Go to the first, previous, next, last section, table of contents.

Examples

Now we show and explain three sample programs written using Bison: a reverse polish notation calculator, an algebraic (infix) notation calculator, and a multi-function calculator. All three have been tested under BSD Unix 4.3; each produces a usable, though limited, interactive desk-top calculator.

These examples are simple, but Bison grammars for real programming languages are written the same way.

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 input files.

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. */

%{
#define YYSTYPE double
#include <math.h>
%}

%token NUM

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

The C declarations section (see section The C Declarations Section) contains two preprocessor directives.

The #define directive defines the macro YYSTYPE, thus specifying the C data type for semantic values of both tokens and groupings (see section Data Types of Semantic Values). The Bison parser will use whatever type YYSTYPE 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.

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

The second section, Bison declarations, provides information to Bison about the token types (see section The Bison Declarations Section). Each terminal symbol that is not a single-character literal must be declared here. (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.

Grammar Rules for rpcalc

Here are the grammar rules for the reverse polish notation calculator.

```
input:    /* empty */
        | input line
;

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

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

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 `|` punctuator 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 Actions.

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.

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 Recursive Rules.

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 write the comment `/* empty */` in it.

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 file.

Explanation of `line`

Now consider the definition of `line`:

```
line:     '\n'
        | exp '\n'  { printf ("\t%.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.

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 Actions.

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 whitespace 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.

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 The Lexical Analyzer Function `yylex`.

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 the ASCII code for that 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`, which was defined at the beginning of the grammar; see section Declarations for `rpcalc`.)

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

Here is the code for the lexical analyzer:

```
/* Lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the ASCII
   character read if not a number.  Skips all blanks
   and tabs, returns 0 for EOF. */

#include <ctype.h>

yylex ()
{
  int c;

  /* skip white space  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* process numbers   */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* return end-of-file  */
  if (c == EOF)
    return 0;
  /* return single chars */
  return c;
```

```
}
```

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.

```
main ()
{
  yyparse ();
}
```

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 `"parse error"`). It is up to the programmer to supply `yyerror` (see section Parser C-Language Interface), so here is the definition we will use:

```
#include <stdio.h>

yyerror (s)  /* Called by yyparse on error */
     char *s;
{
  printf ("%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 section Error Recovery). 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 in the first example.

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 definitions of `yylex`, `yyerror` and `main` go at the end, in the "additional C code" section of the file (see section The Overall Layout of a Bison Grammar).

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

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

```
bison file_name.y
```

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

Compiling the Parser File

Here is how to compile and run the parser 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 rpcalc.tab.c -lm -o rpcalc

# 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
%
```

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--calc */

%{
#define YYSTYPE double
#include <math.h>
%}

/* BISON Declarations */
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */
%right '^'    /* exponentiation        */

/* Grammar follows */
%%
input:    /* empty string */
        | 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. (These tokens are single-character literals, which ordinarily don't need to be declared. We declare them here to specify the associativity.)

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. See section Operator Precedence.

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 Context-Dependent Precedence.

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

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

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 parse 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 section Error Recovery). 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.

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 non-number 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.

Declarations for `mfcalc`

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

```
%{
#include <math.h>  /* For math functions, cos(), sin(), etc. */
#include "calc.h"  /* Contains definition of `symrec'        */
%}
%union {
double     val;  /* For returning numbers.                   */
symrec  *tptr;   /* For returning symbol-table pointers      */
}

%token <val>  NUM        /* Simple double precision number   */
%token <tptr> VAR FNCT   /* Variable and Function            */
%type  <val>  exp

%right '='
%left '-' '+'
%left '*' '/'
%left NEG     /* Negation--unary minus */
```

```
%right '^'    /* Exponentiation        */

/* Grammar follows */

%%
```

The above grammar introduces only two new features of the Bison language. These features allow semantic values to have various data types (see section More Than One Value Type).

The `%union` declaration specifies the entire list of possible types; this is instead of defining `YYSTYPE`. The allowable types are now double-floats (for `exp` and `NUM`) and pointers to entries in the symbol table. See section The Collection of Value Types.

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 information about their data type (placed between angle brackets).

The Bison construct `%type` is used for declaring nonterminal symbols, just as `%token` is used for declaring token types. We have not used `%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 Nonterminal Symbols.

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.

```
input:   /* empty */
       | input line
;

line:
          '\n'
       | exp '\n'   { printf ("\t%.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 */
%%
```

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.

```
/* 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          */
    double (*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 ();
symrec *getsym ();
```

The new version of `main` includes a call to `init_table`, a function that initializes the symbol table. Here it is, and `init_table` as well:

```
#include <stdio.h>

main ()
{
  init_table ();
  yyparse ();
}

yyerror (s)  /* Called by yyparse on error */
     char *s;
{
  printf ("%s\n", s);
}

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

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

/* The symbol table: a chain of `struct symrec'.  */
symrec *sym_table = (symrec *)0;

init_table ()  /* puts arithmetic functions in table. */
{
  int i;
  symrec *ptr;
  for (i = 0; arith_fncts[i].fname != 0; i++)
```

```
    {
      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.

```
symrec *
putsym (sym_name,sym_type)
     char *sym_name;
     int sym_type;
{
  symrec *ptr;
  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 (sym_name)
     char *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;
}
```

The function `yylex` must now recognize variables, numeric values, and the single-character arithmetic operators. Strings of alphanumeric characters with a leading nondigit 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>
yylex ()
{
  int c;

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

  if (c == EOF)
```

```
    return 0;

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

  /* Char starts an identifier => read the name.       */
  if (isalpha (c))
    {
      symrec *s;
      static char *symbuf = 0;
      static int length = 0;
      int i;

      /* Initially make the buffer long enough
         for a 40-character symbol name.  */
      if (length == 0)
        length = 40, 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 (c != EOF && isalnum (c));

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

      s = getsym (symbuf);
      if (s == 0)
        s = putsym (symbuf, VAR);
      yylval.tptr = s;
      return s->type;
    }

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

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.

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.

Go to the first, previous, next, last section, table of contents.