BISON

The YACC-compatible Parser Generator

12 October 1988

by Charles Donnelly and Richard Stallman

Introduction

Bison is a general-purpose parser generator which converts a grammar description into a C program to parse that grammar. Once you are proficient with Bison, you may use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Bison is upward compatible with Yacc: all properly-written Yacc grammars ought to work with Bison with no change. Anyone familiar with Yacc should be able to use Bison with little trouble. You need to be fluent in C programming in order to use Bison or to understand this manual.

We begin with tutorial chapters that explain the basic concepts of using Bison and show three explained examples, each building on the last. If you don't know Bison or Yacc, start by reading these chapters. Reference chapters follow which describe specific aspects of Bison in detail.

Bison was basically written by Robert Corbett, and made Yacc-compatible by Richard Stallman.

Conditions for Using Bison

Bison grammars can be used only in programs that are free software. This is in contrast to what happens with the GNU C compiler and the other GNU programming tools.

The reason Bison is special is that the output of the Bison utility—the Bison parser file—contains a verbatim copy of a sizable piece of Bison, which is the code for the `yyparse` function. (The actions from your grammar are inserted into this function at one point, but the rest of the function is not changed.)

As a result, the Bison parser file is covered by the same copying conditions that cover Bison itself and the rest of the GNU system: any program containing it has to be distributed under the standard GNU copying conditions.

Occasionally people who would like to use Bison to develop proprietary programs complain about this.

We don't particularly sympathize with their complaints. The purpose of the GNU project is to promote the right to share software and the practice of sharing software; it is a means of changing society. The people who complain are planning to be uncooperative toward the rest of the world; why should they deserve our help in doing so?

However, it's possible that a change in these conditions might encourage computer companies to use and distribute the GNU system. If so, then we might decide to change the terms on `yyparse` as a matter of the strategy of promoting the right to share. Such a change would be irrevocable. Since we stand by the copying permissions we have announced, we cannot withdraw them once given.

We mustn't make an irrevocable change hastily. We have to wait until there is a complete GNU system and there has been time to learn how this issue affects its reception.

Bison General Public License

(Clarified 11 Feb 1988)

The license agreements of most software companies keep you at the mercy of those companies. By contrast, our general public license is intended to give everyone the right to share Bison. To make sure that you get the rights we want you to have, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. Hence this license agreement.

Specifically, we want to make sure that you have the right to give away copies of Bison, that you receive source code or else can get it if you want it, that you can change Bison or use pieces of it in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of Bison, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for Bison. If Bison is modified by someone else and passed on, we want its recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

Therefore we (Richard Stallman and the Free Software Foundation, Inc.) make the following terms which say what you must do to be allowed to distribute or change Bison.

Copying Policies

1. You may copy and distribute verbatim copies of Bison source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy a valid copyright notice "Copyright © 1988 Free Software Foundation, Inc." (or with whatever year is appropriate); keep intact the notices on all files that refer to this License Agreement and to the absence of any warranty; and give any other recipients of the Bison program a copy of this License Agreement along with the program. You may charge a distribution fee for the physical act of transferring a copy.
2. You may modify your copy or copies of Bison or any portion of it, and copy and distribute such modifications under the terms of Paragraph 1 above, provided that you also do the following:
   - cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and
   - cause the whole of any work that you distribute or publish, that in whole or in part contains or is a derivative of Bison or any part thereof, to be licensed at no charge to all third parties on terms identical to those contained in this License Agreement (except that you may choose to grant more extensive warranty protection to some or all third parties, at your option).
   - You may charge a distribution fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another unrelated program with this program (or its derivative) on a volume of a storage or distribution medium does not bring the other program under the scope of these terms.

3. You may copy and distribute Bison (or a portion or derivative of it, under Paragraph 2) in object code or executable form under the terms of Paragraphs 1 and 2 above provided that you also do one of the following:
   - accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal shipping charge) a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 1 and 2 above; or,
   - accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

   For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs.
4. You may not copy, sublicense, distribute or transfer Bison except as expressly provided under this License Agreement. Any attempt otherwise to copy, sublicense, distribute or transfer Bison is void and your rights to use the program under this License agreement shall be automatically terminated. However, parties who have received computer software programs from you with this License Agreement will not have their licenses terminated so long as such parties remain in full compliance.
5. If you wish to incorporate parts of Bison into other free programs whose distribution conditions are different, write to the Free Software Foundation at 675 Mass Ave, Cambridge, MA 02139. We have not yet worked out a simple rule that can be stated here, but we will often permit this. We will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software.

Your comments and suggestions about our licensing policies and our software are welcome! Please contact the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, or call (617) 876-3296.

NO WARRANTY

BECAUSE BISON IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY, TO THE EXTENT PERMITTED BY APPLICABLE STATE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, THE FREE SOFTWARE FOUNDATION, INC, RICHARD M. STALLMAN AND/OR OTHER PARTIES PROVIDE BISON "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF BISON IS WITH YOU. SHOULD BISON PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW WILL RICHARD M. STALLMAN, THE FREE SOFTWARE FOUNDATION, INC., AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE BISON AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS) BISON, EVEN IF YOU HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

1 The Concepts of Bison

This chapter introduces many of the basic concepts without which the details of Bison will not make sense. If you do not already know how to use Bison or Yacc, we suggest you start by reading this chapter carefully.

1.1 Languages and Context-Free Grammars

In order for Bison to parse a language, it must be described by a *context-free grammar*. This means that you specify one or more *syntactic groupings* and give rules for constructing them from their parts. For example, in the C language, one kind of grouping is called an 'expression'. One rule for making an expression might be, "An expression can be made of a minus sign and another expression". Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

The most common formal system for presenting such rules for humans to read is *Backus-Naur Form* or "BNF", which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF.

In the formal grammatical rules for a language, each kind of syntactic unit or grouping is named by a *symbol*. Those which are built by grouping smaller constructs according to grammatical rules are called *nonterminal symbols*; those which can't be subdivided are called *terminal symbols* or *token types*. We call a piece of input corresponding to a single terminal symbol a *token*, and a piece corresponding to a single nonterminal symbol a *grouping*.

We can use the C language as an example of what symbols, terminal and nonterminal, mean. The tokens of C are identifiers, constants (numeric and string), and the various keywords, arithmetic operators and punctuation marks. So the terminal symbols of a grammar for C include 'identifier', 'number', 'string', plus one symbol for each keyword, operator or punctuation mark: 'if', 'return', 'const', 'static', 'int', 'char', 'plus-sign', 'open-brace', 'close-brace', 'comma' and many more. (These tokens can be subdivided into characters, but that is a matter of lexicography, not grammar.)

Here is a simple C function subdivided into tokens:

```
int             /* keyword 'int' */
square (x)      /* identifier, open-paren, */
                /* identifier, close-paren */
     int x;     /* keyword 'int', identifier, semicolon */
{               /* open-brace */
  return x * x; /* keyword 'return', identifier, */
                /* asterisk, identifier, semicolon */
}               /* close-brace */
```

The syntactic groupings of C include the expression, the statement, the declaration, and the function definition. These are represented in the grammar of C by nonterminal symbols 'expression', 'statement', 'declaration' and 'function definition'. The full grammar uses dozens of additional language constructs, each with its own nonterminal symbol, in order to express the meanings of these four. The example above is a function definition;

it contains one declaration, and one statement. In the statement, each 'x' is an expression and so is 'x * x'.

Each nonterminal symbol must have grammatical rules showing how it is made out of simpler constructs. For example, one kind of C statement is the `return` statement; this would be described with a grammar rule which reads informally as follows:

> A 'statement' can be made of a 'return' keyword, an 'expression' and a 'semicolon'.

There would be many other rules for 'statement', one for each kind of statement in C.

One nonterminal symbol must be distinguished as the special one which defines a complete utterance in the language. It is called the *start symbol*. In a compiler, this means a complete input program. In the C language, the nonterminal symbol 'sequence of definitions and declarations' plays this role.

For example, '1 + 2' is a valid C expression—a valid part of a C program—but it is not valid as an *entire* C program. In the context-free grammar of C, this follows from the fact that 'expression' is not the start symbol.

The Bison parser reads a sequence of tokens as its input, and groups the tokens using the grammar rules. If the input is valid, the end result is that the entire token sequence reduces to a single grouping whose symbol is the grammar's start symbol. If we use a grammar for C, the entire input must be a 'sequence of definitions and declarations'. If not, the parser reports a syntax error.

1.2 From Formal Rules to Bison Input

A formal grammar is a mathematical construct. To define the language for Bison, you must write a file expressing the grammar in Bison syntax: a *Bison grammar* file. See Chapter 3 [Grammar File], page 31.

A nonterminal symbol in the formal grammar is represented in Bison input as an identifier, like an identifier in C. By convention, it should be in lower case, such as `expr`, `stmt` or `declaration`.

The Bison representation for a terminal symbol is also called a *token type*. Token types as well can be represented as C-like identifiers. By convention, these identifiers should be upper case to distinguish them from nonterminals: for example, `INTEGER`, `IDENTIFIER`, `IF` or `RETURN`. A terminal symbol that stands for a particular keyword in the language should be named after that keyword converted to upper case. The terminal symbol `error` is reserved for error recovery. See Section 3.2 [Symbols], page 32.

A terminal symbol can also be represented as a character literal, just like a C character constant. You should do this whenever a token is just a single character (parenthesis, plus-sign, etc.): use that same character in a literal as the terminal symbol for that token.

The grammar rules also have an expression in Bison syntax. For example, here is the Bison rule for a C `return` statement. The semicolon in quotes is a literal character token, representing part of the C syntax for the statement; the naked semicolon, and the colon, are Bison punctuation used in every rule.

```
stmt:   RETURN expr ';'
        ;
```

See Section 3.3 [Rules], page 33.

1.3 Semantic Values

A formal grammar selects tokens only by their classifications: for example, if a rule mentions the terminal symbol 'integer constant', it means that *any* integer constant is grammatically valid in that position. The precise value of the constant is irrelevant to how to parse the input: if '`x+4`' is grammatical then '`x+1`' or '`x+3989`' is equally grammatical.

But the precise value is very important for what the input means once it is parsed. A compiler is useless if it fails to distinguish between 4, 1 and 3989 as constants in the program! Therefore, each token in a Bison grammar has both a token type and a *semantic value*. See Section 3.5 [Semantics], page 34, for details.

The token type is a terminal symbol defined in the grammar, such as `INTEGER_CONSTANT`, `IDENTIFIER` or `','`. It tells everything you need to know to decide where the token may validly appear and how to group it with other tokens. The grammar rules know nothing about tokens except their types.

The semantic value has all the the rest of the information about the meaning of the token, such as the value of an integer, or the name of an identifier. (A token such as `','` which is just punctuation doesn't need to have any semantic value.)

For example, an input token might be classified as token type `INTEGER` and have the semantic value 4. Another input token might have the same token type `INTEGER` but value 3989. When a grammar rule says that `INTEGER` is allowed, either of these tokens is acceptable because each is an `INTEGER`. When the parser accepts the token, it keeps track of the token's semantic value.

Each grouping can also have a semantic value as well as its nonterminal symbol. For example, in a calculator, an expression typically has a semantic value that is a number. In a compiler for a programming language, an expression typically has a semantic value that is a tree structure describing the meaning of the expression.

1.4 Semantic Actions

In order to be useful, a program must do more than parse input; it must also produce some output based on the input. In a Bison grammar, a grammar rule can have an *action* made up of C statements. Each time the parser recognizes a match for that rule, the action is executed. See Section 3.5.3 [Actions], page 35.

Most of the time, the purpose of an action is to compute the semantic value of the whole construct from the semantic values of its parts. For example, suppose we have a rule which says an expression can be the sum of two expressions. When the parser recognizes such a sum, each of the subexpressions has a semantic value which describes how it was built up. The action for this rule should create a similar sort of value for the newly recognized larger expression.

For example, here is a rule that says an expression can be the sum of two subexpressions:

```
expr: expr '+' expr   { $$ = $1 + $3; }
        ;
```

The action says how to produce the semantic value of the sum expression from the values of the two subexpressions.

1.5 Bison Output: the Parser File

When you run Bison, you give it a Bison grammar file as input. The output is a C source file that parses the language described by the grammar. This file is called a *Bison parser*. Keep in mind that the Bison utility and the Bison parser are two distinct programs: the Bison utility is a program whose output is the Bison parser that becomes part of your program.

The job of the Bison parser is to group tokens into groupings according to the grammar rules—for example, to build identifiers and operators into expressions. As it does this, it runs the actions for the grammar rules it uses.

The tokens come from a function called the *lexical analyzer* that you must supply in some fashion (such as by writing it in C). The Bison parser calls the lexical analyzer each time it wants a new token. It doesn't know what is "inside" the tokens (though their semantic values may reflect this). Typically the lexical analyzer makes the tokens by parsing characters of text, but Bison does not depend on this. See Section 4.2 [Lexical], page 43.

The Bison parser file is C code which defines a function named `yyparse` which implements that grammar. This function does not make a complete C program: you must supply some additional functions. One is the lexical analyzer. Another is an error-reporting function which the parser calls to report an error. In addition, a complete C program must start with a function called `main`; you have to provide this, and arrange for it to call `yyparse` or the parser will never run. See Chapter 4 [Interface], page 43.

Aside from the token type names and the symbols in the actions you write, all variable and function names used in the Bison parser file begin with '`yy`' or '`YY`'. This includes interface functions such as the lexical analyzer function `yylex`, the error reporting function `yyerror` and the parser function `yyparse` itself. This also includes numerous identifiers used for internal purposes. Therefore, you should avoid using C identifiers starting with '`yy`' or '`YY`' in the Bison grammar file except for the ones defined in this manual.

1.6 Stages in Using Bison

The actual language-design process using Bison, from grammar specification to a working compiler or interpreter, has these parts:

1. Formally specify the grammar in a form recognized by Bison (see Chapter 3 [Grammar File], page 31). For each grammatical rule in the language, describe the action that is to be taken when an instance of that rule is recognized. The action is described by a sequence of C statements.
2. Write a lexical analyzer to process input and pass tokens to the parser. The lexical analyzer may be written by hand in C (see Section 4.2 [Lexical], page 43). It could also be produced using Lex, but the use of Lex is not discussed in this manual.
3. Write a controlling function that calls the Bison-produced parser.
4. Write error-reporting routines.

To turn this source code as written into a runnable program, you must follow these steps:

1. Run Bison on the grammar to produce the parser.
2. Compile the code output by Bison, as well as any other source files.
3. Link the object files to produce the finished product.

1.7 The Overall Layout of a Bison Grammar

The input file for the Bison utility is a *Bison grammar file*. The general form of a Bison grammar file is as follows:

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%
Additional C code
```

The '`%%`', '`%{`' and '`%}`' are punctuation that appears in every Bison grammar file to separate the sections.

The C declarations may define types and variables used in the actions. You can also use preprocessor commands to define macros used there, and use `#include` to include header files that do any of these things.

The Bison declarations declare the names of the terminal and nonterminal symbols, and may also describe operator precedence and the data types of semantic values of various symbols.

The grammar rules define how to construct each nonterminal symbol from its parts.

The additional C code can contain any C code you want to use. Often the definition of the lexical analyzer `yylex` goes here, plus subroutines called by the actions in the grammar rules. In a simple program, all the rest of the program can go here.

2 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.

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 input 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. */

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

%token NUM

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

The C declarations section (see Section 3.1.1 [C Declarations], page 31) 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 3.5.1 [Value Type], page 35). 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 3.1.2 [Bison Declarations], page 31). 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.

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 ("\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 3.5.3 [Actions], page 35.

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 3.4 [Recursion], page 34.

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.

2.1.2.2 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.

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 3.5.3 [Actions], page 35.

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.

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 4.2 [Lexical], page 43.

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 2.1.1 [Rpcalc Decls], page 15.)

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;

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

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.

```
main ()
{
  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 `"parse error"`). It is up to the programmer to supply `yyerror` (see Chapter 4 [Interface], page 43), 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 Chapter 6 [Error Recovery], page 55). 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.

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 definitions of `yylex`, `yyerror` and `main` go at the end, in the "additional C code" section of the file (see Section 1.7 [Grammar Layout], page 13).

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.

2.1.7 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
%
```

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--calc */

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

%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 lower the declaration, the higher the precedence. Hence, exponentiation has the highest precedence, unary minus (`NEG`) is next, followed by '`*`' and '`/`', and so on. See Section 5.3 [Precedence], page 49.

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 prece-

dence as `NEG`—in this case the next-to-highest. See Section 5.4 [Contextual Precedence], page 50.

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 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 Chapter 6 [Error Recovery], page 55). 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 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:

```
% acalc
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.

2.4.1 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 3.5.2 [Multiple Types], page 35).

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 3.6.3 [Union Decl], page 40.

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 3.6.4 [Type Decl], page 40.

2.4.2 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.

```
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 */
%%
```

2.4.3 Managing the Symbol Table for `mfcalc`

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.

2.5 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.

3 Bison Grammar Files

Bison takes as input a context-free grammar specification and produces a C-language function that recognizes correct instances of the grammar.

The Bison grammar input file conventionally has a name ending in '`.y`'.

3.1 Outline of a Bison Grammar

A Bison grammar file has four main sections, shown here with the appropriate delimiters:

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%

Additional C code
```

Comments enclosed in '`/* ... */`' may appear in any of the sections.

3.1.1 The C Declarations Section

The *C declarations* section contains macro definitions and declarations of functions and variables that are used in the actions in the grammar rules. These are copied to the beginning of the parser file so that they precede the definition of `yylex`. You can use '`#include`' to get the declarations from a header file. If you don't need any C declarations, you may omit the '`%{`' and '`%}`' delimiters that bracket this section.

3.1.2 The Bison Declarations Section

The *Bison declarations* section contains declarations that define terminal and nonterminal symbols, specify precedence, and so on. In some simple grammars you may not need any declarations. See Section 3.6 [Declarations], page 38.

3.1.3 The Grammar Rules Section

The *grammar rules* section contains one or more Bison grammar rules, and nothing else. See Section 3.3 [Rules], page 33.

There must always be at least one grammar rule, and the first '`%%`' (which precedes the grammar rules) may never be omitted even if it is the first thing in the file.

3.1.4 The Additional C Code Section

The *additional C code* section is copied verbatim to the end of the parser file, just as the *C declarations* section is copied to the beginning. This is the most convenient place to put anything that you want to have in the parser file but which need not come before the definition of `yylex`. For example, the definitions of `yylex` and `yyerror` often go here. See Chapter 4 [Interface], page 43.

If the last section is empty, you may omit the '`%%`' that separates it from the grammar rules.

The Bison parser itself contains many static variables whose names start with '`yy`' and many macros whose names start with '`YY`'. It is a good idea to avoid using any such names (except those documented in this manual) in the additional C code section of the grammar file.

3.2 Symbols, Terminal and Nonterminal

Symbols in Bison grammars represent the grammatical classifications of the language.

A *terminal symbol* (also known as a *token type*) represents a class of syntactically equivalent tokens. You use the symbol in grammar rules to mean that a token in that class is allowed. The symbol is represented in the Bison parser by a numeric code, and the `yylex` function returns a token type code to indicate what kind of token has been read. You don't need to know what the code value is; you can use the symbol to stand for it.

A *nonterminal symbol* stands for a class of syntactically equivalent groupings. The symbol name is used in writing grammar rules. By convention, it should be all lower case.

Symbol names can contain letters, digits (not at the beginning), underscores and periods. Periods make sense only in nonterminals.

There are two ways of writing terminal symbols in the grammar:

- A *named token type* is written with an identifier, like an identifier in C. By convention, it should be all upper case. Each such name must be defined with a Bison declaration such as `%token`. See Section 3.6.1 [Token Decl], page 39.
- A *character token type* (or *literal token*) is written in the grammar using the same syntax used in C for character constants; for example, `'+'` is a character token type. A character token type doesn't need to be declared unless you need to specify its semantic value data type (see Section 3.5.1 [Value Type], page 35), associativity, or precedence (see Section 5.3 [Precedence], page 49).

  By convention, a character token type is used only to represent a token that consists of that particular character. Thus, the token type `'+'` is used to represent the character '`+`' as a token. Nothing enforces this convention, but if you depart from it, your program will confuse other readers.

  All the usual escape sequences used in character literals in C can be used in Bison as well, but you must not use the null character as a character literal because its ASCII code, zero, is the code `yylex` returns for end-of-input (see Section 4.2 [Lexical], page 43).

How you choose to write a terminal symbol has no effect on its grammatical meaning. That depends only on where it appears in rules and on when the parser function returns that symbol.

The value returned by `yylex` is always one of the terminal symbols (or 0 for end-of-input). Whichever way you write the token type in the grammar rules, you write it the same way in the definition of `yylex`. The numeric code for a character token type is simply the ASCII code for the character, so `yylex` can use the identical character constant to generate the requisite code. Each named token type becomes a C macro in the parser file,

so `yylex` can use the name to stand for the code. (This is why periods don't make sense in terminal symbols.) See Section 4.2 [Lexical], page 43.

If `yylex` is defined in a separate file, you need to arrange for the token-type macro definitions to be available there. Use the '`-d`' option when you run Bison, so that it will write these macro definitions into a separate header file *`name`*`.tab.h` which you can include in the other source files that need it. See Chapter 8 [Invocation], page 59.

The symbol `error` is a terminal symbol reserved for error recovery (see Chapter 6 [Error Recovery], page 55); you shouldn't use it for any other purpose. In particular, `yylex` should never return this value.

3.3 Syntax of Grammar Rules

A Bison grammar rule has the following general form:

```
result: components...
        ;
```

where *result* is the nonterminal symbol that this rule describes and *components* are various terminal and nonterminal symbols that are put together by this rule (see Section 3.2 [Symbols], page 32). For example,

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

says that two groupings of type `exp`, with a '`+`' token in between, can be combined into a larger grouping of type `exp`.

Whitespace in rules is significant only to separate symbols. You can add extra whitespace as you wish.

Scattered among the components can be *actions* that determine the semantics of the rule. An action looks like this:

```
{C statements}
```

Usually there is only one action and it follows the components. See Section 3.5.3 [Actions], page 35.

Multiple rules for the same *result* can be written separately or can be joined with the vertical-bar character '`|`' as follows:

```
result:    rule1-components...
        | rule2-components...
        ...
        ;
```

They are still considered distinct rules even when joined in this way.

If *components* in a rule is empty, it means that *result* can match the empty string. For example, here is how to define a comma-separated sequence of zero or more `exp` groupings:

```
expseq:   /* empty */
        | expseq1
        ;

expseq1:  exp
        | expseq1 ',' exp
```

```
        ;
```

It is customary to write a comment '`/* empty */`' in each rule with no components.

3.4 Recursive Rules

A rule is called *recursive* when its *result* nonterminal appears also on its right hand side. Nearly all Bison grammars need to use recursion, because that is the only way to define a sequence of any number of somethings. Consider this recursive definition of a comma-separated sequence of one or more expressions:

```
expseq1:  exp
        | expseq1 ',' exp
        ;
```

Since the recursive use of `expseq1` is the leftmost symbol in the right hand side, we call this *left recursion*. By contrast, here the same construct is defined using *right recursion*:

```
expseq1:  exp
        | exp ',' expseq1
        ;
```

Any kind of sequence can be defined using either left recursion or right recursion, but you should always use left recursion, because it can parse a sequence of any number of elements with bounded stack space. Right recursion uses up space on the Bison stack in proportion to the number of elements in the sequence, because all the elements must be shifted onto the stack before the rule can be applied even once. See Chapter 5 [Algorithm], page 47, for further explanation of this.

Indirect or *mutual* recursion occurs when the result of the rule does not appear directly on its right hand side, but does appear in rules for other nonterminals which do appear on its right hand side. For example:

```
expr:     primary
        | primary '+' primary
        ;

primary:  constant
        | '(' expr ')'
        ;
```

defines two mutually-recursive nonterminals, since each refers to the other.

3.5 The Semantics of the Language

The grammar rules for a language determine only the syntax. The semantics are determined by the semantic values associated with various tokens and groupings, and by the actions taken when various groupings are recognized.

For example, the calculator calculates properly because the value associated with each expression is the proper number; it adds properly because the action for the grouping '`x + y`' is to add the numbers associated with x and y.

3.5.1 The Data Types of Semantic Values

In a simple program it may be sufficient to use the same data type for the semantic values of all language constructs. This was true in the RPN and infix calculator examples (see Section 2.1 [RPN Calc], page 15).

Bison's default is to use type `int` for all semantic values. To specify some other type, define `YYSTYPE` as a macro, like this:

```
#define YYSTYPE double
```

This macro definition must go in the C declarations section of the grammar file (see Section 3.1 [Grammar Outline], page 31).

3.5.2 More Than One Type for Semantic Values

In most programs, you will need different data types for different kinds of tokens and groupings. For example, a numeric constant may need type `int` or `long`, while a string constant needs type `char *`, and an identifier might need a pointer to an entry in the symbol table.

To use more than one data type for semantic values in one parser, Bison requires you to do two things:

- Specify the entire collection of possible data types, with the `%union` Bison declaration (see Section 3.6.3 [Union Decl], page 40).
- Choose one of those types for each symbol (terminal or nonterminal) for which semantic values are used. This is done for tokens with the `%token` Bison declaration (see Section 3.6.1 [Token Decl], page 39) and for groupings with the `%type` Bison declaration (see Section 3.6.4 [Type Decl], page 40).

3.5.3 Actions

An action accompanies a syntactic rule and contains C code to be executed each time an instance of that rule is recognized. The task of most actions is to compute a semantic value for the grouping built by the rule from the semantic values associated with tokens or smaller groupings.

An action consists of C statements surrounded by braces, much like a compound statement in C. It can be placed at any position in the rule; it is executed at that position. Most rules have just one action at the end of the rule, following all the components. Actions in the middle of a rule are tricky and used only for special purposes (see Section 3.5.5 [Mid-Rule Actions], page 36).

The C code in an action can refer to the semantic values of the components matched by the rule with the construct `$`*n*, which stands for the value of the *n*th component. The semantic value for the grouping being constructed is `$$`. (Bison translates both of these constructs into array element references when it copies the actions into the parser file.)

Here is a typical example:

```
exp:    ...
        | exp '+' exp
            { $$ = $1 + $3; }
```

This rule constructs an `exp` from two smaller `exp` groupings connected by a plus-sign token. In the action, `$1` and `$3` refer to the semantic values of the two component `exp` groupings,

which are the first and third symbols on the right hand side of the rule. The sum is stored into `$$` so that it becomes the semantic value of the addition-expression just recognized by the rule. If there were a useful semantic value associated with the '`+`' token, it could be referred to as `$2`.

`$`*n* with *n* zero or negative is allowed for reference to tokens and groupings on the stack *before* those that match the current rule. This is a very risky practice, and to use it reliably you must be certain of the context in which the rule is applied. Here is a case in which you can use this reliably:

```
foo:      expr bar '+' expr  { ... }
        | expr bar '-' expr  { ... }
        ;

bar:      /* empty */
        { previous_expr = $0; }
        ;
```

As long as `bar` is used only in the fashion shown here, `$0` always refers to the `expr` which precedes `bar` in the definition of `foo`.

3.5.4 Data Types of Values in Actions

If you have chosen a single data type for semantic values, the `$$` and `$`*n* constructs always have that data type.

If you have used `%union` to specify a variety of data types, then you must declare a choice among these types for each terminal or nonterminal symbol that can have a semantic value. Then each time you use `$$` or `$`*n*, its data type is determined by which symbol it refers to in the rule. In this example,

```
exp:    ...
        | exp '+' exp
            { $$ = $1 + $3; }
```

`$3` and `$$` refer to instances of `exp`, so they all have the data type declared for the nonterminal symbol `exp`. If `$2` were used, it would have the data type declared for the terminal symbol `'+'`, whatever that might be.

Alternatively, you can specify the data type when you refer to the value, by inserting '`<`*type*`>`' after the '`$`' at the beginning of the reference. For example, if you have defined types as shown here:

```
%union {
  int itype;
  double dtype;
}
```

then you can write `$<itype>1` to refer to the first subunit of the rule as an integer, or `$<dtype>1` to refer to it as a double.

3.5.5 Actions in Mid-Rule

Occasionally it is useful to put an action in the middle of a rule. These actions are written just like usual end-of-rule actions, but they are executed before the parser even recognizes the following components.

A mid-rule action may refer to the components preceding it using `$n`, but it may not refer to subsequent components because it is run before they are parsed.

The mid-rule action itself counts as one of the components of the rule. This makes a difference when there is another action later in the same rule (and usually there is another at the end): you have to count the actions along with the symbols when working out which number *n* to use in `$n`.

The mid-rule action can also have a semantic value. This can be set within that action by an assignment to `$$`, and can referred to by later actions using `$n`. Since there is no symbol to name the action, there is no way to declare a data type for the value in advance, so you must use the '`$<...>`' construct to specify a data type each time you refer to this value.

Here is an example from a hypothetical compiler, handling a `let` statement that looks like '`let (variable) statement`' and serves to create a variable named *variable* temporarily for the duration of *statement*. To parse this construct, we must put *variable* into the symbol table while *statement* is parsed, then remove it afterward. Here is how it is done:

```
stmt:   LET '(' var ')'
                { $<context>$ = push_context ();
                  declare_variable ($3); }
        stmt    { $$ = $6;
                  pop_context ($<context>5); }
```

As soon as '`let (variable)`' has been recognized, the first action is run. It saves a copy of the current semantic context (the list of accessible variables) as its semantic value, using alternative `context` in the data-type union. Then it calls `declare_variable` to add the new variable to that list. Once the first action is finished, the embedded statement `stmt` can be parsed. Note that the mid-rule action is component number 5, so the '`stmt`' is component number 6.

After the embedded statement is parsed, its semantic value becomes the value of the entire `let`-statement. Then the semantic value from the earlier action is used to restore the prior list of variables. This removes the temporary `let`-variable from the list so that it won't appear to exist while the rest of the program is parsed.

Taking action before a rule is completely recognized often leads to conflicts since the parser must commit to a parse in order to execute the action. For example, the following two rules, without mid-rule actions, can coexist in a working parser because the parser can shift the open-brace token and look at what follows before deciding whether there is a declaration or not:

```
compound: '{' declarations statements '}'
        | '{' statements '}'
        ;
```

But when we add a mid-rule action as follows, the rules become nonfunctional:

```
compound: { prepare_for_local_variables (); }
          '{' declarations statements '}'
        | '{' statements '}'
        ;
```

Now the parser is forced to decide whether to run the mid-rule action when it has read no farther than the open-brace. In other words, it must commit to using one rule or the other,

without sufficient information to do it correctly. (The open-brace token is what is called the *look-ahead* token at this time, since the parser is still deciding what to do about it. See Section 5.1 [Look-Ahead], page 47.)

You might think that you could correct the problem by putting identical actions into the two rules, like this:

```
compound: { prepare_for_local_variables (); }
          '{' declarations statements '}'
        | { prepare_for_local_variables (); }
          '{' statements '}'
        ;
```

But this does not help, because Bison does not realize that the two actions are identical. (Bison never tries to understand the C code in an action.)

If the grammar is such that a declaration can be distinguished from a statement by the first token (which is true in C), then one solution which does work is to put the action after the open-brace, like this:

```
compound: '{' { prepare_for_local_variables (); }
          declarations statements '}'
        | '{' statements '}'
        ;
```

Now the first token of the following declaration or statement, which would in any case tell Bison which rule to use, can still do so.

Another solution is to bury the action inside a nonterminal symbol which serves as a subroutine:

```
subroutine: /* empty */
          { prepare_for_local_variables (); }
        ;

compound: subroutine
          '{' declarations statements '}'
        | subroutine
          '{' statements '}'
        ;
```

Now Bison can execute the action in the rule for `subroutine` without deciding which rule for `compound` it will eventually use. Note that the action is now at the end of its rule. Any mid-rule action can be converted to an end-of-rule action in this way, and this is what Bison actually does to implement mid-rule actions.

3.6 Bison Declarations

The *Bison declarations* section of a Bison grammar defines the symbols used in formulating the grammar and the data types of semantic values. See Section 3.2 [Symbols], page 32.

All token type names (but not single-character literal tokens such as `'+'` and `'*'`) must be declared. Nonterminal symbols must be declared if you need to specify which data type to use for the semantic value (see Section 3.5.2 [Multiple Types], page 35).

The first rule in the file also specifies the start symbol, by default. If you want some other symbol to be the start symbol, you must declare it explicitly (see Section 1.1 [Language and Grammar], page 9).

3.6.1 Declaring Token Type Names

The basic way to declare a token type name (terminal symbol) is as follows:

```
%token name
```

Bison will convert this into a `#define` directive in the parser, so that the function `yylex` (if it is in this file) can use the name *name* to stand for this token type's code.

Alternatively you can use `%left`, `%right`, or `%nonassoc` instead of `%token`, if you wish to specify precedence. See Section 3.6.2 [Precedence Decl], page 39.

You can explicitly specify the numeric code for a token type by appending an integer value in the field immediately following the token name:

```
%token NUM 300
```

It is generally best, however, to let Bison choose the numeric codes for all token types. Bison will automatically select codes that don't conflict with each other or with ASCII characters.

In the event that the stack type is a union, you must augment the `%token` or other token declaration to include the data type alternative delimited by angle-brackets (see Section 3.5.2 [Multiple Types], page 35). For example:

```
%union {              /* define stack type */
  double val;
  symrec *tptr;
}
%token <val> NUM      /* define token NUM and its type */
```

3.6.2 Declaring Operator Precedence

Use the `%left`, `%right` or `%nonassoc` declaration to declare a token and specify its precedence and associativity, all at once. These are called *precedence declarations*. See Section 5.3 [Precedence], page 49, for general information on operator precedence.

The syntax of a precedence declaration is the same as that of `%token`: either

```
%left symbols...
```

or

```
%left <type> symbols...
```

And indeed any of these declarations serves the purposes of `%token`. But in addition, they specify the associativity and relative precedence for all the *symbols*:

- The associativity of an operator *op* determines how repeated uses of the operator nest: whether '*x op y op z*' is parsed by grouping *x* with *y* first or by grouping *y* with *z* first. `%left` specifies left-associativity (grouping *x* with *y* first) and `%right` specifies right-associativity (grouping *y* with *z* first). `%nonassoc` specifies no associativity, which means that '*x op y op z*' is considered a syntax error.
- The precedence of an operator determines how it nests with other operators. All the tokens declared in a single precedence declaration have equal precedence and nest

together according to their associativity. When two tokens declared in different precedence declarations associate, the one declared later has the higher precedence and is grouped first.

3.6.3 Declaring the Collection of Value Types

The `%union` declaration specifies the entire collection of possible data types for semantic values. The keyword `%union` is followed by a pair of braces containing the same thing that goes inside a `union` in C. For example:

```
%union {
  double val;
  symrec *tptr;
}
```

This says that the two alternative types are `double` and `symrec *`. They are given names `val` and `tptr`; these names are used in the `%token` and `%type` declarations to pick one of the types for a terminal or nonterminal symbol (see Section 3.6.4 [Type Decl], page 40).

Note that, unlike making a `union` declaration in C, you do not write a semicolon after the closing brace.

3.6.4 Declaring Value Types of Nonterminal Symbols

When you use `%union` to specify multiple value types, you must declare the value type of each nonterminal symbol for which values are used. This is done with a `%type` declaration, like this:

```
%type <type> nonterminal...
```

Here *nonterminal* is the name of a nonterminal symbol, and *type* is the name given in the `%union` to the alternative that you want (see Section 3.6.3 [Union Decl], page 40). You can give any number of nonterminal symbols in the same `%type` declaration, if they have the same value type. Use spaces to separate the symbol names.

3.6.5 Preventing Warnings about Conflicts

Bison normally warns if there are any conflicts in the grammar (see Section 5.2 [Shift/Reduce], page 48), but most real grammars have harmless shift/reduce conflicts which are resolved in a predictable way and would be difficult to eliminate. It is desirable to suppress the warning about these conflicts unless the number of conflicts changes. You can do this with the `%expect` declaration.

The declaration looks like this:

```
%expect n
```

Here *n* is a decimal integer. The declaration says there should be no warning if there are *n* shift/reduce conflicts and no reduce/reduce conflicts. The usual warning is given if there are either more or fewer conflicts, or if there are any reduce/reduce conflicts.

In general, using `%expect` involves these steps:

- Compile your grammar without `%expect`. Use the '`-v`' option to get a verbose list of where the conflicts occur. Bison will also print the number of conflicts.
- Check each of the conflicts to make sure that Bison's default resolution is what you really want. If not, rewrite the grammar and go back to the beginning.

- Add an `%expect` declaration, copying the number *n* from the number which Bison printed.

Now Bison will stop annoying you about the conflicts you have checked, but it will warn you again if changes in the grammer result in additional conflicts.

3.6.6 Declaring the Start-Symbol

Bison assumes by default that the start symbol for the grammar is the first nonterminal specified in the grammar specification section. The programmer may override this restriction with the `%start` declaration as follows:

```
%start symbol
```

3.6.7 Requesting a Pure (Reentrant) Parser

A reentrant program is one which does not alter in the course of execution. Reentrancy is important whenever asynchronous execution is possible; for example, a nonreentrant program may not be safe to call from a signal handler. In systems with multiple threads of control, a nonreentrant program must be called only within interlocks.

The Bison parser is not normally a reentrant program, because it uses two statically allocated variables for communication with `yylex`. These variables are `yylval` and `yylloc`.

The Bison declaration `%pure_parser` says that you want the parser to be reentrant. It looks like this:

```
%pure_parser
```

The effect is that the the two communication variables become automatic variables in `yyparse`, and a different calling convention is used for the lexical analyzer function `yylex`. The convention for calling `yyparse` is unchanged. See Section 4.2 [Lexical], page 43, for the details of this.

3.6.8 Bison Declaration Summary

Here is a summary of all Bison declarations:

`%union` Declare the collection of data types that semantic values may have (see Section 3.6.3 [Union Decl], page 40).

`%token` Declare a terminal symbol (token type name) with no precedence or associativity specified (see Section 3.6.1 [Token Decl], page 39).

`%right` Declare a terminal symbol (token type name) that is right-associative (see Section 3.6.2 [Precedence Decl], page 39).

`%left` Declare a terminal symbol (token type name) that is left-associative (see Section 3.6.2 [Precedence Decl], page 39).

`%nonassoc`
Declare a terminal symbol (token type name) that is nonassociative (using it in a way that would be associative is a syntax error) (see Section 3.6.2 [Precedence Decl], page 39).

`%type` Declare the type of semantic values for a nonterminal symbol (see Section 3.6.4 [Type Decl], page 40).

`%start` Specify the grammar's start symbol (see Section 3.6.6 [Start Decl], page 41).

`%expect` Declare the expected number of shift-reduce conflicts (see Section 3.6.5 [Expect Decl], page 40).

`%pure_parser`
Request a pure (reentrant) parser program (see Section 3.6.7 [Pure Decl], page 41).

4 Parser C-Language Interface

The Bison parser is actually a C function named `yyparse`. Here we describe the interface conventions of `yyparse` and the other functions that it needs to use.

Keep in mind that the parser uses many C identifiers starting with '`yy`' and '`YY`' for internal purposes. If you use such an identifier (aside from those in this manual) in an action or in additional C code in the grammar file, you are likely to run into trouble.

4.1 The Parser Function `yyparse`

You call the function `yyparse` to cause parsing to occur. This function reads tokens, executes actions, and ultimately returns when it encounters end-of-input or an unrecoverable syntax error. You can also write an action which directs `yyparse` to return immediately without reading further.

The value returned by `yyparse` is 0 if parsing was successful (return is due to end-of-input).

The value is 1 if parsing failed (return is due to a syntax error).

In an action, you can cause immediate return from `yyparse` by using these macros:

`YYACCEPT` Return immediately with value 0 (to report success).

`YYABORT` Return immediately with value 1 (to report failure).

4.2 The Lexical Analyzer Function `yylex`

The *lexical analyzer* function, `yylex`, recognizes tokens from the input stream and returns them to the parser. Bison does not create this function automatically; you must write it so that `yyparse` can call it. The function is sometimes referred to as a lexical scanner.

The value that `yylex` returns must be the numeric code for the type of token it has just found, or 0 for end-of-input. See Section 3.2 [Symbols], page 32.

When a token is referred to in the grammar rules by a name, that name in the parser file becomes a C macro whose definition is the proper numeric code for that token type. So `yylex` can use the name to indicate that type.

When a token is referred to in the grammar rules by a character literal, the numeric code for that character is also the code for the token type. So `yylex` can simply return that character code. The null character must not be used this way, because its code is zero and that is what signifies end-of-input.

Here is an example showing these things:

```
yylex()
{
  ...
  if (c == EOF)     /* Detect end of file.  */
    return 0;
  ...
  if (c == '+' || c == '-')
    return c;      /* Assume token type for '+' is '+'.  */
  ...
```

```
  return INT;      /* Return the type of the token.  */
  ...
}
```

This interface has been designed so that the output from the `lex` utility can be used without change as the definition of `yylex`.

In simple programs, `yylex` is often defined at the end of the Bison grammar file. If `yylex` is defined in a separate source file, you need to arrange for the token-type macro definitions to be available there. To do this, use the '`-d`' option when you run Bison, so that it will write these macro definitions into a separate header file *`name`*`.tab.h` which you can include in the other source files that need it. See Chapter 8 [Invocation], page 59.

In an ordinary (nonreentrant) parser, the semantic value of the token must be stored into the global variable `yylval`. When you are using just one data type for semantic values, `yylval` has that type. Thus, if the type is `int` (the default), you might write this in `yylex`:

```
  ...
  yylval = value;  /* Put value onto Bison stack.  */
  return INT;      /* Return the type of the token.  */
  ...
```

When you are using multiple data types, `yylval`'s type is a union made from the `%union` declaration (see Section 3.6.3 [Union Decl], page 40). So when you store a token's value, you must use the proper member of the union. If the `%union` declaration looks like this:

```
%union {
  int intval;
  double val;
  symrec *tptr;
}
```

then the code in `yylex` might look like this:

```
  ...
  yylval.intval = value;  /* Put value onto Bison stack.  */
  return INT;      /* Return the type of the token.  */
  ...
```

If you are using the '`@`*`n`*'-feature (see Section 4.4 [Action Features], page 45) in actions to keep track of the textual locations of tokens and groupings, then you must provide this information in `yylex`. The function `yyparse` expects to find the textual location of a token just parsed in the global variable `yylloc`. So `yylex` must store the proper data in that variable. The value of `yylloc` is a structure, and you need only initialize the members that are going to be used by the actions. The four members are called `first_line`, `first_column`, `last_line` and `last_column`. Note that the use of this feature makes the parser noticeably slower.

When you use the Bison declaration `%pure_parser` to request a pure, reentrant parser, the global communication variables `yylval` and `yylloc` cannot be used. (See Section 3.6.7 [Pure Decl], page 41.) In such parsers the two global variables are replaced by pointers passed as arguments to `yylex`. You must declare them as shown here, and pass the information back by storing it through those pointers.

```
yylex (lvalp, llocp)
```

```
     YYSTYPE *lvalp;
     YYLTYPE *llocp;
{
  ...
  *lvalp = value;  /* Put value onto Bison stack.  */
  return INT;      /* Return the type of the token.  */
  ...
}
```

4.3 The Error Reporting Function `yyerror`

The Bison parser expects to call an error reporting function named `yyerror`, which is supplied by you. It is called by `yyparse` whenever a syntax error is found, and it receives one argument, a string which can be `"parse error"` or `"parser stack overflow"`. (The latter is unlikely, since you have to work really hard to overflow the automatically-extended Bison parser stack.)

The following definition suffices in simple programs:

```
yyerror (s)
     char *s;
{
  fprintf (stderr, "%s\n", s);
}
```

After `yyerror` returns to `yyparse`, the latter will attempt error recovery if you have written suitable error recovery grammar rules (see Chapter 6 [Error Recovery], page 55). If recovery is impossible, `yyparse` will immediately return 1.

The global variable `yynerr` contains the number of syntax errors encountered so far.

4.4 Special Features for Use in Actions

Here is a table of Bison constructs, variables and macros that are useful in actions.

‘`$$`’ Acts like a variable that contains the semantic value for the grouping made by the current rule. See Section 3.5.3 [Actions], page 35.

‘`$`*n*’ Acts like a variable that contains the semantic value for the *n*th component of the current rule. See Section 3.5.3 [Actions], page 35.

‘`$<`*typealt*`>$`’
Like `$$` but specifies alternative *typealt* in the union specified by the `%union` declaration. See Section 3.5.4 [Action Types], page 36.

‘`$<`*typealt*`>`*n*’
Like `$`*n* but specifies alternative *typealt* in the union specified by the `%union` declaration. See Section 3.5.4 [Action Types], page 36.

‘`YYABORT;`’
Return immediately from `yyparse`, indicating failure. See Section 4.1 [Parser Function], page 43.

'`YYACCEPT;`'
: Return immediately from `yyparse`, indicating success. See Section 4.1 [Parser Function], page 43.

'`YYEMPTY`' Value stored in `yychar` when there is no look-ahead token.

'`YYERROR;`'
: Cause an immediate syntax error. This causes `yyerror` to be called, and then error recovery begins. See Chapter 6 [Error Recovery], page 55.

'`yychar`' Variable containing the current look-ahead token. When there is no look-ahead token, the value `YYERROR` is stored here. See Section 5.1 [Look-Ahead], page 47.

'`yyclearin;`'
: Discard the current look-ahead token. This is useful primarily in error rules. See Chapter 6 [Error Recovery], page 55.

'`yyerrok;`'
: Resume generating error messages immediately for subsequent syntax errors. This is useful primarily in error rules. See Chapter 6 [Error Recovery], page 55.

'`@n`' Acts like a structure variable containing information on the line numbers and column numbers of the *n*th component of the current rule. The structure has four members, like this:

```
struct {
  int first_line, last_line;
  int first_column, last_column;
};
```

Thus, to get the starting line number of the third component, use '`@3.first_line`'.

In order for the members of this structure to contain valid information, you must make `yylex` supply this information about each token. If you need only certain members, then `yylex` need only fill in those members.

The use of this feature makes the parser noticeably slower.

5 The Algorithm of the Bison Parser

As Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the *parser stack*. Pushing a token is traditionally called *shifting*.

For example, suppose the infix calculator has read '`1 + 5 *`', with a '`3`' to come. The stack will have four elements, one for each token that was shifted.

But the stack does not always have an element for each token read. When the last *n* tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called *reduction*. Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule. Running the rule's action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping.

For example, if the infix calculator's parser stack contains this:

```
1 + 5 * 3
```

and the next input token is a newline character, then the last three elements can be reduced to 15 via the rule:

```
expr: expr '*' expr;
```

Then the stack contains just these three elements:

```
1 + 15
```

At this point, another reduction can be made, resulting in the single value 16. Then the newline token can be shifted.

The parser tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol (see Section 1.1 [Language and Grammar], page 9).

This kind of parser is known in the literature as a bottom-up parser.

5.1 Look-Ahead Tokens

The Bison parser does *not* always reduce immediately as soon as the last *n* tokens and groupings match a rule. This is because such a simple strategy is inadequate to handle most languages. Instead, when a reduction is possible, the parser sometimes "looks ahead" at the next token in order to decide what to do.

When a token is read, it is not immediately shifted; first it becomes the *look-ahead token*, which is not on the stack. Now the parser can perform one or more reductions of tokens and groupings on the stack, while the look-ahead token remains off to the side. When no more reductions should take place, the look-ahead token is shifted onto the stack. This does not mean that all possible reductions have been done; depending on the token type of the look-ahead token, some rules may choose to delay their application.

Here is a simple case where look-ahead is needed. These three rules define expressions which contain binary addition operators and postfix unary factorial operators ('`!`'), and allow parentheses for grouping.

```
expr:     term '+' expr
        | term
        ;
```

```
term:     '(' expr ')'
        | term '!'
        | NUMBER
        ;
```

Suppose that the tokens '`1 + 2`' have been read and shifted; what should be done? If the following token is '`)`', then the first three tokens must be reduced to form an `expr`. This is the only valid course, because shifting the '`)`' would produce a sequence of symbols `term ')'`, and no rule allows this.

If the following token is '`!`', then it must be shifted immediately so that '`2 !`' can be reduced to make a `term`. If instead the parser were to reduce before shifting, '`1 + 2`' would become an `expr`. It would then be impossible to shift the '`!`' because doing so would produce on the stack the sequence of symbols `expr '!'`. No rule allows that sequence.

The current look-ahead token is stored in the variable `yychar`. See Section 4.4 [Action Features], page 45.

5.2 Shift/Reduce Conflicts

Suppose we are parsing a language which has if-then and if-then-else statements, with a pair of rules like this:

```
if_stmt:
          IF expr THEN stmt
        | IF expr THEN stmt ELSE stmt
        ;
```

(Here we assume that `IF`, `THEN` and `ELSE` are terminal symbols for specific keyword tokens.)

When the `ELSE` token is read and becomes the look-ahead token, the contents of the stack (assuming the input is valid) are just right for reduction by the first rule. But it is also legitimate to shift the `ELSE`, because that would lead to eventual reduction by the second rule.

This situation, where either a shift or a reduction would be valid, is called a *shift/reduce conflict*. Bison is designed to resolve these conflicts by choosing to shift, unless otherwise directed by operator precedence declarations. To see the reason for this, let's contrast it with the other alternative.

Since the parser prefers to shift the `ELSE`, the result is to attach the else-clause to the innermost if-statement, making these two inputs equivalent:

```
if x then if y then win(); else lose;

if x then do; if y then win(); else lose; end;
```

But if the parser chose to reduce when possible rather than shift, the result would be to attach the else-clause to the outermost if-statement, making these two inputs equivalent:

```
if x then if y then win(); else lose;

if x then do; if y then win(); end; else lose;
```

The conflict exists because the grammar as written is ambiguous: either parsing of the simple nested if-statement is legitimate. The established convention is that these ambiguities

are resolved by attaching the else-clause to the innermost if-statement; this is what Bison accomplishes by choosing to shift rather than reduce. (It would ideally be cleaner to write an unambiguous grammar, but that is very hard to do in this case.) This particular ambiguity was first encountered in the specifications of Algol 60 and is called the "dangling `else`" ambiguity.

To avoid warnings from Bison about predictable, legitimate shift/reduce conflicts, use the `%expect` *n* declaration. There will be no warning as long as the number of shift/reduce conflicts is exactly *n*. See Section 3.6.5 [Expect Decl], page 40.

5.3 Operator Precedence

Another situation where shift/reduce conflicts appear is in arithmetic expressions. Here shifting is not always the preferred resolution; the Bison declarations for operator precedence allow you to specify when to shift and when to reduce.

5.3.1 When Precedence is Needed

Consider the following ambiguous grammar fragment (ambiguous because the input '`1 - 2 * 3`' can be parsed in two different ways):

```
expr:     expr '-' expr
        | expr '*' expr
        | expr '<' expr
        | '(' expr ')'
        ...
        ;
```

Suppose the parser has seen the tokens '`1`', '`-`' and '`2`'; should it reduce them via the rule for the addition operator? It depends on the next token. Of course, if the next token is '`)`', we must reduce; shifting is invalid because no single rule can reduce the token sequence '`- 2 )`' or anything starting with that. But if the next token is '`*`' or '`<`', we have a choice: either shifting or reduction would allow the parse to complete, but with different results.

To decide which one Bison should do, we must consider the results. If the next operator token *op* is shifted, then it must be reduced first in order to permit another opportunity to reduce the sum. The result is (in effect) '`1 - (2 op 3)`'. On the other hand, if the subtraction is reduced before shifting *op*, the result is '`(1 - 2) op 3`'. Clearly, then, the choice of shift or reduce should depend on the relative precedence of the operators '`-`' and *op*: '`*`' should be shifted first, but not '`<`'.

What about input like '`1 - 2 - 5`'; should this be '`(1 - 2) - 5`' or '`1 - (2 - 5)`'? For most operators we prefer the former, which is called *left association*. The latter alternative, *right association*, is desirable for assignment operators. The choice of left or right association is a matter of whether the parser chooses to shift or reduce when the stack contains '`1 - 2`' and the look-ahead token is '`-`': shifting makes right-associativity.

5.3.2 How to Specify Operator Precedence

Bison allows you to specify these choices with the operator precedence declarations `%left` and `%right`. Each such declaration contains a list of tokens, which are operators whose precedence and associativity is being declared. The `%left` declaration makes all those operators left-associative and the `%right` declaration makes them right-associative. A third

alternative is `%nonassoc`, which declares that it is a syntax error to find the same operator twice "in a row".

The relative precedence of different operators is controlled by the order in which they are declared. The first `%left` or `%right` declaration declares the operators whose precedence is lowest, the next such declaration declares the operators whose precedence is a little higher, and so on.

5.3.3 Precedence Examples

In our example, we would want the following declarations:

```
%left '<'
%left '-'
%left '*'
```

In a more complete example, which supports other operators as well, we would declare them in groups of equal precedence. For example, `'+'` is declared with `'-'`:

```
%left '<' '>' '=' NE LE GE
%left '+' '-'
%left '*' '/'
```

(Here `NE` and so on stand for the operators for "not equal" and so on. We assume that these tokens are more than one character long and therefore are represented by names, not character literals.)

5.3.4 How Precedence Works

The first effect of the precedence declarations is to assign precedence levels to the terminal symbols declared. The second effect is to assign precedence levels to certain rules: each rule gets its precedence from the last terminal symbol mentioned in the components. (You can also specify explicitly the precedence of a rule. See Section 5.4 [Contextual Precedence], page 50.)

Finally, the resolution of conflicts works by comparing the precedence of the rule being considered with that of the look-ahead token. If the token's precedence is higher, the choice is to shift. If the rule's precedence is higher, the choice is to reduce. If they have equal precedence, the choice is made based on the associativity of that precedence level. The verbose output file made by '`-v`' (see Chapter 8 [Invocation], page 59) says how each conflict was resolved.

Not all rules and not all tokens have precedence. If either the rule or the look-ahead token has no precedence, then the default is to shift.

5.4 Operators with Context-Dependent Precedence

Often the precedence of an operator depends on the context. This sounds outlandish at first, but it is really very common. For example, a minus sign typically has a very high precedence as a unary operator, and a somewhat lower precedence (lower than multiplication) as a binary operator.

The Bison precedence declarations, `%left`, `%right` and `%nonassoc`, can only be used once for a given token; so a token has only one precedence declared in this way. For context-dependent precedence, you need to use an additional mechanism: the `%prec` modifier for rules.

The `%prec` modifier declares the precedence of a particular rule by specifying a terminal symbol whose predecence should be used for that rule. It's not necessary for that symbol to appear otherwise in the rule. The modifier's syntax is:

```
%prec terminal-symbol
```

and it is written after the components of the rule. Its effect is to assign the rule the precedence of *terminal-symbol*, overriding the precedence that would be deduced for it in the ordinary way. The altered rule precedence then affects how conflicts involving that rule are resolved (see Section 5.3 [Precedence], page 49).

Here is how `%prec` solves the problem of unary minus. First, declare a precedence for a fictitious terminal symbol named `UMINUS`. There are no tokens of this type, but the symbol serves to stand for its precedence:

```
...
%left '+' '-'
%left '*'
%left UMINUS
```

Now the precedence of `UMINUS` can be used in specific rules:

```
exp:    ...
        | exp '-' exp
        ...
        | '-' exp %prec UMINUS
```

5.5 Parser States

The function `yyparse` is implemented using a finite-state machine. The values pushed on the parser stack are not simply token type codes; they represent the entire sequence of terminal and nonterminal symbols at or near the top of the stack. The current state collects all the information about previous input which is relevant to deciding what to do next.

Each time a look-ahead token is read, the current parser state together with the type of look-ahead token are looked up in a table. This table entry can say, "Shift the look-ahead token." In this case, it also specifies the new parser state, which is pushed onto the top of the parser stack. Or it can say, "Reduce using rule number *n*." This means that a certain of tokens or groupings are taken off the top of the stack, and replaced by one grouping. In other words, that number of states are popped from the stack, and one new state is pushed.

There is one other alternative: the table can say that the look-ahead token is erroneous in the current state. This causes error processing to begin (see Chapter 6 [Error Recovery], page 55).

5.6 Reduce/Reduce conflicts

A reduce/reduce conflict occurs if there are two or more rules that apply to the same sequence of input. This usually indicates a serious error in the grammar.

For example, here is an erroneous attempt to define a sequence of zero or more `word` groupings.

```
sequence: /* empty */
                { printf ("empty sequence\n"); }
```

```
        | word
                { printf ("single word %s\n", $1); }
        | sequence word
                { printf ("added word %s\n", $2); }
        ;
```

The error is an ambiguity: there is more than one way to parse a single `word` into a `sequence`. It could be reduced directly via the second rule. Alternatively, nothing-at-all could be reduced into a `sequence` via the first rule, and this could be combined with the `word` using the third rule.

You might think that this is a distinction without a difference, because it does not change whether any particular input is valid or not. But it does affect which actions are run. One parsing order runs the second rule's action; the other runs the first rule's action and the third rule's action. In this example, the output of the program changes.

Bison resolves a reduce/reduce conflict by choosing to use the rule that appears first in the grammar, but it is very risky to rely on this. Every reduce/reduce conflict must be studied and usually eliminated. Here is the proper way to define `sequence`:

```
sequence: /* empty */
                { printf ("empty sequence\n"); }
        | sequence word
                { printf ("added word %s\n", $2); }
        ;
```

Here is another common error that yields a reduce/reduce conflict:

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    /* empty */
        | words word
        ;

redirects:/* empty */
        | redirects redirect
        ;
```

The intention here is to define a sequence which can contain either `word` or `redirect` groupings. The individual definitions of `sequence`, `words` and `redirects` are error-free, but the three together make a subtle ambiguity: even an empty input can be parsed in infinitely many ways!

Consider: nothing-at-all could be a `words`. Or it could be two `words` in a row, or three, or any number. It could equally well be a `redirects`, or two, or any number. Or it could be a `words` followed by three `redirects` and another `words`. And so on.

Here are two ways to correct these rules. First, to make it a single level of sequence:

```
sequence: /* empty */
        | sequence word
        | sequence redirect
```

```
        ;
```

Second, to prevent either a `words` or a `redirects` from being empty:

```
sequence: /* empty */
        | sequence words
        | sequence redirects
        ;

words:    word
        | words word
        ;

redirects:redirect
        | redirects redirect
        ;
```

6 Error Recovery

It is not usually acceptable to have the program terminate on a parse error. For example, a compiler should recover sufficiently to parse the rest of the input file and check it for errors; a calculator should accept another expression.

In a simple interactive command parser where each input is one line, it may be sufficient to allow `yyparse` to return 1 on error and have the caller ignore the rest of the input line when that happens (and then call `yyparse` again). But this is inadequate for a compiler, because it forgets all the syntactic context leading up to the error. A syntax error deep within a function in the compiler input should not cause the compiler to treat the following line like the beginning of a source file.

You can define how to recover from a syntax error by writing rules to recognize the special token `error`. This is a terminal symbol that is always defined (you need not declare it) and reserved for error handling. The Bison parser generates an `error` token whenever a syntax error happens; if you have provided a rule to recognize this token in the current context, the parse can continue. For example:

```
stmnts:  /* empty string */
        | stmnts '\n'
        | stmnts exp '\n'
        | stmnts error '\n'
```

The fourth rule in this example says that an error followed by a newline makes a valid addition to any `stmnts`.

What happens if a syntax error occurs in the middle of an `exp`? The error recovery rule, interpreted strictly, applies to the precise sequence of a `stmnts`, an `error` and a newline. If an error occurs in the middle of an `exp`, there will probably be some additional tokens and subexpressions on the stack after the last `stmnts`, and there will be tokens to read before the next newline. So the rule is not applicable in the ordinary way.

But Bison can force the situation to fit the rule, by discarding part of the semantic context and part of the input. First it discards states and objects from the stack until it gets back to a state in which the `error` token is acceptable. (This means that the subexpressions already parsed are discarded, back to the last complete `stmnts`.) At this point the `error` token can be shifted. Then, if the old look-ahead token is not acceptable to be shifted next, the parser reads tokens and discards them until it finds a token which is acceptable. In this example, Bison reads and discards input until the next newline so that the fourth rule can apply.

The choice of error rules in the grammar is a choice of strategies for error recovery. A simple and useful strategy is simply to skip the rest of the current input line or current statement if an error is detected:

```
stmnt: error ';'  /* on error, skip until ';' is read */
```

It is also useful to recover to the matching close-delimiter of an opening-delimiter that has already been parsed. Otherwise the close-delimiter will probably appear to be unmatched, and generate another, spurious error message:

```
primary:  '(' expr ')'
        | '(' error ')'
```

```
...
;
```

Error recovery strategies are necessarily guesses. When they guess wrong, one syntax error often leads to another. In the above example, the error recovery rule guesses that an error is due to bad input within one `stmnt`. Suppose that instead a spurious semicolon is inserted in the middle of a valid `stmnt`. After the error recovery rule recovers from the first error, another syntax error will be found straightaway, since the text following the spurious semicolon is also an invalid `stmnt`.

To prevent an outpouring of error messages, the parser will output no error message for another syntax error that happens shortly after the first; only after three consecutive input tokens have been successfully shifted will error messages resume.

Note that rules which accept the `error` token may have actions, just as any other rules can.

You can make error messages resume immediately by using the macro `yyerrok` in an action. If you do this in the error rule's action, no error messages will be suppressed. This macro requires no arguments; '`yyerrok;`' is a valid C statement.

The previous look-ahead token is reanalyzed immediately after an error. If this is unacceptable, then the macro `yyclearin` may be used to clear this token. Write the statement '`yyclearin;`' in the error rule's action.

For example, suppose that on a parse error, an error handling routine is called that advances the input stream to some point where parsing should once again commence. The next symbol returned by the lexical scanner is probably correct. The previous look-ahead token ought to be discarded with '`yyclearin;`'.

7 Debugging Your Parser

If a Bison grammar compiles properly but doesn't do what you want when it runs, the `yydebug` parser-trace feature can help you figure out why.

To enable compilation of trace facilities, you must define the macro `YYDEBUG` when you compile the parser. You could use '`-DYYDEBUG`' as a compiler option or you could put '`#define YYDEBUG`' in the C declarations section of the grammar file (see Section 3.1.1 [C Declarations], page 31). Alternatively, use the '`-t`' option when you run Bison (see Chapter 8 [Invocation], page 59). I always define `YYDEBUG` so that debugging is always possible.

The trace facility uses `stderr`, so you must add `#include <stdio.h>` to the C declarations section unless it is already there.

Once you have compiled the program with trace facilities, the way to request a trace is to store a nonzero value in the variable `yydebug`. You can do this by making the C code do it (in `main`, perhaps), or you can alter the value with a C debugger.

Each step taken by the parser when `yydebug` is nonzero produces a line or two of trace information, written on `stderr`. The trace messages tell you these things:

- Each time the parser calls `yylex`, what kind of token was read.
- Each time a token is shifted, the depth and complete contents of the state stack (see Section 5.5 [Parser States], page 51).
- Each time a rule is reduced, which rule it is, and the complete contents of the state stack afterward.

To make sense of this information, it helps to refer to the listing file produced by the Bison '`-v`' option (see Chapter 8 [Invocation], page 59). This file shows the meaning of each state in terms of positions in various rules, and also what each state will do with each possible input token. As you read the successive trace messages, you can see that the parser is functioning according to its specification in the listing file. Eventually you will arrive at the place where something undesirable happens, and you will see which parts of the grammar are to blame.

The parser file is a C program and you can use C debuggers on it, but it's not easy to interpret what it is doing. The parser function is a finite-state machine interpreter, and aside from the actions it executes the same code over and over. Only the values of variables show where in the grammar it is working.

8 Invocation of Bison; Command Options

The usual way to invoke Bison is as follows:

```
bison infile
```

Here *infile* is the grammar file name, which usually ends in '`.y`'. The parser file's name is made by replacing the '`.y`' with '`.tab.c`'. Thus, '`bison foo.y`' outputs `foo.tab.c`.

These options can be used with Bison:

'`-d`'
: Write an extra output file containing macro definitions for the token type names defined in the grammar and the semantic value type `YYSTYPE`, as well as a few `extern` variable declarations.

  If the parser output file is named *`name`*`.c` then this file is named *`name`*`.h`.

  This output file is essential if you wish to put the definition of `yylex` in a separate source file, because `yylex` needs to be able to refer to token type codes and the variable `yylval`. See Section 4.2 [Lexical], page 43.

'`-l`'
: Don't put any `#line` preprocessor commands in the parser file. Ordinarily Bison puts them in the parser file so that the C compiler and debuggers will associate errors with your source file, the grammar file. This option causes them to associate errors with the parser file, treating it an independent source file in its own right.

'`-o` *`outfile`*'
: Specify the name *outfile* for the parser file.

  The other output files' names are constructed from *outfile* as described under the '`-v`' and '`-d`' switches.

'`-t`'
: Output a definition of the macro `YYDEBUG` into the parser file, so that the debugging facilities are compiled. See Chapter 7 [Debugging], page 57.

'`-v`'
: Write an extra output file containing verbose descriptions of the parser states and what is done for each type of look-ahead token in that state.

  This file also describes all the conflicts, both those resolved by operator precedence and the unresolved ones.

  The file's name is made by removing '`.tab.c`' or '`.c`' from the parser output file name, and adding '`.output`' instead.

  Therefore, if the input file is `foo.y`, then the parser file is called `foo.tab.c` by default. As a consequence, the verbose output file is called `foo.output`.

'`-y`'
: Equivalent to '`-o y.tab.c`'; the parser output file is called `y.tab.c`, and the other outputs are called `y.output` and `y.tab.h`. The purpose of this switch is to imitate Yacc's output file name conventions.

  If the Bison utility is given the file name `yacc`, then it assumes the '`-y`' option automatically. Thus, Bison can substitute precisely for Yacc.

Appendix A Table of Bison Symbols

`error` A token name reserved for error recovery. This token may be used in grammar rules so as to allow the Bison parser to recognize an error in the grammar without halting the process. In effect, a sentence containing an error may be recognized as valid. On a parse error, the token `error` becomes the current look-ahead token. Actions corresponding to `error` are then executed, and the look-ahead token is reset to the token that originally caused the violation. See Chapter 6 [Error Recovery], page 55.

`YYACCEPT` Pretend that a complete utterance of the language has been read, by making `yyparse` return 0 immediately. See Section 4.1 [Parser Function], page 43.

`YYERROR` Pretend that an unrecoverable syntax error has occurred, by making `yyparse` return 1 immediately. The error reporting function `yyerror` is not called. See Section 4.1 [Parser Function], page 43.

`YYFAIL` Pretend that a syntax error has just been detected: call `yyerror` and then perform normal error recovery if possible (see Chapter 6 [Error Recovery], page 55) or (if recovery is not possible) make `yyparse` return nonzero. See Chapter 6 [Error Recovery], page 55.

`YYSTYPE` Data type of semantic values; `int` by default. See Section 3.5.1 [Value Type], page 35.

`yychar` External integer variable that contains the integer value of the current look-ahead token. Error-recovery rule actions may examine this variable. See Section 4.4 [Action Features], page 45.

`yyclearin`
Macro used in error-recovery rule actions. It clears the previous look-ahead token. See Chapter 6 [Error Recovery], page 55.

`yydebug` External integer variable set to zero by default. If `yydebug` is given a nonzero value, the parser will output information on input symbols and parser action. See Chapter 7 [Debugging], page 57.

`yyerrok` Macro to cause parser to recover immediately to its normal mode after a parse error. See Chapter 6 [Error Recovery], page 55.

`yyerror` User-supplied function to be called by `yyparse` on error. The function receives one argument, a pointer to a character string containing an error message. See Section 4.3 [Error Reporting], page 45.

`yylex` User-supplied lexical analyzer function, called with no arguments to get the next token. See Section 4.2 [Lexical], page 43.

`yylval` External variable in which `yylex` should place the semantic value associated with a token. See Section 4.2 [Lexical], page 43.

`yylloc` External variable in which `yylex` should place the line and column numbers associated with a token. This is needed only if the '`@`' feature is used in the grammar actions. See Section 4.2 [Lexical], page 43.

`yyparse` The parser function produced by Bison; call this function to start parsing. See Section 4.1 [Parser Function], page 43.

`%left` Bison declaration to assign left associativity to token(s). See Section 3.6.2 [Precedence Decl], page 39.

`%nonassoc`
Bison declaration to assign nonassociativity to token(s). See Section 3.6.2 [Precedence Decl], page 39.

`%prec` Bison declaration to assign a precedence to a specific rule. See Section 5.4 [Contextual Precedence], page 50.

`%pure_parser`
Bison declaration to request a pure (reentrant) parser. See Section 3.6.7 [Pure Decl], page 41.

`%right` Bison declaration to assign right associativity to token(s). See Section 3.6.2 [Precedence Decl], page 39.

`%start` Bison declaration to specify the start symbol. See Section 3.6.6 [Start Decl], page 41.

`%token` Bison declaration to declare token(s) without specifying precedence. See Section 3.6.1 [Token Decl], page 39.

`%type` Bison declaration to declare nonterminals. See Section 3.6.4 [Type Decl], page 40.

`%union` Bison declaration to specify several possible data types for semantic values. See Section 3.6.3 [Union Decl], page 40.

These are the punctuation and delimiters used in Bison input:

‘`%%`’ Delimiter used to separate the grammar rule section from the Bison declarations section or the additional C code section. See Section 1.7 [Grammar Layout], page 13.

‘`%{ %}`’ All code listed between ‘`%{`’ and ‘`%}`’ is copied directly to the output file uninterpreted. Such code forms the “C declarations” section of the input file. See Section 3.1 [Grammar Outline], page 31.

‘`/*...*/`’ Comment delimiters, as in C.

‘`:`’ Separates a rule’s result from its components. See Section 3.3 [Rules], page 33.

‘`;`’ Terminates a rule. See Section 3.3 [Rules], page 33.

‘`|`’ Separates alternate rules for the same result nonterminal. See Section 3.3 [Rules], page 33.

Appendix B Glossary

Backus-Naur Form (BNF)
: Formal method of specifying context-free grammars. BNF was first used in the *ALGOL-60* report, 1963. See Section 1.1 [Language and Grammar], page 9.

Context-free grammars
: Grammars specified as rules that can be applied regardless of context. Thus, if there is a rule which says that an integer can be used as an expression, integers are allowed *anywhere* an expression is permitted. See Section 1.1 [Language and Grammar], page 9.

Dynamic allocation
: Allocation of memory that occurs during execution, rather than at compile time or on entry to a function.

Empty string
: Analogous to the empty set in set theory, the empty string is a character string of length zero.

Finite-state stack machine
: A "machine" that has discrete states in which it is said to exist at each instant in time. As input to the machine is processed, the machine moves from state to state as specified by the logic of the machine. In the case of the parser, the input is the language being parsed, and the states correspond to various stages in the grammar rules. See Chapter 5 [Algorithm], page 47.

Grouping
: A language construct that is (in general) grammatically divisible; for example, 'expression' or 'declaration' in C. See Section 1.1 [Language and Grammar], page 9.

Infix operator
: An arithmetic operator that is placed between the operands on which it performs some operation.

Input stream
: A continuous flow of data between devices or programs.

Language construct
: One of the typical usage schemas of the language. For example, one of the constructs of the C language is the `if` statement. See Section 1.1 [Language and Grammar], page 9.

Left associativity
: Operators having left associativity are analyzed from left to right: '`a+b+c`' first computes '`a+b`' and then combines with '`c`'. See Section 5.3 [Precedence], page 49.

Left recursion
: A rule whose result symbol is also its first component symbol; for example, '`expseq1 : expseq1 ',' exp;`'. See Section 3.4 [Recursion], page 34.

Left-to-right parsing
: Parsing a sentence of a language by analyzing it token by token from left to right. See Chapter 5 [Algorithm], page 47.

Lexical analyzer (scanner)
: A function that reads an input stream and returns tokens one by one.

Look-ahead token
: A token already read but not yet shifted. See Section 5.1 [Look-Ahead], page 47.

Nonterminal symbol
: A grammar symbol standing for a grammatical construct that can be expressed through rules in terms of smaller constructs; in other words, a construct that is not a token. See Section 3.2 [Symbols], page 32.

Parse error
: An error encountered during parsing of an input stream due to invalid syntax. See Chapter 6 [Error Recovery], page 55.

Parser
: A function that recognizes valid sentences of a language by analyzing the syntax structure of a set of tokens passed to it from a lexical analyzer.

Postfix operator
: An arithmetic operator that is placed after the operands upon which it performs some operation.

Reduction
: Replacing a string of nonterminals and/or terminals with a single nonterminal, according to a grammar rule. See Chapter 5 [Algorithm], page 47.

Reverse polish notation
: A language in which all operators are postfix operators.

Right recursion
: A rule whose result symbol is also its last component symbol; for example, ‘`expseq1: exp ',' expseq1;`’. See Section 3.4 [Recursion], page 34.

Semantics
: In computer languages the semantics are specified by the actions taken for each instance of the language, i.e., the meaning of each statement. See Section 3.5 [Semantics], page 34.

Shift
: A parser is said to shift when it makes the choice of analyzing further input from the stream rather than reducing immediately some already-recognized rule. See Chapter 5 [Algorithm], page 47.

Single-character literal
: A single character that is recognized and interpreted as is. See Section 1.2 [Grammar in Bison], page 10.

Start symbol
: The nonterminal symbol that stands for a complete valid utterance in the language being parsed. The start symbol is usually listed as the first nonterminal symbol in a language specification. See Section 3.6.6 [Start Decl], page 41.

Symbol table
: A data structure where symbol names and associated data are stored during parsing to allow for recognition and use of existing information in repeated uses of a symbol. See Section 2.4 [Multi-function Calc], page 22.

Token
: A basic, grammatically indivisible unit of a language. The symbol that describes a token in the grammar is a terminal symbol. The input of the Bison parser is a stream of tokens which comes from the lexical analyzer. See Section 3.2 [Symbols], page 32.

Terminal symbol
: A grammar symbol that has no rules in the grammar and therefore is grammatically indivisible. The piece of text it represents is a token. See Section 1.1 [Language and Grammar], page 9.

Index

$

%

@

|

A

B

C

D

E

F

G

I

L

M

N

O

P

R

S

T

U

V

W

Y

Table of Contents