6. yacc

Introduction

yacc provides a general tool for imposing structure on the input to a computer program. The **yacc** user prepares a specification that includes:

- a set of rules to describe the elements of the input
- code to be invoked when a rule is recognized
- either a definition or declaration of a low-level routine to examine the input

yacc then turns the specification into a C language function that examines the input stream. This function, called a parser, works by calling the low-level input scanner. The low-level input scanner, called a lexical analyzer, picks up items from the input stream. The selected items are known as tokens. Tokens are compared to the input construct rules, called grammar rules. When one of the rules is recognized, the user code supplied for this rule (an action), is invoked. Actions are fragments of C language code. They can return values and make use of values returned by other actions.

The heart of the **yacc** specification is the collection of grammar rules. Each rule describes a construct and gives it a name. For example, one grammar rule might be:

```
date : month_name day ',' year ;
```

where **date**, **month_name**, **day**, and **year** represent constructs of interest; presumably, **month_name**, **day**, and **year** are defined in greater detail elsewhere. In the example, the comma is enclosed in single quotes. This means that the comma is to appear literally in the input. The colon and semicolon merely serve as punctuation in the rule and have no significance in evaluating the input. With proper definitions, the input:

```
July 4, 1776
```

might be matched by the rule.

The lexical analyzer is an important part of the parsing function. This user-supplied routine reads the input stream, recognizes the lower-level constructs, and communicates these as tokens to the parser. The lexical analyzer recognizes constructs of the input stream as terminal symbols; the parser recognizes constructs as nonterminal symbols. To avoid confusion, we will refer to terminal symbols as tokens.

There is considerable leeway in deciding whether to recognize constructs using the lexical analyzer or grammar rules. For example, the rules:

```
month_name : 'J' 'a' 'n'  ;
month_name : 'F' 'e' 'b'  ;

             ...

month_name : 'D' 'e' 'c'  ;
```

might be used in the above example. While the lexical analyzer only needs to recognize individual letters, such low-level rules tend to waste time and space and may complicate the specification beyond the ability of **yacc** to deal with it. Usually, the lexical analyzer recognizes the month names and returns an indication that a **month_name** is seen. In this case, **month_name** is a token and the detailed rules are not needed.

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. It is relatively easy to add to the above example the rule:

```
date  :  month '/' day '/' year   ;
```

allowing the expression:

```
7/4/1776
```

as a synonym for:

```
July 4, 1776
```

on input. Usually, this new rule could be slipped into a working system with minimal effort and with little danger of disrupting existing input.

The input being read may not conform to the specifications. Input errors are detected as early as is theoretically possible with a left-to-right scan. Not only is the chance of reading and computing with bad input data substantially reduced, but the bad data usually can be found quickly. Error handling, provided as part of the input specifications, permits the reentry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, **yacc** fails to produce a parser when given a set of specifications. The specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to **yacc**. The former cases represent design errors; the latter cases often can be corrected by making the lexical

analyzer more powerful or by rewriting some of the grammar rules. While **yacc** cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructs that are difficult for **yacc** to handle are also frequently difficult for human beings to handle. Some users have reported that the discipline of formulating valid **yacc** specifications for their input revealed errors of conception or design early in the program development.

The remainder of this chapter describes the following subjects:

- basic process of preparing a **yacc** specification
- parser operation
- handling ambiguities
- handling operator precedences in arithmetic expressions
- error detection and recovery
- the operating environment and special features of the parsers **yacc** produces
- suggestions to improve the style and efficiency of the specifications
- advanced topics

In addition, there are two examples and a summary of the **yacc** input syntax.

Basic Specifications

Names refer to either tokens or nonterminal symbols. **yacc** requires token names to be declared as such. While the lexical analyzer may be included as part of the specification file, it is perhaps more in keeping with modular design to maintain it as a separate file. Like the lexical analyzer, other subroutines may be included as well. Every specification file theoretically consists of three sections: the declarations, (grammar) rules, and subroutines. The sections are separated by double percent signs, **%%** (the percent sign is generally used in **yacc** specifications as an escape character).

A full specification file has the general form:

declarations
%%
rules
%%
subroutines

when all sections are used. The *declarations* and *subroutines* sections are optional.

The smallest legal **yacc** specification is:

```
%%
rules
```

Blanks, tabs, and newlines may not appear in names or multicharacter reserved symbols. They are ignored elsewhere. Comments may appear wherever a name is legal. They are enclosed in /* ... */, as in the C language.

The rules section is made up of one or more grammar rules. A grammar rule has the form:

```
A  :  BODY  ;
```

where **A** represents a nonterminal symbol and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are **yacc** punctuation.

Names may be of any length. They may be made up of letters, dots, underscores, and digits, although a digit may not be the first character of a name. Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes, '. As in the C language, the backslash, \, is an escape character within literals, and all the C language escapes are recognized. Thus, the escape characters:

'**\n**'	newline
'**\r**'	return
'\''	single quote (')
'\\'	backslash (\ )
'**\t**'	tab
'**\b**'	backspace
'**\f**'	form feed
'*\xxx*'	*xxx* in octal notation

are understood by **yacc**. For technical reasons, the NULL character (\0 or 0) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar, |, can be used to avoid rewriting the left-hand side. In addition, the semicolon at the end of a rule is dropped before a vertical bar. Thus, the grammar rules:

```
A  :  B  C  D  ;
A  :  E  F  ;
A  :  G  ;
```

can be given to **yacc** as:

```
A    :  B  C  D
     |  E  F
     |  G
     ;
```

by using the vertical bar. It is not necessary for all grammar rules with the same left side to appear together in the grammar rules section, although this makes the input more readable and easier to change.

If a nonterminal symbol matches the empty string, this condition can be indicated by:

```
epsilon :    ;
```

The blank space following the colon is understood by **yacc** to be a nonterminal symbol named **epsilon**.

Names representing tokens must be declared. This is most simply done by writing:

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

in the declarations section. Every name not defined in the declarations section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the start symbol has particular importance. By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section. It is possible and desirable to declare the start symbol explicitly in the declarations section using the **%start** keyword:

```
%start   symbol
```

The end of the input to the parser is signaled by a special token, called the end-marker. The end-marker is represented by either a zero or a negative number. If the tokens up to but not including the end-marker form a construct that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input. If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate. Usually the end-marker represents some reasonably obvious I/O status, such as end of file or end of record.

Actions

With each grammar rule, the user may associate actions to be performed when the rule is recognized. Actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.

An action is an arbitrary C language statement. As such, it can do input and output, call subroutines, and alter arrays and variables. An action is specified by one or more statements enclosed in curly braces, { and }. For example:

```
A   :  '('  B  ')'
    {
       hello( 1, "abc" );
    }
```


and:

```
XXX   :  YYY  ZZZ
      {
         (void) printf("a message\n");
         flag = 25;
      }
```

are grammar rules with actions.

The dollar sign symbol, **$**, is used to facilitate communication between the actions and the parser; the pseudo-variable **$$** represents the value returned by the complete action. For example, the action:

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

returns the value 1; in fact, that's all it does.

To obtain the values returned by previous actions and by the lexical analyzer, the action may use the pseudo-variables **$1, $2, ...** $*n*. These refer to the values returned by components 1 through *n* of the right side of a rule, with the components being numbered from left to right. If the rule is:

```
A   :  B  C  D   ;
```

then **$2** has the value returned by **C**, and **$3** the value returned by **D**.

The following rule provides a common example:

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

One would expect the value returned by this rule to be the value of the *expr* within the parentheses.

Since the first component of the action is the literal left parenthesis, the desired logical result can be indicated by:

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

By default, the value of a rule is the value of the first element in it (**$1**). Thus, grammar rules of the form:

```
A    :    B     ;
```

frequently need not have an explicit action. In previous examples, all the actions came at the end of rules. Sometimes, it is desirable to get control before a rule is fully parsed. **yacc** permits an action to be written in the middle of a rule as well as at the end. This action is assumed to return a value accessible through the usual **$** mechanism by the actions to the right of it. In turn, it may access the values returned by the symbols to its left. Thus, in the rule below the effect is to set **x** to 1 and **y** to the value returned by **C**:


```
A    :    B
                {
                   $$ = 1;
                }
                C
     {
                x = $2;
                y = $3;
     }
     ;
```

yacc handles actions that do not terminate a rule by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string. The interior action is the action triggered by recognizing this added rule.

yacc treats the above example as if it had been written:

```
$ACT    :   /* empty */
        {
            $$ = 1;
        }
        ;

A       :   B  $ACT  C
        {
            x = $2;
            y = $3;
        }
        ;
```


where **$ACT** is an empty action.

In many applications, output is not done directly by the actions. A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated. Parse trees are particularly easy to construct given routines to build and maintain the tree structure desired. For example, suppose there is a C function node written so that the call:

```
node( L, n1, n2 )
```

creates a node with label **L** and descendants **n1** and **n2** and returns the index of the newly created node. Then you can build a parse tree by supplying actions such as:

```
expr    :   expr  '+'  expr
        {
            $$ = node( '+', $1, $3 );
        }
```

in the specification.

You may define other variables to be used by the actions. Declarations and definitions can appear in the declarations section enclosed in the marks **%{** and **%}**. These declarations and definitions have global scope, so they are known to the action statements and can be made known to the lexical analyzer. For example, the declaration:

```
%{   int variable = 0;   %}
```

could be placed in the declarations section, making **variable** accessible to all the

actions. Users should avoid names beginning with **yy** because the **yacc** parser uses only such names. In the examples shown thus far, all the values are integers. A discussion of values of other types is in the section "Advanced Topics."

Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser. The lexical analyzer is an integer-valued function called **yylex**. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable **yylval**.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by **yacc** or the user. In either case, the **#define** mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name DIGIT has been defined in the declarations section of the **yacc** specification file. To return the appropriate token, the relevant portion of the lexical analyzer might look like:


```
int yylex()
{
      extern int yylval;
      int c;
      ...
      c = getchar();
      ...
      switch (c)
      {
         ...
         case '0':
         case '1':
         ...
         case '9':
         yylval = c - '0';
         return (DIGIT);
         ...
      }
      ...
}
```

The intent is to return a token number of DIGIT and a value equal to the numerical value of the digit. If the lexical analyzer code is placed in the subroutines section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names **if** or **while** will almost certainly cause severe difficulties when the lexical analyzer is compiled. The token name **error** is reserved for error handling and should not be used naively.

In the default situation, token numbers are chosen by **yacc**. The default token number for a literal character is the numerical value of the character in the local character set. Other names are assigned token numbers starting at 257. If the **yacc** command is invoked with the **–d** option, a file called **y.tab.h** is generated. The **y.tab.h** file contains **#define** statements for the tokens.

If you prefer to assign the token numbers yourself, the first appearance of the token name or literal in the declarations section must be followed immediately by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined this way are assigned default definitions by **yacc**. The potential for duplication exists here; make sure that all token numbers are distinct.

For historical reasons, the end-marker must have token number 0 or a negative value. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token on reaching the end of their input.

A useful tool for constructing lexical analyzers is the **lex** utility. Lexical analyzers produced by **lex** are designed to work in close harmony with **yacc** parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules. **lex** can be easily used to produce complicated lexical analyzers, but there remain some languages (such as FORTRAN), which do not fit any theoretical framework and whose lexical analyzers must be crafted by hand.

Parser Operation

yacc turns the specification file into a C language procedure, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex and will not be discussed here. The parser itself, though, is relatively simple and understanding its usage will make treatment of error recovery and ambiguities easier.

The parser produced by **yacc** consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the look-ahead token). The current state is always the one on the top of the stack. The states of the finite state machine are given small integer labels. Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read.

The machine has only four actions available—**shift**, **reduce**, **accept**, and **error**. A step of the parser is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.
2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off the stack and in the look-ahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action occurs, there is always a look-ahead token. For example, in state 56 there may be an action such as:

```
IF    shift 34
```

which says: In state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack). The look-ahead token is cleared.

The **reduce** action keeps the stack from growing without bounds. **reduce** actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side. It may be necessary to consult the look-ahead token to decide whether to **reduce** (usually it is not necessary). In fact, the default action (represented by a dot) is often a **reduce** action.

reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. The action:

refers to grammar rule 18, while the action:

```
IF    shift 34
```

refers to state 34.

Suppose the rule:

```
A    :    x  y  z    ;
```

is being reduced. The **reduce** action depends on the left-hand symbol (A in this

case) and the number of symbols on the right-hand side (three in this case). To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing x, y, and z and no longer serve any useful purpose. Popping these states uncovers the state the parser was in before beginning to process the rule. Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of **A**. A new state is obtained and pushed onto the stack, and parsing continues. There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a **goto** action. In particular, the look-ahead token is cleared by a shift but is not affected by a **goto**. In any case, the uncovered state contains an entry such as:

```
A    goto 20
```

causing state 20 to be pushed onto the stack and become the current state.

In effect, the **reduce** action turns back the clock in the parse popping the states off the stack and goes back to the state where the right-hand side of the rule was first seen. The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off the stacks. The uncovered state is the current state.

The **reduce** action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions. When a **shift** takes place, the external variable **yylval** is copied onto the value stack. After the return from the user code, the reduction is carried out. When the **goto** action is done, the external variable **yyval** is copied onto the value stack. The pseudo-variables **$1**, **$2**, etc., refer to the value stack.

The other two parser actions are conceptually much simpler. The **accept** action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker and shows that the parser has successfully done its job. The **error** action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input. The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

Consider:

```
%token DING DONG DELL
%%
rhyme : sound place
      ;
sound : DING DONG
      ;
place : DELL
      ;
```

as a **yacc** specification.

Invoking **yacc** with the **–v** option produces a file called **y.output** with a human-readable description of the parser. The **y.output** file corresponding to the above grammar (with some statistics stripped off the end) follows.

```
state 0
        $accept  :  _rhyme  $end

        DING  shift 3
        .  error

        rhyme  goto 1
        sound  goto 2

state 1
        $accept  :   rhyme_$end

        $end  accept
        .  error

state 2
        rhyme  :   sound_place

        DELL  shift 5
        .  error

        place   goto 4

state 3
        sound   :   DING_DONG

        DONG  shift 6
        .  error

state 4
        rhyme  :   sound  place_    (1)

        .   reduce  1

state 5
        place  :   DELL_    (3)

        .   reduce  3

state 6
        sound   :   DING  DONG_    (2)

        .   reduce  2
```

The actions for each state are specified and there is a description of the parsing rules being processed in each state. The _ character is used to indicate what has been seen and what is yet to come in each rule.

The following input can be used to track the operations of the parser:

```
DING  DONG  DELL
```

Initially, the current state is state 0. The parser needs to refer to the input to decide between the actions available in state 0, so the first token, DING, is read and becomes the look-ahead token. The action in state 0 on DING is **shift 3**, so state 3 is pushed onto the stack and the look-ahead token is cleared. State 3 becomes the current state. The next token, DONG, is read and becomes the look-ahead token. The action in state 3 on the token DONG is **shift 6**, so state 6 is pushed onto the stack and the look-ahead is cleared. The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by:

```
sound  :  DING  DONG
```

which is rule 2. Two states, 6 and 3, are popped off the stack uncovering state 0. Consulting the description of state 0 (looking for a **goto** on **sound**), produces:

```
sound   goto 2
```

State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token, DELL, must be read. The action is **shift 5**, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared. In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered. The **goto** in state 2 on **place** (the left side of rule 3) is state 4. Now, the stack contains 0, 2, and 4. In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again. In state 0, there is a **goto** on **rhyme** causing the parser to enter state 1. In state 1, the input is read and the end-marker is obtained, indicated by **$end** in the **y.output** file. The action in state 1 (when the end-marker is seen) successfully ends the parse.

It is worthwhile to consider how the parser works when confronted with such incorrect strings as DING DONG DONG, DING DONG, DING DONG DELL DELL, etc. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways. For example, the grammar rule:

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

is a natural way of expressing the fact that one way to form an arithmetic

expression is to put two other expressions together with a minus sign between them. Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is:

```
expr - expr - expr
```

the rule allows this input to be structured either as:

```
( expr - expr ) - expr
```

or as:

```
expr - ( expr - expr )
```

(The first is called left association, the second right association.)

yacc detects such ambiguities when it is attempting to build the parser. Given the input:

```
expr - expr - expr
```

consider the problem that confronts the parser. When the parser has read the second *expr*, the input seen:

```
expr - expr
```

matches the right side of the grammar rule above. The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to **expr** (the left side of the rule). The parser would then read the final part of the input:

```
- expr
```

and again reduce. The effect of this is to take the left associative interpretation.

Alternatively, if the parser sees:

```
expr - expr
```

it could defer the immediate application of the rule and continue reading the input until it sees:

```
expr - expr - expr
```

The parser could then apply the rule to the rightmost three symbols, reducing them to *expr*, which leaves:

```
expr - expr
```

Now the rule can be reduced once more. The effect is to take the right associative interpretation. Thus, having read:

```
expr - expr
```

the parser can do one of two legal things, a shift or a reduction. It has no way of

deciding between them. This is called a **shift-reduce** conflict. It may also happen that the parser has a choice of two legal reductions. This is called a **reduce-reduce** conflict. Note that there are never any **shift-shift** conflicts.

When there are **shift-reduce** or **reduce-reduce** conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice. A rule describing the choice to make in a given situation is called a disambiguating rule.

In situations calling for a default choice, **yacc** invokes two default disambiguating rules:

1. In a **shift-reduce** conflict, the default is to do the shift.
2. In a **reduce-reduce** conflict, the default is to reduce by the earlier grammar rule (in the **yacc** specification).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice. Rule 2 gives the user crude control over the behavior of the parser in this situation, but **reduce-reduce** conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic or because the grammar rules (while consistent) require a more complex parser than **yacc** can construct. The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser. For this reason, **yacc** always reports the number of **shift-reduce** and **reduce-reduce** conflicts resolved by Rule 1 and Rule 2.

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts. Most previous parser generators have considered conflicts to be fatal errors. Our experience suggests that this rewriting is somewhat unnatural and produces slower parsers. Thus, **yacc** will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider:

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

which is a fragment from a programming language involving an **if-then-else** statement. In these rules, IF and ELSE are tokens, *cond* is a nonterminal symbol describing conditional (logical) expressions, and *stat* is a nonterminal symbol describing statements. The first rule will be called the simple **if** rule and the second the **if-else** rule.

These two rules form an ambiguous construction because input of the form:

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be structured according to these rules in two ways, either as:

```
IF  ( C1 )
{
      IF  ( C2 )
            S1
}
ELSE
      S2
```

or as:

```
IF  ( C1 )
{
      IF  ( C2 )
            S1
      ELSE
            S2
}
```

The second interpretation is the one given in most programming languages having this construct; each ELSE is associated with the last preceding un-ELSE'd IF. In this example, consider the case where the parser has seen:

```
IF  (  C1  )  IF  (  C2  )  S1
```

and is looking at the ELSE. It can immediately reduce by the simple **if** rule to get:

```
IF  (  C1  )  stat
```

and then read the remaining input:

```
ELSE  S2
```

and reduce:

```
IF  (  C1  )  stat  ELSE  S2
```

by the **if-else** rule. This leads to the first of the above groupings of the input.

On the other hand, the ELSE may be shifted, S2 read, and then the right-hand portion of:

```
IF  (  C1  )  IF  (  C2  )  S1  ELSE  S2
```

can be reduced by the if-else rule to get:

```
IF  (  C1  )  stat
```

which can be reduced by the simple **if** rule. This leads to the second of the above input groupings, which is usually the desired one.

Once again, the parser can do two valid things—there is a **shift-reduce** conflict. Here, the application of disambiguating rule 1 tells the parser to shift, which leads to the desired grouping.

This **shift-reduce** conflict arises only when there is a particular current input symbol, ELSE, and particular inputs, such as:

```
IF  (  C1  )  IF  (  C2  )  S1
```

have already been seen. In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs. The previously read inputs are characterized by the state of the parser.

The conflict messages of **yacc** are best understood by examining the verbose (**–v**) option output file. For example, the output corresponding to the above conflict state might be:

```
23: shift-reduce conflict (shift 45, reduce 18) on ELSE

state 23

   stat  :  IF  (  cond  )  stat_          (18)
   stat  :  IF  (  cond  )  stat_ELSE  stat

   ELSE     shift 45
            reduce 18
```

where the first line describes the conflict—giving the state and the input symbol. The ordinary state description gives the grammar rules active in the state and the parser actions. Recall that the underline marks the portion of the grammar rules, which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to:

```
IF  (  cond  )  stat
```

and the two grammar rules shown are active at this time. The parser can do two possible things: remain in state 23 or shift into state 45.

If the input symbol is ELSE, it is possible to shift into state 45. State 45 will have, as part of its description, the line:

```
stat  :  IF  (  cond  )  stat  ELSE_stat
```

because the ELSE will have been shifted in this state. In state 23, the alternative action (describing a dot, .), is to be done if the input symbol is not mentioned explicitly in the actions. Here, if the input symbol is not ELSE, the parser reduces to:

```
stat  :  IF  '('  cond  ')'  stat
```

by grammar rule 18.

Once again, notice that the numbers following shift commands refer to other states, while the numbers following reduce commands refer to grammar rule numbers. In the **y.output** file, the rule numbers are printed in parentheses after those rules, which can be reduced. In most states, there is a reduce action possible in the state and this is the default command. The user who encounters unexpected **shift-reduce** conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

Precedence

One common situation exists where the above rules for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions. Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity. It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars. The basic notion is to write grammar rules of the forms:

```
expr  :  expr  OP  expr
```

and:

```
expr  :  UNARY  expr
```

for all binary and unary operators desired. This creates an ambiguous grammar with many parsing conflicts. As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators. This information allows **yacc** to resolve the parsing conflicts according to these rules and construct a parser that realizes the desired precedences and associativities.

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with a **yacc** keyword: **%left**, **%right**, or **%nonassoc**, followed by a list of tokens. All the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. Thus, the segment

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

describes the precedence and associativity of the four arithmetic operators. Plus and minus are left associative and have lower precedence than star and slash, which are also left associative. The keyword **%right** is used to describe right associative operators, and the keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. Thus an expression like:

```
A  .LT.  B  .LT.  C
```

is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in **yacc**. As an example of the behavior of these declarations, the description:


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

%%

expr    :   expr  '='  expr
        |   expr  '+'  expr
        |   expr  '-'  expr
        |   expr  '*'  expr
        |   expr  '/'  expr
        |   NAME
        ;
```

might be used to structure the input:

```
a  =  b  =  c*d  -  e  -  f*g
```

as follows to perform the correct precedence of operators:

```
a = ( b = ( ((c*d)-e) - (f*g) ) )
```

When this mechanism is used, unary operators must, in general, be given a precedence. Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences. An example is unary and binary minus, −.

Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication. The keyword, **%prec**, changes the precedence level associated with a particular grammar rule. The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal.

It causes the precedence of the grammar rule to become that of the following token name or literal. For example, the rules below might be used to give unary minus the same precedence as multiplication:

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

%%

expr    :   expr  '+'  expr
        |   expr  '-'  expr
        |   expr  '*'  expr
        |   expr  '/'  expr
        |   '-'  expr       %prec  '*'
        |   NAME
        ;
```


A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

Precedences and associativities are used by **yacc** to resolve parsing conflicts. They cause the following disambiguating rules:

1. Precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.
3. When there is a **reduce-reduce** conflict or there is a **shift-reduce** conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two default disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a **shift-reduce** conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action—**shift** or **reduce**—associated with the higher precedence. If precedences are equal, then associativity is used. Left associative implies **reduce**; right associative implies **shift**; nonassociating implies **error**.

Conflicts resolved by precedence are not counted in the number of **shift-reduce** and **reduce-reduce** conflicts reported by **yacc**. This means that mistakes in the

specification of precedences may disguise errors in the input grammar. It is a good idea to be sparing with precedences and use them in a cookbook fashion until some experience has been gained. The **y.output** file is useful in deciding whether the parser is actually doing what was intended.

Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and/or, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser restarted after an error. A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, **yacc** provides the token name **error**. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place. The parser pops its stack until it enters a state where the token **error** is legal. It then behaves as if the token **error** were the current look-ahead token and performs the action encountered. The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

To prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the form:

```
stat    :    error
```

means that on a syntax error, the parser attempts to skip over the statement in which the error is seen. More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and starts processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and report a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as the following are somewhat easier:

```
stat    :    error    ';'
```

Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon. All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any cleanup action associated with it performed.

Another form of **error** rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example is one way to do this:


```
input    :    error  '\n'
                  {
                      (void) printf( "Reenter last line: " );
                  }
                  input
          {
             $$ = $4;
          }
          ;
```

There is one potential difficulty with this approach: The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable so there is a mechanism that can force the parser to believe that error recovery has been accomplished. The statement:

```
yyerrok ;
```

in an action resets the parser to its normal mode.

The last example can be rewritten as:

```
input   :   error  '\n'
                {
                   yyerrok;
                   (void) printf( "Reenter last line: " );
                }
                input
        {
            $$ = $4;
        }
        ;
```

which is somewhat better.

As previously mentioned, the token seen immediately after the **error** symbol is the input token at which the error was discovered. Sometimes, this is inappropriate. For example, an error recovery action might find the correct place to resume input. In such a case, the previous look-ahead token must be cleared. The statement:

```
yyclearin ;
```

in an action will have this effect. For example, suppose the action after **error** were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by **yylex** is presumably the first token in a legal statement. The old illegal token must be discarded and the **error** state reset. A rule similar to:

```
stat    :   error
        {
           resynch();
           yyerrok  ;
           yyclearin;
        }
        ;
```

could perform this task.

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Moreover, the user can get control to deal with the error actions required by other portions of the program.

The yacc Environment

When you input a specification to **yacc**, the output is a file of C language subroutines, called **y.tab.c**. The function produced by **yacc** is called **yyparse**(); it is an integer valued function. When this function is called, it in turn repeatedly calls **yylex**(), the lexical analyzer supplied by the user (see "Lexical Analysis"), to obtain input tokens. Eventually, an error is detected, **yyparse**() returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, **yyparse**() returns the value 0.

The user must provide a certain amount of environment for this parser to obtain a working program. For example, as with every C language program, a routine called **main**() must be defined that eventually calls **yparse**(). In addition, a routine called **yyerror**() is needed to print a message when a syntax error is detected.

These two routines must be supplied in one form or another by the user. To ease the initial effort of using **yacc**, a library has been provided with default versions of **main**() and **yerror**(). The library is accessed by a **–ly** argument to the **cc**(1) command or to the loader. The source codes:

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

and:

```
# include <stdio.h>

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

show the triviality of these default programs. The argument to **yerror**() is a string containing an error message, usually the string **syntax error**. The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected. The external integer variable *yychar* contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics. Since the **main**() routine is probably supplied by the user (to read arguments, etc.), the **yacc** library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable **yydebug** is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions are. It is possible to set this variable by using **sdb**.

Hints for Preparing Specifications

This part of the chapter contains miscellaneous hints on preparing efficient, easy to change, and clear specifications. The individual subsections are more or less independent.

Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are a few style hints.

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names. This is useful in debugging.
2. Put grammar rules and actions on separate lines. It makes editing easier.
3. Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by one tab stop and action bodies by two tab stops.
6. Put complicated actions into subroutines defined in separate files.

Example 1 is written following this style, as are the examples in this section (where space permits). The user must decide about these stylistic questions. The central problem, however, is to make the rules visible through the morass of action code.

Left Recursion

The algorithm used by the **yacc** parser encourages so-called left recursive grammar rules. Rules of the form:

```
name    :    name   rest_of_rule   ;
```

match this algorithm.

These rules, such as:

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

and:

```
seq     :   item
        |   seq  item
        ;
```

frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule will be reduced for the first item only; and the second rule will be reduced for the second and all succeeding items.

With right recursive rules, such as:

```
seq     :   item
        |   item  seq
        ;
```

the parser is a bit bigger and the items are seen and reduced from right to left. More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. Thus, the user should use left recursion wherever reasonable.

It is worthwhile to consider whether a sequence with zero elements has any meaning, and if so, to consider writing the sequence specification as:

```
seq     :   /* empty */
        |   seq  item
        ;
```

using an empty rule. Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read. Permitting empty sequences often leads to increased generality. However, conflicts might arise if **yacc** is asked to decide which empty sequence it has seen when it hasn't seen enough to know!

Lexical Tie-Ins

Some lexical decisions depend on context. For example, the lexical analyzer might want to delete blanks normally, but not within quoted strings; or names might be entered into a symbol table in declarations but not in expressions. One way of handling these situations is to create a global flag that is examined by the lexical analyzer and set by actions.

For example, the segment:

```
%{
   int dflag;
%}
  ...  other declarations ...

%%

prog    :   decls  stats
        ;

decls   :   /* empty */
        {
              dflag = 1;
        }
        |   decls  declaration
        ;

stats   :   /* empty */
        {
              dflag = 0;
        }
        |   stats  statement
        ;

     ...  other rules ...
```

specifies a program that consists of zero or more declarations followed by zero or more statements. The flag **dflag** is now 0 when reading statements and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

This back-door approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult, if not impossible, to do otherwise.

Reserved Words

Some programming languages permit you to use words like **if**, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of **yacc**. It is difficult to pass information to the lexical analyzer telling it this instance of **if** is a keyword and that instance is a

variable. You can try it using the mechanism described in the last subsection, but it is difficult.

Ways of making this easier are under advisement. Until then, it is better that the keywords be reserved, i.e., forbidden for use as variable names. There are powerful stylistic reasons for preferring this.

Advanced Topics

This part discusses some advanced features of **yacc**.

Simulating error and accept in Actions

The parsing actions of **error** and **accept** can be simulated in an action by use of the macros YYACCEPT and YYERROR. The YYACCEPT macro causes **yyparse**() to return the value 0. YYERROR causes the parser to behave as if the current input symbol had been a syntax error; **yyerror**() is called, and error recovery takes place. These mechanisms can be used to simulate parsers with multiple end-markers or context sensitive syntax checking.

Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is the same as with ordinary actions, a dollar sign followed by a digit:

```
sent    :   adj  noun  verb  adj  noun
        {
            look at the sentence ...
        }
        ;
adj     :   THE
        {
                $$ = THE;
        }
        |   YOUNG
        {
                $$ = YOUNG;
        }
        ...
        ;
noun    :   DOG
        {
            $$ = DOG;
        }
        |   CRONE
        {
            if( $0 == YOUNG )
            {
                (void) printf( "what?\n" );
            }
            $$ = CRONE;
        }
        ;
        ...
```

In this case, the digit may be 0 or negative. In the action following the word CRONE, a check is made that the preceding token shifted was not YOUNG.

Obviously, this is only possible when a great deal is known about what might precede the symbol **noun** in the input. There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism prevents much trouble, especially when a few combinations are to be excluded from an otherwise regular structure.

Support for Arbitrary Value Types

By default, the values returned by actions and by the lexical analyzer are integers. **yacc** can also support values of other types, including structures. In addition, **yacc** keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type-checked. The **yacc** value stack is declared to be a **union** of the various types of values desired. The user declares the union and associates union member names with each token and nonterminal symbol having a value. When the value is referenced through a **$$** or **$n** construction, **yacc** automatically inserts the appropriate union name so that no unwanted conversions take place. In addition, type checking commands such as **lint** are far more silent.

Three mechanisms are used to provide for this typing. First, there is a way of defining the union. This must be done by the user since other subroutines, notably the lexical analyzer, must know about the union member names. Second, there is a way of associating a union member name with tokens and nonterminals. Finally, there is a mechanism for describing the type of those few values where **yacc** cannot easily determine the type.

To declare the union, the user includes:

```
%union
{
   body of union ...
}
```

in the declaration section. This declares the **yacc** value stack and the external variables **yylval** and **yyval** to have type equal to this union. If **yacc** was invoked with the **–d** option, the union declaration is copied onto the **y.tab.h** file as YYSTYPE.

Once YYSTYPE is defined, the union member names must be associated with the various terminal and nonterminal names. The construction:

```
<name>
```

is used to indicate a union member name. If this follows one of the keywords **%token, %left, %right,** and **%nonassoc,** the union member name is associated with the tokens listed. Thus, saying:

```
%left  <optype>  '+'  '-'
```

causes any reference to values returned by these two tokens to be tagged with the union member name **optype.**

Another keyword, **%type,** is used to associate union member names with

nonterminals. Thus, one might say:

```
%type  <nodetype>  expr  stat
```

to associate the union member **nodetype** with the nonterminal symbols **expr** and **stat.**

There remain a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type. Similarly, reference to left context values (such as **$0**) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between < and > immediately after the first $. The example below shows this usage:

```
rule    :   aaa
                {
                   $<intval>$ = 3;
                }
                bbb
        {
            fun( $<intval>2, $<other>0 );
        }
        ;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Example 2. The facilities in this subsection are not triggered until they are used. In particular, the use of **%type** will turn on these mechanisms. When they are used, there is a strict level of checking. For example, use of **$n** or **$$** to refer to something with no defined type is diagnosed. If these facilities are not triggered, the **yacc** value stack is used to hold **int**s.

yacc Input Syntax

This section has a description of the **yacc** input syntax as a **yacc** specification. Context dependencies, etc. are not considered. Ironically, although **yacc** accepts an LALR(1) grammar, the **yacc** input specification language is most naturally specified as an LR(2) grammar; the sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule; otherwise, it is a continuation of the current rule, which just happens to have an action embedded in it. As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, and comments, etc.) is a colon. If so, it returns the token C_IDENTIFIER. Otherwise, it returns IDENTIFIER. Literals (quoted strings) are also returned as IDENTIFIERs but never as part of

C_IDENTIFIERs.

```
    /* grammar for the input to yacc */

    /* basic entries */
%token      IDENTIFIER    /* includes identifiers and literals */
%token      C_IDENTIFIER  /* identifier (but not literal) followed by a : */
%token      NUMBER        /* [0-9]+ */

     /*     reserved words: %type=>TYPE %left=>LEFT,etc. */

%token      LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token      MARK    /* the %% mark */
%token      LCURL   /* the %{ mark */
%token      RCURL   /* the %} mark */

    /*  ASCII character literals stand for themselves */

%token    spec

%%

spec    :     defs MARK rules tail
        ;
tail    :     MARK
        {
                  In this action, eat up the rest of the file
        }
        |     /* empty: the second MARK is optional */
        ;

defs    :     /* empty */
        |     defs def
        ;
def     :     START IDENTIFIER
        |     UNION
        {
              Copy union definition to output
        }
        |   LCURL
        {
                    Copy C code to output file
        }
            RCURL
        |   rword tag nlist
        ;

rword   :   TOKEN
        |   LEFT
```

```
        |   RIGHT
        |   NONASSOC
        |   TYPE
        ;

tag     :   /* empty: union tag is optional */
        |   '<' IDENTIFIER '>'
        ;

nlist   :  nmno
        |  nlist nmno
        |  nlist ',' nmno
        ;
nmno    :  IDENTIFIER          /* Note: literal illegal with % type */
        |  IDENTIFIER NUMBER   /* Note: illegal with % type */
        ;

   /* rule section */

rules   :  C_IDENTIFIER rbody prec
        |  rules rule
        ;
rule    :  C_IDENTIFIER rbody prec
        |  '|' rbody prec
        ;


rbody   :  /* empty */
        |  rbody IDENTIFIER
        |  rbody act
        ;

act     :  '{'
           {
                 Copy action translate $$ etc.
           }
           '}'
        ;

prec    :  /* empty */
        |  PREC IDENTIFIER
        |  PREC IDENTIFIER act
        |  prec ';'
        ;
```

Examples

1. A Simple Example

This example gives the complete **yacc** applications for a small desk calculator. The calculator has 26 registers labeled **a** through **z** and accepts arithmetic expressions made up of the following operators and assignments:

```
+, -, *, /, %  (mod operator), & (bitwise and), | (bitwise or),
```

If an expression at the top level is an assignment, only the assignment is done; otherwise, the expression is printed. As in the C language, an integer that begins with 0 (zero) is assumed to be octal; otherwise, it is assumed to be decimal.

As an example of a **yacc** specification, the desk calculator does a reasonable job of showing how precedence and ambiguities are used and demonstrates simple recovery. The major oversimplifications are that the lexical analyzer is much simpler than for most applications, and the output is produced immediately line by line. Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.

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

int regs[26];
int base;

%}

%start list

%token DIGIT LETTER

%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS  /* supplies precedence for unary minus */

%%          /* beginning of rules section */

list        :  /* empty */
            |  list stat '\n'
            |  list error '\n'
            {
               yyerrok;
            }
            ;

stat        :  expr
            {
               (void) printf( "%d\n", $1 );
            }
            |  LETTER '=' expr
            {
               regs[$1] = $3;
            }
            ;

expr        :  '(' expr ')'
            {
                  $$ = $2;
            }
            |  expr '+' expr
            {
                  $$ = $1 + $3;
            }
            |  expr '-' expr
            {
                  $$ = $1 - $3;
```

```
        {
        |  expr '*' expr
        {
              $$ = $1 * $3;
        }
        |  expr '/' expr
        {
              $$ = $1 / $3;
        }
        |   exp '%' expr
        {
               $$ = $1 % $3;
        }
        |   expr '&' expr
        {
              $$ = $1 & $3;
        }
        |   expr '|' expr
        {
              $$ = $1 | $3;
        }
        |  '-' expr  %prec UMINUS
        {
              $$ = -$2;
        }
        |  LETTER
        {
              $$ = reg[$1];
        }
        |  number
        ;

number  :   DIGIT
        {
               $$ = $1; base = ($1==0) ? 8 ; 10;
        }
        |   number DIGIT
        {
              $$ = base * $1 + $2;
        }
        ;

%%      /* beginning of subroutines section */

int yylex( )    /* lexical analysis routine */
{               /* return LETTER for lowercase letter, */
                /* yylval = 0 through 25 */
                /* returns DIGIT for digit, yylval = 0 through 9 */
                /* all other characters are returned immediately */
        int c;
                        /*skip blanks*/
        while ((c = getchar()) == ' ')
```

```
            ;

                /* c is now nonblank */

        if (islower(c))
        {
                yylval = c - 'a';
                return (LETTER);
        }
        if (isdigit(c))
        }
                yylval = c - '0';
                return (DIGIT);

        }
        return (c);
}
```


2. An Advanced Example

This section gives an example of a grammar using some of the advanced features. The desk calculator example in Example 1 is modified to provide a desk calculator that does floating point interval arithmetic. The calculator understands floating point constants and supports the arithmetic operations +, – *, /, and unary – **a** through **z**. Moreover, it also understands intervals written:

```
(X,Y)
```

where **X** is less than or equal to **Y**. There are 26 interval valued variables **A** through **Z** that may also be used. The usage is similar to that in Example 1; assignments return no value and print nothing while expressions print the (floating or interval) value.

This example explores several interesting features of **yacc** and C. Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, INTERVAL, by using **typedef**. **yacc** value stack can also contain floating point scalars and integers (used to index into the arrays holding the variable values). Notice that the entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the actions call functions that return structures as well.

It is also worth noting the use of YYERROR to handle error conditions—division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of **yacc** is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions. Note that scalar can be automatically promoted to an interval if the context demands an interval value. This causes many conflicts when the grammar is run through **yacc**: 18 **shift-reduce** and 26 **reduce-reduce**. The problem can be seen by looking at the two input lines:

```
2.5 + (3.5 - 4.)
```

and:

```
2.5 + (3.5, 4)
```

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind. More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval. This problem is evaded by having two rules for each binary interval valued operator — one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically. Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file; in this way, the conflict will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is instructive. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants. The C language library routine **atof()** is used to do the conversion from a character string to a double-precision value. If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar, provoking a syntax error in the parser and thence error recovery.

```
%{

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

typedef struct interval
{
     double lo, hi;
}  INTERVAL;

INTERVAL vmul(), vdiv();

double atof();

double dreg[26];
INTERVAL vreg[26];

%}

%start line

%union
{
  int ival;
  double dval;
  INTERVAL vval;
}

%token <ival> DREG VREG  /* indices into dreg, vreg arrays */


%token <dval> CONST       /* floating point constant */

%type <dval> dexp         /* expression */

%type <vval> vexp         /* interval expression */

  /* precedence information about the operators */

%left   '+' '-'
%left   '*' '/'
%left   UMINUS   /* precedence for unary minus */

%%      /* beginning of rules section */

lines   :  /* empty */
        |  lines line
        ;
line    :  dexp '\n'
```

```
        {
                (void) printf("%15.8f\n",$1);
        }
        |  vexp '\n'
        {
                (void) printf("(%15.8f, %15.8f)\n", $1.lo, $1.hi);

        }
        |  DREG '=' dexp '\n'
        {
              dreg[$1] = $3;
        }
        |  VREG '=' vexp '\n'
        {
              vreg[$1] = $3;
        }
        |  error '\n'
        {
                yyerrok;
        }
        ;

dexp    :  CONST
        |  DREG
        {
               $$ = dreg[$1];
        }
        |  dexp '+' dexp
        {
                 $$ = $1 + $3;
        }
        |  dexp '-' dexp
        {
                 $$ = $1 - $3;
        }
        |  dexp '*' dexp
        {
                 $$ = $1 * $3;
        }
        |  dexp '/' dexp
        {
                 $$ = $1 / $3;
        }
        |  '-' dexp    %prec UMINUS
        {
                $$ = -$2;
        }
        |  '(' dexp')'
        {
                 $$ = $2;
        }
        ;
```

```
vexp    :  dexp
        {
                $$.hi = $$.lo = $1;
        }
        |  '(' dexp ',' dexp ')'
        {
                $$.lo = $2;
                $$.hi = $4;
                if( $$.lo > $$.hi )
                {
                        (void) printf("interval out of order \n");
                        YYERROR;
                }
        }
        |  VREG
        {
                   $$ = vreg[$1];
        }
        |  vexp '+' vexp
        {
                $$.hi = $1.hi + $3.hi;
                $$.lo = $1.lo + $3.lo;
        }
        |  dexp '+' vexp
        {
                $$.hi = $1 + $3.hi;
                $$.lo = $1 + $3.lo;
        }
        |  vexp '-' vexp
        {
                $$.hi = $1.hi - $3.lo;
                $$.lo = $1.lo - $3.hi;
        }
        |  dvep '-' vdep
        {
                $$.hi = $1 - $3.lo;
                $$.lo = $1 - $3.hi
        }
        |  vexp '*' vexp
        {
                $$ = vmul( $1.lo,$.hi,$3 )
        }
        |  dexp '*' vexp
        {
                $$ = vmul( $1, $1, $3 )
        }
        |  vexp '/' vexp
        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 )
        }
        |  dexp '/' vexp
```

```
        {
                if( dcheck( $3 ) ) YYERROR;
                $$ = vdiv( $1.lo, $1.hi, $3 )
        }
        |  '-' vexp    %prec UMINUS
        {
                $$.hi = -$2.lo;$$.lo = -$2.hi
        }
        |  '(' vexp ')'
        }
                $$ = $2
        }
        ;

%%      /* beginning of subroutines section */

# define BSZ 50   /* buffer size for floating point number */

         /* lexical analysis */

int yylex( )
{
        register int c;

                              /* skip over blanks */
        while ((c = getchar()) == ' ')
        ;
        if (isupper(c))
        {
            yylval.ival = c - 'A'
            return (VREG);
        }
        if (islower(c))
        {
            yylval.ival = c - 'a',
            return( DREG );
        }

            /* gobble up digits, points, exponents */

        if (isdigit(c) || c == '.')
        {
            char buf[BSZ+1], *cp = buf;
            int dot = 0, exp = 0;

            for(; (cp - buf) < BSZ ; ++cp, c = getchar())
                {
                    *cp = c;
                     if (isdigit(c))
                       continue;
                     if (c == '.')
                    {
```

```
                    if (dot++ || exp)
                      return ('.');   /* will cause syntax error */
                      continue;
                    }
                    if( c == 'e')
                    {
                     if (exp++)
                       return ('e');   /* will cause syntax error */
                     continue;
                    }
                          /* end of number */
                    break;
                }

                *cp = ' ';
                if (cp - buf >= BSZ)
                   (void) printf("constant too long - truncated\n");
                else
                   ungetc(c, stdin);   /* push back last char read */
                yylval.dval = atof(buf);
                return (CONST);
        }
        return (c);
}
INTERVAL
hilo(a, b, c, d)
        double a, b, c, d;
{
          /* returns the smallest interval containing a, b, c, and d */

          /* used by *,/ routine */
        INTERVAL v;

        if (a > b)
        {
              v.hi = a;
              v.lo = b;
        }
        else
        {
              v.hi = b;
              v.lo = a;
        }
        if (c > d)
        {
              if (c > v.hi)
                  v.hi = c;
              if (d < v.lo)
                  v.lo = d;
        }
        else
        }
```

```
                if (d > v.hi)
                    v.hi = d;
                if (c < v.lo)
                    v.lo = c;
        }
        return (v);
}
INTERVAL
vmul(a, b, v)
        double a, b;
        INTERVAL v;
{
        return (hilo(a * v.hi, a * v,lo, b * v.hi, b * v.lo));
}
dcheck(v)
        INTERVAL v;
{
        if (v.hi >= 0. && v.lo <= 0.)
        {
             (void) printf("divisor interval contains 0.\n");
             return (1);
        }
        return (0);
{

INTERVAL
vdiv(a, b, v)
        double a, b;
        INTERVAL v;
{
   return (hilo(a / v.hi, a / v,lo, b / v.hi, b / v.lo));
}
```