2 Examples

Now we show and explain several sample programs written using Bison: a Reverse Polish Notation calculator, an algebraic (infix) notation calculator — later extended to track "locations" — and a multi-function calculator. All produce usable, though limited, interactive desk-top calculators.

These examples are simple, but Bison grammars for real programming languages are written the same way. You can copy these examples into a source file to try them.

2.1 Reverse Polish Notation Calculator

The first example is that of a simple double-precision *Reverse Polish Notation* calculator (a calculator using postfix operators). This example provides a good starting point, since operator precedence is not an issue. The second example will illustrate how operator precedence is handled.

The source code for this calculator is named `rpcalc.y`. The '`.y`' extension is a convention used for Bison grammar files.

2.1.1 Declarations for `rpcalc`

Here are the C and Bison declarations for the Reverse Polish Notation calculator. As in C, comments are placed between '`/*...*/`'.

```
/* Reverse Polish Notation calculator.  */

%{
  #include <stdio.h>
  #include <math.h>
  int yylex (void);
  void yyerror (char const *);
%}

%define api.value.type {double}
%token NUM

%% /* Grammar rules and actions follow.  */
```

The declarations section (see Section 3.1.1 [The prologue], page 51) contains two preprocessor directives and two forward declarations.

The `#include` directive is used to declare the exponentiation function `pow`.

The forward declarations for `yylex` and `yyerror` are needed because the C language requires that functions be declared before they are used. These functions will be defined in the epilogue, but the parser calls them so they must be declared in the prologue.

The second section, Bison declarations, provides information to Bison about the tokens and their types (see Section 3.1.3 [The Bison Declarations Section], page 56).

The `%define` directive defines the variable `api.value.type`, thus specifying the C data type for semantic values of both tokens and groupings (see Section 3.4.1 [Data Types of Semantic Values], page 61). The Bison parser will use whatever type `api.value.type` is defined as; if you don't define it, `int` is the default. Because we specify '`{double}`', each token and each expression has an associated value, which is a floating point number. C code can use `YYSTYPE` to refer to the value `api.value.type`.

Each terminal symbol that is not a single-character literal must be declared. (Single-character literals normally don't need to be declared.) In this example, all the arithmetic operators are designated by single-character literals, so the only terminal symbol that needs to be declared is `NUM`, the token type for numeric constants.

2.1.2 Grammar Rules for `rpcalc`

Here are the grammar rules for the Reverse Polish Notation calculator.

```
input:
  %empty
| input line
;

line:
  '\n'
| exp '\n'      { printf ("%.10g\n", $1); }
;

exp:
  NUM           { $$ = $1;           }
| exp exp '+'   { $$ = $1 + $2;      }
| exp exp '-'   { $$ = $1 - $2;      }
| exp exp '*'   { $$ = $1 * $2;      }
| exp exp '/'   { $$ = $1 / $2;      }
| exp exp '^'   { $$ = pow ($1, $2); }  /* Exponentiation */
| exp 'n'       { $$ = -$1;          }  /* Unary minus    */
;
%%
```

The groupings of the rpcalc "language" defined here are the expression (given the name `exp`), the line of input (`line`), and the complete input transcript (`input`). Each of these nonterminal symbols has several alternate rules, joined by the vertical bar '`|`' which is read as "or". The following sections explain what these rules mean.

The semantics of the language is determined by the actions taken when a grouping is recognized. The actions are the C code that appears inside braces. See Section 3.4.6 [Actions], page 64.

You must specify these actions in C, but Bison provides the means for passing semantic values between the rules. In each action, the pseudo-variable `$$` stands for the semantic value for the grouping that the rule is going to construct. Assigning a value to `$$` is the main job of most actions. The semantic values of the components of the rule are referred to as `$1`, `$2`, and so on.

2.1.2.1 Explanation of `input`

Consider the definition of `input`:

```
input:
```

```
          %empty
        | input line
        ;
```

This definition reads as follows: "A complete input is either an empty string, or a complete input followed by an input line". Notice that "complete input" is defined in terms of itself. This definition is said to be *left recursive* since `input` appears always as the leftmost symbol in the sequence. See Section 3.3.3 [Recursive Rules], page 60.

The first alternative is empty because there are no symbols between the colon and the first '`|`'; this means that `input` can match an empty string of input (no tokens). We write the rules this way because it is legitimate to type *`Ctrl-d`* right after you start the calculator. It's conventional to put an empty alternative first and to use the (optional) `%empty` directive, or to write the comment '`/* empty */`' in it (see Section 3.3.2 [Empty Rules], page 60).

The second alternate rule (`input line`) handles all nontrivial input. It means, "After reading any number of lines, read one more line if possible." The left recursion makes this rule into a loop. Since the first alternative matches empty input, the loop can be executed zero or more times.

The parser function `yyparse` continues to process input until a grammatical error is seen or the lexical analyzer says there are no more input tokens; we will arrange for the latter to happen at end-of-input.

2.1.2.2 Explanation of `line`

Now consider the definition of `line`:

```
      line:
        '\n'
      | exp '\n'  { printf ("%.10g\n", $1); }
      ;
```

The first alternative is a token which is a newline character; this means that rpcalc accepts a blank line (and ignores it, since there is no action). The second alternative is an expression followed by a newline. This is the alternative that makes rpcalc useful. The semantic value of the `exp` grouping is the value of `$1` because the `exp` in question is the first symbol in the alternative. The action prints this value, which is the result of the computation the user asked for.

This action is unusual because it does not assign a value to `$$`. As a consequence, the semantic value associated with the `line` is uninitialized (its value will be unpredictable). This would be a bug if that value were ever used, but we don't use it: once rpcalc has printed the value of the user's input line, that value is no longer needed.

2.1.2.3 Explanation of `expr`

The `exp` grouping has several rules, one for each kind of expression. The first rule handles the simplest expressions: those that are just numbers. The second handles an addition-expression, which looks like two expressions followed by a plus-sign. The third handles subtraction, and so on.

```
      exp:
        NUM
      | exp exp '+'     { $$ = $1 + $2;    }
```

```
| exp exp '-'     { $$ = $1 - $2;    }
...
;
```

We have used '|' to join all the rules for `exp`, but we could equally well have written them separately:

```
exp: NUM ;
exp: exp exp '+'     { $$ = $1 + $2; };
exp: exp exp '-'     { $$ = $1 - $2; };
...
```

Most of the rules have actions that compute the value of the expression in terms of the value of its parts. For example, in the rule for addition, `$1` refers to the first component `exp` and `$2` refers to the second one. The third component, `'+'`, has no meaningful associated semantic value, but if it had one you could refer to it as `$3`. When `yyparse` recognizes a sum expression using this rule, the sum of the two subexpressions' values is produced as the value of the entire expression. See Section 3.4.6 [Actions], page 64.

You don't have to give an action for every rule. When a rule has no action, Bison by default copies the value of `$1` into `$$`. This is what happens in the first rule (the one that uses `NUM`).

The formatting shown here is the recommended convention, but Bison does not require it. You can add or change white space as much as you wish. For example, this:

```
exp: NUM | exp exp '+' {$$ = $1 + $2; } | ... ;
```

means the same thing as this:

```
exp:
  NUM
| exp exp '+'    { $$ = $1 + $2; }
| ...
;
```

The latter, however, is much more readable.

2.1.3 The `rpcalc` Lexical Analyzer

The lexical analyzer's job is low-level parsing: converting characters or sequences of characters into tokens. The Bison parser gets its tokens by calling the lexical analyzer. See Section 4.6 [The Lexical Analyzer Function `yylex`], page 99.

Only a simple lexical analyzer is needed for the RPN calculator. This lexical analyzer skips blanks and tabs, then reads in numbers as `double` and returns them as `NUM` tokens. Any other character that isn't part of a number is a separate token. Note that the token-code for such a single-character token is the character itself.

The return value of the lexical analyzer function is a numeric code which represents a token type. The same text used in Bison rules to stand for this token type is also a C expression for the numeric code for the type. This works in two ways. If the token type is a character literal, then its numeric code is that of the character; you can use the same character literal in the lexical analyzer to express the number. If the token type is an identifier, that identifier is defined by Bison as a C macro whose definition is the appropriate number. In this example, therefore, `NUM` becomes a macro for `yylex` to use.

The semantic value of the token (if it has one) is stored into the global variable `yylval`, which is where the Bison parser will look for it. (The C data type of `yylval` is `YYSTYPE`, whose value was defined at the beginning of the grammar via '`%define api.value.type {double}`'; see Section 2.1.1 [Declarations for `rpcalc`], page 31.)

A token type code of zero is returned if the end-of-input is encountered. (Bison recognizes any nonpositive value as indicating end-of-input.)

Here is the code for the lexical analyzer:

```
/* The lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the numeric code
   of the character read if not a number.  It skips all blanks
   and tabs, and returns 0 for end-of-input.  */

#include <ctype.h>

int
yylex (void)
{
  int c;

  /* Skip white space.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    continue;
  /* Process numbers.  */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* Return end-of-input.  */
  if (c == EOF)
    return 0;
  /* Return a single char.  */
  return c;
}
```

2.1.4 The Controlling Function

In keeping with the spirit of this example, the controlling function is kept to the bare minimum. The only requirement is that it call `yyparse` to start the process of parsing.

```
int
main (void)
{
  return yyparse ();
}
```

2.1.5 The Error Reporting Routine

When `yyparse` detects a syntax error, it calls the error reporting function `yyerror` to print an error message (usually but not always `"syntax error"`). It is up to the programmer to supply `yyerror` (see Chapter 4 [Parser C-Language Interface], page 97), so here is the definition we will use:

```
#include <stdio.h>

/* Called by yyparse on error.  */
void
yyerror (char const *s)
{
  fprintf (stderr, "%s\n", s);
}
```

After `yyerror` returns, the Bison parser may recover from the error and continue parsing if the grammar contains a suitable error rule (see Chapter 6 [Error Recovery], page 127). Otherwise, `yyparse` returns nonzero. We have not written any error rules in this example, so any invalid input will cause the calculator program to exit. This is not clean behavior for a real calculator, but it is adequate for the first example.

2.1.6 Running Bison to Make the Parser

Before running Bison to produce a parser, we need to decide how to arrange all the source code in one or more source files. For such a simple example, the easiest thing is to put everything in one file, the grammar file. The definitions of `yylex`, `yyerror` and `main` go at the end, in the epilogue of the grammar file (see Section 1.9 [The Overall Layout of a Bison Grammar], page 29).

For a large project, you would probably have several source files, and use `make` to arrange to recompile them.

With all the source in the grammar file, you use the following command to convert it into a parser implementation file:

```
bison file.y
```

In this example, the grammar file is called `rpcalc.y` (for "Reverse Polish CALCulator"). Bison produces a parser implementation file named *`file`*`.tab.c`, removing the '`.y`' from the grammar file name. The parser implementation file contains the source code for `yyparse`. The additional functions in the grammar file (`yylex`, `yyerror` and `main`) are copied verbatim to the parser implementation file.

2.1.7 Compiling the Parser Implementation File

Here is how to compile and run the parser implementation file:

```
# List files in current directory.
$ ls
rpcalc.tab.c  rpcalc.y

# Compile the Bison parser.
# '-lm' tells compiler to search math library for pow.
$ cc -lm -o rpcalc rpcalc.tab.c
```

```
# List files again.
$ ls
rpcalc  rpcalc.tab.c  rpcalc.y
```

The file `rpcalc` now contains the executable code. Here is an example session using `rpcalc`.

```
$ rpcalc
4 9 +
⇒ 13
3 7 + 3 4 5 *+-
⇒ -13
3 7 + 3 4 5 * + - n              Note the unary minus, ‘n’
⇒ 13
5 6 / 4 n +
⇒ -3.166666667
3 4 ^                            Exponentiation
⇒ 81
^D                               End-of-file indicator
$
```

2.2 Infix Notation Calculator: `calc`

We now modify rpcalc to handle infix operators instead of postfix. Infix notation involves the concept of operator precedence and the need for parentheses nested to arbitrary depth. Here is the Bison code for `calc.y`, an infix desk-top calculator.

```
/* Infix notation calculator.  */

%{
  #include <math.h>
  #include <stdio.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */
%define api.value.type {double}
%token NUM
%left '-' '+'
%left '*' '/'
%precedence NEG   /* negation--unary minus */
%right '^'        /* exponentiation */

%% /* The grammar follows.  */
input:
  %empty
| input line
;
```

```
line:
  '\n'
| exp '\n'  { printf ("\t%.10g\n", $1); }
;

exp:
  NUM                { $$ = $1;           }
| exp '+' exp        { $$ = $1 + $3;      }
| exp '-' exp        { $$ = $1 - $3;      }
| exp '*' exp        { $$ = $1 * $3;      }
| exp '/' exp        { $$ = $1 / $3;      }
| '-' exp  %prec NEG { $$ = -$2;          }
| exp '^' exp        { $$ = pow ($1, $3); }
| '(' exp ')'        { $$ = $2;           }
;
%%
```

The functions `yylex`, `yyerror` and `main` can be the same as before.

There are two important new features shown in this code.

In the second section (Bison declarations), `%left` declares token types and says they are left-associative operators. The declarations `%left` and `%right` (right associativity) take the place of `%token` which is used to declare a token type name without associativity/precedence. (These tokens are single-character literals, which ordinarily don't need to be declared. We declare them here to specify the associativity/precedence.)

Operator precedence is determined by the line ordering of the declarations; the higher the line number of the declaration (lower on the page or screen), the higher the precedence. Hence, exponentiation has the highest precedence, unary minus (`NEG`) is next, followed by '`*`' and '`/`', and so on. Unary minus is not associative, only precedence matters (`%precedence`. See Section 5.3 [Operator Precedence], page 109.

The other important new feature is the `%prec` in the grammar section for the unary minus operator. The `%prec` simply instructs Bison that the rule '`| '-' exp`' has the same precedence as `NEG`—in this case the next-to-highest. See Section 5.4 [Context-Dependent Precedence], page 112.

Here is a sample run of `calc.y`:

```
$ calc
4 + 4.5 - (34/(8*3+-3))
6.880952381
-56 + 2
-54
3 ^ 2
9
```

2.3 Simple Error Recovery

Up to this point, this manual has not addressed the issue of *error recovery*—how to continue parsing after the parser detects a syntax error. All we have handled is error reporting with `yyerror`. Recall that by default `yyparse` returns after calling `yyerror`. This means that an erroneous input line causes the calculator program to exit. Now we show how to rectify this deficiency.

The Bison language itself includes the reserved word `error`, which may be included in the grammar rules. In the example below it has been added to one of the alternatives for `line`:

```
line:
  '\n'
| exp '\n'   { printf ("\t%.10g\n", $1); }
| error '\n' { yyerrok;                  }
;
```

This addition to the grammar allows for simple error recovery in the event of a syntax error. If an expression that cannot be evaluated is read, the error will be recognized by the third rule for `line`, and parsing will continue. (The `yyerror` function is still called upon to print its message as well.) The action executes the statement `yyerrok`, a macro defined automatically by Bison; its meaning is that error recovery is complete (see Chapter 6 [Error Recovery], page 127). Note the difference between `yyerrok` and `yyerror`; neither one is a misprint.

This form of error recovery deals with syntax errors. There are other kinds of errors; for example, division by zero, which raises an exception signal that is normally fatal. A real calculator program must handle this signal and use `longjmp` to return to `main` and resume parsing input lines; it would also have to discard the rest of the current line of input. We won't discuss this issue further because it is not specific to Bison programs.

2.4 Location Tracking Calculator: `ltcalc`

This example extends the infix notation calculator with location tracking. This feature will be used to improve the error messages. For the sake of clarity, this example is a simple integer calculator, since most of the work needed to use locations will be done in the lexical analyzer.

2.4.1 Declarations for `ltcalc`

The C and Bison declarations for the location tracking calculator are the same as the declarations for the infix notation calculator.

```
/* Location tracking calculator.  */

%{
  #include <math.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */
%define api.value.type {int}
%token NUM
```

```
%left '-' '+'
%left '*' '/'
%precedence NEG
%right '^'

%% /* The grammar follows.  */
```

Note there are no declarations specific to locations. Defining a data type for storing locations is not needed: we will use the type provided by default (see Section 3.5.1 [Data Types of Locations], page 71), which is a four member structure with the following integer fields: `first_line`, `first_column`, `last_line` and `last_column`. By conventions, and in accordance with the GNU Coding Standards and common practice, the line and column count both start at 1.

2.4.2 Grammar Rules for `ltcalc`

Whether handling locations or not has no effect on the syntax of your language. Therefore, grammar rules for this example will be very close to those of the previous example: we will only modify them to benefit from the new information.

Here, we will use locations to report divisions by zero, and locate the wrong expressions or subexpressions.

```
input:
  %empty
| input line
;

line:
  '\n'
| exp '\n' { printf ("%d\n", $1); }
;

exp:
  NUM           { $$ = $1; }
| exp '+' exp   { $$ = $1 + $3; }
| exp '-' exp   { $$ = $1 - $3; }
| exp '*' exp   { $$ = $1 * $3; }
| exp '/' exp
    {
      if ($3)
        $$ = $1 / $3;
      else
        {
          $$ = 1;
          fprintf (stderr, "%d.%d-%d.%d: division by zero",
                   @3.first_line, @3.first_column,
                   @3.last_line, @3.last_column);
        }
    }
```

```
        | '-' exp %prec NEG     { $$ = -$2; }
        | exp '^' exp           { $$ = pow ($1, $3); }
        | '(' exp ')'           { $$ = $2; }
```

This code shows how to reach locations inside of semantic actions, by using the pseudo-variables `@n` for rule components, and the pseudo-variable `@$` for groupings.

We don't need to assign a value to `@$`: the output parser does it automatically. By default, before executing the C code of each action, `@$` is set to range from the beginning of `@1` to the end of `@n`, for a rule with *n* components. This behavior can be redefined (see Section 3.5.3 [Default Action for Locations], page 73), and for very specific rules, `@$` can be computed by hand.

2.4.3 The `ltcalc` Lexical Analyzer.

Until now, we relied on Bison's defaults to enable location tracking. The next step is to rewrite the lexical analyzer, and make it able to feed the parser with the token locations, as it already does for semantic values.

To this end, we must take into account every single character of the input text, to avoid the computed locations of being fuzzy or wrong:

```
int
yylex (void)
{
  int c;

  /* Skip white space.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ++yylloc.last_column;

  /* Step.  */
  yylloc.first_line = yylloc.last_line;
  yylloc.first_column = yylloc.last_column;

  /* Process numbers.  */
  if (isdigit (c))
    {
      yylval = c - '0';
      ++yylloc.last_column;
      while (isdigit (c = getchar ()))
        {
          ++yylloc.last_column;
          yylval = yylval * 10 + c - '0';
        }
      ungetc (c, stdin);
      return NUM;
    }

  /* Return end-of-input.  */
  if (c == EOF)
```

```
      return 0;

    /* Return a single char, and update location.  */
    if (c == '\n')
      {
        ++yylloc.last_line;
        yylloc.last_column = 0;
      }
    else
      ++yylloc.last_column;
    return c;
  }
```

Basically, the lexical analyzer performs the same processing as before: it skips blanks and tabs, and reads numbers or single-character tokens. In addition, it updates `yylloc`, the global variable (of type `YYLTYPE`) containing the token's location.

Now, each time this function returns a token, the parser has its number as well as its semantic value, and its location in the text. The last needed change is to initialize `yylloc`, for example in the controlling function:

```
int
main (void)
{
  yylloc.first_line = yylloc.last_line = 1;
  yylloc.first_column = yylloc.last_column = 0;
  return yyparse ();
}
```

Remember that computing locations is not a matter of syntax. Every character must be associated to a location update, whether it is in valid input, in comments, in literal strings, and so on.

2.5 Multi-Function Calculator: `mfcalc`

Now that the basics of Bison have been discussed, it is time to move on to a more advanced problem. The above calculators provided only five functions, '`+`', '`-`', '`*`', '`/`' and '`^`'. It would be nice to have a calculator that provides other mathematical functions such as `sin`, `cos`, etc.

It is easy to add new operators to the infix calculator as long as they are only single-character literals. The lexical analyzer `yylex` passes back all nonnumeric characters as tokens, so new grammar rules suffice for adding a new operator. But we want something more flexible: built-in functions whose syntax has this form:

```
function_name (argument)
```

At the same time, we will add memory to the calculator, by allowing you to create named variables, store values in them, and use them later. Here is a sample session with the multi-function calculator:

```
$ mfcalc
pi = 3.141592653589
⇒ 3.1415926536
```

```
sin(pi)
⇒ 0.0000000000
alpha = beta1 = 2.3
⇒ 2.3000000000
alpha
⇒ 2.3000000000
ln(alpha)
⇒ 0.8329091229
exp(ln(beta1))
⇒ 2.3000000000
$
```

Note that multiple assignment and nested function calls are permitted.

2.5.1 Declarations for `mfcalc`

Here are the C and Bison declarations for the multi-function calculator.

```
%{
  #include <stdio.h>  /* For printf, etc. */
  #include <math.h>   /* For pow, used in the grammar.  */
  #include "calc.h"   /* Contains definition of 'symrec'.  */
  int yylex (void);
  void yyerror (char const *);
%}

%define api.value.type union /* Generate YYSTYPE from these types:  */
%token <double>  NUM         /* Simple double precision number.  */
%token <symrec*> VAR FNCT    /* Symbol table pointer: variable and func-
tion.  */
%type  <double>  exp

%precedence '='
%left '-' '+'
%left '*' '/'
%precedence NEG /* negation--unary minus */
%right '^'      /* exponentiation */
```

The above grammar introduces only two new features of the Bison language. These features allow semantic values to have various data types (see Section 3.4.2 [More Than One Value Type], page 62).

The special `union` value assigned to the `%define` variable `api.value.type` specifies that the symbols are defined with their data types. Bison will generate an appropriate definition of `YYSTYPE` to store these values.

Since values can now have various types, it is necessary to associate a type with each grammar symbol whose semantic value is used. These symbols are `NUM`, `VAR`, `FNCT`, and `exp`. Their declarations are augmented with their data type (placed between angle brackets). For instance, values of `NUM` are stored in `double`.

The Bison construct `%type` is used for declaring nonterminal symbols, just as `%token` is used for declaring token types. Previously we did not use `%type` before because nonterminal symbols are normally declared implicitly by the rules that define them. But `exp` must be declared explicitly so we can specify its value type. See Section 3.7.4 [Nonterminal Symbols], page 77.

2.5.2 Grammar Rules for `mfcalc`

Here are the grammar rules for the multi-function calculator. Most of them are copied directly from `calc`; three rules, those which mention `VAR` or `FNCT`, are new.

```
%% /* The grammar follows.  */
input:
  %empty
| input line
;

line:
  '\n'
| exp '\n'   { printf ("%.10g\n", $1); }
| error '\n' { yyerrok;                }
;

exp:
  NUM                { $$ = $1;                         }
| VAR                { $$ = $1->value.var;              }
| VAR '=' exp        { $$ = $3; $1->value.var = $3;     }
| FNCT '(' exp ')'   { $$ = (*($1->value.fnctptr))($3); }
| exp '+' exp        { $$ = $1 + $3;                    }
| exp '-' exp        { $$ = $1 - $3;                    }
| exp '*' exp        { $$ = $1 * $3;                    }
| exp '/' exp        { $$ = $1 / $3;                    }
| '-' exp  %prec NEG { $$ = -$2;                        }
| exp '^' exp        { $$ = pow ($1, $3);               }
| '(' exp ')'        { $$ = $2;                         }
;
/* End of grammar.  */
%%
```

2.5.3 The `mfcalc` Symbol Table

The multi-function calculator requires a symbol table to keep track of the names and meanings of variables and functions. This doesn't affect the grammar rules (except for the actions) or the Bison declarations, but it requires some additional C functions for support.

The symbol table itself consists of a linked list of records. Its definition, which is kept in the header `calc.h`, is as follows. It provides for either functions or variables to be placed in the table.

```
/* Function type.  */
typedef double (*func_t) (double);
```

```
/* Data type for links in the chain of symbols.  */
struct symrec
{
  char *name;  /* name of symbol */
  int type;    /* type of symbol: either VAR or FNCT */
  union
  {
    double var;      /* value of a VAR */
    func_t fnctptr;  /* value of a FNCT */
  } value;
  struct symrec *next;  /* link field */
};

typedef struct symrec symrec;

/* The symbol table: a chain of 'struct symrec'.  */
extern symrec *sym_table;

symrec *putsym (char const *, int);
symrec *getsym (char const *);
```

The new version of `main` will call `init_table` to initialize the symbol table:

```
struct init
{
  char const *fname;
  double (*fnct) (double);
};

struct init const arith_fncts[] =
{
  { "atan", atan },
  { "cos",  cos  },
  { "exp",  exp  },
  { "ln",   log  },
  { "sin",  sin  },
  { "sqrt", sqrt },
  { 0, 0 },
};

/* The symbol table: a chain of 'struct symrec'.  */
symrec *sym_table;
```

```
/* Put arithmetic functions in table.  */
static
void
init_table (void)
{
  int i;
  for (i = 0; arith_fncts[i].fname != 0; i++)
    {
      symrec *ptr = putsym (arith_fncts[i].fname, FNCT);
      ptr->value.fnctptr = arith_fncts[i].fnct;
    }
}
```

By simply editing the initialization list and adding the necessary include files, you can add additional functions to the calculator.

Two important functions allow look-up and installation of symbols in the symbol table. The function `putsym` is passed a name and the type (`VAR` or `FNCT`) of the object to be installed. The object is linked to the front of the list, and a pointer to the object is returned. The function `getsym` is passed the name of the symbol to look up. If found, a pointer to that symbol is returned; otherwise zero is returned.

```
#include <stdlib.h> /* malloc. */
#include <string.h> /* strlen. */

symrec *
putsym (char const *sym_name, int sym_type)
{
  symrec *ptr = (symrec *) malloc (sizeof (symrec));
  ptr->name = (char *) malloc (strlen (sym_name) + 1);
  strcpy (ptr->name,sym_name);
  ptr->type = sym_type;
  ptr->value.var = 0; /* Set value to 0 even if fctn.  */
  ptr->next = (struct symrec *)sym_table;
  sym_table = ptr;
  return ptr;
}

symrec *
getsym (char const *sym_name)
{
  symrec *ptr;
  for (ptr = sym_table; ptr != (symrec *) 0;
       ptr = (symrec *)ptr->next)
    if (strcmp (ptr->name, sym_name) == 0)
      return ptr;
  return 0;
}
```

2.5.4 The mfcalc Lexer

The function yylex must now recognize variables, numeric values, and the single-character arithmetic operators. Strings of alphanumeric characters with a leading letter are recognized as either variables or functions depending on what the symbol table says about them.

The string is passed to getsym for look up in the symbol table. If the name appears in the table, a pointer to its location and its type (VAR or FNCT) is returned to yyparse. If it is not already in the table, then it is installed as a VAR using putsym. Again, a pointer and its type (which must be VAR) is returned to yyparse.

No change is needed in the handling of numeric values and arithmetic operators in yylex.

```
#include <ctype.h>

int
yylex (void)
{
  int c;

  /* Ignore white space, get first nonwhite character.  */
  while ((c = getchar ()) == ' ' || c == '\t')
    continue;

  if (c == EOF)
    return 0;

  /* Char starts a number => parse the number.         */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval.NUM);
      return NUM;
    }
```

Bison generated a definition of YYSTYPE with a member named NUM to store value of NUM symbols.

```
  /* Char starts an identifier => read the name.       */
  if (isalpha (c))
    {
      /* Initially make the buffer long enough
         for a 40-character symbol name.  */
      static size_t length = 40;
      static char *symbuf = 0;
      symrec *s;
      int i;
      if (!symbuf)
        symbuf = (char *) malloc (length + 1);

      i = 0;
```

```
          do
            {
              /* If buffer is full, make it bigger.        */
              if (i == length)
                {
                  length *= 2;
                  symbuf = (char *) realloc (symbuf, length + 1);
                }
              /* Add this character to the buffer.         */
              symbuf[i++] = c;
              /* Get another character.                    */
              c = getchar ();
            }
          while (isalnum (c));

          ungetc (c, stdin);
          symbuf[i] = '\0';

          s = getsym (symbuf);
          if (s == 0)
            s = putsym (symbuf, VAR);
          *((symrec**) &yylval) = s;
          return s->type;
        }

      /* Any other character is a token by itself.        */
      return c;
    }
```

2.5.5 The mfcalc Main

The error reporting function is unchanged, and the new version of `main` includes a call to `init_table` and sets the `yydebug` on user demand (See Section 8.4 [Tracing Your Parser], page 143, for details):

```
/* Called by yyparse on error.  */
void
yyerror (char const *s)
{
  fprintf (stderr, "%s\n", s);
}
```

```
int
main (int argc, char const* argv[])
{
  int i;
  /* Enable parse traces on option -p.  */
  for (i = 1; i < argc; ++i)
    if (!strcmp(argv[i], "-p"))
      yydebug = 1;
  init_table ();
  return yyparse ();
}
```

This program is both powerful and flexible. You may easily add new functions, and it is a simple job to modify this code to install predefined variables such as `pi` or `e` as well.

2.6 Exercises

1. Add some new functions from `math.h` to the initialization list.
2. Add another array that contains constants and their values. Then modify `init_table` to add these constants to the symbol table. It will be easiest to give the constants type `VAR`.
3. Make the program report an error if the user refers to an uninitialized variable in any way except to store a value in it.