (`*SCM* `x`, *SCM* `o`)` [Macro]
`void SCM_SET_CELL_OBJECT_1 (`*SCM* `x`, *SCM* `o`)` [Macro]
`void SCM_SET_CELL_OBJECT_2 (`*SCM* `x`, *SCM* `o`)` [Macro]
`void SCM_SET_CELL_OBJECT_3 (`*SCM* `x`, *SCM* `o`)` [Macro]

Write the Scheme object *o* into field number *n* of the heap object referenced by the non-immediate Scheme value *x*. Values that are written into heap objects as objects should only be read using the `SCM_CELL_OBJECT` macros.

Summary:

- For a non-immediate Scheme object *x* of unknown type, get the type information by using `SCM_CELL_TYPE (x)`.
- As soon as the type information is available, only use the appropriate access methods to read and write data to the different heap object fields.

- Note that field 0 stores the cell type information. Generally speaking, other data associated with a heap object is stored starting from field 1.

9.3 A Virtual Machine for Guile

Enough about data—how does Guile run code?

Code is a grammatical production of a language. Sometimes these languages are implemented using interpreters: programs that run along-side the program being interpreted, dynamically translating the high-level code to low-level code. Sometimes these languages are implemented using compilers: programs that translate high-level programs to equivalent low-level code, and pass on that low-level code to some other language implementation. Each of these languages can be thought to be virtual machines: they offer programs an abstract machine on which to run.

Guile implements a number of interpreters and compilers on different language levels. For example, there is an interpreter for the Scheme language that is itself implemented as a Scheme program compiled to a bytecode for a low-level virtual machine shipped with Guile. That virtual machine is implemented by both an interpreter—a C program that interprets the bytecodes—and a compiler—a C program that dynamically translates bytecode programs to native machine code[2].

This section describes the language implemented by Guile's bytecode virtual machine, as well as some examples of translations of Scheme programs to Guile's VM.

9.3.1 Why a VM?

For a long time, Guile only had a Scheme interpreter, implemented in C. Guile's interpreter operated directly on the S-expression representation of Scheme source code.

But while the interpreter was highly optimized and hand-tuned, it still performed many needless computations during the course of evaluating a Scheme expression. For example, application of a function to arguments needlessly consed up the arguments in a list. Evaluation of an expression like `(f x y)` always had to figure out whether f was a procedure, or a special form like `if`, or something else. The interpreter represented the lexical environment as a heap data structure, so every evaluation caused allocation, which was of course slow. Et cetera.

The solution to the slow-interpreter problem was to compile the higher-level language, Scheme, into a lower-level language for which all of the checks and dispatching have already been done—the code is instead stripped to the bare minimum needed to "do the job".

The question becomes then, what low-level language to choose? There are many options. We could compile to native code directly, but that poses portability problems for Guile, as it is a highly cross-platform project.

So we want the performance gains that compilation provides, but we also want to maintain the portability benefits of a single code path. The obvious solution is to compile to a virtual machine that is present on all Guile installations.

The easiest (and most fun) way to depend on a virtual machine is to implement the virtual machine within Guile itself. Guile contains a bytecode interpreter (written in C) and

[2] Even the lowest-level machine code can be thought to be interpreted by the CPU, and indeed is often implemented by compiling machine instructions to "micro-operations".

a Scheme to bytecode compiler (written in Scheme). This way the virtual machine provides what Scheme needs (tail calls, multiple values, `call/cc`) and can provide optimized inline instructions for Guile as well (GC-managed allocations, type checks, etc.).

Guile also includes a just-in-time (JIT) compiler to translate bytecode to native code. Because Guile embeds a portable code generation library (`https://gitlab.com/wingo/lightening`), we keep the benefits of portability while also benefitting from fast native code. To avoid too much time spent in the JIT compiler itself, Guile is tuned to only emit machine code for bytecode that is called often.

The rest of this section describes that VM that Guile implements, and the compiled procedures that run on it.

Before moving on, though, we should note that though we spoke of the interpreter in the past tense, Guile still has an interpreter. The difference is that before, it was Guile's main Scheme implementation, and so was implemented in highly optimized C; now, it is actually implemented in Scheme, and compiled down to VM bytecode, just like any other program. (There is still a C interpreter around, used to bootstrap the compiler, but it is not normally used at runtime.)

The upside of implementing the interpreter in Scheme is that we preserve tail calls and multiple-value handling between interpreted and compiled code, and with advent of the JIT compiler in Guile 3.0 we reach the speed of the old hand-tuned C implementation; it's the best of both worlds.

Also note that this decision to implement a bytecode compiler does not preclude ahead-of-time native compilation. More possibilities are discussed in Section 9.4.7 [Extending the Compiler], page 875.

9.3.2 VM Concepts

The bytecode in a Scheme procedure is interpreted by a virtual machine (VM). Each thread has its own instantiation of the VM. The virtual machine executes the sequence of instructions in a procedure.

Each VM instruction starts by indicating which operation it is, and then follows by encoding its source and destination operands. Each procedure declares that it has some number of local variables, including the function arguments. These local variables form the available operands of the procedure, and are accessed by index.

The local variables for a procedure are stored on a stack. Calling a procedure typically enlarges the stack, and returning from a procedure shrinks it. Stack memory is exclusive to the virtual machine that owns it.

In addition to their stacks, virtual machines also have access to the global memory (modules, global bindings, etc) that is shared among other parts of Guile, including other VMs.

The registers that a VM has are as follows:

- ip - Instruction pointer
- sp - Stack pointer
- fp - Frame pointer

In other architectures, the instruction pointer is sometimes called the "program counter" (pc). This set of registers is pretty typical for virtual machines; their exact meanings in the context of Guile's VM are described in the next section.

9.3.3 Stack Layout

The stack of Guile's virtual machine is composed of *frames*. Each frame corresponds to the application of one compiled procedure, and contains storage space for arguments, local variables, and some bookkeeping information (such as what to do after the frame is finished).

While the compiler is free to do whatever it wants to, as long as the semantics of a computation are preserved, in practice every time you call a function, a new frame is created. (The notable exception of course is the tail call case, see Section 3.3.2 [Tail Calls], page 24.)

The structure of the top stack frame is as follows:

```
| ...previous frame locals...  |
+==============================+ <- fp + 3
| Dynamic link                 |
+------------------------------+
| Virtual return address (vRA) |
+------------------------------+
| Machine return address (mRA) |
+==============================+ <- fp
| Local 0                      |
+------------------------------+
| Local 1                      |
+------------------------------+
| ...                          |
+------------------------------+
| Local N-1                    |
\------------------------------/ <- sp
```

In the above drawing, the stack grows downward. At the beginning of a function call, the procedure being applied is in local 0, followed by the arguments from local 1. After the procedure checks that it is being passed a compatible set of arguments, the procedure allocates some additional space in the frame to hold variables local to the function.

Note that once a value in a local variable slot is no longer needed, Guile is free to re-use that slot. This applies to the slots that were initially used for the callee and arguments, too. For this reason, backtraces in Guile aren't always able to show all of the arguments: it could be that the slot corresponding to that argument was re-used by some other variable.

The *virtual return address* is the `ip` that was in effect before this program was applied. When we return from this activation frame, we will jump back to this `ip`. Likewise, the *dynamic link* is the offset of the `fp` that was in effect before this program was applied, relative to the current `fp`.

There are two return addresses: the virtual return address (vRA), and the machine return address (mRA). The vRA is always present and indicates a bytecode address. The mRA is only present when a call is made from a function with machine code (e.g. a function that has been JIT-compiled).

To prepare for a non-tail application, Guile's VM will emit code that shuffles the function to apply and its arguments into appropriate stack slots, with three free slots below them. The call then initializes those free slots to hold the machine return address (or NULL), the virtual return address, and the offset to the previous frame pointer (`fp`). It then gets the `ip` for the function being called and adjusts `fp` to point to the new call frame.

In this way, the dynamic link links the current frame to the previous frame. Computing a stack trace involves traversing these frames.

Each stack local in Guile is 64 bits wide, even on 32-bit architectures. This allows Guile to preserve its uniform treatment of stack locals while allowing for unboxed arithmetic on 64-bit integers and floating-point numbers. See Section 9.3.7 [Instruction Set], page 835, for more on unboxed arithmetic.

As an implementation detail, we actually store the dynamic link as an offset and not an absolute value because the stack can move at runtime as it expands or during partial continuation calls. If it were an absolute value, we would have to walk the frames, relocating frame pointers.

9.3.4 Variables and the VM

Consider the following Scheme code as an example:

```
(define (foo a)
  (lambda (b) (vector foo a b)))
```

Within the lambda expression, `foo` is a top-level variable, `a` is a lexically captured variable, and `b` is a local variable.

Another way to refer to `a` and `b` is to say that `a` is a "free" variable, since it is not defined within the lambda, and `b` is a "bound" variable. These are the terms used in the *lambda calculus*, a mathematical notation for describing functions. The lambda calculus is useful because it is a language in which to reason precisely about functions and variables. It is especially good at describing scope relations, and it is for that reason that we mention it here.

Guile allocates all variables on the stack. When a lexically enclosed procedure with free variables—a *closure*—is created, it copies those variables into its free variable vector. References to free variables are then redirected through the free variable vector.

If a variable is ever `set!`, however, it will need to be heap-allocated instead of stack-allocated, so that different closures that capture the same variable can see the same value. Also, this allows continuations to capture a reference to the variable, instead of to its value at one point in time. For these reasons, `set!` variables are allocated in "boxes"—actually, in variable cells. See Section 6.20.7 [Variables], page 419, for more information. References to `set!` variables are indirected through the boxes.

Thus perhaps counterintuitively, what would seem "closer to the metal", viz `set!`, actually forces an extra memory allocation and indirection. Sometimes Guile's optimizer can remove this allocation, but not always.

Going back to our example, `b` may be allocated on the stack, as it is never mutated.

`a` may also be allocated on the stack, as it too is never mutated. Within the enclosed lambda, its value will be copied into (and referenced from) the free variables vector.

`foo` is a top-level variable, because `foo` is not lexically bound in this example.

9.3.5 Compiled Procedures are VM Programs

By default, when you enter in expressions at Guile's REPL, they are first compiled to bytecode. Then that bytecode is executed to produce a value. If the expression evaluates to a procedure, the result of this process is a compiled procedure.

A compiled procedure is a compound object consisting of its bytecode and a reference to any captured lexical variables. In addition, when a procedure is compiled, it has associated metadata written to side tables, for instance a line number mapping, or its docstring. You can pick apart these pieces with the accessors in `(system vm program)`. See Section 6.9.3 [Compiled Procedures], page 250, for a full API reference.

A procedure may reference data that was statically allocated when the procedure was compiled. For example, a pair of immediate objects (see Section 9.2.5.2 [Immediate Objects], page 823) can be allocated directly in the memory segment that contains the compiled bytecode, and accessed directly by the bytecode.

Another use for statically allocated data is to serve as a cache for a bytecode. Top-level variable lookups are handled in this way; the first time a top-level binding is referenced, the resolved variable will be stored in a cache. Thereafter all access to the variable goes through the cache cell. The variable's value may change in the future, but the variable itself will not.

We can see how these concepts tie together by disassembling the `foo` function we defined earlier to see what is going on:

```
scheme@(guile-user)> (define (foo a) (lambda (b) (vector foo a b)))
scheme@(guile-user)> ,x foo
Disassembly of #<procedure foo (a)> at #xf1da30:

   0    (instrument-entry 164)                              at (unknown file):5:0
   2    (assert-nargs-ee/locals 2 1)    ;; 3 slots (1 arg)
   3    (allocate-words/immediate 2 3)                      at (unknown file):5:16
   4    (load-u64 0 0 65605)
   7    (word-set!/immediate 2 0 0)
   8    (load-label 0 7)                ;; anonymous procedure at #xf1da6c
  10    (word-set!/immediate 2 1 0)
  11    (scm-set!/immediate 2 2 1)
  12    (reset-frame 1)                 ;; 1 slot
  13    (handle-interrupts)
  14    (return-values)

----------------------------------------
Disassembly of anonymous procedure at #xf1da6c:

   0    (instrument-entry 183)                              at (unknown file):5:16
   2    (assert-nargs-ee/locals 2 3)    ;; 5 slots (1 arg)
   3    (static-ref 2 152)              ;; #<variable 112e530 value: #<procedure foo (a)>>
   5    (immediate-tag=? 2 7 0)         ;; heap-object?
   7    (je 19)                         ;; -> L2
   8    (static-ref 2 119)              ;; #<directory (guile-user) ca9750>
  10    (static-ref 1 127)              ;; foo
  12    (call-scm<-scm-scm 2 2 1 40)
  14    (immediate-tag=? 2 7 0)         ;; heap-object?
  16    (jne 8)                         ;; -> L1
  17    (scm-ref/immediate 0 2 1)
  18    (immediate-tag=? 0 4095 2308)   ;; undefined?
  20    (je 4)                          ;; -> L1
```

```
   21    (static-set! 2 134)             ;; #<variable 112e530 value: #<procedure foo (a)>>
   23    (j 3)                           ;; -> L2
L1:
   24    (throw/value 1 151)             ;; #(unbound-variable #f "Unbound variable: ~S")
L2:
   26    (scm-ref/immediate 2 2 1)
   27    (allocate-words/immediate 1 4)                        at (unknown file):5:28
   28    (load-u64 0 0 781)
   31    (word-set!/immediate 1 0 0)
   32    (scm-set!/immediate 1 1 2)
   33    (scm-ref/immediate 4 4 2)
   34    (scm-set!/immediate 1 2 4)
   35    (scm-set!/immediate 1 3 3)
   36    (mov 4 1)
   37    (reset-frame 1)                 ;; 1 slot
   38    (handle-interrupts)
   39    (return-values)
```

The first thing to notice is that the bytecode is at a fairly low level. When a program is compiled from Scheme to bytecode, it is expressed in terms of more primitive operations. As such, there can be more instructions than you might expect.

The first chunk of instructions is the outer `foo` procedure. It is followed by the code for the contained closure. The code can look daunting at first glance, but with practice it quickly becomes comprehensible, and indeed being able to read bytecode is an important step to understanding the low-level performance of Guile programs.

The `foo` function begins with a prelude. The `instrument-entry` bytecode increments a counter associated with the function. If the counter reaches a certain threshold, Guile will emit machine code ("JIT-compile") for `foo`. Emitting machine code is fairly cheap but it does take time, so it's not something you want to do for every function. Using a per-function counter and a global threshold allows Guile to spend time JIT-compiling only the "hot" functions.

Next in the prelude is an argument-checking instruction, which checks that it was called with only 1 argument (plus the callee function itself makes 2) and then reserves stack space for an additional 1 local.

Then from `ip` 3 to 11, we allocate a new closure by allocating a three-word object, initializing its first word to store a type tag, setting its second word to its code pointer, and finally at `ip` 11, storing local value 1 (the `a` argument) into the third word (the first free variable).

Before returning, `foo` "resets the frame" to hold only one local (the return value), runs any pending interrupts (see Section 6.22.3 [Asyncs], page 445) and then returns.

Note that local variables in Guile's virtual machine are usually addressed relative to the stack pointer, which leads to a pleasantly efficient `sp[n]` access. However it can make the disassembly hard to read, because the `sp` can change during the function, and because incoming arguments are relative to the `fp`, not the `sp`.

To know what `fp`-relative slot corresponds to an `sp`-relative reference, scan up in the disassembly until you get to a "*n* slots" annotation; in our case, 3, indicating that the frame has space for 3 slots. Thus a zero-indexed `sp`-relative slot of 2 corresponds to the `fp`-relative slot of 0, which initially held the value of the closure being called. This means that Guile doesn't need the value of the closure to compute its result, and so slot 0 was free for re-use, in this case for the result of making a new closure.

A closure is code with data. As you can see, making the closure involved making an object (`ip` 3), putting a code pointer in it (`ip` 8 and 10), and putting in the closure's free variable (`ip` 11).

The second stanza disassembles the code for the closure. After the prelude, all of the code between `ip` 5 and 24 is related to loading the toplevel variable `foo` into slot 1. This lookup happens only once, and is associated with a cache; after the first run, the value in the cache will be a bound variable, and the code will jump from `ip` 7 to 26. On the first run, Guile gets the module associated with the function, calls out to a run-time routine to look up the variable, and checks that the variable is bound before initializing the cache. Either way, `ip` 26 dereferences the variable into local 2.

What follows is the allocation and initialization of the vector return value. `Ip` 27 does the allocation, and the following two instructions initialize the type-and-length tag for the object's first word. `Ip` 32 sets word 1 of the object (the first vector slot) to the value of `foo`; `ip` 33 fetches the closure variable for `a`, then in `ip` 34 stores it in the second vector slot; and finally, in `ip` 35, local `b` is stored to the third vector slot. This is followed by the return sequence.

9.3.6 Object File Format

To compile a file to disk, we need a format in which to write the compiled code to disk, and later load it into Guile. A good *object file format* has a number of characteristics:

- Above all else, it should be very cheap to load a compiled file.
- It should be possible to statically allocate constants in the file. For example, a bytevector literal in source code can be emitted directly into the object file.
- The compiled file should enable maximum code and data sharing between different processes.
- The compiled file should contain debugging information, such as line numbers, but that information should be separated from the code itself. It should be possible to strip debugging information if space is tight.

These characteristics are not specific to Scheme. Indeed, mainstream languages like C and C++ have solved this issue many times in the past. Guile builds on their work by adopting ELF, the object file format of GNU and other Unix-like systems, as its object file format. Although Guile uses ELF on all platforms, we do not use platform support for ELF. Guile implements its own linker and loader. The advantage of using ELF is not sharing code, but sharing ideas. ELF is simply a well-designed object file format.

An ELF file has two meta-tables describing its contents. The first meta-table is for the loader, and is called the *program table* or sometimes the *segment table*. The program table divides the file into big chunks that should be treated differently by the loader. Mostly the difference between these *segments* is their permissions.

Typically all segments of an ELF file are marked as read-only, except that part that represents modifiable static data or static data that needs load-time initialization. Loading an ELF file is as simple as mmapping the thing into memory with read-only permissions, then using the segment table to mark a small sub-region of the file as writable. This writable section is typically added to the root set of the garbage collector as well.

One ELF segment is marked as "dynamic", meaning that it has data of interest to the loader. Guile uses this segment to record the Guile version corresponding to this file. There

is also an entry in the dynamic segment that points to the address of an initialization thunk that is run to perform any needed link-time initialization. (This is like dynamic relocations for normal ELF shared objects, except that we compile the relocations as a procedure instead of having the loader interpret a table of relocations.) Finally, the dynamic segment marks the location of the "entry thunk" of the object file. This thunk is returned to the caller of `load-thunk-from-memory` or `load-thunk-from-file`. When called, it will execute the "body" of the compiled expression.

The other meta-table in an ELF file is the *section table*. Whereas the program table divides an ELF file into big chunks for the loader, the section table specifies small sections for use by introspective tools like debuggers or the like. One segment (program table entry) typically contains many sections. There may be sections outside of any segment, as well.

Typical sections in a Guile `.go` file include:

`.rtl-text`
: Bytecode.

`.data`
: Data that needs initialization, or which may be modified at runtime.

`.rodata`
: Statically allocated data that needs no run-time initialization, and which therefore can be shared between processes.

`.dynamic`
: The dynamic section, discussed above.

`.symtab`
`.strtab`
: A table mapping addresses in the `.rtl-text` to procedure names. `.strtab` is used by `.symtab`.

`.guile.procprops`
`.guile.arities`
`.guile.arities.strtab`
`.guile.docstrs`
`.guile.docstrs.strtab`
: Side tables of procedure properties, arities, and docstrings.

`.guile.docstrs.strtab`
: Side table of frame maps, describing the set of live slots for ever return point in the program text, and whether those slots are pointers are not. Used by the garbage collector.

`.debug_info`
`.debug_abbrev`
`.debug_str`
`.debug_loc`
`.debug_line`
: Debugging information, in DWARF format. See the DWARF specification, for more information.

`.shstrtab`
: Section name string table.

For more information, see the elf(5) man page. See the DWARF specification (`http://dwarfstd.org/`) for more on the DWARF debugging format. Or if you are an adventurous explorer, try running `readelf` or `objdump` on compiled `.go` files. It's good times!

9.3.7 Instruction Set

There are currently about 150 instructions in Guile's virtual machine. These instructions represent atomic units of a program's execution. Ideally, they perform one task without conditional branches, then dispatch to the next instruction in the stream.

Instructions themselves are composed of 1 or more 32-bit units. The low 8 bits of the first word indicate the opcode, and the rest of instruction describe the operands. There are a number of different ways operands can be encoded.

`sn` An unsigned *n*-bit integer, indicating the `sp`-relative index of a local variable.

`fn` An unsigned *n*-bit integer, indicating the `fp`-relative index of a local variable. Used when a continuation accepts a variable number of values, to shuffle received values into known locations in the frame.

`cn` An unsigned *n*-bit integer, indicating a constant value.

`l24` An offset from the current `ip`, in 32-bit units, as a signed 24-bit value. Indicates a bytecode address, for a relative jump.

`i16`
`i32` An immediate Scheme value (see Section 9.2.5.2 [Immediate Objects], page 823), encoded directly in 16 or 32 bits.

`a32`
`b32` An immediate Scheme value, encoded as a pair of 32-bit words. `a32` and `b32` values always go together on the same opcode, and indicate the high and low bits, respectively. Normally only used on 64-bit systems.

`n32` A statically allocated non-immediate. The address of the non-immediate is encoded as a signed 32-bit integer, and indicates a relative offset in 32-bit units. Think of it as `SCM x = ip + offset`.

`r32` Indirect scheme value, like `n32` but indirected. Think of it as `SCM *x = ip + offset`.

`l32`

`lo32` An ip-relative address, as a signed 32-bit integer. Could indicate a bytecode address, as in `make-closure`, or a non-immediate address, as with `static-patch!`.

`l32` and `lo32` are the same from the perspective of the virtual machine. The difference is that an assembler might want to allow an `lo32` address to be specified as a label and then some number of words offset from that label, for example when patching a field of a statically allocated object.

`b1` A boolean value: 1 for true, otherwise 0.

`xn` An ignored sequence of *n* bits.

An instruction is specified by giving its name, then describing its operands. The operands are packed by 32-bit words, with earlier operands occupying the lower bits.

For example, consider the following instruction specification:

`call` *f24:`proc` x8:_ c24:`nlocals`* [Instruction]
The first word in the instruction will start with the 8-bit value corresponding to the *call* opcode in the low bits, followed by *proc* as a 24-bit value. The second word starts with 8 dead bits, followed by the index as a 24-bit immediate value.

For instructions with operands that encode references to the stack, the interpretation of those stack values is up to the instruction itself. Most instructions expect their operands to be tagged SCM values (`scm` representation), but some instructions expect unboxed integers (`u64` and `s64` representations) or floating-point numbers (`f64` representation). It is assumed that the bits for a `u64` value are the same as those for an `s64` value, and that `s64` values are stored in two's complement.

Instructions have static types: they must receive their operands in the format they expect. It's up to the compiler to ensure this is the case.

Unless otherwise mentioned, all operands and results are in the `scm` representation.

9.3.7.1 Call and Return Instructions

As described earlier (see Section 9.3.3 [Stack Layout], page 829), Guile's calling convention is that arguments are passed and values returned on the stack.

For calls, both in tail position and in non-tail position, we require that the procedure and the arguments already be shuffled into place before the call instruction. "Into place" for a tail call means that the procedure should be in slot 0, relative to the `fp`, and the arguments should follow. For a non-tail call, if the procedure is in `fp`-relative slot *n*, the arguments should follow from slot *n*+1, and there should be three free slots between *n*-1 and *n*-3 in which to save the mRA, vRA, and `fp`.

Returning values is similar. Multiple-value returns should have values already shuffled down to start from `fp`-relative slot 0 before emitting `return-values`.

In both calls and returns, the `sp` is used to indicate to the callee or caller the number of arguments or return values, respectively. After receiving return values, it is the caller's responsibility to *restore the frame* by resetting the `sp` to its former value.

`call` *f24:`proc` x8:_ c24:`nlocals`* [Instruction]
Call a procedure. *proc* is the local corresponding to a procedure. The three values below *proc* will be overwritten by the saved call frame data. The new frame will have space for *nlocals* locals: one for the procedure, and the rest for the arguments which should already have been pushed on.

When the call returns, execution proceeds with the next instruction. There may be any number of values on the return stack; the precise number can be had by subtracting the address of *proc*-1 from the post-call `sp`.

`call-label` *f24:`proc` x8:_ c24:`nlocals` l32:`label`* [Instruction]
Call a procedure in the same compilation unit.

This instruction is just like `call`, except that instead of dereferencing *proc* to find the call target, the call target is known to be at *label*, a signed 32-bit offset in 32-bit units from the current `ip`. Since *proc* is not dereferenced, it may be some other representation of the closure.

`tail-call` *x24:_* [Instruction]
Tail-call a procedure. Requires that the procedure and all of the arguments have already been shuffled into position, and that the frame has already been reset to the number of arguments to the call.

`tail-call-label` *x24:_ l32:*`label` [Instruction]
Tail-call a known procedure. As `call` is to `call-label`, `tail-call` is to `tail-call-label`.

`return-values` *x24:_* [Instruction]
Return a number of values from a call frame. The return values should have already been shuffled down to a contiguous array starting at slot 0, and the frame already reset.

`receive` *f12:*`dst` *f12:*`proc` *x8:_ c24:*`nlocals` [Instruction]
Receive a single return value from a call whose procedure was in *proc*, asserting that the call actually returned at least one value. Afterwards, resets the frame to *nlocals* locals.

`receive-values` *f24:*`proc` *b1:*`allow-extra?` *x7:_ c24:*`nvalues` [Instruction]
Receive a return of multiple values from a call whose procedure was in *proc*. If fewer than *nvalues* values were returned, signal an error. Unless *allow-extra?* is true, require that the number of return values equals *nvalues* exactly. After `receive-values` has run, the values can be copied down via `mov`, or used in place.

9.3.7.2 Function Prologue Instructions

A function call in Guile is very cheap: the VM simply hands control to the procedure. The procedure itself is responsible for asserting that it has been passed an appropriate number of arguments. This strategy allows arbitrarily complex argument parsing idioms to be developed, without harming the common case.

For example, only calls to keyword-argument procedures "pay" for the cost of parsing keyword arguments. (At the time of this writing, calling procedures with keyword arguments is typically two to four times as costly as calling procedures with a fixed set of arguments.)

`assert-nargs-ee` *c24:*`expected` [Instruction]
`assert-nargs-ge` *c24:*`expected` [Instruction]
`assert-nargs-le` *c24:*`expected` [Instruction]
If the number of actual arguments is not `==`, `>=`, or `<=` *expected*, respectively, signal an error.

The number of arguments is determined by subtracting the stack pointer from the frame pointer (`fp - sp`). See Section 9.3.3 [Stack Layout], page 829, for more details on stack frames. Note that *expected* includes the procedure itself.

`arguments<=?` *c24:*`expected` [Instruction]
Set the `LESS_THAN`, `EQUAL`, or `NONE` comparison result values if the number of arguments is respectively less than, equal to, or greater than *expected*.

`positional-arguments<=?` *c24:nreq x8:_ c24:expected* [Instruction]
Set the `LESS_THAN`, `EQUAL`, or `NONE` comparison result values if the number of positional arguments is respectively less than, equal to, or greater than *expected*. The first *nreq* arguments are positional arguments, as are the subsequent arguments that are not keywords.

The `arguments<=?` and `positional-arguments<=?` instructions are used to implement multiple arities, as in `case-lambda`. See Section 6.9.5 [Case-lambda], page 256, for more information. See Section 9.3.7.15 [Branch Instructions], page 853, for more on comparison results.

`bind-kwargs` *c24:nreq c8:flags c24:nreq-and-opt x8:_ c24:ntotal n32:kw-offset* [Instruction]
flags is a bitfield, whose lowest bit is *allow-other-keys*, second bit is *has-rest*, and whose following six bits are unused.

Find the last positional argument, and shuffle all the rest above *ntotal*. Initialize the intervening locals to `SCM_UNDEFINED`. Then load the constant at *kw-offset* words from the current *ip*, and use it and the *allow-other-keys* flag to bind keyword arguments. If *has-rest*, collect all shuffled arguments into a list, and store it in *nreq-and-opt*. Finally, clear the arguments that we shuffled up.

The parsing is driven by a keyword arguments association list, looked up using *kw-offset*. The alist is a list of pairs of the form `(kw . index)`, mapping keyword arguments to their local slot indices. Unless `allow-other-keys` is set, the parser will signal an error if an unknown key is found.

A macro-mega-instruction.

`bind-optionals` *f24:nlocals* [Instruction]
Expand the current frame to have at least *nlocals* locals, filling in any fresh values with `SCM_UNDEFINED`. If the frame has more than *nlocals* locals, it is left as it is.

`bind-rest` *f24:dst* [Instruction]
Collect any arguments at or above *dst* into a list, and store that list at *dst*.

`alloc-frame` *c24:nlocals* [Instruction]
Ensure that there is space on the stack for *nlocals* local variables. The value of any new local is undefined.

`reset-frame` *c24:nlocals* [Instruction]
Like `alloc-frame`, but doesn't check that the stack is big enough, and doesn't initialize values to `SCM_UNDEFINED`. Used to reset the frame size to something less than the size that was previously set via alloc-frame.

`assert-nargs-ee/locals` *c12:expected c12:nlocals* [Instruction]
Equivalent to a sequence of `assert-nargs-ee` and `allocate-frame`. The number of locals reserved is *expected* + *nlocals*.

9.3.7.3 Shuffling Instructions

These instructions are used to move around values on the stack.

`mov` *s12:dst s12:src* [Instruction]
`long-mov` *s24:dst x8:_ s24:src* [Instruction]
Copy a value from one local slot to another.

As discussed previously, procedure arguments and local variables are allocated to local slots. Guile's compiler tries to avoid shuffling variables around to different slots, which often makes `mov` instructions redundant. However there are some cases in which shuffling is necessary, and in those cases, `mov` is the thing to use.

`long-fmov` *f24:dst x8:_ f24:src* [Instruction]
Copy a value from one local slot to another, but addressing slots relative to the `fp` instead of the `sp`. This is used when shuffling values into place after multiple-value returns.

`push` *s24:src* [Instruction]
Bump the stack pointer by one word, and fill it with the value from slot *src*. The offset to *src* is calculated before the stack pointer is adjusted.

The `push` instruction is used when another instruction is unable to address an operand because the operand is encoded with fewer than 24 bits. In that case, Guile's assembler will transparently emit code that temporarily pushes any needed operands onto the stack, emits the original instruction to address those now-near variables, then shuffles the result (if any) back into place.

`pop` *s24:dst* [Instruction]
Pop the stack pointer, storing the value that was there in slot *dst*. The offset to *dst* is calculated after the stack pointer is adjusted.

`drop` *c24:count* [Instruction]
Pop the stack pointer by *count* words, discarding any values that were stored there.

`shuffle-down` *f12:from f12:to* [Instruction]
Shuffle down values from *from* to *to*, reducing the frame size by *FROM-TO* slots. Part of the internal implementation of `call-with-values`, `values`, and `apply`.

`expand-apply-argument` *x24:_* [Instruction]
Take the last local in a frame and expand it out onto the stack, as for the last argument to `apply`.

9.3.7.4 Trampoline Instructions

Though most applicable objects in Guile are procedures implemented in bytecode, not all are. There are primitives, continuations, and other procedure-like objects that have their own calling convention. Instead of adding special cases to the `call` instruction, Guile wraps these other applicable objects in VM trampoline procedures, then provides special support for these objects in bytecode.

Trampoline procedures are typically generated by Guile at runtime, for example in response to a call to `scm_c_make_gsubr`. As such, a compiler probably shouldn't emit code

with these instructions. However, it's still interesting to know how these things work, so we document these trampoline instructions here.

`subr-call` *c24:idx* [Instruction]
Call a subr, passing all locals in this frame as arguments, and storing the results on the stack, ready to be returned.

`foreign-call` *c12:cif-idx c12:ptr-idx* [Instruction]
Call a foreign function. Fetch the *cif* and foreign pointer from *cif-idx* and *ptr-idx* closure slots of the callee. Arguments are taken from the stack, and results placed on the stack, ready to be returned.

`builtin-ref` *s12:dst c12:idx* [Instruction]
Load a builtin stub by index into *dst*.

9.3.7.5 Non-Local Control Flow Instructions

`capture-continuation` *s24:dst* [Instruction]
Capture the current continuation, and write it to *dst*. Part of the implementation of `call/cc`.

`continuation-call` *c24:contregs* [Instruction]
Return to a continuation, nonlocally. The arguments to the continuation are taken from the stack. *contregs* is a free variable containing the reified continuation.

`abort` *x24:_* [Instruction]
Abort to a prompt handler. The tag is expected in slot 1, and the rest of the values in the frame are returned to the prompt handler. This corresponds to a tail application of `abort-to-prompt`.

If no prompt can be found in the dynamic environment with the given tag, an error is signalled. Otherwise all arguments are passed to the prompt's handler, along with the captured continuation, if necessary.

If the prompt's handler can be proven to not reference the captured continuation, no continuation is allocated. This decision happens dynamically, at run-time; the general case is that the continuation may be captured, and thus resumed. A reinstated continuation will have its arguments pushed on the stack from slot 0, as if from a multiple-value return, and control resumes in the caller. Thus to the calling function, a call to `abort-to-prompt` looks like any other function call.

`compose-continuation` *c24:cont* [Instruction]
Compose a partial continuation with the current continuation. The arguments to the continuation are taken from the stack. *cont* is a free variable containing the reified continuation.

`prompt` *s24:tag b1:escape-only? x7:_ f24:proc-slot x8:_ l24:handler-offset* [Instruction]
Push a new prompt on the dynamic stack, with a tag from *tag* and a handler at *handler-offset* words from the current *ip*.

If an abort is made to this prompt, control will jump to the handler. The handler will expect a multiple-value return as if from a call with the procedure at *proc-slot*, with the reified partial continuation as the first argument, followed by the values returned to the handler. If control returns to the handler, the prompt is already popped off by the abort mechanism. (Guile's `prompt` implements Felleisen's *–F–* operator.)

If *escape-only?* is nonzero, the prompt will be marked as escape-only, which allows an abort to this prompt to avoid reifying the continuation.

See Section 6.13.5 [Prompts], page 303, for more information on prompts.

`throw` *s12:key s12:args* [Instruction]
Raise an error by throwing to *key* and *args*. *args* should be a list.

`throw/value` *s24:value n32:key-subr-and-message* [Instruction]
`throw/value+data` *s24:value n32:key-subr-and-message* [Instruction]
Raise an error, indicating *val* as the bad value. *key-subr-and-message* should be a vector, where the first element is the symbol to which to throw, the second is the procedure in which to signal the error (a string) or `#f`, and the third is a format string for the message, with one template. These instructions do not fall through.

Both of these instructions throw to a key with four arguments: the procedure that indicates the error (or `#f`, the format string, a list with *value*, and either `#f` or the list with *value* as the last argument respectively.

9.3.7.6 Instrumentation Instructions

`instrument-entry` *x24-_ n32:data* [Instruction]
`instrument-loop` *x24-_ n32:data* [Instruction]
Increase execution counter for this function and potentially tier up to the next JIT level. *data* is an offset to a structure recording execution counts and the next-level JIT code corresponding to this function. The increment values are currently 30 for `instrument-entry` and 2 for `instrument-loop`.

`instrument-entry` will also run the apply hook, if VM hooks are enabled.

`handle-interrupts` *x24:_* [Instruction]
Handle pending asynchronous interrupts (asyncs). See Section 6.22.3 [Asyncs], page 445. The compiler inserts `handle-interrupts` instructions before any call, return, or loop back-edge.

`return-from-interrupt` *x24:_* [Instruction]
A special instruction to return from a call and also pop off the stack frame from the call. Used when returning from asynchronous interrupts.

9.3.7.7 Intrinsic Call Instructions

Guile's instruction set is low-level. This is good because the separate components of, say, a `vector-ref` operation might be able to be optimized out, leaving only the operations that need to be performed at run-time.

However some macro-operations may need to perform large amounts of computation at run-time to handle all the edge cases, and whose micro-operation components aren't

amenable to optimization. Residualizing code for the entire macro-operation would lead to code bloat with no benefit.

In this kind of a case, Guile's VM calls out to *intrinsics*: run-time routines written in the host language (currently C, possibly more in the future if Guile gains more run-time targets like WebAssembly). There is one instruction for each instrinsic prototype; the intrinsic is specified by index in the instruction.

`call-thread` *x24:_ c32:idx* [Instruction]
Call the `void`-returning instrinsic with index *idx*, passing the current `scm_thread*` as the argument.

`call-thread-scm` *s24:a c32:idx* [Instruction]
Call the `void`-returning instrinsic with index *idx*, passing the current `scm_thread*` and the `scm` local *a* as arguments.

`call-thread-scm-scm` *s12:a s12:b c32:idx* [Instruction]
Call the `void`-returning instrinsic with index *idx*, passing the current `scm_thread*` and the `scm` locals *a* and *b* as arguments.

`call-scm-sz-u32` *s12:a s12:b c32:idx* [Instruction]
Call the `void`-returning instrinsic with index *idx*, passing the locals *a*, *b*, and *c* as arguments. *a* is a `scm` value, while *b* and *c* are raw `u64` values which fit into `size_t` and `uint32_t` types, respectively.

`call-scm<-u64` *s24:dst c32:idx* [Instruction]
Call the `SCM`-returning instrinsic with index *idx*, passing the current `scm_thread*` as the argument. Place the result in *dst*.

`call-scm<-u64` *s12:dst s12:a c32:idx* [Instruction]
Call the `SCM`-returning instrinsic with index *idx*, passing `u64` local *a* as the argument. Place the result in *dst*.

`call-scm<-s64` *s12:dst s12:a c32:idx* [Instruction]
Call the `SCM`-returning instrinsic with index *idx*, passing `s64` local *a* as the argument. Place the result in *dst*.

`call-scm<-scm` *s12:dst s12:a c32:idx* [Instruction]
Call the `SCM`-returning instrinsic with index *idx*, passing `scm` local *a* as the argument. Place the result in *dst*.

`call-u64<-scm` *s12:dst s12:a c32:idx* [Instruction]
Call the `uint64_t`-returning instrinsic with index *idx*, passing `scm` local *a* as the argument. Place the `u64` result in *dst*.

`call-s64<-scm` *s12:dst s12:a c32:idx* [Instruction]
Call the `int64_t`-returning instrinsic with index *idx*, passing `scm` local *a* as the argument. Place the `s64` result in *dst*.

`call-f64<-scm` *s12:dst s12:a c32:idx* [Instruction]
Call the `double`-returning instrinsic with index *idx*, passing `scm` local *a* as the argument. Place the `f64` result in *dst*.

`call-scm<-scm-scm` *s8:`dst` s8:`a` s8:`b` c32:`idx`* [Instruction]
Call the `SCM`-returning instrinsic with index *idx*, passing `scm` locals *a* and *b* as arguments. Place the `scm` result in *dst*.

`call-scm<-scm-uimm` *s8:`dst` s8:`a` c8:`b` c32:`idx`* [Instruction]
Call the `SCM`-returning instrinsic with index *idx*, passing `scm` local *a* and `uint8_t` immediate *b* as arguments. Place the `scm` result in *dst*.

`call-scm<-thread-scm` *s12:`dst` s12:`a` c32:`idx`* [Instruction]
Call the `SCM`-returning instrinsic with index *idx*, passing the current `scm_thread*` and `scm` local *a* as arguments. Place the `scm` result in *dst*.

`call-scm<-scm-u64` *s8:`dst` s8:`a` s8:`b` c32:`idx`* [Instruction]
Call the `SCM`-returning instrinsic with index *idx*, passing `scm` local *a* and `u64` local *b* as arguments. Place the `scm` result in *dst*.

`call-scm-scm` *s12:`a` s12:`b` c32:`idx`* [Instruction]
Call the `void`-returning instrinsic with index *idx*, passing `scm` locals *a* and *b* as arguments.

`call-scm-scm-scm` *s8:`a` s8:`b` s8:`c` c32:`idx`* [Instruction]
Call the `void`-returning instrinsic with index *idx*, passing `scm` locals *a*, *b*, and *c* as arguments.

`call-scm-uimm-scm` *s8:`a` c8:`b` s8:`c` c32:`idx`* [Instruction]
Call the `void`-returning instrinsic with index *idx*, passing `scm` local *a*, `uint8_t` immediate *b*, and `scm` local *c* as arguments.

There are corresponding macro-instructions for specific intrinsics. These are equivalent to `call-`*`instrinsic-kind`* instructions with the appropriate intrinsic *idx* arguments.

`add` *dst a b* [Macro Instruction]
`add/immediate` *dst a b/imm* [Macro Instruction]
Add `SCM` values *a* and *b* and place the result in *dst*.

`sub` *dst a b* [Macro Instruction]
`sub/immediate` *dst a b/imm* [Macro Instruction]
Subtract `SCM` value *b* from *a* and place the result in *dst*.

`mul` *dst a b* [Macro Instruction]
Multiply `SCM` values *a* and *b* and place the result in *dst*.

`div` *dst a b* [Macro Instruction]
Divide `SCM` value *a* by *b* and place the result in *dst*.

`quo` *dst a b* [Macro Instruction]
Compute the quotient of `SCM` values *a* and *b* and place the result in *dst*.

`rem` *dst a b* [Macro Instruction]
Compute the remainder of `SCM` values *a* and *b* and place the result in *dst*.

`mod` *dst a b* [Macro Instruction]
Compute the modulo of `SCM` value *a* by *b* and place the result in *dst*.

`logand` *dst a b* [Macro Instruction]
Compute the bitwise `and` of `SCM` values *a* and *b* and place the result in *dst*.

`logior` *dst a b* [Macro Instruction]
Compute the bitwise inclusive `or` of `SCM` values *a* and *b* and place the result in *dst*.

`logxor` *dst a b* [Macro Instruction]
Compute the bitwise exclusive `or` of `SCM` values *a* and *b* and place the result in *dst*.

`logsub` *dst a b* [Macro Instruction]
Compute the bitwise `and` of `SCM` value *a* and the bitwise `not` of *b* and place the result in *dst*.

`lsh` *dst a b* [Macro Instruction]
`lsh/immediate` *a b/imm* [Macro Instruction]
Shift `SCM` value *a* left by `u64` value *b* bits and place the result in *dst*.

`rsh` *dst a b* [Macro Instruction]
`rsh/immediate` *dst a b/imm* [Macro Instruction]
Shifts `SCM` value *a* right by `u64` value *b* bits and place the result in *dst*.

`scm->f64` *dst src* [Macro Instruction]
Convert *src* to an unboxed `f64` and place the result in *dst*, or raises an error if *src* is not a real number.

`scm->u64` *dst src* [Macro Instruction]
Convert *src* to an unboxed `u64` and place the result in *dst*, or raises an error if *src* is not an integer within range.

`scm->u64/truncate` *dst src* [Macro Instruction]
Convert *src* to an unboxed `u64` and place the result in *dst*, truncating to the low 64 bits, or raises an error if *src* is not an integer.

`scm->s64` *dst src* [Macro Instruction]
Convert *src* to an unboxed `s64` and place the result in *dst*, or raises an error if *src* is not an integer within range.

`u64->scm` *dst src* [Macro Instruction]
Convert *u64* value *src* to a Scheme integer in *dst*.

`s64->scm` *scm<-s64* [Macro Instruction]
Convert *s64* value *src* to a Scheme integer in *dst*.

`string-set!` *str idx ch* [Macro Instruction]
Sets the character *idx* (a `u64`) of string *str* to *ch* (a `u64` that is a valid character value).

`string->number` *dst src* [Macro Instruction]
Call `string->number` on *src* and place the result in *dst*.

`string->symbol` *dst src* [Macro Instruction]
Call `string->symbol` on *src* and place the result in *dst*.

`symbol->keyword` *dst src* [Macro Instruction]
Call `symbol->keyword` on *src* and place the result in *dst*.

`class-of` *dst src* [Macro Instruction]
Set *dst* to the GOOPS class of `src`.

`wind` *winder unwinder* [Macro Instruction]
Push wind and unwind procedures onto the dynamic stack. Note that neither are actually called; the compiler should emit calls to *winder* and *unwinder* for the normal dynamic-wind control flow. Also note that the compiler should have inserted checks that *winder* and *unwinder* are thunks, if it could not prove that to be the case. See Section 6.13.10 [Dynamic Wind], page 320.

`unwind` [Macro Instruction]
Exit from the dynamic extent of an expression, popping the top entry off of the dynamic stack.

`push-fluid` *fluid value* [Macro Instruction]
Dynamically bind *value* to *fluid* by creating a with-fluids object, pushing that object on the dynamic stack. See Section 6.13.11 [Fluids and Dynamic States], page 323.

`pop-fluid` [Macro Instruction]
Leave the dynamic extent of a `with-fluid*` expression, restoring the fluid to its previous value. `push-fluid` should always be balanced with `pop-fluid`.

`fluid-ref` *dst fluid* [Macro Instruction]
Place the value associated with the fluid *fluid* in *dst*.

`fluid-set!` *fluid value* [Macro Instruction]
Set the value of the fluid *fluid* to *value*.

`push-dynamic-state` *state* [Macro Instruction]
Save the current set of fluid bindings on the dynamic stack and instate the bindings from *state* instead. See Section 6.13.11 [Fluids and Dynamic States], page 323.

`pop-dynamic-state` [Macro Instruction]
Restore a saved set of fluid bindings from the dynamic stack. `push-dynamic-state` should always be balanced with `pop-dynamic-state`.

`resolve-module` *dst name public?* [Macro Instruction]
Look up the module named *name*, resolve its public interface if the immediate operand *public?* is true, then place the result in *dst*.

`lookup` *dst mod sym* [Macro Instruction]
Look up *sym* in module *mod*, placing the resulting variable (or `#f` if not found) in *dst*.

`define!` *dst mod sym* [Macro Instruction]
Look up *sym* in module *mod*, placing the resulting variable in *dst*, creating the variable if needed.

`current-module` *dst* [Macro Instruction]
Set *dst* to the current module.

`$car` *dst src* [Macro Instruction]
`$cdr` *dst src* [Macro Instruction]
`$set-car!` *x val* [Macro Instruction]
`$set-cdr!` *x val* [Macro Instruction]
`$variable-ref` *dst src* [Macro Instruction]
`$variable-set!` *x val* [Macro Instruction]
`$vector-length` *dst x* [Macro Instruction]
`$vector-ref` *dst x idx* [Macro Instruction]
`$vector-ref/immediate` *dst x idx/imm* [Macro Instruction]
`$vector-set!` *x idx v* [Macro Instruction]
`$vector-set!/immediate` *x idx/imm v* [Macro Instruction]
`$allocate-struct` *dst vtable nwords* [Macro Instruction]
`$struct-vtable` *dst src* [Macro Instruction]
`$struct-ref` *dst src idx* [Macro Instruction]
`$struct-ref/immediate` *dst src idx/imm* [Macro Instruction]
`$struct-set!` *x idx v* [Macro Instruction]
`$struct-set!/immediate` *x idx/imm v* [Macro Instruction]
Intrinsics for use by the baseline compiler. The usual strategy for CPS compilation is to expose the component parts of e.g. `vector-ref` so that the compiler can learn from them and eliminate needless bits. However in the non-optimizing baseline compiler, that's just overhead, so we have some intrinsics that encapsulate all the usual type checks.

9.3.7.8 Constant Instructions

The following instructions load literal data into a program. There are two kinds.

The first set of instructions loads immediate values. These instructions encode the immediate directly into the instruction stream.

`make-short-immediate` *s8:`dst` i16:`low-bits`* [Instruction]
Make an immediate whose low bits are *low-bits*, and whose top bits are 0.

`make-long-immediate` *s24:`dst` i32:`low-bits`* [Instruction]
Make an immediate whose low bits are *low-bits*, and whose top bits are 0.

`make-long-long-immediate` *s24:`dst` a32:`high-bits` b32:`low-bits`* [Instruction]
Make an immediate with *high-bits* and *low-bits*.

Non-immediate constant literals are referenced either directly or indirectly. For example, Guile knows at compile-time what the layout of a string will be like, and arranges to embed that object directly in the compiled image. A reference to a string will use `make-non-immediate` to treat a pointer into the compilation unit as a `scm` value directly.

`make-non-immediate` *s24:dst n32:offset* [Instruction]
Load a pointer to statically allocated memory into *dst*. The object's memory will be found *offset* 32-bit words away from the current instruction pointer. Whether the object is mutable or immutable depends on where it was allocated by the compiler, and loaded by the loader.

Sometimes you need to load up a code pointer into a register; for this, use `load-label`.

`load-label` *s24:dst l32:offset* [Instruction]
Load a label *offset* words away from the current `ip` and write it to *dst*. *offset* is a signed 32-bit integer.

Finally, Guile supports a number of unboxed data types, with their associate constant loaders.

`load-f64` *s24:dst au32:high-bits au32:low-bits* [Instruction]
Load a double-precision floating-point value formed by joining *high-bits* and *low-bits*, and write it to *dst*.

`load-u64` *s24:dst au32:high-bits au32:low-bits* [Instruction]
Load an unsigned 64-bit integer formed by joining *high-bits* and *low-bits*, and write it to *dst*.

`load-s64` *s24:dst au32:high-bits au32:low-bits* [Instruction]
Load a signed 64-bit integer formed by joining *high-bits* and *low-bits*, and write it to *dst*.

Some objects must be unique across the whole system. This is the case for symbols and keywords. For these objects, Guile arranges to initialize them when the compilation unit is loaded, storing them into a slot in the image. References go indirectly through that slot. `static-ref` is used in this case.

`static-ref` *s24:dst r32:offset* [Instruction]
Load a *scm* value into *dst*. The *scm* value will be fetched from memory, *offset* 32-bit words away from the current instruction pointer. *offset* is a signed value.

Fields of non-immediates may need to be fixed up at load time, because we do not know in advance at what address they will be loaded. This is the case, for example, for a pair containing a non-immediate in one of its fields. `static-set!` and `static-patch!` are used in these situations.

`static-set!` *s24:src lo32:offset* [Instruction]
Store a *scm* value into memory, *offset* 32-bit words away from the current instruction pointer. *offset* is a signed value.

`static-patch!` *x24:_ lo32:dst-offset l32:src-offset* [Instruction]
Patch a pointer at *dst-offset* to point to *src-offset*. Both offsets are signed 32-bit values, indicating a memory address as a number of 32-bit words away from the current instruction pointer.

9.3.7.9 Memory Access Instructions

In these instructions, the `/immediate` variants represent their indexes or counts as immediates; otherwise these values are unboxed u64 locals.

`allocate-words` *s12:dst s12:count* [Instruction]
`allocate-words/immediate` *s12:dst c12:count* [Instruction]
Allocate a fresh GC-traced object consisting of *count* words and store it into *dst*.

`scm-ref` *s8:dst s8:obj s8:idx* [Instruction]
`scm-ref/immediate` *s8:dst s8:obj c8:idx* [Instruction]
Load the `SCM` object at word offset *idx* from local *obj*, and store it to *dst*.

`scm-set!` *s8:dst s8:idx s8:obj* [Instruction]
`scm-set!/immediate` *s8:dst c8:idx s8:obj* [Instruction]
Store the `scm` local *val* into object *obj* at word offset *idx*.

`scm-ref/tag` *s8:dst s8:obj c8:tag* [Instruction]
Load the first word of *obj*, subtract the immediate *tag*, and store the resulting `SCM` to *dst*.

`scm-set!/tag` *s8:obj c8:tag s8:val* [Instruction]
Set the first word of *obj* to the unpacked bits of the `scm` value *val* plus the immediate value *tag*.

`word-ref` *s8:dst s8:obj s8:idx* [Instruction]
`word-ref/immediate` *s8:dst s8:obj c8:idx* [Instruction]
Load the word at offset *idx* from local *obj*, and store it to the `u64` local *dst*.

`word-set!` *s8:dst s8:idx s8:obj* [Instruction]
`word-set!/immediate` *s8:dst c8:idx s8:obj* [Instruction]
Store the `u64` local *val* into object *obj* at word offset *idx*.

`pointer-ref/immediate` *s8:dst s8:obj c8:idx* [Instruction]
Load the pointer at offset *idx* from local *obj*, and store it to the unboxed pointer local *dst*.

`pointer-set!/immediate` *s8:dst c8:idx s8:obj* [Instruction]
Store the unboxed pointer local *val* into object *obj* at word offset *idx*.

`tail-pointer-ref/immediate` *s8:dst s8:obj c8:idx* [Instruction]
Compute the address of word offset *idx* from local *obj*, and store it to *dst*.

9.3.7.10 Atomic Memory Access Instructions

`current-thread` *s24:dst* [Instruction]
Write the current thread into *dst*.

`atomic-scm-ref/immediate` *s8:dst s8:obj c8:idx* [Instruction]
Atomically load the `SCM` object at word offset *idx* from local *obj*, using the sequential consistency memory model. Store the result to *dst*.

`atomic-scm-set!/immediate` *s8:obj c8:idx s8:val* [Instruction]
Atomically set the `SCM` object at word offset *idx* from local *obj* to *val*, using the sequential consistency memory model.

`atomic-scm-swap!/immediate` *s24:dst x8:_ s24:obj c8:idx s24:val* [Instruction]
Atomically swap the `SCM` value stored in object *obj* at word offset *idx* with *val*, using the sequentially consistent memory model. Store the previous value to *dst*.

`atomic-scm-compare-and-swap!/immediate` *s24:dst x8:_ s24:obj c8:idx s24:expected x8:_ s24:desired* [Instruction]
Atomically swap the `SCM` value stored in object *obj* at word offset *idx* with *desired*, if and only if the value that was there was *expected*, using the sequentially consistent memory model. Store the value that was previously at *idx* from *obj* in *dst*.

9.3.7.11 Tagging and Untagging Instructions

`tag-char` *s12:dst s12:src* [Instruction]
Make a `SCM` character whose integer value is the `u64` in *src*, and store it in *dst*.

`untag-char` *s12:dst s12:src* [Instruction]
Extract the integer value from the `SCM` character *src*, and store the resulting `u64` in *dst*.

`tag-fixnum` *s12:dst s12:src* [Instruction]
Make a `SCM` integer whose value is the `s64` in *src*, and store it in *dst*.

`untag-fixnum` *s12:dst s12:src* [Instruction]
Extract the integer value from the `SCM` integer *src*, and store the resulting `s64` in *dst*.

9.3.7.12 Integer Arithmetic Instructions

`uadd` *s8:dst s8:a s8:b* [Instruction]
`uadd/immediate` *s8:dst s8:a c8:b* [Instruction]
Add the `u64` values *a* and *b*, and store the `u64` result to *dst*. Overflow will wrap.

`usub` *s8:dst s8:a s8:b* [Instruction]
`usub/immediate` *s8:dst s8:a c8:b* [Instruction]
Subtract the `u64` value *b* from *a*, and store the `u64` result to *dst*. Underflow will wrap.

`umul` *s8:dst s8:a s8:b* [Instruction]
`umul/immediate` *s8:dst s8:a c8:b* [Instruction]
Multiply the `u64` values *a* and *b*, and store the `u64` result to *dst*. Overflow will wrap.

`ulogand` *s8:dst s8:a s8:b* [Instruction]
Place the bitwise `and` of the `u64` values *a* and *b* into the `u64` local *dst*.

`ulogior` *s8:dst s8:a s8:b* [Instruction]
Place the bitwise inclusive `or` of the `u64` values *a* and *b* into the `u64` local *dst*.

`ulogxor` *s8:dst s8:a s8:b* [Instruction]
Place the bitwise exclusive `or` of the `u64` values *a* and *b* into the `u64` local *dst*.

`ulogsub` *s8:dst s8:a s8:b* [Instruction]
Place the bitwise `and` of the `u64` values *a* and the bitwise `not` of *b* into the `u64` local *dst*.

`ulsh` *s8:dst s8:a s8:b* [Instruction]
`ulsh/immediate` *s8:dst s8:a c8:b* [Instruction]
Shift the unboxed unsigned 64-bit integer in *a* left by *b* bits, also an unboxed unsigned 64-bit integer. Truncate to 64 bits and write to *dst* as an unboxed value. Only the lower 6 bits of *b* are used.

`ursh` *s8:dst s8:a s8:b* [Instruction]
`ursh/immediate` *s8:dst s8:a c8:b* [Instruction]
Shift the unboxed unsigned 64-bit integer in *a* right by *b* bits, also an unboxed unsigned 64-bit integer. Truncate to 64 bits and write to *dst* as an unboxed value. Only the lower 6 bits of *b* are used.

`srsh` *s8:dst s8:a s8:b* [Instruction]
`srsh/immediate` *s8:dst s8:a c8:b* [Instruction]
Shift the unboxed signed 64-bit integer in *a* right by *b* bits, also an unboxed signed 64-bit integer. Truncate to 64 bits and write to *dst* as an unboxed value. Only the lower 6 bits of *b* are used.

9.3.7.13 Floating-Point Arithmetic Instructions

`fadd` *s8:dst s8:a s8:b* [Instruction]
Add the `f64` values *a* and *b*, and store the `f64` result to *dst*.

`fsub` *s8:dst s8:a s8:b* [Instruction]
Subtract the `f64` value *b* from *a*, and store the `f64` result to *dst*.

`fmul` *s8:dst s8:a s8:b* [Instruction]
Multiply the `f64` values *a* and *b*, and store the `f64` result to *dst*.

`fdiv` *s8:dst s8:a s8:b* [Instruction]
Divide the `f64` values *a* by *b*, and store the `f64` result to *dst*.

9.3.7.14 Comparison Instructions

`u64=?` *s12:a s12:b* [Instruction]
Set the comparison result to *EQUAL* if the `u64` values *a* and *b* are the same, or `NONE` otherwise.

`u64<?` *s12:a s12:b* [Instruction]
Set the comparison result to `LESS_THAN` if the `u64` value *a* is less than the `u64` value *b* are the same, or `NONE` otherwise.

`s64<?` *s12:a s12:b* [Instruction]
Set the comparison result to `LESS_THAN` if the `s64` value *a* is less than the `s64` value *b* are the same, or `NONE` otherwise.

`s64-imm=?` *s12:*`a` *z12:b* [Instruction]
Set the comparison result to *EQUAL* if the `s64` value *a* is equal to the immediate `s64` value *b*, or `NONE` otherwise.

`u64-imm<?` *s12:*`a` *c12:b* [Instruction]
Set the comparison result to `LESS_THAN` if the `u64` value *a* is less than the immediate `u64` value *b*, or `NONE` otherwise.

`imm-u64<?` *s12:*`a` *s12:b* [Instruction]
Set the comparison result to `LESS_THAN` if the `u64` immediate *b* is less than the `u64` value *a*, or `NONE` otherwise.

`s64-imm<?` *s12:*`a` *z12:b* [Instruction]
Set the comparison result to `LESS_THAN` if the `s64` value *a* is less than the immediate `s64` value *b*, or `NONE` otherwise.

`imm-s64<?` *s12:*`a` *z12:b* [Instruction]
Set the comparison result to `LESS_THAN` if the `s64` immediate *b* is less than the `s64` value *a*, or `NONE` otherwise.

`f64=?` *s12:*`a` *s12:b* [Instruction]
Set the comparison result to *EQUAL* if the f64 value *a* is equal to the f64 value *b*, or `NONE` otherwise.

`f64<?` *s12:*`a` *s12:b* [Instruction]
Set the comparison result to `LESS_THAN` if the f64 value *a* is less than the f64 value *b*, `NONE` if *a* is greater than or equal to *b*, or `INVALID` otherwise.

`=?` *s12:*`a` *s12:b* [Instruction]
Set the comparison result to *EQUAL* if the SCM values *a* and *b* are numerically equal, in the sense of the Scheme `=` operator. Set to `NONE` otherwise.

`heap-numbers-equal?` *s12:*`a` *s12:b* [Instruction]
Set the comparison result to *EQUAL* if the SCM values *a* and *b* are numerically equal, in the sense of Scheme `=`. Set to `NONE` otherwise. It is known that both *a* and *b* are heap numbers.

`<?` *s12:*`a` *s12:b* [Instruction]
Set the comparison result to `LESS_THAN` if the SCM value *a* is less than the SCM value *b*, `NONE` if *a* is greater than or equal to *b*, or `INVALID` otherwise.

`immediate-tag=?` *s24:*`obj` *c16:*`mask` *c16:*`tag` [Instruction]
Set the comparison result to *EQUAL* if the result of a bitwise `and` between the bits of `scm` value *a* and the immediate *mask* is *tag*, or `NONE` otherwise.

`heap-tag=?` *s24:*`obj` *c16:*`mask` *c16:*`tag` [Instruction]
Set the comparison result to *EQUAL* if the result of a bitwise `and` between the first word of `scm` value *a* and the immediate *mask* is *tag*, or `NONE` otherwise.

`eq?` *s12:*`a` *s12:b* [Instruction]
Set the comparison result to *EQUAL* if the SCM values *a* and *b* are `eq?`, or `NONE` otherwise.

There are a set of macro-instructions for `immediate-tag=?` and `heap-tag=?` as well that abstract away the precise type tag values. See Section 9.2.5 [The SCM Type in Guile], page 822.

`fixnum?` *x* [Macro Instruction]
`heap-object?` *x* [Macro Instruction]
`char?` *x* [Macro Instruction]
`eq-false?` *x* [Macro Instruction]
`eq-nil?` *x* [Macro Instruction]
`eq-null?` *x* [Macro Instruction]
`eq-true?` *x* [Macro Instruction]
`unspecified?` *x* [Macro Instruction]
`undefined?` *x* [Macro Instruction]
`eof-object?` *x* [Macro Instruction]
`null?` *x* [Macro Instruction]
`false?` *x* [Macro Instruction]
`nil?` *x* [Macro Instruction]

Emit a `immediate-tag=?` instruction that will set the comparison result to `EQUAL` if *x* would pass the corresponding predicate (e.g. `null?`), or `NONE` otherwise.

`pair?` *x* [Macro Instruction]
`struct?` *x* [Macro Instruction]
`symbol?` *x* [Macro Instruction]
`variable?` *x* [Macro Instruction]
`vector?` *x* [Macro Instruction]
`immutable-vector?` *x* [Macro Instruction]
`mutable-vector?` *x* [Macro Instruction]
`weak-vector?` *x* [Macro Instruction]
`string?` *x* [Macro Instruction]
`heap-number?` *x* [Macro Instruction]
`hash-table?` *x* [Macro Instruction]
`pointer?` *x* [Macro Instruction]
`fluid?` *x* [Macro Instruction]
`stringbuf?` *x* [Macro Instruction]
`dynamic-state?` *x* [Macro Instruction]
`frame?` *x* [Macro Instruction]
`keyword?` *x* [Macro Instruction]
`atomic-box?` *x* [Macro Instruction]
`syntax?` *x* [Macro Instruction]
`program?` *x* [Macro Instruction]
`vm-continuation?` *x* [Macro Instruction]
`bytevector?` *x* [Macro Instruction]
`weak-set?` *x* [Macro Instruction]
`weak-table?` *x* [Macro Instruction]
`array?` *x* [Macro Instruction]
`bitvector?` *x* [Macro Instruction]
`smob?` *x* [Macro Instruction]
`port?` *x* [Macro Instruction]

`bignum?` *x* [Macro Instruction]
`flonum?` *x* [Macro Instruction]
`compnum?` *x* [Macro Instruction]
`fracnum?` *x* [Macro Instruction]
Emit a `heap-tag=?` instruction that will set the comparison result to `EQUAL` if *x* would pass the corresponding predicate (e.g. `null?`), or `NONE` otherwise.

9.3.7.15 Branch Instructions

All offsets to branch instructions are 24-bit signed numbers, which count 32-bit units. This gives Guile effectively a 26-bit address range for relative jumps.

`j` *l24:offset* [Instruction]
Add *offset* to the current instruction pointer.

`jl` *l24:offset* [Instruction]
If the last comparison result is `LESS_THAN`, add *offset*, a signed 24-bit number, to the current instruction pointer.

`je` *l24:offset* [Instruction]
If the last comparison result is `EQUAL`, add *offset*, a signed 24-bit number, to the current instruction pointer.

`jnl` *l24:offset* [Instruction]
If the last comparison result is not `LESS_THAN`, add *offset*, a signed 24-bit number, to the current instruction pointer.

`jne` *l24:offset* [Instruction]
If the last comparison result is not `EQUAL`, add *offset*, a signed 24-bit number, to the current instruction pointer.

`jge` *l24:offset* [Instruction]
If the last comparison result is `NONE`, add *offset*, a signed 24-bit number, to the current instruction pointer.

This is intended for use after a `<?` comparison, and is different from `jnl` in the way it handles not-a-number (NaN) values: `<?` sets `INVALID` instead of `NONE` if either value is a NaN. For exact numbers, `jge` is the same as `jnl`.

`jnge` *l24:offset* [Instruction]
If the last comparison result is not `NONE`, add *offset*, a signed 24-bit number, to the current instruction pointer.

This is intended for use after a `<?` comparison, and is different from `jl` in the way it handles not-a-number (NaN) values: `<?` sets `INVALID` instead of `NONE` if either value is a NaN. For exact numbers, `jnge` is the same as `jl`.

9.3.7.16 Raw Memory Access Instructions

Bytevector operations correspond closely to what the current hardware can do, so it makes sense to inline them to VM instructions, providing a clear path for eventual native compilation. Without this, Scheme programs would need other primitives for accessing raw bytes – but these primitives are as good as any.

`u8-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`s8-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`u16-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`s16-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`u32-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`s32-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`u64-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`s64-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`f32-ref` *s8:dst s8:ptr s8:idx* [Instruction]
`f64-ref` *s8:dst s8:ptr s8:idx* [Instruction]

Fetch the item at byte offset *idx* from the raw pointer local *ptr*, and store it in *dst*. All accesses use native endianness.

The *idx* value should be an unboxed unsigned 64-bit integer.

The results are all written to the stack as unboxed values, either as signed 64-bit integers, unsigned 64-bit integers, or IEEE double floating point numbers.

`u8-set!` *s8:ptr s8:idx s8:val* [Instruction]
`s8-set!` *s8:ptr s8:idx s8:val* [Instruction]
`u16-set!` *s8:ptr s8:idx s8:val* [Instruction]
`s16-set!` *s8:ptr s8:idx s8:val* [Instruction]
`u32-set!` *s8:ptr s8:idx s8:val* [Instruction]
`s32-set!` *s8:ptr s8:idx s8:val* [Instruction]
`u64-set!` *s8:ptr s8:idx s8:val* [Instruction]
`s64-set!` *s8:ptr s8:idx s8:val* [Instruction]
`f32-set!` *s8:ptr s8:idx s8:val* [Instruction]
`f64-set!` *s8:ptr s8:idx s8:val* [Instruction]

Store *val* into memory pointed to by raw pointer local *ptr*, at byte offset *idx*. Multibyte values are written using native endianness.

The *idx* value should be an unboxed unsigned 64-bit integer.

The *val* values are all unboxed, either as signed 64-bit integers, unsigned 64-bit integers, or IEEE double floating point numbers.

9.3.8 Just-In-Time Native Code

The final piece of Guile's virtual machine is a just-in-time (JIT) compiler from bytecode instructions to native code. It is faster to run a function when its bytecode instructions are compiled to native code, compared to having the VM interpret the instructions.

The JIT compiler runs automatically, triggered by counters associated with each function. The counter increments when functions are called and during each loop iteration. Once a function's counter passes a certain value, the function gets JIT-compiled. See Section 9.3.7.6 [Instrumentation Instructions], page 841, for full details.

Guile's JIT compiler is what is known as a *template JIT*. This kind of JIT is very simple: for each instruction in a function, the JIT compiler will emit a generic sequence of machine code corresponding to the instruction kind, specializing that generic template to reference the specific operands of the instruction being compiled.

The strength of a template JIT is principally that it is very fast at emitting code. It doesn't need to do any time-consuming analysis on the bytecode that it is compiling to do its job.

A template JIT is also very predictable: the native code emitted by a template JIT has the same performance characteristics of the corresponding bytecode, only that it runs faster. In theory you could even generate the template-JIT machine code ahead of time, as it doesn't depend on any value seen at run-time.

This predictability makes it possible to reason about the performance of a system in terms of bytecode, knowing that the conclusions apply to native code emitted by a template JIT.

Because the machine code corresponding to an instruction always performs the same tasks that the interpreter would do for that instruction, bytecode and a template JIT also allows Guile programmers to debug their programs in terms of the bytecode model. When a Guile programmer sets a breakpoint, Guile will disable the JIT for the thread being debugged, falling back to the interpreter (which has the corresponding code to run the hooks). See Section 6.26.4.1 [VM Hooks], page 485.

To emit native code, Guile uses a forked version of GNU Lightning. This "Lightening" effort, spun out as a separate project, aims to build on the back-end support from GNU Lightning, but adapting the API and behavior of the library to match Guile's needs. This code is included in the Guile source distribution. For more information, see `https://gitlab.com/wingo/lightening`. As of mid-2019, Lightening supports code generation for the x86-64, ia32, ARMv7, and AArch64 architectures.

The weaknesses of a template JIT are two-fold. Firstly, as a simple back-end that has to run fast, a template JIT doesn't have time to do analysis that could help it generate better code, notably global register allocation and instruction selection.

However this is a minor weakness compared to the inability to perform significant, speculative program transformations. For example, Guile could see that in an expression `(f x)`, that in practice f always refers to the same function. An advanced JIT compiler would speculatively inline f into the call-site, along with a dynamic check to make sure that the assertion still held. But as a template JIT doesn't pay attention to values only known at run-time, it can't make this transformation.

This limitation is mitigated in part by Guile's robust ahead-of-time compiler which can already perform significant optimizations when it can prove they will always be valid, and its low-level bytecode which is able to represent the effect of those optimizations (e.g. elided type-checks). See Section 9.4 [Compiling to the Virtual Machine], page 856, for more on Guile's compiler.

An ahead-of-time Scheme-to-bytecode strategy, complemented by a template JIT, also particularly suits the somewhat static nature of Scheme. Scheme programmers often write code in a way that makes the identity of free variable references lexically apparent. For example, the `(f x)` expression could appear within a `(let ((f (lambda (x) (1+ x)))) ...)` expression, or we could see that `f` was imported from a particular module where we know its binding. Ahead-of-time compilation techniques can work well for a language like Scheme where there is little polymorphism and much first-order programming. They do not work so well for a language like JavaScript, which is highly mutable at run-time and difficult to analyze due to method calls (which are effectively higher-order calls).

All that said, a template JIT works well for Guile at this point. It's only a few thousand lines of maintainable code, it speeds up Scheme programs, and it keeps the bulk of the Guile Scheme implementation written in Scheme itself. The next step is probably to add ahead-of-time native code emission to the back-end of the compiler written in Scheme, to take advantage of the opportunity to do global register allocation and instruction selection. Once this is working, it can allow Guile to experiment with speculative optimizations in Scheme as well. See Section 9.4.7 [Extending the Compiler], page 875, for more on future directions.

Finally, note that there are a few environment variables that can be tweaked to make JIT compilation happen sooner, later, or never. See Section 4.2.2 [Environment Variables], page 38, for more.

9.4 Compiling to the Virtual Machine

Compilers! The word itself inspires excitement and awe, even among experienced practitioners. But a compiler is just a program: an eminently hackable thing. This section aims to describe Guile's compiler in such a way that interested Scheme hackers can feel comfortable reading and extending it.

See Section 6.18 [Read/Load/Eval/Compile], page 382, if you're lost and you just wanted to know how to compile your `.scm` file.

9.4.1 Compiler Tower

Guile's compiler is quite simple – its *compilers*, to put it more accurately. Guile defines a tower of languages, starting at Scheme and progressively simplifying down to languages that resemble the VM instruction set (see Section 9.3.7 [Instruction Set], page 835).

Each language knows how to compile to the next, so each step is simple and understandable. Furthermore, this set of languages is not hardcoded into Guile, so it is possible for the user to add new high-level languages, new passes, or even different compilation targets.

Languages are registered in the module, `(system base language)`:

```
(use-modules (system base language))
```

They are registered with the `define-language` form.

`define-language` [*#:name*] [*#:title*] [*#:reader*] [*#:printer*] [*#:parser=#f*] [*#:compilers='()*] [*#:decompilers='()*] [*#:evaluator=#f*] [*#:joiner=#f*] [*#:for-humans?=#t*] [*#:make-default-environment=make-fresh-user-module*] [*#:lowerer=#f*] [*#:analyzer=#f*] [*#:compiler-chooser=#f*] [Scheme Syntax]

Define a language.

This syntax defines a `<language>` object, bound to *name* in the current environment. In addition, the language will be added to the global language set. For example, this is the language definition for Scheme:

```
(define-language scheme
  #:title "Scheme"
  #:reader      (lambda (port env) ...)
  #:compilers   `((tree-il . ,compile-tree-il))
  #:decompilers `((tree-il . ,decompile-tree-il))
```

```
#:evaluator (lambda (x module) (primitive-eval x))
#:printer write
#:make-default-environment (lambda () ...))
```

The interesting thing about having languages defined this way is that they present a uniform interface to the read-eval-print loop. This allows the user to change the current language of the REPL:

```
scheme@(guile-user)> ,language tree-il
Happy hacking with Tree Intermediate Language!  To switch back, type ‘,L scheme’.
tree-il@(guile-user)> ,L scheme
Happy hacking with Scheme!  To switch back, type ‘,L tree-il’.
scheme@(guile-user)>
```

Languages can be looked up by name, as they were above.

`lookup-language` *name* [Scheme Procedure]

Looks up a language named *name*, autoloading it if necessary.

Languages are autoloaded by looking for a variable named *name* in a module named `(language` *name* `spec)`.

The language object will be returned, or `#f` if there does not exist a language with that name.

When Guile goes to compile Scheme to bytecode, it will ask the Scheme language to choose a compiler from Scheme to the next language on the path from Scheme to bytecode. Performing this computation recursively builds transformations from a flexible chain of compilers. The next link will be obtained by invoking the language's compiler chooser, or if not present, from the language's compilers field.

A language can specify an analyzer, which is run before a term of that language is lowered and compiled. This is where compiler warnings are issued.

If a language specifies a lowerer, that procedure is called on expressions before compilation. This is where optimizations and canonicalizations go.

Finally a language's compiler translates a lowered term from one language to the next one in the chain.

There is a notion of a "current language", which is maintained in the `current-language` parameter, defined in the core `(guile)` module. This language is normally Scheme, and may be rebound by the user. The run-time compilation interfaces (see Section 6.18 [Read/Load/Eval/Compile], page 382) also allow you to choose other source and target languages.

The normal tower of languages when compiling Scheme goes like this:

- Scheme
- Tree Intermediate Language (Tree-IL)
- Continuation-Passing Style (CPS)
- Bytecode

As discussed before (see Section 9.3.6 [Object File Format], page 833), bytecode is in ELF format, ready to be serialized to disk. But when compiling Scheme at run time, you

want a Scheme value: for example, a compiled procedure. For this reason, so as not to break the abstraction, Guile defines a fake language at the bottom of the tower:

- Value

Compiling to `value` loads the bytecode into a procedure, turning cold bytes into warm code.

Perhaps this strangeness can be explained by example: `compile-file` defaults to compiling to bytecode, because it produces object code that has to live in the barren world outside the Guile runtime; but `compile` defaults to compiling to `value`, as its product re-enters the Guile world.

Indeed, the process of compilation can circulate through these different worlds indefinitely, as shown by the following quine:

```
((lambda (x) ((compile x) x)) '(lambda (x) ((compile x) x)))
```

9.4.2 The Scheme Compiler

The job of the Scheme compiler is to expand all macros and all of Scheme to its most primitive expressions. The definition of "primitive expression" is given by the inventory of constructs provided by Tree-IL, the target language of the Scheme compiler: procedure calls, conditionals, lexical references, and so on. This is described more fully in the next section.

The tricky and amusing thing about the Scheme-to-Tree-IL compiler is that it is completely implemented by the macro expander. Since the macro expander has to run over all of the source code already in order to expand macros, it might as well do the analysis at the same time, producing Tree-IL expressions directly.

Because this compiler is actually the macro expander, it is extensible. Any macro which the user writes becomes part of the compiler.

The Scheme-to-Tree-IL expander may be invoked using the generic `compile` procedure:

```
(compile '(+ 1 2) #:from 'scheme #:to 'tree-il)
⇒
#<tree-il (call (toplevel +) (const 1) (const 2))>
```

`(compile` *foo* `#:from 'scheme #:to 'tree-il)` is entirely equivalent to calling the macro expander as `(macroexpand` *foo* `'c '(compile load eval))`. See Section 6.10.9 [Macro Expansion], page 280. `compile-tree-il`, the procedure dispatched by `compile` to `'tree-il`, is a small wrapper around `macroexpand`, to make it conform to the general form of compiler procedures in Guile's language tower.

Compiler procedures take three arguments: an expression, an environment, and a keyword list of options. They return three values: the compiled expression, the corresponding environment for the target language, and a "continuation environment". The compiled expression and environment will serve as input to the next language's compiler. The "continuation environment" can be used to compile another expression from the same source language within the same module.

For example, you might compile the expression, `(define-module (foo))`. This will result in a Tree-IL expression and environment. But if you compiled a second expression, you would want to take into account the compile-time effect of compiling the previous

expression, which puts the user in the `(foo)` module. That is the purpose of the "continuation environment"; you would pass it as the environment when compiling the subsequent expression.

For Scheme, an environment is a module. By default, the `compile` and `compile-file` procedures compile in a fresh module, such that bindings and macros introduced by the expression being compiled are isolated:

```
(eq? (current-module) (compile '(current-module)))
⇒ #f

(compile '(define hello 'world))
(defined? 'hello)
⇒ #f

(define / *)
(eq? (compile '/) /)
⇒ #f
```

Similarly, changes to the `current-reader` fluid (see Section 6.18.6 [Loading], page 392) are isolated:

```
(compile '(fluid-set! current-reader (lambda args 'fail)))
(fluid-ref current-reader)
⇒ #f
```

Nevertheless, having the compiler and *compilee* share the same name space can be achieved by explicitly passing `(current-module)` as the compilation environment:

```
(define hello 'world)
(compile 'hello #:env (current-module))
⇒ world
```

9.4.3 Tree-IL

Tree Intermediate Language (Tree-IL) is a structured intermediate language that is close in expressive power to Scheme. It is an expanded, pre-analyzed Scheme.

Tree-IL is "structured" in the sense that its representation is based on records, not S-expressions. This gives a rigidity to the language that ensures that compiling to a lower-level language only requires a limited set of transformations. For example, the Tree-IL type `<const>` is a record type with two fields, `src` and `exp`. Instances of this type are created via `make-const`. Fields of this type are accessed via the `const-src` and `const-exp` procedures. There is also a predicate, `const?`. See Section 6.6.17 [Records], page 221, for more information on records.

All Tree-IL types have a `src` slot, which holds source location information for the expression. This information, if present, will be residualized into the compiled object code, allowing backtraces to show source information. The format of `src` is the same as that returned by Guile's `source-properties` function. See Section 6.26.2 [Source Properties], page 477, for more information.

Although Tree-IL objects are represented internally using records, there is also an equivalent S-expression external representation for each kind of Tree-IL. For example, the S-expression representation of `#<const src: #f exp: 3>` expression would be:

```
(const 3)
```

Users may program with this format directly at the REPL:

```
scheme@(guile-user)> ,language tree-il
Happy hacking with Tree Intermediate Language!  To switch back, type ',L scheme'.
tree-il@(guile-user)> (call (primitive +) (const 32) (const 10))
⇒ 42
```

The `src` fields are left out of the external representation.

One may create Tree-IL objects from their external representations via calling `parse-tree-il`, the reader for Tree-IL. If any source information is attached to the input S-expression, it will be propagated to the resulting Tree-IL expressions. This is probably the easiest way to compile to Tree-IL: just make the appropriate external representations in S-expression format, and let `parse-tree-il` take care of the rest.

`<void>` *src* [Scheme Variable]
`(void)` [External Representation]

An empty expression. In practice, equivalent to Scheme's `(if #f #f)`.

`<const>` *src exp* [Scheme Variable]
`(const` *exp*`)` [External Representation]

A constant.

`<primitive-ref>` *src name* [Scheme Variable]
`(primitive` *name*`)` [External Representation]

A reference to a "primitive". A primitive is a procedure that, when compiled, may be open-coded. For example, `cons` is usually recognized as a primitive, so that it compiles down to a single instruction.

Compilation of Tree-IL usually begins with a pass that resolves some `<module-ref>` and `<toplevel-ref>` expressions to `<primitive-ref>` expressions. The actual compilation pass has special cases for calls to certain primitives, like `apply` or `cons`.

`<lexical-ref>` *src name gensym* [Scheme Variable]
`(lexical` *name gensym*`)` [External Representation]

A reference to a lexically-bound variable. The *name* is the original name of the variable in the source program. *gensym* is a unique identifier for this variable.

`<lexical-set>` *src name gensym exp* [Scheme Variable]
`(set!` (*lexical* *name gensym*`)` *exp*`)` [External Representation]

Sets a lexically-bound variable.

`<module-ref>` *src mod name public?* [Scheme Variable]
`(@` *mod name*`)` [External Representation]
`(@@` *mod name*`)` [External Representation]

A reference to a variable in a specific module. *mod* should be the name of the module, e.g. `(guile-user)`.

If *public?* is true, the variable named *name* will be looked up in *mod*'s public interface, and serialized with `@`; otherwise it will be looked up among the module's private bindings, and is serialized with `@@`.

`<module-set>` *src mod name public? exp* [Scheme Variable]
`(set! (@ mod name) exp)` [External Representation]
`(set! (@@ mod name) exp)` [External Representation]
Sets a variable in a specific module.

`<toplevel-ref>` *src name* [Scheme Variable]
`(toplevel name)` [External Representation]
References a variable from the current procedure's module.

`<toplevel-set>` *src name exp* [Scheme Variable]
`(set! (toplevel name) exp)` [External Representation]
Sets a variable in the current procedure's module.

`<toplevel-define>` *src name exp* [Scheme Variable]
`(define name exp)` [External Representation]
Defines a new top-level variable in the current procedure's module.

`<conditional>` *src test then else* [Scheme Variable]
`(if test then else)` [External Representation]
A conditional. Note that *else* is not optional.

`<call>` *src proc args* [Scheme Variable]
`(call proc . args)` [External Representation]
A procedure call.

`<primcall>` *src name args* [Scheme Variable]
`(primcall name . args)` [External Representation]
A call to a primitive. Equivalent to `(call (primitive name) . args)`. This construct is often more convenient to generate and analyze than `<call>`.

As part of the compilation process, instances of `(call (primitive name) . args)` are transformed into primcalls.

`<seq>` *src head tail* [Scheme Variable]
`(seq head tail)` [External Representation]
A sequence. The semantics is that *head* is evaluated first, and any resulting values are ignored. Then *tail* is evaluated, in tail position.

`<lambda>` *src meta body* [Scheme Variable]
`(lambda meta body)` [External Representation]
A closure. *meta* is an association list of properties for the procedure. *body* is a single Tree-IL expression of type `<lambda-case>`. As the `<lambda-case>` clause can chain to an alternate clause, this makes Tree-IL's `<lambda>` have the expressiveness of Scheme's `case-lambda`.

`<lambda-case>` *req opt rest kw inits gensyms body alternate* [Scheme Variable]
`(lambda-case ((req opt rest kw inits gensyms)` [External Representation]
`body) [alternate])`
One clause of a `case-lambda`. A `lambda` expression in Scheme is treated as a `case-lambda` with one clause.

req is a list of the procedure's required arguments, as symbols. *opt* is a list of the optional arguments, or `#f` if there are no optional arguments. *rest* is the name of the rest argument, or `#f`.

kw is a list of the form, `(allow-other-keys? (keyword name var) ...)`, where *keyword* is the keyword corresponding to the argument named *name*, and whose corresponding gensym is *var*, or `#f` if there are no keyword arguments. *inits* are tree-il expressions corresponding to all of the optional and keyword arguments, evaluated to bind variables whose value is not supplied by the procedure caller. Each *init* expression is evaluated in the lexical context of previously bound variables, from left to right.

gensyms is a list of gensyms corresponding to all arguments: first all of the required arguments, then the optional arguments if any, then the rest argument if any, then all of the keyword arguments.

body is the body of the clause. If the procedure is called with an appropriate number of arguments, *body* is evaluated in tail position. Otherwise, if there is an *alternate*, it should be a `<lambda-case>` expression, representing the next clause to try. If there is no *alternate*, a wrong-number-of-arguments error is signaled.

`<let>` *src names gensyms vals exp* [Scheme Variable]
`(let` *names gensyms vals exp*`)` [External Representation]
Lexical binding, like Scheme's `let`. *names* are the original binding names, *gensyms* are gensyms corresponding to the *names*, and *vals* are Tree-IL expressions for the values. *exp* is a single Tree-IL expression.

`<letrec>` *in-order? src names gensyms vals exp* [Scheme Variable]
`(letrec` *names gensyms vals exp*`)` [External Representation]
`(letrec*` *names gensyms vals exp*`)` [External Representation]
A version of `<let>` that creates recursive bindings, like Scheme's `letrec`, or `letrec*` if *in-order?* is true.

`<prompt>` *escape-only? tag body handler* [Scheme Variable]
`(prompt` *escape-only? tag body handler*`)` [External Representation]
A dynamic prompt. Instates a prompt named *tag*, an expression, during the dynamic extent of the execution of *body*, also an expression. If an abort occurs to this prompt, control will be passed to *handler*, also an expression, which should be a procedure. The first argument to the handler procedure will be the captured continuation, followed by all of the values passed to the abort. If *escape-only?* is true, the handler should be a `<lambda>` with a single `<lambda-case>` body expression with no optional or keyword arguments, and no alternate, and whose first argument is unreferenced. See Section 6.13.5 [Prompts], page 303, for more information.

`<abort>` *tag args tail* [Scheme Variable]
`(abort` *tag args tail*`)` [External Representation]
An abort to the nearest prompt with the name *tag*, an expression. *args* should be a list of expressions to pass to the prompt's handler, and *tail* should be an expression that will evaluate to a list of additional arguments. An abort will save the partial continuation, which may later be reinstated, resulting in the `<abort>` expression evaluating to some number of values.

There are two Tree-IL constructs that are not normally produced by higher-level compilers, but instead are generated during the source-to-source optimization and analysis passes that the Tree-IL compiler does. Users should not generate these expressions directly, unless they feel very clever, as the default analysis pass will generate them as necessary.

`<let-values>` *src names gensyms exp body* [Scheme Variable]
`(let-values` *names gensyms exp body*`)` [External Representation]

Like Scheme's `receive` – binds the values returned by evaluating `exp` to the `lambda`-like bindings described by *gensyms*. That is to say, *gensyms* may be an improper list.

`<let-values>` is an optimization of a `<call>` to the primitive, `call-with-values`.

`<fix>` *src names gensyms vals body* [Scheme Variable]
`(fix` *names gensyms vals body*`)` [External Representation]

Like `<letrec>`, but only for *vals* that are unset `lambda` expressions.

`fix` is an optimization of `letrec` (and `let`).

Tree-IL is a convenient compilation target from source languages. It can be convenient as a medium for optimization, though CPS is usually better. The strength of Tree-IL is that it does not fix order of evaluation, so it makes some code motion a bit easier.

Optimization passes performed on Tree-IL currently include:

- Open-coding (turning toplevel-refs into primitive-refs, and calls to primitives to prim-calls)
- Partial evaluation (comprising inlining, copy propagation, and constant folding)

9.4.4 Continuation-Passing Style

Continuation-passing style (CPS) is Guile's principal intermediate language, bridging the gap between languages for people and languages for machines. CPS gives a name to every part of a program: every control point, and every intermediate value. This makes it an excellent medium for reasoning about programs, which is the principal job of a compiler.

9.4.4.1 An Introduction to CPS

Consider the following Scheme expression:

```
(begin
  (display "The sum of 32 and 10 is: ")
  (display 42)
  (newline))
```

Let us identify all of the sub-expressions in this expression, annotating them with unique labels:

```
(begin
  (display "The sum of 32 and 10 is: ")
  |k1      k2
  k0
  (display 42)
  |k4      k5
  k3
```

```
(newline))
|k7
k6
```

Each of these labels identifies a point in a program. One label may be the continuation of another label. For example, the continuation of `k7` is `k6`. This is because after evaluating the value of `newline`, performed by the expression labelled `k7`, we continue to apply it in `k6`.

Which expression has `k0` as its continuation? It is either the expression labelled `k1` or the expression labelled `k2`. Scheme does not have a fixed order of evaluation of arguments, though it does guarantee that they are evaluated in some order. Unlike general Scheme, continuation-passing style makes evaluation order explicit. In Guile, this choice is made by the higher-level language compilers.

Let us assume a left-to-right evaluation order. In that case the continuation of `k1` is `k2`, and the continuation of `k2` is `k0`.

With this example established, we are ready to give an example of CPS in Scheme:

```
(lambda (ktail)
  (let ((k1 (lambda ()
              (let ((k2 (lambda (proc)
                          (let ((k0 (lambda (arg0)
                                      (proc k4 arg0))))
                            (k0 "The sum of 32 and 10 is: "))))
                (k2 display))))
        (k4 (lambda _
              (let ((k5 (lambda (proc)
                          (let ((k3 (lambda (arg0)
                                      (proc k7 arg0))))
                            (k3 42)))))
                (k5 display))))
        (k7 (lambda _
              (let ((k6 (lambda (proc)
                          (proc ktail))))
                (k6 newline)))))
    (k1))
```

Holy code explosion, Batman! What's with all the lambdas? Indeed, CPS is by nature much more verbose than "direct-style" intermediate languages like Tree-IL. At the same time, CPS is simpler than full Scheme, because it makes things more explicit.

In the original program, the expression labelled `k0` is in effect context. Any values it returns are ignored. In Scheme, this fact is implicit. In CPS, we can see it explicitly by noting that its continuation, `k4`, takes any number of values and ignores them. Compare this to `k2`, which takes a single value; in this way we can say that `k1` is in a "value" context. Likewise `k6` is in tail context with respect to the expression as a whole, because its continuation is the tail continuation, `ktail`. CPS makes these details manifest, and gives them names.

9.4.4.2 CPS in Guile

Guile's CPS language is composed of *continuations*. A continuation is a labelled program point. If you are used to traditional compilers, think of a continuation as a trivial basic block. A program is a "soup" of continuations, represented as a map from labels to continuations.

Like basic blocks, each continuation belongs to only one function. Some continuations are special, like the continuation corresponding to a function's entry point, or the continuation that represents the tail of a function. Others contain a *term*. A term contains an *expression*, which evaluates to zero or more values. The term also records the continuation to which it will pass its values. Some terms, like conditional branches, may continue to one of a number of continuations.

Continuation labels are small integers. This makes it easy to sort them and to group them into sets. Whenever a term refers to a continuation, it does so by name, simply recording the label of the continuation. Continuation labels are unique among the set of labels in a program.

Variables are also named by small integers. Variable names are unique among the set of variables in a program.

For example, a simple continuation that receives two values and adds them together can be matched like this, using the `match` form from `(ice-9 match)`:

```
(match cont
  (($ $kargs (x-name y-name) (x-var y-var)
      ($ $continue k src ($ $primcall '+ #f (x-var y-var))))
   (format #t "Add ~a and ~a and pass the result to label ~a"
           x-var y-var k)))
```

Here we see the most common kind of continuation, `$kargs`, which binds some number of values to variables and then evaluates a term.

`$kargs` *names vars term* [CPS Continuation]

Bind the incoming values to the variables *vars*, with original names *names*, and then evaluate *term*.

The *names* of a `$kargs` are just for debugging, and will end up residualized in the object file for use by the debugger.

The *term* in a `$kargs` is always a `$continue`, which evaluates an expression and continues to a continuation.

`$continue` *k src exp* [CPS Term]

Evaluate the expression *exp* and pass the resulting values (if any) to the continuation labelled *k*. The source information associated with the expression may be found in *src*, which is either an alist as in `source-properties` or is `#f` if there is no associated source.

There are a number of expression kinds. Above you see an example of `$primcall`.

`$primcall` *name param args* [CPS Expression]

Perform the primitive operation identified by `name`, a well-known symbol, passing it the arguments *args*, and pass all resulting values to the continuation.

param is a constant parameter whose interpretation is up to the primcall in question. Usually it's `#f` but for a primcall that might need some compile-time constant information – such as `add/immediate`, which adds a constant number to a value – the parameter holds this information.

The set of available primitives includes many primitives known to Tree-IL and then some more; see the source code for details. Note that some Tree-IL primcalls need

to be converted to a sequence of lower-level CPS primcalls. Again, see `(language tree-il compile-cps)` for full details.

The variables that are used by `$primcall`, or indeed by any expression, must be defined before the expression is evaluated. An equivalent way of saying this is that predecessor `$kargs` continuation(s) that bind the variables(s) used by the expression must *dominate* the continuation that uses the expression: definitions dominate uses. This condition is trivially satisfied in our example above, but in general to determine the set of variables that are in "scope" for a given term, you need to do a flow analysis to see what continuations dominate a term. The variables that are in scope are those variables defined by the continuations that dominate a term.

Here is an inventory of the kinds of expressions in Guile's CPS language, besides `$primcall` which has already been described. Recall that all expressions are wrapped in a `$continue` term which specifies their continuation.

`$const` *val* [CPS Expression]
Continue with the constant value *val*.

`$prim` *name* [CPS Expression]
Continue with the procedure that implements the primitive operation named by *name*.

`$call` *proc args* [CPS Expression]
Call *proc* with the arguments *args*, and pass all values to the continuation. *proc* and the elements of the *args* list should all be variable names. The continuation identified by the term's *k* should be a `$kreceive` or a `$ktail` instance.

`$values` *args* [CPS Expression]
Pass the values named by the list *args* to the continuation.

`$prompt` *escape? tag handler* [CPS Expression]
There are two sub-languages of CPS, *higher-order CPS* and *first-order CPS*. The difference is that in higher-order CPS, there are `$fun` and `$rec` expressions that bind functions or mutually-recursive functions in the implicit scope of their use sites. Guile transforms higher-order CPS into first-order CPS by *closure conversion*, which chooses representations for all closures and which arranges to access free variables through the implicit closure parameter that is passed to every function call.

`$fun` *body* [CPS Expression]
Continue with a procedure. *body* names the entry point of the function, which should be a `$kfun`. This expression kind is only valid in higher-order CPS, which is the CPS language before closure conversion.

`$rec` *names vars funs* [CPS Expression]
Continue with a set of mutually recursive procedures denoted by *names*, *vars*, and *funs*. *names* is a list of symbols, *vars* is a list of variable names (unique integers), and *funs* is a list of `$fun` values. Note that the `$kargs` continuation should also define *names*/*vars* bindings.

The contification pass will attempt to transform the functions declared in a `$rec` into local continuations. Any remaining `$fun` instances are later removed by the closure conversion pass. If the function has no free variables, it gets allocated as a constant.

`$const-fun` *label* [CPS Expression]

A constant which is a function whose entry point is *label*. As a constant, instances of `$const-fun` with the same *label* will not allocate; the space for the function is allocated as part of the compilation unit.

In practice, `$const-fun` expressions are reified by CPS-conversion for functions whose call sites are not all visible within the compilation unit and which have no free variables. This expression kind is part of first-order CPS.

Otherwise, if the closure has free variables, it will be allocated at its definition site via an `allocate-words` primcall and its free variables initialized there. The code pointer in the closure is initialized from a `$code` expression.

`$code` *label* [CPS Expression]

Continue with the value of *label*, which should denote some `$kfun` continuation in the program. Used when initializing the code pointer of closure objects.

However, If the closure can be proven to never escape its scope then other lighter-weight representations can be chosen. Additionally, if all call sites are known, closure conversion will hard-wire the calls by lowering `$call` to `$callk`.

`$callk` *label proc args* [CPS Expression]

Like `$call`, but for the case where the call target is known to be in the same compilation unit. *label* should denote some `$kfun` continuation in the program. In this case the *proc* is simply an additional argument, since it is not used to determine the call target at run-time.

To summarize: a `$continue` is a CPS term that continues to a single label. But there are other kinds of CPS terms that can continue to a different number of labels: `$branch`, `$throw`, and `$prompt`.

`$branch` *kf kt src op param args* [CPS Term]

Evaluate the branching primcall *op*, with arguments *args* and constant parameter *param*, and continue to *kt* with zero values if the test is true. Otherwise continue to *kf*.

The `$branch` term is like a `$continue` term with a `$primcall` expression, except that instead of binding a value and continuing to a single label, the result of the test is not bound but instead used to choose the continuation label.

The set of operations (corresponding to *op* values) that are valid in a *$branch* is limited. In the general case, bind the result of a test expression to a variable, and then make a `$branch` on a `true?` op referencing that variable. The optimizer should inline the branch if possible.

`$throw` *src op param args* [CPS Term]

Throw a non-resumable exception. Throw terms do not continue at all. The usual value of *op* is `throw`, with two arguments *key* and *args*. There are also some specific primcalls that compile to the VM `throw/value` and `throw/value+data` instructions; see the code for full details.

The advantage of having `$throw` as a term is that, because it does not continue, this allows the optimizer to gather more information from type predicates. For example,

if the predicate is `char?` and the *kf* continues to a throw, the set of labels dominated by *kt* is larger than if the throw notationally continued to some label that would never be reached by the throw.

`$prompt` *k kh src escape? tag* [CPS Term]
Push a prompt on the stack identified by the variable name *tag*, which may be escape-only if *escape?* is true, and continue to *kh* with zero values. If the body aborts to this prompt, control will proceed at the continuation labelled *kh*, which should be a `$kreceive` continuation. Prompts are later popped by `pop-prompt` primcalls.

At this point we have described terms, expressions, and the most common kind of continuation, `$kargs`. `$kargs` is used when the predecessors of the continuation can be instructed to pass the values where the continuation wants them. For example, if a `$kargs` continuation *k* binds a variable *v*, and the compiler decides to allocate *v* to slot 6, all predecessors of *k* should put the value for *v* in slot 6 before jumping to *k*. One situation in which this isn't possible is receiving values from function calls. Guile has a calling convention for functions which currently places return values on the stack. A continuation of a call must check that the number of values returned from a function matches the expected number of values, and then must shuffle or collect those values to named variables. `$kreceive` denotes this kind of continuation.

`$kreceive` *arity k* [CPS Continuation]
Receive values on the stack. Parse them according to *arity*, and then proceed with the parsed values to the `$kargs` continuation labelled *k*. As a limitation specific to `$kreceive`, *arity* may only contain required and rest arguments.

`$arity` is a helper data structure used by `$kreceive` and also by `$kclause`, described below.

`$arity` *req opt rest kw allow-other-keys?* [CPS Data]
A data type declaring an arity. *req* and *opt* are lists of source names of required and optional arguments, respectively. *rest* is either the source name of the rest variable, or `#f` if this arity does not accept additional values. *kw* is a list of the form `((`*keyword name var*`) ...)`, describing the keyword arguments. *allow-other-keys?* is true if other keyword arguments are allowed and false otherwise.

Note that all of these names with the exception of the *var*s in the *kw* list are source names, not unique variable names.

Additionally, there are three specific kinds of continuations that are only used in function entries.

`$kfun` *src meta self tail clause* [CPS Continuation]
Declare a function entry. *src* is the source information for the procedure declaration, and *meta* is the metadata alist as described above in Tree-IL's `<lambda>`. *self* is a variable bound to the procedure being called, and which may be used for self-references. *tail* is the label of the `$ktail` for this function, corresponding to the function's tail continuation. *clause* is the label of the first `$kclause` for the first `case-lambda` clause in the function, or otherwise `#f`.

`$ktail` [CPS Continuation]

A tail continuation.

`$kclause` *arity cont alternate* [CPS Continuation]

A clause of a function with a given arity. Applications of a function with a compatible set of actual arguments will continue to the continuation labelled *cont*, a `$kargs` instance representing the clause body. If the arguments are incompatible, control proceeds to *alternate*, which is a `$kclause` for the next clause, or `#f` if there is no next clause.

9.4.4.3 Building CPS

Unlike Tree-IL, the CPS language is built to be constructed and deconstructed with abstract macros instead of via procedural constructors or accessors, or instead of S-expression matching.

Deconstruction and matching is handled adequately by the `match` form from `(ice-9 match)`. See Section 7.8 [Pattern Matching], page 706. Construction is handled by a set of mutually builder macros: `build-term`, `build-cont`, and `build-exp`.

In the following interface definitions, consider `term` and `exp` to be built by `build-term` or `build-exp`, respectively. Consider any other name to be evaluated as a Scheme expression. Many of these forms recognize `unquote` in some contexts, to splice in a previously-built value; see the specifications below for full details.

`build-term` *,val* [Scheme Syntax]
`build-term` (*$continue k src exp*) [Scheme Syntax]
`build-exp` *,val* [Scheme Syntax]
`build-exp` (*$const val*) [Scheme Syntax]
`build-exp` (*$prim name*) [Scheme Syntax]
`build-exp` (*$fun kentry*) [Scheme Syntax]
`build-exp` (*$const-fun kentry*) [Scheme Syntax]
`build-exp` (*$code kentry*) [Scheme Syntax]
`build-exp` (*$rec names syms funs*) [Scheme Syntax]
`build-exp` (*$call proc* (*arg ...*)) [Scheme Syntax]
`build-exp` (*$call proc args*) [Scheme Syntax]
`build-exp` (*$callk k proc* (*arg ...*)) [Scheme Syntax]
`build-exp` (*$callk k proc args*) [Scheme Syntax]
`build-exp` (*$primcall name param* (*arg ...*)) [Scheme Syntax]
`build-exp` (*$primcall name param args*) [Scheme Syntax]
`build-exp` (*$values* (*arg ...*)) [Scheme Syntax]
`build-exp` (*$values args*) [Scheme Syntax]
`build-exp` (*$prompt escape? tag handler*) [Scheme Syntax]
`build-term` (*$branch kf kt src op param* (*arg ...*)) [Scheme Syntax]
`build-term` (*$branch kf kt src op param args*) [Scheme Syntax]
`build-term` (*$throw src op param* (*arg ...*)) [Scheme Syntax]
`build-term` (*$throw src op param args*) [Scheme Syntax]
`build-term` (*$prompt k kh src escape? tag*) [Scheme Syntax]
`build-cont` *,val* [Scheme Syntax]
`build-cont` (*$kargs* (*name ...*) (*sym ...*) *term*) [Scheme Syntax]

`build-cont` (*$kargs names syms term*) [Scheme Syntax]
`build-cont` (*$kreceive req rest kargs*) [Scheme Syntax]
`build-cont` (*$kfun src meta self ktail kclause*) [Scheme Syntax]
`build-cont` (*$kclause ,arity kbody kalt*) [Scheme Syntax]
`build-cont` (*$kclause* (*req opt rest kw aok?*) *kbody*) [Scheme Syntax]
Construct a CPS term, expression, or continuation.

There are a few more miscellaneous interfaces as well.

`make-arity` *req opt rest kw allow-other-keywords?* [Scheme Procedure]
A procedural constructor for `$arity` objects.

`rewrite-term` *val* (*pat term*) ... [Scheme Syntax]
`rewrite-exp` *val* (*pat exp*) ... [Scheme Syntax]
`rewrite-cont` *val* (*pat cont*) ... [Scheme Syntax]
Match *val* against the series of patterns *pat...*, using `match`. The body of the matching clause should be a template in the syntax of `build-term`, `build-exp`, or `build-cont`, respectively.

9.4.4.4 CPS Soup

We describe programs in Guile's CPS language as being a kind of "soup" because all continuations in the program are mixed into the same "pot", so to speak, without explicit markers as to what function or scope a continuation is in. A program in CPS is a map from continuation labels to continuation values. As discussed in the introduction, a continuation label is an integer. No label may be negative.

As a matter of convention, label 0 should map to the `$kfun` continuation of the entry to the program, which should be a function of no arguments. The body of a function consists of the labelled continuations that are reachable from the function entry. A program can refer to other functions, either via `$fun` and `$rec` in higher-order CPS, or via `$const-fun`, `$callk`, and allocated closures in first-order CPS. The program logically contains all continuations of all functions reachable from the entry function. A compiler pass may leave unreachable continuations in a program; subsequent compiler passes should ensure that their transformations and analyses only take reachable continuations into account. It's OK though if transformation runs over all continuations if including the unreachable continuations has no effect on the transformations on the live continuations.

The "soup" itself is implemented as an *intmap*, a functional array-mapped trie specialized for integer keys. Intmaps associate integers with values of any kind. Currently intmaps are a private data structure only used by the CPS phase of the compiler. To work with intmaps, load the `(language cps intmap)` module:

```
(use-modules (language cps intmap))
```

Intmaps are functional data structures, so there is no constructor as such: one can simply start with the empty intmap and add entries to it.

```
(intmap? empty-intmap) ⇒ #t
(define x (intmap-add empty-intmap 42 "hi"))
(intmap? x) ⇒ #t
(intmap-ref x 42) ⇒ "hi"
(intmap-ref x 43) ⇒ error: 43 not present
```

```
(intmap-ref x 43 (lambda (k) "yo!")) ⇒ "yo"
(intmap-add x 42 "hej") ⇒ error: 42 already present
```

`intmap-ref` and `intmap-add` are the core of the intmap interface. There is also `intmap-replace`, which replaces the value associated with a given key, requiring that the key was present already, and `intmap-remove`, which removes a key from an intmap.

Intmaps have a tree-like structure that is well-suited to set operations such as union and intersection, so there are also the binary `intmap-union` and `intmap-intersect` procedures. If the result is equivalent to either argument, that argument is returned as-is; in that way, one can detect whether the set operation produced a new result simply by checking with `eq?`. This makes intmaps useful when computing fixed points.

If a key is present in both intmaps and the associated values are not the same in the sense of `eq?`, the resulting value is determined by a "meet" procedure, which is the optional last argument to `intmap-union`, `intmap-intersect`, and also to `intmap-add`, `intmap-replace`, and similar functions. The meet procedure will be called with the two values and should return the intersected or unioned value in some domain-specific way. If no meet procedure is given, the default meet procedure will raise an error.

To traverse over the set of values in an intmap, there are the `intmap-next` and `intmap-prev` procedures. For example, if intmap x has one entry mapping 42 to some value, we would have:

```
(intmap-next x) ⇒ 42
(intmap-next x 0) ⇒ 42
(intmap-next x 42) ⇒ 42
(intmap-next x 43) ⇒ #f
(intmap-prev x) ⇒ 42
(intmap-prev x 42) ⇒ 42
(intmap-prev x 41) ⇒ #f
```

There is also the `intmap-fold` procedure, which folds over keys and values in the intmap from lowest to highest value, and `intmap-fold-right` which does so in the opposite direction. These procedures may take up to 3 seed values. The number of values that the fold procedure returns is the number of seed values.

```
(define q (intmap-add (intmap-add empty-intmap 1 2) 3 4))
(intmap-fold acons q '()) ⇒ ((3 . 4) (1 . 2))
(intmap-fold-right acons q '()) ⇒ ((1 . 2) (3 . 4))
```

When an entry in an intmap is updated (removed, added, or changed), a new intmap is created that shares structure with the original intmap. This operation ensures that the result of existing computations is not affected by future computations: no mutation is ever visible to user code. This is a great property in a compiler data structure, as it lets us hold a copy of a program before a transformation and use it while we build a post-transformation program. Updating an intmap is O(log n) in the size of the intmap.

However, the O(log n) allocation costs are sometimes too much, especially in cases when we know that we can just update the intmap in place. As an example, say we have an intmap mapping the integers 1 to 100 to the integers 42 to 141. Let's say that we want to transform this map by adding 1 to each value. There is already an efficient `intmap-map` procedure in the `(language cps utils)` module, but if we didn't know about that we might do:

```
(define (intmap-increment map)
  (let lp ((k 0) (map map))
    (let ((k (intmap-next map k)))
      (if k
          (let ((v (intmap-ref map k)))
            (lp (1+ k) (intmap-replace map k (1+ v))))
          map))))
```

Observe that the intermediate values created by `intmap-replace` are completely invisible to the program – only the last result of `intmap-replace` value is needed. The rest might as well share state with the last one, and we could update in place. Guile allows this kind of interface via *transient intmaps*, inspired by Clojure's transient interface (`http://clojure.org/transients`).

The in-place `intmap-add!` and `intmap-replace!` procedures return transient intmaps. If one of these in-place procedures is called on a normal persistent intmap, a new transient intmap is created. This is an O(1) operation. In all other respects the interface is like their persistent counterparts, `intmap-add` and `intmap-replace`. If an in-place procedure is called on a transient intmap, the intmap is mutated in-place and the same value is returned.

If a persistent operation like `intmap-add` is called on a transient intmap, the transient's mutable substructure is then marked as persistent, and `intmap-add` then runs on a new persistent intmap sharing structure but not state with the original transient. Mutating a transient will cause enough copying to ensure that it can make its change, but if part of its substructure is already "owned" by it, no more copying is needed.

We can use transients to make `intmap-increment` more efficient. The two changed elements have been marked **like this**.

```
(define (intmap-increment map)
  (let lp ((k 0) (map map))
    (let ((k (intmap-next map k)))
      (if k
          (let ((v (intmap-ref map k)))
            (lp (1+ k) (intmap-replace! map k (1+ v))))
          (persistent-intmap map)))))
```

Be sure to tag the result as persistent using the `persistent-intmap` procedure to prevent the mutability from leaking to other parts of the program. For added paranoia, you could call `persistent-intmap` on the incoming map, to ensure that if it were already transient, that the mutations in the body of `intmap-increment` wouldn't affect the incoming value.

In summary, programs in CPS are intmaps whose values are continuations. See the source code of `(language cps utils)` for a number of useful facilities for working with CPS values.

9.4.4.5 Compiling CPS

Compiling CPS in Guile has three phases: conversion, optimization, and code generation.

CPS conversion is the process of taking a higher-level language and compiling it to CPS. Source languages can do this directly, or they can convert to Tree-IL (which is probably easier) and let Tree-IL convert to CPS later. Going through Tree-IL has the advantage of running Tree-IL optimization passes, like partial evaluation. Also, the compiler from

Tree-IL to CPS handles assignment conversion, in which assigned local variables (in Tree-IL, locals that are `<lexical-set>`) are converted to being boxed values on the heap. See Section 9.3.4 [Variables and the VM], page 830.

After CPS conversion, Guile runs some optimization passes over the CPS. Most optimization in Guile is done on the CPS language. The one major exception is partial evaluation, which for historic reasons is done on Tree-IL.

The major optimization performed on CPS is contification, in which functions that are always called with the same continuation are incorporated directly into a function's body. This opens up space for more optimizations, and turns procedure calls into `goto`. It can also make loops out of recursive function nests. Guile also does dead code elimination, common subexpression elimination, loop peeling and invariant code motion, and range and type inference.

The rest of the optimization passes are really cleanups and canonicalizations. CPS spans the gap between high-level languages and low-level bytecodes, which allows much of the compilation process to be expressed as source-to-source transformations. Such is the case for closure conversion, in which references to variables that are free in a function are converted to closure references, and in which functions are converted to closures. There are a few more passes to ensure that the only primcalls left in the term are those that have a corresponding instruction in the virtual machine, and that their continuations expect the right number of values.

Finally, the backend of the CPS compiler emits bytecode for each function, one by one. To do so, it determines the set of live variables at all points in the function. Using this liveness information, it allocates stack slots to each variable, such that a variable can live in one slot for the duration of its lifetime, without shuffling. (Of course, variables with disjoint lifetimes can share a slot.) Finally the backend emits code, typically just one VM instruction, for each continuation in the function.

9.4.5 Bytecode

As mentioned before, Guile compiles all code to bytecode, and that bytecode is contained in ELF images. See Section 9.3.6 [Object File Format], page 833, for more on Guile's use of ELF.

To produce a bytecode image, Guile provides an assembler and a linker.

The assembler, defined in the `(system vm assembler)` module, has a relatively straightforward imperative interface. It provides a `make-assembler` function to instantiate an assembler and a set of `emit-`*`inst`* procedures to emit instructions of each kind.

The `emit-`*`inst`* procedures are actually generated at compile-time from a machine-readable description of the VM. With a few exceptions for certain operand types, each operand of an emit procedure corresponds to an operand of the corresponding instruction.

Consider `allocate-words`, from see Section 9.3.7.9 [Memory Access Instructions], page 848. It is documented as:

`allocate-words` *s12:dst s12:nwords* [Instruction]

Therefore the emit procedure has the form:

`emit-allocate-words` *asm dst nwords* [Scheme Procedure]

All emit procedure take the assembler as their first argument, and return no useful values.

The argument types depend on the operand types. See Section 9.3.7 [Instruction Set], page 835. Most are integers within a restricted range, though labels are generally expressed as opaque symbols. Besides the emitters that correspond to instructions, there are a few additional helpers defined in the assembler module.

`emit-label` *asm label* [Scheme Procedure]

Define a label at the current program point.

`emit-source` *asm source* [Scheme Procedure]

Associate *source* with the current program point.

`emit-cache-ref` *asm dst key* [Scheme Procedure]
`emit-cache-set!` *asm key val* [Scheme Procedure]

Macro-instructions to implement compilation-unit caches. A single cache cell corresponding to *key* will be allocated for the compilation unit.

`emit-load-constant` *asm dst constant* [Scheme Procedure]

Load the Scheme datum *constant* into *dst*.

`emit-begin-program` *asm label properties* [Scheme Procedure]
`emit-end-program` *asm* [Scheme Procedure]

Delimit the bounds of a procedure, with the given *label* and the metadata *properties*.

`emit-load-static-procedure` *asm dst label* [Scheme Procedure]

Load a procedure with the given *label* into local *dst*. This macro-instruction should only be used with procedures without free variables – procedures that are not closures.

`emit-begin-standard-arity` *asm req nlocals alternate* [Scheme Procedure]
`emit-begin-opt-arity` *asm req opt rest nlocals alternate* [Scheme Procedure]
`emit-begin-kw-arity` *asm req opt rest kw-indices allow-other-keys? nlocals alternate* [Scheme Procedure]
`emit-end-arity` *asm* [Scheme Procedure]

Delimit a clause of a procedure.

The linker is a complicated beast. Hackers interested in how it works would do well do read Ian Lance Taylor's series of articles on linkers. Searching the internet should find them easily. From the user's perspective, there is only one knob to control: whether the resulting image will be written out to a file or not. If the user passes `#:to-file? #t` as part of the compiler options (see Section 9.4.2 [The Scheme Compiler], page 858), the linker will align the resulting segments on page boundaries, and otherwise not.

`link-assembly` *asm #:page-aligned?=#t* [Scheme Procedure]

Link an ELF image, and return the bytevector. If *page-aligned?* is true, Guile will align the segments with different permissions on page-sized boundaries, in order to maximize code sharing between different processes. Otherwise, padding is minimized, to minimize address space consumption.

To write an image to disk, just use `put-bytevector` from `(ice-9 binary-ports)`.

Compiling object code to the fake language, `value`, is performed via loading objcode into a program, then executing that thunk with respect to the compilation environment. Normally the environment propagates through the compiler transparently, but users may specify the compilation environment manually as well, as a module. Procedures to load images can be found in the `(system vm loader)` module:

```
(use-modules (system vm loader))
```

`load-thunk-from-file` *file* [Scheme Variable]
`scm_load_thunk_from_file` (*file*) [C Function]
Load object code from a file named *file*. The file will be mapped into memory via `mmap`, so this is a very fast operation.

`load-thunk-from-memory` *bv* [Scheme Variable]
`scm_load_thunk_from_memory` (*bv*) [C Function]
Load object code from a bytevector. The data will be copied out of the bytevector in order to ensure proper alignment of embedded Scheme values.

Additionally there are procedures to find the ELF image for a given pointer, or to list all mapped ELF images:

`find-mapped-elf-image` *ptr* [Scheme Variable]
Given the integer value *ptr*, find and return the ELF image that contains that pointer, as a bytevector. If no image is found, return `#f`. This routine is mostly used by debuggers and other introspective tools.

`all-mapped-elf-images` [Scheme Variable]
Return all mapped ELF images, as a list of bytevectors.

9.4.6 Writing New High-Level Languages

In order to integrate a new language *lang* into Guile's compiler system, one has to create the module `(language` *lang* `spec)` containing the language definition and referencing the parser, compiler and other routines processing it. The module hierarchy in `(language brainfuck)` defines a very basic Brainfuck implementation meant to serve as easy-to-understand example on how to do this. See for instance `http://en.wikipedia.org/wiki/Brainfuck` for more information about the Brainfuck language itself.

9.4.7 Extending the Compiler

At this point we take a detour from the impersonal tone of the rest of the manual. Admit it: if you've read this far into the compiler internals manual, you are a junkie. Perhaps a course at your university left you unsated, or perhaps you've always harbored a desire to hack the holy of computer science holies: a compiler. Well you're in good company, and in a good position. Guile's compiler needs your help.

There are many possible avenues for improving Guile's compiler. Probably the most important improvement, speed-wise, will be some form of optimized ahead-of-time native compilation with global register allocation. A first pass could simply extend the compiler to

also emit machine code in addition to bytecode, pre-filling the corresponding JIT data structures referenced by the `instrument-entry` bytecodes. See Section 9.3.7.6 [Instrumentation Instructions], page 841.

The compiler also needs help at the top end, adding new high-level compilers. We have JavaScript and Emacs Lisp mostly complete, but they could use some love; Lua would be nice as well, but whatever language it is that strikes your fancy would be welcome too.

Compilers are for hacking, not for admiring or for complaining about. Get to it!

Appendix A GNU Free Documentation License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Concept Index

This index contains concepts, keywords and non-Schemey names for several features, to make it easier to locate the desired sections.

!

#

(

.

/

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Procedure Index

This is an alphabetical list of all the procedures and macros in Guile. It also includes Guile's Autoconf macros.

When looking for a particular procedure, please look under its Scheme name as well as under its C name. The C name can be constructed from the Scheme names by a simple transformation described in the section See Section 6.1 [API Overview], page 99.

#

$

%

&

'

(

*

+

,

—

.

/

<

=

>

@

_

`

1

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Variable Index

This is an alphabetical list of all the important variables and constants in Guile.

When looking for a particular variable or constant, please look under its Scheme name as well as under its C name. The C name can be constructed from the Scheme names by a simple transformation described in the section See Section 6.1 [API Overview], page 99.

%

&

*

<

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Type Index

This is an alphabetical list of all the important data types defined in the Guile Programmers Manual.

$

&

(

<

A

B

R5RS Index

G

I

L

M

N

O

P

Q

R

S

T

V

W

Z