To define the possible symbol types, in the definition section we add a %union declaration:

```
%union {
        double dval;
        int vblno;
}
```

The contents of the declaration are copied verbatim to the output file as the contents of a C **union** declaration defining the type YYSTYPE as a C typedef. The generated header file *y.tab.h* includes a copy of the definition so that you can use it in the lexer. Here is the *y.tab.h* generated from this grammar:

```
#define NAME 257
#define NUMBER 258
#define UMINUS 259
typedef union {
        double dval;
        int vblno;
} YYSTYPE;
extern YYSTYPE yylval;
```

The generated file also declares the variable **yylval**, and defines the token numbers for the symbolic tokens in the grammar.

Now we have to tell the parser which symbols use which type of value. We do that by putting the appropriate field name from the union in angle brackets in the lines in the definition section that defines the symbol:

```
%token <vblno> NAME
%token <dval> NUMBER

%type <dval> expression
```

The new declaration **%type** sets the type for non-terminals which otherwise need no declaration. You can also put bracketed types in **%left**, **%right**, or **%nonassoc**. In action code, yacc automatically qualifies symbol value references with the appropriate field name, e.g., if the third symbol is a **NUMBER**, a reference to **$3** acts like **$3.dval**.

The new, expanded parser was shown in Example 3-2. We've added a new start symbol **statement_list** so that the parser can accept a list of statements, each ended by a newline, rather than just one statement. We've also added an action for the rule that sets a variable, and a new rule at the end that turns a **NAME** into an **expression** by fetching the value of the variable.

We have to modify the lexer a little (Example 3-3). The literal block in the lexer no longer declares **yylval**, since its declaration is now in *y.tab.h*. The lexer doesn't have any automatic way to associate types with tokens, so you have to put in explicit field references when you set **yylval**. We've used the real number pattern from Chapter 2 to match floating point numbers. The action code uses **atof()** to read the number, then assigns the value to **yylval.dval**, since the parser expects the number's value in the **dval** field. For variables, we return the index of the variable in the variable table in **yylval.vblno**. Finally, we've made "\n" a regular token, so we use a dollar sign to indicate the end of the input.

A little experimentation shows that our modified calculator works:

```
% ch3-03
2/3
= 0.666667
a = 2/7
a
= 0.285714
z = a+1
z
= 1.28571
a/z
= 0.222222
$
```

Symbol Tables

Few users will be satisfied with single character variable names, so now we add the ability to use longer variable names. This means we need a *symbol table*, a structure that keeps track of the names in use. Each time the lexer reads a name from the input, it looks the name up in the symbol table, and gets a pointer to the corresponding symbol table entry. Elsewhere in the program, we use symbol table pointers rather than name strings, since pointers are much easier and faster to use than looking up a name each time we need it.

Since the symbol table requires a data structure shared between the lexer and parser, we created a header file *ch3hdr.h* (see Example 3-4). This symbol table is an array of structures each containing the name of the variable and its value. We also declare a routine **symlook()** which takes a name as a text string and returns a pointer to the appropriate symbol table entry, adding it if it is not already there.

Example 3-4: Header for parser with symbol table ch3hdr.h

```
#define NSYMS 20  /* maximum number of symbols */

struct symtab {
      char *name;
      double value;
} symtab[NSYMS];

struct symtab *symlook();
```

The parser changes only slightly to use the symbol table, as shown in Example 3-5. The value for a NAME token is now a pointer into the symbol table rather than an index as before. We change the %union and call the pointer field **symp**. The %token declaration for NAME changes appropriately, and the actions that assign to and read variables now use the token value as a pointer so they can read or write the value field of the symbol table entry.

The new routine **symlook()** is defined in the user subroutines section of the yacc specification, as shown in Example 3-6. (There is no compelling reason for this; it could as easily have been in the lex file or in a file by itself.) It searches through the symbol table sequentially to find the entry corresponding to the name passed as its argument. If an entry has a **name** string and it matches the one that **symlook()** is searching for, it returns a pointer to the entry, since the name has already been put into the table. If the **name** field is empty, we've looked at all of the table entries that are in use, and haven't found this symbol, so we enter the name into the heretofore empty table entry.

We use **strdup()** to make a permanent copy of the name string. When the lexer calls **symlook()**, it passes the name in the token buffer **yytext**. Since each subsequent token overwrites **yytext**, we need to make a copy ourselves here. (This is a common source of errors in lex scanners; if you need to use the contents of **yytext** after the scanner goes on to the next token, always make a copy.) Finally, if the current table entry is in use but doesn't match, **symlook()** goes on to search the next entry.

This symbol table routine is perfectly adequate for this simple example, but more realistic symbol table code is somewhat more complex. Sequential search is too slow for symbol tables of appreciable size, so use hashing or some other faster search function. Real symbol tables tend to carry considerably more information per entry, e.g., the type of a variable, whether it is a simple variable, an array or structure, and how many dimensions if it is an array.

Example 3-5: Rules for parser with symbol table ch3-04.y

```
%{
#include "ch3hdr.h"
#include <string.h>
%}

%union {
      double dval;
      struct symtab *symp;
}
%token <symp> NAME
%token <dval> NUMBER
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%type <dval> expression
%%
statement_list:  statement '\n'
      |     statement_list statement '\n'
      ;

statement: NAME '=' expression     { $1->value = $3; }
      |     expression             { printf("= %g\n", $1); }
      ;

expression: expression '+' expression { $$ = $1 + $3; }
      |     expression '-' expression { $$ = $1 - $3; }
      |     expression '*' expression { $$ = $1 * $3; }
      |     expression '/' expression
                  {     if($3 == 0.0)
                              yyerror("divide by zero");
                        else
                              $$ = $1 / $3;
                  }
      |     '-' expression %prec UMINUS   { $$ = -$2; }
      |     '(' expression ')'      { $$ = $2; }
      |     NUMBER
      |     NAME                    { $$ = $1->value; }
      ;
%%
```

Example 3-6: Symbol table routine ch3-04.pgm

```
/* look up a symbol table entry, add if not present */
struct symtab *
symlook(s)
char *s;
{
      char *p;
      struct symtab *sp;
```

Example 3-6: Symbol table routine ch3-04.pgm (continued)

```
    for(sp = symtab; sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if(sp->name && !strcmp(sp->name, s))
            return sp;

        /* is it free */
        if(!sp->name) {
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1);    /* cannot continue */
} /* symlook */
```

The lexer also changes only slightly to accommodate the symbol table (Example 3-7). Rather than declaring the symbol table directly, it now also includes *ch3hdr.h*. The rule that recognizes variable names now matches "[A-Za-z][A-Za-z0-9]*", any string of letters and digits starting with a letter. Its action calls **symlook()** to get a pointer to the symbol table entry, and stores that in **yylval.symp**, the token's value.

Example 3-7: Lexer with symbol table ch3-04.l

```
%{
#include "y.tab.h"
#include "ch3hdr.h"
#include <math.h>
%}
%%
([0-9]+|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?) {
        yylval.dval = atof(yytext);
        return NUMBER;
    }
[ \t] ;         /* ignore whitespace */

[A-Za-z][A-Za-z0-9]*    {    /* return symbol pointer */
        yylval.symp = symlook(yytext);
        return NAME;
    }
"$"   { return 0; }
\n    |
.     return yytext[0];
%%
```


There is one minor way in which our symbol table routine is better than those in most programming languages: since we allocate string space dynamically, there is no fixed limit on the length of variable names:*

```
% ch3-04
foo = 12
foo /5
= 2.4
thisisanextremelylongvariablenamewhichnobodywouldwanttotype = 42
3 * thisisanextremelylongvariablenamewhichnobodywouldwanttotype
= 126
$
%
```

Functions and Reserved Words

The next addition we make to the calculator adds mathematical functions for square root, exponential, and logarithm. We want to handle input like this:

```
s2 = sqrt(2)
s2
= 1.41421
s2*s2
= 2
```

The brute force approach makes the function names separate tokens, and adds separate rules for each function:

```
%token SQRT LOG EXP
 . . .
%%
expression: . . .
      |     SQRT '(' expression ')'{ $$ = sqrt($3); }
      |     LOG '(' expression ')' { $$ = log($3); }
      |     EXP '(' expression ')' { $$ = exp($3); }
```

In the scanner, we have to return a SQRT token for "sqrt" input and so forth:

```
sqrt  return SQRT;
log   return LOG;
exp   return EXP;

[A-Za-z][A-Za-z0-9]*    {    . . .
```

*Actually, there is a limit due to the maximum token size that lex can handle, but you can make that rather large. See "yytext" in Chapter 6, *A Reference for Lex Specifications*.