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 fro ma 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 π (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 picket calculator, and is substantially less portable. 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 specificatoin 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 connectiong 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 noc1 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, bt this convention is not enforced by yacc itself, unlike cc and .c.) The grammar part is the first half of hoc.y:

```
$ cat hoc.y
%{
#define YYSTYPE double  /* data type of yacc stack */
%}
%token  NUMBER
```

```
%left   '+' '-'   /* left associative, same precedence */
%left   '*' '/'   /* left assoc., 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 te `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.