Chapter 8

# PROGRAM DEVELOPMENT

## in the Unix Programming Environment

The UNIX system was originally meant as a program development environment. In this chapter we'll talk about some of the tools that are particularly suited for developing programs. Our vehicle is a substantial program, an interpreter for a programming language comparable in power to BASIC. We chose to implement a language because it's representative of problems encountered in large programs. Furthermore, many programs can profitably be viewed as languages that convert a systematic input into a sequence of actions and outputs, so we want to illustrate the language development tools.

In this chapter, we will cover specific lessons about

- `yacc`, a parser generator, a program that generates a parser from a grammatical description of a language;
- `make`, a program for specifying and controlling the process by which a complicated program is compiled;
- `lex`, a program analogous to `yacc`, for making lexical analyzers.

We also want to convey some notions of how to go about such a project—the importance of starting with something small and letting it grow; language evolution; and the use of tools.

We will describe the implementation of the language in six stages, each of which would be useful even if the development went no further. These stages closely parallel the way that we actually wrote the program.

(1) A four-function calculator, providing `+ - * /` and parentheses, that operates on floating point numbers. One expression is typed on each line; its value is printed immediately.

(2) Variables with names `a` through `z`. This version also has unary minus and some defenses against errors.

(3) Arbitrarily-long variable names, built-in functions for `sin`, `exp`, etc., useful constants like $\pi$ (spelled `PI` because of typographic limitations), and an exponentiation operator.

(4) A change in internals: code is generated for each statement and subsequently interpreted, rather than being evaluated on the fly. No new features are added, but it leads to (5).

(5) Control flow: `if-else` and `while`, statement grouping with `{` and `}`, and relational operators like `>`, `<=`, etc.

(6) Recursive functions and procedures, with arguments. We also added statements for input and for output of strings as well as numbers.

The resulting language is described in Chapter 9, where it serves as the main example in our presentation of the UNIX document preparation software. Appendix 2 is the reference manual.

This is a very long chapter, because there's a lot of detail involved in getting a non-trivial program written correctly, let alone presented. We are assuming that you understand C, and that you have a copy of the *Unix Programmer's Manual*, Volume 2, close at hand, since we simply don't have space to explain every nuance. Hang in, and be prepared to read the chapter a couple of times. We have also included all of the code for the final version in Appendix 3, so you can see more easily how the pieces fit together.

By the way, we wasted a lot of time debating names for this language but never came up with anything satisfactory. We settled on `hoc`, which stands for "high-order calculator." The versions are thus `hoc1`, `hoc2`, etc.

## Stage 1: A four-function calculator

This section describes the implementation of `hoc1`, a program that provides about the same capabilities as a minimal pocket calculator. It has only four functions: `+`, `-`, `*`, and `/`, but it does have parentheses that can be nested arbirarily deeply, which few pocket calculators provide. If you type an expression followed by RETURN, the anwer will be printed on the next line:

```
$ hoc1
4*3*2
        24
(1+2) * (3+4)
        21
1/2
        0.5
355/113
        3.1415929
-3-4
hoc1: syntax error near line 4    It doesn't have unary minus yet
$
```

**Grammars.** Ever since Backus-Naur Form was developed for Algol, languages have been described by formal grammars. The grammar for `hoc1` is small and simple in its abstract representation:

```
list:  expr \n
       list expr \n
expr:  NUMBER
       expr + expr
       expr - expr
       expr * expr
       expr / expr
       ( expr )
```

In other words, a `list` is a sequence of expressions, each followed by a newline. An expression is a number, or a pair of expressions joined by an operator, or a pharenthesized expression.

This is not complete. Among other things, it does not specify the normal precedence and associativity of the operators, nor does it attach a meaning to any construct. And although `list` is defined in terms of `expr`, and `expr` is defined in terms of `NUMBER`, `NUMBER` itself is nowhere defined. These details have to be filled in to go from a sketch of the language to a working program.

**Overview of yacc.** `yacc` is a *parser generator*,† that is, a program for converting a grammatical specification of a language like the one above into a parser that will parse statements in the language. `yacc` provides a way to associate meanings with the components of the grammar in such a way that as the parsing takes place, the meaning can be "evaluated" as well. The stages in using `yacc` are the following.

First, a grammar is written, like the one above, but more precise. This specifies the syntax of the language. `yacc` can be used at this stage to warn of errors and ambiguities in the grammar.

Second, each rule or *production* of the grammar can be augmented with an *action*—a statement of what to do when an instance of that grammatical form is found in a program being parsed. The "what to do" part is written in C, with conventions for connecting the grammar to the C code. This defines the semantics of the language.

Third, a *lexical scanner* is needed, which will read the input being parsed and break it up into meaningful chunks for the parser. A `NUMBER` is an example of a lexical chunk that is several characters long; single-character operators like `+` and `*` are also chunks. A lexical chunk is called a *token*.

Finally, a controlling routine is needed, to call the parser that `yacc` built.

`yacc` proceses the grammar and the semantic actions into a parsing function, named `yyparse`, and writes it out as a file of C code. If `yacc` finds no errors, the parser, the lexical analyzer, and the control routines can be compiled, perhaps linked with other C routines, and executed. The operation of this program is to call repeatedly upon the lexical analyzer for tokens, recognize the grammatical (syntactic) structure in the input, and perform the semantic actions as each grammatical rule is recognized. The entry to the lexical analyzer must be named

† `yacc` stands for "yet another compiler-compiler," a comment by its creator, Steve Johnson, on the number of such programs extant at the time it was being developed (around 1972). `yacc` is one of a handful that have flourished.

yylex, since that is the function that yyparse calls each time it wants another token. (All names used by yacc start with y.)

To be somewhat more precise, the input to yacc takes this form:

```
%{
C statements like #include, declarations, etc. This section is optional
%}
yacc declarations: lexical tokens, grammar variables,
   precedence and associativity information
%%
grammar rules and actions
%%
more C statements (optional):
main() { ...; yyparse(); ... }
yylex() { ... }
...
```

This is processed by yacc and the result is written into a file called y.tab.c, whose layout is like this:

```
C statements from between %{ and %}, if any
C statements from after second %%, if any:
main() { ...; yyparse(); ...  }
yylex() { ...  }
...
yyparse() { parser, which calls yylex() }
```

It is typical of the UNIX approach that yacc produces C instead of a compiled object (.o) file. This is the most flexible arrangement—the generated code is portable and amenable to other processing whenever someone has a good idea.

yacc itself is a powerful tool. It takes some effort to learn, but the effort is repaid many times over. yacc-generated parsers are small, efficient, and correct (though the semantic actions are your own responsibility); many nasty parsing problems are taken care of automatically. Language-recognizing programs are easy to build, and (probably more important) can be modified repeatedly as the language definition evolves.

**Stage 1 program.** The source code for hoc1 consists of a grammar with actions, a lexical routine yylex, and a main, all in one file hoc.y. (yacc filenames traditionally end in .y, but this convention is not enforced by yacc itself, unlike cc with .c files) The grammar part is the first half of hoc.y:

```
%{
#include <stdio.h>        includes needed for code later on
#include <ctype.h>
#define YYSTYPE double    /* data type of yacc stack */
%}
```

```
%token  NUMBER
%left   '+' '-'   /* left associative, same precedence */
%left   '*' '/'   /* left associative, higher precedence */
%%
list:     /* nothing */
        | list '\n'
        | list expr '\n'  { printf("\t%.8g\n", $2); }
        ;
expr:     NUMBER          { $$ = $1; }
        | expr '+' expr   { $$ = $1 + $3; }
        | expr '-' expr   { $$ = $1 - $3; }
        | expr '*' expr   { $$ = $1 * $3; }
        | expr '/' expr   { $$ = $1 / $3; }
        | '(' expr ')'    { $$ = $2; }
        ;
%%
        /* end of grammar */
...
```

There's a lot of new information packed into these few lines. We are not going to explain all of it, and certainly not how the parser works—for that, you will have to read the `yacc` manual.

Alternate rules are separated by '`|`'. Any grammar rule can have an associated action, which will be performed when an instance of that rule is recognized in the input. An action is a sequence of C statements enclosed in braces `{` and `}`. Within an action, `$`$n$ (that is, `$1`, `$2`, etc.) refers to the value returned by the $n$th component of the rule, and `$$` is the value to be returned as the value of the whole rule. So for example, in the rule

```
expr:  NUMBER  { $$ = $1; }
```

`$1` is the value returned by recognizing `NUMBER`; that value is to be returned as the value of the `expr`. The particular assignment `$$=$1` can be omitted—`$$` is always set to `$1` unless you explicitly set it to something else.

At the next level, when the rule is

```
expr:  expr '+' expr  { $$ = $1 + $3; }
```

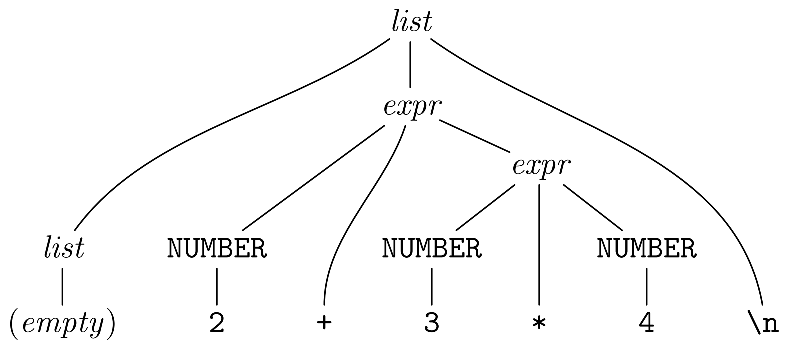
the value of the result `expr` is the sum of the values from the two component `expr`'s. Notice that `'+'` is `$2`; every component is numbered.

At the level above this, an expressoin followed by a newline `'\n'` is recognized as a list and its value is printed. If the end of the input follows such a construction, the parsing process terminates cleanly. A `list` can be an empty string; this is how blank input lines are handled.

`yacc` input is free form; our format is the recommended standard.

In this implementation, the act of recognizing or parsing the input also causes immediate evaluation of the expression. In more complicated situations (including `hoc4` and its successors), the parsing process generates code for later execution.

You may find it helpful to visualize parsing as drawing a *parse tree* like the one in the figure, and to imaginge values being computed and propagated up the tree from the leaves towards the root.




Parse Tree for `2 + 3 * 4`

The values of incompletely-recognized rules are actually kept on a stack; this is how the values are passed from one rule to the next. The data type of this stack is normally an `int`, but since we are processing floating point numbers, we have to override the default. The definition

```
#define YYSTYPE double
```

sets the stack type to `double`.

Syntactic classes that will be recognized by the lexical analyzer have to be declared unless they are single character literals like `+` and `-`. The declaration `%token` declares one or more such objects. Left or right associativity can be specified if appropriate by using `%left` or `%right` instead of `%token`. (Left associativity means that `a-b-c` will be parsed as `(a-b)-c` instead of `a-(b-c)`.) Precedence is determined by order of appearance: tokens in the same declaration are at the same level of precedence; tokens declared later are of higher precedence. In this way the grammar proper is ambiguous (that is, there are multiple ways to parse some inputs), but the extra information in the declarations resolves the ambiguity.

The rest of the code is the routines in the second half of the file `hoc.y`:

*Continuing* `hoc.y`

```
char    *progname;      /* for error messages */
int     lineno = 1;

int main(int argc, char *argv[])   /* hoc1 */
{
        progname = argv[0];
        yyparse();
        return 0;
}
```

`main` calls `yyparse` to parse the input. Looping from one expression to the next is done entirely within the grammar, by the sequence of productions for `list`. It would have been equally acceptable to put a loop around the call to `yyparse` in `main` and have the action for `list` print the value and return immediately.

`yyparse` in turn calls `yylex` repeatedly for input tokens. Our `yylex` is easy: it skips blanks and tabs, converts strings of digits into a numeric value, counts input lines for error reporting, and returns any other character as itself. Since the grammar expects to see only `+`, `-`, `*`, `/`, `(`, `)`, and `\n`, any other character will cause `yyparse` to report an error. Returning a `0` signals "end of file" to `yyparse`.

*Continuing* `hoc.y`

```
int yylex(void)          /* hoc1 */
{
        int c;

        while ((c=getchar()) == ' ' || c == '\t')
                ;
        if (c == EOF)
                return 0;
        if (c == '.' || isdigit(c)) {   /* number */
                ungetc(c, stdin);
                scanf("%lf", &yylval);
                return NUMBER;
        }
        if (c == '\n')
                lineno++;
        return c;
}
```

The variable `yylval` is used for communication between the parser and the lexical analyzer; it is defined by `yyparse`, and has the same type as the `yacc` stack. `yylex` returns the *type* of a token as its function value, and sets `yylval` to the *value* of the token (if there is one). For instance, a floating point number has the type `NUMBER` and a value like 12.34. For some tokens, especially single characters like `'+'` and `'\n'`, the grammar does not use the value, only the type. In that case, `yylval` need not be set.

The `yacc` declaration `%token NUMBER` is converted into a `#define` statement in the `yacc` output file `y.tab.c`, so `NUMBER` can be used as a constant anywhere in the C program. `yacc` chooses values that won't collide with ASCII characters.

If there is a syntax error, `yyparse` calls `yyerror` with a string containing the cryptic message "`syntax error`." The `yacc` user is expected to provide a `yyerror`; ours just passes the string on to another function, `warning`, which prints somewhat more information. Later versions of `hoc` will make direct use of `warning`.

```
void yyerror(char *s)  /* called for yacc syntax error */
{
        warning(s, 0);
}

void warning(char *s, char *t)  /* print warning message */
```

```
{
        fprintf(stderr, "%s: %s", progname, s);
        if (t) fprintf(stderr, " %s", t);
        fprintf(stderr, " near line %d\n", lineno);
}
```

This marks the end of the routines in `hoc.y`.

Compilation of a `yacc` program is a two-step process:

```
$ yacc hoc.y                Leaves output in y.tab.c
$ cc y.tab.c -o hoc1        Leaves executable program in hoc1
$ hoc1
2/3
        0.66666667
-3-4
hoc1: syntax error near line 1
$
```

**Exercise 8-1.** Examine the structure of the `y.tab.c` file. □

**Making changes—unary minus.** We claimed earlier that using `yacc` makes it easy to change a language. As an illustration, let's add unary minus to `hoc1`, so that expressions like `-3-4` are evaluated, not rejected as syntax errors.

Exactly two lines have to be added to `hoc.y`. A new token `UNARYMINUS` is added to the end of the precedence section, to make unary minus have highest precedence:

```
%left  '+' '-'
%left  '*' '/'
%left  UNARYMINUS      /* new */
```

The grammar is augmented with one more production for `expr`:

```
expr:    NUMBER                       { $$ = $1; }
       | '-' expr  %prec UNARYMINUS   { $$ = -$2; }  /* new */
```

The `%prec` says that a unary minus sign (that is, a minus sign before an expression) has the precedence of `UNARYMINUS` (high); the action is to change the sign. A minus sign between two expressions takes the default precedence.

**Exercise 8-2.** Add the operators `%` (modulus or remainder) and unary `+` to `hoc1`. Suggestion: look at `frexp`(3) [or rather `fmod`(3)]. □

**A digression on `make`.** It's a nuisance to have to type two commands to compile a new version of `hoc1`. Although it's certainly easy to make a shell file that does the job, there's a better way, one that will generalize nicely later on when there is more than one source file in the program. The program `make` reads a specification of how the components of a program depend on each other, and how to process them to create an up-to-date version of the program. It checks the times at which the various components were last modified, figures out the minimum amount of recompilation that has to be done to make a consistent new version, then runs the processes. `make` also understands the intricacies of multi-step processes like `yacc`, so these tasks can be put into a `make` specification without spelling out the individual steps.

`make` is most useful when the program being created is large enough to be spread over several source files, but it's handy even for something as small as `hoc1`. Here is the `make` specification for `hoc1`, which `make` expects in a file called `Makefile` or `makefile`.

```
hoc1: hoc.o
	$(CC) $(CFLAGS) hoc.o -o hoc1
```

The second line is indented with a tab, not with blanks! This makefile says that `hoc1` depends on `hoc.o`, and that `hoc.o` is converted into `hoc1` by running the C compiler and putting the output in `hoc1`. `make` already knows how to convert the `yacc` source file in `hoc.y` to an object file `hoc.o`:

```
$ make                   Make the first thing in Makefile, hoc1
yacc hoc.y
cc -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cc hoc.o -o hoc1
$ make                   Do it again
`hoc1' is up to date.    make realizes it's unnecessary
$
```

## Stage 2: Variables and error recovery

The next step (a small one) is to add "memory" to `hoc1`, to make `hoc2`. The memory has 26 variables, named `a` to `z`. This isn't very elegant, but it's an easy and useful intermediate step. We'll also add some error handling. If you try `hoc1`, you'll recognize that its approach to syntax errors is to print a message and die, and its treatment of arithmetic errors like division by zero is reprehensible:

```
$ hoc1
1/0
Floating exception - core dumped
$
```

The changes needed for these new features are modest, about 35 lines of code. The lexical analyzer `yylex` has to recognize letters as variables; the grammar has to include productions of the form

```
expr:   VAR
      | VAR '=' expr
```

An expression can contain an assignment, which permits multiple assignments like

```
x = y = z = 0
```

The easiest way to store the values of the variables is in a 26-element array; the single-letter variable name can be used to index the array. But if the grammar is to process both variable names and values in the same stack, `yacc` has to be told that its stack contains a union of a `double` and an `int`, not just a `double`. This is done with the `%union` declaration near the top. A `#define` or a `typedef` is fine for setting the stack to a basic type like `double`, but the `%union` mechanism is required for union types because `yacc` checks for consistency in expressions like `$$=$2`.

Here is the grammar part of `hoc.y` for `hoc2`:

```
%{
double mem[26];          /* memory for variables 'a' to 'z' */
%}
%union {
        double  val;     /* actual value */
        int     index;   /* index into mem[] */
}
%token  <val>   NUMBER
%token  <index> VAR      /* VAR is index member of union */
%type   <val>   expr     /* expr is val member of union */
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYPM
%%
list:     /* nothing */
        | list '\n'
        | list expr '\n'      { printf("\t%.8g\n", $2); }
        | list error '\n'     { yyerrok; }
        ;
expr:     NUMBER
        | VAR            { $$ = mem[$1]; }
        | VAR '=' expr   { $$ = mem[$1] = $3; }
        | expr '+' expr  { $$ = $1 + $3; }
        | expr '-' expr  { $$ = $1 - $3; }
        | expr '*' expr  { $$ = $1 * $3; }
        | expr '/' expr  { if ($3 == 0.0)
                              execerror("division by zero", "");
                           $$ = $1 / $3; }
        | '(' expr ')'   { $$ = $2; }
        | '-' expr  %prec UNARYPM  { $$ = -$2; }
        ;
```

```
%%
        /* end of grammar */
...
```

The `%union` says that stack elements hold either a `double` (a number, the usual case), or an `int`, which is an index into the array `mem`. The `%token` declarations have been augmented with a type indicator. The `%type` declaration specifies that `expr` is the `<val>` member of the union, i.e., a `double`. The type information makes it possible for `yacc` to generate references to the corect members of the union. Notice also that `=` is right-associative, while the other operators are left-associative.

Error handling comes in several pieces. The obvious one is a test for a zero divisor; if one occurs, an error routine `execerror` is called.

A second test is to catch the "floating point exception" signal that occurs when a floating point number overflows. The signal is set in `main`.

The final part of error recovery is the addition of a production for `error`. "`error`" is a reserved word in a `yacc` grammar; it provides a way to anticipate and recover from a syntax error. If an error occurs, `yacc` will eventually try to use this production, recognize the error as grammatically "correct," and thus recover. The action `yyerrok` sets a flag in the parser that permits it to get back into a sensible parsing state. Error recovery is difficult in any parser; you should be aware that we have taken only the most elementary steps here, and have skipped rapidly over `yacc`'s capabilities as well.

The actions in the `hoc2` grammar are not much changed. Here is `main`, to which we have added `setjmp` to save a clean state suitable for resuming after an error. `execerror` does the matching `longjmp`.

```
...
#include <signal.h>
#include <setjmp.h>
jmp_buf begin;

int main(int argc, char *argv[])  /* hoc2 */
{
        void fpecatch(int);

        progname = argv[0];
        setjmp(begin);
        signal(SIGFPE, fpecatch);
        yyparse();
        return 0;
}

void execerror(char *s, char *t)  /* run-time error recovery */
{
        warning(s, t);
        longjmp(begin, 0);
}
```

```
void fpecatch(int signum)  /* catch floating point exceptions */
{
        execerror("floating point exception", (char *) 0);
}
```

For debugging, we found it convenient to have `execerror` call `abort`(3), which causes a core dump that can be perused with `adb` or `sdb` [or `gdb`]. Once the program is fairly robust, `abort` is replaced by `longjmp`.

The lexical analyzer is a little different in `hoc2`. There is an extra test for a lower-case letter, and since `yylval` is now a union, the proper member has to be set before `yylex` returns. Here are the parts that have changed:

```
int yylex(void)  /* hoc2 */
...
        if (c == '.' || isdigit(c)) {  /* number */
                ungetc(c, stdin);
                scanf("%lf", &yylval.val);
                return NUMBER;
        }
        if (islower(c)) {
                yylval.index = c - 'a';  /* ASCII only */
                return VAR;
        }
...
```

Again, notice how the token type (e.g., `NUMBER`) is distinct from its value (e.g., 3.1416);

Let us illustrate variables and error recovery, the new things in `hoc2`:

```
$ hoc2
x = 355
        355
y = 113
        113
p = x/z                                z is undefined and thus zero
hoc2: division by zero near line 4     Error recovery
x/y
        3.1415929
1e30 * 1e30                            Overflow
hoc2: floating point exception near line 5
...
```

Actually, the PDP-11 requires special arrangements to detect floating point overflow, but on most other machines `hoc2` behaves as shown.

**Exercise 8-3.** Add a facility for remembering the most recent value computed, so that it does not have to be retyped in a sequence of related computations. One solution is to make it one of the variables, for instance '`p`' for 'previous.' □

**Exercise 8-4.** Modify `hoc` so that a semicolon can be used as an expression terminator equivalent to a newline. □

## Stage 3: Arbitrary variable names; built-in functions

This version, `hoc3`, adds several major new capabilities, and a corresponding amount of extra code. The main feature is access to built-in functions:

```
sin    cos    atan    exp    log    log10    sqrt    int    abs
```

We have also added an exponentiation operator '`^`'; it has the highest precendence, and is right-associative.

Since the lexical analyzer has to cope with built-in names longer than a single character, it isn't much extra effort to permit variable names to be arbitrarily long as well. We will need a more sophisticated symbol table to keep track of these variables, but once we have it, we can pre-load it with names and values for some useful constants:

| | | |
|---|---|---|
| `DEG` | 57.29577951308232087680 | $180/\pi$, degrees per radian |
| `E` | 2.71828182845904523536 | $e$, base of natural logarithms |
| `GAMMA` | 0.57721566490153286060 | $\gamma$, Euler-Mascheroni constant |
| `PHI` | 1.61803398874989484820 | $(\sqrt{5}+1)/2$, the golden ratio |
| `PI` | 3.14159265358979323846 | $\pi$, circular transcendental number |

The result is a useful calculator:

```
$ hoc3
1.5^2.3
        2.5410306
exp(2.3*log(1.5))
        2.5410306
sin(PI/2)
        1
atan(1)*DEG
        45
...
```

We have also cleaned up the behavior a little. In `hoc2`, the assignment `x=`*expr* not only causes the assignment but also prints the value, because all expressions are printed:

```
$ hoc2
x = 2 * 3.14159
        6.28318                Value printed for assignment to variable
```

In `hoc3`, a distinction is made between assignments and expressions; values are printed only for expressions:

```
$ hoc3
x = 2 * 3.14159            Assignment: no value is printed
x                          Expression:
        6.28318              value is printed
```

The program that results from all these changes is big enough (about 250 lines) that it is best split into separate files for easier editing and faster compilation. There are now five files instead of one:

| | |
|---|---|
| `hoc.y` | Grammar, `main`, `yylex` (as before) |
| `hoc.h` | Global data structures for inclusion |
| `symbol.c` | Symbol table routines: `lookup`, `install` |
| `init.c` | Built-ins and constants: `init` |
| `math.c` | Interfaces to math routines: `Sqrt`, `Log`, etc. |

This requires that we learn more about how to organize a multi-file C program, and more about `make` so it can do some of the work for us.

We'll get back to `make` shortly. First, let us look at the symbol table code. A symbol has a name, a type (it's either a `VAR` or a `BLTIN`), and a value. If the symbol is a `VAR`, the value is a `double`; if the symbol is a built-in, the value is a pointer to a function that returns a `double`. This information is needed in `hoc.y`, `symbol.c`, and `init.c`. We could just make three copies, but it's too easy to make a mistake or forget to update one copy when a change is made. Instead we put the common information into a header file `hoc.h` that will be included by any file that needs it. (The suffix `.h` is conventional but not enforced by any program.) We will also add to the `Makefile` the fact that these files depend on `hoc.h`, so that when it changes, the necessary recompilations are done too. Here is `hoc.h`:

```
typedef struct Symbol {  /* symbol table entry */
        char    *name;
        short   type;           /* VAR, BLTIN, UNDEF */
        union {
                double  val;            /* if VAR */
                double  (*ptr)();       /* if BLTIN */
        } u;
        struct Symbol   *next;   /* to link to another */
} Symbol;

Symbol *install(char *s, int t, double d);
Symbol *lookup(char *s);

void init(void);
void execerror(char *s, char *t);
```

The type `UNDEF` is a `VAR` that has not yet been assigned a value.

The symbols are linked together in a list using the `next` field in `Symbol`. The list itself is local to `symbol.c`; the only access to it is through the functions `lookup` and `install`. This makes it easy to change the symbol table organization if it becomes necessary. (We did that once.) `lookup` searches the list for a particular name and returns a pointer to the `Symbol` with that name if found, and zero otherwise. The symbol table uses linear search, which is entirely adequate for our interactive calculator, since variables are looked up only during parsing, not execution. `install` puts a variable with its associated type and value at the head of the list. `emalloc` calls `malloc`(3), the standard storage allocator, and checks the result. These three routines are the contents of `symbol.c`. The file `y.tab.h` is generated by running `yacc -d`; it contains `#define` statements that `yacc` has generated for tokens like `NUMBER`, `VAR`, `BLTIN`, etc. Here is `symbol.c`:

```
#include "hoc.h"
#include "y.tab.h"
#include <stdlib.h>
#include <string.h>

void *emalloc(unsigned nbytes);

static Symbol *symlist = 0;  /* symbol table: linked list */

Symbol *lookup(char *s)  /* find s in symbol table */
{
        Symbol *sp;
        for (sp = symlist; sp; sp = sp->next)
                if (strcmp(sp->name, s) == 0)
                        return sp;
        return 0;  /* not found */
}

Symbol *install(char *s, int t, double d)  /* add s to symtab */
{
        Symbol *sp = emalloc(sizeof(Symbol));
        sp->name = emalloc(strlen(s)+1);  /* +1 for '\0' */
        strcpy(sp->name, s);
        sp->type = t;
        sp->u.val = d;
        sp->next = symlist;  /* put at front of list */
        symlist = sp;
        return sp;
}

void *emalloc(unsigned nbytes)  /* check return from malloc */
{
        void *p = malloc(nbytes);
        if (!p) execerror("out of memory", 0);
        return p;
}
```

The file `init.c` contains definitions for the constants (`PI`, etc.) and function pointers for built-ins; they are installed in the symbol table by the function `init`, which is called by `main`. Here is `init.c`:

```c
#include "hoc.h"
#include "y.tab.h"
#include <math.h>

extern double Log(), Log10(), Exp(), Sqrt(), integer();

static struct {        /* Constants */
    char    *name;
    double  cval;
} consts[] = {
    { "PI",    3.14159265358979323846 },
    { "E",     2.71828182845904523536 },
    { "GAMMA", 0.57721566490153286060 },  /* Euler */
    { "DEG",  57.29577951308232087680 },  /* deg/radian */
    { "PHI",   1.61803398874989484820 },  /* golden ratio */
    { 0,       0 }
};

static struct {        /* Built-ins */
    char    *name;
    double  (*func)();
} builtins[] = {
    { "sin",    sin     },
    { "cos",    cos     },
    { "atan",   atan    },
    { "log",    Log     },  /* checks argument */
    { "log10",  Log10   },  /* checks argument */
    { "exp",    Exp     },  /* checks argument */
    { "sqrt",   Sqrt    },  /* checks argument */
    { "int",    integer },
    { "abs",    fabs    },
    { 0,        0       }
};

void init(void)  /* install constants and built-ins in symtab */
{
        int i;
        Symbol *sp;

        for (i = 0; consts[i].name; i++)
                install(consts[i].name, VAR, consts[i].cval);
        for (i = 0; builtins[i].name; i++) {
                sp = install(builtins[i].name, BLTIN, 0.0);
                sp->u.ptr = builtins[i].func;
        }
}
```

The data is kept in tables rather than being wired into the code because tables are easier to read and to change. The tables are declared `static` so that they are visible only within this file rather than throughout the program. We'll come back to the math routines like `Log` and `Sqrt` shortly.

With the foundation in place, we can move on to the changes in the grammar that make use of it. Here is `hoc.y`:

```
%{
#include "hoc.h"
#include <stdio.h>
extern double Pow();
%}
%union {
        double  val;    /* actual value */
        Symbol *sym;    /* symbol table pointer */
}
%token  <val>   NUMBER
%token  <sym>   VAR BLTIN UNDEF
%token  <val>   expr asgn
%right  '='
%left   '+' '-'
%left   '*' '/'
%left   UNARYPM
%right  '^'     /* exponentiation */
%%
list:     /* nothing */
        | list       '\n'
        | list asgn  '\n'
        | list expr  '\n'    { printf("\t%.8g\n", $2); }
        | list error '\n'    { yyerrok; }
        ;
asgn:     VAR '=' expr  { $$ = $1->u.val = $3; $1->type = VAR; }
        ;
expr:     NUMBER
        | VAR { if ($1->type == UNDEF)
                   execerror("undefined variable", $1->name);
                $$ = $1->u.val; }
        | asgn
        | BLTIN '(' expr ')'  { $$ = (*($1->u.ptr))($3); }
        | expr '+' expr  { $$ = $1 + $3; }
        | expr '-' expr  { $$ = $1 - $3; }
        | expr '*' expr  { $$ = $1 * $3; }
        | expr '/' expr  { if ($3 == 0.0)
                              execerror("division by zero", "");
                           $$ = $1 / $3; }
```

```
        | expr '^' expr  { $$ = Pow($1, $3); }
        | '(' expr ')'   { $$ = $2; }
        | '-' expr  %prec UNARYPM  { $$ = -$2; }
        | '+' expr  %prec UNARYPM  { $$ = $2; }
        ;
%%
        /* end of grammar */
...
```

The grammar now has `asgn`, for assignment, as well as `expr`; an input line that contains just

```
VAR = expr
```

is an assignment, and so no value is printed. Notice, by the way, how easy it was to add exponentiation to the grammar, including its right associativity.

The `yacc` stack has a different `%union`: instead of referring to a variable by its index in a 26-element table, there is a pointer to an object of type `Symbol`. The header file `hoc.h` contains the definition of thi stype.

The lexical analyzer recognizes variable names, looks them up in the symbol table, an ddecides whether they are variables (`VAR`) or built-ins (`BLTIN`). The type returned by `yylex` is one of these; both user-defined variables and pre-defined variables like `PI` are `VAR`'s.

One of the properties of a variable is whether or not it has been assigned a value, so the use of an undefined variable can be reported as an error by `yyparse`. The test for whether a variable is defined has to be in the grammar, not in the lexical analyzer. When a `VAR` is recognized lexically, its context isn't yet known; we don't want a complaint that `x` is undefined when the context is a perfectly legal one such as the left side of an assignment like `x=1`.

Here is the revised part of `yylex`:

```
int yylex(void)  /* hoc3 */
...
        if (isalpha(c)) {
                Symbol *sp;
                char sbuf[100], *p = sbuf;
                do {
                        *p++ = c;
                } while ((c=getchar()) != EOF && isalnum(c));
                ungetc(c, stdin);
                *p = '\0';
                if ((sp=lookup(sbuf)) == 0)
                        sp = install(sbuf, UNDEF, 0.0);
                yylval.sym = sp;
                return sp->type == UNDEF ? VAR : sp->type;
        }
...
```

[Required exercise: find the buffer overrun bug and fix it!]

`main` has one extra line, which calls the initialization routine `init` to install built-ins and pre-defined names like `PI` in the symbol table. Here it is:

```
#include <setjmp.h>
#include <signal.h>

int main(int argc, char *argv[])  /* hoc3 */
{
        void fpecatch(int);

        progname = argv[0];
        init();
        setjmp(begin);
        signal(SIGFPE, fpecatch);
        yyparse();
        return 0;
}
```

The only remaining file is `math.c`. Some of the standard mathematical functions need an error-checking interface for messages and recovery—for example the standard function `sqrt` silently returns zero if its argument is negative. The code in `math.c` uses the error tests found in Section 2 of the *UNIX Programmer's Manual*. This is more reliable and portable than writing our own tests, since presumably the specific limitations of the routines are best reflected in the "official" code. The header file `<math.h>` contains type declarations for the standard mathematical functions. `<errno.h>` contains names for the errors that can be incurred. Here is `math.c`:

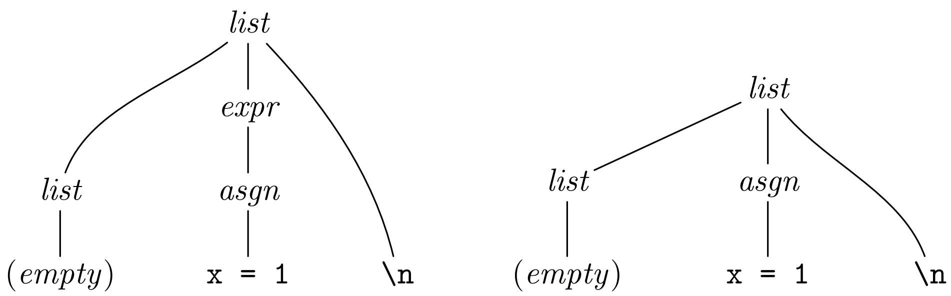
```
#include "hoc.h"
#include <math.h>
#include <errno.h>

double errcheck(double d, char *s);

double Log(double x)   { return errcheck(log(x),   "log");   }
double Log10(double x) { return errcheck(log10(x), "log10"); }
double Exp(double x)   { return errcheck(exp(x),   "exp");   }
double Sqrt(double x)  { return errcheck(sqrt(x),  "sqrt");  }
double Pow(double x, double y) {
                         return errcheck(pow(x,y), "pow");   }

double integer(double x) { return (double)(long) x; }

double errcheck(double d, char *s)
{ /* check result of library call */
        if (errno == EDOM) {
                errno = 0;
                execerror(s, "argument out of domain");
        }
```

```
        else if (errno == ERANGE) {
                errno = 0;
                execerror(s, "result out of range");
        }
        return d;
}
```

An interesting (and ungrammatical) diagnostic appears when we run `yacc` on the new grammar:

```
$ yacc hoc.y
conflicts: 1 shift/reduce
$
```

The "shift/reduce" message means that the `hoc3` grammar is ambiguous: the single line of input `x = 1` can be parsed in two ways:




The parser can decide that the *asgn* should be reduced to an *expr* and then to a *list*, as in the parse tree on the left, or it can decide to use the following `\n` immediately ("shift") and convert the whole thing to a *list* without the intermediate rule, as in the tree on the right. Given the ambiguity, `yacc` chooses to shift, since this is almost always the right thing to do with real grammars. You should try to understand such messages, to be sure that `yacc` has made the right decision.† Running `yacc` with the option `-v` produces a voluminous file called `y.output` that hints at the origin of conflicts.

**Exercise 8-5.** As `hoc3` stands, it's legal to say `PI = 3`. Is this a good idea? How would you change `hoc3` to prohibit assignment to "constants"? □

**Exercise 8-6.** Add the built-in function `atan2(y,x)`, which returns the angle whose tangent is `y/x`. Add the built-in `rand()`, which returns a floating point random variable uniformly distributed on the interval $(0, 1)$. How do you have to change the grammar to allow for built-ins with different numbers of arguments? □

**Exercise 8-7.** How would you add a facility to execute commands from within `hoc`, similar to the `!` feature of other UNIX programs? □

**Exercise 8-8.** Revise the code in `math.c` to use a table instead of the set of essentially identical functions that we presented. □

---

† The `yacc` message "reduce/reduce conflict" indicates a serious problem, more often the symptom of an outright error in the grammar than an intentional ambiguity.

**Another digression on `make`.** Since the program for `hoc3` now lives on five files, not one, the `Makefile` is more complicated:

```
YFLAGS = -d                    # force creation of y.tab.h
OBJS = hoc.o init.o math.o symbol.o        # abbreviation

hoc3: $(OBJS)
        $(CC) $(OBJS) -lm -o hoc3

hoc.o: hoc.h

init.o symbol.o: hoc.h y.tab.h

pr:
        @pr hoc.y hoc.h init.c math.c symbol.c Makefile

clean:
        rm -f $(OBJS) y.tab.[ch]
```

The `YFLAGS = -d` line adds the option `-d` to the `yacc` command line generated by `make`; this tells `yacc` to produce the `y.tab.h` file of `#define` statements. The `OBJS = ...` line defines a shorthand for a construct to be used several times subsequently. The syntax is not the same as for shell variables—the parentheses are mandatory. The flag `-lm` causes the math library to be searched for the mathematical functions.

`hoc3` now depends on four `.o` files; some of the `.o` files depend on `.h` files. Given these dependencies, `make` can deduce what recompilation is needed after changes are made to any of the files involved. If you want to see what `make` will do without actually running the processes, try

```
$ make -n
```

On the other hand, if you want to force the file times into a consistent state, the `-t` ("touch") option will update them without doing any compilation steps.

Notice that we have added not only a set of dependencies for the source files but miscellaneous utility routines as well, all neatly encapsulated in one place. By default, `make` makes the first thing listed in the `Makefile`, but if you name an item that labels a dependency rule, like `symbol.o` or `pr`, that will be made instead. An empty dependency is taken to mean that the item is never "up to date," so that action will always be done when requested. Thus

```
$ make pr | lpr
```

produces the listing you asked for on a line printer. (The leading `@` in "`@pr`" suppresses the echo of the command being executed by `make`.) And

```
$ make clean
```

removes the `yacc` output files and the `.o` files.

This mechanism of empty dependencies in the `Makefile` is often preferable to a shell file as a way to keep all the related computations in a single file. And `make` is not restricted to program development—it is valuable for packaging any set of operations that have time dependencies.

**A digression on `lex`.** The program `lex` creates lexical analyzers in a manner analogous to the way that `yacc` creates parsers: you write a specification of the lexical rules of your language, using regular expressions and fragments of C to be executed when a matching string is found. `lex` translates that into a recognizer. `lex` and `yacc` cooperate by the same mechanism as the lexical analyzers we have already written. We are not going into great detail on `lex` here; the following discussion is mainly to interest you in learning more.

First, here is the `lex` program, from the file `lex.l`; it replaces the function `yylex` that we have used so far.

```
%{
#include "hoc.h"
#include "y.tab.h"
extern int lineno;
%}
%%
[ \t]   { ; }   /* skip blanks and tabs */
[0-9]+\.?|[0-9]*\.[0-9]+ {
        sscanf(yytext, "%lf", &yylval.val); return NUMBER; }
[a-zA-Z][a-zA-Z0-9]* {
        Symbol *s;
        if ((s = lookup(yytext)) == 0)
                s = install(yytext, UNDEF, 0.0);
        yylval.sym = s;
        return s->type == UNDEF ? VAR : s->type; }
\n      { lineno++; return '\n'; }
.       { return yytext[0]; }   /* everything else */
```

Each "rule" is a regular expression like those in `egrep` or `awk`, except that `lex` recognizes C-style escapes like `\t` and `\n`. The action is enclosed in braces. The rules are attempted in order, and constructs like `*` and `+` match as long a string as possible. If the rule matches the next part of the input, the action is performed. The input string that matched is accessible in a `lex` string called `yytext`.

The `Makefile` has to be changed to use `lex`:

```
YFLAGS = -d
OBJS = hoc.o lex.o init.o math.o symbol.o

hoc3: $(OBJS)
        $(CC) $(OBJS) -lm -ll -o hoc3

hoc.o: hoc.h

lex.o init.o symbol.o: hoc.h y.tab.h
...
```

Again, `make` knows how to get from a `.l` file to the proper `.o`; all it needs from us is the dependency information. (We also have to add the `lex` library `-ll` to the list searched by cc since the lex-generated recognizer is not self-contained.) The output is spectacular and completely automatic:

```
$ make
yacc -d hoc.y
conflicts: 1 shift/reduce
cc  -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
lex  lex.l
cc  -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc  -c init.c
cc  -c math.c
cc  -c symbol.c
cc hoc.o lex.o init.o math.o symbol.o -lm -ll -o hoc3
$
```

If a signle file is changed, the single command make is enough to make an up-to-date version:

```
$ touch lex.l                    Change modified-time of  lex.l
$ make
lex  lex.l
cc  -c lex.yy.c
rm lex.yy.c
mv lex.yy.o lex.o
cc hoc.o lex.o init.o math.o symbol.o -lm -ll -o hoc3
$
```

We debated for quite a while whether to treat lex as a digression, to be illustrated briefly and then dropped, or as the primary tool for lexical analysis once the language got complicated. There are arguments on both sides. The main problem with lex (aside from requiring that the user learn yet another language) is that it tends to be slow to run and to produce bigger and slower recognizers than the equivalent C versions. It is also somewhat harder to adapt its input mechanism if one is doing anything unusual, such as error recovery or even input from files. None of these issues is serious in the context of hoc. The main limitation is space: it takes more pages to describe the lex version, so (regretfully) we will revert to C for subsequent lexical analysis. It is a good exercise to do the lex versions, however.

**Exercise 8-9.** Compare the sizes of the two versions of hoc3. Hint: see size(1). □

## Stage 4: Compmilation into a machine

We are heading towards `hoc5`, an interpreter for a language with control flow. `hoc4` is an intermediate step, providing the same functions as `hoc3`, but implemented within the interpreter framework of `hoc5`. We actually wrote `hoc4` this way, since it gives us two programs that should behave identically, which is valuable for debugging. As the input is parsed, `hoc4` generates code for a simple computer instead of immediately computing answers. Once the end of a statement is reached, the generated code is executed ("interpreted") to compute the desired result.

The simple computer is a *stack machine*: when an operand is encountered, it is pushed onto a stack (more precisely, code is generated to push it onto a stack); most operators operate on items on the top of the stack. For example, to handle the assignment

```
x = 2 * y
```

the following code is generated:

| | |
|---|---|
| `constpush` | *Push a constant onto stack* |
| `  2` | *... the constant* `2` |
| `varpush` | *Push symbol table pointer onto stack* |
| `  y` | *... for the variable* `y` |
| `eval` | *Evaluate: replace pointer by value* |
| `mul` | *Multiply top two items; product replaces them* |
| `varpush` | *Push symbol table pointer onto stack* |
| `  x` | *... for the variable* `x` |
| `assign` | *Store value in variable, pop pointer* |
| `pop` | *Clear top value from stack* |
| `STOP` | *End of instruction sequence* |

When this code is executed, the expression is evaluated and the result is stored in `x`, as indicated by the comments. The final `pop` clears the value off the stack because it is not needed any longer.

Stack machines usually result in simple interpreters, and ours is no exception: it's just an array containing operators and operands. The operators are the machine instructions; each is a function call with its arguments, if any, following the instruction. Other operands may already be on the stack, as they were in the example above.

The symbol table code for `hoc4` is identical to that for `hoc3`; the initialization in `init.c` and the mathematical functions in `math.c` are the same as well. The grammar is the same as for `hoc3`, but the actions are quite different. Basically, each action generates machine instructions and any arguments that go with them. For example, three items are generated for a `VAR` in an expression: a `varpush` instruction, the symbol table pointer for the variable, and an `eval` instruction that will replace the symbol table pointer by its value when executed. The code for '`*`' is just `mul`, since the operands for that will already bo on the stack.

```
%{
#include "hoc.h"
```

```
#define code2(c1,c2)    code(c1); code(c2)
#define code3(c1,c2,c3) code(c1); code(c2); code(c3)
%}
%union{
        Symbol  *sym;   /* symbol table pointer */
        Inst    *inst;  /* machine instruction */
}
%token  <sym>   NUMBER VAR BLTIN UNDEF
%right  '='
%left   '+' '-'
%left   '*' '-'
%left   UNARYPM
%right  '^'     /* exponentiation */
%%
list:     /* nothing */
        | list       '\n'
        | list asgn  '\n'  { code2(drop, STOP); return 1; }
        | list expr  '\n'  { code2(print, STOP); return 1; }
        | list error '\n'  { yyerrok; }
        ;
asgn:     VAR '=' expr   { code3(varpush, (Inst)$1, assign); }
        ;
expr:     NUMBER         { code2(constpush, (Inst)$1); }
        | VAR            { code3(varpush, (Inst)$1, eval); }
        | asgn
        | BLTIN '(' expr ')' { code2(bltin, (void*)$1->u.ptr); }
        | expr '+' expr  { code(add); }
        | expr '-' expr  { code(sub); }
        | expr '*' expr  { code(mul); }
        | expr '/' expr  { code(div); }
        | expr '^' expr  { code(power); }
        | '(' expr ')'
        | '-' expr  %prec UNARYPM  { code(negate); }
        | '+' expr  %prec UNARYPM
        ;
%%
        /* end of grammar */
...
```

`Inst` is the data type of a machine instruction (a pointer to a function returning an `int`), which we will return to shortly. Notice that the arguments to `code` are function names, that is, pointers to functions, or other values that are coerced to function pointers.

We have changed main somewhat. The parser now returns after each statement or expression; the code that it generated is executed. yyparse returns zero at end of file.

```
#include <ctype.h>
#include <setjmp.h>
#include <signal.h>
#include <stdio.h>

int main(int argc, char *argv[])  /* hoc4 */
{
        void fpecatch(int);

        progname = argv[0];
        init();
        setjmp(begin);
        signal(SIGFPE, fpecatch);

        for (initcode(); yyparse(); initcode())
                execute(prog);

        return 0;
}
```

The lexical analyzer is only a little different. The main change is that numbers have to be preserved, not used immediately. The easiest way to do this is to install them in the symbol table along with the variables. Here is the changed part of yylex:

```
int yylex(void)  /* hoc4 */
...
        if (c == '.' || isdigit(c)) {  /* number */
                double d;
                ungetc(c, stdin);
                scanf("%lf", &d);
                yylval.sym = install("", NUMBER, d);
                return NUMBER;
        }
...
```

Each element on the interpreter stack is either a floating point value or a pointer to a symbol table entry; the stack data type is a union of these. The machine itself is an array of pointers that point either to routines like mul that perform an operation, or to data in the symbol table. The header file hoc.h has to be augmented to include these data structures and function declarations for the interpreter, so they will be known where necessary throughout the program. (By the way, we chose to put all this information in one file instead of two. In a larger program, it might be better to divide the header information into several files so that each is included only where really needed.) Here is hoc.h:

```
typedef struct Symbol {  /* symbol table entry */
        char   *name;
        short   type;    /* VAR, BLTIN, UNDEF */
        union {
                double  val;         /* if VAR */
                double  (*ptr)();    /* if BLTIN */
        } u;
        struct Symbol   *next;  /* to link to another */
} Symbol;

Symbol *install(char *s, int t, double d);
Symbol *lookup(char *s);

void init(void);
void execerror(char *s, char *t);

typedef union Datum {    /* interpreter stack type */
        double  val;     /* for literal numbers */
        Symbol *sym;     /* for variables */
} Datum;

typedef void (*Inst)();  /* machine instruction */
#define STOP (Inst) 0

extern  Inst prog[];
extern  void constpush(), varpush(), drop();
extern  void add(), sub(), mul(), divide(), power(), negate();
extern  void eval(), assign(), bltin(), print();

extern  void initcode(void);
extern  Inst *code(Inst f);
extern  void execute(Inst *p);
```

[Renamed `div` to `divide` to avoid collision with `div`(3) from the standard library.]

The routines that execute the machine instructions and manipulate the stack are kept in a new file called `code.c`. Since it is about 150 lines long, we will show it in pieces.

```
#include "hoc.h"
#include "y.tab.h"
#include <stdio.h>

#define NSTACK  256
static  Datum   stack[NSTACK];  /* the stack */
static  Datum  *stackp;         /* next free spot on stack */

#define NPROG   2000
static  Inst    prog[NPROG];    /* the machine */
static  Inst   *progp;          /* next free spot (code gen) */
static  Inst   *pc;             /* program counter (runtime) */
```

```
void initcode(void)  /* initialize for code generation */
{
        stackp = stack;
        progp = prog;
}
```

The stack is manipulated by calls to `push` and `pop`:

```
static void push(Datum d)  /* push d onto stack */
{
        if (stackp >= &stack[NSTACK])
                execerror("stack overflow", 0);
        *stackp++ = d;
}

static Datum pop(void)  /* pop and return top from stack */
{
        if (stackp <= stack)
                execerror("stack underflow", 0);
        return *--stackp;
}
```

The machine is generated during parsing by calls to the function `code`, which simply puts an instruction into the next free spot in the array `prog`. It returns the location of the instruction (which is not used in `hoc4`).

```
Inst *code(Inst f)  /* install one instruction or operand */
{
        Inst *oprogp = progp;
        if (progp >= &prog[NPROG])
                execerror("program too big", 0);
        *progp++ = f;
        return oprogp;
}
```

Execution of the machine is simple; in fact, it's rather neat how small the routine is that "runs" the machine once it's set up:

```
void execute(Inst *p)  /* run the machine */
{
        for (pc = p; *pc != STOP; )
                (*(*pc++))();
}
```

Each cycle executes the function pointed to by the instruction pointed to by the program counter `pc`, and increments `pc` so it's ready for the next instruction. An instruction with opcode `STOP` terminates the loop. Some instructions, such as `constpush` and `varpush`, also increment `pc` to step over any arguments that follow the instruction.

```
void constpush(void)  /* push constant onto stack */
{
        Datum d;
        d.val = ((Symbol *) *pc++)->u.val;
        push(d);
}

void varpush(void)  /* push variable onto stack */
{
        Datum d;
        d.sym = (Symbol *) (*pc++);
        push(d);
}

void drop(void)  /* pop and discard top item from stack */
{
        (void) pop();
}
```

[`drop` is the instruction form of `pop`; in the book, `pop` is used as an instruction, but this fails with recent C compilers because it has a different signature.]

The rest of the machine is easy. For instance, the arithmetic operations are all basically the same, and were created by editing a single prototype. Here is `add`:

```
void add(void)  /* add top two elems on stack */
{
        Datum d2 = pop();
        Datum d1 = pop();
        d1.val += d2.val;
        push(d1);
}
```

The remaining routines are equally simple.

```
void eval(void)  /* evaluate variable on stack */
{
        Datum d = pop();
        if (d.sym->type == UNDEF)
                execerror("undefined variable", d.sym->name);
        d.val = d.sym->u.val;
        push(d);
}

void assign(void)  /* assign to top var next value */
```

```
{
        Datum d1 = pop();
        Datum d2 = pop();
        if (d1.sym->type != VAR && d1.sym->type != UNDEF)
                execerror("assignment to non-variable",
                          d1.sym->name);
        d1.sym->u.val = d2.val;
        d1.sym->type = VAR;
        push(d2);  /* push back value because asgn is an expr */
}

void print(void)  /* pop top value from stack, print it */
{
        Datum d = pop();
        printf("\t%.8g\n", d.val);
}

void bltin(void)  /* evaluate built-in on top of stack */
{
        Datum d = pop();
        d.val = (*(double (*)())(void*)(*pc++))(d.val);
        push(d);
}
```

The hardest part is the cast in `bltin`, which says that `*pc` should be cast to "pointer to function returning a `double`," and that function executed with `d.val` as argument. [The intermediate `(void*)` cast silences a compiler warning because an `Inst` returns `int`, not `double`.]

The diagnostics in `eval` and `assign` should never occur if everything is working properly; we left them in in case some program error causes the stack to be curdled. The overhead in time and space is small compared to the benefit of detecting the error if we make a careless change in the program. (We did, several times).

C's ability to manipulate pointers to functions leads to compact and efficient code. An alternative, to make the operators constants and combine the semantic functions into a big `switch` statement in `execute`, is straightforward and is left as an exercise.

**A third digression on `make`.** As the source code for `hoc` grows, it becomes more and more valuable to keep track mechanically of what has changed and what depends on that. The beauty of `make` is that it automates jobs that we would otherwise do by hand (and get wrong sometimes) or by creating a specialized shell file.

We have made two improvements to the `Makefile`. The first is based on the observation that although several files depend on the `yacc`-defined constants in `y.tab.h`, there's no need to recompile them unless the constants change—changes to the C code in `hoc.y` don't affect anything else. In the new `Makefile` the `.o` files depend on a new file `x.tab.h` that is updated only when the *contents* of `y.tab.h` change. The second improvement is to make the rule for `pr` (printing the source files) depend on the source files, so that only changed files are printed.

The first of these changes is a great time-saver for larger programs when the grammar is static but the semantics are not (the usual situation). The second change is a great paper-saver.

Here is the new `Makefile` for `hoc4`:

```
YFLAGS = -d
OBJS = hoc.o code.o init.o math.o symbol.o

hoc4: $(OBJS)
	$(CC) $(OBJS) -o hoc4 -lm

hoc.o code.o init.o symbol.o: hoc.h
code.o init.o symbol.o: x.tab.h

x.tab.h: y.tab.h
	-cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h

pr: hoc.y hoc.h code.c init.c math.c symbol.c
	@pr $?
	@touch pr

clean:
	rm -f $(OBJS) [xy].tab.[ch]
```

The '`-`' before `cmp` tells `make` to carry on even if the `cmp` fails; this permits the process to work even if `x.tab.h` doesn't exist. (The `-s` option causes `cmp` to produce no output but set the exit status.) The symbol `$?` expands into the list of items from the rule that are not up to date. Regrettably, `make`'s notational conventions are at best loosely related to those of the shell.

To illustrate how these operate, suppose that everything is up to date. Then:

```
$ touch hoc.y                    Change date of hoc.y
$ make
yacc -d hoc.y
conflicts: 1 shift/reduce
cc  -c y.tab.c
rm y.tab.c
mv y.tab.o hoc.o
cmp -s x.tab.h y.tab.h || cp y.tab.h x.tab.h
cc hoc.o code.o init.o math.o symbol.o -o hoc4 -lm
$ make -n pr                     Print changed files
pr hoc.y
touch pr
$
```

Notice that nothing was recompiled except `hoc.y`, because the `y.tab.h` file was the same as the previous one.

**Exercise 8-10.** Make the sizes of `stack` and `prog` dynamic, so that `hoc4` never runs out of space if memory can be obtained by calling `malloc`. □

**Exercise 8-11.** Modify `hoc4` to use a `switch` on the type of operation in `execute` instead of calling functions. How do the versions compare in lines of source code and execution spead? How are they likely to compare in case of maintenance and growth? □