symbol is the grammar's start symbol. If we use a grammar for C, the entire input must be a `sequence of definitions and declarations'. If not, the parser reports a syntax error.

The parser tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.

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

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

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

There is one other alternative: the table can say that the look-ahead token is erroneous in the current state. This causes error processing to begin.

A Yacc/Bison Example

The following is a Yacc/Bison input file which defines a reverse polish notation calculator. The file created by Yacc/Bison simulates the calculator. The details of the example are explained in later sections.

```
/* Reverse polish notation calculator. */
%{
#define YYSTYPE double
#include <math.h>
%}
%token NUM
%% /* Grammar rules and actions follow */
input : /* empty */
   | input line
;
line : '\n'
   | exp '\n'  { printf ("\t%.10g\n", $1); }
;
exp : NUM             { $$ = $1;         }
   | exp exp '+'     { $$ = $1 + $2;    }
   | exp exp '-'     { $$ = $1 - $2;    }
   | exp exp '*'     { $$ = $1 * $2;    }
   | exp exp '/'     { $$ = $1 / $2;    }
   /* Exponentiation */
   | exp exp '^'     { $$ = pow ($1, $2); }
   /* Unary minus    */
  | exp 'n'        { $$ = -$1;        }
;
%%
/* Lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the ASCII
   character read if not a number.  Skips all blanks
   and tabs, returns 0 for EOF. */
#include <ctype.h>
yylex ()
{ int c;
  /* skip white space  */
  while ((c = getchar ()) == ' ' || c == '\t')
    ;
  /* process numbers   */
  if (c == '.' || isdigit (c))
    {
      ungetc (c, stdin);
      scanf ("%lf", &yylval);
      return NUM;
    }
  /* return end-of-file  */
  if (c == EOF)
    return 0;
  /* return single chars */
  return c;
}
main ()     /* The ``Main'' function to make this stand-alone  */
{
  yyparse ();
}
#include <stdio.h>
yyerror (s)  /* Called by yyparse on error */
     char *s;
{
  printf ("%s\n", s);
}
```

The Yacc/Bison Input File

Yacc/Bison takes as input a context-free grammar specification and produces a C-language function that recognizes correct instances of the grammar. The input file for the Yacc/Bison utility is a *Yacc/Bison grammar file*. The Yacc/Bison grammar input file conventionally has a name ending in `.y`.

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

```
%{
C declarations
%}
Yacc/Bison declarations
%%
Grammar rules
%%
Additional C code
```

Comments enclosed in `/* ... */` may appear in any of the sections. The `%%`, `%{` and `%}` are punctuation that appears in every Yacc/Bison grammar file to separate the sections.

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

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

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

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

The Declarations Section

The C Declarations Section

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