How to Scheme

Jonathan Lam

October 11, 2020

> . . . the half page of code on the bottom of page 13. . . was Lisp in itself. These were "Maxwell's Equations of Software!" This is the whole world of programming in a few lines that I can put my hand over.
>
> Alan Kay

First installation of the How-To: Tech series. This text makes the assumption that the reader already has a base understanding of an imperative programming language such as C.

Contents

1 Why?

> With a few very basic principles at its foundation, it [LISP] has shown a remarkable stability. Besides that, LISP has been the carrier for a considerable number of in a sense our most sophisticated computer applications. LISP has jokingly been described as "the most intelligent way to misuse a computer". I think that description a great compliment because it transmits the full flavour of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.
>
> Edsger W. Dijkstra, at his ACM Turing Award lecture

> Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.
>
> Greenspun's Tenth Rule

1.1 Why Lisp?

tl;dr: Lisp will make you a better programmer. And hopefully you'll love it too.

Lisp was a pioneer in programming languages in many ways. Lisp was first conceived in a research paper by John McCarthy at MIT in 1960, called "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I," and was heavily based on lambda calculus (which is Turing-complete). Besides FORTRAN, it is the oldest programming language in widespread use today.

Lisp pioneered ideas (some of which are still revolutionary) such as:

- source code as a data structure (homoiconicity) and the REPL
- recursion (and tail-call optimization)
- garbage collection
- first-class functions and continuations
- closures

The innovativeness of Lisp and Scheme was well-put by Brian Harvey, when defending Lisp as the language of choice for UC Berkeley's 61A (which uses Scheme and SICP as its textbook):

> During those entire 50 years, people have been saying "Lisp is impractical"; "Lisp is too slow"; "procedure calling is too expensive"; "only professors care about Lisp." They're still saying it. But meanwhile, users — real users — who would never dare give their bosses a program written in Lisp are demanding Lisp's ideas in the programming languages they do use. Today you take recursion for granted, but it was a radical idea when Lisp introduced it. (Fortran didn't have recursive procedures until fairly late in its history; the early personal computer users made do with BASIC, which, in those early versions, had no procedure calling at all.) Users of strongly typed languages demanded, and got, Lisp's heterogeneous lists. Today, the radical Lisp idea that's invading the mainstream is first class procedures. Guido van Rossum, the inventor of Python, hates Lisp, but he was dragged kicking and screaming by users into providing [a half-assed version of] lambda in Python. Even C++, a notorious can of worms, has added lambda in its most recent version. Lambda in Java is coming in 2013. In another decade they'll probably discover first class continuations.

1.2 Why Scheme?

First of all, note that there is no single Lisp language. Lisp is an idea that stemmed from McCarthy's original papers and includes languages that remain close in programming paradigms (and usually syntax). Thus Lisp has "dialects," or language implementations, that you can use directly.

Scheme is a dialect of Lisp popularized by the textbook Structure and Interpretation of Computer Programs, and is popular both for enthusiasts and researchers for its minimal, pure design. (That being said, not much comes out of the box.) Like the broader umbrella of Lisp, Scheme is characterized by a standard and is not a language in and of itself; it too has "dialects."

Other popular Lisp dialects include Clojure (JVM variant: has the benefits and deficiencies of the JVM) and Common Lisp (a larger and more complicated language than Scheme, perhaps more practical because of its industry extensions).

1.3 Why Chez Scheme?

Chez Scheme is an implementation of Scheme that is highly standards-compliant (see R6RS), well-maintained (by Cisco), well-documented (see Chez Scheme Version 9 User's Guide), and fast (see R7RS benchmarks).

There are many other popular implementation of Scheme, including but not limited to: Racket, Guile, Chicken, Racket, Stalin.

1.4 Practical Lisp

You won't see that much of Lisp out there. Lisp became very popular in the 1980s for its use in AI because of its ability to house complex, higher-order expressions very easily, but the "AI winter" and the increased performance of languages like C and C++ eventually took over.

Now, there are not many well-known examples of Lisp, and most of them are for enthusiast applications and domain-specific languages (DSLs). Emacs has its own dialect of Lisp (Emacs Lisp) used to script the editor. Some Linux tools like xbindkeys use Lisp (in this case, Guile Scheme) as a DSL (similar to Groovy's use as a DSL).

Of course, it's likely that many companies also use Lisp (usually Common Lisp or Clojure) as their secret sauce.

2 Syntax overview

2.1 Expressions

To understand Lisp, you have to understand "prefix notation" (a.k.a., Polish notation) of operators. This is as opposed to "infix operations," which is what you typically see with common binary math operators. Take these examples:

$$1 + 2 \equiv (+\ 2\ 1)$$
$$(2 + 4) * 3 \equiv (*\ (+\ 2\ 4)\ 3)$$
$$\frac{f(x+h) - f(x)}{h} \equiv (/\ (-\ (f\ (+\ x\ h))\ (f\ x))\ h)$$
$$3 \equiv 3$$
$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \equiv (/\ (-\ (\text{sqrt}\ (-\ (*\ b\ b)\ (*\ 4\ a\ c)))\ b)\ 2\ a)$$

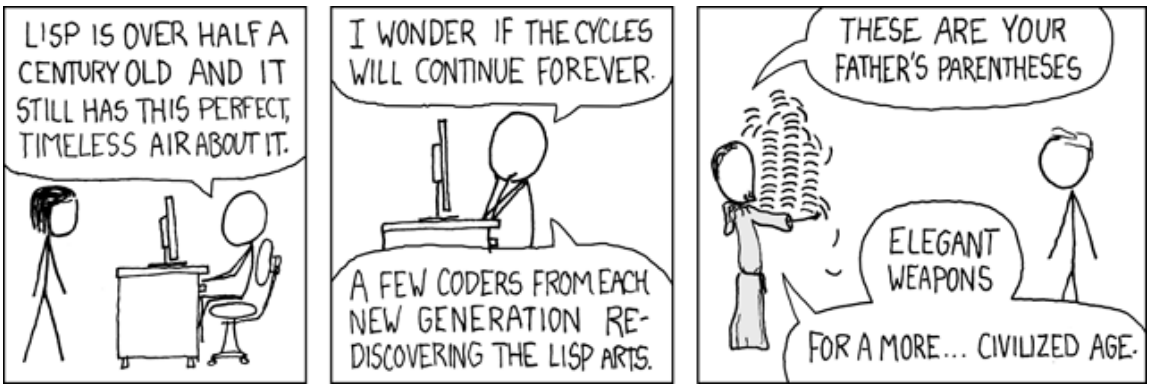

Figure 1: xkcd: Lisp Cycles

It may be confusing at first because the operator is not between the operands, but this is not always a bad thing. Some benefits are:

- It is much easier to parse. (If you've ever written a calculator program, you'd know. You don't need a complicated syntax parser; all you need is a stack for an efficient (linear-time) parsing. In fact, many old calculators used to require Polish notation as input.)
- There is a very clear separation between operator and operands.
- Order of operations becomes explicit with parentheses. (If functions are limited to a fixed number of input arguments (fixed arity; no variadic arguments or overloading), then no parentheses are needed at all (!!!) and order of operations is still explicit. I.e., if + and / were explicitly binary operations (i.e., take exactly two operands), then the expressions

$$(2 + 4) * 3 \equiv *\ +\ 2\ 4\ 3$$
$$2 + 4 * 3 \equiv +\ 2\ *\ 4\ 3$$

are completely unambiguous. However, Lisp often does allow for variadic arguments, so parentheses are necessary.)

- Variadic arguments on conventionally fixed-arity operators only requires one use of the operator (when parentheses are used). E.g., when multiplying five numbers in infix notation vs. prefix notation:

  $$a * b * c * d * e \equiv (* \ a \ b \ c \ d \ e)$$

  The latter is arguably nicer to read.

- This makes operators look like functions. In fact, in Lisp, operators and functions become synonymous; you can think of an operator as a function, where the operand(s) is (are) the parameters to the function. (This may be familiar to you if you've used Python or C++, where operator overloading looks just like function overloading.) In fact, functions conventionally already follow the prefix notation, in that invocation involves the name of the function (the operator) followed by its parameters (the operands).

- From a pedagogical view, it changes your perspective on expressions: rather than writing operators between operands, you have to write out all of the operands before closing the parentheses on the operator. Theoretically, this is closer to the way function (operator) computation works, since all parameters (operands) are evaluated before a function (operator) is applied to it.

2.2 Functions

As mentioned in the previous section, there's little distinction between operators and functions. You can think of the addition operator + as a function that takes an arbitrary number of numeric inputs and returns their sum. Other examples of builtin functions include:

$$\text{(abs -5)} \Rightarrow 5$$
$$\text{(1+ 4)} \Rightarrow 5$$
$$\text{(mod 7 2)} \Rightarrow 1$$

(Note that 1+ is the name of a function in Scheme that increments its argument by 1. In some languages, this would not be considered a valid function name.)

You can declare your own functions as well. As mentioned previously, functions are essentially operators: they take parameters and return some result based on them. The function $2x + 3 * y$ looks like the following:

```
(lambda (x y) (+ (* 2 x) (* 3 y))))
```

To use this function, you need to place it before its operands, like for all of the other functions. For example, the previous expression, applied with $x = 5$ and $y = 2$ looks like:

```
((lambda (x y) (+ (* 2 x) (* 3 x))) 5 2)
```

But this is a little hairy. What if we want to reuse this function? We can declare a variable to store this function. I.e., we can do the following:

$$f(x, y) \coloneqq 2x + 3y$$
$$f(2, 3)$$
$$f(5, 2)$$

```
(define f (lambda (x y) (+ (* 2 x) (* 3 x))))
(f 2 3)
(f 5 2)
```

A shorthand syntax for defining a (named) function is:

```
(define (f x y)
  (+ (* 2 x) (* 3 y)))
```

Note that this `define` syntax can be used to assign names to other expression as well.

2.3 Linked lists come free

The original name LISP came from "LISt Preprocessor" in McCarthy's original paper.

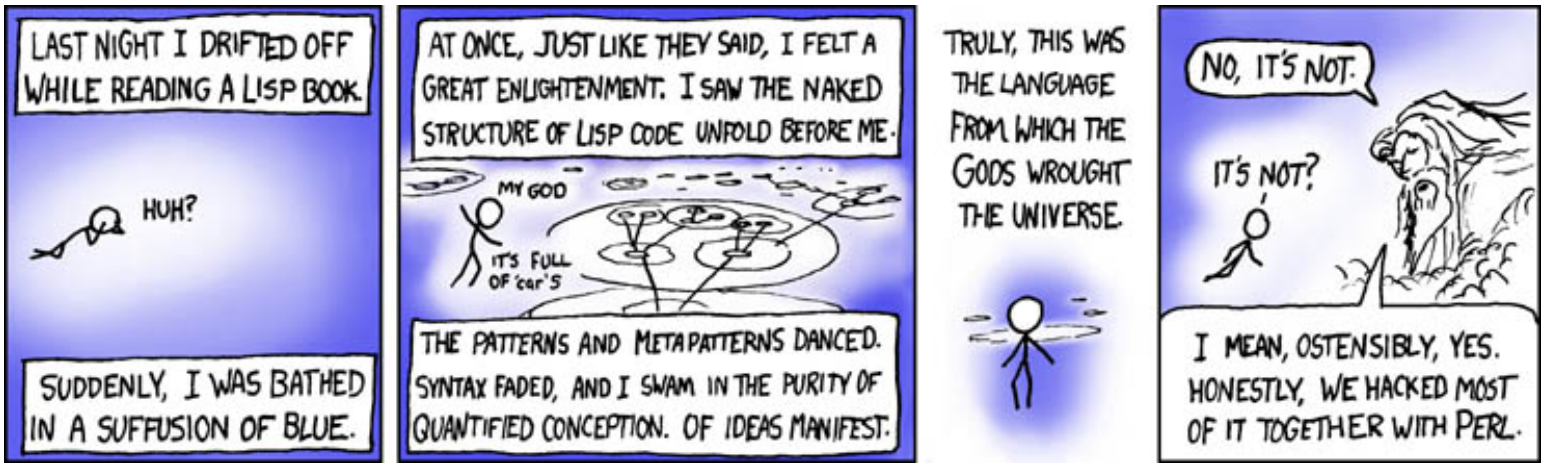

Figure 2: xkcd: LISP. (Note the reference to `cars`.)

TODO: working here

> What makes Lisp cool is not that programs are data (because all programs are data), but that they are a particular kind of data. In most programming languages, programs are strings. Strings are in fact data. In Lisp, programs are not strings, they are linked lists (that happen to have a string representation). And this turns out to make all the difference.
>
> Rondam Ramblings

2.4 Code as a data structure (homoiconicity)

TODO: working here

2.5 Control flow: loops, recursion, and conditionals

TODO: working here

2.6 Other data types and keywords

Besides lists, there are also lambdas, vectors, strings, hashtables (which are slightly implementation-dependent), symbols, and record types. All of these are described in R6RS.

3 Functional vs. imperative (vs. objected-oriented) paradigm

Some of the definitions you've seen above may be different than those of a conventional textbook on C or Java, and more similar to that of a math class (e.g., something along the line of $f(x) = 2x+3$).

R6RS (the current Scheme standard) defines a procedure (they do not use the term "function," but it is functionally the same term) as:

> A procedure is, slightly simplified, an abstraction of an expression over objects.

The K&R C bible, on the other hand, defines the function as:

> In C, a function is equivalent to a subroutine or function in Fortran, or a procedure or function in Pascal. A function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation.

There is a separation between functions and other expressions (e.g., variables, statements, and literal values). Here, functions are more of an encapsulation of some computation, potentially with some return value; however, in Lisp (and other declarative programming languages), a function is purely defined by its result (i.e., it can be thought of as an expression, the only difference being that it can take arguments and thus be flexibly reusable).

Another difference is I've only mentioned variables offhandedly when talking about Scheme and Lisp, but rather focused on the idea of expressions. Usually, the second order of business after showing basic arithmetic expressions in programming tutorials is how to define variables and use or redefine their values. We can do this in Scheme using the `define` and **`let`** keywords, but typically we can mostly avoid using many variable declarations except for top-level functions, and small, localized bindings. Rather than storing and manipulating variables with instructions, we are composing expressions on top of other expressions. This goes hand-in-hand with the idea of immutability, discussed later.

Many modern programming languages are often called "multi-paradigm," in that they have aspects that fall under each of these categories. Languages like Python, JavaScript, or Scala, for example, allow imperative looping constructs but also declarative higher-order functions to perform the same tasks, declarative-style. Many of these languages also support object-oriented programming as well. Thus the distinction between paradigms is, in the case of many languages, more of a conceptual mode than anything else. Even Scheme has some imperative programming concepts embedded in the language; it is very hard to completely neglect certain aspects of it.

3.1 Imperative paradigm

tl;dr: Everything is an instruction (statement).

An example would be to filter a list (array) based on some predicate (e.g., positive numbers only) in C, and generate a new list (array) without mutating the original:

```
int l1[] = {-5, 3, 2, 4, 0},
        len = sizeof(l1) / sizeof(int),
        *l2 = (int *) malloc(sizeof(l1)),
        i, j;

for (i = j = 0; i < len; ++i)
```

```
        if (l1[i] > 0)
                l2[j++] = l1[i];

len = j;
l2 = (int *) realloc((void *) l2, len * sizeof(int));
```

Even if a data structure like a linked list were defined and used here, the implementation would (most likely) still involve worrying about looping over the linked list data structure and deallocation of the original list, if it wasn't needed anymore.

> Declarative programming is a paradigm . . . that expresses the logic of a computation without describing its control flow" (imperative programming does the latter).
>
> Wikipedia article on declarative programming

3.2 Declarative (and functional) paradigm

tl;dr: Everything is an expression.

The Wikipedia quote is very good. Declarative programming is a broader term than functional programming, and contrasts strongly with imperative programming. Rather than tell the computer the exact steps on how to perform a task, declarative programming involves providing higher-level operations to act on. This typically involves higher-order functions (and therefore, treating functions as first-class objects), which is called functional programming. Most modern declarative languages are functional and treat functions as first-class objects (e.g., JavaScript, Python, and even C++ after the introduction of lambdas), but some do not require this complexity (e.g., SQL).

Using the previous example of sorting a list, this Lisp operation uses the higher-order function `filter`, which takes a predicate function as a parameter:

```
(let ([l (list -5 3 2 4 0)])
  (filter (lambda (x) (> x 0)) l))
```

You can easily tell that the intent of the code is very clear: we attempt to filter an input list with a function that tests whether the list element is positive. This kind of simplicity is key to making code more legible and less prone to semantic error.

Two concepts that are often associated with declarative programming are *immutability* and *pure functions*. Immutability is the idea that an object never changes once it is created. Every time you modify a variable, for instance, it creates a new instance in memory and doesn't touch the old value in memory. While this may seem wildly inefficient, note that the new object can still reference the unchanged parts of the old object, so only the changed part has to be copied. E.g., modifying one field of an immutable 1000-field object (see next section) would involve creating a new object, allocating a new field and referencing it from the object, and referencing the other 999 fields from the old object; very little new memory has to be allocated. Immutability can actually

save space when doing things like copying an object; no extra memory is allocated until the copy or the original is modified. Enforcing immutability also helps with predictability and maintainability, as its contents will not change unless you modify it; in OOP, it is possible that the value is modified in memory by some other code or reference to that memory. Pure functions are functions that do not modify any other variables or cause any other "side effects," which has the same beneficial effect on state predictability. It is often considered good practice in declarative programming to limit yourself to pure functions when possible.

3.3 Object-oriented paradigm

tl;dr: Everything is an object.

In short, object-oriented programming can be thought of as abstracting imperative programming into a more declarative paradigm by modeling all code as objects, and the actions performed on those objects. Typically the code is still imperative, in that it is instruction-by-instruction, but usually this is grouped into "methods" that are performed on objects, similar to how higher-level functions use functions to perform actions on other objects in declarative programming.

For example, in Java:

```
List<Integer> l = Arrays.stream(new Integer[] {-5, 3, 2, 4, 0})
        .filter(x -> x > 0)
        .collect(Collectors.toList());
```

What you can draw from this example is that there is a fair amount of abstraction going on, with many chained method calls being applied on objects, and again the higher-order function `filter`, this time applying a Java lambda on an object rather than a simple Lisp list expression.

This is a newer paradigm with much literature behind it, so I will not go into it in depth.

4 Resources

4.1 Interesting reads

Scheme vs Python (i.e., why SICP and Lisp as a teaching language) https://people.eecs.berkeley.edu/~bh/proglang.html

Lisp in Less Than 200 Lines of C https://carld.github.io/2017/06/20/lisp-in-less-than-200-lines-of-c.html

Greenspun's Tenth Rule https://en.wikipedia.org/wiki/Greenspun's_tenth_rule

Lisp as the Maxwell's equations of software http://www.michaelnielsen.org/ddi/lisp-as-the-maxwells-equations-of-software/

Yes, code is data, but that's not what makes Lisp cool http://blog.rongarret.info/2018/02/yes-code-is-data-but-thats-not-what.html

Where lisps dynamic nature really shines? https://www.reddit.com/r/lisp/comments/cz1gj5/where_lisps_dynamic_nature_really_shines/

4.2 Reference texts

Structure and Interpretation of Computer Programs https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book.html

Common Lisp: A Gentle Introduction to Symbolic Computation https://www.cs.cmu.edu/~dst/LispBook/book.pdf

R6RS: The Revised[6] Report on the Algorithmic Language Scheme http://www.r6rs.org/

4.3 Relevant xkcd

Lisp https://xkcd.com/224/

Lisp cycles https://xkcd.com/297/

4.4 Learning resources

Exercism https://exercism.io/