

Semantic Analysis of the C language

In this unit we begin discussion of semantic analysis with an emphasis on the C language.

Compile-Time vs Run-Time

We can speak of the compiler as existing in two worlds: compile-time, when the compiler is actually executing, and run-time, when the target program will be executing. The compiler performs actions, creates data structures, etc. at compile time which result in the emission of code which, in turn, when executed at run-time, creates and manipulates values and takes other actions specified in the original source code.

In contrast, an interpreter exists in only one time, because source code is acted upon immediately. We could think of a compiler as an interpreter, the actions of which are to emit the target code (e.g. assembly language on the target machine).

Certain values can be computed at compile-time, such as the size of (most) variables, expressions involving constants, etc. Others, such as the value inside of a variable, are indeterminate at compile-time. Part of the optimizer's job is to trace out the flow of control and data through the program and attempt to predict, to the extent possible, useful constraints on values which may lead to optimization of target code.

Syntax-Directed Translation

In unit 2 we have seen the use of embedded (semantic) actions within a yacc/bison grammar, and the passing of synthesized and inherited attributes along the value stack. It is possible to use these tools to perform the translation from source to target language in **one pass**, as the source language is being parsed. This is also known as **Syntax-Directed Translation**. The example below compiles expressions to a simple intermediate language in which there is an unbounded number of temporary registers:

```
%debug
%error-verbose
%{
#define YYDEBUG 1
#define YYSTYPE char *

int tmpcounter;

char *tmp()
{
 char buf[256];
        sprintf(buf,"%%TMP%04d",++tmpcounter);
        return strdup(buf);
}

%}
%token NUM
%token NL
%left '+' '-'
%left '*' '/'
%nonassoc '('
%%
start:
        expr_line
        |start expr_line
        ;

expr_line:
        expr NL
        |NL
        ;

expr:   NUM             {$$=$1;}
   |    expr '+' expr   {printf("\t%s=ADD %s,%s\n",$$=tmp(),$1,$3);}
   |    expr '-' expr   {printf("\t%s=SUB %s,%s\n",$$=tmp(),$1,$3);}
   |    expr '/' expr   {printf("\t%s=DIV %s,%s\n",$$=tmp(),$1,$3);}
   |    expr '*' expr   {printf("\t%s=MUL %s,%s\n",$$=tmp(),$1,$3);}
   |    '-' expr %prec '(' {printf("\t%s=MUL %s,-1\n",$$=tmp(),$2);}
   |    '(' expr ')'    {$$=$2;}
   ;

%%
main()
{
        yydebug=0;
        yyparse();
}

yyerror(char *err)
{
        fprintf(stderr,"syntax error:%s\n",err);
}
```

The `$$` value of each symbol is the temporary register in which its run-time value will be found. E.g.

```
INPUT:
3*4+10*5
OUTPUT:
        %TMP0003=MUL 3,4
        %TMP0004=MUL 10,5
        %TMP0005=ADD %TMP0003,%TMP0004
```

(Note that in the above example code, there is no "garbage collection." The `strdup` contains an implicit `malloc` and at no point is there a `free`. This would be a "memory leak" problem in a real compiler)

At one time, a one-pass compiler was extremely desirable, because memory and disk space was limited. However, writing a one-pass compiler, using syntax-directed translation, constrains us to following the exact parse tree that the syntax generates. This becomes cumbersome as language complexity grows, because the structure of the source language grammar might not reflect well the order in which target code needs to be generated.

In modern compilers, the embedded semantic actions create an intermediate representation (IR) in memory, which is then re-processed one or multiple times, refining and optimizing it, and finally transforming it into the target language. We'll discuss IR in more detail in Unit 5. Not all of the IR corresponds to executable (run-time) constructs. E.g. the compiler may use the same techniques to build compile-time representations of data types.

Abstract Syntax Trees

Many compilers use a graphical form of representation known as an **Abstract Syntax Tree** (AST). It is called abstract because, while it is related to the parse tree, it is not literally the parse tree. Certain grammar symbols, which exist purely for parsing purposes, might not be reflected in the AST, or the AST may have a different ordering or structure. The AST may not actually even be a "tree," rather it is often in the form of a directed graph.

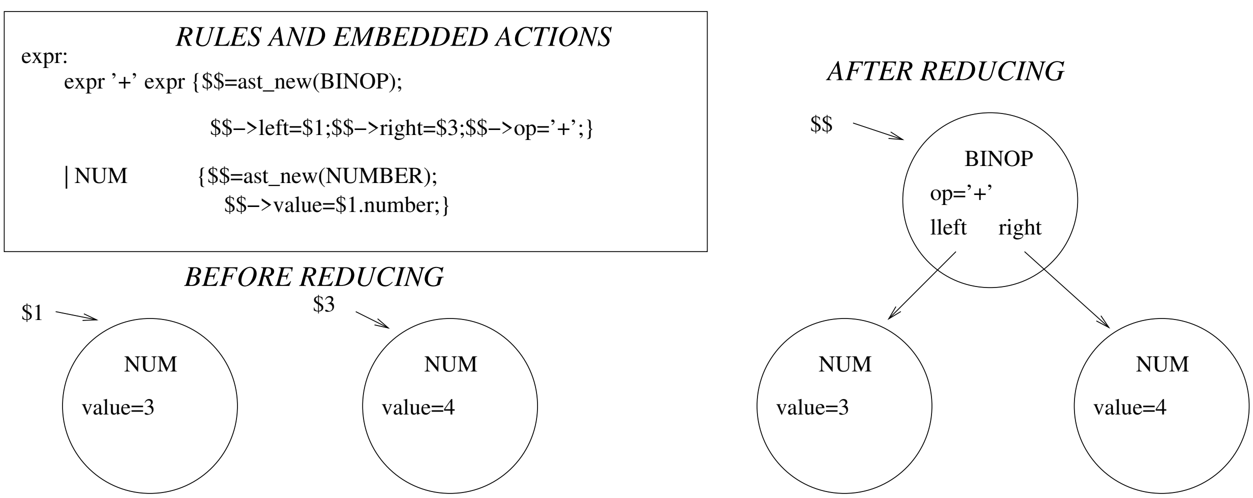
An AST consists of nodes and edges. Each node may have attributes associated with it. In implementing an AST in C, the nodes are often held in dynamically allocated structs, with the edges being pointers within those structs. Each node has a different set of attributes, depending on the node type, and this makes the AST a polymorphic data structure, in that edges from one AST node can point to another AST node of arbitrary type.

In many cases, construction of an AST using embedded actions is straightforward and follows naturally from the actual syntax. As an example, let's look at a passage from the "expression calculator" grammar, re-written to generate an AST:

```
/* Note: some syntax needed to implement a polymorphic AST has been elided */
expr:     NUM                {$$=ast_new(AST_NUM);
                                        $$->number=$1.number;}
          |expr '+' expr     {$$=ast_new(AST_BINOP);
                                        $$->op='+';
                                        $$->left=$1;
                                        $$->right=$2;}
```

In the first rule, an AST node is created for a terminal, and is filled in with a copy all of the information needed regarding that terminal. This information had been passed up from the lexer in `yylval` and was automatically copied to the yacc/bison value stack as `$1`.

In the second rule, assume that the semantic values from the two instances of the rhs non-terminal `expr` are pointers to correct AST representations of those subexpressions. Then by creating a new AST node to represent the addition operation, and connecting it to the two subexpressions, the semantic value associated with the lhs `expr` is, by induction, the correct representation of that expression.



The Symbol Table

Symbolic names are a critical part of any programming language. The symbol table subsystem of a compiler is responsible for keeping track of which symbolic names have been seen, and the attributes associated with each instance. In the C language, these names are called "identifiers", and there are 10 different purposes for which identifiers may be used (not counting pre-processor uses):

- variable name (including function formal parameters)
- function name
- typedef name

• enumeration constant
• struct tag
• union tag
• enum tag
• label
• struct member
• union member

Note that, from a lexical standpoint, there is no distinction between these uses. The regular expression pattern for a variable name is clearly the same as that for a function name, or a structure tag.

The symbol table could be conceptually thought of as having the following abstract interface:

• `create`: Create a new, empty symbol table.
• `destroy`: Destroy a symbol table including any storage which it consumes.
• `lookup`: Given an existing symbol table and scope, a name, and a namespace class, return the associated symbol table entry, if it exists, otherwise return the fact that the symbol does not exist. The lookup operation must go through the stack of scopes until the outermost (file scope) before concluding that the entry does not exist. The entry contains attributes, which will be discussed below.
• `enter`: Given an existing symbol table, a name, a namespace class, and a set of attributes, enter this symbol in the table. A boolean parameter can determine whether, if a symbol with the same name and namespace already exists in the table, to replace the definition or to return an error.

Identifiers and attributes

The attributes which are stored with each symbol will vary depending on the use of the identifier (e.g. the 10 identifier classes in C). For error reporting purposes, all symbol table entries should contain the file name and line number at which the symbol was first defined (entered into the symbol table). This will greatly assist in giving useful error messages in the case of duplicate or conflicting definitions. In addition, for discussion purposes, here is an example (not necessarily exhaustive or complete) of the attributes which the compiler needs to maintain during compilation for each identifier class:
• variable: type, storage class, offset within stack frame (for automatic storage class only)
• function: type (includes return type and arguments), storage class (extern or static), presence of `inline` specifier, whether a definition for the function (the actual code) has been seen yet.
• struct/union tag: symbol table containing member definitions, whether definition is complete or not.
• enum tag: once the definition is complete, nothing. One might try to link back to all of

the enum constant names which are defined under this tag, for error reporting purposes.

• enum constant: value. One might want to link to the enum tag too. In the C language, identifiers which are enum constants simply become ints in the expression, i.e. the enum tag does not propagate. If `x` is of type `enum e`, the type of the expression `(x)` is `int`, not `enum e`.

• statement label: Intermediate code or assembly language label of the basic block begun by the C-language label. (This will make more sense once code generation is discussed).

• typedef name: equivalent type.

• struct/union member: type, offset (within struct only), bit field width and bit offset.

Namespaces

It is possible in C for the same identifier name, in the same scope, to refer to different things, because that identifier use would be unambiguous based on surrounding syntax. The following are the groups, or **name spaces**, which C defines:

• Labels, because they are used either with the `goto` keyword, or with the syntax `label ':' statement`. Both uses are unambiguous.

• Tags: e.g. `foo` in `struct foo`, have their own namespace. Their use is unambiguous because it is always preceded by the struct, union or enum keyword. However, struct, union and enum tags are all lumped together. E.g.

```
int f;
struct f {int f;} g;                    //OK
union f {int h;} u;           //Not OK, tag f is already used
```

• Each struct or union definition creates an independent, private namespace which is in effect a mini-scope or mini- symbol table. The member names of any struct or union, therefore, can not conflict with member names of any other struct or union, nor with any of the other identifier classes. This is possible because member names can only be used as the right-hand operand to a . or -> operator, and the structure or union to which they belong is then implied by the type of the left-hand operand.

• All other identifier classes: enum constants, typedef names, variable or function names.

Scopes and Visibility

A particular instance of an identifier is not necessarily **visible** at every line of the program. The C standard identifies the following *scopes* of visibility:

• File scope: an identifier with file scope is visible from its point of declaration until the end of that `.c` file (not considering pre-processor directives such as `#include`). File scope applies to declarations seen outside of any function definitions, i.e. "global" declarations.

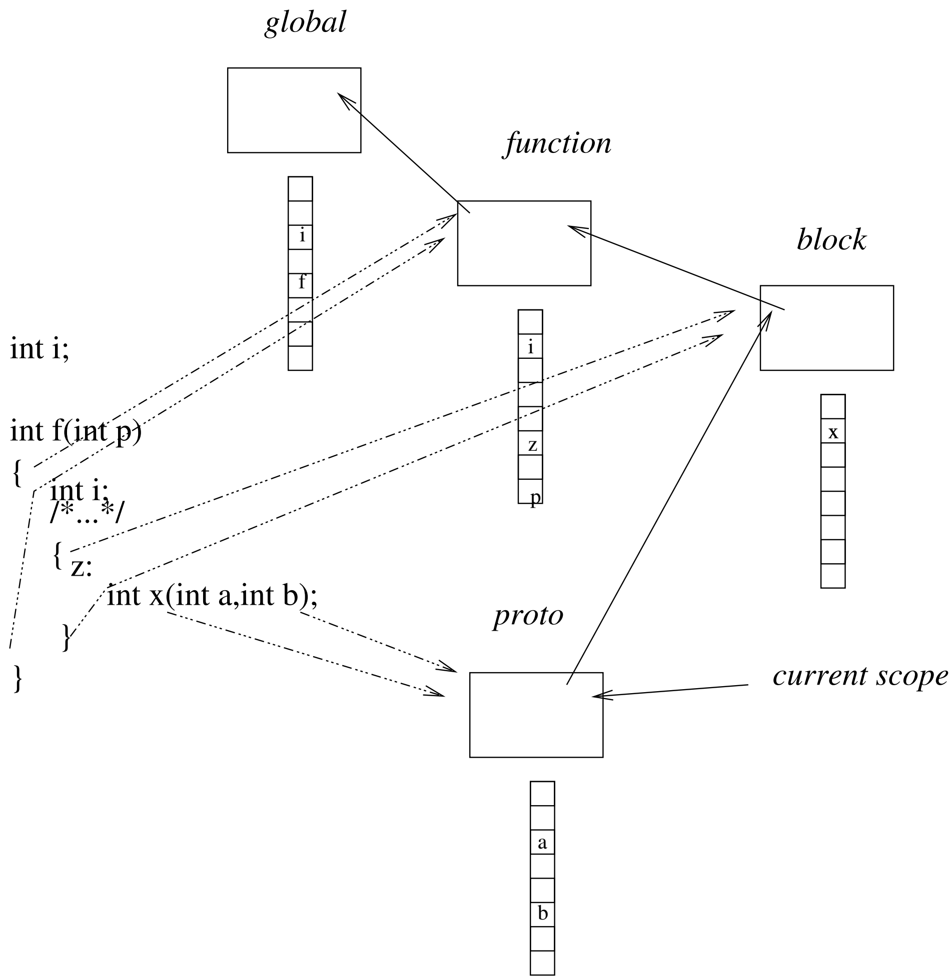
• Block scope: An opening curly brace within the body of a function introduces a new block scope. Identifiers declared within a block are visible from that point until the end (closing curly brace) of that block. Note that just as blocks are nestable, so too are block scopes. Block scopes can be thought of as translucent with the innermost block on top. Declarations which happen in an inner block obscure the visibility of identifiers appearing in an outer block, until that inner block is done. Also, declarations in a block scope may obscure file scope.

• Function scope: Function scope is associated with the outermost block of a function. Function and block scopes are very similar, the primary distinction being in the handling of statement labels. Labels are always placed into the scope of the function in which they appear. If a label is defined inside of an inner block scope, it is nonetheless visible throughout the entire function. Note that in the definition of a function, the formal parameters (which appear inside the parentheses, plus the declarators between there and the opening brace in classic C) have a scope as if they had been declared at the beginning of the main block of the function.

• Prototype scope: The C standard uses this term to handle the case of identifiers which appear in a prototyped function declarator, e.g.:

```
unsigned fu(int a,double d);
```

This is not the same as the *definition* of the function, which includes the actual code associated with it. The identifiers which appear in the prototype are not meaningful, just the list of types of the arguments, and so the introduction of this term "prototype scope" is of dubious value to the compiler-writer. During a function definition, however, we'll see that the prototype scope gets promoted to be the function's scope.



In implementing a symbol table, one approach might be to create a stack (or linked list) of scopes, each pointing to its associated symbol table. When looking up an identifier, the symbol table at the top of the stack, corresponding to the innermost scope, is consulted first, then if nothing is found, the next one, etc. until the bottom of the stack, representing file scope, is reached. To handle the different namespaces, one could maintain an array of symbol tables for each scope, one element per name space. Alternatively, one could add a namespace tag to each symbol table entry and only consider a match to occur if both the name and the namespace are identical.

Point of Identifier Installation

The point at which an identifier is installed into the symbol table depends on identifier class:

• variable name: End of declarator (see below "Declarators and Abstract Declarators") in which variable name is mentioned.

• function name: End of declarator in which function name is mentioned, or point of use in the case of implicit declaration (see below)

• typedef name: End of declarator in which typedef name is mentioned.
• struct/union tag: See discussion of structure and union incomplete types.
• struct/union member: End of declarator, appearing within a declaration within a struct/union definition, in which member name is mentioned.
• enum tag: end of enum definition (closing brace)
• enum constant name: after name is mentioned (possibly with explicit value) in enum definition.
• label: definition point at labeled statement, or implicit forward declaration when used in goto.

Declarators and Abstract Declarators

The term **declarator** is defined in the C standard's grammar as the syntactic element in which the identifier is mentioned, and resembles a tiny subset of the C expression syntax. For example, in the following code:

```
const int a,*pa,*ap[10];
```

The tokens `const int` form the **declaration specifiers**, and each of the items delimited by commas is a declarator. Declarators are formed with identifiers, pointer constructs, array constructs, function constructs, and parenthesized declarators. They follow the same rules of operator precedence as the C expression grammar, and indeed C's declaration syntax may informally be described as "you declare it like you would use it later on." There are also **abstract declarators** which are identical to "normal" declarators except for the ommision of identifiers. They are used in abstract type names, which can appear only in cast and sizeof expressions, or in function prototypes. For example `(int **[])` is an abstract type of (array(unknown size, of (pointer to(pointer to(int))))) This "missing identifier" syntax is often confusing.

Forward Declarations

Generally speaking, in the C language, identifiers must be declared in some visible scope before they are used. When a name is used prior to explicit declaration or definition, that is known as a **forward declaration**. This is allowed only under these specific circumstances:

• struct and union tags may be referenced before their member list is declared. This is known as an incomplete struct/union type.
• statement labels may be used in a `goto` before their definition point (a labeled statement) is seen. There may be 0 or more `goto` uses before the definition point, but only one definition point is allowed. At the time of the forward reference, the label may be installed in the symbol table as an incomplete label. Then upon seeing the labeled statement, the symbol table entry would be marked as complete. If no such definition point is encountered by the end of the function, this is a fatal error.

• functions may be called without first declaring the identifier to be the name of a function. When this occurs, an **implicit declaration** is made in the current scope, and the identifier is assumed to have type `(int ())`, a function returning int and taking an unknown argument list.

Re-declaration of Identifiers

Generally it is an error when a declaration attempts to install a name in the symbol table when that same name (in the same namespace) is already installed in the current scope. There are exceptions:

• Redeclaration of a function name is permitted as long as the new declaration matches the existing one in terms of type (there are complicated rules for determining whether two function prototypes are equivalent) and storage class.

• Variable declarations with an explicit `extern` storage class do not conflict with previous or future declarations of the same variable without the `extern`, so long as the declared types are equivalent, and the subsequent declaration does not have storage class `static`, `auto` or `register`. This allows, in a compilation from multiple .c files, for the variable to be declared in a header file as `extern`, with that same header file `#include`'d in the .c file in which the variable is actually declared.

• Variable declarations in global scope without an explicit storage class and without an initializer may be repeated, so long as the declared types are equivalent. E.g. `int i; int i;` is a valid C program. This is known as the "Common Block" model.

Storage Class and Duration

The scope of a variable (where it is lexically visible in the program) is not necessarily the same as its lifetime. Variables declared in global (file) scope have a duration which is the duration of the entire program. Variables declared inside a function scope generally have a duration which is that of the function. However, `static` variables declared within a function have local scope, but global duration.

The C syntax allows for certain storage class specifiers to be given in a declaration or function definition to control duration. These are **auto**, **register**, **extern** and **static**. The semantics of these storage classes are well documented in the C standard and other texts and will not be covered here. However, while these four storage classes define the semantics that are visible to the programmer, as compiler writers we need to consider additional storage classes. For example, a local variable and a formal function parameter both nominally have `auto` storage class. However, when we generate target code, we need to distinguish between these two conditions, because the way we access local variables in assembly language will differ from parameters. We'll see some more examples and issues in the unit on target code generation.

Storage class can only be applied to declarations of a variable or function. As we shall see in a later unit, all that a storage class does is determine how a variable is accessed at run-time, or in the case of a function, whether the function name is made visible outside of the local file. Therefore, it is meaningless, and not allowed by C, to attach a storage class to a structure, union or enum definition (i.e. the definition of the tag with its members, as opposed to a declaration of a variable of struct, union or enum type), to a typedef, to a label, or to a structure or union member (their storage class is "inherited" from the structure or union in which they appear).

When a storage class keyword is omitted in a function or variable declaration, a default storage class is assumed. In some cases, the lack of a storage class is not quite the same as when that same default storage class is specified explicitly. We'll see a good example of that in a subsequent unit when the linker is covered, with regard to the difference between `int i;` and `extern int i;` in global scope. Or the reader may refer to H&S or the C standard.

The storage class is an attribute of a variable or function symbol, distinct from the type. E.g.

```
static int a;
```

The type of a is `int`, not `static int`.

Type Systems

Almost all programming languages have type systems. Type systems are a way of allowing programmer to express intent rather than mechanics. The compiler must keep track of the types of variables, expressions, etc. in order to generate the proper code.

Enforcement of type rules catches certain types of programmer errors. Some languages are very strongly typed, to the extent that they simply can't do certain things which violate their type model. Other languages such as C are more loosely typed: the compiler may complain, but the programmer can generally override type-safety decisions.

Programming languages can have both static and dynamic typing. C has only static. Dynamic typing requires run-time support, e.g. inserting a tag into objects to encode their type and cause the execution of the appropriate method. Interpreted languages typically have greater support for dynamic typing.

We'll soon see how ASTs can be used to represent type information.

Types in C

This is not intended to be a comprehensive reference on the C language, but merely to illustrate the issues facing the writer of a compiler for C. The reader is referred to the C language standard, the textbook, or Harbison & Steele.

Central to the compiler's operation is the ability to represent C language data types, including potentially complicated types, and to manipulate and analyze these type representations. Typically, an AST is used to represent types as they are constructed within declarations or abstract type names (used in prototypes and casting). ASTs can also be used to represent expressions. The AST can be queried in a number of ways:

• Calculate the `sizeof` an expression or type
• Calculate the resultant type when an operator is applied to a subexpression or subexpressions
• Generate code to calculate the value (rvalue) of an expression
• Generate code to calculate the lvalue of an expression. The lvalue is a name or expression that can be used to store a value.
• Generate code to calculate the run-time address of an expression

In the pages that follow, we will see how types can be represented with ASTs, and how some of the above analysis can be performed. Some analyses, particularly those involving code generation, will be covered in later units.

Scalar types

C defines certain basic, scalar arithmetic types: the integers and the reals. In C99, the complex type is also defined. Integers consist of char, short, int, long and long long sizes. Although commonly these are represented in 8, 16, 32, 32 and 64 bits, the C standard does not require any particular number of bits for any given type. In fact, the int type is specifically designed to be the "best" integer type for a certain target architecture. On a native 64-bit system this might be 64 bits, while at the other end of the spectrum, it might be 16 bits for an embedded system. Integer types may be signed or unsigned. By default, all integer types are signed unless specifically made unsigned by the `unsigned` keyword (or the appearance of a `U` suffix on an integer constant). An exception is made for `char`, which the C standard says may be either signed or unsigned (most implementations make it signed by default). The C99 standard introduces _Bool as an integer type.

The C standard defines three levels of precision for non-integer arithmetic types: float, double and long double. Again, the choice of representation and the exact precision is implementation-specific. In most cases, IEEE-754 floating point format is used, and floats are 32 bits long, doubles are 64 bits, and long doubles may be 80 or 128 bits (long double was added in C99.) C99 complex types likewise can have float, double or long double precision, and are in fact internally handled as a struct with a real and imaginary component.

While the types of variables are explicitly declared, the compiler needs to know the types of constants. They may be specified directly by the programmer using suffixes (e.g. UL, LL), or the type can be inferred by the value and knowledge of the ranges of values representable in each type. This is discussed further in the texts.

The `void` type is a special scalar type which is used to indicate the absence of a value.

This fixed set of arithmetic types suggests that, within the compiler, a single 8-bit word with bit fields could be used to represent them, or alternatively a bit-field struct. Many years ago, when memory was scarce and compilers were very "heavy" programs, such bit-packing tricks were essential.

Enum types

enum types are equivalent to ints in C. It is never an error to assign an int to an enum or vice versa, and no range checking is required. Like structures and unions, enums are defined once, possibly with an associated tag, and can be used later with a typedef name or the word `enum` followed by the tag. Unlike structure or union members, the identifiers listed within the enum definition are not in an independent scope, but instead get placed into the symbol table currently in scope. enum constant names are in "everything else" name space and must be unique from variable, function and typedef names visible in the same scope. The only moderately tricky thing for the compiler is to keep track, during an enum definition, of the last integer value associated with the last constant name, so values can automatically be assigned if not specified.

enums are not "first class types". If they were, assigning a value of type `enum apple` to `enum orange` would be a type mismatch error! enums are simply aliases for integers, and this feature was not present in original C because almost all of the same functionality can be accomplished with the preprocessor.

Type Qualifiers

Type qualifiers were first introduced to the C language in ANSI C (C-89), with the addition of the keywords `const` and `volatile`. These qualifiers modify scalar, pointer, array, struct or union types, and must be "carried around" by the compiler. C99 added a new qualifier `restrict` which can only be applied to pointer types (and array types, but only in prototypes). Qualifiers do not change the actual representation of a type, but they do create compile-time restrictions on how a type can be used, and provide hints to the compiler for optimization.

Composite Types

More complicated types are created from the basic scalar types by applying pointer, array or function declarators.

Pointers

Within the compiler, a pointer type can be represented with an AST node indicating a pointer, and then a pointer to the underlying type. Pointers can also have a qualifier, e.g. `int * const ip;` declares a ponter which does not change, but which points to an unqualified int (contrast w/ `const int *ip;`).

Arrays

An array type is slightly more complicated. The AST node must contain a pointer to the element type of the array (the element type can not be a function, void, or a type of unknown size), and there must be an indication of the array size (number of elements). The size can be a specific constant expression known at compile-time, or it can be unknown (e.g. formal parameter to a function or referencing an extern array). C99 adds complexity by allowing for variable-length arrays and variably-modified array types. The size of the array is then tied to a potentially arbitrary expression and is not known at compile time. The compiler must insert run-time phantom statements to compute the size of the array, allocate storage for it (when declaring an array variable of variable length), and to perform proper pointer arithmetic. Variable-length arrays are not permitted for extern or static storage class. The reason for this will become apparent when the typical assembly language and linker directives for declaring such variables are covered in Unit 6. The binding or computation of a variable-length type happens at run-time at the point when the declarator is encountered, and is then fixed for the lifetime of that type instance. Therefore:

```
f()
{
        int x=5;
        int a[x];  // similar to int *a=alloca(x*sizeof (int))
        x=10;
        printf("%d\n",sizeof a/sizeof (int));
}
```

gives the answer 5. C99 allows for qualifiers within an array declarator, but only when used in a function prototype, in which case the array type is automatically converted to a pointer type anyway.

Functions

A function type needs a pointer to the return type (functions may return any type except "array of..." or "function...") and a list of pointers to the types of the arguments. The latter is the prototype for the function, and is optional to retain compatibility with classic C. Therefore, the declaration

```
int f();
```

declares that f is a function which returns int and takes an unspecified argument list, while

```
int f(void);
```

declares f specifically as taking no arguments. The compiler must also handle the presence of the variable argument list specifier `...` which may appear at the end of the argument list only.

Structures and unions

Structures and unions have two aspects. A *definition* of a new struct or union lists the members, and is similar to a list of variable declarations which might appear within a block. Each definition introduces a new, distinct type. Once defined, that type can be *referenced* to create variables of struct or union type, pointers to such, abstract type names, etc. There are two mechanisms for doing this: either saying the reserved word `struct` (or `union`) followed by the tag, or by using a typedef alias. These methods are equivalent. Struct/union definitions can appear within a declaration, abstract type name or cast, and need not have a tag. However, if the definition has no tag and is not somehow captured with a typedef, there is no way to get back at it later.

So, when the compiler encounters a struct or union definition, it must create a new symbol table, which can only contain identifiers of the identifier class *member*. I.e. other identifier uses, such as variable name, function name, label, etc. can not be installed into this mini-scope. Also note that structure, union and enum tags are not installed in this mini-scope, but rather are placed in the nearest enclosing scope. The member mini-scope will be exited at the closing brace of the struct/union definition, but the underlying symbol table will not be destroyed. The internal representation of a struct/union type includes this symbol table, and also some helpful information such as the size of the struct/union and the tag (if any) which defines it.

struct or union definitions can be incomplete, i.e. there is no member list. This is necessary for self-referential structures, e.g.:

```
struct leaf { /* etc. */ };   // Enclosing scope

{  // def of structs rooted in this scope
          struct leaf;

          struct node {
                    struct leaf *leaf;
                    struct node *node;
                    struct foo {
                              int a;
                    } bar;
          };

          struct leaf {
                    struct node *node;
                    struct leaf *leaf;
                    struct leaf myself; // NOT VALID, INCOMPLETE TYPE
          };
```

```
        struct foo f1;                          // struct foo defined above
}
```

Incomplete struct/union types are a form of forward declaration, i.e. use before definition. They can be used in any situation where it is not necessary to know `sizeof(type)`. E.g. one can always declare a pointer to an incomplete struct/union type, but it is never valid to declare an instance of the struct/union itself, whether a variable or a struct/union member, when that type is incomplete. Nor would an array of incomplete structs be valid.

The syntax shown above, with a struct tag and nothing else, e.g. `struct leaf;` is a special construction. It tells the compiler to hide any pre-existing definition of that tag (i.e. if `struct leaf` were visible outside of the outermost set of braces) and start a new, incomplete definition. This construction does not work for enum.

The definition of a struct or union is considered complete after the closing brace of its member list. Therefore, the member `myself` above is not valid, because the definition of `struct leaf` is not yet complete. The compiler must be able to determine the sizeof each member, because it must allocate offsets. Variable-size arrays can not be structure members. C99 allows for one array of unspecified ([], not variable) size to be the last member of a structure. This legitimizes a long-standing C programming trick of creating variably sized structs which have a buffer at the end. Refer to the texts for more information.

Installation of the struct/union type definition into the symbol table begins as soon as the opening brace is seen. E.g.

```
1  struct A {int a;}

2  f()
3  {
4    struct A {
5       struct A *p;
6       int b;} A1;
7    A1.p= & A1;
8    A1.p->a = 1;  //error: no such member a
9  }
```

At line 4, a new incomplete definition of struct A is begun in the scope of function f. At line 5, this incomplete definition has hidden the definition from line 1, therefore the member p refers to the definition begun on line 4.

The "binding" of a particular struct or union type to its tag takes place once. E.g.

```
f()
{
 struct A {int a;};
 struct B {struct A *ap;};
 /*...*/
 {
   struct A {double z};
```

```
   struct B b;                          /* Contains a ptr to the struct A defined
                                           in the scope of function f(), NOT
                                           the instance of struct A in this block scope */
         b.ap=0;                        /* Therefore this line works */
 }
}
```

Members of structs and unions can't have storage classes, but they can have qualifiers, because qualifiers are part of the type. Since only members can be declared within a struct/union definition, use of a typedef is not permitted within the definition, nor can a member be defined that has function type. Any nested struct, union or enum definitions are placed in the nearest enclosing (file, function, block or prototype) scope. This is true regardless of how deeply nested these definitions are. If this were not the case, they would not be visible outside of the member list, and would be fairly useless. Note in the example above the definition of `struct foo`. Another example:

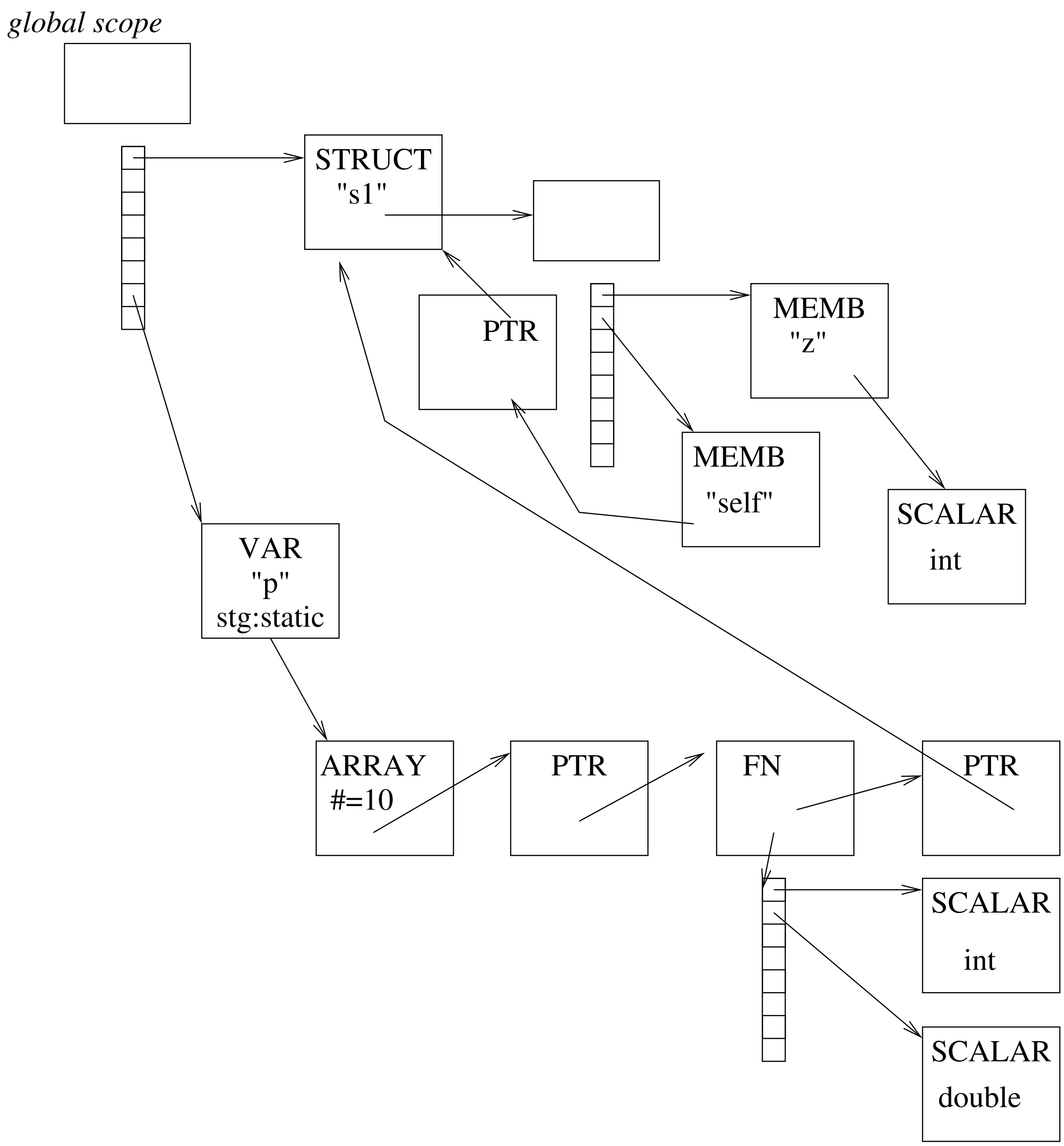
```
int f(struct foo {int a;} *p)
{
   struct foo q;            //OK
         q.a=1;             //OK
}

struct foo z;                           //ERROR

int g()
{
        struct foo {int a;} b;
        f(&b);                          //UGLY, GIVES WARNING, STILL WORKS
}
```

The definition of structure tag `foo` occurs inside of a prototype scope. Because the declarator for `f` is a function definition, this prototype scope, in effect, gets "promoted" to a function scope. Another way of looking at it is that it was a function scope all along, but in practical terms, it is not possible during parsing to determine whether a particular function declarator is going to be merely a prototyped declaration of the function's return value and parameter list, or is going to be followed by a pair of braces giving the actual function definition. Either way, the definition of `struct foo` is valid until the closing brace of the function `f`. It is not visible in file scope and thus the following line is an error. In addition, there is no valid way to call function f with a matching prototype, because the definition of `struct foo` disappears. While one could lay out another structure with the exact same member list, it is still not the identical type. For this reason, including a structure or union definition inside of a prototype is considered bad practice. These peculiarities should illustrate that a compiler must be careful about the nesting of scopes.

AST type representation vs parse tree

Below is a possible AST representation of the following code

```
static struct s1 {
         int z;
         struct s1 *self;
} * (*p[10])(int, double);
```



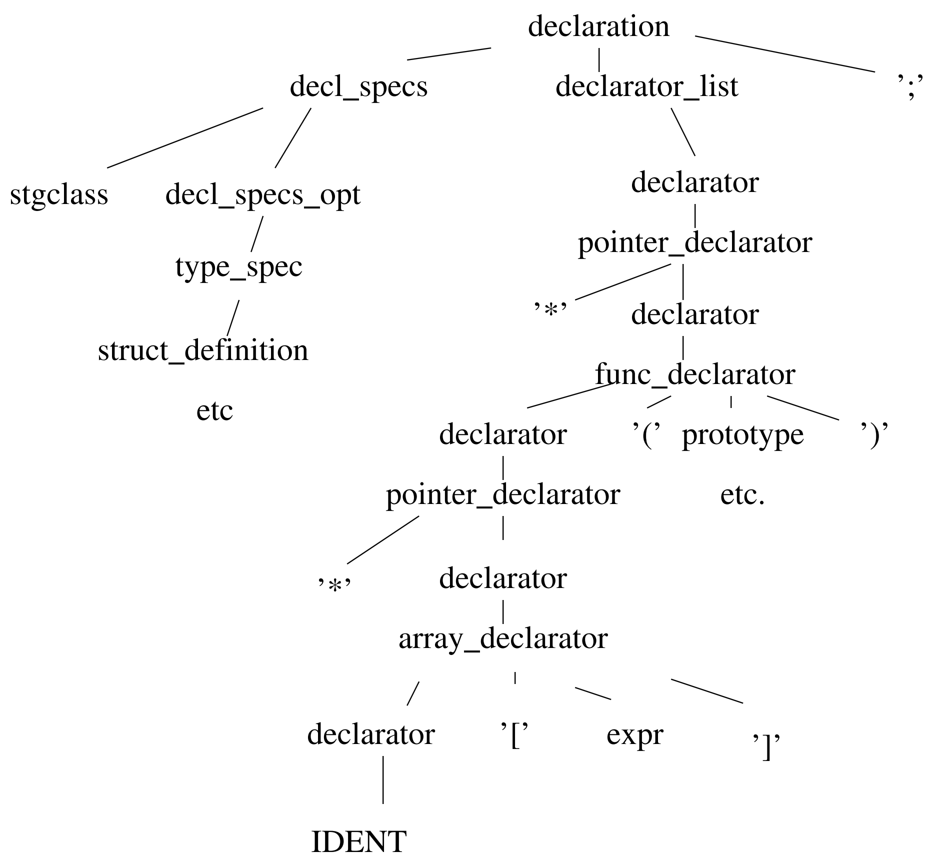
One of the challenges in constructing this representation is the nature of the C declaration syntax, which is "inside-out". A C declaration is of the form:

```
declaration:
         declaration_specifiers       declarator_list ';'
```

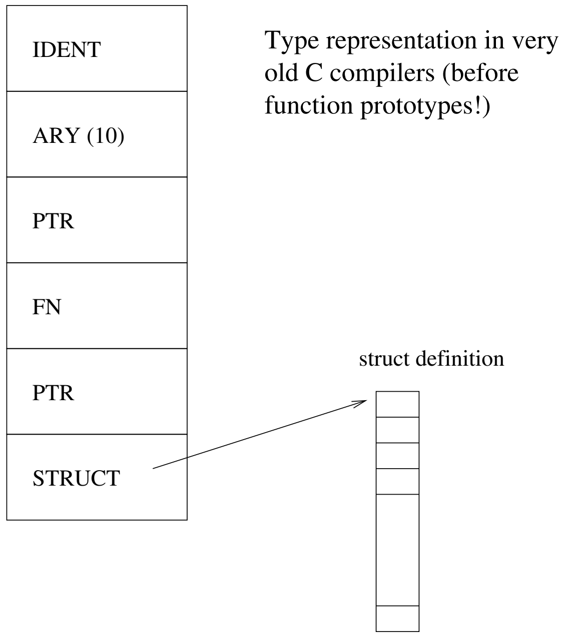
The declaration_specifiers may contain simple type specifiers, such as `int`, typedef names, or struct/union/enum references (possibly with an embedded definition). The specifiers may also be modified by qualifiers (e.g. const, volatile, restrict) or storage class.

The declarator is a limited subset of the general C expression syntax, as described previously ("Declarators and Abstract Declarators")

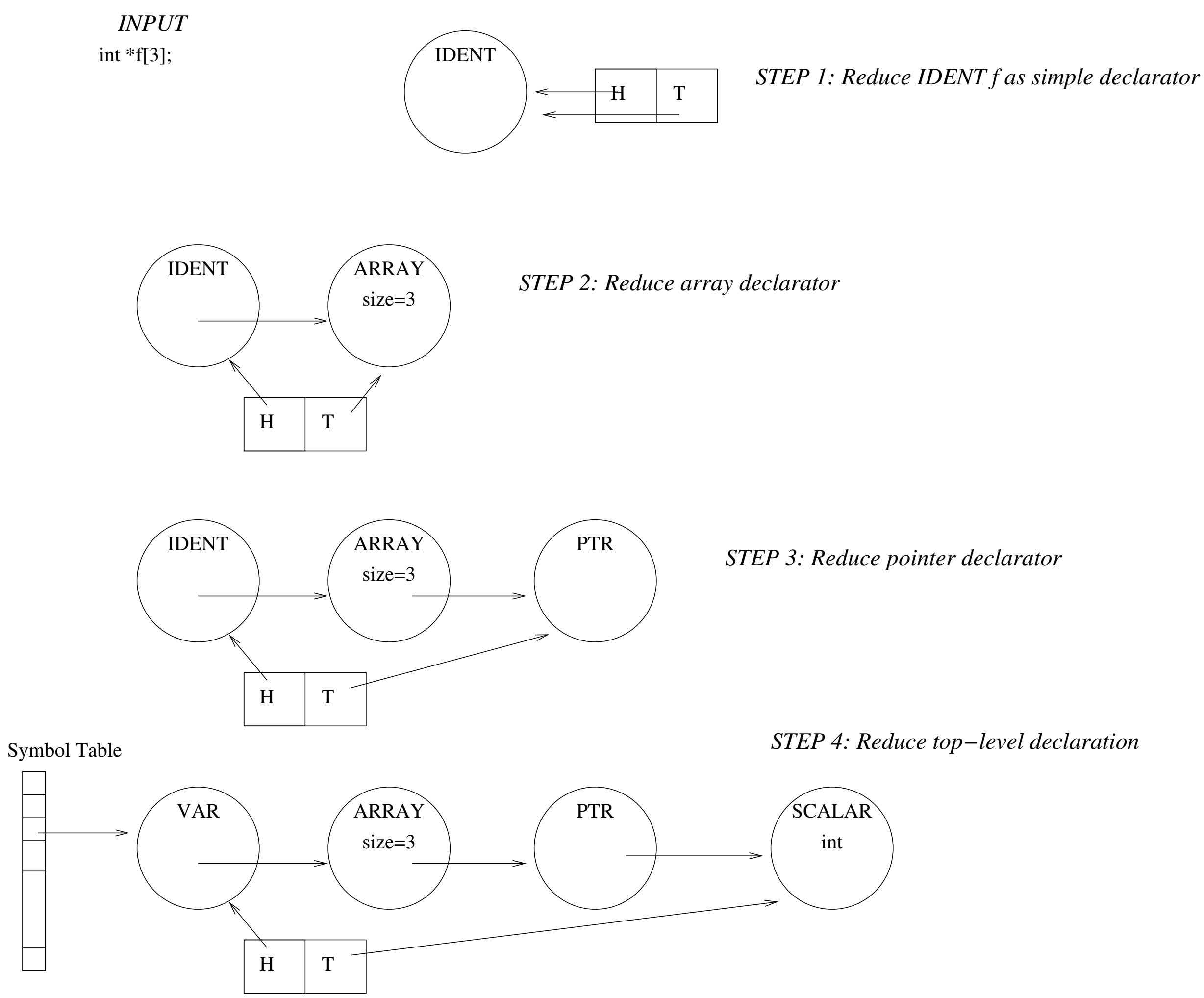
Unfortunately, the AST which would be built "naturally" by evaluating the declaration from the parse tree comes out wrong:



There are classically a number of different approaches for rectifying this when writing a C compiler. One method is to allow the AST to be built "backwards", and then make a second pass to reverse it. In the original C compilers, before function prototypes and other ugliness mucked up the language, this was a straightforward approach. Any data type could be represented as a list (in the first C compilers, it was implemented by a fixed-length array, limiting to complexity of the data type to 16 levels) and the list could be trivially reversed after parsing the declarator. In fact, the odd nature of C declarations was based on being able to re-use the same (LL-driven, hand-crafted, top-down) parser code for expressions and declarations!



Unfortunately the above approach is now obsolete. Function prototypes and variable-length arrays make types into a tree rather than a simple list. Prototypes also introduce new scopes in the middle of processing a declarator. With suitable care, we could create the type representation AST with hidden "reverse" pointers that allow us to get back to the beginning (the identifier). Another method is to use, as the semantic value of the rules involved in a declaration, a pair of AST node pointers, one of which tracks the "deepest" node, the other which tracks the "top" frontier, and use these pointers to construct the AST the correct way (this is illustrated below for a simpler declaration).



We might be tempted to just install the identifier into the symbol table as soon as we see it in the declarator, but this falls apart for two reasons. (1) We can't allow the identifier to become visible in the scope in which it is to be installed until the *end of the overall declarator* in which it is mentioned. We'll see some odd cases with typedefs below which illustrate this timining quandry. (2) Although we could tack on the base type at the end as we are about to reduce the top-level declarator (i.e. make the connection between PTR and STRUCT in the complicated example above), the storage class is also something that needs to get recognized, and its place is in the symbol table entry (the storage class is NOT part of the type!)

Typedefs

The use of the `typedef` keyword in a declaration causes any identifiers declared in that declaration to be installed into the symbol table as typedef names. The typedef name can then be used in a subsequent declaration, or abstract type name, and is completely identical to the original type. Typedefs can't be done with simple macros substitution. Consider:

```
typedef int (*apf[10])(int);

apf *papf;
#define APF int (*apf[10])(int)
APF *papf;          // FAIL!
```

A storage class keyword can not appear in conjunction with `typedef` keyword, because typedef names do not have a storage class, nor do they capture a storage class for use later.

Type qualifiers can appear in a typedef definition and they are recorded as part of the type. Additional qualifiers can be added when the typedef is used as a typedef reference in a subsequent declaration.

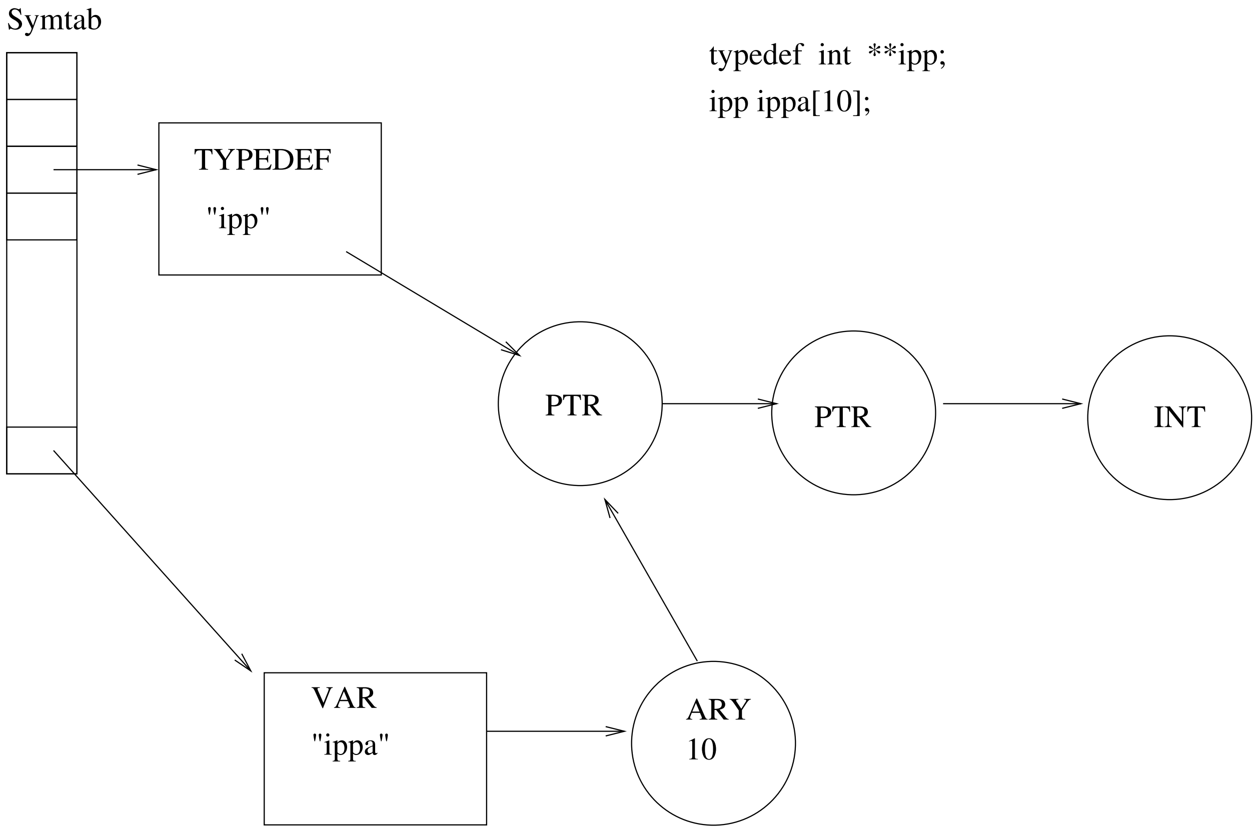
Although one can create a typedef of type function, the "function-ness" inherent in that typedef name can not be used to define a function. E.g.

```
typedef int ftd(void);

ftd *fp;  //OK

ftd f               //ERROR
{
          /*...*/
}
```

In C-99, variable-length arrays introduce a new wrinkle. The dimension of the array is calculated at the time of the typedef, and remains static thereafter. The array dimension is not recalculated when the typedef name is used to declare a variable.

```
int x=3;

main()
{
typedef int a[x];
        {
          int x=5;
          a ary;
                printf("%d\n",sizeof ary);      //prints 3*sizeof(int)
        }
}
```

In theory, typedefs are easy. When a declaration is seen with the `typedef` keyword in the declaration_specifiers, the identifier(s) is installed in the symbol table of the current scope, but as a symbol type of TYPEDEF instead of VARIABLE or FUNCTION. When that typedef identifier is later used as a type specifier in the declaration_specifiers, the AST representation of the type that was captured previously during the typedef definition then becomes part of the type of the function or variable being declared. Since the type captured in a typedef never changes, it is even permissible to share that part of the AST:



Breaking the Lexer Abstraction Barrier

Unfortunately, the typedef mechanism is going to shatter the false sense of comfort that we have gained with context-free grammars! In a theoretical world, the lexical analyzer would stand alone, feeding a sequence of tokens as demanded to the parser, based on a fixed set of lexical rules. However, consider this:

```
typedef int itd;
/*...*/
f()
{
        itd (x);
}
```

A human reading this code would hopefully recognize that `itd` is a typedef name, and therefore the line of code within the function f is a declaration of a variable `x` as type `int`. (the parentheses are redundant but are in this example to force the issue of ambiguity!) However, if the lexer simply returns IDENT as the token class for `itd`, this syntax is indistinguishable from a function call. While stylistically we would put whitespace in between the declaration specifiers and the declarator list, and omit whitespace between the function name and its arguments in an expression, this is not part of the grammar, because whitespace is ignored.

In other words, the grammar is **not context-free**, and since after the opening brace either a declaration or a statement (consisting of a function call expression) are both valid, we would have a reduce-reduce conflict here.

That the use of an identifier as a type specifier would break context-freeness of the grammar was understood at the time this feature was inserted into the C language (mid-1980s). However, there was an easy "kluge": When the lexer recognizes the pattern for an IDENTIFIER, it reaches "up" into the semantic analysis part of the compiler and asks the symbol table if that identifier is visible as a typedef name. If so, the lexer returns a different token code, e.g. TYPEDEFNAME. This resolves the ambiguity above, but we need a few more kluges to handle all of the cases. Consider this example:

```
typedef int itd;

g()
{
 itd *itd;
}
```

Upon entry to the scope attached to function g, itd is a typedef name visible from global scope. The declaration will install a new symbol itd into the inner scope (function scope) as a variable name. Because of the "timing rules" this installation doesn't take place until the end of the declarator in which the second instance of itd is mentioned, i.e. at the semicolon. However, to be a valid declarator, that second instance must be sent up by the lexer as IDENT, not TYPEDEFNAME, because TYPEDEFNAME '*' TYPEDEFNAME ';' is not a valid token sequence. With a simple secondary kluge, we can still work this out: after seeing a TYPEDEFNAME, the lexer sets a flag stopping further translation of IDENTs to TYPEDEFNAMEs. The flag is reset by the parser via an embedded mid-rule action just before the semicolon, so the lexer is ready for the next TYPEDEFNAME that potentially begins the next declaration.

In K&R C, type names can only appear in the declaration_specifiers portion of a declaration, so the approach above is sufficient. When ANSI C introduced function prototypes, things rapidly got ugly:

```
typedef int itd;

f1()
{
 itd itd(itd ij, itd itd);
}
```

This mess declares itd to be, within the scope of f1, the name of a function which returns int and takes two int arguments, which have been given the (useless) names ij and itd. To clarify this, let us add suffixes to the ambiguous identifier showing the scope in which is is being recognized and its class (T for typedefname, I for identifier).

```
//scope0==global
typedef int itdT0;
```

```
f1()
{
//scope1==function
//scope2==prototype
 itdT0 itdI1(itdT0 ijI2, itdT0 itdI2);
}
```

Note that itdI1 must be recognized as an identifier which is being declared. That declaration will be rooted in scope 1, but the C standard says that a declaration doesn't take effect until the end of the *declarator* in which that identifier is mentioned. Therefore, as we enter the prototype scope, itdT1 is not yet visible. Then itdT0 gives the type of the first prototype parameter. itdI2 declares itd to be the name of the second formal parameter, which will obscure the itdT0 declaration. Note that if we reverse the order of the prototype parameter list, we get in trouble:

```
 itdT0 itdI1(itdT0 itdI2,itdI2 ijI2);
```

because the declaration of itdI2 as the first parameter name obscures itdT0, and this takes effect when the comma is seen, as that ends the declarator in which itdI2 is mentioned. Then the second parameter declaration is invalid because itd is not recognized as a typedef name.

If we turn OFF the IDENT->TYPEDEFNAME transformation in the lexer as soon as it performs an instance of this transformation, we must turn it back ON again upon reaching the end of a complete declarator, or the opening parenthesis of the parameter type list of a function declarator. We must turn it OFF again after the closing) of the function declarator.

Even this approach is not 100% foolproof. The C standard discusses cases which are truly ambiguous and gives an arbitrary way of resolving the ambiguity. The gcc compiler, in order to comply with this, has to resort to GLR parsing rather than the lexer kluge approach. It allows IDENT to be a type_specifier. This results in a reduce-reduce conflict, and the GLR parser "splits", trying both possible parse trees. A semantic value tie-breaking rule (see the Bison documentation) is then used to determine which of the two possible parses is the correct one. Here is another example, this time using abstract type names, which can't be solved using the "lexer kluge" approach:

```
typedef int itd;

f4()
{
          itd ff(itd (itd));
}
```

Here ff is being declared as a function. The return type of ff is int. That much is perfectly clear. The argument which ff takes is ambiguous. It could be another function (you may ask: How can a function take a function? In reality, any argument of function or array type is actually passed as a pointer. But C89 and C99 allow the [] or () notation in prototypes as a syntactic convenience). So this could be equivalent to:

```
int ff(int _DUMMY(int))
```

ff takes one argument which is a (pointer to a) function returning int and taking one int.

But it could also be equivalent to:

```
int ff(int itd)
```

i.e., ff takes one integer argument which we give the (useless) name itd and surround with redundant parentheses. Section 6.7.5.3 of the C standard, item #11, states that if in a parameter declaration the identifier could be either a typedef name or a parameter name, it must be recognized as a typedef name, so the first interpretation is correct.

But wait, there's more!

```
typedef int t;

struct s {
        unsigned t:4;               //bit field named t is unsigned int
        const t:5;                  //unnamed bit field of type t
};
```

Within the structure definition above, which uses the bit field syntax, the first occurrence of the identifier t declares a structure member named t. `t` can't be a typedef name, since `unsigned` can not modify a typedef name (and `unsigned` by itself is a valid type name which is the same as `unsigned int`). But in the next line, `t` *is* a typedef name, which is *qualified* by const, and the declarator is :5, an anonymous bit field of width 5. Again, stylistically whitespace should appear between t and :, but this is not relevant. The first instance of t does not obscure the typedef declaration t (from global scope), because structure members are not in the same namespace class as typedef names.

If these typedef examples seem insane, it should be taken as a lesson in programming language "creep." Many programming languages, such as C, start out simple, elegant, and suitable for the purpose for which they were designed. As time goes by, efforts to "improve" the language result in new syntax. Often, complicated interactions among these added features, and the original structure of the language, can lead to difficult situations which were not anticipated as each new feature was added. Had the typedef mechanism originally been added to the language by creating two new keywords, let us say `typedef` to create the typedef name, and `typeref` to use it, in the same manner as struct/union tags, then the context-free nature of the grammar would not have been broken. However, this approach was rejected at the time because it was desired to have typedefs appear to be "first class" types which could be used in the manner as built-in types such as `int`. In retrospect this concern was a misplaced aesthetic. Types created by typedefs are not first-class, since array types can not be made assignable by this mechanism.

Type Algebra

We will now consider how types interact with expression operators. We can view these rules as a sort of type algebra, and we can systematically code routines to evaluate these rules, using the AST representation of types.

Type Equivalence and Compatibility

The C standard uses the term "compatible types" in a slightly confusing way. Two types are compatible if they are identical, or if they are "close enough" to be equivalent. This concept is used whenever two types interact with each other, e.g. in assignment. When two non-identical types are compatible, the result is a *composite type* which represents a compromise, of sorts, between the two types. Determination of type compatibility for complicated types can be performed recursively, using simple rules:

Arithmetic types are compatible only if they are the same type. long is not compatible with short. signed int is not compatible with unsigned int. Type qualifiers also break compatibility: const int is not compatible with int.

Each enum definition is a distinct type and is not compatible with other enum types. However, any enum type is compatible with int, and the result is that enum type. Furthermore, since enums generally get converted to ints in any expression, this is rarely an issue.

Two array types are compatible if their element types are compatible, and if their sizes are compatible too. The latter is defined as follows: if both array types define a specific size, they must be equal. However, if either or both sizes are not defined, they are compatible. The composite of the two array types has an element type which is the composite of the two element types. If either type defined a size, the composite has that defined size. In C99, the array types can be qualified and if so the qualifiers must match.

```
extern int a[];
extern int a[10];

main()
{
        printf("%d \n",sizeof a);    // 10*sizeof(int)
}
```

The re-declaration of `a` is permitted if both are `extern` AND if the declared types are "compatible." The array of unknown size is compatible with the array of specified size (and their base types are compatible, in fact identical). The composite type is the array of known size (10 elements) therefore sizeof(a) is defined and is 10*sizeof(int).

Two function types are compatible if their return types are compatible, and furthermore their argument lists must be compatible. The latter introduces some complicated rules for mixing prototype and non-prototype forms of function type specifiers. The reader is referred to the C standard. If both function types are in prototype form, the number of arguments must be equal, the use of variable arguments must be the same, and each of the arguments' types must be compatible.

Each structure or union definition creates a distinct type which is not compatible with anything else. It is the tag, or a typedef alias, which captures this. Thus:

```
struct {int a;int b;} x;
```

```
struct {int a; int b;} y;
```

the types of x and y are not compatible, even though the elements are declared the same way. Of course, when C programs are compiled from separate source files, the compiler's type checking systems are bypassed. If x and y above were in separate .c files, it is important that the code work as intended. The C standard guarantees that the compiler will always lay out structure and union types in a consistent way, so the members will always be in consistent places.

Two pointer types are compatible if their pointed-to types are compatible. The composite type is a pointer to the composite of the pointed-to types. Pointers may be qualified (e.g. a const pointer to an int, as opposed to a pointer to a const int) and if so those qualifiers must agree exactly for compatibility to hold.

Conversions

Values of a certain type can be converted to another type in several ways:
• An explicit cast expression
• Implicit conversions which happen to the arguments of many operators (unary, binary and assignment conversions)
• The conversion of array and function types to pointer types in most expressions
• Implicit conversions of actual arguments to a function call
• Implicit conversion of return value from function (as if by assignment)
• An area of memory which contains a value of one type is viewed as being of a different type, either by mistake or on purpose.

Conversions between integer types do not involve any change to the actual bit values, as long as the types are the same size. E.g. the expression

```
a=(unsigned int)b;
```

where b was defined as just int, does nothing different from simply `a=b`. When the destination type is smaller than the source, the higher-order bits are discarded. This may create a loss of information if the source value was outside of the smaller range of the destination. When the destination is wider than the source, the higher-order bits are filled with 0 if the source is unsigned, but if the source is signed, the most significant bit (the sign bit) of the source is extended to fill the higher-order bits. Most processors have an instruction to do this automatically.

Conversion of a floating point type to an integer results in truncation of the fractional portion, e.g. 3.5 becomes 3 and -1.7 becomes -1. The floating point source may exceed the range of the integer destination (e.g. 1e+20 can be represented in a 32-bit float, but not a 32-bit int) and the results are then undefined. Conversion of an integer type to floating point generally works out, but it is possible that the floating point type is not precise enough to represent the integer exactly. A 32-bit float can hold the entire range of a 64-bit long long (2**64 is approximately 16e+18), but the significand/mantissa is not large enough to hold it exactly (a float has 23 bits of precision and a double has 53. It would take a long double to hold the entire range of a long long int with no loss of precision). Similar rules hold for conversions between floating types of different size.

Pointers may be converted to integers and vice-versa. It is not guaranteed that a memory address (pointer) will fit into an int, or even a long. E.g. the model might be that ints and longs are 32 bits, but addresses are 64. However, in almost all cases, there is some integer type which can hold the pointer (long long in the previous example). Pointers and integers are not "compatible types", though, so implicit conversions between them generate a warning, but explicit casts are OK. An exception is made for the constant 0, which can always be implicitly converted to any pointer type because it is the NULL pointer.

Pointers to different types may generally be converted without any actual changing of bits. De-referencing a pointer of one type as another type may produce undefined results, e.g.

```
f()
{
int a;
float *fp;
          a=1;
          fp=(float *)&a;
          printf("%g\n",*fp);
}
```

The printf %g above is an example of the last category of conversions mentioned previously, where a given bit pattern in memory is viewed as another type. The result will be garbage. Additional complexity is introduced on architectures which impose alignment restrictions on certain operations. In the following example, the de-referencing of the pointer may result in a run-time fatal error because the value of buffer is not necessarily aligned to a 4-byte boundary.

```
g(char *buffer)
{
 long *lp;
         lp=(long *)buffer;
         printf("%d \n",*lp);
}
```

The C standard imposes an additional restriction that a pointer to a function can not be converted to a pointer to a variable or vice-versa. This is because on some architectures, function and variable pointers may have vastly different representations (e.g. a PIC microcontroller). However, a function pointer can still be converted to an integer, and that integer converted to a variable pointer, so the reasoning behind this restriction is dubious.

Casts

A type cast expression is used to force an explicit conversion from one type to another. Arithmetic types can be freely cast among each other. Pointers can be cast to and from integer types. Pointers to one type can be cast to pointers to another type, including pointers to void, except that restrictions are imposed to prevent casting between function and variable pointers (but this can be circumvented by casting through an integer type). Casting any type to `void` means discarding its value. structures, unions, arrays and functions can not be cast. Type qualifiers can also be cast away.

Function/Array Pointer Equivalence

An expression of type array of E is converted to a pointer to E in every context except:
• When the array is the argument to the & (address-of) operator
• When the array is the argument to the sizeof operator
• When a string literal (which is really an array of char) is used as an initializer.

This conversion could be stated in another way: in most contexts, the name of an array is really the address of its first element. The reason for this is that arrays are not "first class types" but are really a syntactic convenience to hide pointer arithmetic. The C implementation of arrays makes perfect sense to assembly language programmers, but is confusing to just about everyone else.

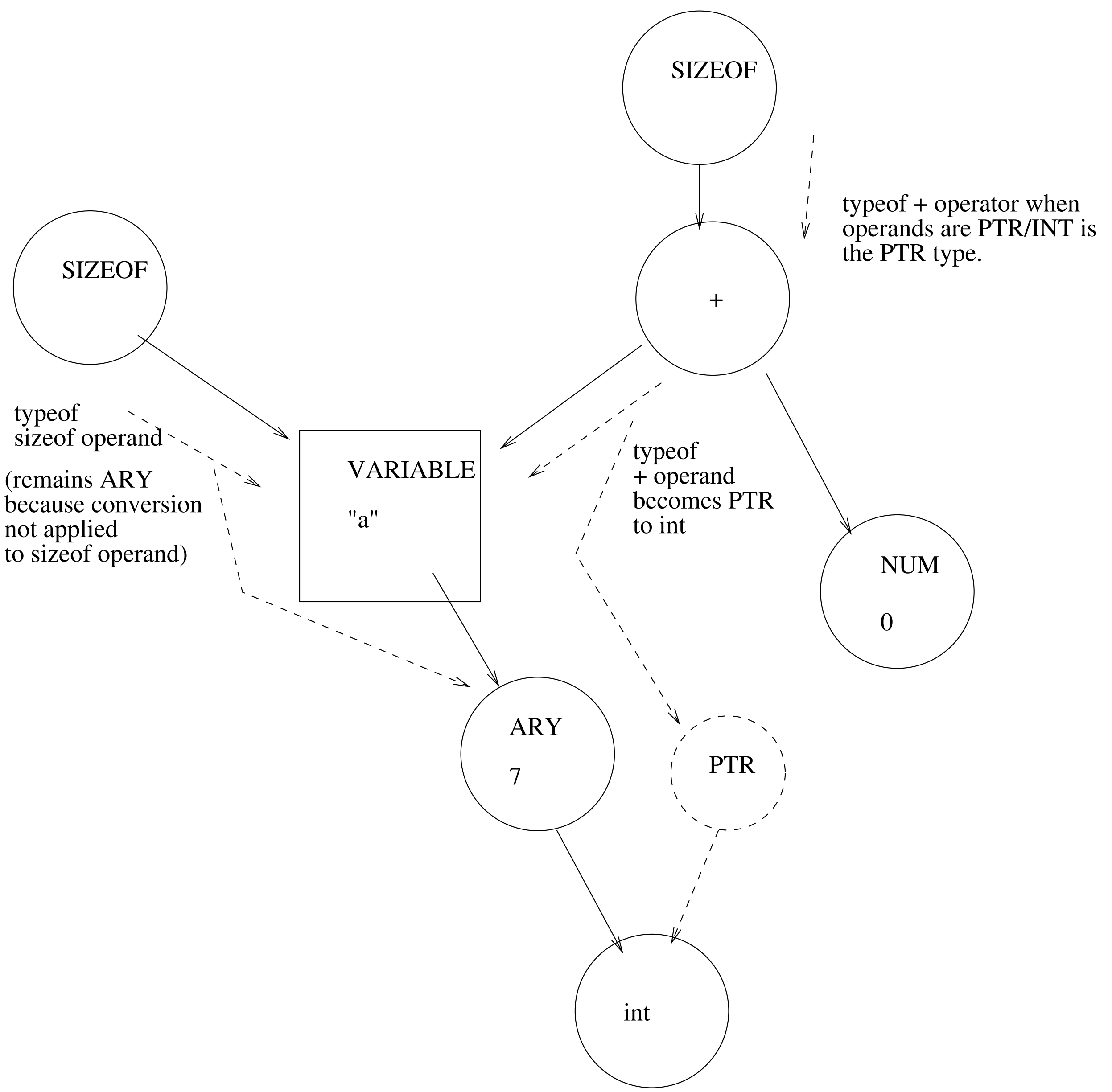
An expression of type function is converted to a pointer to function in every context except as the operand to the & or sizeof operators (furthermore it is invalid to apply sizeof to a function name). One could also say that this conversion does not take place when the function name is used to call the function, or one could say the conversion happens and that's OK because a function can be called with a pointer to a function too. Again, another way of saying all this is that a function name is really the address of its first machine-language instruction.

The "shielding" effect of sizeof and & extends only to the expression which is the immediate operand of the operator. It does not recurse to deeper levels of the expression or the type.

Let us examine some cases:
```
main()
{
int a[7];
          printf("%d %d\n",sizeof a,sizeof(a+0));
}
```

The variable a has type array(7,int). In the first case, a is the direct operand of sizeof, and thus its type is not converted to a pointer. sizeof(int [7]) is 7*sizeof(int), or 28 on a 32-bit architecture. In the second case, a appears as the operand of the + operator. Its type is therefore converted to pointer(int). As we'll see later in this unit, when the addition operator has operands of types pointer and int, the result is the pointer type. Therefore the second printf output is sizeof(int *), or 4 on a 32-bit architecture.

We can visualize the way a compiler could perform this "type algebra" using an AST. The solid portions represent the AST as constructed from the two parsed sizeof expressions. When it comes time to evaluate the *value* of `sizeof`, the compiler must determine the actual type of its operand. It can do so by recursively exploring the AST, applying various rules.



The type of the expression `&a` would be pointer(array(7,int)), not pointer(pointer(int)), because a, as the operand to the address-of operator, does not get converted. This is important for multi-dimensional array pointer arithmetic:

```
int b[8][10];
int c[4][5][6];
main()
```

```
{
        printf("%p %p %p %p\n",b,&b+1,b+1,*b+1);
        printf("%p %p\n",c,*c+1);
}
```

The type of variable b is array(8,array(10,int)). Consider the four uses of b in the printf. In the first case, b is subject to conversion, and the outer array type becomes a pointer: pointer(array(10,int)). To analyze the next use, recall (or read ahead to learn) that the expression p+i, with operands i and p of type integer and pointer(T), respectively, computes p+i*sizeof(T). In the expression &b, the variable b is not subject to type conversion as the operand of the address-of operator. Therefore it has a type of pointer(array(8,array(10,int)), and the value computed is 8*10*sizeof(int)*1 greater than the base address of b. In other words, it correctly computes the address of the "next" int [8][10] array beyond b.

In the third case, b's type does get converted to pointer(array(10,int)), and the expression value is 10*sizeof(int) beyond the base address, i.e. the address of the second row of the array b.

In the fourth case, b's type is converted to pointer(array(10,int)), and is the operand of the dereference (*) operator, which takes operands of type pointer(T) and results in a type T. Therefore the subexpression *b has type array(10,int). BUT, this subexpression which is an operand to + is also subject to array-pointer conversion, and therefore the operands to + are of type int and pointer(int), and the address is incremented by just one element.

In the second printf, c is of type array(4,array(5,array(6,int))). c is converted to pointer(array(5,array(6,int)). After the * operator the type is array(5,array(6,int)). This is now subject to conversion again, but note that only the outermost array type is affected, and the type becomes pointer(array(6,int)). The address is incremented by 6*sizeof(int).

Array/pointer conversion does not take place when a string literal is used to initialize an array variable

```
char a[]="hello";
char *p="goodbye";
```

The array dimension of a is unspecified, but because of the initializer, the type of a is (char [6]), allowing an extra byte for the NUL terminator. In the second example, the string does not initialize an array variable, and is promoted to a pointer. In K&R C, an initialized array declaration was allowed only when the variable has extern or static storage class, but C-89 and C-99 allow the initialization in all cases. This leads to some unfortunate inconsistency:

```
f()
{
 char a[]="allowed";
 char b[8];
          b="denied";
}
```

Both a and b are variables of `auto` storage class. The initialization of a requires the compiler to generate code similar to:

```
char a[8];strcpy(a,"allowed");
```

But if we miss the opportunity to initialize the array during declaration, that opportunity is lost forever! The string "denied" is promoted to (char *), and the variable b is also promoted, but since array names are never lvalues, the assignment is an error. In other words, while initialized declarations seem to be similar to a declaration followed by an assignment, they are not totally equivalent! This will become even more evident with respect to global variables when we explore target code generation and the linker model.

Assignment Conversions

When the types of the left and right side of an assignment (including the implicit assignment in a return statement) are not identical, a conversion will take place. This conversion is the same as if an explicit cast were made, except certain combinations are not allowed. Any arithmetic type can be assigned to any other arithmetic type, unless the destination type is qualified with `const`. structs and unions can only be assigned if both sides are of identical type. Any pointer to a variable can be assigned to a void * and vice-versa. Any pointer can be assigned to any other pointer if they point to compatible types. The integer constant 0 (NULL pointer) can be assigned to any pointer.

If the above criteria are not met, a warning is generally given, and then the assignment is attempted as if an explicit cast were used. If such a cast is not permissible, the assignment is in error (e.g. attempt to assign a struct into an int).

Unary and Binary Arithmetic promotions

In most cases, values of integer type smaller than int (short, char) are promoted to int. This reduces the complexity of the code which the compiler must generate for expressions. Generally, int is the natural register size of an architecture, so it isn't any faster or cheaper to do 8-bit integer operations if the machine is naturally 32 bit.

One thing which ANSI C "broke" involves promotions of unsigned short and unsigned char. In K&R C, these were promoted to unsigned int, but in Standard C, they are promoted to int, as long as int is strictly wider than the original type and can thus represent the entire range. The problem with this is that the expression then loses its unsignedness, which can sometimes be a problem.

K&R C promoted floats to doubles, but ANSI C doesn't require this. Unlike integer operations, it is quite likely that single-precision floating point operators are indeed faster than double, so if the programmer isn't asking for the additional precision, why do it?

When an expression operator is presented with two subexpressions of differing types, one of them must be promoted before the operation makes sense. So, after performing the unary promotions described above, if the types are still different, the "lesser" type is promoted. E.g. when adding a double to an int, the int is promoted to double. Adding an int to a long long results in promotion to long long.

The rules of binary promotions are somewhat complicated. (unsigned int) OP (signed int) results in (unsigned int). But (unsigned int) OP (signed long) results in (signed long) because the latter is a "bigger" or "higher ranking" integer type. This behavior also changed from K&R to ISO C or C99. Many C programmers do not fully understand the promotion rules. If uncertain, it is best to use explicit casts.

In classic C without prototypes, the actual types of the arguments are not known when the compiler sees the function call. To simplify argument marshalling, classic C promotes floats to doubles, and promotes char and short as described above. In Standard C, with a prototype in effect, these promotions are not required to be performed. This is a good optimization for floats, as discussed above, but not so important for integer types. In fact, most implementations will promote char and short arguments anyway. If the function has a prototype that specifies variadic arguments, those arguments must always be promoted because their type is not known.

Expressions

The C language has an unusually rich expression syntax. There are 16 levels of operator precedence defined. Below is an outline of the expression syntax and some of the semantic analysis issues. We will look at the type algebra of each operator, as well as some issues in evaluating rvalues and lvalues. We'll have to come back to this in the next unit as we discuss code generation.

An expression is said to produce *side-effects* if it can modify the value of an object, i.e. includes pre or post increment or decrement operators, assignment operators or function calls. If an expression statement produces no side-effects, such as `1+2;` then a warning may be generated unless the expression value is explicitly cast to `(void)`.

Some expressions have equivalent forms, e.g. a+=b is identical to a=a+b; The compiler writer may choose, during AST construction, to transform some expressions into their equivalent form, so as to simplify the number of cases needed in later analysis.

Primary Expressions (Level 16)

• `IDENT`: if the name of a variable, this is an lvalue, unless the variable is of array type, in which case the expression is converted to a pointer to the element type and is not an lvalue. This conversion doesn't happen if this primary expression is the operand of a sizeof or & operator. Technically, a variable of array type is an lvalue (it has storage associated with it and we can take its address), but because of this default conversion to a pointer type, this distinction really doesn't matter.

The name of a function is also converted to a pointer to a function except when applied to sizeof or &. Again, this distinction is a hair-splitter, since sizeof a function is not valid, and there is no reason to use & on a function name.

If the identifier is the name of an enum constant, then this expression has the value of that constant, and an integer type of the compiler's choosing (generally just int). enum constants can't be lvalues.

Other uses of an identifier (tags, members, labels, typedef names) are not a primary expression.

• literal: Integer, floating-point, character and string literals are all primary expressions. Their type is inferred from their lexical pattern, including possible modifiers such as `UL`, and (in the case of integers) by their value in relation to the range of values representable by each scalar type. string literals are of type array of char, but they are converted to pointer to char in all cases except as the operand to a sizeof, operand to &, or when being used to initialize an array of char. Technically, a string literal is an lvalue, but because of the default conversion to a pointer type, do we really care? It is implementation-specific whether a string literal can be modified.

```
char *s="ABC";s[0]='X';                // Produces a SIGSEGV on most UNIX implementations
```

In the example above, the compiler does not flag an error, but at run-time a fatal exception is raised (SIGSEGV), at least on most UNIX implementations, because they place string literals into the read-only program text.

• `( expr )` : Parentheses do nothing, the value and type is that of the enclosed expression. They are used for style or to escape the normal precedence and associativity rules. Typically in building an AST for an expression, one would not even bother with parenthesized expressions.

Postfix expressions (also Level 16)

• `e1[e2]`: An array subscripting expression is defined as exactly equivalent to `*(e1+e2)`. Since addition is commutative, this imples that `e2[e1]` is also equivalent, although that would be stylistically strange. e1 must be a pointer type, and e2 must be an integer type, or vice-versa. As a result of the implicit binary + operator, the usual binary promotions and conversions are applied to e1 and e2. See discussion of the + operator and pointer arithmetic. If the pointer type is "pointer to T", the result of the subscript expression is type "T". The result is always an lvalue.

• `e1.IDENT`: e1 must be of struct or union type and the definition of that type must be complete. IDENT must be a member within that type. The result is of the type of the member. If either the member or e1 have qualifiers, the result type is the inclusive-or of those qualifiers. The result is an lvalue iff e1 is an lvalue (struct and unions returned by a function are not lvalues).

• `e1->IDENT`: This is exactly equivalent to `(*e1).IDENT`.

• Function Call: The type of the expression appearing to the left of the parentheses which surround the (possibly empty) actual argument list must be "pointer to function...". Since an expression of function type gets converted to a pointer to a function, this works as intended. It also means that given a function pointer, it is not necessary to use * explicitly to call the function, although it is usually good style to do so. The arguments are converted as if by assignment to the types specified in the prototype, but arguments in the variable portion of the prototype are subject to the usual promotions discussed above, as are any arguments of a function called without a prototype visible. Calling a function using an undefined symbol F results in implicit declaration of that symbol as int F(), as discussed in an earlier section. The result type is the return type of the function, and is never an lvalue.

• `e++`: e must be a modifiable lvalue of integer, enum, real or pointer type. The effect of this expression is similar to `(temp=e,e=e+1,temp)`. e is subjected to the binary conversions because of the increment, and the assignment conversions when storing the result back in e. The result type is the type of e (before any conversions) and is never an lvalue. E.g.

```
char c; printf("%d",sizeof(c++));
```

prints 1, because the result type is char, not int, even though the increment occurs in the int type.

`e—`: see above

• Compound literals: C99 introduces a way to create a literal of any aggregate type including struct, union or array. This is actually somewhat handy. The syntax is documented in the C standard and Harbison & Steele. The result is an lvalue and is modifiable unless a const qualifier is used. Compound literals are approximately equivalent to having declared a variable of the same type with an initializer and then forgetting the name of that variable after it is used in an expression.

Unary Expressions (Level 15)

• `sizeof ( type_name )` and `sizeof expr` : The sizeof operator returns the size, in bytes, of the given expression or of the given type name. The return type is `size_t`, which is a typedef defined in `stddef.h`. This type is an integer large enough to represent the largest possible object. On 64-bit architectures, `size_t` is going to be a `long long`, At one time, the thought of a single array exceeding 2GB was somewhat scary, but today might be seen in large applications. `size_t` is not a new scalar type, and its correspondence to an actual type such as `int` is something which is fixed into the compiler when the compiler is built for a specific target architecture. The result of sizeof is never an lvalue. As with casts, the type_name could include struct, union or enum definitions which could be referenced later as long as a tag was given, but this is considered poor style. If `expr` is given, it must be a valid expression, but it is NOT evaluated (but see below about variable-length arrays). The usual conversions do not apply to the top-level operand, but they do to any inner expressions:

```
char c;
int i1,i2;
```

```
        i1=sizeof(c);       // parens redundant
        i2=sizeof(c+1); // parens needed for order of ops
```

i1 has the value 1, but i2 has the value 4, because c is promoted. Note that parentheses are not required when the `expr` form is used. However, many programmers have difficulty remembering all 16 precedence levels and use them for clarity. It is an error to apply sizeof to an expression or type name with an incomplete type. Before C99, the value of sizeof was truly constant and known at compile-time. When C99 variably-modified abstract array types are used with sizeof, the length control expression will be evaluated:

```
main()
{
int i,j;

        i=1;
        j=sizeof(char [i++]);                   //side-effect in control expression!
        printf("i=%d j=%d\n",i,j);     //i is 2, j is 1
}
```

However, this example:

```
main()
{
        int i;
        i=1;
        char a[i++];
        printf("%d %d\n",sizeof(a),i);
        printf("%d %d\n",sizeof(a),i);
        printf("%d %d\n",sizeof(a),i);
}
```

gives inconsistent results depending on compiler version. It should print `1 2` each time, because the size control expression should only be evaluated once, when the variable is declared. Aren't variable-length arrays a wonderful feature?!

`sizeof(void)` is defined as 1, because a `(void *)` is the basic "pointer to memory" (whereas in classic C it was a `(char *)`) and pointer arithmetic has to work. But the `void` data type can't be the base type of an array, a struct/union member, or a variable/parameter.

• `+e`

`−e`: Unary plus and unary negation require an operand of any arithmetic type. The result has the same type as the operand (after unary promotions) and is never an lvalue. The usual unary promotions are applied to the operand.

• `!e`: Logical NOT performs the usual unary promotions and is identical to (e)==0. and therefore the result is of type int, and the operand may be any arithmetic or pointer type. The result is never an lvalue.

• `~e`: Bitwise NOT requires an operand of integral type. The usual unary promotions are performed. The result is of the (possibly promoted) type of the operand and is never an lvalue.

• `&e`: The operand must either be an lvalue or a function or array type. If the operand is of type T, the result is of type pointer to T. The usual conversions and promotions are not applied to the top-level operand. Note that if variable a is of type array of int, taking &a yields type "pointer to array of int". This does make a difference when it comes to pointer arithmetic. It is invalid to apply the address-of operator to a variable of register storage class, although many compilers will allow it with a warning. Applying & to a function is somewhat superfluous since function types are normally converted to pointers anyway.

• `*e`: The operand must be of type pointer to T, and the result is of type T. The result is an lvalue unless T is a function or array type. The usual unary conversions are applied to the operand. If a is an array type, `*a` is equivalent to `a[0]`.

• `—e`

`++e`: These pre-increment and pre-decrement operators require a modifiable lvalue operand and their result is never an lvalue. They are equivalent to `e–=1` and `e+=1`.

Casts (Level 14)

`(type_name) e`: An explicit cast conversion, as previously described. `type_name` is an *abstract declaration*, which has the same syntax as a regular declaration but omits the identifier which would normally be declared. Abstract declarations may contain qualifiers such as const, but they can not include storage class specifiers, or the `typedef` keyword. The result is never an lvalue and has the type `type_name`. casts are listed at precedence 14, while all other Unary Expressions are at 15. This resolves a potential parsing ambiguity:

```
sizeof(int)*p
          is NOT sizeof ( (int)*p))
          it IS  (sizeof(int))*p
```

Multiplicative expressions (Level 13)

• `e1 * e2`

• `e1 / e2`

• `e1 % e2`: Each operand must be of arithmetic type. The unary promotion conversions are performed on e1 and e2 to widen integers small than int, then the binary conversions are performed to promote e1 or e2 to the greater common type if they are not already the same type. The result is of this type and is never an lvalue.

Additive expressions (Level 12)

• `e1 + e2`

• `e1 - e2`: The same unary and binary conversions are applied as discused for multiplicative operators immediately above. The operands may both be arithmetic types, or one operand may be of pointer type (it can not be a pointer to a function, a pointer to void, or a pointer to anything else whose size is not known) and the other of integer type.

In the latter case, *pointer arithmetic* is performed. Let us say e1 is the pointer type and e2 is the integer type. For addition, the actual value of the pointer, which is a memory address, is incremented by `e2*sizeof(*e1)`. For subtraction, it is decremented. The resulting type is the pointer type.

For subtraction only, BOTH operands may be pointers to the same (or compatible) types. Then the resulting memory address is `(e1-e2)/sizeof(*e1)`. This is not exactly how the standard defines it, but it is how the compiler will probably implement it. If either pointer is not really pointing at an object of the appropriate type, the results are not defined. The result type of subtracting two pointers is of type `ptrdiff_t`, which is going to be an integer type large enough to hold any memory address. As with `size_t`, `ptrdiff_t` is an architecture-specific typedef (to some kind of int type) The result of an additive expression is never an lvalue.

Shift expressions (Level 11)

- `e1 << e2`
- `e1 >> e2`: Each operand must have integer type and the unary promotions are applied to each separately. There is no need to perform binary conversions to promote the operands to a common type. The result type is never an lvalue and is of the (possibly converted) type of the left operand. Results are undefined if e2 is negative.

Relational expressions (Level 10)

- `e1 < e2`
- `e1 > e2`
- `e1 <= e2`
- `e1 >= e2`: The operands must be integer or real types, or pointers to the same or compatible types. The binary conversions are applied. The result is an int which is either 0 or 1 and is never an lvalue. For comparisons between integers, signedness matters, as we shall see when we study the assembly language which must be emitted. Some compilers will catch an error such as `if (ui<0)` where `ui` is an unsigned int. The C standard explains rules for comparisons between pointers which are somewhat theoretical. In most architectures, the pointers are compared as if they were unsigned integers of the appropriate width.

Equality expressions (Level 9)

- `e1 == e2`
- `e1 != e2`: The usual binary conversions are performed on the operands, which may be of any arithmetic type, or pointers to compatible types. Pointers of any type may always be compared against a void *, or against the constant 0. structs and unions can not be compared, although they can be assigned. (The reason is that because of padding issues

(to be covered in a later unit) two structures or unions may have identical member values, but their memory images may differ in the padding regions.) The result of an equality expression is an int which is either 0 or 1 and is never an lvalue.

Bitwise expressions (Levels 8, 7, 6)

• `e1 & e2`
• `e1 ^ e2`
• `e1 | e2`: These are listed in decreasing order of precedence. The usual binary conversions are applied to the operands, which must be of integer type. The result is not an lvalue. In the author's opinion, the bitwise operators should have been placed above the relational and equality operators. Consider: `if (i&0x0F==3)`, this compares 0x0F to 3 and then bitwise AND's the resulting 1 or 0 with the integer i. This is almost never what is desired! Unfortunately it would be a little late and cause a lot of confusion to change it now.

Logical expressions (Levels 5, 4)

• `e1 && e2`
• `e1 || e2`: The && operator has higher precedence. These operators are sometimes called "short-circuit" operators because it is possible that e2 will not be evaluated if the result can be determined solely from e1 (i.e. e1 is zero for && or non-zero for ||). This is put to good use with C idioms such as

```
char *p;
/*...*/
if (p && p[0]!='A')
```

The operands to && and || are each subject to the usual unary promotions and can be of any arithmetic or pointer type. They are compared to 0 as if by the == or != operator. Note the implicit comparison between a pointer and NULL above. The result of the && or || operators is an int and is never an lvalue.

Ternary expression (Level 3)

• `e1?e2:e3`: No other language before C had anything like the ternary, or conditional operator! e1 must be arithmetic or pointer type, and is compared to 0 as if by the == operator. If e1 is non-zero, then e2 is evaluated and that becomes the result of the expression, otherwise e3 is evaluated. Only one of these expressions is evaluated. If both contain side-effects, only the side-effects of the selected expression happen. e2 and e3 can be of any type and are subject to the usual unary conversions. If they are not of the same type, the rules are somewhat complex - basically they must be compatible as if by assignment, so the expression `a=b?c:d` has to work. Refer to the C standard or *Harbison & Steele, pg. 245*. The result of the ternary expression is not an lvalue, although C++ allows `(c?a:b)=v`, as do some GCC C compilers.

Assignment expressions (Level 2)

- `e1=e2`
- `e1 += e2`
- `e1 -= e2`
- `e1 *= e2`
- `e1 /= e2`
- `e1 %= e2`
- `e1 ^= e2`
- `e1 |= e2`
- `e1 &= e2`
- `e1 <<= e2`
- `e1 >>= e2` : All of the assignment operators are at the same precedence level and are right-associative. The compound assignment operators are equivalent to a simple assignment combined with the binary operator, e.g. a+=b is equivalent to a=a+b. The types of e1 and e2 must be compatible, after applying the usual promotions and the assignment conversions. e1 must be a modifiable lvalue and can not be qualified by `const`. It is not possible to "assign away" qualifiers under a pointer type. E.g. it is not valid to assign a (const int *) to a variable of type (int *), because doing so would allow the underlying (const int) to be modified. Of course, the programmer can circumvent this with an explicit cast.

The resulting type of an assignment is the original type of e1. The result is never an lvalue. In very early versions of C, structs and unions could not be assigned. This is no longer the case, but the assignment operator still can not be used to copy a whole array. This is because if e1 is of array type, the usual conversions are applied, making it a pointer type which is not an lvalue.

As an interesting aside, in very early versions of C, compound assignment operators were reversed. E.g. `a=+3;`. This was borrowed from the predecessor language B and cut down on the complexity of the compiler, which was being hand-coded. Of course, `a=*b` now becomes ambiguous because it might also be `a= *b`. The former is a compound assignment operator and the latter is a simple assignment and a pointer dereference. This form of compound assignment was deprecated in the 1980s.

Comma Expressions (Level 1)

- `e1,e2`: The comma, or sequential operator is also an invention of C. e1 is fully evaluated. Any value which it may produce is discarded, (but side-effects still happen) then e2 is evaluated and the result is the value of the overall expression. There is no requirement that e1 and e2 be of compatible types. The result type is the type of e2 and is not an lvalue. Harbison & Steele says that unary conversions are applied, but the current ISO C standard does not, and gcc seems to follow the latter, e.g. `sizeof (i,c)` where i is an int and c is a char, is 1.

Note that syntactically, a comma expression is at the top level of the definition of an expression. There are places in the grammar where an expression is permitted but this could cause confusion with other uses of the comma token, and therefore in those places the grammar specifies an assignment_expression (i.e. starting at Level 2). These places are function calls, enumeration value declarations, initializers, bit field length specifiers in struct/union definitions.