Zyacc

A Parser-Generator
Edition 1.0, for Zyacc Version 1.
July 1997
Adapted from The Bison Manual

Zerksis D. Umrigar

This is the first edition of the Zyacc documentation.

Introduction

Zyacc is a general-purpose parser generator that converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. Once you are proficient with Zyacc, you may use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.

Zyacc is largely upward compatible with Yacc and Bison: all properly-written Yacc and Bison grammars ought to work with Zyacc with minimal change. Anyone familiar with Yacc or Bison should be able to use Zyacc with little trouble. This manual uses the spellings Zyacc, Bison and Yacc to refer to the specific programs, while using the spelling yacc to refer to any one of the above programs. You need to be fluent in C programming in order to use Zyacc or to understand this manual.

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

Acknowledgements

The bulk of this manual is derived from the well-written Bison manual (see section "Bison Manual" in *Bison: The YACC-compatible Parser Generator*). The changes made by the present author include reformatting the examples and adding sections specific to Zyacc.

Many of the algorithms used within Zyacc is based on work by others. See the `zyacc/refs.bib` file included with the distribution for some of the references.

Zyacc Enhancements

Zyacc provides the following enhancements:

- Supports inherited attributes which can be uniquely evaluated in a left-to-right parse (see Section 3.5.7 [Inherited Attributes], page 41).
- Supports semantic tests which allow the outcome of runtime semantic tests to affect parsing decisions (see Section 3.5.8 [Semantic Tests], page 43).
- Permits remote interactive debugging of generated parsers either by using a textual interface or by using a java-based GUI (see Chapter 8 [Debugging], page 76). The debugger allows the setting of breakpoints on any grammar symbol and selective display of the current parser state.
- Allows named attribute variables which make maintaining grammars easier (see Section 3.5.6 [Named Attributes], page 39).
- Can generate its parser description files in HTML which can then be viewed using any Web browser (see Section 9.3 [Zyacc Options], page 87).
- Provides a `%look` directive to check (at parser construction time) whether a reduction requires lookahead (see Section 3.6.11 [Specifying the Lookahead], page 52).
- Allows multiple start nonterminals and allows a call to the parsing function to be made for a particular start nonterminal (see Section 3.6.7 [Start Decl], page 49).
- It is possible to avoid having types which are only used in describing nonterminal semantics written into the generated '`.h`' file (see Section 3.5.1 [Value Type], page 35).

- Allows multiple-character quoted literal tokens (see Section 3.6.3 [Multi-Character Lits], page 48).
- Allows command-line options to be specified from within the parser file (see Section 3.6.10 [Option Decl], page 52).

1 The Concepts of Zyacc

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

1.1 Languages and Context-Free Grammars

In order for Zyacc 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 Zyacc is essentially machine-readable BNF.

Not all context-free languages can be handled by Zyacc, only those that are LALR(1). In brief, this means that it must be possible to tell how to parse any portion of an input string with just a single token of look-ahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1). See Section 5.7 [Mysterious Reduce/Reduce Conflicts], page 69, for more information on this.

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 Zyacc 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 Zyacc Input

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

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

The Zyacc 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. Zyacc also permits literal token names consisting of multiple characters — in that case, it is necessary to define an alias used for referring to that token external to the parser.

Given this notation, it is easy to express grammar rules in Zyacc syntax. For example, here is the Zyacc 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 Zyacc punctuation used in every rule.

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

See Section 3.3 [Syntax of Grammar 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 Zyacc grammar has both a token type and a *semantic value*. See Section 3.5 [Defining Language Semantics], page 35, for details.

The token type is a terminal symbol defined in the grammar, such as `INTEGER`, `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 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 Zyacc 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 Zyacc Output: the Parser File

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

The job of the Zyacc 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 Zyacc 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 Zyacc does not depend on this. See Section 4.2 [The Lexical Analyzer Function `yylex`], page 54.

The Zyacc 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 [Parser C-Language Interface], page 54.

Aside from the token type names and the symbols in the actions you write, all variable and function names used in the Zyacc 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 Zyacc grammar file except for the ones defined in this manual.

1.6 Stages in Using Zyacc

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

1. Formally specify the grammar in a form recognized by Zyacc (see Chapter 3 [Zyacc Grammar Files], 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 [The Lexical Analyzer Function `yylex`], page 54). 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 Zyacc-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 Zyacc on the grammar to produce the parser.
2. Compile the code output by Zyacc, as well as any other source files.
3. Link the object files to produce the finished product.

1.7 The Overall Layout of a Zyacc Grammar

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

```
%{
C declarations
%}

Zyacc declarations

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

The '`%%`', '`%{`' and '`%}`' are punctuation that appears in every Zyacc 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 Zyacc 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 six sample programs written using Zyacc. The first three examples illustrate features of Zyacc which are also present in yacc. These are a reverse polish notation calculator, an algebraic (infix) notation calculator, and a multi-function calculator. The last three illustrate features which are unique in Zyacc: these include a differently sugared implementation of the multi-function calculator, a calculator which evaluates polynomials and a lazy person's calculator which allows omitting some sets of parentheses.

The code shown in this manual has been extracted automatically from code which has been tested. These examples are simple, but Zyacc grammars for real programming languages are written the same way.

If you have access to the Zyacc distribution, you will find these examples under the `doc` directory.

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 Zyacc input files.

2.1.1 Declarations for `rpcalc`

Here are the C and Zyacc declarations for the reverse polish notation calculator. As in C, comments are placed between '`/*...*/`'.

```
/* Reverse polish notation calculator. */

%{
#include <math.h>
#include <stdio.h>

#define YYSTYPE double

int yylex(void);
void yyerror(const char *errMsg);

%}

%token NUM_TOK

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

The C declarations section (see Section 3.1.1 [The C Declarations Section], 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 [Data Types of Semantic Values], page 35).

The Zyacc 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, Zyacc declarations, provides information to Zyacc about the token types (see Section 3.1.2 [The Zyacc Declarations Section], 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_TOK`, the token type for numeric constants (since Zyacc will `#define NUM_TOK` adding the '`_TOK`' suffix prevents it from clashing with identifiers used for other purposes).

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_TOK
  | 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 Zyacc 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 [Recursive Rules], 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_TOK
  | exp exp '+'     { $$ = $1 + $2;    }
  | exp exp '-'     { $$ = $1 - $2;    }
  ...
  ;
```

Note that there is no semantic action specified in the case when a `exp` is a `NUM_TOK`. That is because if a rule has no associated semantic action, then Zyacc automatically generates the implicit action `{ $$= $1; }`, which is exactly what we need in this case.

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

```
exp
  : NUM_TOK
  ;
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, Zyacc by default copies the value of `$1` into `$$`. This is what happens in the first rule (the one that uses `NUM_TOK`).

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

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

means the same thing as this:

```
exp
  : NUM_TOK
  | 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 Zyacc parser gets its tokens by calling the lexical analyzer. See Section 4.2 [The Lexical Analyzer Function `yylex`], page 54.

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_TOK` 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 Zyacc 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 Zyacc as a C macro whose definition is the appropriate number. In this example, therefore, `NUM_TOK` 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 Zyacc 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 [Declarations for `rpcalc`], page 8.)

A token type code of zero is returned if the end-of-file is encountered. (Zyacc 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 in yylval and the token NUM_TOK, or the ASCII
 * character read if not a number.  Skips all blanks
 * and tabs, returns 0 for EOF.
 */

#include <ctype.h>

int
yylex(void)
{
  int c;

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

2.1.4 The Controlling Function

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

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

2.1.5 The Error Reporting Routine

When `yyparse` detects a syntax error, it calls the error reporting function `yyerror` to print an error message (usually but not always `"parse error"`). It is up to the programmer to supply `yyerror` (see Chapter 4 [Parser C-Language Interface], page 54), so here is the definition we will use:

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

After `yyerror` returns, the Zyacc parser may recover from the error and continue parsing if the grammar contains a suitable error rule (see Chapter 6 [Error Recovery], page 71). 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 Zyacc to Make the Parser

Before running Zyacc 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 [The Overall Layout of a Zyacc Grammar], page 7).

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:

```
zyacc file_name.y
```

In this example the file was called '`rpcalc.y`' (for "Reverse Polish CALCulator"). Zyacc produces a file named '*file_name*`.tab.c`', removing the '`.y`' from the original file name. The file output by Zyacc 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 Zyacc 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
$
```

On an error, `rpcalc` simply gives up after printing an error message as shown below:

```
$ rpcalc
1 2 + +
parse error
$ echo $?
1
$
```

The `echo $?` makes the shell (the Bourne shell or Korn shell) print the exit code of the last command (`rpcalc`) in this case. The value 1 is the value returned by the `return yyparse()` statement in `main()`.

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 Zyacc code for '`calc.y`', an infix desk-top calculator.

```
/* Infix notation calculator--calc */

%{
#include <math.h>
#include <stdio.h>

#define YYSTYPE double

int yylex(void);
void yyerror(const char *errMsg);
%}

/* zyacc declarations */
%token NUM_TOK
%left '-' '+'
%left '*' '/'
%left NEG        /* negation--unary minus */
%right '^'       /* exponentiation   */

/* Grammar follows */
%%
input
  : /* empty */
  | input line
  ;

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

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

Operator precedence is determined by the line ordering of the declarations; the higher the line number of the declaration (lower on the page or screen), the higher the precedence. Hence, exponentiation has the highest precedence, unary minus (`NEG`) is next, followed by '`*`' and '`/`', and so on. See Section 5.3 [Operator Precedence], page 64.

The other important new feature is the `%prec` in the grammar section for the unary minus operator. The `%prec` simply instructs Zyacc that the rule '`| '-' exp`' has the same precedence as `NEG`—in this case the next-to-highest. See Section 5.4 [Context-Dependent Precedence], page 66.

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 Zyacc 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 Zyacc; its meaning is that error recovery is complete (see Chapter 6 [Error Recovery], page 71). 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 Zyacc programs.

2.4 Multi-Function Calculator: `mfcalc`

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

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

function_name `(`*argument*`)`

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

```
$ mfcalc
pi = 3.141592653589
        3.141592654
sin(pi)
        7.932657935e-13
alpha = beta1 = 2.3
        2.3
alpha
        2.3
ln(alpha)
        0.8329091229
exp(ln(beta1))
        2.3
$
```

Note that multiple assignment and nested function calls are permitted.

The implementation given below has the scanner first *intern* identifiers into a separate *string-space* so that identifiers with the same spelling share the same string-space entry. The scanner returns the intern'd representation of the identifier to the parser. The parser uses the intern'd representation to access a symbol table which keeps track of the properties of the identifier (whether it is a variable or function, the value associated with the identifier).

2.4.1 Declarations for `mfcalc`

The previous grammars had only a single semantic type: the `double` value associated with `NUM_TOK`s and `exp`s. However, now we will have identifiers used for variables and functions: they will need a semantic type different from `double`. The C and Zyacc declarations given below for the multi-function calculator use a couple of new Zyacc features to allow multiple semantic types (see Section 3.5.2 [More Than One Value Type], page 35).

```
%{
#include <ctype.h>
#include <math.h>
#include <stdio.h>

int yylex(void);
void yyerror(const char *errMsg);

/* Typedef typical unary <math.h> function. */
typedef double (*MathFnP)(double input);

/* Interface to symbol table. */
static double getIDVal(const char *name);
static MathFnP getIDFn(const char *name);
static void setIDVal(const char *name, double val);
static void setIDFn(const char *name, MathFnP fnP);

%}
%union {
  double val;          /* For returning numbers. */
  const char *id;      /* For returning identifiers. */
}

%token <val>  NUM_TOK    /* Double precision number */
%token <id>   ID_TOK     /* Identifiers. */
%type  <val>  exp

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

/* Grammar follows */
%%
```

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_TOK`) and `char *` pointers for the names of variables and functions. See Section 3.6.4 [The Collection of Value Types], page 48.

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_TOK`, `ID_TOK`, and `exp`. Their declarations are augmented with information about their data type (placed between angle brackets).

The Zyacc 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 3.6.5 [Nonterminal Symbols], page 49.

The C declarations section above also declares the functions used to interface to the symbol table. The symbol table is a mapping from identifiers to either `double` values (for identifiers which are variables) or function pointers (for identifiers whicha re functions). The '`get`' functions are used to get the value associated with an identifier; the '`set`' functions are used to change the value.

2.4.2 Grammar Rules for `mfcalc`

The grammar rules for the multi-function calculator are identical to those for `calc` except for three additional rules for `exp` shown below:

```
exp
  : ID_TOK               { $$= getIDVal($1); }
  | ID_TOK '=' exp       { setIDVal($1, $3); $$= $3; }
  | ID_TOK '(' exp ')'   { $$= (*(getIDFn($1)))($3); }
```

The above additional rules correspond to the cases when an expression is an identifier, an assignment or a function application respectively.

- If an expression is an identifier, we merely lookup its value in the symbol table using `getIDVal()`.
- If an expression is an assignment statement, we use the symbol table interface to set the value of the left-hand side identifier to the value of the right-hand side expression using `setIDVal()`.
- If an expression is a function application, we lookup the function pointer associated with the identifier in the symbol table using `getIDFn` and apply the corresponding function to the value of the expression.

2.4.3 The `mfcalc` Symbol Table

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

The symbol table itself consists of a linked list of records. It provides for either functions or variables to be placed in the table. Its definition is as follows:

```
/* Symbol table ADT. */

/* Possible types for symbols. */
typedef enum { VAR_SYM, FN_SYM } SymType;

typedef struct Sym {
  const char *name;      /* Name of symbol. */
  SymType type;          /* Type of symbol. */
  union {
    double var;          /* Value of a VAR_SYM. */
    MathFnP fn;          /* Value of a FN_SYM. */
  } value;
  struct Sym *succ;      /* Link field. */
} Sym;

/* The symbol table: a chain of Sym's.*/
static Sym *symTab;
```

The `Sym` type contains the `name` of the identifier and a `type` field which classifies the symbol as either a variable (`type == VAR_SYM`) or function (`type == FN_SYM`). Depending on this `type` field, the symbol's `value` is in the `union` field `var` for a variable or in `fn` for a function. The `succ` field is used to chain all symbols together in a LIFO chain.

The heart of the symbol table module is the `getSym()` routine shown below:

```
/* Search symTab for name.  If doCreate, then create an entry for
 * it if it is not there.  Return pointer to Sym entry.
 */
static Sym *
getSym(const char *name, unsigned doCreate)
{
  Sym *p;
  for (p= symTab; p != NULL && p->name != name; p= p->succ) ;
  if (p == NULL && doCreate) {
    p= malloc(sizeof(Sym));
    p->name= name; p->succ= symTab; symTab= p;
  }
  return p;
}
```

`getSym()` searches the linear chain of `Sym`s rooted in `symTab` for an identifier with a specified `name`. If it finds one, it returns a pointer to the corresponding `Sym`; otherwise if `doCreate` is zero it simply returns `NULL`; if `doCreate` is non-zero it creates a new entry for `name` and links it in at the head of the `symTab` chain.

To get the value or function associated with an identifier, the symbol table interface routines `getIDVal()` and `getIDFn()` shown below can be used:

```
/* Get value associated with name; signal error if not ok. */
static double
getIDVal(const char *name)
{
  const Sym *p= getSym(name, 0);
  double val= 1.0; /* A default value. */
  if (!p) fprintf(stderr, "No value for %s.\n", name);
  else if (p->type != VAR_SYM)
    fprintf(stderr, "%s is not a variable.\n", name);
  else val= p->value.var;
  return val;
}

/* Get function associated with name; signal error if not ok. */
static MathFnP
getIDFn(const char *name)
{
  const Sym *p= getSym(name, 0);
  MathFnP fn= sin; /* A default value. */
  if (!p) fprintf(stderr, "No value for %s.\n", name);
  else if (p->type != FN_SYM)
    fprintf(stderr, "%s is not a function.\n", name);
  else fn= p->value.fn;
```

```
    return fn;
  }
```

The routines take care of printing out a error message if the `name` is not found or is of an inappropriate type.

Changing the values of a symbol is done in a straight-forward manner as shown below:

```
  /* Unconditionally set name to a VAR_SYM with value val. */
  static void
  setIDVal(const char *name, double val)
  {
    Sym *p= getSym(name, 1);
    p->type= VAR_SYM; p->value.var= val;
  }

  /* Unconditionally set name to a FN_SYM with fn ptr fnP. */
  static void
  setIDFn(const char *name, MathFnP fnP)
  {
    Sym *p= getSym(name, 1);
    p->type= FN_SYM; p->value.fn= fnP;
  }
```

It is necessary to preload the symbol table with the functions which will be provided by `mfcalc`. This is done as shown below:

```
  /* Initial functions. */
  struct {
    const char *name;       /* Name of function. */
    MathFnP fn;             /* Corresponding <math.h> function. */
  } initFns[]= {
    { "sin", sin },
    { "cos", cos },
    { "atan", atan },
    { "ln", log },
    { "exp", exp },
    { "sqrt", sqrt }
  };

  static void
  initSyms(void)
  {
    const unsigned n= sizeof(initFns)/sizeof(initFns[0]);
    unsigned i;
    for (i= 0; i < n; i++) {
      setIDFn(getID(initFns[i].name), initFns[i].fn);
    }
  }
```

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

The new version of `main` includes a call to `initSyms`, the function defined above which initializes the symbol table:

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

2.4.4 The `mfcalc` Scanner

The function `yylex` must now recognize numeric values, single-character arithmetic operators and identifiers. The recognition of numeric values and single-character arithmetic operators is exactly as before. In order to recognize identifiers, the following code fragment is added to `yylex()` after the code for recognizing numbers:

```
/* Char starts an identifier => read the name. */
if (isalpha(c)) {
  ungetc(c, stdin);
  yylval.id= readID();
  return ID_TOK;
}
```

After checking if the current character is a letter, the code fragment calls `readID()` after pushing back the current character. `readID()` reads the current character into a dynamically sized buffer as shown below. After completing the read, `readID()` calls `getID()` routine which interfaces to the string-space. `getID()` will return a unique `char *` pointer for the identifier. If the returned `char *` pointer is not equal to the dynamically allocated buffer, then the identifier had been seen previously and the dynamic buffer is freed.

```
/* Read alphanumerics from stdin into a buffer.  Check
 * if identical to previous ident: if so return pointer
 * to previous, else return pointer to new buffer.
 * Assumes char after ident is not an EOF.
 */
static const char *
readID(void)
{
  enum { SIZE_INC= 40 };
  unsigned size= SIZE_INC;
  char *buf= malloc(size);
  unsigned i= 0;
  int c;
  const char *ident;

  do { /* Accumulate stdin into strSpace. */
    c= getchar();
```

```
      if (i >= size) buf= realloc(buf, size*= 2);
      buf[i++]= c;
    } while (isalnum(c));

    ungetc(c, stdin); buf[i - 1]= '\0'; /* Undo extra read. */
    buf= realloc(buf, i); /* Resize buf to be only as big as needed. */

    ident= getID(buf);     /* Search string-space. */

    if (ident != buf) free(buf); /* Previously existed. */
    return ident;
  }
```

2.4.5 The `mfcalc` String Space

The string-space is maintained as a linked list of `Ident`s as shown below. The `getID()` function does a linear search through the linked list for its `name` argument: if found it returns a pointer to the previously entered `name`, if not found, it adds a new entry at the head of the list and returns the `name` argument.

```
/* String space ADT to map identifiers into IDNums. */
typedef struct Ident {
  const char *name;      /* NUL-terminated chars of identifier. */
  struct Ident *succ;    /* Next entry in linear chain. */
} Ident;

/* The string space is a chain of Ident's. */
static Ident *strSpace;

static const char *
getID(const char *name)
{
  Ident *p;
  for (p= strSpace; p != NULL; p= p->succ) {
    if (strcmp(name, p->name) == 0) break;
  }
  if (!p) {
    p= malloc(sizeof(Ident));
    p->name= name;
    p->succ= strSpace; strSpace= p;
  }
  return p->name;
}
```

2.4.6 Why a Separate String Space

It is possible to design `mfcalc` so that the scanner interns identifiers directly using the symbol table, avoiding the need for a separate string space. Though that is adequate for `mfcalc`, such an approach may be unwieldy for more complex applications because of the reasons outlined below.

A symbol table is a mapping from identifiers to objects like variables and functions; in many applications (like programming languages with separate name-spaces or multiple scopes) this mapping is typically one-to-many. Hence maintaining a symbol table usually requires understanding the context within which a identifier is used. If the scanner is solely responsible for maintaining the symbol table, then it must keep track of the context within which an identifier is used — this complicates the scanner. If other higher level modules like the parser share the maintenance of the symbol table with the scanner, then the symbol table interface may become complex due to order-dependencies and feedback between modules.

The separate string table avoids some of these complications. Though the implementation given for `mfcalc` required two searches for each identifier (one within the string space, the other within the symbol table), it is possible to get by with a single search.

2.5 The Multi Function Calculator Using a Sugared Syntax

One disadvantage of the `$`*i* syntax used for specifying the semantic attributes of grammar symbols is that correctly specifying the *i* is often tedious, especially with long rules. More seriously, if a rule changes then the *i* in a `$`*i* may also change: this can make maintaining such grammars error-prone.

Another disadvantage is that the attributes of grammar symbols are declared in section 1 of the grammar file (using `%union`, `%type` and other declarations), rather than near where they are actually used in section 2. This conflicts with modern software engineering practice, where entities are declared near or at the point of first use.

Zyacc permits syntactic sugar which overcomes these deficiencies. It allows named attribute variables to refer to the semantic attributes, thus making it possible to refer to semantic attributes without having to count grammar symbols. It also allows the semantic attributes of a nonterminal to be declared on the left-hand side of rules for that nonterminal.

The syntax is illustrated by repeating the grammar for the `mfcalc` using the new syntax:

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

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

exp(double $v)
  : ID_TOK($id)                 { $v= getIDVal($id); }
  | ID_TOK($id) '=' exp($v)     { setIDVal($id, $v); }
```

```
    | ID_TOK($id) '(' exp($v1) ')'{ $v= (*(getIDFn($id)))($v1); }
    | NUM_TOK($v)
    | exp($v1) '+' exp($v2)       { $v= $v1 + $v2; }
    | exp($v1) '-' exp($v2)       { $v= $v1 - $v2; }
    | exp($v1) '*' exp($v2)       { $v= $v1 * $v2; }
    | exp($v1) '/' exp($v2)       { $v= $v1 / $v2; }
    | '-' exp($v1)  %prec NEG     { $v= -$v1; }
    | exp($v1) '^' exp($v2)       { $v= pow($v1, $v2); }
    | '(' exp($v1) ')'            { $v= $v1; }
    ;
  /* End of grammar */
  %%
```

The semantic attributes of a grammar symbol are written within parentheses following the grammar symbols using a positional notation similar to that of function application in C. The types of the semantic attributes for a nonterminal are declared on the left-hand side of a rule using a syntax similar to that for function prototype declarations in C.

In the above example, `exp` has a single semantic attribute which represents the value of the expression; its value is computed on-the-fly as an `exp` is parsed. This attribute is named `$v` and is declared to be of type `double` on the common left-hand side of the rules for `exp`. This declaration is similar to that of a formal parameter in a function definition.

The attributes of right-hand side symbols detail the flow of semantic information among the grammatical constructs constituting the rule. For example in the first rule for `exp` repeated below, the terminal `ID_TOK` representing a variable has an attribute referred to `$id` within that rule. The action for that rule computes the attribute `$v` for the left-hand side `exp` in terms of this `$id` using the function `getIDVal()` which looks up the value of the variable using the current symbol table.

```
  exp(double $v)
    : ID_TOK($id)                 { $v= getIDVal($id); }
    | ID_TOK($id) '=' exp($v)     { setIDVal($id, $v); }
```

Similarly, in the second rule which describes an assignment expression, the `$v` for the left-hand side `exp` is computed via the `exp` on the right-hand side: this is indicated by using the same name `$v` for both occurrences.

The sugared syntax allows the attributes of a nonterminal to be declared when that nonterminal occurs on the left-hand side of a rule. Since this can never happen for a terminal symbol, the attributes of terminal symbols must still be declared in section 1 of the Zyacc file as follows:

```
  %token <val>(double $v) NUM_TOK          /* Double precision number */
  %token <id>(const char *$id) ID_TOK      /* Identifiers. */
```

We have enhanced the `<val>` type tag previously used for a `NUM_TOK` by a parenthesized list containing declarations for its semantic attributes, namely a single attribute named `$v` of type `double`. Similarly an `ID_TOK` has a semantic type with type tag `<id>` with a single `const char *` semantic attribute named `$id`.

If the sugared syntax is used for the entire grammar, then there is no need for a `%union` directive. In fact, Zyacc generates one automatically; for the above grammar, it will generate something similar to:

```
typedef union {
  struct { double v; } val;       /* NUM_TOK */
  struct { const char *id; } id; /* ID_TOK */
} YYSTYPE;
```

The `union` has a separate `struct` field for each sugared `%token` declaration and for each nonterminal.

The semantic attributes of a terminal are stored in the field in the `union` which has a name identical to the <*type*> tag used in the `%token` declaration for that terminal. The fields in the `struct` are identical to the attributes declared for that <*type*> tag in the token declaration, except that the '`$`'s are removed. Hence the above `%token` declarations result in the fields `val` and `id` in the union, with each of them being a `struct` containing the fields `v` and `id` repectively.

Since the programmer controls the names used in the fields in the `union` for terminals, the programmer should use the same names when setting up semantic information for the terminals in the scanner. For the above grammar, the only change necessary in the scanner is to change assignments to `yylval` to access the appropriate fields in the `union`. Specifically, the code for numbers and identifiers in `yylex()` is changed to:

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

/* Char starts an identifier => read the name. */
if (isalpha(c)) {
  ungetc(c, stdin);
  yylval.id.id= readID();
  return ID_TOK;
}
```

The semantic attributes for each nonterminal are stored in a separate `struct` field within the `union`. The types of the fields used within the `struct` for a nonterminal are identical to the types declared for the attributes of that nonterminal. However, the names used for and within this field are implementation defined; that should not be a problem, as there is no need for the programmer to access them directly.

Using these named attributes overcomes the disadvantages mentioned earlier for the numeric `$`-variables. However, the resulting grammars tend to be somewhat more verbose. The new notation does not by itself add anything to the power of the grammars, which is why we refer to it as syntactic sugar. However when used in conjunction with the features mentioned in the next couple of sections, the named attribute variables do add to the power of the grammars.

2.6 A Multi Function Calculator Which Evaluates Polynomials

Consider enhancing our calculator with an evaluation operator `@`, which evaluates a polynomial. This operator can be used as in `x @ [5, 2, 3]` to denote the polynomial `5*x^2 + 2*x + 3`.

The polynomial coefficients within the square brackets are allowed to be arbitrary expressions (including nested polynomial evaluations). The `@` operator should associate to the left and have the highest precedence (greater than that of exponentiation). An example of the use of this polynomial calculator is shown below:

```
$ polycalc
3@[4, 5, 1]
        52
2 * 3@[4, 5, 1]
        104
3@[4, 2@[8, 4, 2, 1], 1]
        292
$
```

It is easy enough to force the `@` operator to have the required precedence by simply adding a `%left` declaration after the declaration for the exponentiation operator as shown below.

```
%right '^'              /* Exponentiation */
%left '@'               /* Polynomial application */
```

Unfortunately, evaluating the polynomial is not that easy.

One solution to this problem is to not evaluate the polynomial as its coefficients are being parsed, but to merely store its coefficient values in some data structure which will be the semantic attribute for the coefficient list. Once the coefficient list has been parsed, the data structure containing the coefficients and the value of the point at which evaluation is requested can be passed to some action routine which evaluates the polynomial at that point. Assuming that the coefficients have been evaluated into an (`n+1`)-element array `coeffs[]`, a scheme like the following is typical for the action routine:

```
/* Evaluate nth-degree polynomial
 * coeffs[n]*point^n + coeffs[n-1]*point^(n-1) + ... + coeffs[0]
 */
double evalPoly(double point, double coeffs[], int n)
{
  double sum= 0;
  int i;
  for (i= n; i >= 0; i--) sum= sum*point + coeffs[i];
  return sum;
}
```

Unfortunately, this requires auxiliary storage and a non-trivial data structure to handle nested polynomials. Fortunately in Zyacc, the computation represented by the above code can be performed on-the-fly during parsing using a special type of semantic attributes known as *inherited attributes* described below.

All the attributes seen in the previous examples are known as *synthesized attributes*. An attribute for a construct is said to be a *synthesized attribute* if the value of that attribute directly depends only on the semantic attributes of the constituents of that construct and not on the semantic attributes of a surrounding construct. For example, the value of a `exp` is independent of the larger `exp` within which it may appear.

A little reflection shows that synthesized attributes are not sufficient to evaluate polynomials on-the-fly during parsing: the contribution made by each coefficient to the final value depends not only on the value of the coefficient but also on the point at which the polynomial is being evaluated (the value of the expression before the `@` operator), as well as the position of the coefficient in the list of polynomial coefficients.

Inherited attributes allow us to get around this problem: they allow us to pass in information about the surrounding context to a grammatical construct. Inherited attributes are declared using the `%in` keyword as shown in the following extract of the `polycalc` grammar:

```
coeffs(%in double $sum, %in double $point, %out double $v)
  : exp($v1)                     { $v= $sum*$point + $v1; }
  | coeffs($sum, $point, $v1)
    ',' exp($v2)                 { $v= $v1*$point + $v2; }
  ;

exp(double $v)
  : exp($v1) '@'
    '[' coeffs(0.0, $v1, $v) ']'
```

`polycalc` is derived from `mfcalc`. Besides adding an operator declaration for `@` as described above, these rules represent the only other addition needed to `mfcalc`. `coeffs` is a new nonterminal used to represent a comma-separated list of polynomial coefficient expressions. It has three semantic attributes: the first two are inherited attributes `$sum` and `$point` and correspond to the variables with the same name used in the `evalPoly` C-function. The last attribute is a synthesized attribute declared using `%out` (the `%out` is *usually* optional) which is the value of the entire polynomial at point.

The first rule for `coeffs`

```
coeffs(%in double $sum, %in double $point, %out double $v)
  : exp($v1)                     { $v= $sum*$point + $v1; }
```

deals with the situation where there is only a single polynomial coefficient which has not been processed: in that case the value of the polynomial is the value of the polynomial so far (`$sum`) multiplied by the value of the point (`$point`) plus the value of the coefficient (`$v1`).

The second rule

```
coeffs(%in double $sum, %in double $point, %out double $v)
  : coeffs($sum, $point, $v1)
    ',' exp($v2)                 { $v= $v1*$point + $v2; }
```

deals with the situation when there is more than one remaining coefficient. In that case, the remaining coefficients can be decomposed into all but the last coefficient (described by the first `coeffs` on the right-hand side) and the last coefficient (described by the `','` followed by `exp`). If we assume that the value of the polynomial represented by all but the last coefficient is `$v1`, then the value of the entire polynomial is `$v1*$point + $v2` where `$v2` is the value of the last coefficient.

Notice that the computation performed by the above rules is identical to the computation performed within the **for**-loop of the `evalPoly` function. The initialization of `sum` to `0` is performed in the rule for `exp`

```
exp(double $v)
  : exp($v1) '´
    '[' coeffs(0.0, $v1, $v) ']'
```

which passes in a value of `0.0` for the `$sum` inherited attribute of `coeffs` and the value of the point (`$v1`) for the `$point` inherited attribute. The synthesized attribute `$v` computed for `coeffs` is the value of the left-hand side `exp`.

2.7 A Calculator Which Omits Some Parentheses

The previously developed multi-function calculator `mfcalc` requires its (single) function arguments to be within parentheses. Consider enhancing it for lazy people who prefer not to type those parentheses, as illustrated by the following interaction log:

```
$
lazycalc
exp ln 7
        7
pi = 3.141592653589
        3.141592654
sin pi/4
        0.7071067812
(sin pi/4)^2 + (cos pi/4)^2
        1
$
```

A first attempt may be to simply declare a right-associative precedence level for function application between that of binary addition and multiplication operators as shown below (without any semantic attributes):

```
 ...
%right '='
%left '-' '+'
%right FN                   /* Prefix function application. */
%left '*' '/'
%left NEG                   /* Negation--unary minus */
%right '^'                  /* Exponentiation */
 ...
%%
 ...
exp
  : ID_TOK exp %prec FN
 ...
```

Unfortunately, a little reflection shows that this is not enough. Given simply the token sequence `ID_TOK - ID_TOK`, a human would not know whether the sequence represents a function (given by the first `ID_TOK`) applied to a unary minus expression (`- ID_TOK`), or a subtraction of the variable represented by the second `ID_TOK` from the variable represented by the first `ID_TOK`. If a human being cannot distinguish between these two meanings purely on the basis of syntax, then there is no way that Zyacc can do so.

The way a human would distinguish between the above two sequences would be to consider the semantics for the first `ID_TOK`. If the first `ID_TOK` represents a function, then the sequence would correspond to a function application, else to a subtraction expression. However, this means that the way a token sequence is parsed depends on the semantics of the tokens in the sequence.

Generic yacc does not allow any way for the semantics to affect a parse. However, Zyacc allows arbitrary semantic predicates to affect a parse. A *semantic predicate* can be any arbitrary C expression E written on the right-hand side of a rule as `%test(`E`)`. If at parse time, the rule in which a semantic predicate occurs is potentially applicable, then the predicate is evaluated: if the predicate succeeds (returns non-zero), then the rule in which the `%test` is embedded wins out over other rules. So for our `lazycalc` example, the rule for function application becomes

```
exp(double $v)
  : ID_TOK($id)  %test(isFn($id))
    exp($v1) %prec FN           { $v= (*(getIDFn($id)))($v1); }
```

`isFn()` is a trivial C function which interfaces to the symbol table, returning 1 iff its argument is a function.

```
/* Return nonzero iff name is a function. */
static unsigned
isFn(const char *name)
{
  Sym *p= getSym(name, 0);
  return (p != NULL && p->type == FN_SYM);
}
```

Semantic predicates allow relatively clean solutions to problems which are otherwise rather painful to solve. They should not be overused. In particular, they should not be used when simpler mechanisms suffice as they make a Zyacc grammar harder to understand — to understand Zyacc's parsing decisions when semantic predicates are used it is no longer sufficient to merely consider the statically-defined grammar rules, but it is also necessary to consider the semantics defined at parse time.

2.8 Exercises

These example programs are 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. The following exercises suggest several simple enhancements.

1. Add some new functions from '`math.h`' to the initialization list for `mfcalc`.
2. Modify `mfcalc` to add another array that contains constants and their values. Then modify `initSyms` to add these constants to the symbol table. It will be easiest to give the constants type `VAR_SYM`.
3. Modify your solution to the previous exercise, so that the values of constants cannot be modified. (Hint: introduce a new symbol type `CONST_SYM`).
4. Is it possible to modify `mfcalc` so as to allow implicit multiplication: i.e. your calculator should allow `2 + 3 x` as equivalent to `2 + 3*x`?
5. How would you modify `mfcalc` to add array variables such that an array is accessed using *arrayName*`(`*index*`)` where *arrayName* is an `ID_TOK` representing the name of the array and *index* is an `exp` representing the value used to index the array. Syntactically, an array access is identical to a function application, and your answer should concentrate on how your parser can distinguish between the two.

3 Zyacc Grammar Files

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

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

3.1 Outline of a Zyacc Grammar

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

```
%{
C declarations
%}

Zyacc 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 `yyparse`. 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 Zyacc Declarations Section

The *Zyacc 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 [Zyacc Declarations], page 46.

3.1.3 The Grammar Rules Section

The *grammar rules* section contains one or more Zyacc grammar rules, and nothing else. See Section 3.3 [Syntax of Grammar 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 `yyparse`. For example, the definitions of `yylex` and `yyerror` often go here. See Chapter 4 [Parser C-Language Interface], page 54.

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

The Zyacc 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 Zyacc 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 Zyacc 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 Zyacc declaration such as `%token`. See Section 3.6.1 [Token Type Names], page 47.
- 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 [Data Types of Semantic Values], page 35), associativity, or precedence (see Section 5.3 [Operator Precedence], page 64).

  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 Zyacc 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 4.2.1 [Calling Convention for `yylex`], page 54).

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 4.2.1 [Calling Convention for `yylex`], page 54.

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 Zyacc, 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 9 [Invoking Zyacc], page 86.

The symbol `error` is a terminal symbol reserved for error recovery (see Chapter 6 [Error Recovery], page 71); you shouldn't use it for any other purpose. In particular, `yylex` should never return this value.

3.3 Syntax of Grammar Rules

A Zyacc 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 Zyacc 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 Zyacc 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 [The Zyacc Parser Algorithm], page 62, 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 Defining Language Semantics

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 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 [Reverse Polish Notation Calculator], page 8).

Zyacc'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 [Outline of a Zyacc Grammar], page 31).

Only types mentioned in a explicit `%union` declaration and types used for named attributes (see Section 3.5.6 [Named Attributes], page 39) for *terminal* symbols form part of `YYSTYPE`. Hence if you use named attributes for nonterminals, then those associated types do not form part of `YYSTYPE`. Since `YYSTYPE` is often used by a front-end component like a scanner, and the types used for nonterminals are more back-end oriented, the fact that the nonterminal types are not part of `YYSTYPE`, avoids an egregious coupling between front and back ends.

3.5.2 More Than One Value Type

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, Zyacc requires you to do two things:

- Specify the entire collection of possible data types, with the `%union` Zyacc declaration (see Section 3.6.4 [The Collection of Value Types], page 48).
- 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` Zyacc declaration (see Section 3.6.1 [Token Type Names], page 47) and for groupings with the `%type` Zyacc declaration (see Section 3.6.5 [Nonterminal Symbols], page 49).

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 [Actions in Mid-Rule], page 37).

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 `$$`. (Zyacc 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`.

If you don't specify an action for a rule, Zyacc supplies a default: `$$ = $1`. Thus, the value of the first symbol in the rule becomes the value of the whole rule. Of course, the default rule is valid only if the two data types match. There is no meaningful default action for an empty rule; every empty rule must have an explicit action unless the rule's value does not matter.

`$`*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; }
  ;
```

`$1` and `$3` 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. The action can set its value with an assignment to `$$`, and actions later in the rule can refer to the value 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.

There is no way to set the value of the entire rule with a mid-rule action, because assignments to `$$` do not have that effect. The only way to set the value for the entire rule is with an ordinary action at the end of the rule.

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 Tokens], page 62.)

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 Zyacc does not realize that the two actions are identical. (Zyacc 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 Zyacc 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 Zyacc 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 Zyacc actually does to implement mid-rule actions.

3.5.6 Named Attributes

In addition to the `$`*i* numbered attribute notation discussed previously, Zyacc allows the attributes of grammar symbols to be referred to using named attributes. The first character in a attribute name must be '$' and the remaining characters can consist of alphanumerics or the underscore character '_'. The syntax for using named attributes is similar to that for function parameters in C.

Declarations for named attributes result in automatic additions of fields to the `%union` declaration (see Section 3.5.2 [Multiple Types], page 35). In fact, there is no need for a `%union` declaration if the grammar contains only named attributes without any numbered attributes being used at all.

An example of the use of named attributes in contained in the `nmcalc` program discussed in the tutorial section (see Section 2.5 [MfCalc with Sugared Syntax], page 24).

3.5.6.1 Named Terminal Attributes

The named attributes for terminal symbols must be declared in a parenthesized list following the `<`*type*`>` tag in a `%token`, `%left`, `%right` or `%nonassoc` declaration. Here are some examples of terminal attribute declarations:

```
%token  <id>(const char *$id, unsigned $lineNum) ID_TOK
%left   <addOp>(unsigned $lineNum)               '+' '-'
```

The first example declares that the terminal `ID_TOK` has its semantic attributes packaged together with type tag `<id>`. Each such `<id>` package contains two attributes: an attribute named `$id` of type `const char *` and an attribute named `$lineNum` of type `unsigned`. Similarly, the second example declares that the tokens `'+'` and `'-'` are left-associative with their attributes packaged together in packages of type tag `<addOp>`. Each such package contains a single `unsigned` attribute named `$lineNum`.

Each occurrence of `<`*type*`>(`*Type1* `$`*a1*`,` ..., *TypeM* `$`*aM*`)` results in the field `struct {`*Type1* *a1*`;` ...; *TypeM* *aM*`;}` *type*`;` being added to the `%union` declaration (see Section 3.5.2 [Multiple Types], page 35). For instance, the above two examples would add the following fields to the `%union`:

```
struct { const char *id; unsigned lineNum; } id;
struct { unsigned lineNum; } addOp;
```

The above fields would be used within the scanner to ensure that the correct semantic value is passed via `yylval` (see Section 4.2.2 [Token Values], page 55). For the above examples, appropriate assignments may be

```
yylval.id.id= ...; yylval.id.lineNum= ...;
...
yylval.addOp.lineNum= ...;
```

3.5.6.2 Named Nonterminal Attributes

When a nonterminal is used on the left-hand side of a rule, its semantic attributes should be declared within a parenthesized list following that nonterminal. For example, a comma-separated list of identifiers (represented by `ID_TOK`s) may have two attributes consisting of a list of the identifiers and the length of the list respectively:

```
idList(List *$idList, unsigned $len)
  : ID_TOK($id)
        { $idList= appendToList(NULL, $id); $len= 1; }
  | idList($idList1, $len1) ID_TOK($id)
        { $idList= appendToList($idList1, $id); $len= $len1 + 1; }
```

where `appendToList()` appends its second argument to the list represented by its first argument.

If the same nonterminal is used on the left-hand side of several rules, then though the attributes may have different names, their types must be identical, modulo spelling of whitespace. Hence the above rules could be written as:

```
idList(List *$idList, unsigned $len)
  : ID_TOK($id)
        { $idList= appendToList(NULL, $id); $len= 1; }
  ;
idList(List    *$idList2, unsigned    $len2)
  : idList($idList1, $len1) ID_TOK($id)
        { $idList2= appendToList($idList1, $id); $len2= $len1 + 1; }
  ;
```

However

```
idList(List *$idList, unsigned $len)
  : ID_TOK($id)
        { $idList= appendToList(NULL, $id); $len= 1; }
  ;
idList(List*$idList2, unsigned $len2)
  : idList($idList1, $len1) ID_TOK($id)
        { $idList2= appendToList($idList1, $id); $len2= $len1 + 1; }
  ;
```

would be illegal as the second declaration has no whitespace after the '`List*`'.

The types using named attributes for nonterminal symbols are not added to the `YYSTYPE` declaration, but are merely maintained internally within the generated parser. This has the advantage that the scanner need not know about types used only for nonterminal attributes.

3.5.7 Inherited Attributes

The numeric `$`*i* variables in a generic yacc are a form of synthesized attributes — they allow a grammatical construct to pass information up to its surrounding context. Inherited attributes allow a construct to inherit information from its surrounding context. Unfortunately, yacc supports inherited attributes only in a very limited and dangerous way: the programmer uses `$`*i* variables with *i* `<= 0` (see Section 3.5.3 [Actions], page 35).

Hence using yacc is like using a programming language where procedures are not allowed to have input parameters. The only method of passing information into such procedures is by using global variables — the exact method used by many yacc programs.

3.5.7.1 Inherited Attributes Notation

Zyacc supports a form of inherited attributes which can be evaluated during LR parsing — grammars with such attributes are called LR-attributed grammars. LR-attributed grammars are a subset of the L-attributed grammars (where all attributes can be evaluated in a single left-to-right pass). On the other hand, they are a superset of the S-attributed grammars (those supported by yacc) which permit only synthesized attributes.

An inherited attribute is declared by starting the declaration with the `%in` keyword as in:

```
idList(%in int $type, unsigned $len)
  : idList($type, $len1)
    ID_TOK($id)
    { addIDToSymTab($id, $type); $len= $len1 + 1; }
  | /* empty */
    { $len= 0; }
  ;
```

where `$type` is an inherited attribute defining the type for all the identifiers (`ID_TOK`s) in a list of identifiers (`idList`), `$len` is a synthesized attribute giving the number of identifiers in the list and `addIDToSymTab(`*id*`,` *type*`)` adds identifier *id* with type *type* to the symbol table.

This rule could represent a list of identifiers within a declaration, with their type being inherited from the context established by the declaration. That type is represented by the inherited attribute $type. The synthesized attribute $len computes the number of identifiers in the list. $type is passed down unchanged through the recursive nonterminal IDList, being used to install the type of an identifier into the symbol table at every stage of the recursion. $len is initialized to 0 for the empty identifier list and then incremented for every identifier in the list. This pattern of attribute passing is similar to the pattern of parameter passing in any applicative programming language.

A synthesized attribute is declared by starting a declaration with the `%out` keyword. An attribute declaration must have a `%in` or `%out` keyword only if the declaration is the first declaration for a left-hand side nonterminal and the previous rule was not terminated by a '`;`'. If a declaration does not have a `%in` or `%out` keyword, then the omitted keyword defaults to `%out`.

Terminal symbols can only have synthesized attributes.

3.5.7.2 Attribute Flow Restrictions

Since the attributes in a Zyacc grammar are intended to be evaluated completely during a left-to-right parse, there are certain restrictions on the information flow among the attributes. To understand these restrictions, one first needs to understand which attribute occurrences *define* attribute values and which attribute occurrences *apply* a value.

The *defining attribute positions* in a rule are the positions on the left-hand side whose attributes have been declared %in and the positions on the right-hand side whose attributes have been declared %out. The values of the corresponding attributes are computed outside the rule.

The *applied attribute positions* in a rule are the positions on the left-hand side whose attributes have been declared %out and the positions on the right-hand side whose attributes have been declared %in. The values of the corresponding attributes are computed within the rule.

We now describe some (fairly natural) restrictions on the pattern of attributes in a rule.

- On the left-hand side of a rule, we are only allowed to have distinct attribute declarations for the attributes of the rule nonterminal.
- On the right-hand side of a rule, we are allowed to have attribute expressions (any C-expression involving some (or possibly no) named attributes subject to the following restrictions.
  1. An attribute expression occurring in a %out attribute position of a right-hand side grammar symbol (a defining position) can only consist of a single attribute variable. Furthermore, this must be the first occurrence of that attribute variable on the right-hand side.
  2. An attribute expression occurring in a %in attribute position of a right-hand side grammar symbol (a applied position) can only contain attribute variables which have occurred earlier in defining positions of the rule (earlier %out positions in the right-hand side or %in position on the left-hand side).
  3. A particular attribute variable can only occur once in a defining attribute position.
  4. The final terminating action (if any) of a rule can only contain attribute variables which have occurred earlier in the rule.
  5. If a mid-rule action (see Section 3.5.5 [Mid-Rule Actions], page 37) contains the first occurrence of an attribute variable (this would make it a %out attribute for the newly created nonterminal), then that attribute variable must occur as the attribute expression for some later nonterminal on the right-hand side.

Rules 1 and 2 express the left-to-right attribute flow. Rule 3 ensures that all attributes are well-defined: i.e. each attribute only receives a single value within a rule. Rule 4 expresses the fact that it would not make any sense to compute an attribute in the final terminating action without being able to pass the value somewhere. Rule 5 allows zyacc to infer a type for the %out attribute variable of the newly created nonterminal.

By having both attribute declarations and the above rules, there is some redundancy. In fact, zyacc uses the above rules to infer the `%in/%out` specifier for an attribute and checks for consistency with the declaration.

The above rules preclude the possibility of ensuring that the attribute values computed at two defining positions are the same by merely using the same attribute variable at those positions. Consider the rule

```
exp(%out Type $type)
  : exp($type) operator($type) term($type)
  ;
```

where the single attribute of `operator` is declared `%out`. The intent here is that the type of the `operator` must be identical to the types of its two operands. Unfortunately, since all positions on the right-hand side are defining positions, the above rule violates rule 3. We can rewrite the above rule as

```
exp(%out Type $type)
  : exp($type1) operator($type) term($type2)
    { if ($type1 != $type || $type2 != $type) error(); }
  ;
```

or use semantic tests (see Section 3.5.8 [Semantic Tests], page 43).

3.5.7.3 Attribute Conflicts

Zyacc evaluates the values of inherited attributes during parsing. If the value of an inherited attribute cannot be evaluated unambiguously, then Zyacc signals an attribute conflict.

Consider rules like the following:

```
s(%in int $a)
  : a($a + 1) ...
  | a($a + 2) ...
```

When the parser is in a state where it is looking for a `s`, it is clearly also in a state where it is looking for a `a`. It needs to evaluate the inherited attribute for `a`, but without knowing which rule is going to succeed (which may require unbounded lookahead), it does not know whether to evaluate it as `$a + 1` or `$a + 2`. This attribute value conflict is reported as an error by zyacc.

In the limited experience obtained so far with Zyacc, attribute conflicts are usually not much of a problem to work around. A common situation is the following:

```
s
  : a(1) ...
  | a(1) ...
```

Here there is really no conflict because equal attribute values are being passed to both uses of `a`. However, since Zyacc does not understand any C it does not know that the two values are equal and signals a attribute conflict. A simple workaround is to use:

```
s
  : a1 ...
  | a1 ...
  ;
a1
  : a(1)
  ;
```

This workaround is similar to the workaround needed in yacc when a reduce-reduce conflict arises because identical internal actions in same rules are reduced in the same state (see Section 5.6 [Reduce/Reduce], page 67).

3.5.8 Semantic Tests

It is often necessary to make a parsing decision based on the semantics of what is being parsed. Though generic yacc does not allow the semantics to affect the parse, Zyacc does provide a method by which the outcome of runtime semantic tests can affect parsing decisions.

We first present a prototypical example where semantic tests are useful in making a parsing decision. We then present the details of how semantic tests operate. Finally, we present another example where semantic tests can be used to implement dynamically defined operator precedences and associativity. Another example of the use of semantic tests can be found in the `lazycalc` program in the tutorial section (see Section 2.7 [Lazy Calc], page 29).

3.5.8.1 A Prototypical Example

Consider a language which uses an expression like *identifier*`(`*arguments*`)` to express both a function call and indexing of a array. Consider writing a 1-pass compiler for this language using a on-the-fly code-generation strategy (the parser emits the code as it parses). Since different code would need to be emitted for indexing an array versus passing arguments to a function, the parser would need to distinguish between the two situations before the *arguments* are parsed. However, since both situations are syntactically identical, the parser would require semantic information to disambiguate between them. Within the limited facilities of yacc, the programmer typically resorts to a time-honored lexical hack: if the scanner is about to deliver an identifier, it looks ahead to see if the next token is a left parenthesis; if so, it looks up the symbol table and deliver either a `FUNCTION_ID_TOK` or `ARRAY_ID_TOK` depending on the kind of the identifier. In contrast, zyacc can use the following schema to solve this problem:

```
exp
  : fnID '(' indexExpList ')'
  | arrayID '(' fnArgList ')'
  ;
fnID
  : ID_TOK($id) %test(idKind($id) == FN_KIND)
  ;
arrayID
  : ID_TOK($id) %test(idKind($id) == ARRAY_KIND)
  ;
```

In the above we have assumed that the terminal `ID_TOK` has a single `%out` attribute containing the identifier, and `idKind()` which returns the current kind of an identifier is part of the interface to the symbol-table. The symbol table lookup has been shifted into the parser and any required lookahead analysis is handled by the parser generator.

Similar syntactic ambiguities occur in many languages. One of the more notorious is the `typedef` problem in C, where a `typedef` identifier is redefined within an inner scope (see Section 7.1 [Semantic Tokens], page 73). Lexical hacks for that situation are rather complex and/or incorrect. It is possible that the above %test facility could provide a relatively clean solution.

3.5.8.2 Semantic Test Details

Syntactically, each occurrence of a `%test(`*E*`)` can be regarded as a unique empty nonterminal.

The expression *E* within the `%test(`*E*`)` can be any C-expression. If the value of the test expression is 0, then the test fails, otherwise it succeeds. If the test expression contains attribute variables, then those attribute variables are required to have had a previous defining occurrence in the rule (this restriction may be lifted in the future to allow the programmer to write tests like `%test(($result= expensiveLookup()) == SOME_VALUE)`).

When the parser enters a particular state, all the applicable `%tests` for that state are executed sequentially in the order of their occurrence in the source program. If any of these `%tests` succeed, then the parser changes state as though it had succeeded in parsing the nonterminal corresponding to the succeeding `%test`. If all the `%tests` fail, then the parser takes whatever action it would have taken if rules with the applicable `%tests` had been deleted from that state.

When executing `%tests`, zyacc guarantees that its lookahead `yychar` is well-defined. This makes it possible to let the semantic actions in a `%test` refer to the lookahead.

3.5.8.3 Semantic Test Example

The fact that the lookahead token `yychar` is well-defined whenever a `%test` is evaluated makes possible the following overkill for the classic arithmetic expression grammar:

```
/* Test for %tests. Resolve operator priorities using %tests. */

%token <val>(int $v) DIGIT

%{
#include <ctype.h>
#include <stdio.h>

static unsigned char pri[128];
void yyerror();
int yylex();

enum { ADD_P= 1, MULT_P, UMINUS_P };

%}

%%
Lines
  : Lines Exp($v) '\n' { printf("%d\n", $v); }
  | error '\n'
  | /* empty */
  ;
Exp(int $v)
  : Exp($v1) '+' Exp($v2) %test(pri['+'] >= pri[yychar])
      { $v= $v1 + $v2; }
  | Exp($v1) '-' Exp($v2) %test(pri['-'] >= pri[yychar])
      { $v= $v1 - $v2; }
  | Exp($v1) '*' Exp($v2) %test(pri['*'] >= pri[yychar])
      { $v= $v1 * $v2; }
  | Exp($v1) '/' Exp($v2) %test(pri['/'] >= pri[yychar])
      { $v= $v1 / $v2; }
  | '-' Exp($v1) %test(UMINUS_P >= pri[yychar])
      { $v= -$v1; }
  | '+' Exp($v1) %test(UMINUS_P >= pri[yychar])
      { $v= $v1; }
  | DIGIT($v)
```

```
37   | '(' Exp($v) ')'
38   ;
39
40 %%
41
42 int yylex() {
43   int c= getchar();
44   while (isspace(c) && c != '\n') c= getchar();
45   if (c == EOF) return 0;
46   else if isdigit(c) { yylval.val.v= c - '0'; return DIGIT; }
47   else return c;
48 }
49
50 void yyerror(const char *s) {
51   printf("%s\n", s);
52 }
53
54 int main() {
55   pri['+']= pri['-']= ADD_P;
56   pri['*']= pri['/']= MULT_P;
57   return yyparse();
58 }
```

The code above examines the lookahead to instruct the parser whether to shift or to reduce (see Chapter 5 [Algorithm], page 62). The rules for the operators on lines 23 through 36 decide to reduce if the operator involved in the rule has priority greater than or equal to the priority of the incoming symbol in yychar. Otherwise the lookahead is shifted.

The usual method for parsing arithmetic expressions is to either use multiple levels of nonterminals like `exp`, `term`, `factor` to specify the precedence levels, or to specify static disambiguating priorities for the operators using yacc's %left, %right and %nonassoc declarations (see Section 2.2 [Infix Calc], page 15). The method in the example is similar to the second alternative but disambiguates dynamically using the operator priorities at parse time. This makes it possible to write parsers for languages whose syntax can vary during parsing. The Prolog parser distributed with this package uses the ideas outlined by the above example.

3.6 Zyacc Declarations

The *Zyacc declarations* section of a Zyacc grammar defines the symbols used in formulating the grammar and the data types of semantic values. See Section 3.2 [Symbols], page 32. Additionally, a `%look` directive is allowed within each grammar rule to declare the lookahead properties of the grammar rule (see Section 3.6.11 [Specifying the Lookahead], page 52).

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 [More Than One Value Type], page 35) from a explicit `%union` declaration; you should not declare the types of a non-terminal if it uses named attributes (see Section 3.5.6 [Named Attributes], page 39).

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 [Languages and Context-Free Grammars], page 3).

3.6.1 Token Type Names

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

```
%token name
```

Zyacc 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 [Operator Precedence], page 47.

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_TOK 300
```

It is generally best, however, to let Zyacc choose the numeric codes for all token types. Zyacc 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 [More Than One Value Type], page 35).

For example:

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

3.6.2 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 [Operator Precedence], page 64, 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`. Like `%token`, they also allow alternate names for a token so that a token can be referred to using a multi-character literal name within the grammar file. 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 Multi-Character Literal Tokens

Zyacc allows you to specify both a literal name for a token as well as an identifier (as above). Zyacc consistently uses single quotes to delimit literal names (as opposed to Bison's use of single quotes for single character literals and double quotes for multi-character literals). Within the declaration, the two alternate names for a token may occur in either order, separated by a '`=`'. This allows you to refer to a token internally within the grammar file using the possibly more readable literal name, while using the identifier name to refer to the token in other files. For example, in

```
%left                 '<<=' = LSH_ASSGN  RSH_ASSGN = '>>=' 300
%token <lineNum>       'for' = FOR_TOK
```

the first declaration defines two tokens. The first token has two alternate names `'<<='` and `LSH_ASSGN`; since a value has not been specified for it, Zyacc will choose a value. The second token has two alternate names `RSH_ASSGN` and `'>>='`. Its value has been specified as 300.

The second declaration illustrates the fact that the alternate names can even be used with declarations which involve a `<`*type*`>` tag.

With the above declarations the grammar can contain rules like:

```
stmt
  : 'for' '(' exp ';' exp ';' exp ')' stmt
  ;
exp
  : exp '<<=' exp
  | exp '>>=' exp
  ;
```

Whether the above is more readable than the alternative using `LSH_ASSGN`, `RSH_ASSGN` and `FOR_TOK` is a matter of personal preference.

3.6.4 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 3.6.5 [Nonterminal Symbols], page 49).

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

3.6.5 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 3.6.4 [The Collection of Value Types], page 48). 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.

3.6.6 Suppressing Conflict Warnings

Zyacc normally warns if there are any conflicts in the grammar (see Section 5.2 [Shift/Reduce Conflicts], page 63), 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. Zyacc will also print the number of conflicts.
- Check each of the conflicts to make sure that Zyacc'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 Zyacc printed.

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

3.6.7 The Start-Symbol

Zyacc 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*

3.6.8 Multiple Start Symbols

Unlike other yaccs, Zyacc allows the user to declare more than one start nonterminal. If the user does so, then `#define` statements are added to the generated parser for each start nonterminal and the programmer can call the generated parser function (see Section 4.1 [Parser Function], page 54) with its first parameter set to the desired start nonterminal. If a pure parser has also been requested using the `pure_parser` directive (see Section 4.2.4 [Pure Calling], page 57), then the additional arguments for the pure parser follow the start nonterminal in the argument list for the parsing function.

The following example uses this feature to parse a line as either a prefix, infix or suffix arithmetic expression depending on the first character in the line. It accepts lines like the following:

```
i(1+2)*3+4
s12+3*4+
p+*+1234
```

The grammar uses the `%look` directive (see Section 3.6.11 [Specifying the Lookahead], page 52) to ensure that the lookahead is clear after each call to the parsing function. Here it is:

```
%start infix, prefix, suffix

%%
infix
  : iExp '\n' %look(0) { return $1; }
  ;
iExp
  : iExp '+' iTerm { $$= $1 + $3; }
  | iTerm
  ;
iTerm
  : iTerm '*' iFactor { $$= $1 * $3; }
  | iFactor
  ;
iFactor
  : digit
  | '(' iExp ')' { $$= $2; }
  ;
digit
  : '0' { $$= 0; }
  | '1' { $$= 1; }
  | '2' { $$= 2; }
  | '3' { $$= 3; }
  | '4' { $$= 4; }
  | '5' { $$= 5; }
  | '6' { $$= 6; }
  | '7' { $$= 7; }
  | '8' { $$= 8; }
  | '9' { $$= 9; }
  ;

prefix
  : pExp '\n' %look(0) { return $1; }
  ;
pExp
  : '+' pExp pExp { $$= $2 + $3; }
  | '*' pExp pExp { $$= $2 * $3; }
  | digit
```

```
    ;

suffix
  : sExp '\n' %look(0) { return $1; }
  ;
sExp
  : sExp sExp '+' { $$= $1 + $2; }
  | sExp sExp '*' { $$= $1 * $2; }
  | digit
  ;
```

The scanner function simply returns the next non-blank input character. The parser driver shown below decides on which start nonterminal to use depending on the first character of each input line:

```
int main()
/* Call infix grammar if line starts with 'i'; suffix grammar if line
 * starts with a 's'; prefix grammar if line starts with a 'p'.
 */
{
  int c;
  while ((c= getchar()) != EOF) {
    int z= yyparse(c == 'i' ? infix : (c == 's' ? suffix : prefix));
    printf("%d\n", z);
  }
  return 0;
}
```

3.6.9 A Pure (Reentrant) Parser

A *reentrant* program is one which does not alter in the course of execution; in other words, it consists entirely of *pure* (read-only) code. 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 Zyacc parser is not normally a reentrant program, because it uses statically allocated variables for communication with `yylex`. These variables include `yylval` and `yylloc`.

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

```
%pure_parser
```

The effect is that the two communication variables become local variables in `yyparse`, and a different calling convention is used for the lexical analyzer function `yylex`. See Section 4.2.4 [Calling Conventions for Pure Parsers], page 57, for the details of this. The variable `yynerrs` also becomes local in `yyparse` (see Section 4.3 [The Error Reporting Function `yyerror`], page 59). The convention for calling `yyparse` itself is unchanged.

3.6.10 Option Declaration

The Zyacc declarations section can contain `%option` directives followed by command-line options using only the form with long option names (see Chapter 9 [Invoking Zyacc], page 86), with any option value specified within the same word as the option name. The options actually specified on the command-line override the options specified in the source file using this directive. `%option` directives must precede all other directives.

3.6.11 Specifying the Lookahead

When a Zyacc grammar rule is reduced, the parser may or may not require a lookahead token from the scanner before it can make the reduction decision. This can make a difference is situations where the scanner and parser are very tightly coupled, as when they are both accessing the same file. Typically, one needs to look at the generated parser description file(s) to check the relative states of the parser and scanner. Zyacc provides the `%look` directive to automate this process.

Unlike other declarations, `%look` declarations occur within the rules in section 2 of the source file. If the body of a rule contains the declaration `%look(0)`, then that specifies that the rule is **always** reduced in a context which does not require a lookahead token from the scanner. If the body of a rule contains the declaration `%look(1)`, then that specifies that the rule is **always** reduced in a context which does require a lookahead token from the scanner. If these specifications are violated, then Zyacc outputs a warning message at parser generation time.

The advantage of these declarations is that it makes grammars more maintainable. Rather than having to manually redo an involved analysis regarding lookahead decisions each time a grammar is modified, Zyacc does the analysis whenever it processes the grammar and reports any violations.

An example of the use of the `%look(0)` directive to ensure that the lookahead is clear before the parsing function returns can be found in Section 3.6.8 [Multiple Start Symbols], page 49.

3.6.12 Zyacc Declaration Summary

Here is a summary of all Zyacc declarations:

`%union` Declare the collection of data types that semantic values may have (see Section 3.6.4 [The Collection of Value Types], page 48).

`%token` Declare a terminal symbol (token type name) with no precedence or associativity specified (see Section 3.6.1 [Token Type Names], page 47).

`%right` Declare a terminal symbol (token type name) that is right-associative (see Section 3.6.2 [Operator Precedence], page 47).

`%left` Declare a terminal symbol (token type name) that is left-associative (see Section 3.6.2 [Operator Precedence], page 47).

`%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 [Operator Precedence], page 47).

`%type` Declare the type of semantic values for a nonterminal symbol (see Section 3.6.5 [Nonterminal Symbols], page 49).

`%start` Specify the grammar's start symbol (see Section 3.6.7 [The Start-Symbol], page 49).

`%expect` Declare the expected number of shift-reduce conflicts (see Section 3.6.6 [Suppressing Conflict Warnings], page 49).

`%pure_parser`
Request a pure (reentrant) parser program (see Section 3.6.9 [A Pure (Reentrant) Parser], page 51).

`%option` Specify a command-line option from within the grammar file (see Section 3.6.10 [Option Decl], page 52).

3.7 Multiple Parsers in the Same Program

Most programs that use Zyacc parse only one language and therefore contain only one Zyacc parser. But what if you want to parse more than one language with the same program? Then you need to avoid a name conflict between different definitions of `yyparse`, `yylval`, and so on.

The easy way to do this is to use the option '`-p` *prefix*' (see Chapter 9 [Invoking Zyacc], page 86). This renames the interface functions and variables of the Zyacc parser to start with *prefix* instead of '`yy`'. You can use this to give each parser distinct names that do not conflict.

The precise list of symbols renamed is `yyparse`, `yylex`, `yyerror`, `yynerrs`, `yylval`, `yychar` and `yydebug`. For example, if you use '`-p c`', the names become `cparse`, `clex`, and so on.

All the other variables and macros associated with Zyacc are not renamed. These others are not global; there is no conflict if the same name is used in different parsers. For example, `YYSTYPE` is not renamed, but defining this in different ways in different parsers causes no trouble (see Section 3.5.1 [Data Types of Semantic Values], page 35).

4 Parser C-Language Interface

The Zyacc 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. Zyacc 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.

In simple programs, `yylex` is often defined at the end of the Zyacc 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 Zyacc, 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 9 [Invoking Zyacc], page 86.

4.2.1 Calling Convention for `yylex`

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.

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. See Section 3.2 [Symbols], page 32.

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`.

This interface changes if the `%pure` directive is used to generate a *pure_parser* parser (see Section 4.2.4 [Pure Calling], page 57).

4.2.2 Semantic Values of Tokens

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 Zyacc 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 3.6.4 [The Collection of Value Types], page 48). 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 Zyacc stack. */
  return INT;          /* Return the type of the token. */
  ...
```

4.2.3 Textual Positions of Tokens

If you are using the '`@`*n*'-feature (see Section 4.4 [Special Features for Use in Actions], page 59) 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. This feature has not currently been tested.

The data type of `yylloc` has the name `YYLTYPE`.

Here is a simple example to illustrate the use of this feature. The following grammar accepts lists consisting of only balanced parentheses. For each top-level list, it prints out its starting and ending coordinates. For example, for the inputs:

```
  ( ( ) )

( ( )
  (  )
 )
```

it would print out

```
(1, 2) -- (1, 9)
(3, 0) -- (5, 2)
```

Here is the program:

```
%%
s
  : s list
        { printf("(%d, %d) -- (%d, %d)\n", @2.first_line, @2.first_column,
                 @2.last_line, @2.last_column);
        }
  | /* empty */
  | error
  ;
list
  : '(' listSeq ')'
  | '(' ')'
  ;
listSeq
  : listSeq list
  | list
  ;

%%

static void yyerror(s)
  const char *s;
{
  printf("%s\n", s);
}

static int lineN= 1;
static int colN= 0;
```

```
int
yylex() {
  int c;
  while (isspace(c= getchar())) {
    if (c == '\n') {
      lineN++; colN= 0;
    }
    else {
      colN++;
    }
  }
  if (c == EOF) return 0;
  yylloc.first_line= yylloc.last_line= lineN;
  yylloc.first_column= colN; yylloc.last_column= ++colN;
  return c;
}



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

4.2.4 Calling Conventions for Pure Parsers

When you use the Zyacc declaration `%pure_parser` to request a pure, reentrant parser, the global communication variables `yylval` and `yylloc` cannot be used. (See Section 3.6.9 [A Pure (Reentrant) Parser], page 51.) 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 Zyacc stack.  */
  return INT;      /* Return the type of the token.  */
  ...
}
```

If the grammar file does not use the '`@`' constructs to refer to textual positions, then the type `YYLTYPE` will not be defined. In this case, omit the second argument; `yylex` will be called with only one argument.

You can pass parameter information to a reentrant parser in a reentrant way. Define the macro `YYPARSE_PARAM` as a variable name. The resulting `yyparse` function then accepts one argument, of type `void *`, with that name.

When you call `yyparse`, pass the address of an object, casting the address to `void *`. The grammar actions can refer to the contents of the object by casting the pointer value back to its proper type and then dereferencing it. Here's an example. Write this in the parser:

```
%{
struct parser_control
{
  int nastiness;
  int randomness;
};

#define YYPARSE_PARAM parm
%}
```

Then call the parser like this:

```
struct parser_control
{
  int nastiness;
  int randomness;
};

...

{
  struct parser_control foo;
  ...  /* Store proper data in foo.  */
  value = yyparse ((void *) &foo);
  ...
}
```

In the grammar actions, use expressions like this to refer to the data:

```
((struct parser_control *) parm)->randomness
```

If you wish to pass the additional parameter data to `yylex`, define the macro `YYLEX_PARAM` just like `YYPARSE_PARAM`, as shown here:

```
%{
struct parser_control
{
  int nastiness;
  int randomness;
};

#define YYPARSE_PARAM parm
#define YYLEX_PARAM parm
%}
```

You should then define `yylex` to accept one additional argument—the value of `parm`. (This makes either two or three arguments in total, depending on whether an argument of type `YYLTYPE` is passed.) You can declare the argument as a pointer to the proper object type, or you can declare it as `void *` and access the contents as shown above.

4.3 The Error Reporting Function `yyerror`

The Zyacc parser detects a *parse error* or *syntax error* whenever it reads a token which cannot satisfy any syntax rule. A action in the grammar can also explicitly proclaim an error, using the macro `YYERROR` (see Section 4.4 [Special Features for Use in Actions], page 59).

The Zyacc parser expects to report the error by calling an error reporting function named `yyerror`, which you must supply. It is called by `yyparse` whenever a syntax error is found, and it receives one argument. For a parse error, the string is normally `"parse error"`.

If you define the macro `YYERROR_VERBOSE` in the Zyacc declarations section (see Section 3.1.2 [The Zyacc Declarations Section], page 31), then Zyacc provides a more verbose and specific error message string instead of just plain `"parse error"`. It doesn't matter what definition you use for `YYERROR_VERBOSE`, just whether you define it. `YYERROR_VERBOSE` is not currently implemented; possibly a more general version of it will be implemented in a future version of Zyacc.

The parser can detect one other kind of error: stack overflow. This happens when the input contains constructions that are very deeply nested. It isn't likely you will encounter this, since the Zyacc parser extends its stack automatically up to a very large limit. But if overflow happens, `yyparse` calls `yyerror` in the usual fashion, except that the argument string is `"parser stack overflow"`.

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 71). If recovery is impossible, `yyparse` will immediately return 1.

The variable `yynerrs` contains the number of syntax errors encountered so far. Normally this variable is global; but if you request a pure parser (see Section 3.6.9 [A Pure (Reentrant) Parser], page 51) then it is a local variable which only the actions can access.

4.4 Special Features for Use in Actions

Here is a table of Zyacc 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 [Data Types of Values in Actions], page 36.

'`$<`*typealt*`>`*n*'
Like `$`*n* but specifies alternative *typealt* in the union specified by the `%union` declaration. See Section 3.5.4 [Data Types of Values in Actions], page 36.

'`YYABORT;`'
: Return immediately from `yyparse`, indicating failure. See Section 4.1 [The Parser Function `yyparse`], page 54.

'`YYACCEPT;`'
: Return immediately from `yyparse`, indicating success. See Section 4.1 [The Parser Function `yyparse`], page 54.

'`YYBACKUP (`*token*`, `*value*`);`'
: Unshift a token. This macro is allowed only for rules that reduce a single value, and only when there is no look-ahead token. It installs a look-ahead token with token type *token* and semantic value *value*; then it discards the value that was going to be reduced by this rule.

  If the macro is used when it is not valid, such as when there is a look-ahead token already, then it reports a syntax error with a message '`cannot back up`' and performs ordinary error recovery.

  In either case, the rest of the action is not executed.

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

'`YYERROR;`'
: Cause an immediate syntax error. This statement initiates error recovery just as if the parser itself had detected an error; however, it does not call `yyerror`, and does not print any message. If you want to print an error message, call `yyerror` explicitly before the '`YYERROR;`' statement. See Chapter 6 [Error Recovery], page 71.

'`YYRECOVERING`'
: This macro stands for an expression that has the value 1 when the parser is recovering from a syntax error, and 0 the rest of the time. See Chapter 6 [Error Recovery], page 71.

'`yychar`'
: Variable containing the current look-ahead token. (In a pure parser, this is actually a local variable within `yyparse`.) When there is no look-ahead token, the value `YYEMPTY` is stored in the variable. See Section 5.1 [Look-Ahead Tokens], page 62.

'`yyclearin;`'
: Discard the current look-ahead token. This is useful primarily in error rules. See Chapter 6 [Error Recovery], page 71.

'`yyerrok;`'
: Resume generating error messages immediately for subsequent syntax errors. This is useful primarily in error rules. See Chapter 6 [Error Recovery], page 71.

'`@`*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 Zyacc Parser Algorithm

As Zyacc 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 [Languages and Context-Free Grammars], page 3).

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

5.1 Look-Ahead Tokens

The Zyacc 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 [Special Features for Use in Actions], page 59.

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*. Zyacc 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 Zyacc 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 Zyacc 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 3.6.6 [Suppressing Conflict Warnings], page 49.

The definition of `if_stmt` above is solely to blame for the conflict, but the conflict does not actually appear without additional rules. Here is a complete Zyacc input file that actually manifests the conflict:

```
%token IF THEN ELSE variable
%%
stmt
  : expr
  | if_stmt
  ;

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

expr:     variable
        ;
```

5.3 Operator Precedence

Another situation where shift/reduce conflicts appear is in arithmetic expressions. Here shifting is not always the preferred resolution; the Zyacc 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 Zyacc 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 such as '`1 - 2 - 5`'; should this be '`(1 - 2) - 5`' or should it be '`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 Specifying Operator Precedence

Zyacc 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 in the file 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 [Context-Dependent Precedence], page 66.)

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 9 [Invoking Zyacc], page 86) 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 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 Zyacc 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 precedence 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 [Operator Precedence], page 64).

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 number 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 71).

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

maybeword
  : /* empty */
      { printf ("empty maybeword\n"); }
  | word
      { printf ("single word %s\n", $1); }
  ;
```

The error is an ambiguity: there is more than one way to parse a single `word` into a `sequence`. It could be reduced to a `maybeword` and then into a `sequence` 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 for `sequence`.

There is also more than one way to reduce nothing-at-all into a `sequence`. This can be done directly via the first rule, or indirectly via `maybeword` and then the second 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.

Zyacc 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
```

```
        ;
```

5.7 Mysterious Reduce/Reduce Conflicts

Sometimes reduce/reduce conflicts can occur that don't look warranted. Here is an example:

```
%token ID

%%
def
  : param_spec return_spec ','
  ;
param_spec
  : type
  | name_list ':' type
  ;
return_spec
  : type
  | name ':' type
  ;
type
  : ID
  ;
name
  : ID
  ;
name_list
  : name
  |    name ',' name_list
  ;
```

It would seem that this grammar can be parsed with only a single token of look-ahead: when a `param_spec` is being read, an `ID` is a `name` if a comma or colon follows, or a `type` if another `ID` follows. In other words, this grammar is LR(1).

However, Zyacc, like most parser generators, cannot actually handle all LR(1) grammars. In this grammar, two contexts, that after an `ID` at the beginning of a `param_spec` and likewise at the beginning of a `return_spec`, are similar enough that Zyacc assumes they are the same. They appear similar because the same set of rules would be active—the rule for reducing to a `name` and that for reducing to a `type`. Zyacc is unable to determine at that stage of processing that the rules would require different look-ahead tokens in the two contexts, so it makes a single parser state for them both. Combining the two contexts causes a conflict later. In parser terminology, this occurrence means that the grammar is not LALR(1).

In general, it is better to fix deficiencies than to document them. But this particular deficiency is intrinsically hard to fix; parser generators that can handle LR(1) grammars are hard to write and tend to produce parsers that are very large. In practice, Zyacc is more useful as it is now.

When the problem arises, you can often fix it by identifying the two parser states that are being confused, and adding something to make them look distinct. In the above example, adding one rule to `return_spec` as follows makes the problem go away:

```
%token BOGUS
...
%%
...
return_spec
  : type
  |    name ':' type
    /* This rule is never used.  */
  |    ID BOGUS
  ;
```

This corrects the problem because it introduces the possibility of an additional active rule in the context after the ID at the beginning of return_spec. This rule is not active in the corresponding context in a param_spec, so the two contexts receive distinct parser states. As long as the token BOGUS is never generated by yylex, the added rule cannot alter the way actual input is parsed.

In this particular example, there is another way to solve the problem: rewrite the rule for return_spec to use ID directly instead of via name. This also causes the two confusing contexts to have different sets of active rules, because the one for return_spec activates the altered rule for return_spec rather than the one for name.

```
param_spec
  : type
  | name_list ':' type
  ;
return_spec
  : type
  |    ID ':' type
  ;
```

5.8 Stack Overflow, and How to Avoid It

The Zyacc parser stack can overflow if too many tokens are shifted and not reduced. When this happens, the parser function yyparse returns a nonzero value, pausing only to call yyerror to report the overflow.

By defining the macro YYMAXDEPTH, you can control how deep the parser stack can become before a stack overflow occurs. Define the macro with a value that is an integer. This value is the maximum number of tokens that can be shifted (and not reduced) before overflow. It must be a constant expression whose value is known at compile time.

The stack space allowed is not necessarily allocated. If you specify a large value for YYMAXDEPTH, the parser actually allocates a small stack at first, and then makes it bigger by stages as needed. This increasing allocation happens automatically and silently. Therefore, you do not need to make YYMAXDEPTH painfully small merely to save space for ordinary inputs that do not need much stack.

The default value of YYMAXDEPTH, if you do not define it, is 10000.

You can control how much stack is allocated initially by defining the macro YYINITDEPTH. This value too must be a compile-time constant integer. The default is 200.

6 Error Recovery

It is not usually acceptable to have a 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 Zyacc 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 Zyacc 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, Zyacc 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;`'.

The macro `YYRECOVERING` stands for an expression that has the value 1 when the parser is recovering from a syntax error, and 0 the rest of the time. A value of 1 indicates that error messages are currently suppressed for new syntax errors.

7 Handling Context Dependencies

The Zyacc paradigm is to parse tokens first, then group them into larger syntactic units. In many languages, the meaning of a token is affected by its context. Although this violates the basic Zyacc paradigm, a relatively clean way to handle context dependencies is by using semantic tests (see Section 3.5.8 [Semantic Tests], page 43). This section documents other techniques (known as *kludges*) which may enable you to write Zyacc parsers for such languages.

(Actually, "kludge" means any technique that gets its job done but is neither clean nor robust.)

7.1 Semantic Info in Token Types

The C language has a context dependency: the way an identifier is used depends on what its current meaning is. For example, consider this:

```
foo (x);
```

This looks like a function call statement, but if `foo` is a typedef name, then this is actually a declaration of `x`. How can a Zyacc parser for C decide how to parse this input?

The method used in GNU C is to have two different token types, `IDENTIFIER` and `TYPENAME`. When `yylex` finds an identifier, it looks up the current declaration of the identifier in order to decide which token type to return: `TYPENAME` if the identifier is declared as a typedef, `IDENTIFIER` otherwise.

The grammar rules can then express the context dependency by the choice of token type to recognize. `IDENTIFIER` is accepted as an expression, but `TYPENAME` is not. `TYPENAME` can start a declaration, but `IDENTIFIER` cannot. In contexts where the meaning of the identifier is *not* significant, such as in declarations that can shadow a typedef name, either `TYPENAME` or `IDENTIFIER` is accepted—there is one rule for each of the two token types.

This technique is simple to use if the decision of which kinds of identifiers to allow is made at a place close to where the identifier is parsed. But in C this is not always so: C allows a declaration to redeclare a typedef name provided an explicit type has been specified earlier:

```
typedef int foo, bar, lose;
static foo (bar);        /* redeclare bar as static variable */
static int foo (lose);   /* redeclare foo as function */
```

Unfortunately, the name being declared is separated from the declaration construct itself by a complicated syntactic structure—the "declarator".

As a result, the part of Zyacc parser for C needs to be duplicated, with all the nonterminal names changed: once for parsing a declaration in which a typedef name can be redefined, and once for parsing a declaration in which that can't be done. Here is a part of the duplication, with actions omitted for brevity:

```
initdcl
  : declarator maybeasm '=' init
  | declarator maybeasm
  ;

notype_initdcl
  : notype_declarator maybeasm '=' init
```

```
        | notype_declarator maybeasm
        ;
```

Here `initdcl` can redeclare a typedef name, but `notype_initdcl` cannot. The distinction between `declarator` and `notype_declarator` is the same sort of thing.

There is some similarity between this technique and a lexical tie-in (described next), in that information which alters the lexical analysis is changed during parsing by other parts of the program. The difference is here the information is global, and is used for other purposes in the program. A true lexical tie-in has a special-purpose flag controlled by the syntactic context.

7.2 Lexical Tie-ins

One way to handle context-dependency is the *lexical tie-in*: a flag which is set by Zyacc actions, whose purpose is to alter the way tokens are parsed.

For example, suppose we have a language vaguely like C, but with a special construct '`hex (`*hex-expr*`)`'. After the keyword `hex` comes an expression in parentheses in which all integers are hexadecimal. In particular, the token '`a1b`' must be treated as an integer rather than as an identifier if it appears in that context. Here is how you can do it:

```
%{
int hexflag;
%}
%%
...
expr
  : IDENTIFIER
  | constant
  | HEX '('
      { hexflag = 1; }
    expr ')'
      { hexflag = 0; $$ = $4; }
  | expr '+' expr
      { $$ = make_sum ($1, $3); }
  ...
  ;
constant
  : INTEGER
  | STRING
  ;
```

Here we assume that `yylex` looks at the value of `hexflag`; when it is nonzero, all integers are parsed in hexadecimal, and tokens starting with letters are parsed as integers if possible.

The declaration of `hexflag` shown in the C declarations section of the parser file is needed to make it accessible to the actions (see Section 3.1.1 [The C Declarations Section], page 31). You must also write the code in `yylex` to obey the flag.

7.3 Lexical Tie-ins and Error Recovery

Lexical tie-ins make strict demands on any error recovery rules you have. See Chapter 6 [Error Recovery], page 71.

The reason for this is that the purpose of an error recovery rule is to abort the parsing of one construct and resume in some larger construct. For example, in C-like languages, a typical error recovery rule is to skip tokens until the next semicolon, and then start a new statement, like this:

```
stmt
  : expr ';'
  | IF '(' expr ')' stmt { ... }
  ...
  | error ';' { hexflag = 0; }
  ;
```

If there is a syntax error in the middle of a '`hex (`*expr*`)`' construct, this error rule will apply, and then the action for the completed '`hex (`*expr*`)`' will never run. So `hexflag` would remain set for the entire rest of the input, or until the next `hex` keyword, causing identifiers to be misinterpreted as integers.

To avoid this problem the error recovery rule itself clears `hexflag`.

There may also be an error recovery rule that works within expressions. For example, there could be a rule which applies within parentheses and skips to the close-parenthesis:

```
expr
  : ...
  | '(' expr ')' { $$ = $2; }
  | '(' error ')'
  ...
```

If this rule acts within the `hex` construct, it is not going to abort that construct (since it applies to an inner level of parentheses within the construct). Therefore, it should not clear the flag: the rest of the `hex` construct should be parsed with the flag still in effect.

What if there is an error recovery rule which might abort out of the `hex` construct or might not, depending on circumstances? There is no way you can write the action to determine whether a `hex` construct is being aborted or not. So if you are using a lexical tie-in, you had better make sure your error recovery rules are not of this kind. Each rule must be such that you can be sure that it always will, or always won't, have to clear the flag.

8 Debugging Your Parser

Zyacc provides facilities to get a debugger compiled into your program. The resulting program can be debugged in several different ways:

1. You can interact with the program using a command-line interface and control it using simple single letter debugging commands. This approach has the disadvantage that debugger I/O is interspersed with the I/O of your program.
2. As in (1) above, but separate processes are used to run your program and the debugger interface. These processes communicate using a socket-based interface. Hence the program being debugged can be on a computer different from the one on which you are doing the debugging, or it could be on the same one (the usual case). When debugging and running your program on the same computer, you can run your program in one window and debug it from another window. This avoids the problem in (1) where debugger and program I/O are interspersed.
3. As in (2) above, but the interaction uses a GUI java-based application, rather than a command-line interface. Most of the facilities of the command-line interface are available via the GUI.
4. By specifying certain environmental variables when you run your program, you can get it to generate HTML files which can then be accessed using any Web browser. The GUI used in (3) is then run as a java applet by accessing the generated HTML files using your favorite Web browser. This approach has the advantage that the GUI can automatically use your Web browser to display the current state of the parser in the HTML files generated using the `--HTML` option (see Chapter 9 [Invoking Zyacc], page 86).

8.1 Building Debugging Parsers

To build a parser which can be debugged, it is necessary to define a C preprocessor symbol when compiling the generated parser and to link your program with the Zyacc library. Optionally, if you would like token semantics to be printed out, then you need to define an auxiliary function.

8.1.1 Compiling Debugging Parsers

To enable compilation of debugging facilities, you must define the C preprocessor macro `YY_ZYACC_DEBUG` to be nonzero when you compile the parser. This can be done in any of the following ways:

1. You can define the `YY_ZYACC_DEBUG` macro in the C-declarations section of section 1 of your Zyacc source file.
2. You can use the `-t` or `--debug` option when you run Zyacc (see Chapter 9 [Invoking Zyacc], page 86). This results in a definition of `YY_ZYACC_DEBUG` automatically being added to the generated parser.
3. You can define the `YY_ZYACC_DEBUG` macro on the compiler command line when you compile the generated parser. For most compilers, this can typically be done by specifying the option `-DYY_ZYACC_DEBUG`.

The third approach is the recommended approach as it defers the debugging decision to the latest possible point in the build process.

For compatibility with other parser generators, the macro `YYDEBUG` can be used instead of `YY_ZYACC_DEBUG`. Zyacc provides `YY_ZYACC_DEBUG` in addition to `YYDEBUG` for the following reason: `YYDEBUG` is also used to turn on debugging in lex-based scanner generators; if a project uses both lex and yacc, and its compilation is controlled by a Makefile then it can sometimes be inconvenient to turn on debugging in the parser but not in the scanner, or vice-versa.

8.1.2 Linking Debugging Parsers

A debugging parser must be linked with the Zyacc library. Depending on your system, you may also need to link it with the networking library — typically you need to specify `-lsocket` and `-lnsl` on the link command-line. For most compilers, this is typically done using the `-L` option (for specifying the library path) and the `-l` option (for specifying the library name). For example, if your project consists of a parser with object file `parse.o` and some helper functions in `helpers.o` and the Zyacc library is installed in the `lib` subdirectory of your home directory, then a typical link command line is:

```
cc parse.o helpers.o -L$HOME/lib -lzyacc -lsocket -lnsl -o my_prj
```

It is usually necessary that the `-L` and `-l` options occur *after* the object files.

8.1.3 Printing Token Semantics

If you would like to have the debugger print out tokens using their semantics (rather than the token names), then you need to define a function to print them out. If your function is *semFn*, then it should have the prototype

```
void semFn(FILE *out, int tokNum, void *yylvalP);
```

and should print in `FILE` out, the semantics associated with the token whose token number is `tokNum` and whose `yylval` semantic value (see Section 4.2.2 [Token Values], page 55) is pointed to by `yylvalP`.

You may call this function whatever you like. You should communicate the name of the function to Zyacc by defining (in the C declarations section of the source file) the C preprocessor macro `YY_SEM_FN` to the chosen name.

If you are only using the textual interface, then this function may print anything you wish. However, if you wish to use the GUI, then it is imperative that the function **not print any newlines**.

If you are using the GUI, then you may also want to define the C macro `YY_SEM_MAX` to specify the maximum number of characters your semantic function will print. If you do not define this macro, then a default value is used.

8.2 Environmental Variables

Since the program being run is *your* program and Zyacc has no control over the arguments accepted by your program from the command-line, all arguments to the debugger are provided via environmental variables.

If your shell is derived from the C-shell (`csh`, `tcsh`, etc), then you can specify an environmental variable using the `setenv` command. For example,

```
setenv ZYDEBUG_PORT 1
```

If your shell is derived from the Bourne-shell (`sh`, `ksh`, `bash`, etc), then you can specify an environmental variable using the `export` command. For example,

```
export ZYDEBUG_PORT=1
```

If you prefer to avoid cluttering up your environment with gratuitous definitions, then the `sh`-based derivatives provide a neat alternative. You can simply type the variable definitions immediately before the name of your program, as illustrated by the following example:

```
ZYDEBUG_PORT=1 my_prg my_arg1 my_arg2
```

The environmental variables used by the Zyacc debugger are listed below. They are discussed in more detail later.

`ZYDEBUG_APPLET`
: Specifies the name of a applet to be generated by the debugger.

`ZYDEBUG_CODEBASE`
: Specifies the relative path to the java-based GUI debugger.

`ZYDEBUG_HTMLBASE`
: Specifies the relative path to the HTML files describing your parser.

`ZYDEBUG_PORT`
: Suggests a socket port to be used by the debugger.

`ZYDEBUG_SRCBASE`
: Specifies the relative path to the source file for your parser.

8.3 Debugging Parsers Using a Textual Interface

Parsers can be debugged using a textual interface with a single process for both your program and the debugging interface, or by using a separate process for your program and a separate process for the debugger interface. The single process alternative is fine as long as your program does not generate any terminal I/O. If your program does generate terminal I/O, then sorting out your program's I/O from the debugger's I/O could get messy and multi-process debugging is to be preferred.

In a networked environment, the multi-process debugger also allows *remote debugging*, with the program being debugged running on one computer and the debugging interface running on another computer. For this to work, your network environment must support BSD-style sockets, — this is usually the case for most modern systems.

8.3.1 Starting a Single Process Textual Debugger

To debug your program using a single process textual interface you must first compile and link your program as outlined in Section 8.1 [Building Debugging Parsers], page 76. Then you should start your program the way your normally do specifying any command-line arguments required. You should not specify any environmental variables. Your program starts up as normal and when the parsing function `yyparse` is first entered, the debugger takes control and allows you to interact with it using the debugger commands (see Section 8.3.3 [Textual Debugger Commands], page 79). (As mentioned earlier, if your program does any terminal I/O, then the debugger interaction will be interspersed with the interaction with your program).

8.3.2 Starting a Multiple Process Textual Debugger

To debug your program using a textual interface with multiple processes you must first compile and link your program as outlined in Section 8.1 [Building Debugging Parsers], page 76. To start your program, you should define the environmental variables described below and then start your program the way your normally do specifying any command-line arguments required. To make the debugger start in multiple-process mode, the environmental variable (see Section 8.2 [Environmental Variables], page 77) `ZYDEBUG_PORT` must be defined when the program is started. The value specified for `ZYDEBUG_PORT` should be a suggested socket port number to use: Zyacc merely uses it as a starting point in its search for a free socket and it is usually best to specify it simply as 1. Once it finds a free port, it outputs its number to the terminal as follows:

```
zydebug port: 6001
```

Given the port number, you can start that portion of the debugger which communicates with your program. To do this, you should run the `zydebug` program (which should have been installed along with Zyacc) giving it the above socket number:

```
zydebug 6001
```

If you want to run `zydebug` on a computer different from the one where your program is running then simply type:

`zydebug` *HOST PORT*

where *HOST* is the network address (hostname or dotted IP address) of the computer on which your program is running, and *PORT* is the port number output when you started your program. For example, if I started the program above on host `alpha.romeo.com`, I would use:

```
zydebug alpha.romeo.com 6001
```

8.3.3 Textual Debugger Commands

Irrespective of the manner in which you start the debugger, the commands it accepts are always the same: a single letter followed by an optional argument. These commands allow you to specify breakpoints and displaypoints. When the parser encounters a *breakpoint* the debugger suspends execution of the parser and allows you to interact with it. When the parser encounters a `displaypoint`, the debugger displays information about the state of the parse. The concepts of displaypoints and breakpoints are orthogonal — i.e. it is possible to display information at a point in the parse without suspending parser execution at that point, or vice-versa.

The commands which are relevant to human users are the following:

`b [`*breakSpec*`]`
: Set or list breakpoint(s) as specified by *breakSpec*.

  At a breakpoint, the parser stops and the user can type in commands (it may or may not display its current state, depending on whether or not a displaypoint is set too). If breakSpec is omitted then list all breakpoints. breakSpec can have one of the following forms with the specified meaning:

  TERM Terminal with name *TERM* is about to be shifted.

  NON˙TERM
  Any rule with LHS nonterminal named *NON_TERM* is about to be reduced.

RULE˙NUM
: Rule number *RULE_NUM* is about to be reduced.

`%n` Reduce action on any nonterminal.

`%t` Shift action on any terminal.

`*` Both shift and reduce actions.

`B [`*breakSpec*`]`
: Clear breakpoint(s) as specified by *breakSpec*.

At a breakpoint, the parser stops and the user can type in commands (it may or may not display its current state, depending on whether or not a displaypoint is set too). If *breakSpec* is omitted then clear all breakpoints. *breakSpec* is as specified for the `b` command.

`c [`*temporaryBreakSpec*`]`
: Continue execution until a breakpoint is entered.

If *temporaryBreakSpec* is specified, then it specifies a temporary break point which is automatically cleared whenever the next breakpoint is entered. *temporaryBreakSpec* can have the same form as *breakSpec* for the `b` command.

`d [`*displaySpec*`]`
: Set or list displaypoint(s) as specified by *displaySpec*.

At a displaypoint, the parser displays its current state (it may or may not stop to let the user interact with it, depending on whether or not a breakpoint is set too). If displaySpec is omitted then list all displaypoints. *displaySpec* can have the same form as *breakSpec* for the `b` command.

`D [`*displaySpec*`]`
: Clear displaypoint(s) as specified by *displaySpec*.

At a displaypoint, the parser displays its current state (it may or may not stop to let the user interact with it, depending on whether or not a breakpoint is set too). If displaySpec is omitted then clear all displaypoints. *displaySpec* can have the same form as *breakSpec* for the `b` command.

`h [`*cmd*`]` Print help on single-letter command *cmd*. If *cmd* is omitted, give help on all commands.

`l [`*listSpec*`]`
: List terminals, non-terminals or rules as specified by *listSpec*.

If *listSpec* is omitted, then all rules are listed. Otherwise *listSpec* can be one of the following:

`%n` List all non-terminal symbols.

`%t` List all terminal symbols.

RULE˙NUM
: List rule with number *RULE_NUM*.

`m [`*depth*`]` Set maximum *depth* printed for the stack.

If depth is 0 or omitted, then entire stack is printed.

`n`
: Execute parser till next shift action.

  Equivalent to `c %t`.

`p`
: Print current parser state.

`q`
: Quit debugger and run parser without debugging.

`s`
`<blank line>`
: Single-step parser to next shift or reduce action.

  Equivalent to `c *`.

8.4 Debugging Parsers Using a Graphical User Interface

It is also possible to use a java-based GUI to debug parsers generated by Zyacc. This can be done in two ways:

1. The GUI can be used as a standalone java application. To do this, you should have a java runtime system installed on your computer.
2. The GUI can be used as an applet from within a web browser which supports java.

8.4.1 Starting a GUI Debugger as a Java Application

To debug your program using a GUI java application you must first compile and link your program as outlined in Section 8.1 [Building Debugging Parsers], page 76. To use the GUI as a standalone application, you should first start your program the way your normally do, specifying any command-line arguments required. The following environmental variables (see Section 8.2 [Environmental Variables], page 77) are used by your program to control the setup of the debugger.

`ZYDEBUG_PORT`
: This is required. As discussed earlier (see Section 8.3.2 [Starting a Multiple Process Textual Debugger], page 79), it's value is best specified simply as `1`.

`ZYDEBUG_SRCBASE`
: This should be the relative path to the parser source file (the '`.y`' file) from the directory which is current when the parsing function `yyparse` is first entered. (The directory in which `yyparse` is entered is usually the same directory in which your program executable resides, assuming that your program has not performed any `chdir()` calls and that you started the program in the directory in which it resides). If it not specified, it is assumed that your parser source file lives in the directory the parsing function was in when it was first entered.

When your program starts it will output the port number of the socket it has connected to; for example:

```
    zydebug port: 6001
```

This port number will be used to connect the GUI to your executing program.

To start the GUI, you must run the java runtime system on your machine, telling it where to find the java classfiles referenced by `zdu.zydebug.ZYDebug` which form the GUI and providing it with arguments telling it how to connect to your executing program. The exact procedures may be different on your machine, but as of this writing, the most common setup is as follows:

Java is started by simply using the command `java`. For the `java` command to work it must be in your `PATH` (see Section 8.2 [Environmental Variables], page 77) or you should specify the full path name. For java to find the java classfiles for the debugger, those files must be in your `CLASSPATH` (see Section 8.2 [Environmental Variables], page 77). Finally, you must specify the port number output by your program, (optionally preceeded by the hostname or dotted IP address of the computer on which your program is running, if it is different from the one on which you are running the GUI).

The following example shows the starting a debugger GUI under a `csh` or derivatives.

```
% setenv CLASSPATH /usr/local/share/classes
% java zdu.zydebug.ZYDebug 6001 &
```

Under `sh` or derivatives, the following command can be used to connect to a program running on a different machine:

```
$ CLASSPATH=/usr/local/share/classes java zdu.zydebug.ZYDebug pear 6001 &
```

where `pear` is the name of the machine on which the program is running.

8.4.2 Starting a GUI Debugger as a Java Applet

A java applet can only be run by being embedded within an HTML file which provides the environment in which the applet lives. Among other things, the HTML file provides the arguments to the applet.

The debugging applet talks to the program being debugged using a socket referred to by a port number. Since this port number can vary for different executions of the program being debugged, it has to be provided as an argument to the applet. As an applet gets its argument from its associated HTML file, and this particular port number argument can be different for different executions of the program being debugged, the HTML file associated with the debugging applet has to be generated dynamically.

The HTML file is generated by your enhanced program when it is started (actually a second HTML file is generated as well). Using environmental variables (see Section 8.2 [Environmental Variables], page 77) you can specify the names of the HTML files, as well as paths to various other resources required by the applet.

To debug your program using a GUI java applet you must first compile and link your program as outlined in Section 8.1 [Building Debugging Parsers], page 76. You should then start your program the way your normally do, specifying any command-line arguments required. The following environmental variables (see Section 8.2 [Environmental Variables], page 77) are used by your program to control the setup of the debugger.

`ZYDEBUG_APPLET`
: This is required. It specifies the root name used for the generated HTML documents which are used to access the debugging applet.

`ZYDEBUG_CODEBASE`
: This should provide the path to the debugger's java classfiles relative to the directory where the generated HTML documents live (relative to the `DOCBASE` directory in java

terminology). If not specified, then it is assumed that the java classfiles reside in the same directory as the generated HTML documents.

`ZYDEBUG_HTMLBASE`
: This should provide the path to the parser's HTML description files generated using the `--HTML` option see Chapter 9 [Invoking Zyacc], page 86), relative to the directory where the generated HTML documents live (relative to the `DOCBASE` directory in java terminology). If not specified, then it is assumed that the HTML description files reside in the same directory as the generated HTML documents.

`ZYDEBUG_PORT`
: This is not required. As discussed earlier See Section 8.3.2 [Starting a Multiple Process Textual Debugger], page 79, it's value is best specified simply as `1`.

`ZYDEBUG_SRCBASE`
: This is exactly as for the standalone GUI application. It should be the relative path to the parser source file (the '`.y`' file) from the directory which is current when the parsing function `yyparse` is first entered. (The directory in which `yyparse` is entered is usually the same directory in which your program executable resides, assuming that your program has not performed any `chdir()` calls and that you started the program in the directory in which it resides). If it not specified, it is assumed that your parser source file lives in the directory the parsing function was in when it was first entered.

Usually, the parser source file and the HTML description files are in the same directory as the program executable and all you need to specify are `ZYDEBUG_APPLET` and `ZYDEBUG_CODEBASE`.

Consider the following more complicated situation:

- You would like to put the generated HTML files in `$HOME/tmp` with the name `XXX`.
- The java classfiles reside in `/usr/local/share/classes`.
- The HTML files describing the parser and the source file reside in the parent directory of the directory in which you start the program.
- Your enhanced executable program is called `foo` and takes a single argument `bar`.

Under `csh` and derivatives, you can use the following sequence of commands:

```
setenv ZYDEBUG_APPLET $HOME/tmp/XXX
setenv CODEBASE /usr/local/share/classes
setenv HTMLBASE `pwd`/..
setenv SRCBASE `pwd`/..
./foo bar
```

Under `sh` and derivatives, the following command suffices:

```
ZYDEBUG_APPLET=$HOME/tmp/XXX CODEBASE=/usr/local/share/classes \
  HTMLBASE=`pwd`/.. SRCBASE=`pwd`/.. \
  ./foo bar
```

Having generated the HTML files, you can now use a browser to start the debugger's GUI and debug your parser. There are two methods of doing this available in most browsers:

- View the generated HTML file using the local file access option provided by most browsers. This will work only if you are using the browser on the same machine as the one on which your program is executing.

- View the generated HTML file as a regular remote URL. For this to work, all the files required by the applet (the java classfiles, the parser source file, and the HTML description files) must be accessible by the HTTP daemon on the remote machine. This usually requires that they be within a public html area.

8.4.3 Using the Debugger GUI

As the applet is talking to your compiled program which has not been modified in any way, it is not possible to have the applet restart your program once it has completed its parse. Instead you will have to restart the parser and the debugger's GUI applet.

The applet has four main windows. Clockwise from the top-left they are the following:

Parse Forest Window

Shows the current parse-forest. The nodes on top are the nodes currently on the stack. Terminal nodes are in red, non-terminal nodes are in green and error nodes are in pink. The last active node is highlighted in yellow. Each node contains text of the form *S/Sym*, where *S* is the state at which that node was created and *Sym* is the grammar symbol or token semantics corresponding to the node. *The nodes in the top row correspond to nodes currently on the parse stack.*

Each non-leaf node in the forest is clickable. Clicking on such a node hides all its subtrees; clicking again on that node displays the subtrees again. This can be useful as the parse tree typically gets pretty large for practical parsers.

Trace Window

Shows the parse stack in gray (each entry is in the same format as a parse tree node), the current lookahead in red and the following action in blue.

Breakpoint Window

This allows you to set/clear breakpoints on all or selected nonterminals and terminals. Clicking a line in the window sets a breakpoint on the symbol displayed on that line; clicking it again clears the breakpoint. The currently selected breakpoints are highlighted.

Source Window

Shows the parser source file. During a reduction, the line corresponding to the reduction is highlighted.

The debugger is controlled by the following controls:

Shadows Checkbox

This checkbox controls whether or not the parser shows crude shadows while displaying the parse forest. It is useful to avoid having shadows cluttering up the display of large forests.

Update Checkbox

Selecting this checkbox results in the parser displaying the current state in the LR(0) machine in a browser frame. For this to work, the parser should have been generated using the `--HTML` option (see Chapter 9 [Invocation], page 86), and the environmental variable `ZYDEBUG_HTMLBASE` should have been specified when the parser was started.

Step Button
: Steps the parser by a single step.

Next Button
: Steps the parser to the next shift action.

Continue Button
: Steps the parser till the next breakpoint. If no breakpoints are set, then the parser runs to completion. As mentioned above, it is not possible to restart the parser.

8.5 Tradeoffs between Debugging Approaches

The popular adage "a picture is worth a thousand words" may be true in the real world, but with the primitive visualization techniques used by the GUI debugger, it may not hold for the GUI debugger. For practical parsers, the parse forest displayed by the debugger rapidly becomes unmanageable, even after hiding many large subtrees. If the large amount of screen real estate taken up by the parse forest was occupied instead by words more information might be conveyed.

It is probably best to use the GUI debugger only under the following conditions:

- Only a limited amount of text is being parsed — this prevents inordinate growth of the displayed parse forest.
- To explain and understand the operation of parsers in general and LALR(1) parsers in particular.
- You are a novice and don't feel like climbing the learning curve hillock represented by the textual commands.

Another problem with GUI debugging is that it can be quite slow as the java GUI has to build up the parse forest, trace output, etc. I have not had a chance to analyze its performance, but I would not be surprised if it is spending a fair amount of time merely collecting garbage.

For debugging most practical parsers, I would recommend the textual interface.

9 Invoking Zyacc

The command line needed to invoke Zyacc has the format:

`zyacc [`*Options List*`]` *yacc-file* `[`*yacc-file*`...]`

Here *yacc-file* specifies the name(s) of the grammar file, usually ending in '`.y`'. Unless the `--output` or `--yacc` options are specified, the name of the generated parser file is created by replacing the '`.y`' with '`.tab.c`'. Thus, the '`zyacc foo.y`' filename yields '`foo.tab.c`', and the '`zyacc hack/foo.y`' filename yields '`hack/foo.tab.c`'.

- If no *yacc-file*s are specified on the command line, then a help message is printed on the standard output.
- If multiple *yacc-files*s are specified, then their concatenation is treated as a single logical file.
- A *yacc-file* specified by the single character '`-`' stands for the standard input.

9.1 Option Conventions

A word which constitutes a command-line argument has two possible types: it is a *option word* if it begin with a '`-`' or '`--`' (with certain exceptions noted below), or if it follows an option word which requires an argument. Otherwise it is a *non-option word*. An option word specifies the value of a Zyacc option; a non-option word specifies a file name.

- Options with short single character names must begin with a single '`-`'.
- Options with long multiple-character names must begin with '`--`'. The name can consists of any alphanumeric characters along with '`-`' and '`_`' characters.
- When a option is specified using a long name, it is sufficient to specify an unambiguous prefix of its name.
- If an option has a value which must be one of several prespecified values, then it is sufficient to specify an unambiguous prefix of the value, in a manner similar to long option names.
- It is possible to specify an option value in the same word as the option name. For short option names, there should not be any intervening characters between the short name and the value. For long option names, the long name should be separated from the value using a single '`=`' character.
- If an option has an *optional* value, then the value must be provided in the same word as the option name, as outlined above.
- If an option has a *required* value, then the value may be provided in the same word as the option name as outlined above, or it may be provided in the next word. In the latter case, the entire next word is taken to be the value (even if it looks like an option starting with '`-`' or '`--`').
- Short names for multiple options which are not allowed to have any values may be combined into a single word. For example, if Zyacc had options '`-l`' and '`-7`' which were not allowed to have any values (it does not), then instead of specifying them using two words as '`-l -7`', they can be specified using a single word '`-l7`'.
- If an option is given two incompatible values, then the option which is specified later dominates.
- If the option consisting simply of the two characters '`--`' is specified, then all the remaining words on the command line will not be treated as options irrespective of whether they start with '`-`' or '`--`'. This makes it possible to specify file names starting with a '`-`'.

- Option words and non-option words may be arbitrarily interspersed.
- Command-line options always override the options specified elsewhere (see Section 9.2 [Option Sources], page 87).

9.2 Option Sources

Besides the command-line, Zyacc can read its options from several different sources. In order of increasing priority these sources are the following:

- The file '`zyacc.opt`'. The file should contain only option names and values separated by whitespace (newline counts as whitespace). In addition it may contain comments enclosed within '`/*`' and '`*/`'. The file is searched for using Zyacc's search list (see Section 9.4 [Data Search List], page 89).

  The use of this file for setting defaults, makes it possible for a site to setup default options for Zyacc different from its builtin defaults. It also makes it easy to drop Zyacc into a GUI toolset where options are set using a graphical user interface.
- The environment variable `ZYACC_OPTIONS`. If this variable is set, then its value should contain only options and option values separated by whitespace as on the command-line. The procedure for setting environment variables depends on the system you are using: under the UNIX shell `csh` the `setenv` command can be used, under the MS-DOS command-interpreter the `set` command can be used; under the UNIX shell `sh` or `ksh` the `export` command can be used.
- It is also possible to specify options directly within the Zyacc source file using the `%option` directives (see Section 3.1.2 [Zyacc Declarations], page 31).

Options specified by the environment variable `ZYACC_OPTIONS` overrides the options specified in the '`zyacc.opt`' file. Options specified in the Zyacc source file override options specified in the '`zyacc.opt`' file or `ZLEX_OPTIONS` environment variable. Finally, command-line options always override options specified by all other sources.

9.3 Zyacc Options

A list of the available options follows:

`--build-display`
: Display the parameters used to build zyacc and exit.

`--debug[=1|0]`
`-t[1|0]`
: Output a definition of the macro `YYDEBUG` into the parser file, so that the debugging facilities are compiled. See Chapter 8 [Debugging Your Parser], page 76.

`--defines[=1|0]`
`-d[1|0]`
: Output a .h definitions file containing the semantic value type `YYSTYPE` and C preprocessor macro definitions for all the token type names defined in the grammar (default: `0`).

  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 4.2.2 [Semantic Values of Tokens], page 55.

`--file-prefix` *prefix*
`-b` *prefix*
: Specify a *prefix* to use for all Zyacc output file names. The names are chosen as if the input file were named '*prefix*`.c`'.

`--grammar[=1|0]`
`-g[1|0]`
: Output a reference grammar file to <stdout> and exit (default: `0`).

`--help`
`-h`
: Print summary of options and exit.

`--HTML[=1|0]`
`-H[1|0]`
: Write a HTML parser description in .html file. This is similar to the `--verbose` option, but you can use any WWW browser like Netscape or Lynx to browse the parser description file and follow hot links within the file (default: `0`).

`--lines`
: Output #line directives in generated parser file (default: `1`). If Zyacc puts these directives in the parser file then the C compiler and debuggers can associate errors with your source file, the grammar file. If this option is specified as `0`, then errors will be associated with the parser file, treating it an independent source file in its own right.

`--longer-rule-prefer[=1|0]`
: When resolving a reduce-reduce conflict prefer the longer rule (default: `0`). If this option is specified as `0`, then the reduce-reduce conflicts are resolved in favor of the rule which occurs *earlier* in the source file.

`--output-file` *outFile*
`-o` *outFile*
: Specify the name outFile for the parser file. The other output files' names are constructed from *outfile* as described under the `--defines` and `--verbose` switches.

`--name-prefix` *prefix*
`-p` *prefix*
: Specify *prefix* to be used for all external symbols. Rename the external symbols used in the parser so that they start with *prefix* instead of '`yy`'. The precise list of symbols renamed is `yychar`, `yydebug`, `yyerror`, `yylex`, `yynerrs`, `yylval` and `yyparse`.

  For example, if you use '`-p c`', the names become `cchar`, `cdebug`, and so on.

  See Section 3.7 [Multiple Parsers in the Same Program], page 53.

`--term-prefix` *suffix*
: Specify suffix string to be appended to all external terminal names (default `""`). For example, if you specify `--term-prefix _TOK`, then the string `_TOK` will be appended to all terminal names exported from the grammar; if you use `ID` as a terminal name within your grammar, its external name will be `ID_TOK`.

`--verbose[=1|0]`
`-v[1|0]`
: Write verbose parser description in .output file (default: `0`). The extra output file contains 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`'.

`--version`
`-V` Print version number and exit.

`--yacc[=1|0]`
`-y[1|0]` Name output files like YACC (default: `0`) 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. Thus, the following shell script can substitute for Yacc:

```
zyacc -y $*
```

9.4 Data Search List

When Zyacc is run, it looks for certain data files (a skeleton file '`zyaccskl.c`' and an options file '`zyacc.opt`' (see Section 9.2 [Option Sources], page 87)) in certain standard directories (the skeleton file *must* exist, but the option file need not exist). The search list specifying these standard directories is fixed when Zyacc is installed; it can be printed out using Zyacc's '`--help`' option (see Section 9.3 [Zyacc Options], page 87).

The search list consists of a list of colon-separated directory names (the directory names may or may not have terminating slashes) or environment variables (starting with a '`$`'). If a directory name starts with a '`$`', then the first (only the first) '`$`' must be repeated. An empty component in the search list specifies the current directory. Typically the search list contains the current directory. Also typically, the environment variable `ZYACC_SEARCH_PATH` is present in the search list — this causes Zyacc to check if the variable is set in the environment. If it is, then Zyacc expects it to specify a search list which it recursively searches.

Typically, the search list compiled into Zyacc looks something like the following:

```
$ZYACC_SEARCH_PATH:.:$HOME:/usr/local/share/zyacc
```

Since the search list will typically contain an environment variable like `ZYACC_SEARCH_PATH` it is possible to change the set of standard directories searched by Zyacc even after installation by specifying a value for the variable. For example, if with the above search list, `ZYACC_SEARCH_PATH` is set to `/usr/lib:/usr/opt/lib`, then the effective search list becomes:

```
/usr/lib:/usr/opt/lib:.:$HOME:/usr/local/share/zyacc
```

10 Known Incompatibilities with Bison

The following lists Bison features which are not available or work differently in Zyacc:

- In Bison multi-character literals are enclosed within double-quotes, whereas Zyacc uses single-quotes for both single and multi-character literals. The Bison approach is more consistent with C; the Zyacc approach avoids making another character special (the Zyacc approach predates the public release of this feature in Bison).
- Zyacc does not have any public interface to retrieve a token number given its name, similar to Bison's `yytname` table.
- Zyacc does not support Bison's `YYBACKUP()` macro.
- Bison allows the use of `%type` declarations to declare the types of terminal symbols, even though the manual seems to imply that `%type` can be used only for declaring the types of nonterminals. This feature appears useful.
- Zyacc's debugging facilities are quite different from Bison's trace facility.

11 Bugs and Deficiencies

Internal versions of the Zyacc parser generator have been used by me since 1995. It has been used by about 45 students in compiler construction courses. There have been some major rewrites.

11.1 Suspicions

Normally, when a program has been completed, one has a reasonable idea where bugs might still lurk. On that basis, for what they are worth, I present my current suspicions:

The Parser Generator

I feel fairly good about the code which actually generates parsers. Under normal error-free operation, the weakest areas may be the code which does the LALR computation and the graph coloring for packing inherited attributes (since most states of most parsers can get by with SLR lookahead and most programs I've written didn't make very heavy use of inherited attributes, the code has just not been exercised enough).

The bugs most likely to manifest themselves will probably be `assert`ion failures **caused by an erroneous Zyacc source file**. In that case, you can simply correct the error in the Zyacc source file and continue on with reasonable confidence (after submitting a bug report, of course).

The Runtime System

The runtime system has been fairly stable for a while. The weakest area may be error processing.

11.2 Bug Reports

First you will need to be sure that you have found a Zyacc bug:

- If when running Zyacc, it bombs with a assertion failure or a core dump, you can be sure that you have uncovered a bug. However, even though you have uncovered a Zyacc bug, the problem may have been caused by an error in your source file; you can fix the error and continue working.
- If when running a Zyacc generated parser, it bombs or does not do what you intended it to do, it is much more difficult to be sure whether the problem is within the Zyacc runtime system or in your code. If it bombs within the main parser function, make sure that it is not within your actions.

If you are sure that you have uncovered a bug, try to distil it down to a test program which is as short as possible while still exhibiting the bug. Record a log which exhibits the bug. Make sure that you mention the version of Zyacc you are using in your bug report.

Bug reports can be mailed to:

```
zdu@acm.org
```

Appendix A Zyacc Symbols

`error`
: A token name reserved for error recovery. This token may be used in grammar rules so as to allow the Zyacc 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 71.

`YYABORT`
: Macro to 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 [The Parser Function `yyparse`], page 54.

`YYACCEPT`
: Macro to pretend that a complete utterance of the language has been read, by making `yyparse` return 0 immediately. See Section 4.1 [The Parser Function `yyparse`], page 54.

`YYBACKUP`
: Macro to discard a value from the parser stack and fake a look-ahead token. See Section 4.4 [Special Features for Use in Actions], page 59.

`YYERROR`
: Macro to 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 71), or (if recovery is impossible) make `yyparse` return 1. See Chapter 6 [Error Recovery], page 71.

`YYERROR_VERBOSE`
: Macro that you define with `#define` in the Zyacc declarations section to request verbose, specific error message strings when `yyerror` is called.

`YYINITDEPTH`
: Macro for specifying the initial size of the parser stack. See Section 5.8 [Stack Overflow], page 70.

`YYLEX_PARAM`
: Macro for specifying an extra argument (or list of extra arguments) for `yyparse` to pass to `yylex`. See Section 4.2.4 [Calling Conventions for Pure Parsers], page 57.

`YYLTYPE`
: Macro for the data type of `yylloc`; a structure with four members. See Section 4.2.3 [Textual Positions of Tokens], page 55.

`YYMAXDEPTH`
: Macro for specifying the maximum size of the parser stack. See Section 5.8 [Stack Overflow], page 70.

`YYPARSE_PARAM`
: Macro for specifying the name of a parameter that `yyparse` should accept. See Section 4.2.4 [Calling Conventions for Pure Parsers], page 57.

`YYRECOVERING`
: Macro whose value indicates whether the parser is recovering from a syntax error. See Section 4.4 [Special Features for Use in Actions], page 59.

`YYSTYPE`
: Macro for the data type of semantic values; `int` by default. See Section 3.5.1 [Data Types of Semantic Values], page 35.

`yychar` External integer variable that contains the integer value of the current look-ahead token. (In a pure parser, it is a local variable within `yyparse`.) Error-recovery rule actions may examine this variable. See Section 4.4 [Special Features for Use in Actions], page 59.

`yyclearin`
Macro used in error-recovery rule actions. It clears the previous look-ahead token. See Chapter 6 [Error Recovery], page 71.

`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 8 [Debugging Your Parser], page 76.

`yyerrok` Macro to cause parser to recover immediately to its normal mode after a parse error. See Chapter 6 [Error Recovery], page 71.

`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 [The Error Reporting Function `yyerror`], page 59.

`yylex` User-supplied lexical analyzer function, called with no arguments to get the next token. See Section 4.2 [The Lexical Analyzer Function `yylex`], page 54.

`yylval` External variable in which `yylex` should place the semantic value associated with a token. (In a pure parser, it is a local variable within `yyparse`, and its address is passed to `yylex`.) See Section 4.2.2 [Semantic Values of Tokens], page 55.

`yylloc` External variable in which `yylex` should place the line and column numbers associated with a token. (In a pure parser, it is a local variable within `yyparse`, and its address is passed to `yylex`.) You can ignore this variable if you don't use the '`@`' feature in the grammar actions. See Section 4.2.3 [Textual Positions of Tokens], page 55.

`yynerrs` Global variable which Zyacc increments each time there is a parse error. (In a pure parser, it is a local variable within `yyparse`.) See Section 4.3 [The Error Reporting Function `yyerror`], page 59.

`yyparse` The parser function produced by Zyacc; call this function to start parsing. See Section 4.1 [The Parser Function `yyparse`], page 54.

`%left` Zyacc declaration to assign left associativity to token(s). See Section 3.6.2 [Operator Precedence], page 47.

`%nonassoc`
Zyacc declaration to assign nonassociativity to token(s). See Section 3.6.2 [Operator Precedence], page 47.

`%prec` Zyacc declaration to assign a precedence to a specific rule. See Section 5.4 [Context-Dependent Precedence], page 66.

`%pure_parser`
Zyacc declaration to request a pure (reentrant) parser. See Section 3.6.9 [A Pure (Reentrant) Parser], page 51.

`%right` Zyacc declaration to assign right associativity to token(s). See Section 3.6.2 [Operator Precedence], page 47.

`%start`	Zyacc declaration to specify the start symbol. See Section 3.6.7 [The Start-Symbol], page 49.
`%token`	Zyacc declaration to declare token(s) without specifying precedence. See Section 3.6.1 [Token Type Names], page 47.
`%type`	Zyacc declaration to declare nonterminals. See Section 3.6.5 [Nonterminal Symbols], page 49.
`%union`	Zyacc declaration to specify several possible data types for semantic values. See Section 3.6.4 [The Collection of Value Types], page 48.

These are the punctuation and delimiters used in Zyacc input:

'`%%`'	Delimiter used to separate the grammar rule section from the Zyacc declarations section or the additional C code section. See Section 1.7 [The Overall Layout of a Zyacc Grammar], page 7.
'`%{ %}`'	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 [Outline of a Zyacc Grammar], page 31.
'`/*...*/`'	Comment delimiters, as in C.
'`:`'	Separates a rule's result from its components. See Section 3.3 [Syntax of Grammar Rules], page 33.
'`;`'	Terminates a rule. See Section 3.3 [Syntax of Grammar Rules], page 33.
'`|`'	Separates alternate rules for the same result nonterminal. See Section 3.3 [Syntax of Grammar 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 [Languages and Context-Free Grammars], page 3.
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 [Languages and Context-Free Grammars], page 3.
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 [The Zyacc Parser Algorithm], page 62.
Grouping	A language construct that is (in general) grammatically divisible; for example, 'expression' or 'declaration' in C. See Section 1.1 [Languages and Context-Free Grammars], page 3.
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 [Languages and Context-Free Grammars], page 3.
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 [Operator Precedence], page 64.
Left recursion	A rule whose result symbol is also its first component symbol; for example, '`expseq1 : expseq1 ',' exp;`'. See Section 3.4 [Recursive Rules], 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 [The Zyacc Parser Algorithm], page 62.

Lexical analyzer (scanner)
: A function that reads an input stream and returns tokens one by one. See Section 4.2 [The Lexical Analyzer Function `yylex`], page 54.

Lexical tie-in
: A flag, set by actions in the grammar rules, which alters the way tokens are parsed. See Section 7.2 [Lexical Tie-ins], page 74.

Look-ahead token
: A token already read but not yet shifted. See Section 5.1 [Look-Ahead Tokens], page 62.

LALR(1)
: The class of context-free grammars that Zyacc (like most other parser generators) can handle; a subset of LR(1). See Section 5.7 [Mysterious Reduce/Reduce Conflicts], page 69.

LR(1)
: The class of context-free grammars in which at most one token of look-ahead is needed to disambiguate the parsing of any piece of input.

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 71.

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 [The Zyacc Parser Algorithm], page 62.

Reentrant
: A reentrant subprogram is a subprogram which can be in invoked any number of times in parallel, without interference between the various invocations. See Section 3.6.9 [A Pure (Reentrant) Parser], page 51.

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 [Recursive Rules], 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 [Defining Language Semantics], page 35.

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 [The Zyacc Parser Algorithm], page 62.

Single-character literal
: A single character that is recognized and interpreted as is. See Section 1.2 [From Formal Rules to Zyacc Input], page 4.

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 3.6.7 [The Start-Symbol], page 49.

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 17.

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 Zyacc 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 [Languages and Context-Free Grammars], page 3.

Appendix C Conditions

The conditions for using Zyacc are identical to the conditions for using Bison shown below.

C.1 Conditions for Using Bison

As of Bison version 1.24, we have changed the distribution terms for `yyparse` to permit using Bison's output in non-free programs. Formerly, Bison parsers could be used only in programs that were free software.

The other GNU programming tools, such as the GNU C compiler, have never had such a requirement. They could always be used for non-free software. The reason Bison was different was not due to a special policy decision; it resulted from applying the usual General Public License to all of the Bison source code.

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.) When we applied the GPL terms to the code for `yyparse`, the effect was to restrict the use of Bison output to free software.

We didn't change the terms because of sympathy for people who want to make software proprietary. **Software should be free.** But we concluded that limiting Bison's use to free software was doing little to encourage people to make other software free. So we decided to make the practical conditions for using Bison match the practical conditions for using the other GNU tools.

C.2 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, 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 show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.
1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.
2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:
   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

Table of Contents