For more information on shift/reduce parsing, see Chapter 8. For a discussion of what yacc has to do to turn your specification into a working C program, see the classic compiler text by Aho, Sethi, and Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986, often known as the "dragon book" because of the cover illustration.

A Yacc Parser

A yacc grammar has the same three-part structure as a lex specification. (Lex copied its structure from yacc.) The first section, the definition section, handles control information for the yacc-generated parser (from here on we will call it the parser), and generally sets up the execution environment in which the parser will operate. The second section contains the rules for the parser, and the third section is C code copied verbatim into the generated C program.

We'll first write parser for the simplest grammar, the one in Figure 3-1, then extend it to be more useful and realistic.

The Definition Section

The definition section includes declarations of the tokens used in the grammar, the types of values used on the parser stack, and other odds and ends. It can also include a literal block, C code enclosed in %{ %} lines. We start our first parser by declaring two symbolic tokens.

```
%token NAME NUMBER
```

You can use single quoted characters as tokens without declaring them, so we don't need to declare "=", "+", or "–".

The Rules Section

The rules section simply consists of a list of grammar rules in much the same format as we used above. Since ASCII keyboards don't have a → key, we use a colon between the left- and right-hand sides of a rule, and we put a semicolon at the end of each rule:

```
%token NAME NUMBER
%%
statement:  NAME '=' expression
        |   expression
        ;
```



```
expression: NUMBER '+' NUMBER
        |   NUMBER '-' NUMBER
        ;
```

Unlike lex, yacc pays no attention to line boundaries in the rules section, and you will find that a lot of whitespace makes grammars easier to read. We've added one new rule to the parser: a statement can be a plain expression as well as an assignment. If the user enters a plain expression, we'll print out its result.

The symbol on the left-hand side of the first rule in the grammar is normally the start symbol, though you can use a **%start** declaration in the definition section to override that.

Symbol Values and Actions

Every symbol in a yacc parser has a *value*. The value gives additional information about a particular instance of a symbol. If a symbol represents a number, the value would be the particular number. If it represents a literal text string, the value would probably be a pointer to a copy of the string. If it represents a variable in a program, the value would be a pointer to a symbol table entry describing the variable. Some tokens don't have a useful value, e.g., a token representing a close parenthesis, since one close parenthesis is the same as another.

Non-terminal symbols can have any values you want, created by code in the parser. Often the action code builds a parse tree corresponding to the input, so that later code can process a whole statement or even a whole program at a time.

In the current parser, the value of a *NUMBER* or an *expression* is the numerical value of the number or expression, and the value of a *NAME* will be a symbol table pointer.

In real parsers, the values of different symbols use different data types, e.g., *int* and *double* for numeric symbols, *char ** for strings, and pointers to structures for higher level symbols. If you have multiple value types, you have to list all the value types used in a parser so that yacc can create a C *union* typedef called *YYSTYPE* to contain them. (Fortunately, yacc gives you a lot of help ensuring that you use the right value type for each symbol.)

In the first version of the calculator, the only values of interest are the numerical values of input numbers and calculated expressions. By default

yacc makes all values of type *int*, which is adequate for our first version of the calculator.

Whenever the parser reduces a rule, it executes user C code associated with the rule, known as the rule's *action*. The action appears in braces after the end of the rule, before the semicolon or vertical bar. The action code can refer to the values of the right-hand side symbols as **$1**, **$2**, ..., and can set the value of the left-hand side by setting **$$**. In our parser, the value of an *expression* symbol is the value of the expression it represents. We add some code to evaluate and print expressions, bringing our grammar up to that used in Figure 3-2.

```
%token NAME NUMBER
%%
statement:  NAME '=' expression
        |   expression          { printf("= %d\n", $1); }
        ;

expression: expression '+' NUMBER   { $$ = $1 + $3; }
        |   expression '-' NUMBER   { $$ = $1 - $3; }
        |   NUMBER                  { $$ = $1; }
        ;
```

The rules that build an expression compute the appropriate values, and the rule that recognizes an expression as a statement prints out the result. In the expression building rules, the first and second numbers' values are **$1** and **$3**, respectively. The operator's value would be **$2**, although in this grammar the operators do not have interesting values. The action on the last rule is not strictly necessary, since the default action that yacc performs after every reduction, before running any explicit action code, assigns the value **$1** to **$$**.

The Lexer

To try out our parser, we need a lexer to feed it tokens. As we mentioned in Chapter 1, the parser is the higher level routine, and calls the lexer **yylex()** whenever it needs a token from the input. As soon as the lexer finds a token of interest to the parser, it returns to the parser, returning the token code as the value. Yacc defines the token names in the parser as C preprocessor names in *y.tab.h* (or some similar name on MS-DOS systems) so the lexer can use them.

Here is a simple lexer to provide tokens for our parser:

```
%{
#include "y.tab.h"
extern int yylval;
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
[ \t] ;         /* ignore whitespace */
\n      return 0;   /* logical EOF */
.       return yytext[0];
%%
```

Strings of digits are numbers, whitespace is ignored, and a newline returns an end of input token (number zero) to tell the parser that there is no more to read. The last rule in the lexer is a very common catch-all, which says to return any character otherwise not handled as a single character token to the parser. Character tokens are usually punctuation such as parentheses, semicolons, and single-character operators. If the parser receives a token that it doesn't know about, it generates a syntax error, so this rule lets you handle all of the single-character tokens easily while letting yacc's error checking catch and complain about invalid input.

Whenever the lexer returns a token to the parser, if the token has an associated value, the lexer must store the value in **yylval** before returning. In this first example, we explicitly declare **yylval**. In more complex parsers, yacc defines **yylval** as a *union* and puts the definition in *y.tab.h*.

We haven't defined **NAME** tokens yet, just **NUMBER** tokens, but that is OK for the moment.

Compiling and Running a Simple Parser

On a UNIX system, yacc takes your grammar and creates *y.tab.c*, the C language parser, and *y.tab.h*, the include file with the token number definitions. Lex creates *lex.yy.c*, the C language lexer. You need only compile them together with the yacc and lex libraries. The libraries contain usable default versions of all of the supporting routines, including a **main()** that calls the parser **yyparse()** and exits.

```
% yacc -d ch3-01.y    # makes y.tab.c and "y.tab.h
% lex ch3-01.l        # makes lex.yy.c
% cc -o ch3-01 y.tab.c lex.yy.c -ly -ll    # compile and link C files
% ch3-01
99+12
= 111
% ch3-01
```