Module-4

LEX AND YACC

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex.

Lex and Yacc

The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named yylex. The yylex program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected. See Figure 1.

```
          +---------+
Source -> |  Lex  |  -> yylex
          +---------+


          +---------+
Input ->  | yylex | -> Output


          +---------+
```

A YACC Parser

The structure of a lex file is intentionally similar to that of a yacc file; files are divided up into three sections, separated by lines that contain only two percent signs, as follows:

Definition section

%%

Rules section

%%

C code section

- The **definition** section is the place to define macros and to import header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section is the most important section; it associates patterns with C statements. Patterns are simply regular expressions. When the lexer sees some text in the input matching a given pattern, it executes the associated C code. This is the basis of how lex operates.
- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file and link it in at compile time.

Example:

```
/*** Definition section ***/

%{
/* C code to be copied verbatim */
#include <stdio.h>
%}

/* This tells lex to read only one input file */
```

```
%%
   /*** Rules section ***/

   /* [0-9]+ matches a string of one or more digits */
[0-9]+  {
          /* yytext is a string containing the matched text. */
          printf("Saw an integer: %s\n", yytext);
       }

.      {  /* Ignore all other characters. */   }

%%
/*** C Code section ***/

int main(void)
{
   /* Call the lexer, then quit. */
   yylex();
   return 0;
```

REGULAR EXPRESSIONS:

Regular expression specifies a set of strings to be matched. It contains text characters and operator characters The letters of the alphabet and the digits are always text characters; thus the regular expression integer matches the string integer wherever it appears and the expression

a57D

looks for the string a57D.

<u>Operators:</u>

The operator characters are

" \ [ ] ^ - ? . * + | ( ) $ / { } % <>

and if they are to be used as text characters, an escape should be used. The quotation mark operator (") indicates that whatever is contained between a pair of quotes is to be taken as text characters.

Thus

```
xyz"++"
```

matches the string xyz++ when it appears.

- Note that a part of a string may be quoted. It is harmless but unnecessary to quote an ordinary text character; the expression

```
"xyz++"
```

is the same as the one above. Thus by quoting every non-alphanumeric character being used as a text character, the user can avoid remembering the list above of current operator characters, and is safe should further extensions to Lex lengthen the list.

- An operator character may also be turned into a text character by preceding it with \ as in

```
xyz\+\+
```

which is another, less readable, equivalent of the above expressions.

Another use of the quoting mechanism is to get a blank into an expression; blanks or tabs end a rule. Any blank character not contained within []must be quoted.

- Several normal C escapes with \ are recognized: \n is newline, \t is tab, and \b is backspace. To enter \ itself, use \\. Since newline is illegal in an expression, \n must be used; it is not required to escape tab and backspace. Every character but blank, tab, newline and the list above is always a text character.
- Character classes. Classes of characters can be specified using the operator pair []. The construction [abc] matches a single character, which may be a, b, or c. Within

square brackets, most operator meanings are ignored. Only three characters are special: these are \ - and ^. The - character indicates ranges.

For example:

[a-z0-9<>_]indicates the character class containing all the lower case letters, the digits, the angle brackets, and underline. Ranges may be given in either order.

- Using - between any pair of characters which are not both upper case letters, both lower case letters, or both digits is implementation dependent and will get a warning message. If it is desired to include the character - in a character class, it should be first or last; thus

  [-+0-9]

matches all the digits and the two signs.

In character classes, the ^ operator must appear as the first character after the left bracket; it indicates that the resulting string is to be complemented with respect to the computer character set. Thus , [^abc] matches all characters except a, b, or c, including all special or control characters

or [^a-zA-Z]

is any character which is not a letter. The \ character provides the usual escapes within character class brackets.

- <u>Optional expressions</u>.: The operator ? indicates an optional element of an expression. Thus ab?c

matches either ac or abc.

- <u>Repeated expressions</u>: Repetitions of classes are indicated by the operators * and +.

Ex: a*

is any number of consecutive a characters, including zero, while a+ is one or more instances of a.

For example [a-z]+

is all strings of lower case letters.

Defining regular expressions in Lex :

Character	**Meaning**
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	A character class. Matches *any* character in the brackets. If the first character is ^ then it indicates a negation pattern. Example: [abC] matches either of a, b, and C.
*****	Match *zero* or more occurrences of the preceding pattern.
+	Matches *one* or more occurrences of the preceding pattern.
?	Matches *zero or one* occurrences of the preceding pattern.
$	Matches end of line as the last character of the pattern.
{ }	Indicates how many times a pattern can be present. Example: A{1,3} implies one or three occurrences of A may be present.
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table.
^	Negation.
|	Logical OR between expressions.

"<some symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
( )	Groups a series of regular expressions.

Examples of regular expressions:

Regular expression	**Meaning**
joke[rs]	Matches either jokes or joker.
A{1,2}shis+	Matches AAshis, Ashis, AAshi, Ashi.
(A[b-e])+	Matches zero or one occurrences of A followed by any character from b to e.

Tokens in Lex are declared like variable names in C. Every token has an associated expression. (Examples of tokens and expression are given in the following table.) Using the examples in our tables, we'll build a word-counting program. Our first task will be to show how tokens are declared.

Examples of token declarations

Token	**Associated expression**	**Meaning**
number	([0-9])+	1 or more occurrences of a digit
chars	[A-Za-z]	Any character
blank	" "	A blank space
word	(chars)+	1 or more occurrences of *chars*
variable	(chars)+(number)*(chars)*(number)*	

7.3. USING LEX:

If *lex.l* is the file containing the **lex** specification, the C source for the lexical analyzer is produced by running **lex** with the following command:

```
lex  lex.l
```

lex produces a C file called *lex.yy.c*.

Options

There are several options available with the **lex** command. If you use one or more of them, place them between the command name **lex** and the filename argument.

The **-t** option sends **lex**'s output to the standard output rather than to the file *lex.yy.c*.

The **-v** option prints out a small set of statistics describing the so-called finite automata that **lex** produces with the C program *lex.yy.c*.

WORD COUNTING PROGRAM

In this section we can add C variable declarations. We will declare an integer variable here for our word-counting program that holds the number of words counted by the program. We'll also perform token declarations of Lex.

Declarations for the word-counting program

```
%{
int wordCount = 0;
%}
chars [A-za-z\_\'\.\"]
numbers ([0-9])+
delim [" "\n\t]
whitespace {delim}+
words {chars}+
%%
```

The double percent sign implies the end of this section and the beginning of the second of the three sections in Lex programming.

Lex rules for matching patterns

Let's look at the Lex rules for describing the token that we want to match. (We'll use C to define what to do when a token is matched.) Continuing with our word-counting program, here are the rules for matching tokens.

Lex rules for the word-counting program

```
{words} { wordCount++; /*
increase the word count by one*/ }
{whitespace} { /* do
nothing*/ }
{numbers} { /* one may
want to add some processing here*/ }
%%
```

C code

The third and final section of programming in Lex covers C function declarations (and occasionally the main function) Note that this section has to include the yywrap() function. Lex has a set of functions and variables that are available to the user. One of them is yywrap. Typically, yywrap() is defined as shown in the example below.

C code section for the word-counting program

```
void main()
{
    yylex(); /* start the analysis*/
    printf(" No of words:
    %d\n", wordCount);
}
int yywrap()
{
    return 1;
}
```

LEXER

lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a **lexical analyzer**, **lexer** or **scanner**. A lexer often exists as a single function which is called by a parser or another function

Token

A **token** is a string of characters, categorized according to the rules as a symbol (e.g. IDENTIFIER, NUMBER, COMMA, etc.). The process of forming tokens from an input stream of characters is called **tokenization** and the lexer categorizes them according to a symbol type. A token can look like anything that is useful for processing an input text stream or text file.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parenthesis as tokens, but does nothing to ensure that each '(' is matched with a ')'.

Consider this expression in the C programming language:

```
sum=3+2;
```

Tokenized in the following table:

lexeme	**token type**
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number

; End of statement

Tokens are frequently defined by regular expressions, which are understood by a lexical analyzer generator such as lex. The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is called "tokenizing." If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

Examples:

1. Write a Lex source program to copy an input file while adding 3 to every positive number divisible by 7.

```
%%
            int k;
      [0-9]+  {
            k = atoi(yytext);
            if (k%7 == 0)
               printf("%d", k+3);
            else
               printf("%d",k);
            }
```

to do just that. The rule [0-9]+ recognizes strings of digits; atoi converts the digits to binary and stores the result in k. The operator % (remainder) is used to check whether k is divisible by 7; if it is, it is incremented by 3 as it is written out. It may be objected that this program will alter such input items as 49.63 or X7. Furthermore, it increments the absolute value of all negative numbers divisible by 7. To avoid this, just add a few more rules after the active one, as here:

```
        %%
                        int k;
        -?[0-9]+              {
                        k = atoi(yytext);
                        printf("%d",
                          k%7 == 0 ? k+3 : k);
}
        -?[0-9.]+             ECHO;
        [A-Za-z][A-Za-z0-9]+  ECHO;
```

Numerical strings containing a "." or preceded by a letter will be picked up by one of the last two rules, and not changed. The if-else has been replaced by a C conditional expression to save space; the form a?b:c means "if a then b else c".

2. Write a Lex program that histograms the lengths of words, where a word is defined as a string of letters.

```
int lengs[100];
          %%
          [a-z]+  lengs[yyleng]++;
          .      |
          \n      ;
          %%
          yywrap()
          {
          int i;
          printf("Length  No. words\n");
```

```
        for(i=0; i<100; i++)

            if (lengs[i] > 0)

                printf("%5d%10d\n",i,lengs[i]);

        return(1);

        }
```

3.. Write a lex program to find the number of vowels and consonants.

```
%{
/* to find vowels and consonents*/
int vowels = 0;
int consonents = 0;
%}
%%
[ \t\n]+
[aeiouAEIOU] vowels++;
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ] consonents++;
.
%%
main()
{
yylex();
printf(" The number of vowels = %d\n", vowels);
printf(" number of consonents = %d \n", consonents);
return(0);
}
```

The same program can be executed by giving alternative grammar. It is as follows: Here a file is opened which is given as a argument and reads to text and counts the number of vowels and consonants.

```
%{
unsigned int vowelcount=0;
unsigned int consocount=0;
%}
vowel [aeiouAEIOU]
consonant [bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]
eol \n

%%

{vowel} { vowelcount++;}
{consonant} { consocount++; }

%%
main(int argc,char *argv[])
{
if(argc > 1)
  {
    FILE *fp;
fp=fopen(argv[1],"r");
if(!fp)
    {
fprintf(stderr,"could not open %s\n",argv[1]);
exit(1);
    }
yyin=fp;
  }
yylex();
printf(" vowelcount=%u  consonantcount=%u\n ",vowelcount,consocount);
return(0);
}
```

4. Write a Lex program to count the number of words, characters, blanks and lines in a given text.

```
%{
        unsigned int charcount=0;
int wordcount=0;
int linecount=0;
int blankcount =0;
%}
word[^ \t\n]+
eol \n
%%
[ ] blankcount++;
{word} { wordcount++; charcount+=yyleng;}
{eol} {charcount++; linecount++;}
. { ECHO; charcount++;}
%%
main(argc, argv)
int argc;
char **argv;
{
if(argc > 1)
   {
        FILE *file;
        file = fopen(argv[1],"r");
        if(!file)
        {
        fprintf(stderr, "could not open %s\n", argv[1]);
        exit(1);
        }
        yyin = file;
yylex();
printf("\nThe number of characters = %u\n", charcount);
printf("The number of wordcount = %u\n", wordcount);
printf("The number of linecount = %u\n", linecount);
printf("The number of blankcount = %u\n", blankcount);
return(0);
    }
else
printf(" Enter the file name along with the program \n");
}
```

5. Write a lex program to find the number of positive integer, negative integer, positive floating positive number and negative floating point number.

```
%{
        int posnum = 0;
        int negnum = 0;
int posflo = 0;
        int negflo = 0;
%}
%%
[\n\t ];
  ([0-9]+) {posnum++;}
-?([0-9]+) {negnum++; }
  ([0-9]*\.[0-9]+)  { posflo++; }
-?([0-9]*\.[0-9]+) { negflo++; }
. ECHO;
%%
main()
 {
        yylex();
printf("Number of positive numbers = %d\n", posnum);
        printf("number of negative numbers = %d\n", negnum);
        printf("number of floting positive number = %d\n", posflo);
        printf("number of floating negative number = %d\n", negflo);
 }
```

6. Write a lex program to find the given c program has right number of brackets. Count the number of comments. Check for while loop.

```
%{
  /* find main, comments, {, (, ), } */
int comments=0;
int opbr=0;
int clbr=0;
int opfl=0;
int clfl=0;
int j=0;
int k=0;
%}
%%
"main()" j=1;
"/*"[ \t].*[ \t]"*/" comments++;
"while("[0-9a-zA-Z]*")"[ \t]*\n"{"[ \t]*.*"}"  k=1;
^[ \t]*"{"[ \t]*\n
^[ \t]*"}"                          k=1;
"(" opbr++;
")" clbr++;
"{"  opfl++;
 "}" clfl++;
```

```
[^ \t\n]+
. ECHO;
%%
main(argc, argv)
int argc;
char *argv[];
{
if (argc > 1)
   {
          FILE *file;
          file = fopen(argv[1], "r");
          if (!file)
        {
          printf("error opeing a file \n");
exit(1);
        }
          yyin = file;
     }
yylex();
if(opbr != clbr)
printf("open brackets is not equal to close brackets\n");
if(opfl != clfl)
                printf("open flower brackets is not equal to close flower brackets\n");
          printf(" the number of comments = %d\n",comments);
if (!j)
          printf("there is no main function \n");
if (k)
printf("there is loop\n");
else printf("there is no valid for loop\n");
return(0);
}
```

6. Write a lex program to replace scanf with READ and printf with WRITE statement also find the number of scanf and printf.

```
%{
int pc=0,sc=0;
%}
%%
printf fprintf(yyout,"WRITE");pc++;
scanf fprintf(yyout,"READ");sc++;
. ECHO;
%%
main(int argc,char* argv[])
{
if(argc!=3)
  {
printf("\nUsage: %s <src><dest>\n",argv[0]);
return;
  }
yyin=fopen(argv[1],"r");
yyout=fopen(argv[2],"w");
yylex();
printf("\nno. of printfs:%d\nno. of scanfs:%d\n",pc,sc);
}
```

7. Write a lex program to find whether the given expression is valid.

```
%{
  #include <stdio.h>
int valid=0,ctr=0,oc = 0;
%}
NUM [0-9]+
OP  [+*/-]
%%
{NUM}({OP}{NUM})+ {
valid = 1;
for(ctr = 0;yytext[ctr];ctr++)
                         {
                    switch(yytext[ctr])
                            {
                    case '+':
case '-':
case '*':
case '/': oc++;
                            }
                          }
               }
{NUM}\n {printf("\nOnly a number.");}
\n { if(valid) printf("valid \n operatorcount = %d",oc);
else  printf("Invalid");
valid = oc = 0;ctr=0;
   }
%%
main()
{
yylex();
}

/*      Another solution for the same problem       */

%{
int oprc=0,digc=0,top=-1,flag=0;
char stack[20];
```

```
%}
digit [0-9]+
opr [+*/-]
%%
[ \n\t]+
['(']  {stack[++top]='(';}
[')'] {flag=1;
if(stack[top]=='('&&(top!=-1))
                    top--;
else
               {
                    printf("\n Invalid expression\n");
                    exit(0);
                    }
        }
{digit}  {digc++;}
{opr}/['('] { oprc++; printf("%s",yytext);}
{opr}/{digit} {oprc++; printf("%s",yytext);}
. {printf("Invalid "); exit(0);}
%%
main()
{
yylex();
if((digc==oprc+1||digc==oprc) && top==-1)
{
printf("VALID");
printf("\n oprc=%d\n digc=%d\n",oprc,digc);
}
else
printf("INVALID");
}
```

8.Write a lex program to find the given sentence is simple or compound.

```
%{
int flag=0;
%}
%%
(" "[aA][nN][dD]" ")|(" "[oO][rR]" ")|(" "[bB][uU][tT]" ") flag=1;
. ;
%%
main()
{yylex();
if (flag==1)
        printf("COMPOUND SENTENCE \n");
else
        printf("SIMPLE SENTENCE \n");
}
```

9. Write a lex program to find the number of valid identifiers.

```
%{
int count=0;
%}
%%
(" int ")|(" float ")|(" double ")|(" char ")

{
int ch; ch = input();
for(;;)
  {
if (ch==',') {count++;}
else
```

```
            if(ch==';') {break;}
            ch = input();
   }
count++;
}
%%
main(int argc,char *argv[])
{
yyin=fopen(argv[1],"r");
yylex();
printf("the no of identifiers used is %d\n",count);
}
```

RECOMMENDED QUESTIONS:

1. write the specification of lex with an example? (10)
2. what is regular expressions? With examples explain? (8)
3. write a lex program to count the no of words , lines , space, characters? (8)
4. write a lex program to count the no of vowels and consonants? (8)
5. what is lexer- parser communication? Explain? (5)
6. write a program to count no of words by the method of substitution? (7)

YACC

USING YACC

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine**.**

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters or in terms of higher level constructs such as names and numbers. The user supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specification. Yacc is written in portable C.

Yacc provides a general tool for imposing structure on the input to a computer program. User prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input.

Grammars:

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. For example, one grammar rule might be

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

Here, date, month_name, day, and year represent structures of interest in the input process; presumably, month_name, day, and year are defined elsewhere. The comma ``,'' is enclosed in single quotes; this implies 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 controlling the input. Thus, with proper definitions, the input

July 4, 1776

might be matched by the above rule.

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizing the lower level structures, and communicates these tokens to the parser. For historical reasons, a structure recognized by the lexical analyzer is called a terminal symbol, while the structure recognized by the parser is called a nonterminal symbol. To avoid confusion, terminal symbols will usually be referred to as tokens.

Basic Specifications:

Every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%'' marks. (The percent ``%'' is generally used in Yacc specifications as an escape character.)

In other words, a full specification file looks like

declarations

%%

rules

%%

programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

%%

rules

Blanks, tabs, and newlines are ignored except that they may not appear in names or multi-character reserved symbols. Comments may appear wherever a name is legal; they are enclosed in /* . . . */, as in C and PL/I.

The rules section is made up of one or more grammar rules.

A grammar rule has the form:

A:BODY;

A represents a non terminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are Yacc punctuation. Names may be of arbitrary length, and may be made up of letters, dot ``.'', underscore ``_'', and non-initial digits. Upper and lower case letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

AYACC PARSER

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

'\n' newline

'\r' return

'\'' single quote ``'''

'\\' backslash ``\''

'\t' tab

'\b' backspace

'\f' form feed

'\xxx' ``xxx'' in octal

For a number of technical reasons, the NUL 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 can be 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
            ;
```

- It is not necessary that all grammar rules with the same left side appear together in the grammar rules section, although it makes the input much more readable, and easier to change.
- If a nonterminal symbol matches the empty string, this can be indicated in the obvious way:
- **empty : ;**
- 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 non-terminal symbol. Every non-terminal symbol must appear on the left side of at least one rule.

- Of all the nonterminal symbols, one, called the start symbol, has particular importance. The parser is designed to recognize the start symbol; thus, this symbol represents the largest, most general structure described by the grammar rules. 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 in fact 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 endmarker. If the tokens up to, but not including, the endmarker form a structure which matches the start symbol, the parser function returns to its caller after the end-marker is seen; it accepts the input. If the endmarker is seen in any other context, it is an error.
- It is the job of the user-supplied lexical analyzer to return the endmarker when appropriate; see section 3, below. Usually the endmarker 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 Yacc: Yet Another Compiler-Compiler performed each time the rule is recognized in the input process.

- These 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 statement, and as such can do input and output, call subprograms, and alter external vectors 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
            { printf("a message\n");
              flag = 25; }
```

are grammar rules with actions.

To facilitate easy communication between the actions and the parser, the action statements are altered slightly. The symbol ``dollar sign'' ``$'' is used as a signal to Yacc in this context.

To return a value, the action normally sets the pseudo-variable ``$$'' to some value. For example, an action that does nothing but return the value 1 is

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

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables $1, $2, . . ., which refer to the values returned by the components of the right side of a rule, reading from left to right. Thus, if the rule is

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

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

As a more concrete example, consider the rule

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

The value returned by this rule is usually the value of the expr in parentheses. This 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 the examples above, all the actions came at the end of their 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.

The user 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 the lexical analyzer. For example,

%{ int variable = 0; %}

could be placed in the declarations section, making variable accessible to all of the actions. The Yacc parser uses only names beginning in ``yy"; the user should avoid such names.

In these examples, all the values are integers.

Lexer

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 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 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 chosen by the user. In either case, the ``# define" mechanism of C 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. The relevant portion of the lexical analyzer might look like:

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. Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT will be defined as the token number associated with the token DIGIT.

- This mechanism leads to clear, easily modified lexical analyzers; the only pitfall is the need to avoid using any token names in the grammar that are reserved or significant in C 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.
- The token numbers may be chosen by Yacc or by the user. In the default situation, the 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.

Compiling and running a SimpleParser:

Yacc turns the specification file into a C program, which parses the input according to the specification given. The algorithm used to go from the specification to the parser is complex. however, is relatively simple, and understanding how it works, while not strictly necessary, will nevertheless make treatment of error recovery and ambiguities much more comprehensible.

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 lookahead 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 lookahead token has been read.

The machine has only four actions available to it, called shift, reduce, accept, and error. A move of the parser is done as follows:

1. Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls yylex to obtain the next token.

2. Using the current state, and the lookahead 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 lookahead token being processed or left alone.

The shift action is the most common action the parser takes. Whenever a shift action is taken, there is always a lookahead token. For example, in state 56 there may be an action:

IF shift 34

which says, in state 56, if the lookahead 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 lookahead token to decide whether to reduce, but usually it is not; in fact, the default action (represented by a ``.'') is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, leading to some confusion. The action

reduce 18

refers to grammar rule 18, while the action

IF shift 34

refers to state 34. Suppose the rule being reduced is

A : x y z ;

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. After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule. Using thisuncovered 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, pushed onto the stack, and parsing continues.

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 returnedfrom 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.

Arithmetic Expressions and Ambiguity:

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 of forming 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 as either

(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. It is instructive to consider the problem that confronts the parser when it is given an input such as

expr - expr - expr

When the parser has read the second expr, the input that it has 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, when the parser has seen

expr - expr

it could defer the immediate application of the rule, and continue reading the input until it had seen

expr - expr - expr

It could then apply the rule to the rightmost three symbols, reducing them to expr and leaving

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 two legal things, a shift or a reduction, and 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 which choice to make in a given situation is called a disambiguating rule.

Yacc invokes two **disambiguating** rules by default:

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 input sequence).

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

Yacc always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

As an example of the power of disambiguating rules, consider a fragment from a programming language involving an ``if-then-else" construction:

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

        |    IF  '(' cond  ')'  stat  ELSE  stat

        ;
```

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, since input of the form

EXAMPLE:

IF (C1) IF (C2) S1 ELSE S2

can be structured according to these rules in two ways:

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

or

```
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 situation 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.

- Once again the parser can do two valid things - there is a shift/reduce conflict. The application of disambiguating rule 1 tells the parser to shift in this case, which leads to the desired grouping.
- This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs already seen, such as

IF (C1) IF (C2) S1

- 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.

stat : IF '(' cond ')' stat

- 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 after those rules which can be reduced.

Variables and Typed Tokens

There is one common situation where the rules given above 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 associatively. 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 form

  expr : expr OP expr

  and

  expr : UNARY expr

  for all binary and unary operators desired. This creates a very 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 is sufficient to allow Yacc to resolve the parsing conflicts in accordance with 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 of 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,

```
%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

  %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

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 '-'; 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. %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, to make unary minus have the same precedence as multiplication the rules might resemble:

```
%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.

The precedence and associatively are used by Yacc to resolve parsing conflicts; they give rise to disambiguating rules. Formally, the rules work as follows:

1. The 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 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 the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and 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 an essentially ``cookbook" fashion, until some experience has been gained. The y.output file is very useful in deciding whether the parser is actually doing what was intended.

Recursive rules:

The algorithm used by the Yacc parser encourages so called ``left recursive" grammar rules: rules of the form

name : name rest_of_rule ;

These rules frequently arise when writing specifications of sequences and lists:

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

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

It is worth considering whether a sequence with zero elements has any meaning, and if so, consider writing the sequence specification with an empty rule:

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

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

RUNNING BOTH LEXER AND PARSER:

The yacc program gets the tokens from the lex program. Hence a lex program has be written to pass the tokens to the yacc. That means we have to follow different procedure to get the executable file.

- **i.** The lex program <lexfile.l> is fist compiled using lex compiler to get **lex.yy.c.**
- **ii.** The yacc program <yaccfile.y> is compiled using yacc compiler to get **y.tab.c.**
- **iii.** Using c compiler b+oth the lex and yacc intermediate files are compiled with the lex library function. **cc y.tab.c lex.yy.c –ll.**
- **iv.** If necessary out file name can be included during compiling with –o option.

Examples

1. Write a Yacc program to test validity of a simple expression with +, - , /, and *.

/* Lex program that passes tokens */

```
%{
        #include "y.tab.h"
        extern int yyparse();
%}
%%
[0-9]+ { return NUM;}
[a-zA-Z_][a-zA-Z_0-9]* { return IDENTIFIER;}
[+-] {return ADDORSUB;}
[*/] {return PROORDIV;}
[)(] {return yytext[0];}
[\n] {return '\n';}
%%
int main()
{
  yyparse();
}
/* Yacc program to check for valid expression */

%{
#include<stdlib.h>
extern int yyerror(char * s);
extern int yylex();
%}
%token NUM
%token ADDORSUB
%token PROORDIV
%token IDENTIFIER
%%
input :
      | input line
              ;
line       : '\n'
              | exp '\n' { printf("valid"); }
              | error '\n' { yyerror; }
              ;
exp        : exp ADDORSUB term
              | term
              ;
term       : term PROORDIV factor
              | factor
              ;
factor : NUM
       | IDENTIFIER
       | '(' exp ')'
              ;
%%
```

```
int yyerror(char *s)
{
  printf("%s","INVALID\n");
}

/* yacc program that gets token from the c porogram */

%{
#include <stdio.h>
#include <ctype.h>
%}
%token NUMBER LETTER
%left '+' '-'
%left '*' '/'
%%
line:line expr '\n' {printf("\nVALID\n");}
   | line '\n'
   |
   |error '\n' { yyerror ("\n INVALID"); yyerrok;}
   ;
expr:expr '+' expr
   |expr '-' expr
   |expr '*'expr
   |expr '/' expr
   | NUMBER
   | LETTER
   ;
%%
main()
{
yyparse();
}
yylex()
{
char c;
while((c=getchar())==' ');
if(isdigit(c)) return NUMBER;
if(isalpha(c)) return LETTER;
return c;
}
yyerror(char *s)
{
printf("%s",s);
}
```

2. Write a Yacc program to recognize validity of a nested 'IF' control statement and display levels of nesting in the nested if.

```
/* Lex program to pass tokens */

%{

        #include "y.tab.h"

%}

digit [0-9]

num {digit} + ("." {digit}+)?

binopr [+-/*%^=><&|"= ="| "!=" | ">=" | "<="

unopr [~!]

char [a-zA-Z_]

id {char}({digit} | {char})*

space [ \t]
```

```
%%
   {space} ;
   {num} return  num;
   { binopr } return binopr;
   { unopr } return unopr;
   { id} return id
 "if"  return if
  . return yytext[0];
%%
NUMBER {DIGIT}+
/* Yacc program to check for the valid expression */
%{
#include<stdio.h>
int cnt;
%}
%token binopr
%token unop
%token num
%token id
%token if
%%
foo: if_stat { printf("valid: count = %d\n", cnt); cnt = 0;
```

```
                exit(0);
            }
        | error { printf(“Invalid \n”); }
if_stat: token_if ‘(‘ cond ‘)’ comp_stat {cnt++;}
cond: expr
        ;
expr:   sim_exp
        | ‘(‘ expr ‘)’
        | expr binop factor
        | unop factor
        ;
factor: sim_exp
        | ‘(‘ expr ‘)’
        ;
sim_exp:        num
        | id
        ;
sim_stat:       expr ‘;’
        | if
        ;
stat_list:      sim_stat
```

```
            | stat_list sim_stat
            ;
comp_stat:      sim_stat
            | '{' stat_list '}'
            ;
%%
main()
{
            yyparse();
}
yyerror(char *s)
{
            printf("%s\n", s);
            exit(0);
}
```

3. Write a Yacc program to recognize a valid arithmetic expression that uses +, - , / , *.

```
%{
            #include<stdio.h>
            #include <type.h>
%}
```

```
% token num

% left '+' '-'

% left '*' '/'

%%

st        :  st expn '\n'  {printf ("valid \n"); }

          |

          |  st '\n'

          | error '\n'  { yyerror ("Invalid \n"); }

          ;

%%

void main()

          {

                    yyparse (); return 0 ;

          }

          yylex()

          {

                    char c;

                    while (c = getch () ) == ' ')

                              if (is digit (c))

                              return num;
                              return c;
          }
          yyerror (char *s)
          {
                    printf("%s", s);
          }
```

4. Write a yacc program to recognize an valid variable which starts with letter followed by a digit. The letter should be in lowercase only.

```
/*      Lex program to send tokens to the yacc program      */


%{
        #include "y.tab.h"
%}
%%
[0-9] return digit;
[a-z]  return letter;
[\n]  return yytext[0];
.  return 0;
%%


/*      Yacc program to validate the given variable         */
```

```
%{
        #include<type.h>
%}
% token  digit letter ;
%%
ident   :  expn '\n' { printf ("valid\n"); exit (0); }
        ;
expn    : letter
        | expn letter
        | expn digit
        | error  { yyerror ("invalid \n");  exit (0); }
        ;
%%
main()
{
        yyparse();
}
yyerror (char *s)
{
        printf("%s", s);
}



/*        Yacc program which has c program to pass tokens                */



%{
#include <stdio.h>
#include <ctype.h>
%}
%token LETTER DIGIT
%%
st:st LETTER DIGIT  '\n' {printf("\nVALID");}
  | st '\n'
  |
  | error '\n' {yyerror("\nINVALID");yyerrok;}
  ;
%%
 main()
{
yyparse();
}

yylex()
{
char c;
while((c=getchar())==' ');
if(islower(c))  return LETTER;
if(isdigit(c)) return DIGIT;
return c;
}
```

```
yyerror(char *s)
  {
  printf("%s",s);
  }
```

5.Write a yacc program to evaluate an expression (simple calculator program).

```
/*        Lex program to send tokens to the Yacc program     */
%{

                #include" y.tab.h"

                expern int yylval;

%}

%%

[0-9]    digit

char[_a-zA-Z]

id       {char} ({ char } | {digit })*

%%

{digit}+ {yylval = atoi (yytext);

                return num;

         }

{id}     return name

[ \t]    ;

\n       return 0;

.        return yytext [0];
```

```
%%

/*        Yacc Program to work as a calculator          */

%{

          #include<stdio.h>

          #include <string.h>

          #include <stdlib.h>

%}

% token num name

% left '+' '-'

% left '*' '/'

% left unaryminus

%%

st        : name '=' expn

          | expn  { printf ("%d\n" $1);  }

          ;

expn      :  num  { $$ = $1 ; }

          | expn '+' num { $$ = $1 + $3; }

          | expn '-'  num  { $$ = $1 - $3; }

          | expn '*' num  { $$ = $1 * $3; }

          | expn '/'  num  { if (num == 0)

                    { printf ("div by zero \n");
                    exit (0);
                  }
                  else
                   { $$ = $1 / $3; }
          | '(' expn ')'  { $$ = $2; }
          ;
%%
main()
{
          yyparse();
}
yyerror (char *s)
{
          printf("%s", s);
}
```

5. Write a yacc program to recognize the grammar { a^{n}b for n >= 0}.

```
/*        Lex program to pass tokens to yacc program          */
%{
          #include "y.tab.h"
%}
[a] { return  a ;  printf("returning A to yacc \n"); }
```

```
[b]  return  b

[\n]  return yytex[0];

.  return error;

%%


/*        Yacc program to check the given expression              */


%{

          #include<stdio.h>

%}

% token a b error

%%

input    : line

          | error

          ;

line     : expn '\n' { printf(" valid new line char \n"); }

          ;

expn     : aa expn bb

          | aa

          ;

aa       : aa a
```

```
        | a
        ;
bb      : bb b
        | b
        ;
error   : error  { yyerror ( " " ) ; }

%%
main()
{
        yyparse();
}
yyerror (char *s)
{
        printf("%s", s);
}
```

```
/* Yacc to evaluate the expression and has c program for tokens */

%{
/* 6b.y   {A^NB  N >=0}    */

#include <stdio.h>
%}
%token A B
%%
st:st reca endb '\n'      {printf("String belongs to grammar\n");}
  | st endb  '\n'          {printf("String belongs to grammar\n");}
  | st '\n'
  | error '\n'       {yyerror ("\nDoes not belong to grammar\n");yyerrok;}
|
  ;
reca: reca enda | enda;
enda:A;
endb:B;
%%
main()
{
yyparse();
}
yylex()
{
char c;
while((c=getchar())==' ');
if(c=='a')
   return A;
if(c=='b')
   return B;
return c;
}
yyerror(char *s)
 {
fprintf(stdout,"%s",s);
}
```

7. Write a program to recognize the grammar { $a^n b^n$ | n >= 0 }

```
/*      Lex program to send tokens to yacc program          */



%{

        #include "y.tab.h"

%}

[a] {return  A ;  printf("returning A to yacc \n"); }

[b]  return B

[\n]  return yytex[0];

.  return error;

%%

/*      yacc program that evaluates the expression  */

%{

        #include<stdio.h>

%}

% token a b error



%%



input   : line

        | error
        ;
line    : expn '\n' { printf(" valid new line char \n"); }
        ;
expn    : aa expn bb
        |
        ;
error   : error  { yyerror ( " " ) ; }

%%
main()
{
        yyparse();
}
yyerror (char *s)
{
        printf("%s", s);
}

/*      Yacc program which has its own c program to send tokens */
%{
/* 7b.y   {A^NB^N  N >=0}    */
```

```
#include <stdio.h>
%}
%token A B
%%
st:st reca endb '\n'      {printf("String belongs to grammar\n");}
  | st  '\n'              {printf("N value is 0,belongs to grammar\n");}
  |
  | error '\n'
                         {yyerror ("\nDoes not belong to grammar\n");yyerrok;}
;
reca: enda reca endb | enda;
enda:A;
endb:B;
%%
main()
{
yyparse();
}
yylex()
{
char c;
while((c=getchar())==' ');
if(c=='a')
   return A;
if(c=='b')
   return B;
return c;
}
yyerror(char *s)
 {
fprintf(stdout,"%s",s);
```

```
}
```

8. Write a Yacc program t identify a valid IF statement or IF-THEN-ELSE statement.

```
/*      Lex program to send tokens to yacc program          */



%{
#include "y.tab.h"
%}
CHAR [a-zA-Z0-9]
%x CONDSTART
%%
<*>[ ]  ;
<*>[ \t\n]+  ;
<*><<EOF>>  return 0;
if return(IF);
else return(ELSE);
then return(THEN);
\(  {BEGIN(CONDSTART);return('(');}
<CONDSTART>{CHAR}+ return COND;
<CONDSTART>\)  {BEGIN(INITIAL);return(')');}
{CHAR}+ return(STAT) ;
%%

/*  Yacc program to check for If and IF Then Else statement          */



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

```
%token IF COND THEN STAT ELSE
%%
Stat:IF '(' COND ')' THEN STAT {printf("\n VALId Statement");}
   | IF '(' COND ')' THEN STAT ELSE STAT {printf("\n VALID Statement");}
   |
   ;
%%
main()
{
 printf("\n enter statement ");
 yyparse();
}
yyerror (char *s)
{
 printf("%s",s);
}

/*        Yacc program that has c program to send tokens        */


%{

          #include <stdio.h>

          #include <ctype.h>

%}

%token if simple

% noassoc reduce

% noassoc else

%%
```

```
start     : start st '\n'
          |
          ;
st        : simple
          | if_st
          ;
if_st     : if st %prec reduce  { printf ("simple\n"); }
          | if st else st          {printf ("if_else \n"); }
          ;
%%
int yylex()
{
  int c;
          c = getchar();
          switch ( c )
          {
                    case 'i' : return if;
                    case 's' : return simple;
                    case 'e' : return else;
                    default : return c;
          }
}

main ()

{

  yy parse();

}

yyerror (char *s)

{

           printf("%s", s);

}
```

RECOMMENDED QUESTIONS:

1. give the specification of yacc program? give an example? (8)
2. what is grammar? How does yacc parse a tree? (5)
3. how do you compile a yacc file? (5)
4. explain the ambiguity occurring in an grammar with an example? (6)
5. explain shift/reduce and reduce/reduce parsing ? (8)
6. write a yacc program to test the validity of an arthimetic expressions? (8)
7. write a yacc program to accept strings of the form anbn , n>0? (8)