Thinkage Ltd.
YAY Reference Manual

Last Revision: November 11, 1995

Table of Contents

1. Introduction

YAY is a tool for writing compilers and other programs that parse input according to strict "grammar" rules. In the terminology of parsing theory, YAY can handle LALR(2) grammars with disambiguating rules. This is a fairly broad class of grammars -- most of the computing languages in use today can be described using LALR(2) specifications.

For a precise definition of this sort of grammar, see any standard text on parsing theory, e.g. *Principles of Compiler Design* by A.V.Aho and J.D.Ullman, Addison-Wesley, 1977, or *LR Parsing* by Aho and Johnson, in Computing Surveys.

The appearance of YAY input is based on the input to YACC (Yet Another Compiler-Compiler), written by S.C.Johnson of Bell Telephone Laboratories, Murray Hill, N.J. The LALR(1) version of YAY was written by K.W.Lalonde of the Software Development Group of the University of Waterloo. Enhancements to allow YAY to handle LALR(2) grammars were made by P.J.Fraser, also of SDG.

The parsing algorithm used by YAY is derived from the article "Methods for Computing LALR(k) Lookahead" by B.B.Kristensen and O.L.Madsen, *ACM Transactions on Programming Languages and Systems*, Vol.3, No.1, January 1981, pp.60-82. Those interested in reading this article should first have a good grasp of parsing theory principles.

1.1 A Note to Novices

YAY can produce anything from a simple parser for a desk calculator program to a very complicated parser for a programming language. Those who are using YAY for complex tasks have to know all the idiosyncrasies of the program, including a good deal about the internal workings of YAY. On the other hand, the internal workings are mostly irrelevant to someone who is making an easy straightforward parser.

For this reason, novices may want to skip some the sections of the manual which describe very technical aspects of YAY. This is particularly true on first reading, when it is more important to get an overview of how YAY is used than to try to comprehend every little detail. We therefore suggest that novices concentrate on the following sections, and only look at other sections if the requirements of your parser make it necessary.

- All of Chapter 2
- All of Chapter 3
- All of Chapter 4

- -- Sections 6.0, 6.1, 6.2
- -- all of Chapter 7
- -- Sections 9.3, 9.5, 9.6

Novices may be particularly interested in Appendix A which gives an example of YAY input for a simple parser.

2. How YAY Works

The input to YAY describes the rules of a grammar. YAY uses these rules to produce the source code for a program that will parse the grammar. This source code can then be compiled to obtain a program that reads input, parses it according to the grammar, and takes action based on the result.

The source code produced by YAY consists of a number of data tables that represent the grammar, plus a C function named `"yyparse"`. By default, all the symbol names used by YAY begin with `"yy"`. This is an historical convention, dating back to YAY's predecessor YACC. Users can avoid conflicts with YAY names by avoiding symbols that start with `"yy"`.

Users who want to use a different prefix may indicate this with a line of the form

```
%prefix PREFIX
```

at the beginning of their YAY input. For example,

```
%prefix ww
```

asks for a prefix of `"ww"` instead of `"yy"`. The prefix chosen should be one or two characters long -- longer prefixes will lead to name conflicts on machines where external names are truncated to six characters during the loading process. In addition, at least one of the characters in the prefix should be a lower case letter (because YAY uses an all upper case version of the prefix for some special names, and this has to be different from the real prefix).

Different prefixes are useful when two YAY-produced parsers are to be merged into a single program. For the sake of convenience, however, the `"yy"` convention will be used in this manual.

2.1 "yyparse" and "yylex"

`"yyparse"` returns a value of 0 if the input it parses is valid according to the given grammar rules. It returns a 1 if the input is invalid and error recovery is impossible. `"yyparse"` does not do its own lexical analysis. In other words, it does not pull the input apart into tokens ready for parsing. Instead, it calls a routine called `"yylex"` every time it wishes to obtain a token from the input.

`"yylex"` returns a value indicating the *type* of token that has been obtained. If the token has an actual *value*, this value (or some representation of the value, e.g. a pointer to a string containing the value) is returned in an external variable named `"yylval"`.

It is up to the user to write a `"yylex"` routine that breaks the input into tokens and returns the tokens one by one to `"yyparse"`. We will say more about the lexical analyzer in Section 3.3.

2.2 Grammar Rules

The grammar rules given to YAY not only describe what inputs are valid according to the grammar but also specify what action should be taken when a given input is encountered. For example, if the parser recognizes a statement that assigns a value to a variable, the parser should either perform the assignment itself or take some action to ensure that the assignment will eventually take place.

As an example, if the parser is part of an interactive desk calculator, it could carry out arithmetic calculations as soon as the instructions are recognized. However, if the parser is acting as the first pass in a multi-pass compiler, it may simply encode the input in a way that will be used in a later code-generation pass.

In summary, the user must provide a number of things when using YAY to produce a parser:

(1) grammar rules indicating what input is and isn't valid;

(2) a lexical analyzer (`"yylex"`) that breaks raw input into tokens for the parsing routine `"yyparse"`;

(3) any source code or functions that may be needed to perform appropriate actions once particular inputs are recognized;

(4) a mainline routine that performs any necessary initializations, calls `"yyparse"`, then performs possible clean-up actions. The simplest kind of mainline is just a function `"main"` that calls `"yyparse"`, then returns.

2.3 Notes on Compiling Source Code Produced by YAY

By default, C source code produced by YAY contains the line

```
#define YYSCTAB const
```

It then uses the `YYSCTAB` manifest to declare various data objects as **`const`**. If you will be compiling the YAY output with an old C compiler that doesn't recognize the **`const`** qualifier, you should change the definition to read

```
    #define YYSCTAB
```

3 Input to YAY

This chapter describes the input to YAY when you are defining an LALR(1) grammar. Chapter 9 will talk about additional needs for LALR(2) grammars.

The input to YAY is broken into three sections:

-- declarations
-- grammar rules
-- programs

The contents of each section will be described shortly, but first we will give some overall rules for YAY input.

Sections of YAY input are separated by the symbol `%%`. The general lay-out of YAY input is therefore

```
declarations
%%
grammar rules
%%
programs
```

The declarations section may be omitted if no declarations are necessary. This will mean that the input starts with the first `%%`. The programs section may also be omitted, from the second `%%` on. The simplest input for YAY is therefore

```
%%
grammar rules
```

Blanks, tabs, and new-lines are used to separate items in YAY input. These are called *white-space* characters. Any place one white-space character is valid, any number of blanks, tabs, and/or new-lines may be used. This means, for example, that the `%%` to separate sections does not have to be on a line all by itself. However, giving it a line of its own makes the YAY input easier to read.

Comments may appear anywhere a blank is valid. As in C, comments begin with `/*` and end with `*/`.

Identifiers used in YAY input may be of arbitrary length, and may consist of all letters (upper and lower case), all digits, and the characters dot (`.`) and underscore (_). The first character of an identifier may not be a digit. YAY distinguishes between upper and lower case letters, so `"this"` and `"THIS"` and `"This"` are all different symbols.

Literals in YAY input consist of a single ASCII character enclosed in single quotes, e.g. `'c'`. The standard C escape sequences are recognized:

```
\b   -- backspace
\n   -- new-line
\r   -- carriage return
\t   -- tab
\'   -- single quote
\\   -- backslash
\xxx -- any ASCII character
        ("xxx" is octal representation)
```

For technical reasons, the null character `'\000'` should never appear in YAY input.

3.1 The Declarations Section

The declarations section describes many of the identifiers that will be used in the rest of the YAY input. There are two types of declarations:

(a) Token declarations;

(b) Declarations of functions and variables used in the actions that the parser takes when a particular input is recognized.

The declarations section can also specify rules for the precedence and binding of operators used in the grammar. For example, the standard order of arithmetic operations would normally be set up in the declarations section.

3.1.1 Token Declarations

All ASCII characters are automatically recognized as tokens. For example, `'a'` stands for a token that is the literal character `"a"`.

Other tokens are declared with statements of the form

```
%token name1 name2 name3 ...
```

This tells YAY that the given names will refer to tokens. For example,

```
%token INTNUM
```

indicates that the identifier `INTNUM` will refer to a particular type of token returned by the lexical analyzer "`yylex`". If `INTNUM` stands for any integer number token, you might have the following code in "`yylex`".

```
c = getchar();
if ( (c >= '0') && (c <= '9') ) {
    yylval = 0;
    do {
        yylval = (yylval * 10) + (c - '0');
        c = getchar();
    } while (c >= '0' && c <= '9');
    ungetc(c, stdin);
    return( INTNUM );
}
```

"`yylex`" returns `INTNUM` to indicate that a certain kind of token (an integer number) has been returned. The actual value of this number is returned in "`yylval`". The grammar rules in the YAY input would dictate where an `INTNUM` token is valid.

In the C source code produced by YAY, the identifiers named in a `%token` statement will appear as constants set up with **`#define`**. The first named token has a #defined value of `257`, the next is #defined as `258`, and so on. Token values start at `257` so they will not conflict with any of the `256` ASCII characters.

Because token identifiers are set up as manifest constants, they must not conflict with reserved words or other identifiers that will be used by the parser. For example,

```
%token if yyparse ...
```

will almost certainly lead to errors when you try to compile the source code output of YAY. By convention, this manual will always create token names in upper case, and we recommend that you follow the same practice.

3.1.2 Precedence and Binding

Parsers that evaluate expressions usually have to establish the order in which various operations are carried out. For example, parsers for arithmetic expressions usually carry out multiplications before additions. Two factors affect order of operation: precedence and binding.

Precedence dictates which of two **different** operations should be carried out first. For example, in

```
A + B * C
```

the standard arithmetic rules of precedence dictate that the multiplication should take place before the addition. Operations that should be carried out first are said to have a *higher* precedence than operations that should be performed later.

Different operators can sometimes have the same precedence. In C, for example, addition and subtraction are similar enough to share the same precedence.

Binding indicates which of two **similar** operations should be carried out first. By "similar", we mean operations with the same precedence (e.g. addition and subtraction in C). For example, C chooses to parse

```
A - B - C
```

as

```
(A - B) - C
```

while a language like APL would use

```
A - (B - C)
```

If we evaluate the first operation before the second (as C does), we say the operation is *left-associative*. If we evaluate the second operation before the first (as APL does), we say the operation is *right-associative.*

Occasionally, a compiler may have operations which are not associative. For example, Fortran regards

```
A .GT. B .GT. C
```

as invalid. In this case, we say the operation is *non-associative*. The precedence and associativity of operator tokens may be declared in the declarations section using the keywords

```
%left
%right
%nonassoc
```

For example,

```
%left '+' '-'
```

indicates that the + and − operations have the same precedence and are left-associative.

Associativity declarations should be given in order of precedence. Operations with lowest precedence are listed first, and those with highest precedence are listed last. Operations with equal precedence are listed on the same line. For example,

```
%right '='
%left  '+' '-'
%left  '*' '/' '%'
```

says that = has a lower precedence than + and -, which in turn have a lower precedence than `*`, `/`, and `%`. = is also right-associative, so that

```
A = B = C
```

is parsed as

```
A = (B = C)
```

Because of the way YAY specifies precedence and associativity, operators with equal precedence will always have the same associativity. For example, if A and B have equal precedence, their precedence must have been set with one of

```
%left A B
%right A B
%nonassoc A B
```

which means A and B must have the same associativity.

The names supplied with `%right`, `%left`, and `%nonassoc` may be literals or YAY identifiers. If they are identifiers, they are regarded as token names. YAY generates a `%token` directive for such names if they have not already been declared. For example, in

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

`UMINUS` is taken to be a token identifier. There is no need to define `UMINUS` as a token identifier -- a `%token` directive will be generated automatically if necessary. It is perfectly valid to have an explicit

```
%token UMINUS
```

if you want. However, it must precede the `%left` declaration.

For a more technical discussion of how precedence and binding rules affect a parser, see Chapter 8.

3.1.3 Variable/Function Declarations

The declarations section may contain standard C declarations for variables and/or functions used in the actions specified in the grammar rules section. All such declarations should be included in a block that begins with `%{` and ends with `%}`. For example,

```
%{
    int i, j, k;
    static float x = 1.0;
%}
```

gives a few variable declarations. These declarations are essentially transferred "as is" to the *beginning* of the source code that is produced by YAY. This means that they will be *external* to `"yyparse"` and therefore **`extern`** definitions.

3.1.4 Summary

The source code produced by YAY contains the following.

-- code from the declarations section
-- code given in the programs section
-- parsing tables produced by YAY to represent the grammar
-- the `"yyparse"` routine

3.2 Grammar Rules

A YAY grammar rule has the general form

```
identifier : definition ;
```

A colon separates the definition from the identifier being defined. A semicolon marks the end of the definition.

The identifiers defined in the grammar rule section are known as *non-terminal symbols.* "Non-terminal" suggests that these symbols aren't the "end of the line". Instead, they are made up of smaller things: tokens or other non-terminal symbols.

Here is a simple example of the definition of a non-terminal symbol.

```
paren_expr : '(' expr ')' ;
```

This says that a `"paren_expr"` consists of a left parenthesis, followed by an `"expr"`, followed by a right parenthesis. The `"expr"` is either a token or a non-terminal symbol defined in another grammar rule. This grammar rule could be interpreted to say that a parenthesized expression consists of a normal expression inside parentheses.

A non-terminal symbol may have more than one definition. For example, we might define `"if"` statements with

```
if_stat : IF '(' expr ')' stat ;
if_stat : IF '(' expr ')' stat ELSE stat ;
```

This definition assumes that `IF` and `ELSE` are tokens recognized by the lexical analyzer (which means that this parser's "`yylex`" can recognize keywords). The definition also assumes that "`expr`" and "`stat`" are non-terminal symbols defined elsewhere.

When a single symbol has more than one meaning, YAY lets you join the various possibilities into a single definition. Different meanings are separated by or-bars (|). Thus you could write

```
if_stat : IF '(' expr ')' stat
        | IF '(' expr ')' stat ELSE stat ;
```

This technique is highly recommended, since it makes YAY input more readable.

Definitions in a grammar may be recursive. For example,

```
list : item
     | list ',' item;
```

defines "`list`" to be one or more items separated by commas.

```
intexp : '(' intexp ')'
       | intexp '+' intexp
       | intexp '-' intexp
       | intexp '*' intexp
       | intexp '/' intexp
       | INTNUM ;
```

says that an integer expression can be another integer expression in parentheses, the sum of integer expressions, the difference of integer expressions, the product of integer expressions, the quotient of integer expressions, or an integer number standing on its own (where `INTNUM` is a token recognized by the lexical analyzer).

In recursive symbol definitions, it is often useful to have the empty string as one of the possible definitions. For example,

```
program :
      /* the empty string */
      | statement ';' program ;
```

defines a program as zero or more statements separated by semicolons.

Our definition of "`list`" above was an example of *left recursion* because "`list`" was on the left in the recursive definition. The definition of "`program`" was an example of *right recursion.* In Section 9.6, we discuss the pros and cons of the two types of recursion.

3.2.1 Recognition Actions

In addition to defining what a non-terminal symbol is, a grammar rule usually describes what to do if the non-terminal symbol is encountered in parser input. We call this a *recognition action*.

Recognition actions are specified as part of the grammar rule. They are enclosed in brace brackets in the definition, as in

```
break_stat : BREAK ';'
    { breakfn(); };
```

In this definition, `"break_stat"` is a non-terminal symbol made up of the token known as `BREAK`, followed by a semicolon. If this symbol is recognized, the parser will invoke a function named `"breakfn"`. Presumably this is a user-defined function that handles a `"break;"` statement.

Note that a semicolon is needed to mark the end of the definition even though the recognition action ends in a brace bracket. Programmers who are used to the way C works should bear this in mind.

For compatibility with some versions of YACC, YAY lets you put an equals sign (=) before the opening brace that begins a recognition action, as in

```
break_stat : BREAK ';'
    = { breakfn(); };
```

When a symbol has more than one definition, a different recognition action may be associated with each definition. We will show some examples of this in a moment.

3.2.2 Token and Symbol Values

One of the most common recognition actions is to return a value. For example, if an input is recognized as an expression to be evaluated, the parser may want to return the resulting value of the expression. To return a value, the recognition action merely assigns the value to a special variable named `$$`. For example,

```
hexdigit : '0' { $$ = 0; }
         | '1' { $$ = 1; }
                 ...
         | 'A' { $$ = 10; }
         | 'B' { $$ = 11; }
         | 'C' { $$ = 12; }
         | 'D' { $$ = 13; }
```

```
                  | 'E' { $$ = 14; }
                  | 'F' { $$ = 15; } ;
```

could be a way to convert hexadecimal digits into numeric values. In this case, `"yylex"` just returns the digits it finds and `"yyparse"` performs the actual conversion.

Another common recognition action is to return a value based on one or more of the items that make up the non-terminal symbol. Inside the recognition action, `$1` stands for the value of the first item in the symbol, `$2` stands for the value of the second item, and so on. If the item is a token, its value is the `"yylval"` value returned by `"yylex"` when the token was read. If the item is non-terminal symbol, its value is the `$$` value set by the recognition action associated with the symbol. Thus we might write

```
intexp : '(' intexp ')'
              {
                  $$ = $2;
              }
          /* value of parenthesized expression
             is expression inside parentheses */
       | intexp '+' intexp
              {
                  $$ = $1 + $3 ;
              }
         /* value of addition is sum of two
            expressions */
       | intexp '-' intexp
              {
                  $$ = $1 - $3 ;
              }
         /* value of subtraction is difference
            of two expressions */
       |  /* and so on */
       ;
```

Note that this particular definition shows that each part of a multiple definition may have a different recognition action.

In the source code for `"yyparse"`, this set of actions will be turned into a large **`switch`** statement. The cases of the **`switch`** will be the various possible recognition actions.

If no recognition action is specified for a definition, the default is

```
{ $$ = $1 ; }
```

This action just returns the value of the first item (if the item has a value).

3.2.3 Precedence in the Grammar Rules

In our discussion of the declarations section, we showed how precedence could be assigned to *operators*. Precedence can also be assigned to *grammar rules,* and this is done in the grammar input section.

One way to give a grammar rule a precedence uses the `%prec` construct.

```
%prec TOKEN
```

in a grammar rule indicates that the rule should have the same precedence as the specified token.

As a simple example, consider the case of the unary minus operator. Suppose our declaration section contains

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

In the grammar rules section, we can write

```
exp : exp '+' exp
    | exp '-' exp
    | exp '*' exp
    | exp '/' exp
    | '-' exp %prec UMINUS
    | /* and so on */
    ;
```

We could not directly set up a precedence for the unary minus, since we had already set up a precedence for the − token. Instead, we created a token named UMINUS and gave it the precedence we wanted to assign the unary minus. When we wrote out the grammar rule for the unary minus, we added

```
%prec UMINUS
```

to show that this rule should have the precedence of `UMINUS`.

As another example, we might set up precedence rules for the right shift and left shift operations of C with

```
%left RS LS
    ...
exp :
    | exp '<' '<' exp %prec LS
    | exp '>' '>' exp %prec RS
    ...
```

In this way we give the shift operations the proper precedence and avoid confusing them with the comparison operations > and <. Of course, another way to resolve this problem is to make the lexical analyzer clever enough to recognize >> and << and to return the RS or LS tokens directly.

Although symbols like `UMINUS`, `LS`, and `RS` are treated as tokens, they do not have to correspond to actual input. They may just be placeholders for operator tokens that have two different meanings.

We should note that the use of `%prec` is relatively rare in YAY. People don't usually think of `%prec` in their first draft of a grammar. `%prec` is only added in later drafts, when it is needed to resolve conflicts that appear when the rules are run through YAY.

If a grammar rule is not assigned a precedence using `%prec`, the precedence of the rule is taken from the last *token* in the rule. For example, if the rule is

```
expr : expr '+' expr
```

the last token in the rule is + (since `"expr"` is a non-terminal symbol, not a token). Thus the precedence of the rule is the same as the precedence of +.

If the last token in a rule has no assigned precedence, the rule will not have a precedence. This can result in some surprises if you aren't careful. For example, if you define

```
expr : expr '+' expr ';'
```

the last token in the rule is **;** so the rule probably won't have a precedence even if + does.

A rule that contains no tokens cannot have a precedence.

3.2.4 The Start Symbol

The first non-terminal symbol defined in the rules section is called the *start symbol*. This symbol is taken to be the largest, most general structure described by the grammar rules. For example, if you are generating the parser for a compiler, the start symbol should describe what a complete program looks like in the language to be parsed.

If you do not want the first grammar rule to be taken as the start symbol, you can use the directive

```
%start NAME
```

in your rules section. This indicates that the non-terminal symbol named `NAME` is the start symbol. `NAME` must be defined somewhere in the rules section.

The start symbol must be all-encompassing: every other rule in the grammar must be related to the start symbol. In a sense, the grammar forms a "tree": the root is the start symbol, the first set of branches are the symbols that make up the start symbol, the next set of branches are the symbols that make up the first set, and so on. Any symbol that is "outside" this tree is reported as a *useless variable* in YAY output. Useless variables are ignored by the parser -- the parser is looking for a complete start symbol, and nothing else.

3.2.5 The End Marker

The end of parser input is marked by a special token called the *end marker*. This token is often written as `$end`; the value of the token is zero.

It is the job of the lexical analyzer "yylex" to return a zero to indicate `$end` when the end of input is reached (e.g. at end of file, or at a keyword that indicates end of input).

`"yyparse"` terminates when it has parsed a start symbol followed by the end marker.

3.2.6 Declarations in yyparse

You can specify C declarations that will be local to `"yyparse"` in much the same way that you specify external declarations in the Declarations Section. Enclose the declarations in `%{` and `%}` symbols, as in

```
%{
    /* External declaratios */
%}
%%
/* Rules Section starts here */
{%
    /* Declarations here will be
       local to "yyparse" */
%}
/* Rules */
%%
/* Program section */
```

You may also put declarations at the start of recognition action code.

3.3 The Programs Section

The programs section of YAY input may contain functions that should be linked in with the `"yyparse"` routine. YAY itself does nothing with these functions -- it simply adds the source code on the end of the source code produced from the grammar rules. In this way, the functions can be compiled at the same time that the YAY-produced code is compiled.

Of course, these additional functions could be compiled separately and linked with the YAY-produced code later on (once everything is in object code format). Separate compilation of modules is strongly recommended for large parsers. However, functions that are compiled separately need a special mechanism if they want to use any manifests that are defined in the YAY-produced code, and it is sometimes simpler to make the program part of the YAY input.

For example, consider the case of `"yylex"`. Every time `"yylex"` obtains a token from the input, it returns to `"yyparse"` with a value that indicates the type of token found. Obviously then, `"yylex"` and `"yyparse"` must agree on which return values indicate which kind of tokens. Since `"yyparse"` already refers to tokens using manifest constants (created in the declarations section with the `%token` directive), it makes sense for `"yylex"` to use the same manifest constants. The lexical analyzer can do this very easily if it is compiled along with `"yyparse"`.

Size might be the determining factor. With very simple parsers, it's easier to put `"yylex"` in the Programs Section. With larger parsers, the advantages of separate compilation are well worth the extra effort.

If you are going to compile `"yylex"` or other routines separately from `"yyparse"`, use the

```
Header=file
```

option on the YAY command line. YAY will write out the manifest definitions to the file of your choice. This file can then be #included to obtain these definitions for `"yylex"` or any other routine that needs them. The manifests are already included in the generated parser code, so you only need them for separately compiled modules.

3.3.1 The Lexical Analyzer

The lexical analyzer `"yylex"` reads input and breaks it into tokens. It is `"yylex"` that determines what constitutes a token. For example, some lexical analyzers may return numbers one digit at a time while others collect numbers in their entirety before passing them to the parser.

Similarly, some lexical analyzers may recognize keywords such as `IF` or `WHILE` and will tell the parser that an `IF` token or `WHILE` token has been found. Others may not be designed to recognize keywords, so it is up to the parser itself to distinguish between keywords and other things like variable names.

As noted before, each token named in the declarations section of the YAY input is set up as a manifest constant. The value of the first token named is `257`, the value of the next is `258`, and so on. You can also set your own values for tokens by placing a positive integer after the first appearance of any token in the declarations section. For example,

```
%token AA 56
```

assigns a value of `56` to the manifest (token) symbol `AA`. This mechanism is very seldom needed, and we recommend that users avoid it whenever possible.

There is little else to say about requirements for `"yylex"`. If it wishes to return the value of a token as well as an indication of its type, the value is assigned to the external variable `"yylval"`. By default, `"yylval"` is defined as an **`int`** value but it can also be used to hold other types of values. For more information, see the description of `%union` in Chapter 7.

4. Internal Structures

In order to use YAY effectively, it is helpful to understand some of the internal workings of the parser that YAY produces. In this chapter, we will look at some of these structures.

To give us a point of reference, suppose we have produced a parser with grammar

```
%token NUM
%left '+' '-'
%left '*' '/'
%%
expr : NUM
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '(' expr ')'
     ;
```

4.1 States

As the parser reads in token after token, it switches between various *states*. You can think of a state as point where the parser says, "I have read *this* particular sequence of input tokens and now I am looking for one of *these* tokens."

For example, a parser for the C language might be in a state where it has finished reading a complete statement and is ready for the start of a new statement. It therefore expects some token that can legitimately start a statement (e.g. a keyword like `IF` or `WHILE`, or the name of a variable for an assignment).

In this state, it reads a token. Say it finds the token corresponding to the keyword `IF`. It then switches to a new state where it says "I have seen an `IF` and now I want to see the `(` that begins the `IF` condition." When it finds the `(`, it switches again to a state that says "I have found `IF(` and now I want the start of a condition expression."

States break the parsing process into simple steps. At each step, the parser knows what it has seen and what it is looking for next.

YAY assigns numbers to every possible state the parser can enter. The 0th state is always the one that describes the parser's condition before it has read any input. Other states are numbered arbitrarily.

Sometimes a particular input can only be the start of one construct. For example, the `FOR` keyword in C can only be the start of a `FOR` statement, and the `FOR` statement only has one form.

On the other hand, a grammar may have several non-terminal symbols that start the same way. In our sample grammar, all of

```
    expr '+' expr
    expr '-' expr
    expr '*' expr
    expr '/' expr
```

start with an `"expr"`. If the parser finds that the input begins with an `"expr"`, the parser has no idea which rule matches the input until it has read the operator following the first `"expr"`.

The parser chooses which state it will enter next by looking at the next input token. This token is called the *lookahead symbol* for that state.

4.2 Diagramming States

YAY uses simple diagrams to describe the various states of the parser. These diagrams show what the parser has seen and what it is looking for next. The diagrams are given in the Parser Description report prouced by YAY (see Chapter 6).

For example, consider the state where the parser has just read a complete `"expr"` at the beginning of a larger expression. It is now in a state where it expects to see one of the operators `+`, `-`, `*`, or `/`, or perhaps the `$end` marker indicating the end of input. YAY diagrams this state as

```
    $accept:  expr.$end
    expr:     expr.'+' expr
    expr:     expr.'-' expr
    expr:     expr.'*' expr
    expr:     expr.'/' expr
```

This lists the possible grammar constructs that the parser may be working on. (The first line `$accept` stands for the Start symbol.) The `"."` indicates how much the parser has read so far.

If the lookahead symbol is `*`, the parser will switch to a state diagrammed by

```
    expr:     expr '*'.expr
```

In this state, the parser knows that the next thing to come will be another `"expr"`. This means that the only valid tokens that can be read next are `(` or `NUM`, since those are the only things that start a valid `"expr"`.

4.3 State Actions

There are several possible actions that the parser can take in a state:

(a) *accept* the input;

(b) *shift* to a new state;

(c) *reduce* one or more input tokens to a single non-terminal symbol, according to a grammar rule;

(d) *goto* a new state;

(e) raise an *error* condition.

In order to decide which action to take, the parser checks the lookahead symbol (except in states where the parser can only take one possible action so the lookahead symbol is irrelevant).

This means that a typical state has a series of possible actions based upon the possible values of the lookahead symbol. In YAY output, you might see

```
'+'   shift 8
'-'   shift 7
'*'   shift 6
'/'   shift 5
')'   shift 9
 .    error
```

This says that if the parser is in this state and the lookahead symbol is `+`, the parser will shift to State 8. If the lookahead symbol is `-`, the parser will shift to State 7, and so on.

The `"."` in the final line stands for any other token not mentioned in the preceding list. If the parser finds any unexpected tokens in this particular state, it will take the `error` action.

The sections that follow explain precisely what each state action means and what the parser does to handle these actions.

4.3.1 The Accept Action

The Accept action only happens when the parser is in a state that indicates it has seen a complete input and the lookahead symbol is the end marker `$end`. When the parser takes the Accept action, `"yyparse"` terminates and returns a zero to indicate that the input was correct.

4.3.2 The Shift Action

The Shift action happens when the parser is part way through a grammar construct and a new token is read in. As an example, State 4 in our sample parser is diagrammed with

```
expr:     expr.'+' expr
expr:     expr.'-' expr
expr:     expr.'*' expr
expr:     expr.'/' expr
expr:     '(' expr.')'

'+'   shift 8
'-'   shift 7
'*'   shift 6
'/'   shift 5
')'   shift 9
 .    error
```

This shows that the parser will shift to various other states depending on the value of the lookahead symbol. For example, if the lookahead symbol is `*` the parser shifts to State 6 which has the diagram

```
expr:     expr '*'.expr

NUM   shift 2
'('   shift 1
 .    error

expr  goto 11
```

In this new state, the parser has further shifts it can make, depending on the next lookahead symbol.

When the parser shifts to a new state, it saves the previous state on a stack called the *state stack*. The stack provides a history of the states that the parser has passed through while it was reading input. It is also a control mechanism as will be seen later in this chapter.

Paralleling the state stack is a *value* stack that records the values of tokens and non-terminal symbols encountered while parsing. The value of a token is the "`yylval`" value returned by "`yylex`" at the time the token was read. The value of a non-terminal symbol is the `$$` value set by the recognition action associated with that symbol's definition. If the definition didn't have an associate recognition action, the value of the symbol is the value of the first item in the symbol's definition.

At the same time that the Shift action pushes the current state onto the state stack, it also pushes the "`yylval`" value of the lookahead symbol (token) onto the value stack.

4.3.3 The Reduce Action

The Reduce action takes place in states where the parser has recognized all the items that make up a non-terminal symbol. For example, the diagram of State 9 in our sample grammar is

```
expr:     '(' expr ')'.

  .    reduce (6)
```

At this point, the parser has seen all three components that make up the non-terminal symbol "`expr`":

```
'('
expr
')'
```

As the line

```
  .    reduce (6)
```

shows, it doesn't matter what the lookahead symbol is at this point. The non-terminal symbol has been recognized, and the parser is ready for a Reduce action. (Note that the (6) in the above line just means that the parser has recognized the non-terminal symbol defined in rule (6) of the grammar. We will say more about this in a later chapter.)

The Reduce action performs a number of operations. First, it pops states off the state stack. If the recognized non-terminal symbol had N components, a reduction pops N-1 states off the stack. In other words, the parser goes back to the state it was in when it first began to gather the recognized construct.

Next, the Reduce action pops values off the value stack. If the definition that is being reduced consisted of N items, the Reduce action conceptually pops N values off the stack. The topmost value on the stack is assigned to `$N`, the next to `$N-1`, and so on down to `$1`.

Once the Reduce action has gathered all the `$X` values, the parser invokes the recognition action that was associated with the grammar rule being reduced. This recognition will use the `$X` values to come up with a `$$` value for the non-terminal symbol. This value is pushed onto the value stack, thereby replacing the N values that were previously on the stack.

If the non-terminal symbol had no recognition action, or if the recognition action did not set `$$`, the parser puts the value of `$1` back on the stack. (In reality, the value is never popped off.)

Lastly, the Reduce action sets things up so that the lookahead symbol seems to be the non-terminal symbol that was just recognized. For example, it may say that the lookahead symbol is now an `"expr"` instead of a token.

4.3.4 The Goto Action

The Goto action is a continuation of the Reduce process. Goto is almost the same as Shift -- the only difference is that the Goto action takes place when the lookahead symbol is a non-terminal symbol while a Shift takes place when the lookahead symbol is a token.

For example, State 6 in our sample grammar reads

```
expr:    expr '*'.expr

NUM   shift 2
'('   shift 1
 .    error

expr  goto 12
```

The first time the parser enters this state, the lookahead symbol will be a token and the parser will Shift on the token into some state where it will begin to gather an `"expr"`. When it has a complete `"expr"`, it will perform a Reduce action that will return to this state and set the lookahead symbol to `"expr"`. Now when the parser has to decide what to do next, it sees that it has an `"expr"` for the lookahead symbol and therefore takes the Goto action and moves to State 12.

The Shift action pushes the current state onto the state stack. The Goto does not have to do this -- the state was on the stack already. Similarly, Shift pushes a value onto the value stack, but Goto does not, since the value corresponding to the non-terminal symbol was already put on the value stack by the Reduce action.

When the parser reaches the new state, the lookahead symbol will be restored to whatever it was at the time of the Reduce action.

Essentially then, a Goto is like a Shift except that it takes place when you come *back* to a state with the Reduce action. Also, a Shift is based on the value of a single input token while a Goto is based on a non-terminal symbol.

4.3.5 The Error Action

The parser takes the Error action when it encounters any input token which cannot validly appear in a particular input location. When this happens, the parser raises the `"error"` condition. Since error-handling can be quite complicated, we will devote the whole of the next chapter to the subject.

5. Error-Handling

If a piece of input is invalid, the parser can do nothing with it. Except in extreme cases, however, it is inappropriate for the parser to stop all processing as soon as an error is found. Instead, the parser should skip over the invalid input and resume parsing as soon after the error as possible. In this way, the parser can find many syntax errors in one pass through the input, saving time and trouble for the user.

YAY therefore tries to generate a parser that can "restart" as soon as possible after an error occurs. YAY does this by letting you specify points at which the parser can pick up after errors. You may also dictate what special processing should take place if an error is encountered at one of these points.

5.1 The "error" Symbol

YAY's error handling facilities use the keyword **`error`** to signify an erroneous input. As a result, **`error`** may not be used as the name of a user-defined token or non-terminal symbol.

You should put "**`error`**" in your grammar rules where error recovery might take place. For example, you might write

```
statement: error
    | /* other definitions of a statement */;
```

This tells YAY that errors may occur in statements, and that after an error, the parser is free to restart parsing at the end of a complete statement.

5.2 The Error Condition

As noted in the last chapter, YAY will take the Error action if it finds an input that is not valid in a particular location. The Error action has the following steps.

(a) See if the current state has a Shift action associated with the **`error`** symbol. If it does, shift on this action.

(b) If the current state has no such action, pop the state off the stack and check the next state. Also pop off the top value on the value stack, so that the state stack and value stack stay in synch.

(c) Repeat (b) until the parser finds a state that can shift on the **`error`** symbol.

(d) Take the Shift action associated with the **`error`** symbol. Note that this pushes the current state on the stack, i.e. the state that is capable of handling errors. No new value is pushed onto the value stack -- the parser keeps whatever value was already associated with the state that can handle errors.

When the parser shifts out of the state that can handle errors, the lookahead symbol will be whatever token caused the error condition in the first place. The parser will then try to proceed with normal processing.

Of course, it is quite possible that the original lookahead symbol is invalid in the new context. If the lookahead symbol causes an error again, it is discarded and the error condition stays in effect. The parser will continue to read new tokens and discard them until it finds a token that can validly follow the **`error`**. The parser will then take whatever action is associated with the valid token.

In a typical grammar, the state that has been handling errors will eventually be popped off the stack in a Reduce operation.

Notice that the parser always *Shifts* on the **`error`** token. It will never Reduce on **`error`**, even if the grammar has a state where **`error`** is associated with a Reduce action.

In some situations, an error condition will be raised and the parser will pop all the way to the bottom of the state stack without finding a state that can handle the **`error`** symbol. For example, the grammar may have no provisions for error recovery. In this case, `"yyparse"` simply terminates and returns a 1 to its caller.

5.3 Examples

As a simple example, consider a parser for a simple desk calculator. All statements end in a semicolon. Thus we might see the rule

```
statement : var '=' expr ';'
          | expr ';'
          | error ';'
          ;
```

When an error occurs in input, the parser will pop back through the state stack until it comes to a state where the **`error`** symbol is recognized. For example, the state might be diagrammed as

```
$accept:   .statement $end

error  shift 2
NUM    shift 4
.      error

var       goto 7
expr      goto 3
statement goto 5
```

If an error occurs anywhere in an input statement, the parser will pop back to this state, then Shift to State 2. State 2 will look like

```
statement:   error.';'

';'   shift 6
 .    error
```

In other words, the next token must be a semicolon. If it isn't, another error occurs. The parser pops back to the previous state and takes the **`error`** shift again. Input will be discarded token by token until a semicolon is found. When the semicolon is found, the parser will be able to shift from State 2 to State 6 which is

```
statement:  error ';'.

  .    reduce (3)
```

The erroneous line is reduced to a "`statement`" non-terminal symbol.

Now this example is simple, but it has its drawbacks. It will get you into trouble if the grammar has any concept of block structure or parenthesization. Why? Once an error occurs, the rule

```
statement : error ';'
```

effectively tells the parser to discard absolutely everything until it finds a **`;`** character. If you have a parser for C, for example, it would skip over important characters like `)` or `}` until it found a semicolon. Your parentheses and braces would be out of balance for the rest of the input and the whole parsing process would be a waste of time. The same principle applies to any rule that shows the **`error`** token followed by some other non-null symbol: it can lead to hopeless confusion in a lot of grammars.

It is safer to write the rule in a form like

```
statement : error
          | ';'
          | /* other stuff */
```

In this case, the **`error`** token only matches material until the parser finds something else it recognizes (e.g. the semicolon). Once this happens, the **`error`** state will be reduced to a "`statement`" symbol and popped off the stack. Parsing can then proceed as usual.

5.4 Error Recognition Actions

The easiest way to generate an error message is to associate a recognition action with the grammar rule that recognizes the error. You can do something simple like

```
statement: error
    {
        printf("You made an error!\n");
    }
```

or you can be fancier, as in

```
line: error '\n' prompt line
    { $$ = $4; };
prompt: /* null token */
    { printf("Please re-enter line.\n"); };
```

If an error occurs, the parser skips until it finds a new-line character. After the new-line, it will always find a null token matching `"prompt"`, and the recognition action for `"prompt"` will print out the message

```
Please re-enter line
```

The final symbol in the rule is another `"line"`, and the action after the **`error`** rule shows that the result of the rule (`$$`) should be the material associated with the second input line.

All this means that if the user makes a mistake entering an input line, the parser prints out an error message and accepts a second input line in place of the first. This allows for an interactive user to correct an input line that was incorrectly typed the first time.

Of course, this set-up will only work if the user doesn't make an error the second time the line is typed too. If the next token he or she types is also invalid, the parser will discard the token and decide that it's still gobbling up the original error.

5.5 The yyclearin Macro

After an Error action, the parser will restore the lookahead symbol to the value it had at the time the error was detected. However, this is sometimes undesirable.

For example, your grammar may have a recognition action associated with the **`error`** symbol and this may read through the next lot of input until it finds the next sure-to-be-valid data. If this happens, you certainly don't want the parser to pick up the old lookahead symbol again once error recovery is finished.

If you want the parser to throw away the old lookahead symbol after an error, put

```
yyclearin ;
```

in the recognition action associated with the **`error`** symbol. `"yyclearin"` is a macro that expands into code that discards the lookahead symbol.

5.6 The yyerror Function

The first thing the parser does when it performs the Error action is to call a function named `"yyerror"`. This happens *before* the parser begins going down the state stack in search of a state that can handle the **`error`** symbol. `"yyerror"` must be supplied by the user -- note that its name is in lower case.

The simplest `"yyerror"` functions either abort the parsing job or just return so that the parser can perform its standard error handling.

The `"yyerror"` function is passed one argument: a character string describing the type of error that just took place. This string is almost always

```
"Syntax error"
```

The only other argument strings that might be passed are

```
"Not enough space for parser stacks"
"Parser stack overflow"
```

which are used when the parser runs out of memory for the state stack.

Once `"yyerror"` returns to `"yyparse"`, the parser will proceed popping down the stack in search of a state that can handle errors.

If another error is encountered soon after the first, `"yyerror"` will *not* be called again. The parser considers itself to be in a "potential error" situation until it finds three correct tokens in a row. This avoids the torrents of error messages that often occur as the parser wades through input in search of some recognizable sequence.

Once the parser has found three correct tokens in a row, it leaves the "potential error" situation. If a new error is found later on, `"yyerror"` will be called again.

5.6.1 Changing yychar in yyerror

The `"yyerror"` function may change the value of `"yychar"` (the variable that indicates the type of token just read by `"yylex"`). If `"yyerror"` does this, `"yyparse"` immediately gets rid of the error condition and tries to make a shift or reduce using the new `"yychar"` token. If there is a valid action that can be taken, `"yyparse"` will proceed as if the error never happened.

This behavior can help you deal with at least two kinds of unusual situations:

(a) When `"yyerror"` is attempting to correct a problem by supplying the correct input. For example, it may be possible to determine what input should have been received; the classic example is a compiler which can figure out that a particular statement should have ended in a semicolon. In this case, `"yyerror"` can supply the missing semicolon by assigning an appropriate value to `"yychar"`. If `"yyerror"` guessed correctly, action can proceed as if the problem didn't occur.

(b) When a particular token can be interpreted in different ways in different contexts. For example, suppose that `X` can be a keyword in some contexts and a variable name in others. You can write `"yylex"` so that it always interprets `X` as a keyword. If `"yyerror"` is invoked because that keyword is invalid in a particular context, `"yyerror"` can change `"yychar"` and see if the input is valid when `X` is interpreted as a variable name.

If `"yyerror"` changes `"yychar"`, `"yyerror"` may have to inform `"yylex"` of the situation so that `"yylex"` can return the proper input token the next time it is called.

5.7 The yyerrflag Variable

`"yyerrflag"` is an external integer variable whose value must be `0`, `1`, `2`, or `3`. `YYABORT` (described in a later section) is called if `"yyerrflag"` does not have one of these values.

As we mentioned earlier, `"yyparse"` doesn't leave its error state until three consecutive valid tokens have been read. `"yyerrflag"` is used in this process.

If `"yyerrflag"` is zero, `"yyerror"` will be invoked the next time an error is encountered. As soon as `"yyerror"` is invoked, it sets `"yyerrflag"` to `3`. Each time a `"yyparse"` shifts on a valid token, `"yyerrflag"` is decremented, until it gets to zero. At this point, `"yyparse"` leaves its error state; if a new error occurs, `"yyerror"` will be called again.

There are two ways in which actions can use `"yyerrflag"`. First, they can check the variable to see if there has been a recent error. For example, if an action encounters a semantic error, it can check `"yyerrflag"` to see if it is non-zero (meaning that there has been a recent syntax error). If `"yyerrflag"` is non-zero, the semantic error is almost certainly a result of the previous syntax error. Using this information, the action may choose to modify or suppress the semantic error message.

Second, actions can set `"yyerrflag"` to a value if they want to prevent calls to `"yyerror"`. As long as `"yyerrflag"` is non-zero, `"yyerror"` will not be called. This may be useful if an action is smart enough to realize that it is still recovering from a previous error and `"yyerror"` should not be called again for a while.

5.8 The yyerrok Macro

In some situations, you may want `"yyerror"` to be called even if the parser hasn't seen three correct tokens since the last error.

For example, suppose you have a parser for a line by line desk calculator. A line of input contains errors, so `"yyerror"` is called. `"yyerror"` prints an error message to the user, throws away the rest of the line, and prompts for new input. If the next line contains an error in the first three tokens, the parser will normally start discarding input *without* calling `"yyerror"` again. This means that `"yyerror"` doesn't print an error message for the user, even though the input line is wrong.

To avoid this problem, you can explicitly tell the parser to leave its "potential error" state, even if it hasn't yet seen three correct tokens. Simply say

```
yyerrok ;
```

as part of the error recognition action. For example, you might have the rule

```
expr : error
     {
         yyerrok;
         printf("Please re-enter line.\n");
         yyclearin;
     }
```

`"yyerrok"` is a macro that is expanded into code that takes the parser out of its "potential error" state and lets it start fresh. More precisely, `"yyerrok"` sets `"yyerrflag"` to zero.

5.9 Other Error Support Routines

`YYABORT` halts `"yyparse"` in midstream and immediately returns a `1`. To the function that called `"yyparse"`, this means that `"yyparse"` failed for some reason.

`YYACCEPT` halts the parser in midstream and returns a `0`. To the function that called `"yyparse"`, this means that `"yyparse"` terminated successfully, even if the entire input has not yet been scanned.

`YYERROR` (note that this is upper case) is a macro that "fakes" an error. When `YYERROR` is encountered in the code, the parser will react as if it just saw an error and will go about recovering from the error. In Chapter 9, we will give an example of how `YYERROR` can be useful.

6. YAY Output

YAY can produce several output files. Options on the YAY command line dictate which files are actually generated.

The most important output file is the one containing source code that can be compiled into the actual parser. The name of this file is specified with the `Parser=file` command line option.

Another possible output file contains manifest definitions. The name of this file is specified with `Header=file` on the command line. This file is a distillation of the declarations section of the YAY input. For example, all the `%token` directives are restated in terms of manifest definitions.

```
%token IF
```

would appear as

```
#define IF 257
```

in the C version of the manifests (assuming that `IF` was the first token in the declarations section). By including this file with

```
#include "filename"
```

separately compiled modules can make use of all the pertinent definitions in the YAY input.

The third output file that YAY can produce is called the Parser Description. The name of the file is specified with `Description=file` on the command line. The Parser Description is split into three sections:

(a) a summary of the grammar rules;

(b) a list of state descriptions;

(c) a list of statistics for the parser generated by YAY.

In the sections that follow, we will show what the Parser Description looks like for the following grammar:

```
%token IF ELSE A
%%
stmt : IF stmt ELSE stmt
     | IF stmt
     | A
     ;
```

6.1 Rules Summary

The rules summary section of the Parser Description begins with the command line used to invoke YAY. This is intended to serve as a "heading" for the output material. We'll use the command line

```
yay infile.y description=out +verbose
```

YAY input is in the file `infile.y` and output is written to the file `out`. We have specified the `+Verbose` option because this provides the largest amount of output. See Appendix C for more information on the YAY command line.

Next comes a summary of the grammar rules. In our example, we will have

```
Rules:
    (0)  $accept:  stmt $end
    (1)  stmt:     IF stmt ELSE stmt
    (2)  stmt:     IF stmt
    (3)  stmt:     A
```

The 0th rule will always be the definition for a symbol named `$accept`. This describes what a complete input looks like: the Start symbol followed by the end marker. Other rules are those given in the grammar.

YAY puts a formfeed character on the line after the last grammar rule so that the next part of the Parser Description starts on a new page.

6.2 State Descriptions

The Parser Description output contains complete descriptions of every possible state. For example, here is the description of one state from our sample grammar.

```
State 2
    stmt :  IF.stmt ELSE stmt
    stmt :  IF.stmt
```

```
IF   shift 2
A    shift 1
.    error

stmt     goto 4
```

By now, this sort of diagram should be familiar to you. The numbers after the word `"shift"` indicate the state to which the parser will Shift if the lookahead symbol happens to be `IF` or `A`. If the lookahead symbol is anything else, the parser raises the error condition and starts error recovery.

If the parser pops back to state 2 by means of a Reduce Action, the lookahead symbol will now be `"stmt"` and the parser will Goto state 4.

As another example of a state, here is State 1.

```
State 1
    stmt :  A.  (3)

    .      reduce (3)
```

This is the state that is entered when an `A` token has been found. The (3) on the end of the first line is a *rule number*. It indicates that this particular line sums up the whole of the third grammar rule that was specified in the YAY input. The line

```
.    reduce (3)
```

indicates that no matter what token comes next, we can reduce this particular input using grammar rule (3) and say that we have successfully collected a valid `"stmt"`. The parser will perform a reduction by popping the top state off the stack and setting the lookahead symbol to `"stmt"`.

It is important to distinguish between

```
A  shift 1
```

in State 2 and

```
.  reduce (3)
```

in State 1. In the Shift instruction, the number that follows is the number of a *state*. In the Reduce instruction, the number that follows is the number of a *grammar rule* (using the numbers given to the grammar rules in the first part of the Parser Description). The Parser Description always encloses rule numbers in parentheses, and leaves state numbers as they are.

The complete state descriptions for the grammar are given below.

```
State 0
         $accept: .stmt $end

    IF    shift 2
    A     shift 1
    .     error

    stmt    goto 3

State 1
    (3)  stmt:     A.

    .     reduce (3)

State 2
         stmt:     IF.stmt ELSE stmt
         stmt:     IF.stmt

    IF    shift 2
    A     shift 1
    .     error

    stmt    goto 4

State 3
         $accept:  stmt.$end

    $end  accept
    .     error

State 4
         stmt:     IF stmt.ELSE stmt
    (2)  stmt:     IF stmt. [ $end ELSE ]

  Shift/reduce conflict (5,2) on ELSE
    ELSE  shift 5
    .     reduce (2)

State 5
         stmt:     IF stmt ELSE.stmt

    IF    shift 2
    A     shift 1
    .     error

    stmt    goto 6

State 6
    (1)  stmt:     IF stmt ELSE stmt.

    .     reduce (1)
```

The parser always begins in State 0, i.e. in a state where no input has been read yet. An acceptable input is a `"stmt"` followed by the end marker. When a `"stmt"` has been collected, the parser goes to State 3. In State 3, the required end marker `$end` indicates that the input should be accepted. If anything else is found, it is excess input and means an error.

In State 4, the rule labelled (2) has

```
[ $end ELSE ]
```

on the end. This just means that the parser expects to see one of these two tokens next.

6.3 Conflict Summary

As noted in the sample output, there is a shift/reduce conflict in State 4. If an `ELSE` is encountered while the parser is in this state, the parser doesn't know whether to shift (into State 5) or to reduce using rule (2). Thus YAY prints the message

```
Shift/reduce conflict (5,2) on ELSE
```

After the state descriptions, YAY prints a summary of all such conflicts. With our sample grammar, the output contains

```
Conflicts:
    State  Token        Action
        4  ELSE         shift 5
        4  ELSE         reduce (2)
```

Each line in this summary gives the number of the state in which the conflict occurs, the token(s) that cause the conflict, and the conflicting actions.

6.4 Parser Statistics

The last section of the Parser Description is a set of statistics summarizing YAY's work. Here are the stats we got when we ran our sample grammar through YAY.

```
4 rules, 5 tokens, 2 variables, 7 states
Memory:  max = 8K
States: 3 wasted, 4 resets
Items:  18, 0 kernel, (2,0) per state, maxival=16 (1 w/s)
Lalr:  1 calls, 2 recurs, (0 trans, 12 epred)
Actions:  0 entries, gotos: 0 entries
Exceptions: 1 states, 4 entries
Simple state elim: 0%, Error default elim: 33%
```

```
Optimizer: in 0, out 0
Size of tables:  24 bytes
2 seconds, final mem = 4
1 shift/reduce conflict
```

Some of these values are machine independent (e.g. the number of rules), others are machine dependent (e.g. the amount of memory used), and some can be different every time you run the job (e.g. time elapsed while YAY was running).

The statistic lines are described below. Many of these will be of no interest to the normal user; YAY only generates them for the use of those maintaining the YAY software. A number of the statistics refer to shift/reduce or reduce/reduce conflicts. We will discuss these in a later chapter.

`4 rules, 5 tokens, 2 variables, 7 states`

The four rules are the grammar rules given in the first part of the Parser Description. The five tokens are `A`, `IF`, `ELSE`, `$end`, and `error` (which is always defined, even if we don't use it in this grammar). The two variables are the non-terminal symbols, `"stmt"` and the special `$accept`. The seven states are states 0 to 6.

`Memory:  max = 8K`

This gives the maximum amount of dynamic memory that YAY required while producing the parser. This line may also have a "success rate" that tells how often YAY succeeded in having enough memory to handle a situation and how often it had to ask for more memory.

`States: 3 wasted, 4 resets`

The algorithm that constructs states from the grammar rules makes a guess at the number of states it will need, very early on in the YAY process. If this guess is too high, the excess states are said to be "wasted".

When creating states from the various grammar rules, it is sometimes found that a state from one rule duplicates the state from another (for example, there were two rules that started with `IF` in our example above). In the final parsing tables, such duplicate states are merged together into a single state. The number of "resets" is the number of duplicate states formed, then merged.

`Items: 18, 0 kernel, (2,0) per state, maxival=16 (1 w/s)`

A state is made of items, and the kernel items are an important subset of these: the size of the resulting parsing tables and the running time for YAY are proportional to the number of items and kernel items. The rest of the statistics in this line are not of interest to normal users.

`Lalr: 1 call, 2 recurs, (0 trans, 12 epred)`

This gives the number of calls and recursive calls to the conflict resolution routine. The parenthesized figures are related to the same process. In some ways, this is a measure of the complexity of the grammar being parsed. This line will not appear if there are no reduce/reduce or shift/reduce conflicts in your grammar.

`Actions: 0 entries, gotos: 0 entries`

This gives the number of entries in the tables `"yyact"` and `"yygo"`. `"yyact"` keeps track of the possible Shifts that a program may make and `"yygo"` keeps track of the "gotos" that take place at the end of states.

`Exceptions: 1 states, 4 entries`

This gives the number of entries in the table `"yydef"`, yet another table used in YAY. `"yydef"` keeps track of the possible Reduce, Accept, and Error actions that a program may make.

`Simple state elim: 0%, Error default elim: 33%`

The percentage figures indicate how much table space could be saved through various optimization processes. The better written your grammar, the greater the percentage of space that can be saved. Therefore, high percentages here are an indication of a well-written grammar.

`Optimizer: in 0, out 0`

More optimization statistics: not of interest to normal YAY users.

`Size of tables: 24 bytes`

The size of the tables generated to represent the parsing rules. This size is given in bytes on the host machine, so it will be inaccurate if a cross-compiler is being used on the eventual source code output. The size does not include stack space used by `"yyparse"` or debug tables obtained by defining `YYDEBUG`.

`2 seconds, final mem = 4`

Total real time that YAY used to produce the parser, and the final dynamic memory of the parser (in K bytes).

`1 shift/reduce conflict`

Number of conflicts in the grammar.

7. Types

Earlier we mentioned that `"yylval"` is **`int`** by default, as are `$$`, `$1`, `$2`, etc. If you want these to have different types, you may redeclare them in the declarations section of the YAY input. This is done with a statement of the form

```
%union {
     /* possible types for "yylval" and
      * $$, $1, $2, etc.  */
}
```

For example, suppose "yylval" can be either integer or floating point. You might write

```
%union {
     int intval;
     float realval;
}
```

in the declarations section of the YAY input. YAY converts the `%union` statement into the following C source.

```
typedef union {
    int intval;
    float realval;
} YYSTYPE;
```

`"yylval"` is always declared to have type `YYSTYPE`. If no `%union` statement is given in the YAY input, it will use

```
#define YYSTYPE int
```

Once `YYSTYPE` has been defined as a union, you may specify a particular interpretation of the union by including a statement of the form

```
%type <interpretation> SYMBOL
```

in the declarations section of the YAY input. The "interpretation" enclosed in the angle brackets is the name of the interpretation of the union variable that you want to use. The `SYMBOL` name is the name of a non-terminal symbol defined in the grammar rules. For example, you might write

```
%type <intval> intexp
%type <realval> realexp
```

to indicate that an integer expression has an integer value and a real expression has a floating point value.

Tokens may also be declared to have types. This is done in the `%token` statement in the declaration section, and again shows the union interpretation in angle brackets, e.g.

```
%token <realval> floatnum
```

If you use types in your YAY input, YAY will enforce compatibility of types in all expressions. For example, if you write

```
$$ = $2
```

in an action, YAY will demand that the two corresponding tokens have the same type; otherwise, the assignment will be marked as invalid. The reason for this is that YAY must always know what interpretation of the union is being used in order to generate correct code.

The Default Action

The default action associated with any rule can be written as

```
$$ = $1
```

which means that the value of associated with `$1` on the value stack is assigned to `$$` on the value stack when the rule is reduced. If, for example, `$1` is an integer, then `$$` will be the same integer after the reduction occurs.

On the other hand, suppose that the recognition action associated with a rule explicitly states

```
$$ = $1
```

This explicit assignment may not have the same effect as the implicit assignment. For example, suppose that you define

```
%union {
    float floatval;
    int intval;
}
```

Also suppose that the type associated with `$$` is `"floatval"` and the type associated with `$1` is `"intval"`. Then the explicit statement

```
$$ = $1
```

performs an integer to floating point conversion when the value of `$1` is assigned to `$$`, whereas the implicit statement did an integer to integer assignment and did *not* perform this conversion. You must therefore be careful and think about the effects of implicit vs. explicit assignments.

8. Ambiguities

Suppose we have a grammar with the rule

```
expr : expr '-' expr ;
```

and the parser is reading an expression of the form

```
expr - expr - expr
```

The parser reads this token by token, of course, so after three tokens it will have

```
expr - expr
```

The parser recognizes this form. In fact, the parser could reduce this right away into a single `"expr"` according to the given grammar rule.

However, the parser has a problem. At this point, the parser doesn't know what comes next, and perhaps the entire line will be something like

```
expr - expr * expr
```

If it is, the precedence rules say that the multiplication should be performed before the subtraction, so handling the subtraction first is incorrect. The parser must therefore read another token to see if it is really all right to deal with the subtraction now, or if the correct action is to skip the subtraction for the moment and deal with whatever follows the second `"expr"`.

In terms of parser states, this problem boils down to a choice between *reducing* the expression

```
expr - expr
```

or *shifting* and acquiring more input before making a reduction. This is known as a *shift/reduce conflict.* Sometimes a parser must also choose between two possible reductions. This kind of situation is called a *reduce/reduce conflict.*

In case you are curious, there is no such thing as a shift/shift conflict. To see why this is impossible, suppose that we have the following definitions.

```
thing : a b
      | a c
      ;
b : T rest_of_b;
c : T rest_of_c;
```

If the parser is in the state where it has seen `"a"`, you would have the diagram

```
thing : a.b
thing : a.c
```

You might think that if the lookahead symbol was the token `T`, the parser would be confused, since `T` is the first token of both `"b"` and `"c"`. However, there is no confusion at all. The parser just Shifts to a state that we could diagram as

```
b : T.rest_of_b
c : T.rest_of_c
```

8.1 Resolving Conflicts by Precedence

The precedence directives (`%left`, `%right`, and `%nonassoc`) let YAY-produced parsers resolve shift/reduce conflicts in an obvious way:

(a) The precedence of a Shift operation is defined to be the precedence of the token on which the Shift takes place.

(b) The precedence of a Reduce operation is defined to be the precedence of the grammar rule that the Reduce operation uses.

If you have a shift/reduce conflict, and the Shift and Reduce operations both have a precedence, the parser chooses the operation with the higher precedence.

8.2 Disambiguating Rules

Precedence cannot resolve conflicts if one or both conflicting operations have no precedence. For example, consider the following.

```
statmt:  IF '(' cond ')' statmt
    |    IF '(' cond ')' statmt ELSE statmt ;
```

Given this rule, how should the parser interpret the input

```
IF ( cond1 ) IF ( cond2 ) statmt1 ELSE statmt2
```

There are two equally valid interpretations of this input:

```
IF ( cond1 ) {
     IF ( cond2 ) statmt1
     ELSE statmt2
}
```

and

```
IF ( cond1 ) {
     IF ( cond2 ) statmt1
}
ELSE statmt2
```

In a typical grammar, the `IF` and `IF-ELSE` statements would not have a precedence, so precedence could not resolve the conflict. Thus consider what happens at the point when the parser has read

```
IF ( cond1 ) IF ( cond2 ) statmt1
```

and has just picked up `ELSE` as the lookahead symbol.

(1) It can immediately Reduce the

```
IF ( cond2 ) statmt1
```

using the first definition of `"statmt"` and obtain

```
IF ( cond1 ) statmt ELSE ...
```

thereby associating the `ELSE` with the first `IF`.

(2) It can Shift, which means ignoring the first part (the `IF` with `"cond1"`) and going on to handle the second part, thereby associating the `ELSE` with the second `IF`.

In this case, most programming languages choose to associate the `ELSE` with the second `IF`, i.e. they want the parser to Shift instead of Reduce. Because of this (and other similar situations), YAY-produced parsers are designed to use the following rule to resolve shift/reduce conflicts.

Disambiguating Rule 1:

If there is a shift/reduce conflict in situations where no precedence rules have been created to resolve the conflict, the default action is to Shift. The conflict will also be reported in the YAY output so you can check that Shifting is actually what you want. If it isn't what you want, the grammar rules will have to be rewritten.

This is known as a disambiguating rule because it helps YAY-produced parsers resolve ambiguities. The rule is only used in situations where precedence rules cannot resolve the conflict. If both the Shift operation and the Reduce operation have an assigned precedence, the parser can compare precedences and decide which operation to perform first. Even if the precedences are equal, the precedences must have originated from either `%left`, `%right`, or `%nonassoc`, so the parser knows how to handle the situation.

The only time the disambiguating rule is needed is when one or both of the Shift or Reduce operations does not have an assigned precedence.

In a similar vein, YAY-produced parsers use the following rule to resolve reduce/reduce conflicts.

Disambiguating Rule 2:

> If there is a reduce/reduce conflict, the parser will always Reduce by the rule that was given *first* in the rules section of the YAY input. Again, the conflict will be reported in the YAY output so that users may ensure that the choice of Reduction is correct.

Note that precedence is *not* consulted in reduce/reduce conflicts. YAY always Reduces by the earliest grammar rule, regardless of precedence.

Finally, YAY-produced parsers use a third disambiguating rule to prevent excessive errors.

Disambiguating Rule 3:

> If the parser is asked to choose between shifting on **`error`** or on any other item, it will choose the non-error item.

The disambiguating rules are simple to state, but they can have complex repercussions if the grammar is non-trivial. If the grammar is sufficiently complicated, these simple rules for resolving conflicts may not be capable of handling all the necessary intricacies in the way you want. You should pay close attention to all conflicts noted in the parsing table report produced by YAY and should ensure that the default actions taken by the parser are the desired ones.

Note: the conflict between the rules

```
statmt:  IF '(' cond ')' statmt
     |   IF '(' cond ')' ELSE statmt ;
```

can be resolved by adding

```
%left ELSE
```

to the declarations section of the YAY input. This puts a precedence on the `ELSE` action and says that it is left-associative. This is another example of using a precedence to resolve an ambiguity.

8.3 Conflicts in YAY Output

If your grammar has shift/reduce or reduce/reduce conflicts, you will also see a table of conflicts in the statistics section of the Parser Description. For example, if we change the rules section of our sample grammar to

```
stmt : IF stmt ELSE stmt
     | IF stmt
     | stmt stmt
     | A ;
```

we get the following conflict report.

```
Conflicts:
    State  Token      Action
        5  IF         shift 2
        5  IF         reduce (3)
        5  A          shift 1
        5  A          reduce (3)
```

This shows that State 5 has two shift/reduce conflicts. If the parser is in State 5 and encounters an IF token, it can shift to state 2 or reduce using rule 3. If the parser encounters an A token, it can shift to state 1 or reduce using rule 3. This is summarized in the final statistics with the line

```
2 shift/reduce conflicts
```

Reading the conflict report shows you what action the parser takes in case of a conflict -- the parser always takes the *first* action shown in the report. This action will be chosen in accordance with the two disambiguating rules.

9. Advanced Features

In this chapter, we describe more specialized features of YAY.

9.1 LALR(2) Grammars

An LALR(2) grammar has one more level of "lookahead" than an LALR(1) grammar. When trying to decide how to parse a given input, an LALR(1) parser sometimes looks at the next input token to see if that makes a difference. In the same situation, an LALR(2) parser can look at the next *two* tokens.

LALR(2) grammars are described in the same way as LALR(1) grammars. To indicate that you want an LALR(2) parser, you must put the option `+LR2` on the command line that invokes YAY.

An LALR(2) parser can perform a *lookahead* action as well as Shift, Reduce, Goto, Accept, and Error. For example, in a state diagram you might see

```
A    shift 1
B    reduce (2)
C    lookahead
```

This means that if token `A` is encountered in this state, the parser will shift to State 1; if token `B` is encountered, it will reduce using Rule (2); and if token `C` is encountered, it will lookahead.

9.1.1 The Lookahead Action

A state that has a Lookahead action has a list of possible states the parser can enter next. These states conflict with each other -- the Lookahead action isn't needed if there is no conflict. The parser attempts to resolve the conflict by reading one additional token. If the grammar is set up properly, this token will be valid in only one of the possible contexts, so the parser can choose that rule instead of the other possibilities.

Thus the Lookahead action tests all the possibilities in the list. It begins by making a private copy of the state stack. (Actually, it just sets things up so that it looks like it has a separate copy of the state stack -- it doesn't really make a copy.) Next, it calls on a routine named `"yy2lex"` (described later) to obtain a second lookahead token.

The Lookahead action then begins going through the possible rules on its list. Some of these rules may be Shifts, while others are Reduces -- the reductions are what make it necessary to simulate a copy of the state stack, because you don't want to play with the true stack.

The Lookahead action takes the action associated with each possibility in the list, entering new states through Shifts or Reduces. It then sees if the second lookahead token causes an error in this new state. If an error occurs, the parser pops back to the original Lookahead state and checks the next possibility on the list.

When it finally finds a rule where the second lookahead symbol doesn't cause an error, it pops back to the original Lookahead state one last time. It gets rid of the stack set-up that the Lookahead action was using and returns to the old state stack. It then takes the action that it has discovered in the list of possibilities.

Note that the Lookahead action takes the first possible rule where the second lookahead symbol is valid. It is, of course, possible that there are other valid possibilities in the list. The order of the list of possibilities is based on the standard disambiguating rules: a shift comes first, followed by reductions in the order in which the corresponding grammar rules were given.

9.1.2 The yy2lex Routine

In order to perform the lookahead, `"yyparse"` calls a routine called `"yy2lex"`. This is a user-written lexical analyzer, just as `"yylex"` is. It returns the same kind of token value that `"yylex"` does. However, `"yy2lex"` should not set the `"yylval"` value.

For some applications, `"yy2lex"` could even be the same routine as `"yylex"`. There are good reasons why the two should be separate, however. Suppose, for the sake of argument, that `"yyparse"` called `"yylex"` for all input. `"yyparse"` might call `"yylex"` for the next token, then immediately call `"yylex"` again for a lookahead. This second call could overwrite important values like `"yylval"` before they had been used, thereby confusing things greatly. For such reasons, `"yy2lex"` and `"yylex"` sometimes have to be different routines.

In practice, parsers rarely call `"yy2lex"` -- the routine is only needed in special circumstances. This means that `"yy2lex"` usually can be much simpler than `"yylex"`: `"yylex"` must be able to handle all possible input tokens, while `"yy2lex"` is only called in a few special cases. To find out the tokens that `"yy2lex"` must be able to handle, run your grammar through YAY and get a list of the states where the parser performs a Lookahead instead of a Shift or Reduce. These instances are the only ones where the parser will call `"yy2lex"`, so the associated tokens are the only ones that the routine has to handle.

The values obtained by `"yy2lex"` are thrown away once the parser has used them in its lookahead. What this means is that the parser doesn't remember what `"yy2lex"` returned, so it will call on `"yylex"` to re-read the token for normal parsing. This means one of two things:

(a) `"yy2lex"` can duplicate the processing that `"yylex"` does, then leave the pertinent information around for `"yylex"` to obtain. This means that `"yylex"` must have a way of knowing when `"yy2lex"` has already read the next token (e.g. `"yy2lex"` can set a flag).

(b) `"yy2lex"` can cheat and only do minimal processing. For example, `"yy2lex"` may only have to look at the first character of the next token to figure out what is coming next. `"yy2lex"` can then `"ungetc"` that character and return an appropriate token value. When `"yylex"` is called, it reads the full token in the usual manner.

The second approach is sufficient for most grammars. It simplifies both `"yylex"` and `"yy2lex"`.

9.1.3 Conflicts

With an additional level of lookahead, there is the potential for a staggering number of conflicts. For example, you may lookahead to a new state to resolve one conflict, only to find that the new state also has a lookahead to resolve a conflict.

If an LALR(2) parser is performing a lookahead and enters a state where the secondary lookahead token invokes another Lookahead action, the parser ruthlessly resolves the second Lookahead action by choosing the first possibility on the second Lookahead list. This is not very elegant, so you should avoid this double lookahead situation.

Conflicts of this nature are reported in the usual conflict table at the end of the State Description output.

9.2 Multiple Actions

A rule can have more than one action. For example, you might have

```
a : A1 {action1} A2 {action2} A3 {action3};
```

The non-terminal symbol `"a"` consists of symbols `A1`, `A2`, and `A3`. When `A1` is recognized, `"action1"` is invoked; when `A2` is recognized, `"action2"` is invoked; and when `A3` is recognized (and therefore the entire symbol `"a"`), `"action3"` is invoked. In this case,

`$1` `--` is the value of `A1`
`$2` `--` is the value of `$$` in `action1`
`$3` `--` is the value of `A2`
`$4` `--` is the value of `$$` in `action2`
`$5` `--` is the value of `A3`

If types are involved, multiple actions become more complicated. If `"action1"` mentions `$$`, there is no way for YAY to guess what type `$$` has, since it is not really associated with a token or non-terminal symbol. You must therefore state it explicitly by specifying the appropriate type name in angle brackets between the two `$` signs. If we had

```
%union {
    int intval;
    float realval;
}
```

you might say

```
$<realval>$
```

in place of `$$` in the action, to show that the result had type **`float`**. In the same way, if `"action2"` refers to `$2` (the result of `action1`), you might say

```
$<realval>2
```

To deal with multiple actions, YAY changes the form of the given grammar rule and creates grammar rules for dummy symbols. The dummy symbols have names made up of a `$` followed by the rule number. For example,

```
a : A1 {action1} A2 {action2} A3 {action3};
```

might be changed to the rules

```
$21 : /* null */ {action1} ;
$22 : /* null */ {action2} ;
a : A1 $21 A2 $22 A3 {action3};
```

These rules will be shown in the Rules Summary of the Parser Description report.

Note that this technique can introduce conflicts. For example, suppose you have the definitions

```
a : A1 {action1} A2 X;
b : A1 A2 Y;
```

These are changed to

```
$50 : /* null */ {action1};
a : A1 $50 A2 X;
b : A1 A2 Y;
```

The definitions of `"a"` and `"b"` give a shift/reduce conflict because the parser can't tell whether `A1` followed by `A2` has the null string for `$50` in the middle. It has to decide whether to Reduce `$50` or to Shift to a state diagrammed by

```
b : A1 A2.Y
```

This particular conflict could be resolved by using an LALR(2) parser instead of LALR(1). However, there are more complicated examples in which this is not possible.

There is another situation to watch for. Consider the grammar

```
a : A1 c A2 X ;
b : A1 c A2 Y ;
c : /* nothing */ {action} ;
```

You might think that this is equivalent to

```
a : A1 {action} A2 X;
b : A1 {action} A2 Y;
```

but it is not. The first will not produce a conflict, but the second one will. The second is converted into

```
a : A1 $50 A2 X;
b : A1 $51 A2 Y;
$50 : {action} ;
$51 : {action} ;
```

even when the action is the same for both `"a"` and `"b"`. If the parser finds an `A1` followed by a null string followed by `A2`, it doesn't know if it should interpret the null string as `$50` or `$51`; therefore, it gets a conflict. Obviously, it is better to use the first format, where the identical actions are associated with a separate rule.

9.3 Selection Preference

A *selection preference* can be added to a grammar rule to help resolve conflicts. The following input shows a simple example of how a selection preference can resolve conflicts between two rules.

```
a1 : a b ['+' '-']
     { /* Action 1 */ } ;
a2 : a b
     { /* Action 2 */ } ;
```

The selection preference is indicated by zero or more *tokens* inside square brackets. If the token that follows the `"b"` is one of the tokens inside the square brackets, the parser will use the grammar rule for `"a1"`. If the token that follows the `"b"` is not one of the given tokens, the parser will use the rule for `"a2"`. In this way, the conflict between the two rules is resolved -- the preference tells which one to select, depending on the value of the lookahead token.

Notice that a selection preference states that a rule *should* be used when the next token is one of the ones listed in the brackets and should *not* be used if it is not in the brackets.

The lookahead token is merely used to determine which rule to select. It is *not* part of the rule itself. For example, suppose you have

```
a1 : a b ['+' '-'] ;
a2 : a b ;
xx : a1 op expr ;
```

and suppose you have an `"a"`, a `"b"`, and + as the lookahead token. The + indicates that the `"a"` and `"b"` should be reduced to `"a1"`. The parser does this and finds that the `"a1"` is part of the `"xx"` rule. The + lookahead token will be associated with the `"op"` symbol in the `"xx"` rule. In other words, a selection preference does not "use up" an input token; it just looks at the token value to help resolve a conflict.

The `$end` symbol may be used inside square brackets to indicate end-of-file in the input being parsed. For example,

```
statement : line ['\n' $end]
```

shows that this rule is preferred if a `"line"` is followed by a new-line character or the end of the file.

The square brackets of a selection preference may contain no tokens, as in

```
x : y z [];
```

This says that the parser should never use this rule unless it cannot be avoided.

When there are many selection preferences in the same state, the parser must compare them to check for conflicts. To do this, YAY chooses the most likely rule (based on preference) and compares this to the other possible rules. In order to understand what we mean by "most likely", an example will help. Consider the grammar

```
a : b [ 'x' ]
  | b [ 'y' ]
  | b [ ]
  ;
```

YAY must consider what happens when a `"b"` symbol has been encountered. If the `"b"` is followed by an `'x'`, the most likely rule is

```
a : b [ 'x' ]
```

If the `"b"` is followed by a `'y'`, the most likely rule is

```
a : b [ 'y' ]
```

If the `"b"` is followed by any other token, the most likely rule is

```
a : b [ ]
```

Once the most likely rule has been chosen, it will be compared to the other rules in the set to make sure that there are no conflicts. (In a previous release, YAY would compare every pair of rules, without choosing a most likely one. In this case, the rules

```
a : b [ 'x' ]
a : b [ 'y' ]
```

would conflict with each other, because there was no way to choose between the two if the next token was not `'x'` or `'y'`. In this release, YAY will compare the most likely rule to all the others, but will not compare all the possible pairs.)

Selection preferences can also be stated using the construct

```
[^ T1 T2 T3 ...]
```

where the first character is a caret (`^`) and `T1`, `T2`, etc. are tokens. When this is put on the end of a rule, it indicates that the rule should be used if the lookahead token is *not* one of the listed tokens. For example,

```
a1 : a b
   { /* Action 1 */ } ;
a2 : a b [^ '+' '-']
   { /* Action 2 */ } ;
```

says that rule `"a2"` should be used if the token after the `"b"` is *not* `+` or `-`. If the token is `+` or `-`, `"a2"` should not be used (so `"a1"` will be).

The construct `[^]` is the reverse of `[]`. It is used to indicate a rule that should *always* be taken unless there is another rule that must be taken because of a selection preference.

Selection preference constructs can be put in the middle of rules as well as on the end. For example, we could write

```
expr : expr ['+' '-'] op expr
       { /* Action 1 */ }
     | expr op expr
       { /* Action 2 */ } ;
```

This states that if the first `"expr"` is followed by `+` or `-` you want to use the first rule; otherwise, you want to use the second. Note that the preference does not use up the `+` or `-` token: you still need a symbol (`"op"`) to represent such tokens.

Selection preferences that appear in the middle of a rule are implemented in the same way as multiple actions, using dummy rules. The above example would result in something like

```
$23 : ['+' '-'] ;
expr : expr $23 op expr
        { /* Action 1 */ }
      | expr op expr
        { /* Action 2 */ } ;
```

(where the `23` in `$23` is just a number we chose at random). The dummy rule that is created is a null string with the selection preference on the end. The first token for "`op`" will be the + or − that was the lookahead token in rule `$23`.

If a selection preference in the middle of a rule is immediately followed by an action, only one dummy rule is created to handle both the action and the preference.

In most cases, a selection preference counts as a `$N` symbol, but it has no associated value. For example, in

```
expr : expr ['+' '-'] op expr
```

we have

`$1 --` first "`expr`"
`$2 --` no value
`$3 --` "`op`"
`$4 --` second "`expr`"

If the preference is followed by an action, the preference and the action count as a single `$N` symbol whose value is equal to the `$$` value of the action. For example, in

```
expr : expr ['+' '-'] {action} op expr
```

we have

`$1 --` first "`expr`"
`$2 -- $$` of `action`
`$3 -- op`
`$4 --` second "`expr`"

The **`%prec`** construct is incompatible with rules that contain selection preferences, because the preference is all that is needed to resolve conflicts. For this reason, YAY issues an error message if a rule contains both a preference and the **`%prec`** construct.

Selection preferences can be used to resolve most conflicts. Indeed, there may be cases where the most practical course of action is to write a number of conflicting rules that contain selection preferences to resolve the conflicts, as in

```
expr : expr ['+' '-'] op expr
     | expr ['*' '/' '%'] op expr
     | expr ['&' '|'] op expr
               ...
```

`Note:` selection preferences of the form

```
[error]
[^ error]
```

are not useful. Selection preferences are implemented through (dummy) Reduce actions, but the parser's error-handling routines always look for Shift actions and ignore Reductions.

9.4 Negative Numbers in $N Constructs

YAY lets you use constructs like `$0`, `$-1`, `$-2`, and so on in recognition actions. These were once important, but the techniques for specifying multiple actions have made them obsolete. YAY only supports the constructs for compatibility with older grammars.

To understand what these constructs mean, it is important to think in terms of the state stack. Each `$N` construct is associated with a state on the stack; the value of `$N` is the value of the token or non-terminal symbol associated with the state at the time of a Reduce operation. (Recall that recognition actions are executed when the appropriate Reduce action takes place.)

`$1` is the value associated with the state that found the first component of the grammar rule; `$2` is the value associated with the second state, and so on.

`$0` is the value associated with the state that was on top of the stack before the first component of the grammar rule was found.

`$-1` is the value associated with the state before that, and so on. All of these states are still on the stack, and their value can be obtained in this way.

As an artificial example, suppose that a grammar has the rules

```
stmt : IF condition stmt
     | WHILE condition stmt
condition : /* something */
          { /* action */ }
```

The action associated with the condition can use the `$-1` construct to find out if the preceding token was `IF` or `WHILE`. (Of course, this assumes that the only items that can precede a condition are the `IF` and `WHILE` tokens.) There are occasionally times when this sort of information is needed.

9.5 Lists and Handling Null Strings

Grammars often define lists of items. There are two common ways to do this:

```
list : item
     | list item ;
```

or

```
list : /* null */
     | list item ;
```

The first definition means that every "list" has at least one item. The second allows zero-length lists.

Using the second definition is sometimes necessary or convenient, but it can lead to difficulties. To understand why, consider a grammar with

```
list1 : /* null */
      | list1 item1 ;
list2 : /* null */
      | list2 item2 ;
list  : list1
      | list2 ;
```

When the parser is in a position to look for a `"list"`, it automatically finds a null string, then gets a conflict because it can't decide if the null string is an instance of `"list1"` or `"list2"`. This problem is less likely to happen if you define

```
list1 : item1
      | list1 item1 ;
list2 : item2
      | list2 item2 ;
list  : /* null */
      | list1
      | list2
      ;
```

The parser can determine if it has a `"list1"` or a `"list2"` by seeing if the list starts with `"item1"` or `"item2"`.

A YAY-produced parser avoids infinite recursions that might result from matching the same null string over and over again. If the parser matches a null string in a particular state, goes through a few more states and shifts once more into the state where the null string was matched, it will not match the null string again. Without this behaviour, you get infinite recursions on null strings. However, the behaviour occasionally gets in the way if you *want* to match more than one null string in a row. For example, consider how you might write the grammar rules for types that may be used in a C cast operation, as in

```
char_ptr = (char *) float_ptr;
```

The rules for the parenthesized cast expression might be written as

```
cast : '(' basic_type opt_abstract ')' ;
opt_abstract : /* null */
             | abstract;
abstract : '(' abstract ')'
         | opt_abstract '[' ']'
         | opt_abstract '(' ')'
         | '*' opt_abstract
         ;
```

Consider what happens with a cast like

```
(int *[])
```

This will be interpreted as `*` followed by a null `"opt_abstract"` followed by a null `"opt_abstract"` followed by square brackets. However, the parser will *not* accept two null `"opt_abstracts"` in a row, and will take some other course of action.

To correct this problem, you must rewrite the grammar rules. Instead of using the `"opt_abstract"` rules, have rules with and without an `"abstract"`.

```
cast : '(' basic_type abstract ')' ;
abstract : /* null */
         | abstract '[' ']'
         | '[' ']'
         | abstract '(' ')'
         | '(' ')'
         | '*' abstract
         | '*'
         ;
```

9.6 Right vs. Left Recursion

Chapter 3 mentioned left and right recursion. For example, if a program consists of a number of statements separated by semicolons, we might define it with right recursion as

```
program : statement
        | statement ';' program ;
```

or with left recursion as

```
program : statement
        | program ';' statement ;
```

If you think about the way that the state stack works, you will see that the second way is much to be preferred. Consider, for example, the way something like

```
S1 ; S2 ; S3 ; S4
```

would be handled (where all the `Sn`'s are statements).

With right recursion, the parser would gather `"S1;"` and then go looking for a program. To gather this program, it would gather `S2`. It would then look at the lookahead symbol `;` and see that this program had the form

```
statement ';' program
```

The parser would then go ahead and gather the program after the semicolon. But after `S3`, it would find another semicolon, so would begin gathering yet another program. If you work the process through, you will find that the state stack grows to seven entries (one for each `Sn` and one for each `;`) before the first Reduce takes place.

On the other hand, if you have the left recursion

```
program : program ';' statement
```

and the same input, the parser will perform a Reduce as soon as it sees

```
S1 ; S2
```

This is Reduced to a single state corresponding to the non-terminal symbol `"program"`. The parser reads `";S3"` and Reduces

```
program ; S3
```

to `"program"` again. The process repeats for the last statement.

If you follow this through, the state stack never grows longer than three states, as compared to the seven that are required for the right recursive rule. With right recursion, no reduction takes place until the entire list of elements has been read; with left recursion, a reduction takes place as each new list element is encountered. Left recursion can therefore save a lot of stack space.

The choice of left or right recursion can also affect the order in which recognition actions are performed. Suppose `T` is a token. If we define

```
x : /* null */
  | y ',' x {a1} ;
y : T {a2} ;
```

then the input

```
T , T , T
```

would execute recognition actions in the order

```
{a2} {a2} {a2} {a1} {a1} {a1}
```

The `{a2}` actions are performed each time a `T` is Reduced to `"y"`. The `{a1}` actions don't happen until the entire list has been read, because right recursion reads the entire list before any Reduce actions take place.

On the other hand, if you define

```
x : /* null */
  | x ',' y {a1} ;
y : T {a2};
```

the recognition actions for the same input will take place in the order

```
{a2} {a1} {a2} {a1} {a2} {a1}
```

With left recursion, Reduce actions take place every time a new element is read in for the list.

This means that if you want the action order

```
{a2} {a2} {a2} {a1} {a1} {a1}
```

you must use right recursion even though it takes more stack space.

9.7 YYDEBUG

If you **`#define`** a symbol named `YYDEBUG` in the declarations section and set the variable `"yydebug"` to a non-zero value, your parser will print a good deal of debugging information as it parses input. In this description, we will describe the output you may see.

Every time `"yylex"` obtains a token, the parser prints

```
read T (VALUE)
```

`T` is the name of the token and `VALUE` is the numeric value. Thus if `"yylex"` has read an `IF` token, you might see

```
read IF (257)
```

Every time the parser enters a state, it will print out

```
state N (X), char (C)
```

where `N` is the state number as given in the State Description report, and `X` and `C` are other integers. `X` is another number for the state -- YAY actually renumbers the states and grammar rules after it generates the State Description report in order to improve the parser's efficiency, and `X` gives the state number after renumbering. `C` is the token type of the lookahead symbol if the symbol is a token. If the symbol is not a token, or if there is no lookahead symbol at the moment, `C` is `-1`. As an example,

```
state 6 (22), char (-1)
```

indicates that the parser has entered State 6 on the State Description Report (State 22 after renumbering) and that the current lookahead symbol is not a token.

Every time the parser performs a Shift action, it prints

```
shift N (X)
```

where `N` is the number of the state to which the parser is shifting and `X` is the number of the same state after renumbering.

Every time the parser performs a Reduce action, it prints

```
reduce N (X), pops M (Y)
```

This says that the parser has Reduced by grammar rule `N` (renumbered to `X`). After the reduction, the state on top of the state stack was State `M` (renumbered to `Y`).

9.8 Important Symbols

Debugging a parser produced by YAY can be a very difficult task, since the code is only partly produced by user input. The remaining code is standard software produced by YAY itself. The situation is aggravated by another problem: the state and rule numbers shown in the State Description report are not the same as the numbers used when the parser actually runs. In the interests of optimization, the parser sorts the states into a more convenient order. Thus the *internal* state number used by the program is usually not the same as the *external* state number known to the user.

To help those who are examining parser code using a symbolic debugger, here are a few of the important variables that the parser uses.

`yyval`

holds the value `$$` at the time of a reduction. This has the type `YYSTYPE`.

`yychar`

holds the most recent token value returned by `"yylex"`.

`yystate`

is the *internal* number of the current state.

`yyps`

points to the current top of the state stack. Thus `yyps[0]` is the internal number of the current state, `yyps[-1]` is the internal number of the previous state, and so on.

`yypv`

points to the top of the current value stack. The entries in this stack have the type `YYSTYPE`. When a Reduce operation executes a recognition action, this pointer is moved down the stack to the point where

```
yypv[1] = $1
yypv[2] = $2
    etc.
```

`yyi`

is the internal number of the rule being reduced by a Reduce action.

`yyrmap`

is an array present only when `YYDEBUG` is defined. It is used to convert internal rule numbers to external ones. For example,

```
yyrmap[yyi]
```

is the external number of the rule being reduced by a Reduce action.

`yysmap`

is an array present only when `YYDEBUG` is defined. It is used to convert internal state numbers to external ones. For example,

```
yysmap[yystate]
```

is the external number of the current state.

9.9 How YYERROR May Be Used

The `YYERROR` macro creates an artificial error condition. To show how this can be useful, suppose we have a line-by-line desk calculator that allows parenthesization of expressions and suppose we have a variable named `"depth"` that keeps track of how deeply parentheses are nested. Every time the parser finds an opening parenthesis, it adds 1 to `"depth"`. Every time it finds a closing parenthesis, it subtracts 1.

Consider how the following definitions will work.

```
expr : lp expr ')'
          {depth--;}
     | lp error
          {depth--;}
     ;
lp : '(' {depth++;};
```

If no error occurs, the `"depth"` variable is incremented and decremented correctly. If an error does occur, however, what happens? Your `"yyerror"` routine is called on to recover from the error in the middle of an expression. Often, it is more reasonable to postpone this recovery until you reach a point where you have a whole expression. Therefore, you might use the following alternate definition.

```
expr : lp error
           {depth--; YYERROR;}
     ;
line : error '\n' prompt line
       { $$ = $4; } ;
prompt : /* null token */
       {printf("Please re-enter line.\n");};
```

Now, what happens when the grammar is asked to parse a line like

```
1 + (( a +
```

When the end of the line is encountered, the parser recognizes an error has occurred. Going up the stack, the first state ready to handle the error is

```
expr : lp error ;
```

At this point, the parser will Reduce the input

```
( a +
```

into an `"expr"`. The Reduction executes the recognition action: it decrements `"depth"`, then signals that an error has taken place. The Error action begins popping the stack again. It will find the previous opening parenthesis, recognize another

```
lp error
```

construct and perform another reduction. The parenthesis count is again decremented, and another error condition generated.

This time, the grammar rule that deals with the error is the definition of `"line"`. An error message is issued and a new line is requested. In this way, the parser has worked its way back to error-handling code that can deal with the situation. Along the way, the parser correctly decremented the `"depth"` variable to account for the missing parentheses.

This method of dealing with errors decrements `"depth"` for every unbalanced opening parenthesis on the line. This corrects the `"depth"` count properly. Our first definition (without the `YYERROR` call) would only have decremented `"depth"` once.

This example is somewhat contrived, of course -- you could always just set `"depth"` to zero whenever you started a new line of input. The usefulness of the technique is more apparent in situations where you obtain memory with `"malloc"` whenever you get an opening delimiter and free the memory with `"free"` whenever you get a closing delimiter. In this case, it is obvious that you need to do precisely as many `"free"` operations as `"malloc"` operations, so you must raise the error condition for each unbalanced opening delimiter.

You might think that the symbol `"lp"` is unnecessary, and you could just define

```
expr : '(' {depth++;} expr ')' {depth--;}
     | '(' error {depth--;} ;
```

However, this would not work in general. There is no guarantee that the action

```
{depth++;}
```

would be executed in all cases, particularly if the token after the `'('` was one that could not start an expression.

As an interesting example of another way to use `YYERROR`, consider the following (taken from a parser for the Pascal programming language).

```
label_list : label_list ',' label
           | label
           | error
           | error [LABEL CONST VAR PROC FUNC BEGIN]
               {YYERROR; /* more code */};
```

This deals with errors in two different ways:

(a) If an error is followed by one of the tokens `LABEL`, `CONST`, etc. (representing the beginning of new declaration sections in Pascal), the input is reduced to a complete `"label_list"` input is Reduced to a complete `"label_list"` and an appropriate action is taken. This action uses `YYERROR` to raise the error condition, but only *after* the reduction has taken place.

(b) The other rule is used when the parser finds an error which is not followed by one of the listed tokens. This corresponds to an error in the middle of a label list and requires a different sort of handling. In this case, error-handling is allowed to take place immediately, without reduction, because there may be more `"label_list"` to come.

This kind of approach can be used to distinguish different kinds of errors that may take place in a particular situation.

9.10 The Default Action

The default action is the one that is taken when the parser finds a token which has no specified effect in the current state. Understanding how default actions work will help you understand what is going on when a YAY-produced parser encounters an error.

In a state diagram, the default action is marked with a `"."`. The default will always be a Reduce or Error action, chosen according to the following rules.

(a) If the state has no Shift actions and only one Reduce, the default will be the Reduce action.

(b) Apart from (a), an empty rule will never have Reduce as a default.

(c) If a state has more than one Reduce action, the parser examines the "popularity" of each Reduce. For example, if Reduction A is used with any of three different input tokens and Reduction B is used with only one input token, Reduction A is three times as "popular" as B. If one Reduce action is more than twice as popular as its closest contender (i.e. if it is taken on more than twice as many input tokens), and if that Reduce action is associated with a rule that contains reductions on at least *five* tokens, the popular Reduce action is made the default.

(d) In all other cases, the default action will be an Error action. For example, Error is chosen when a rule has more than one Reduce action, and there is no Reduce that is more than twice as popular as all the other contenders.

Note: YAY's predecessor YACC always chooses the most popular Reduce action as default (if there is one). It does not use the same requirements as (c) above. As a result of this difference YAY's parser tables are about 20% larger than YACC's, but a YAY-generated parser usually detects errors much earlier than a parser generated by YACC.

Because of the way default actions are treated, a YAY-produced parser will sometimes begin reducing when it encounters an error. Several reductions may take place before the error state is finally triggered. Thus your grammar may need some way to determine what actions have taken place between the time the erroneous token was read in and the time that the error was actually triggered.

9.11 Invalid Tokens

It is invalid to say either

```
%token X 0
    or
%token X 256
```

The value 0 is reserved for the end marker and 256 is reserved for `"error"`.

9.12 Dynamic Stack Allocation

The manifest `YYSSIZE` is used to determine the size of the state and value stacks used by `"yyparse"`. `YYSSIZE` gives the maximum number of elements that these stacks will be expected to hold; the size of each value element is dictated by the `YYSTYPE` type and the size of each state element is determined by YAY. The default value of `YYSSIZE` is 150.

By increasing the value of `YYSSIZE`, you can allow for grammars with a larger number of pending states.

If you **`#define`** `YYALLOC` in the declarations section, the state and value stacks used by `"yyparse"` will be allocated dynamically via `"malloc"` and freed before `"yyparse"` returns. What this effectively means is that `"yyparse"` makes itself re-entrant by saving a number of externals when it begins execution and restoring them upon completion. The externals involved are

```
yylval   yyval   yypvt
yynerrs  yychar  yyerrflag
```

If you `"longjmp"` out of `"yyparse"` (due to an action), the externals are *not* restored, and `"yyparse"` will not be re-entrant.

If you use `YYALLOC`, `"yyerror"` will be called if `"malloc"` fails to allocate space for the state and value stacks.

If you **`#define`** `YYALLOC` with a value greater than 10, the parser allocates the state and value stacks dynamically, beginning with a size of `YYSSIZE`. If this is not big enough to hold the two stacks, `"yyparse"` attempts to use `"realloc"` to grow the stacks by the amount given by `YYALLOC`. For example, if `YYALLOC` is 20, `"yyparse"` tries to grow the stacks to a size that will allow 20 additional elements (whenever `"yyparse"` needs more space for the stacks). `"yyerror"` is called if the call to `"realloc"` fails to allocate additional space.

If you set up your parser in such a way that it may grow the stacks, you must be careful not to take a pointer into a stack in one action and use that pointer inside a different action. The reason is that the stack may be grown using `"realloc"` in between the two actions. Since `"realloc"` may actually move the entire stack, the pointer will no longer be valid. Thus you should not create pointers with expressions like

```
&($1)
```

and expect those pointers to be valid in other actions. Within the code of a single action, however, such pointers **`will`** remain valid.

If you **`#define`** `YYSTATIC`, both the state and value stacks will be static. Otherwise, the state stack will be "auto" (allocated on the program stack) and the value stack will be static. Defining `YYALLOC` saves both stack space and static space; defining `YYSTATIC` saves stack space.

9.13 Synchronizing the Lookahead

If you **`#define`** `YYSYNC`, the parser will always have a lookahead token when it performs a shift or reduce action. If the symbol is not defined, the parser will only obtain a lookahead token if the value of the token is needed.

You would define `YYSYNC` in situations where you wanted to keep track of the line number on which various situations occurred (e.g. errors). If `"yylex"` *always* does a lookahead, you know that the line number of the token you are working with is the line number of the second last token read. If `"yylex"` sometimes does not do a lookahead, you don't know if the current line number in `"yylex"` is the line number for the current token or for a lookahead token.

You would avoid defining `YYSYNC` in situations where some actions do their own reading. For example, suppose that `/*` is a token that indicates the beginning of a comment. You could create an action for that token which reads all input up to a closing `*/`. With this kind of action, you would not want `"yylex"` to read a lookahead token, since that token would be the first token inside the comment, not the first token after the comment.

9.14 YYSHIFT

The generated parser program invokes a macro named `YYSHIFT` whenever the parser performs a shift action in response to a token. The macro is invoked without arguments. By default, `YYSHIFT` is defined to do nothing; however, users can redefine `YYSHIFT` if desired, in the token definition part of the YAY input.

As an example of how you might use `YYSHIFT`, consider a situation where input is freeform and may be split over many lines of text, including the insertion of blank lines in the input. You may want to keep a count of the number of lines you've read as you parse the input so that you can provide error messages like "Line 12: Syntax error".

When the parser finishes reading a line, it often has to read ahead to the next token before it can decide whether to reduce what it already has or keep on reading additional input. Since there may be blank lines in the input, this means that the parser may actually read ahead several lines before finding the next token. When it sees the next token, the parser may decide to reduce what it has seen already before shifting in response to the token.

While the reduction is happening, you want your line count to reflect the last line of input. You do not want the line count to reflect the line where the parser found the new token, because the parser isn't really using that token yet; it's working with older input. You only want to update the line count when the parser is ready to shift on the new token.

That's where `YYSHIFT` comes in. You can define the `YYSHIFT` macro to update the line count at the point when the token is actually used, not when it is first read.

`YYSHIFT` is *not* invoked if a token is discarded (because of error handling or some other reason). It is only invoked when the parser shifts on a token.

Appendix A
An Example

This appendix gives a simple example of YAY input. We have omitted the recognition actions of the grammar rules in order to keep the example as simple as possible.

The parser implements a simple string manipulation language. It has two types of objects: strings and variables. Strings can have a maximum of 100 characters and are enclosed in double quotes. Variable names are a maximum of eight characters long.

There are three operations:

(a) = assigns a string to a variable;

(b) `JOIN` or + concatenates strings or the contents of variables;

(c) `PRINT` outputs strings or the contents of variables.

Every statement is a single input line and blanks are used to separate tokens on the line. Multiple = assignments are permitted as in

```
A = B = "hello"
```

The keyword `PRINT` can only appear once on the line.

If a line does not contain an assignment, the value of the expression on the line is merely printed out. This means that it is valid to have a line that only contains a string or a variable name. The end of the input file marks the end of input.

Here are the declarations section of the YAY input and the `"yylex"` routine (both written in C).

```
%{   /* declarations */
#include <stdio.h>
#define MAXSTRING 100
char str[MAXSTRING];
#define MAXVAR 8
char var[MAXVAR];
%}

%token VARIABLE STRING
%nonassoc PRINT
%right '='
%left JOIN
```

```
%union {
     char *cstr;
}
%%

program : stat
        | program stat
        ;
stat : printexpr '\n'
     | expr '\n'
     ;
printexpr : PRINT expr ;
expr : VARIABLE '=' expr
     | JOIN expr expr
     | STRING
     | VARIABLE
     ;
%%

int yylex()
{
    char c, *stringet(), *wordget();
    extern YYSTYPE yylval;
    while ( (c=getchar()) == ' ' )
           /* skip blanks */;
    switch(c) {
      case '\n':    /* end of line */
          return('\n');
      case EOF:     /* end of file */
          return(0);
      case '+':     /* same as JOIN */
          return(JOIN);
      case '"':     /* start of string */
          yylval.cstr = stringet();
          return(STRING);
      default:      /* keyword or variable */
          yylval.cstr = wordget(c);
          if ( !strcmp("JOIN",yylval.cstr) )
              return(JOIN);
          else if ( !strcmp("PRINT",yylval.cstr) )
              return(PRINT);
          else return(VARIABLE);
    }
}

char *stringet()
{
    extern char str[];
    int i;
```

```
    for ( i=0; i < MAXSTRING; i++ ) {
        str[i] = getchar();
        if ( (str[i]=='"') || (str[i]=='\n') )
            break;
    }
    str[i] = '\0';  /* mark end of string */
    return(str);
}

char *wordget(char c)
{
    extern char var[];
    int i;
    var[0] = c;  /* first letter obtained already */
    for ( i=1; i < MAXVAR; i++ ) {
        var[i] = getchar();
        if ( (var[i]==' ') || (var[i]=='\n') )
            break;
    }
    var[i] = '\0';
    return(var);
}

void yyerror(char *s)
{
    fprintf(stderr,s);
}
```

Note that the lexical analyzer always flags the keywords `JOIN` and `PRINT` as operations. This means that you cannot use `JOIN` or `PRINT` as variable names. In fact, this is a general limitation of LALR(1) parsers. It is very difficult to have names that are keywords in some contexts and variable names in others, without a great deal of fiddling. Therefore, we recommend that you resign yourself to having *reserved* keywords, i.e. keywords forbidden for use as variable names.

The above example omitted recognition actions in the Rules Section to make things easier to read.

Appendix B
YAY vs. YACC

YAY differs from its predecessor YACC in a number of respects. These are summarized below.

(a) YACC does not support the following.

-- all the LALR(2) features described in 9.1
-- selection preferences, as described in 9.3
-- the use of `YYSTATIC`, `YYSYNC`, or `YYALLOC` (9.12)

(b) If a state has several Reduce action, YACC always chooses the most popular of these as the default action. YAY's rules for choosing a default are given in 9.10. In general, YAY's approach will detect errors more quickly (sooner after they actually occur).

(c) The two programs handle `YYERROR` differently. YAY always reduces and pops the current state off the stack before generating the artificial error condition. YACC doesn't do this. With YACC, you cannot use `YYERROR` inside a rule with "error" in it; YAY, however, will resolve the current rule and then trigger error handling.

(d) Output from the two programs differs slightly.

As a rule of thumb, YAY accepts any YACC grammar and creates a parser that behaves the same for correct input. However, the YAY version usually detects errors earlier than the YACC parser, and has more Error actions in the state tables.

Appendix C
The YAY Command Line

Syntax:

```
yay sourcefile Parser=outfile [option]*

(+|-)LR2 (-)             (+|-)Verbose (-)
(+|-)Warnings (+)        Description=file
Header=file              INSTallation=file
Language=C|C++ (C)       Parser=file
```

Examples:

```
yay cgram.y parse=myparse.c
c myparse.c

yay ccgram.y lang=c++ pars=myparse.cpp
```

Options:

`sourcefile`

is a file containing YAY input.

`Language=C`

produces parsing tables in the C programming language. This is the default.

`Language=C++`

produces parsing tables in the C++ programming language. Note that YAY only produces the tables; the routines that use the tables to parse input are predefined.

`+LR2`

says that the YAY input describes an LALR(2) grammar. Without `+LR2`, YAY assumes that the grammar is LALR(1). The manual describes modifications that need to be made for LALR(2) grammars.

`Description=file`

translates the parsing tables into a format that humans can read, and writes this output into the given file.

`Header=file`

writes token definitions and other information necessary for separate compilation, to the named file.

`INSTallation=file`

tells YAY where to find the installation file. The installation file tells where various software components have been installed. For more information, see the section on *Installation Files* below.

If you do not specify an `INSTallation=` option on the command line, YAY checks for an environment variable named `YAY_INST` and uses its value as the name of the installation file. If this environment variable does not exist, YAY uses the default installation file.

`Parser=file`

writes the resulting source code for the parser into the named file. If this option is omitted, YAY just checks the syntax of your input.

`+Verbose`

produces verbose output -- everything that can be flagged is flagged.

`-Warnings`

suppresses a number of warning messages that YAY normally issues.

Description:

YAY converts your context-free grammar into a C or C++ program that is written to the file specified by the `Parser=` option.

If you use the `Description=` option, YAY writes a full description of the grammar to the specified file. YAY only displays a brief message on the standard output, summarizing conflicts (and other information if you specify `+Verbose`). On the other hand, if you do not use the `Description=` option, YAY writes more information to standard output, including descriptions of the states where conflicts occur. In this case, YAY actually provides additional information to help you identify the source of the conflicts; if you ask for a description file, YAY outputs less information when reporting the conflicts because it assumes you can track down additional information by looking at the description file. For this reason, you can sometimes get a quicker idea of what has gone wrong if you do not ask for a description file.

C++ Parsers

In general, you only need to use `Language=C++` if you intend `YYSTYPE` to contain a C++ object with constructors. If you intend to compile the parser with C++ but the `%union` statement does not have any elements that need constructors, it's best to use `Language=C` to get more efficient C code.

If `YYSTYPE` does contain elements that need constructors, you need to define an appropriate constructor-like function for the `YYSTYPE` type. This function should have the prototype

```
void name(YYSTYPE *p)
```

where `"name"` can be any valid name. In the declarations section of the grammar, you must then add the statement

```
#define YYSTYPE_INIT name
```

where `"name"` is the name of the constructor-like function.

With `Language=C++`, the `%union` statement generates a structure type rather than a union, since C++ does not allow objects with constructors to belong to unions.

In many cases, the same grammar may be processed with either `Language=C` or `Language=C++`.

Installation Files:

An installation file specifies the pathnames for software and data files used by YAY. Installation files are text files made up of comment lines and option lines.

Comment lines:

Any line whose first non-blank character is `#` will be taken as a comment. Blank lines are also considered comments.

Option lines:

Option lines have the format

```
Keyword=pathname
```

In this documentation, keywords are written with some letters in upper case and some in lower case. You may abbreviate keywords by omitting any or all of the letters shown in lower case. The remaining letters may be entered in either upper or lower case; the documentation simply uses upper case to show which characters may not be omitted.

In this version of YAY, there is only one valid option line:

```
Library=pathname
```

The pathname should be the directory containing the YAY parser template files (e.g. `yyparse.c`).

Notes:

If you define `YYALLOC`, the parser allocates its state and value stacks dynamically via malloc and free. This shrinks your parser and helps prevent stack overflows.