Chapter 3. Using Bison

The previous chapter concentrated on flex alone. In this chapter we turn our attention to bison, although we use flex to generate our lexical analyzers. Where flex recognizes regular expressions, bison recognizes entire grammars. Flex divides the input stream into pieces (*tokens*), and then bison takes these pieces and groups them together logically. In this chapter we'll finish the desk calculator we started in Chapter 1, starting with simple arithmetic and then adding built-in functions, user variables, and finally user-defined functions.

How a Bison Parser Matches Its Input

Bison takes a grammar that you specify and writes a parser that recognizes valid "sentences" in that grammar. We use the term *sentence* here in a fairly general way—for a C language grammar, the sentences are syntactically valid C programs. Programs can be syntactically valid but semantically invalid, for example, a C program that assigns a string to an int variable. Bison handles only the syntax; other validation is up to you. As we saw in Chapter 1, a grammar is a series of rules that the parser uses to recognize syntactically valid input. For example, here is a version of the grammar we'll use later in this chapter in a calculator:

```
statement:   NAME '=' expression

expression:  NUMBER '+' NUMBER
           | NUMBER '−' NUMBER
```

The vertical bar, `|`, means there are two possibilities for the same symbol; that is, an expression can be either an addition or a subtraction. The symbol to the left of the : is known as the *left-hand side* of the rule, often abbreviated LHS, and the symbols to the right are the *right-hand side*, usually abbreviated RHS. Several rules may have the same left-hand side; the vertical bar is just shorthand for this. Symbols that actually appear in the input and are returned by the lexer are terminal symbols or tokens, while those that appear on the left-hand side of each rule are nonterminal symbols or nonterminals. Terminal and nonterminal symbols must be different; it is an error to write a rule with a token on the left side.

The usual way to represent a parsed sentence is as a tree. For example, if we parsed the input `fred = 12 + 13` with this grammar, the tree would look like Figure 3-1.

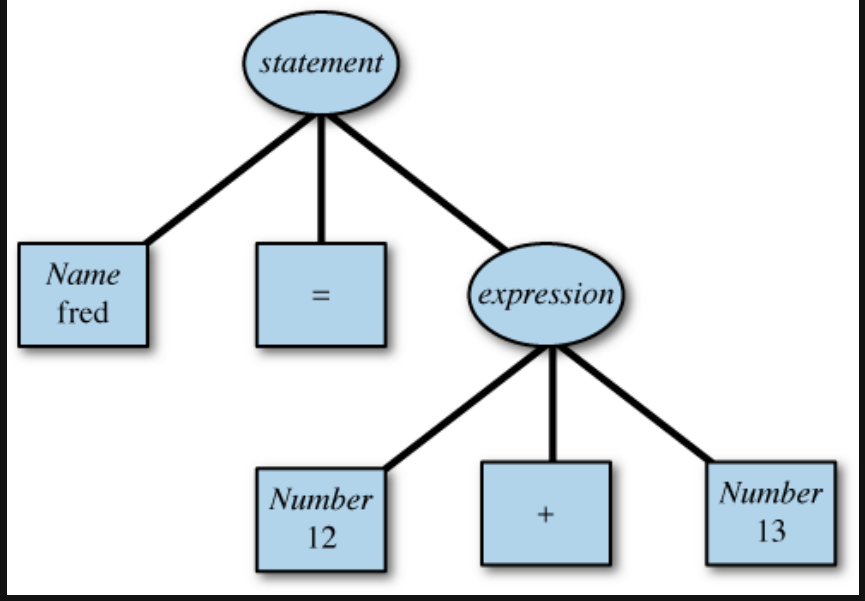

Figure 3-1. Expression parse tree

In this example, `12 + 13` is an expression, and `fred = expression` is a statement. A bison parser doesn't automatically create this tree as a data structure, although as we will see, it is not hard to do so yourself. Every grammar includes a start symbol, the one that has to be at the root of the parse tree. In this grammar, `statement` is the start symbol. Rules can refer directly or indirectly to themselves; this important ability makes it possible to parse arbitrarily long input sequences. Let's extend our grammar to handle longer arithmetic expressions:

```
expression: NUMBER
          | expression '+' NUMBER
          | expression '−' NUMBER
```

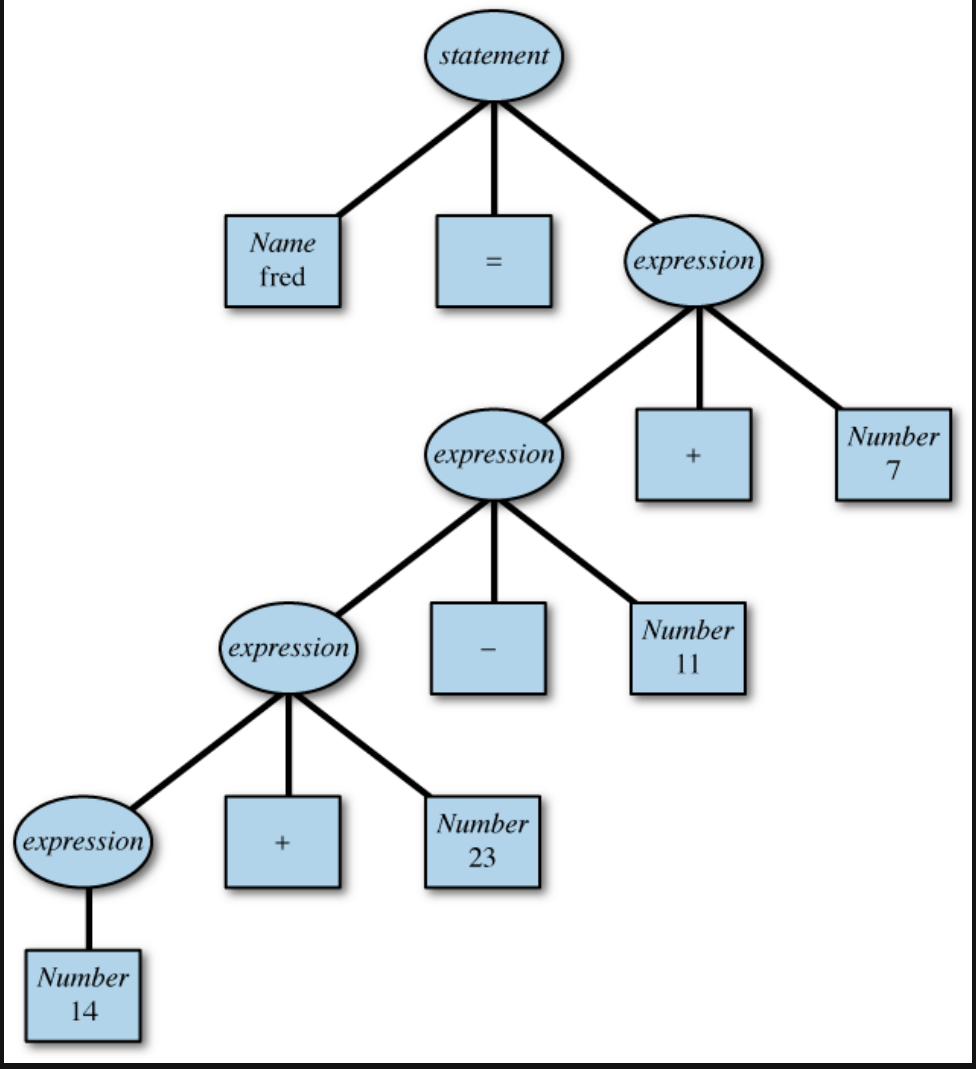

Figure 3-2. Parse tree with recursive rules

Now we can parse a sequence like `fred = 14 + 23 – 11 + 7` by applying the expression rules repeatedly, as in Figure 3-2. Bison can parse recursive rules very efficiently, so we will see recursive rules in nearly every grammar we use.

Shift/Reduce Parsing

A bison parser works by looking for rules that might match the tokens seen so far. When bison processes a parser, it creates a set of states, each of which reflects a possible position in one or more partially parsed rules. As the parser reads tokens, each time it reads a token that doesn't complete a rule, it pushes the token on an internal stack and switches to a new state reflecting the token it just read. This action is called a *shift*. When it has found all the symbols that constitute the right-hand side of a rule, it pops the right-hand side symbols off the stack, pushes the left-hand side symbol onto the stack, and switches to a new state reflecting the new symbol on the stack. This action is called a *reduction*, since it usually reduces the number of items on the stack.[10] Whenever bison reduces a rule, it executes user code associated with the rule. This is how you actually do something with the material that the parser parses. Let's look how it parses the input `fred = 12 + 13` using the simple rules in

Figure 3-1. The parser starts by shifting tokens on to the internal stack one at a time:

```
fred
fred =
fred = 12
fred = 12 +
fred = 12 + 13
```

At this point it can reduce the rule `expression: NUMBER + NUMBER`, so it pops the 12, the plus, and the 13 from the stack and replaces them with *`expression`*:

```
fred = expression
```

Now it reduces the rule `statement: NAME = expression`, so it pops `fred`, `=`, and *`expression`* and replaces them with *`statement`*. We've reached the end of the input and the stack has been reduced to the start symbol, so the input was valid according to the grammar.

What Bison's LALR(1) Parser Cannot Parse

Bison parsers can use either of two parsing methods, known as LALR(1) (Look Ahead Left to Right with a one-token lookahead) and GLR (Generalized Left to Right). Most parsers use LALR(1), which is less powerful but considerably faster and easier to use than GLR. In this chapter we'll describe LALR parsing and save GLR for Chapter 9.

Although LALR parsing is quite powerful, you can write grammars that it cannot handle. It cannot deal with ambiguous grammars, ones in which the same input can match more than one parse tree. It also cannot deal with grammars that need more than one token of lookahead to tell whether it has matched a rule. (But bison has a wonderful hack to deal with the most common kind of ambiguous grammars, which we'll come to in a few pages.) Consider this extremely contrived example:

```
phrase:  cart_animal AND CART
       | work_animal AND PLOW

cart_animal: HORSE | GOAT

work_animal: HORSE | OX
```

This grammar isn't ambiguous, since there is only one possible parse tree for any valid input, but bison can't handle it because it requires two symbols of lookahead. In particular, in the input `HORSE AND CART` it cannot tell whether `HORSE` is a `cart_animal`, or a `work_animal` until it sees

CART, and bison cannot look that far ahead. If we changed the first rule to this:

```
phrase: cart_animal CART
      | work_animal PLOW
```

bison would have no trouble, since it can look one token ahead to see whether an input of HORSE is followed by CART, in which case the horse is a cart_animal, or by PLOW, in which case it is a work_animal. In practice, the rules about what bison can handle are not as complex and confusing as they may seem here. One reason is that bison knows exactly what grammars it can parse and what it cannot. If you give it one that it cannot handle, it will tell you, so there is no problem of overcomplex parsers silently failing. Another reason is that the grammars that bison can handle correspond pretty well to ones that people really write. As often as not, a grammatical construct that confuses bison will confuse people as well, so if you have some latitude in your language design, you should consider changing the language to make it more understandable both to bison and to its users. For more information on shift/reduce parsing, see Chapter 7. For a discussion of what bison has to do to turn your specification into a working C program, a good reference is Dick Grune's Parsing Techniques: A Practical Guide. He offers a download of an old but quite adequate edition at http://www.cs.vu.nl/~dick/PTAPG.html.

A Bison Parser

A bison specification has the same three-part structure as a flex specification. (Flex copied its structure from the earlier lex, which copied its structure from yacc, the predecessor of bison.) The first section, the definition section, handles control information for 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.

Bison creates the C program by plugging pieces into a standard skeleton file. The rules are compiled into arrays that represent the state machine that matches the input tokens. The actions have the $N and @N values translated into C and then are put into a switch statement within yyparse() that runs the appropriate action each time there's a reduction. Some bits of the skeleton have multiple versions from which bison chooses depending on what options are in use; for example, if the parser uses the locations feature, it includes code to handle location data.

In this chapter we take the simple calculator example from Chapter 1 and extend it significantly. First, we rewrite it to take advantage of some handy bison shortcuts and change it to produce a reusable data structure

rather than computing the values on the fly. Later, we'll add more complex syntax for loops and functions and show how to implement them in a simple interpreter.[11]

Abstract Syntax Trees

One of the most powerful data structures used in compilers is an *abstract syntax tree* (AST). In Chapter 1 we saw a parse tree, a tree that has a node for every rule used to parse the input string. In most real grammars, there are rules that exist to manage grouping but that add no meaning to the program. In the calculator example, the rules `exp: term` and `term: factor` exist only to tell the parser the relative precedence of the operators. An AST is basically a parse tree that omits the nodes for the uninteresting rules.

Once a parser creates an AST, it's straightforward to write recursive routines that "walk" the tree. We'll see several tree walkers in this example.

An Improved Calculator That Creates ASTs

This example is big enough to be worth dividing into several source files, so we'll put most of the C code into a separate file, which means we also need a C header file to declare the routines and data structures used in the various files (Example 3-1).

Example 3-1. Calculator that builds an AST: header fb3-1.h

```
/*
 * Declarations for a calculator fb3-1
 */

/* interface to the lexer */
extern int yylineno; /* from lexer */
void yyerror(char *s, ...);

/* nodes in the abstract syntax tree */
struct ast {
  int nodetype;
  struct ast *l;
  struct ast *r;
};

struct numval {
  int nodetype;   /* type K for constant */
  double number;
};
```

```
/* build an AST */
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newnum(double d);

/* evaluate an AST */
double eval(struct ast *);

/* delete and free an AST */
void treefree(struct ast *);
```

The variable `yylineno` and routine `yyerror` are familiar from the flex example. Our `yyerror` is slightly enhanced to take multiple arguments in the style of `printf`.

The AST consists of nodes, each of which has a node type. Different nodes have different fields, but for now we have just two kinds, one that has pointers to up to two subnodes and one that contains a number. Two routines, `newast` and `newnum`, create AST nodes; `eval` walks an AST and returns the value of the expression it represents; and `treefree` walks an AST and deletes all of its nodes.

Example 3-2. Bison parser for AST calculator

```
/* calculator with AST */

%{
#  include <stdio.h>
#  include <stdlib.h>
#  include "fb3-1.h"
%}

%union {
  struct ast *a;
  double d;
}

/* declare tokens */
%token <d> NUMBER
%token EOL

%type <a> exp factor term
```

Example 3-2 shows the bison parser for the AST calculator. The first section of the parser uses the `%union` construct to declare types to be used in the values of symbols in the parser. In a bison parser, every symbol, both tokens and nonterminals, can have a value associated with it. By default, the values are all integers, but useful programs generally need more sophisticated values. The `%union` construct, as its name suggests, is used to create a C language `union` declaration for symbol values. In this

case, the union has two members; `a`, which is a pointer to an AST, and `d`, which is a double precision number.

Once the union is defined, we need to tell bison what symbols have what types of values by putting the appropriate name from the union in angle brackets (`< >`). The token `NUMBER`, which represents numbers in the input, has the value `<d>` to hold the value of the number. The new declaration `%type` assigns the value `<a>` to `exp`, `factor`, and `term`, which we'll use as we build up our AST.

You don't have to declare a type for a token or declare a nonterminal at all if you don't use the symbol's value. If there is a `%union` in the declarations, bison will give you an error if you attempt to use the value of a symbol that doesn't have an assigned type. Keep in mind that any rule without explicit action code gets the default action `$$ = $1;`, and bison will complain if the LHS symbol has a type and the RHS symbol doesn't have the same type.

```
%%
calclist: /* nothing */
| calclist exp EOL {
     printf("= %4.4g\n", eval($2));    evaluate and print the AST
     treefree($2);                     free up the AST
     printf("> ");
 }

 | calclist EOL { printf("> "); } /* blank line or a comment */
 ;

exp: factor
 | exp '+' factor { $$ = newast('+', $1,$3); }
 | exp '-' factor { $$ = newast('-', $1,$3);}
 ;

factor: term
 | factor '*' term { $$ = newast('*', $1,$3); }
 | factor '/' term { $$ = newast('/', $1,$3); }
 ;

term: NUMBER { $$ = newnum($1); }
 | '|' term { $$ = newast('|', $2, NULL); }
 | '(' exp ')' { $$ = $2; }
 | '-' term { $$ = newast('M', $2, NULL); }
 ;
%%
```

Literal Character Tokens

The rules section has two significant changes from the version in Chapter 1. One is that the rules now use literal tokens for the operators.

Rather than giving every token a name, it's also possible to use a single quoted character as a token, with the ASCII value of the token being the token number. (Bison starts the numbers for named tokens at 258, so there's no problem of collisions.) By convention, literal character tokens are used to represent input tokens consisting of the same character; for example, the token `'+'` represents the input token `+`, so in practice they are used only for punctuation and operators. There's no need to declare literal character tokens unless you need to declare the type of their values.

The other change to the parser is that it creates an AST for each expression rather than evaluating it on the fly. In this version of the calculator, we create an AST for each expression and then evaluate the AST, print the result, and free the AST. We call `newast()` to create each node in the AST. Each node has an operator type, which is generally the same as the token name. Notice that unary minus creates a node of type `M` to distinguish it from binary subtraction.

Example 3-3. Lexer for AST calculator

```
/* recognize tokens for the calculator */
%option noyywrap nodefault yylineno
%{
# include "fb3-1.h"
# include "fb3-1.tab.h"
%}

/* float exponent */
EXP     ([Ee][-+]?[0-9]+)

%%
"+" |
"-" |
"*" |
"/" |
"|" |
"(" |
")"     { return yytext[0]; }
[0-9]+"."[0-9]*{EXP}? |
"."?[0-9]+{EXP}? { yylval.d = atof(yytext); return NUMBER; }

\n      { return EOL; }
"//".*
[ \t]   { /* ignore whitespace */ }
.       { yyerror("Mystery character %c\n", *yytext); }
%%
```

The lexer, shown in Example 3-3, is a little simpler than the version in Chapter 1. We use a common idiom for the single-character operators, handling them all with the same rule that returns `yytext[0]`, the char-

acter itself, as the token. We do still have names for `NUMBER` and `EOL`. We also handle floating-point numbers, using a version of the pattern from Chapter 2, and change the internal representation of numbers to double. Since `yylval` is now a union, the double value has to be assigned to `yylval.d`. (Flex does not automate the management of token values as bison does.)

Example 3-4. C routines for AST calculator

```
#  include <stdio.h>
#  include <stdlib.h>
#  include <stdarg.h>
#  include "fb3-1.h"

struct ast *
newast(int nodetype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->l = l;
  a->r = r;
  return a;
}

struct ast *
newnum(double d)
{
  struct numval *a = malloc(sizeof(struct numval));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'K';
  a->number = d;
  return (struct ast *)a;
}
```

Finally, there is a file of routines called from the parser, shown in Example 3-4.[12] First come the two routines to create AST nodes, by `malloc`-ing a node and filling it in. All AST nodes have a `nodetype` as the first field, so a tree walk can tell what kind of nodes it's walking through.

```
double
eval(struct ast *a)
```

```
{
  double v;   calculated value of this subtree

  switch(a->nodetype) {
  case 'K': v = ((struct numval *)a)->number; break;

  case '+': v = eval(a->l) + eval(a->r); break;
  case '-': v = eval(a->l) - eval(a->r); break;
  case '*': v = eval(a->l) * eval(a->r); break;
  case '/': v = eval(a->l) / eval(a->r); break;
  case '|': v = eval(a->l); if(v < 0) v = -v; break;
  case 'M': v = -eval(a->l); break;
  default: printf("internal error: bad node %c\n", a->nodetype);
  }
  return v;
}

void
treefree(struct ast *a)
{
  switch(a->nodetype) {

    /* two subtrees */
  case '+':
  case '-':
  case '*':
  case '/':
    treefree(a->r);

    /* one subtree */
  case '|':
  case 'M':
    treefree(a->l);

    /* no subtree */
  case 'K':
    free(a);
    break;

  default: printf("internal error: free bad node %c\n", a->nodetype);
  }
}
```

Next we have the two tree-walking routines. They each make what's known as a *depth-first* traversal of the tree, recursively visiting the subtrees of each node and then the node itself. The `eval` routine returns the value of the tree or subtree from each call, and the `treefree` doesn't have to return anything.

```
void
yyerror(char *s, ...)
{
```

```
  va_list ap;
  va_start(ap, s);

  fprintf(stderr, "%d: error: ", yylineno);
  vfprintf(stderr, s, ap);
  fprintf(stderr, "\n");
}

int
main()
{
  printf("> ");
  return yyparse();
}
```

Finally, we have `yyerror` and `main`. This version of `yyerror` uses varargs to accept a `printf`-style argument list, which turns out to be convenient when generating error messages.

Building the AST Calculator

This program now has three source files and a header file, so of course we use `make` to build it.

```
fb3-1:  fb3-1.l fb3-1.y fb3-1.h
        bison -d fb3-1.y
        flex -ofb3-1.lex.c fb3-1.l
        cc -o $@ fb3-1.tab.c fb3-1.lex.c fb3-1funcs.c
```

Notice the `-o` flag to `flex`. Bison automatically names its generated C file to match the `.y` file, but flex always calls its C file `lex.yy.c` unless you tell it otherwise.

Shift/Reduce Conflicts and Operator Precedence

The expression parser uses three different symbols, `exp`, `factor`, and `term`, to set the precedence and associativity of operators. Although this parser is still reasonably legible, as grammars add more operators with more levels of precedence, they become increasingly hard to read and maintain. Bison provides a clever way to describe the precedence separately from the rules in the grammar, which makes the grammar and parser smaller and easier to maintain. First, we'll just make all expressions use `exp` symbols:

```
%type <a> exp
```

```
%%
 ...
exp: exp '+' exp { $$ = newast('+', $1,$3); }
   | exp '-' exp { $$ = newast('-', $1,$3);}
   | exp '*' exp { $$ = newast('*', $1,$3); }
   | exp '/' exp { $$ = newast('/', $1,$3); }
   | '|' exp     { $$ = newast('|', $2, NULL); }
   | '(' exp ')' { $$ = $2; }
   | '-' exp     { $$ = newast('M', $2, NULL); }
   | NUMBER      { $$ = newnum($1); }
 ;
%%
```

But this grammar has a problem: It is extremely ambiguous. For example, the input `2+3*4` might mean `(2+3)*4` or `2+(3*4)`, and the input `3-4-5-6` might mean `3-(4-(5-6))` or `(3-4)-(5-6)` or any of a lot of other possibilities. Figure 3-3 shows the two possible parses for `2+3*4`.

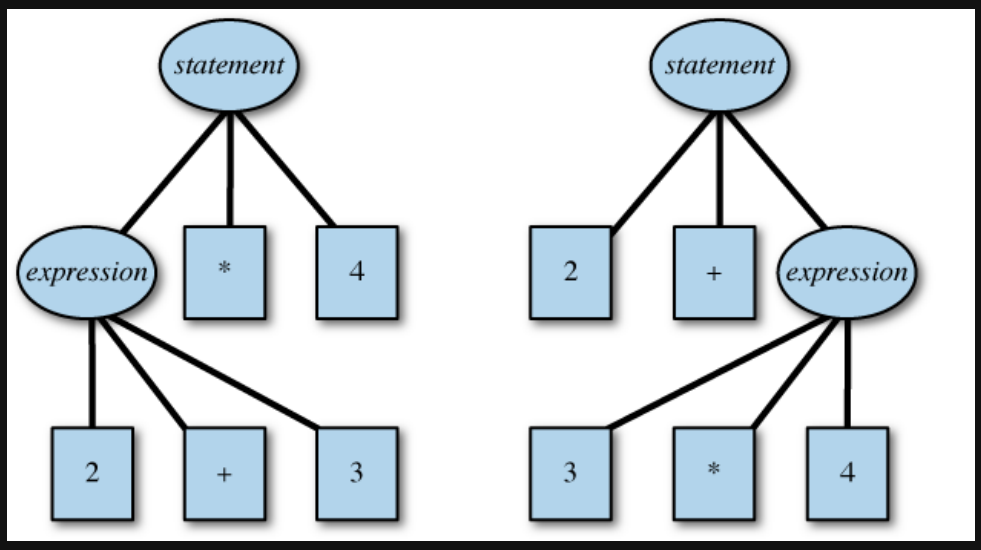

Figure 3-3. Two expression parse trees

If you compile this grammar as it stands, bison will tell you that there are 24 *shift/reduce conflicts*, which are states where it cannot tell whether it should shift the token on the stack or reduce a rule first. For example, when parsing `2+3*4`, the parser goes through these steps (we abbreviate `exp` as `E` here):

```
2          shift NUMBER
E          reduce E → NUMBER
E +        shift +
E + 3      shift NUMBER
E + E      reduce E → NUMBER
```

At this point, the parser looks at the `*` and could either reduce `2+3` using:

```
exp:       exp '+' exp
```

to an expression or shift the `*`, expecting to be able to reduce:

```
     exp:       exp '*' exp
```

later on.

The problem is that we haven't told bison about the precedence and associativity of the operators. *Precedence* controls which operators execute first in an expression. Mathematical and programming tradition (dating back past the first Fortran compiler in 1956) says that multiplication and division take precedence over addition and subtraction, so `a+b*c` means `a+(b*c)`, and `d/e-f` means `(d/e)-f`. In any expression grammar, operators are grouped into levels of precedence from lowest to highest. The total number of levels depends on the language. The C language is notorious for having too many precedence levels, a total of 15 levels.

Associativity controls the grouping of operators at the same precedence level. Operators may group to the left, for example, `a-b-c` in C means `(a-b)-c`, or to the right, for example, `a=b=c` in C means `a=(b=c)`. In some cases, operators do not group at all; for example, in Fortran `A.LE.B.LE.C` is invalid.

There are two ways to specify precedence and associativity in a grammar, implicitly and explicitly. So far, we've specified them implicitly, by using separate nonterminal symbols for each precedence level. This is a perfectly reasonable way to write a grammar, and if bison didn't have explicit precedence rules, it would be the only way.

But bison also lets you specify precedence explicitly. We can add these lines to the declaration section to tell it how to resolve the conflicts:

```
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS

%type <a> exp

%%
 ...
exp: exp '+' exp { $$ = newast('+', $1,$3); }
   | exp '-' exp { $$ = newast('-', $1,$3);}
   | exp '*' exp { $$ = newast('*', $1,$3); }
   | exp '/' exp { $$ = newast('/', $1,$3); }
   | '|' exp     { $$ = newast('|', $2, NULL); }
   | '(' exp ')' { $$ = $2; }
   | '-' exp %prec UMINUS{ $$ = newast('M', NULL, $2); }
   | NUMBER      { $$ = newnum($1); }
  ;
```

Each of these declarations defines a level of precedence, with the order of the `%left`, `%right`, and `%nonassoc` declarations defining the order of precedence from lowest to highest. They tell bison that `+` and `-` are left associative and at the lowest precedence level; `*` and `/` are left associative and at a higher precedence level; and `|` and `UMINUS`, a pseudotoken standing for unary minus, have no associativity and are at the highest precedence. (We don't have any right-associative operators here, but if we did, they'd use `%right`.) Bison assigns each rule the precedence of the rightmost token on the right-hand side; if that token has no precedence assigned, the rule has no precedence of its own. When bison encounters a shift/reduce conflict, it consults the table of precedence, and if all the rules involved in the conflict have a precedence assigned, it uses precedence to resolve the conflict.

In our grammar, all of the conflicts occur in the rules of the form `exp OP exp`, so setting precedence for the four operators allows it to resolve all of the conflicts. This parser using precedence is slightly smaller and faster than the one with the extra rules for implicit precedence, since it has fewer rules to reduce.

The rule for negation includes `%prec UMINUS`. The only operator in this rule is `-`, which has low precedence, but we want unary minus to have higher precedence than multiplication rather than lower. The `%prec` tells bison to use the precedence of `UMINUS` for this rule.

When Not to Use Precedence Rules

You can use precedence rules to fix any shift/reduce conflict that occurs in the grammar. This is usually a terrible idea. In expression grammars, the cause of the conflicts is easy to understand, and the effect of the precedence rules is clear. In other situations, precedence rules fix shift/reduce problems, but it can be extremely difficult to understand just what effect they have on the grammar.

Use precedence in only two situations: in expression grammars and to resolve the "dangling else" conflict in grammars for if/then/else language constructs. (See the section IF/THEN/ELSE for examples of the latter.) Otherwise, if you can, you should fix the grammar to remove the conflict. Remember that conflicts mean that bison can't create a parser for a grammar, probably because it's ambiguous. This means there are multiple possible parses for the same input and the parser that bison created chose one of them. Except in the two previous cases, this usually points to a problem in your language definition. In some cases, if a grammar is ambiguous to bison, it's almost certainly ambiguous to humans, too. See Chapter 8 for more information on finding and repairing conflicts, as well as the advanced bison features that let you use ambiguous grammars if you really want to do so.

An Advanced Calculator

The final example in this chapter extends the calculator to make it a small but somewhat realistic compiler. We'll add named variables and assignments; comparison expressions (greater, less, equal, etc.); flow control with if/then/else and while/do; built-in and user-defined functions; and a little error recovery. The previous version of the calculator didn't take much advantage of the AST representation of expressions, but in this one, the AST is the key to the implementation of flow control and user functions. Here's an example of defining a user function, and then calling it, using a built-in function as one of the arguments:

```
> let avg(a,b) = (a+b)/2;
Defined avg
> avg(3, sqrt(25))
=    4
```

As before, we start with the declarations, shown in Example 3-5.

Example 3-5. Advanced calculator header fb3-2.h

```
/*
 * Declarations for a calculator fb3-1
 */

/* interface to the lexer */
extern int yylineno; /* from lexer */
void yyerror(char *s, ...);

/* symbol table */
struct symbol {         /* a variable name */
  char *name;
  double value;
  struct ast *func;     /* stmt for the function */
  struct symlist *syms; /* list of dummy args */
};

/* simple symtab of fixed size */
#define NHASH 9997
struct symbol symtab[NHASH];

struct symbol *lookup(char*);

/* list of symbols, for an argument list */
struct symlist {
  struct symbol *sym;
  struct symlist *next;
};
```

```
struct symlist *newsymlist(struct symbol *sym, struct symlist *next);
void symlistfree(struct symlist *sl);
```

The symbol table is adapted from the one used in the previous chapter. In the calculator, each symbol can potentially be both a variable and a user-defined function. The `value` field holds the symbol's value as a variable, the `func` field points to the AST for the user code for the function, and `syms` points to a linked list of the dummy arguments, which are themselves symbols. (In the previous example, `avg` is the function, and `a` and `b` are the dummy arguments.) The C functions `newsymlist` and `symlistfree` create and free them.

```
/* node types
 *  + - * / |
 *  0-7 comparison ops, bit coded 04 equal, 02 less, 01 greater
 *  M unary minus
 *  L expression or statement list
 *  I IF statement
 *  W WHILE statement
 *  N symbol ref
 *  = assignment
 *  S list of symbols
 *  F built in function call
 *  C user function call
 */

enum bifs {                     /* built-in functions */
  B_sqrt = 1,
  B_exp,
  B_log,
  B_print
};

/* nodes in the abstract syntax tree */
/* all have common initial nodetype */

struct ast {
  int nodetype;
  struct ast *l;
  struct ast *r;
};

struct fncall {                 /* built-in function */
  int nodetype;                 /* type F */
  struct ast *l;
  enum bifs functype;
};

struct ufncall {                /* user function */
  int nodetype;                 /* type C */
  struct ast *l;                /* list of arguments */
  struct symbol *s;
```

```
};

struct flow {
  int nodetype;                 /* type I or W */
  struct ast *cond;             /* condition */
  struct ast *tl;               /* then branch or do list */
  struct ast *el;               /* optional else branch */
};

struct numval {
  int nodetype;                 /* type K */
  double number;
};

struct symref {
  int nodetype;                 /* type N */
  struct symbol *s;
};

struct symasgn {
  int nodetype;                 /* type = */
  struct symbol *s;
  struct ast *v;                /* value */
};

/* build an AST */
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newcmp(int cmptype, struct ast *l, struct ast *r);
struct ast *newfunc(int functype, struct ast *l);
struct ast *newcall(struct symbol *s, struct ast *l);
struct ast *newref(struct symbol *s);
struct ast *newasgn(struct symbol *s, struct ast *v);
struct ast *newnum(double d);
struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast

/* define a function */
void dodef(struct symbol *name, struct symlist *syms, struct ast *stmts);

/* evaluate an AST */
double eval(struct ast *);

/* delete and free an AST */
void treefree(struct ast *);
```

This version has considerably more kinds of nodes in the AST. As before, each kind of node starts with a `nodetype` that the tree-walking code can use to tell what kind of node it is.[13] The basic `ast` node is also used for comparisons, with each kind of comparison (less, less-equal, equal, etc.) being a different type, and for lists of expressions.

Built-in functions have a `fncall` node with the AST of the argument (the built-ins each take one argument) and an enum that says which built-in

function it is. There are three standard functions, `sqrt`, `exp`, and `log`, as well as `print`, a function that prints its argument and returns the argument as its value. Calls to user functions have a `ufncall` node with a pointer to the function, which is an entry in the symbol table, and an AST, which is a list of the arguments.

Flow control expressions if/then/else and while/do use a `flow` node with the control expression, the then branch or do list, and the optional else branch.

Constants are `numval` as before; references to symbols are `symref` with a pointer to the symbol in the symbol table; and assignments are `symasgn` with a pointer to the symbol to be assigned and the AST of the value to assign to it.

Every AST has a value. The value of an if/then/else is the value of the branch taken; the value of while/do is the last value of the do list; and the value of a list of expressions is the last expression.[14] Finally, we have C procedures to create each kind of AST node and a procedure to create a user-defined function.

Advanced Calculator Parser

Example 3-6 shows the parser for the advanced AST calculator.

Example 3-6. Advanced calculator parser fb3-2.y

```
/* calculator with AST */

%{
#  include <stdio.h>
#  include <stdlib.h>
#  include "fb3-2.h"
%}

%union {
  struct ast *a;
  double d;
  struct symbol *s;             /* which symbol */
  struct symlist *sl;
  int fn;                       /* which function */
}

/* declare tokens */
%token <d> NUMBER
%token <s> NAME
%token <fn> FUNC
%token EOL

%token IF THEN ELSE WHILE DO LET
```

```
%nonassoc <fn> CMP
%right '='
%left '+' '-'
%left '*' '/'
%nonassoc '|' UMINUS

%type <a> exp stmt list explist
%type <sl> symlist

%start calclist
%%
```

The `%union` here defines many kinds of symbol values, which is typical in realistic bison parsers. As well as a pointer to an AST and a numeric value, a value can be a pointer to the symbol table for a user symbol, a list of symbols, or a subtype of a comparison or function token. (We use the word *symbol* somewhat confusingly here, both for names used in the bison grammar and for names that the user types into the compiled program. We'll say *user symbol* for the latter when the context isn't otherwise clear.)

There's a new token `FUNC` for the built-in functions, with the value indicating which function, and six reserved words, `IF` through `LET`. The token `CMP` is any of the six comparison operators, with the value indicating which operator. (This trick of using one token for several syntactically similar operators helps keep down the size of the grammar.)

The list of precedence declarations starts with the new `CMP` and `=` operators.

A `%start` declaration identifies the top-level rule, so we don't have to put it at the beginning of the parser.

Calculator Statement Syntax

```
stmt: IF exp THEN list           { $$ = newflow('I', $2, $4, NULL); }
   | IF exp THEN list ELSE list  { $$ = newflow('I', $2, $4, $6); }
   | WHILE exp DO list           { $$ = newflow('W', $2, $4, NULL); }
   | exp
;

list: /* nothing */ { $$ = NULL; }
   | stmt ';' list { if ($3 == NULL)
                       $$ = $1;
                     else
                       $$ = newast('L', $1, $3);
                   }
   ;
```

Our grammar distinguishes between statements (`stmt`) and expressions (`exp`). A statement is either a flow of control (if/then/else or while/do) or an expression. The if and while statements take lists of statements, with each statement in the list being followed by a semicolon. Each rule that matches a statement calls a routine to build an appropriate AST node.

The design of the syntax here is largely arbitrary, and one of the nice things about using bison to build a parser is that it's easy to experiment with variations. The interplay among bits of the syntax can be quite subtle; if, for example, the definition of `list` had put semicolons between rather than after each statement, the grammar would be ambiguous unless the grammar also added closing `FI` and `ENDDO` tokens to indicate the end of if/then and while/do statements.

The definition of `list` is right recursive, that is, `stmt ; list` rather than `list stmt ;` . It doesn't make any difference to the language recognized, but it makes it easier to build the list of statements linked from head to tail rather than from tail to head. Each time the `stmt ; list` rule is reduced, it creates a link that adds the statement to the head of the list so far. If the rule were `list stmt ;` , the statement would need to go at the tail of the list, which would require either a more complex circularly linked list or else reversing the list at the end (as we did with the list of references in Chapter 1).

One disadvantage of right recursion rather than left is that right recursion puts up all of the yet-to-be-reduced statements on the parser stack and then reduces them all at the end of the list, while left recursion builds the list a statement at a time as the input is parsed. In a situation like this, where the list is unlikely to be more than a few items long, it doesn't matter, but in a language where the list might be a list of thousands of items, it's worth making the list with a left recursive rule and then reversing it to prevent parser stack overflow. Some programmers also find left recursion to be easier to debug, since it tends to produce output after each statement rather than all at once at the end.

Calculator Expression Syntax

```
exp: exp CMP exp          { $$ = newcmp($2, $1, $3); }
   | exp '+' exp          { $$ = newast('+', $1,$3); }
   | exp '-' exp          { $$ = newast('-', $1,$3);}
   | exp '*' exp          { $$ = newast('*', $1,$3); }
   | exp '/' exp          { $$ = newast('/', $1,$3); }
   | '|' exp              { $$ = newast('|', $2, NULL); }
   | '(' exp ')'          { $$ = $2; }
   | '-' exp %prec UMINUS { $$ = newast('M', $2, NULL); }
   | NUMBER               { $$ = newnum($1); }
   | NAME                 { $$ = newref($1); }
   | NAME '=' exp         { $$ = newasgn($1, $3); }
```

```
    | FUNC '(' explist ')' { $$ = newfunc($1, $3); }
    | NAME '(' explist ')' { $$ = newcall($1, $3); }
  ;

  explist: exp
   | exp ',' explist  { $$ = newast('L', $1, $3); }
  ;
  symlist: NAME       { $$ = newsymlist($1, NULL); }
   | NAME ',' symlist { $$ = newsymlist($1, $3); }
  ;
```

The expression syntax is a modestly expanded version of the expression syntax in the previous example. A new rule for `CMP` handles the six comparison operators, using the value of the `CMP` to tell which operator it was, and a rule for assignments creates an assignment node.

There are separate rules for built-in functions identified by a reserved name (`FUNC`) and user functions identified by a user symbol (`NAME`).

A rule for `explist` , a list of expressions, builds an AST of the expressions used for the actual arguments to a function call. A separate rule for `symlist` , a list of symbols, builds a linked list of symbols for the dummy arguments in a function definition. Both are right recursive to make it easier to build the list in the desired order.

Top-Level Calculator Grammar

```
  calclist: /* nothing */
    | calclist stmt EOL {
       printf("= %4.4g\n> ", eval($2));
       treefree($2);
      }
    | calclist LET NAME '(' symlist ')' '=' list EOL {
                         dodef($3, $5, $8);
                         printf("Defined %s\n> ", $3->name); }

    | calclist error EOL { yyerrok; printf("> "); }
   ;
```

The last bit of grammar is the top level, which recognizes a list of statements and function declarations. As before, the top level evaluates the AST for a statement, prints the result, and then frees the AST. A function definition is just saved for future use.

Basic Parser Error Recovery

The last rule in the parser provides a small amount of error recovery. Because of the way that bison parsers work, it's rarely worth the effort to try to correct errors, but it's at least possible to recover to a state where

the parser can continue. The special pseudo-token `error` indicates an error recovery point. When a bison parser encounters an error, it starts discarding symbols from the parser stack until it reaches a point where an `error` token would be valid; then it discards input tokens until it finds one it can shift in its current state, and then continues parsing from there. If the parse fails again, it discards more stack symbols and input tokens until either it can resume parsing or the stack is empty and the parse fails. To avoid a cascade of misleading error messages, the parser normally suppresses any parse error messages after the first one until it has successfully shifted three tokens in a row. The macro `yyerrok` in an action tells the parser that recovery is done, so subsequent error messages will be produced.

Although it's possible in principle to add lots of error rules to try to do lots of error recovery, in practice it's rare to have more than one or two error rules. The error token is almost always used to resynchronize at a punctuation character in a top-level recursive rule as we do here.

If the symbols discarded in error recovery from the stack have values that point to allocated storage, the error recovery process will leak storage, since the discarded values are never freed. In this example, we don't worry about it, but bison does provide a feature to tell the parser to call your code to free discarded values, described in Chapter 6.

The Advanced Calculator Lexer

Example 3-7. Advanced calculator lexer fb3-2.l

```
/* recognize tokens for the calculator */
%option noyywrap nodefault yylineno
%{
# include "fb3-2.h"
# include "fb3-2.tab.h"
%}

/* float exponent */
EXP     ([Ee][-+]?[0-9]+)

%%
 /* single character ops */
"+" |
"-" |
"*" |
"/" |
"=" |
"|" |
"," |
";" |
"(" |
")"     { return yytext[0]; }
```

```
 /* comparison ops, all are a CMP token */
">"     { yylval.fn = 1; return CMP; }
"<"     { yylval.fn = 2; return CMP; }
"<>"    { yylval.fn = 3; return CMP; }
"=="    { yylval.fn = 4; return CMP; }
">="    { yylval.fn = 5; return CMP; }
"<="    { yylval.fn = 6; return CMP; }

 /* keywords */

"if"    { return IF; }
"then"  { return THEN; }
"else"  { return ELSE; }
"while" { return WHILE; }
"do"    { return DO; }
"let"   { return LET;}

 /* built-in functions */
"sqrt"  { yylval.fn = B_sqrt; return FUNC; }
"exp"   { yylval.fn = B_exp; return FUNC; }
"log"   { yylval.fn = B_log; return FUNC; }
"print" { yylval.fn = B_print; return FUNC; }

 /* names */
[a-zA-Z][a-zA-Z0-9]*  { yylval.s = lookup(yytext); return NAME; }

[0-9]+"."[0-9]*{EXP}? |
"."?[0-9]+{EXP}? { yylval.d = atof(yytext); return NUMBER; }

"//".*
[ \t]  /* ignore whitespace */

\\\n { printf("c> "); } /* ignore line continuation */

\n   { return EOL; }

.       { yyerror("Mystery character %c\n", *yytext); }
%%
```

The lexer, shown in Example 3-7, adds a few new rules to the previous example. There are a few new single-character operators. The six comparison operators all return a `CMP` token with a lexical value to distinguish them.

The six keywords and four built-in functions are recognized by literal patterns. Note that they have to precede the general pattern to match a name so that they're matched in preference to the general pattern. The name pattern looks up the name in the symbol table and returns a pointer to the symbol.

As before, a newline (`EOL`) marks the end of an input string. Since a function or expression might be too long to type on a single line, we allow continuation lines. A new lexer rule matches a backslash and newline and doesn't return anything to the parser, making the continuation invisible to the parser. But it does print a prompt for the user.

Reserved Words

In this grammar, the words `if`, `then`, `else`, `while`, `do`, `let`, `sqrt`, `exp`, `log`, and `print` are *reserved* and can't be used as user symbols. Whether you want to allow users to use the same name for two things in the same program is debatable. On the one hand, it can make programs harder to understand, but on the other hand, users are otherwise forced to invent names that do not conflict with the reserved names.

Either can be taken to extremes. COBOL has more than 300 reserved words, so nobody can remember them all, and programmers resort to strange conventions like starting every variable name with a digit to be sure it doesn't conflict with a reserved word. On the other hand, PL/I has no reserved words at all, so you can write the following:

```
    IF IF = THEN THEN ELSE = THEN; ELSE ELSE = IF;
```

Bison parsers are a lot easier to write if you reserve the keywords; otherwise, you need to carefully design your language so at each point where the lexer is reading a token, either a name or a keyword is valid, but not both, and you have to provide extensive feedback to the lexer so it knows which to do. Having written lexers for Fortran, which has no reserved words and (in its classic versions) ignores all whitespace, I strongly encourage you to reserve your keywords.

Building and Interpreting ASTs

Finally, we have the file of helper code, shown in Example 3-8. Some of this file is the same as the previous example; the `main` and `yyerror` are unchanged and aren't repeated here.

The key code builds and evaluates the ASTs. First, we have the symbol table management, which should be familiar from the examples in Chapter 2.

Example 3-8. Advanced calculator helper functions fb3-2func.c

```
/*
 * helper functions for fb3-2
 */
#  include <stdio.h>
```

```
#  include <stdlib.h>
#  include <stdarg.h>
#  include <string.h>
#  include <math.h>
#  include "fb3-2.h"

/* symbol table */
/* hash a symbol */
static unsigned
symhash(char *sym)
{
  unsigned int hash = 0;
  unsigned c;

  while(c = *sym++) hash = hash*9 ^ c;

  return hash;
}

struct symbol *
lookup(char* sym)
{
  struct symbol *sp = &symtab[symhash(sym)%NHASH];
  int scount = NHASH;           /* how many have we looked at */

  while(--scount >= 0) {
    if(sp->name && !strcmp(sp->name, sym)) { return sp; }

    if(!sp->name) {             /* new entry */
      sp->name = strdup(sym);
      sp->value = 0;
      sp->func = NULL;
      sp->syms = NULL;
      return sp;
    }

    if(++sp >= symtab+NHASH) sp = symtab; /* try the next entry */
  }
  yyerror("symbol table overflow\n");
  abort(); /* tried them all, table is full */

}
```

Next come the procedures to build the AST nodes and symlists. They all allocate a node and then fill in the fields appropriately for the node type. An extended version of `treefree` recursively walks an AST and frees all of the nodes in the tree.

```
struct ast *
newast(int nodetype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));
```

```
  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = nodetype;
  a->l = l;
  a->r = r;
  return a;
}

struct ast *
newnum(double d)
{
  struct numval *a = malloc(sizeof(struct numval));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'K';
  a->number = d;
  return (struct ast *)a;
}

struct ast *
newcmp(int cmptype, struct ast *l, struct ast *r)
{
  struct ast *a = malloc(sizeof(struct ast));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = '0' + cmptype;
  a->l = l;
  a->r = r;
  return a;
}

struct ast *
newfunc(int functype, struct ast *l)
{
  struct fncall *a = malloc(sizeof(struct fncall));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'F';
  a->l = l;
  a->functype = functype;
  return (struct ast *)a;
}
```

```
struct ast *
newcall(struct symbol *s, struct ast *l)
{
  struct ufncall *a = malloc(sizeof(struct ufncall));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'C';
  a->l = l;
  a->s = s;
  return (struct ast *)a;
}

struct ast *
newref(struct symbol *s)
{
  struct symref *a = malloc(sizeof(struct symref));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = 'N';
  a->s = s;
  return (struct ast *)a;
}

struct ast *
newasgn(struct symbol *s, struct ast *v)
{
  struct symasgn *a = malloc(sizeof(struct symasgn));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
  a->nodetype = '=';
  a->s = s;
  a->v = v;
  return (struct ast *)a;
}

struct ast *
newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *el)
{
  struct flow *a = malloc(sizeof(struct flow));

  if(!a) {
    yyerror("out of space");
    exit(0);
  }
```

```
  a->nodetype = nodetype;
  a->cond = cond;
  a->tl = tl;
  a->el = el;
  return (struct ast *)a;
}

/* free a tree of ASTs */
void
treefree(struct ast *a)
{
  switch(a->nodetype) {

    /* two subtrees */
  case '+':
  case '-':
  case '*':
  case '/':
  case '1':  case '2':  case '3':  case '4':  case '5':  case '6':
  case 'L':
    treefree(a->r);

    /* one subtree */
  case '|':
  case 'M': case 'C': case 'F':
    treefree(a->l);

    /* no subtree */
  case 'K': case 'N':
    break;

  case '=':
    free( ((struct symasgn *)a)->v);
    break;

    /* up to three subtrees */
  case 'I': case 'W':
    free( ((struct flow *)a)->cond);
    if( ((struct flow *)a)->tl) treefree( ((struct flow *)a)->tl);
    if( ((struct flow *)a)->el) treefree( ((struct flow *)a)->el);
    break;

  default: printf("internal error: free bad node %c\n", a->nodetype);
  }

  free(a); /* always free the node itself */
}

struct symlist *
newsymlist(struct symbol *sym, struct symlist *next)
{
  struct symlist *sl = malloc(sizeof(struct symlist));

  if(!sl) {
```

```
    yyerror("out of space");
    exit(0);
  }
  sl->sym = sym;
  sl->next = next;
  return sl;
}

/* free a list of symbols */
void
symlistfree(struct symlist *sl)
{
  struct symlist *nsl;

  while(sl) {
    nsl = sl->next;
    free(sl);
    sl = nsl;
  }
}
```

The heart of the calculator is `eval`, which evaluates an AST built up in the parser. Following the practice in C, comparisons return 1 or 0 depending on whether the comparison succeeds, and tests in if/then/else and while/do treat any nonzero as true.

For expressions, we do the familiar depth-first tree walk to compute the value. An AST makes it straightforward to implement if/then/else: Evaluate the condition AST to decide which branch to take, and then evaluate the AST for the path to be taken. To evaluate while/do loops, a loop in eval evaluates the condition AST, then the body AST, repeating as long as the condition AST remains true. Any AST that references variables that are changed by an assignment will have a new value each time it's evaluated.

```
static double callbuiltin(struct fncall *);
static double calluser(struct ufncall *);

double
eval(struct ast *a)
{
  double v;

  if(!a) {
    yyerror("internal error, null eval");
    return 0.0;
  }

  switch(a->nodetype) {
    /* constant */
  case 'K': v = ((struct numval *)a)->number; break;
```

```
  /* name reference */
case 'N': v = ((struct symref *)a)->s->value; break;

  /* assignment */
case '=': v = ((struct symasgn *)a)->s->value =
    eval(((struct symasgn *)a)->v); break;

  /* expressions */
case '+': v = eval(a->l) + eval(a->r); break;
case '-': v = eval(a->l) - eval(a->r); break;
case '*': v = eval(a->l) * eval(a->r); break;
case '/': v = eval(a->l) / eval(a->r); break;
case '|': v = fabs(eval(a->l)); break;
case 'M': v = -eval(a->l); break;

  /* comparisons */
case '1': v = (eval(a->l) > eval(a->r))? 1 : 0; break;
case '2': v = (eval(a->l) < eval(a->r))? 1 : 0; break;
case '3': v = (eval(a->l) != eval(a->r))? 1 : 0; break;
case '4': v = (eval(a->l) == eval(a->r))? 1 : 0; break;
case '5': v = (eval(a->l) >= eval(a->r))? 1 : 0; break;
case '6': v = (eval(a->l) <= eval(a->r))? 1 : 0; break;

/* control flow */
/* null expressions allowed in the grammar, so check for them */

/* if/then/else */
case 'I':
  if( eval( ((struct flow *)a)->cond) != 0) {   check the condition
    if( ((struct flow *)a)->tl) {               the true branch
      v = eval( ((struct flow *)a)->tl);
    } else
      v = 0.0;               /* a default value */
  } else {
    if( ((struct flow *)a)->el) {               the false branch
      v = eval(((struct flow *)a)->el);
    } else
      v = 0.0;               /* a default value */
  }
  break;

/* while/do */
case 'W':
  v = 0.0;           /* a default value */

  if( ((struct flow *)a)->tl) {
    while( eval(((struct flow *)a)->cond) != 0) evaluate the condition
      v = eval(((struct flow *)a)->tl);         evaluate the target statements
  }
  break;                     /* value of last statement is value of while/d

/* list of statements */
case 'L': eval(a->l); v = eval(a->r); break;
```

```
    case 'F': v = callbuiltin((struct fncall *)a); break;

    case 'C': v = calluser((struct ufncall *)a); break;

    default: printf("internal error: bad node %c\n", a->nodetype);
    }
    return v;
  }
```

Evaluating Functions in the Calculator

The trickiest bits of code in the evaluator handle functions. Built-in functions are relatively straightforward: They determine which function it is and call specific code to do the function.

```
  static double
  callbuiltin(struct fncall *f)
  {
    enum bifs functype = f->functype;
    double v = eval(f->l);

   switch(functype) {
   case B_sqrt:
     return sqrt(v);
   case B_exp:
     return exp(v);
   case B_log:
     return log(v);
   case B_print:
     printf("= %4.4g\n", v);
     return v;
   default:
     yyerror("Unknown built-in function %d", functype);
     return 0.0;
   }
  }
```

User-Defined Functions

A function definition consists of the name of the function, a list of dummy arguments, and an AST that represents the body of the function. Defining the function simply saves the argument list and AST in the function's symbol table entry, replacing any previous version.

```
  /* define a function */
  void
  dodef(struct symbol *name, struct symlist *syms, struct ast *func)
  {
    if(name->syms) symlistfree(name->syms);
```

```
    if(name->func) treefree(name->func);
    name->syms = syms;
    name->func = func;
  }
```

Say you define a function to calculate the maximum of its two arguments:

```
  > let max(x,y) = if x >= y then x; else y;;
  > max(4+5,6+7)
```

The function has two dummy arguments, x and y. When the function is called, the evaluator does this:

1. Evaluate the actual arguments, 4+5 and 6+7 in this case.
2. Save the current values of the dummy arguments and assign the values of the actual arguments to them.
3. Evaluate the body of the function, which will now use the actual argument values when it refers to the dummy arguments.
4. Put back the old values of the dummies.
5. Return the value of the body expression.

The code to do this counts the arguments, allocates two temporary arrays for the old and new values of the dummy arguments, and then does the steps described earlier.

```
  static double
  calluser(struct ufncall *f)
  {
    struct symbol *fn = f->s;     /* function name */
    struct symlist *sl;           /* dummy arguments */
    struct ast *args = f->l;      /* actual arguments */
    double *oldval, *newval;      /* saved arg values */
    double v;
    int nargs;
    int i;

    if(!fn->func) {
      yyerror("call to undefined function %s", fn->name);
      return 0;
    }

    /* count the arguments */
    sl = fn->syms;
    for(nargs = 0; sl; sl = sl->next)
      nargs++;

    /* prepare to save them */
    oldval = (double *)malloc(nargs * sizeof(double));
    newval = (double *)malloc(nargs * sizeof(double));
    if(!oldval || !newval) {
```

```
    yyerror("Out of space in %s", fn->name); return 0.0;
  }

  /* evaluate the arguments */
  for(i = 0; i < nargs; i++) {
    if(!args) {
      yyerror("too few args in call to %s", fn->name);
      free(oldval); free(newval);
      return 0.0;
    }

    if(args->nodetype == 'L') { /* if this is a list node */
      newval[i] = eval(args->l);
      args = args->r;
    } else {                    /* if it's the end of the list */
      newval[i] = eval(args);
      args = NULL;
    }
  }

  /* save old values of dummies, assign new ones */
  sl = fn->syms;
  for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;

    oldval[i] = s->value;
    s->value = newval[i];
    sl = sl->next;
  }

  free(newval);

  /* evaluate the function */
  v = eval(fn->func);

  /* put the real values of the dummies back */
  sl = fn->syms;
  for(i = 0; i < nargs; i++) {
    struct symbol *s = sl->sym;

    s->value = oldval[i];
    sl = sl->next;
  }

  free(oldval);
  return v;
}
```

Using the Advanced Calculator

This calculator is flexible enough to do some useful calculations. Example 3-9 shows the function `sq` to compute square roots iteratively

using Newton's method, as well as an auxiliary function `avg` to compute the average of two numbers.

Example 3-9. Computing square roots with the calculator

```
> let sq(n)=e=1; while |((t=n/e)-e)>.001 do e=avg(e,t);;
Defined sq
> let avg(a,b)=(a+b)/2;
Defined avg
> sq(10)
= 3.162
> sqrt(10)
= 3.162
> sq(10)-sqrt(10)
= 0.000178      accurate to better than the .001 cutoff
```

Exercises

1. Try some variants of the syntax in the enhanced calculator. In the previous example, the `sq` function has to end with two semicolons to close off both the while loop and the let statement, which is pretty clunky. Can you change the syntax to make it more intuitive? If you add closing symbols to conditional statements if/then/else/fi and loops while/do/done, can you make the syntax of statement lists more flexible?
2. In the last example, user functions evaluate all of the actual arguments, put them in a temporary array, and then assign them to the dummy arguments. Why not just do them one at a time and set the dummies as the actuals are evaluated?

---

[10] It is possible to have rules with empty right-hand sides, in which case a reduction ends up with one more item than it started with, but we call them reductions anyway.

---

[11] Then we'll stop; the world has plenty of script interpreter programs already. For practical purposes, you'll be much better off adding a few extensions to an existing well-debugged scripting language such as Python, Perl, or Lua rather than writing yet another one.

[12] These could have gone in the third section of the parser, but it's easier to debug your program if you don't put a lot of extra code in the parser file.

---

[13] This is a classic C trick that doesn't work in C++. If you write your parser in C++, as we describe in Chapter 9, you'll have to use an explicit union of structures within the AST structure to get a similar result.

---

[14] This design is vaguely based on the ancient BLISS system programming language. It would be perfectly possible to design an AST that treated expressions and statements separately, but the implementation is a little simpler this way.