CHAPTER 1

Introduction to LEX

Lex and YACC helps you write programs that transforms structured input. Lex generates C code for lexical analyzer where as YACC generates Code for Syntax analyzer. Lexical analyzer is build using a tool called LEX. Input is given to LEX and lexical analyzer is generated.

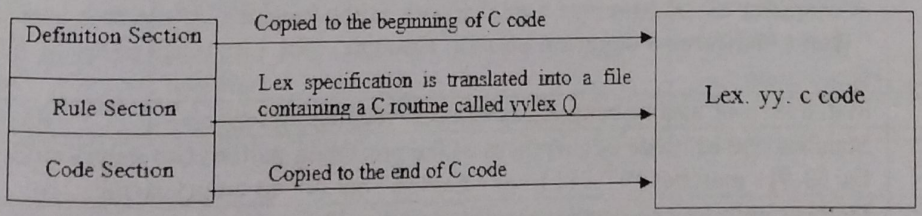
Lex is a UNIX utility. It is a program generator designed for lexical processing of character input streams. Lex generates C code for lexical analyzer. It uses the **patterns** that match **strings in the input** and converts **the strings** to tokens. Lex helps you by taking a set of descriptions of possible tokens and producing a C routine, which we call a lexical analyzer. The token descriptions that Lex uses are known as regular expressions.

1.1 Steps in writing LEX Program:

1st step: Using gedit create a file with extension l. For example: prg1.l
2nd Step: lex prg1.l
3rd Step: cc lex.yy.c –ll
4th Step: ./a.out

1.2 Structure of LEX source program:

1st Col	2nd Col	3rd Col	4th Col
	DEFINITION SECTION		
%%			
	RULE SECTION		
%%			
	CODE SECTION		



%% is a delimiter to the mark the beginning of the Rule section. The second %% is optional, but the first is required to mark the beginning of the rules. The definitions and the code subroutines are often omitted.

Lex variables

yyin	Of the type FILE*. This points to the current file being parsed by the lexer.
yyout	Of the type FILE*. This points to the location where the output of the lexer will be written. By default, both yyin and yyout point to standard input and output.
yytext	The text of the matched pattern is stored in this variable (char*).
yyleng	Gives the length of the matched pattern.
yylineno	Provides current line number information. (May or may not be supported by the lexer.)

Lex functions

yylex()	The function that starts the analysis. It is automatically generated by Lex.
yywrap()	This function is called when end of file (or input) is encountered. If this function returns 1, the parsing stops. So, this can be used to parse multiple files. Code can be written in the third section, which will allow multiple files to be parsed. The strategy is to make yyin file pointer (see the preceding table) point to a different file until all the files are parsed. At the end, yywrap() can return 1 to indicate end of parsing.
yyless(int n)	This function can be used to push back all but first 'n' characters of the read token.
yymore()	This function tells the lexer to append the next token to the current token.

1.3. Regular Expressions

It is used to describe the pattern. It is widely used to in lex. It uses meta language. The character used in this meta language are part of the standard ASCII character set. An expression is made up of symbols. Normal symbols are characters and numbers, but there are other symbols that have special meaning in Lex. The following two tables define some of the symbols used in Lex and give a few typical examples.

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.(no empty string) Ex: [0-9]+ matches "1","111" or "123456" but not an empty string.
?	Matches zero or one occurrences of the preceding pattern. Ex: -?[0-9]+ matches a signed number including an optional leading minus.
?	Matches zero or one occurrences of the preceding pattern. Ex: -?[0-9]+ matches a signed number including an optional leading minus.
$	Matches end of line as the last character of the pattern.
{ }	.1) Indicates how many times a pattern can be present. Example: A{1,3} implies one to three occurrences of A may be present.

Character	Meaning
	2) If they contain name, they refer to a substitution by that name. Ex: {digit}
\	Used to escape meta characters. Also used to remove the special meaning of characters as defined in this table. Ex: \n is a newline character, while "*" is a literal asterisk.
^	Negation.
|	Matches either the preceding regular expression or the following regular expression. Ex: cow|sheep|pig matches any of the three words.
"< 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 together into a new regular expression. Ex: (01) represents the character sequence 01. Parentheses are useful when building up complex patterns with *,+ and |

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.
[0-9]	0 or 1 or 2 or.....9
[0-9]+	1 or 111 or 12345 or ...At least one occurrence of preceding exp
[0-9]*	Empty string (no digits at all) or one or more occurrence.
-?[0-9]+	-1 or +1 or +2 .....
[0.9]*\.[0.9]+	0.0.4.5 or .31415 But won't match 0 or 2

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)*	

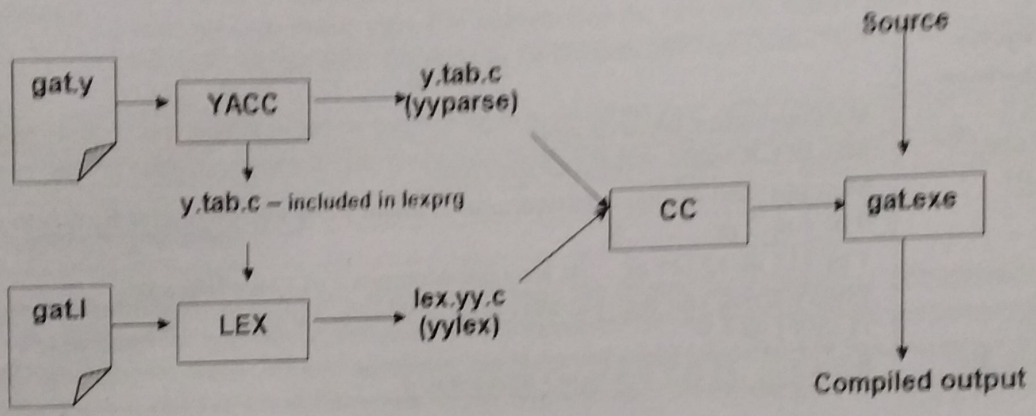
CHAPTER 2

Introduction to YACC

YACC provides a general tool for imposing structure on the input to a computer program. The input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name. YACC prepares a specification of the input process. YACC generates a function to control the input process. This function is called a parser.

The name is an acronym for "Yet Another Compiler Compiler". YACC generates the code for the parser in the C programming language. YACC was developed at AT& T for the UNIX operating system. YACC has also been rewritten for other languages, including Java, ADA

The function parser calls the lexical analyzer to pick up the tokens from the input stream. These tokens are organized according to the input structure rules. The input structure rule is called as grammar. When one of the rule is recognized, then user code supplied for this rule (user code is action) is invoked. Actions have the ability to return values and makes use of the values of other actions.



2.1 Steps in writing YACC Program:

1st step:	Using gedit editor create a file with extension y. For example: prg1.y
2nd Step:	YACC –d prg1.y
3rd Step:	lex prg1.l
4th Step:	cc y.tab.clex.yy.c -ll
5th Step:	/a.out

When we run YACC, it generates a parser in file y.tab.c and also creates an include file y.tab.h. To obtain tokens, YACC calls yylex. Function yylex has a return type of int, and returns the token. Values associated with the token are returned by lex in variable yylval.

2.2 Structure of YACC source program:

Basic Specification:

Every YACC specification file consists of three sections. The declarations, Rules (of grammars), programs. The sections are separated by double percent "%%" marks. The % is generally used in YACC specification as an escape character.

The general format for the YACC file is very similar to that of the Lex file.

1st Col	2nd Col	3rd Col	4th Col
	DEFINITION SECTION		
%%			
	RULE SECTION		
%%			
	CODE SECTION		

%% is a delimiter to the mark the beginning of the Rule section.

Definition Section

%union	It defines the Stack type for the Parser. It is a union of various datas/structures/ objects
%token	These are the terminals returned by the yylex function to the YACC. A token can also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as %token <stack member>tokenName. Ex: %token NAME NUMBER
%type	The type of a non-terminal symbol in the Grammar rule can be specified with this.The format is %type <stack member>non-terminal.
%noassoc	Specifies that there is no associatively of a terminal symbol.
%left	Specifies the left associatively of a Terminal Symbol
%right	Specifies the right associatively of a Terminal Symbol.
%start	Specifies the L.H.S non-terminal symbol of a production rule which should be taken as the starting point of the grammar rules.
%prec	Changes the precedence level associated with a particular rule to that of the following token name or literal

Rules Section

The rules section simply consists of a list of grammar rules. A grammar rule has the form:

A: BODY

A represents a nonterminal name, the colon and the semicolon are YACC punctuation and BODY represents names and literals. The names used in the body of a grammar rule may represent tokens or nonterminal symbols. The literal consists of a character enclosed in single quotes.

Names representing tokens must be declared as follows in the declaration sections:
%token name1 name2...

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 no terminal symbols, one, called the start symbol has a particular importance. The parser is designed to recognize the start symbol. By default the start symbol is taken to be the left hand side of the first grammar rule in the rules section.

With each grammar rule, the user may associate actions to be. These actions may return values, and may obtain the values returned by the previous actions. Lexical analyzer can return values for tokens, if desired. An action is an arbitrary C statement. Actions are enclosed in curly braces.

CHAPTER 3

Introduction toUNIX

Basic UNIX commands

Folder/Directory Commands and Options

Action	UNIX options &file Spec
Check current Print Working Directory	**pwd**
Return to user's home folder	**cd**
Up one folder	**cd ..**
Make directory	**mkdir** projl
Remove empty directory	**rmdir**/usr/sam
Remove directory-recursively	**rm -r**

File Listing Commands and Options

Action	UNIX options &file
List directory tree-recursively	**ls -r**
List last access dates of files, with hidden files	**ls -l -a**
List files by reverse date	**ls -t -r *.***
List files verbosely by size of file	**ls -l -s *.***
List files recursively including contents of other directories	**ls -R *.***
List files with x anywherein the name	**ls | grep x**

File Manipulation Commands and Options

Action	UNIX options&filespec
Create new(blank)file	**touch** *afilename*
Copy old file to new file. -p preserve file attributes(e.g. ownership and edit dates)-rcopyrecursively through directory structure-a archive, combines the flags-p– Rand-d	Cpold.filenew.file
Move old.file(-**I** interactively flag prompts before over	**mv** –I old.file/tmp
Remove file(-intention)	**rm**–i sam.txt
Compare two files and show differences	**diff**

File Utilities

Action	UNIX options &filespec
View a file	**vifile.txt**
Concatenate files	**Cat file1 file2 to standard**
Counts-lines,-words, and- characters in a file	**wc**-l
Displays line-by-line differences between pairs of text files.	**diff**
calculator	**bc**
calendar for September, 1752 (when leap years began)	**Cal** 9 1752

Controlling program execution for C-shell

&	Run job in background
^c	Kill job in foreground
^z	Suspend job in foreground
Fg	Restart suspended job in foreground
Bg	Run suspended job in background
;	Delimit commands on same line
()	Group commands on same line
!	re-run earlier commands from history list
jobs	List current jobs

Controlling program input/output for C-shell

|	Pipe output to input
>	Redirect output to a storage file
<	Redirect input from a storage file
>>	Append redirected output to a storage file
tee	Copy input to both file and next program
script	Make file record of a terminal activity

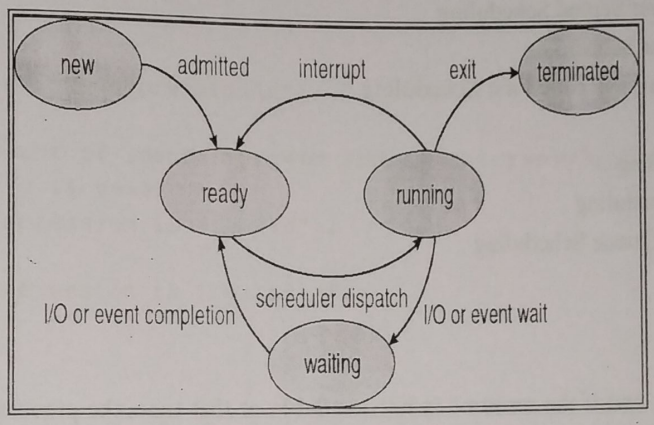
CHAPTER 4

Introduction to Operating Systems

An Operating System is a program that manages the Computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

A Process is a program in execution. As a process executes, it changes *state*

- **New** : The process is being created
- **Running** : Instructions are being executed
- **Waiting** : The process is waiting for some event to occur
- **Ready** : The process is waiting to be assigned to a process
- **Terminated** : The process has finished execution



Apart from the program code, it includes the current activity represented by

- Program Counter.
- Contents of Processor registers.
- Process Stack which contains temporary data like function parameters, return addresses and local variables.
- Data section which contains global variables
- Heap for dynamic memory allocation

A Multi-programmed system can have many processes running simultaneously with the CPU multiplexed among them. By switching the CPU between the processes, the OS can make the computer more productive. There is Process Scheduler which selects the process among many processes that are ready, for program execution on the CPU. Switching the CPU to another process requires performing a state save of the current process and a state restore of new process, this is Context Switch.

4.1 Scheduling Algorithms

CPU Scheduler can select processes from ready queue based on various scheduling algorithms. Different scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. The scheduling criteria include.

- CPU utilization:
- Throughput: The number of processes that are completed per unit time.
- Waiting time: The sum of periods spent waiting in ready queue.
- Turnaround time: The interval between the times of submission of process to the time of completion.
- Response time: The time from submission of a request until the first response is produced.

The different scheduling algorithms are

- FCFS: First Come First Served Scheduling
- SJF: Shortest Job First Scheduling
- SRTF: Shortest Remaining Time First Scheduling
- Priority Scheduling
- Round Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

4.2 Deadlocks

A process requests resources: and if the resource is not available at that time, the process enters a waiting state. Sometimes, a waiting process is never able to change state, because the resource is has requested is held by another process which is also waiting. This situation is called Deadlock.
Deadlock is characterized by four necessary conditions.

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

Deadlock can be handled in one of these ways,

- Deadlock Avoidance
- Deadlock Detection and Recover

1a. Write a LEX program to recognize valid arithmetic expression. Identifiers in the expression could be only integers and operators could be + and *. Count the identifiers & operators present and print them separately.

```
%{
    int id=0, op=0, v=0;
%}

%%
[a-zA-Z0-9]+    { id++; printf("\nIdentifier:"); ECHO; }
[\+\-\*\/\=]    { op++; printf("\nOperartor:"); ECHO;  }
"(" v++;
")" v--;
.|\n  ;
%%

int main()
{
    printf("Enter the expression\n");
    yylex();
    printf("\nCount of identifiers=%d and operators=%d",id,op);
    if((op+1)==id && v==0)
    printf("\nExpression is Valid\n");
    else
    printf("\nExpression is Invalid\n");
    return 0;
}
```

Execution Steps:

lex<lexfilename.l>
cc lex.yy.c –ll
./a.out

Output:

Enter the expression
(5+a)*45
Identifier:5
Operator:+
Identifier:a
Operator:*
Identifier:45
^D
Count of identifiers=3 and operators=2
Expression is Valid

1 b. Write YACC program to evaluate arithmetic expression involving operators: +, -, *, and /

Lex Part

```
%{
    #include "y.tab.h"
    extern yylval;
%}

%%
[0-9]+          { yylval=atoi(yytext); return num; }
[\+\-\*\/]      { returnyytext[0]; }
[)]       { return yytext[0]; }
[(]       { return yytext[0]; }
.               { ; }
\n              { return 0; }
%%
```

YACC Part

```
%{
    #include<stdio.h>
    #include<stdlib.h>
%}
%token num
%left '+''-'
%left '*''/'

%%
input:exp {printf("%d\n",$$);exit(0);}
exp:exp'+'exp {$$=$1+$3;}
|exp'-'exp{$$=$1-$3;}
|exp'*'exp{$$=$1*$3;}
|exp'/'exp { if($3==0){printf("Divide by Zero\n");exit(0);}
else $$=$1/$3;}
|'('exp')'{$$=$2;}
|num{$$=$1;};
%%

intyyerror()
{
    printf("error");
    exit(0);
}
```

```
int main()
{
    printf("Enter an expression:\n");
    yyparse();
}
```

Execution Steps:

yacc -d 1b.y
lex 1b.l
cc y.tab.clex.yy.c -ll
./a.out

Output:

Enter an expression:
(2+3)*5+9
34

2. Develop, Implement and execute a program using YACC tool to recognize all strings ending with preceded by *n a's* using the grammar $a^n b$ (note: input *n* value).

Lex Part

```
%{
        #include "y.tab.h"
%}
%%
a       { return A; }
b       { return B; }
[\n]    { return '\n' ; }
%%
```

YACC Part

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

%token A B

%%
input:s '\n' { printf("Successful Grammar\n");exit(0); }
s: A M B| B
M: A M | ;
%%

int main()
{
        printf("Enter a String\n");
        yyparse();
}
intyyerror()
{
        printf("Error \n");
        exit(0);
}
```

Output 1 :
Enter a string
aaab
Successful Grammar

Output 2 :
Enter A String
aaa
Error

3. **Design, develop and implement YACC/C program to construct *Predictive / LL(1) Parsing Table* for the grammar rules: $A \rightarrow aBa$, $B \rightarrow bB \mid \varepsilon$. Use this table to parse the sentence: *abba$*.**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char ip[20],stack[20];
int main()
{
  char m[2][3][3]={ {"aBa","E","E"}, {"n","bB","E"} };
  int size[2][3]= {3,1,1,1,2,1};
  inti,j,k,n,row,col,flag=0;
  intp,q,r;
  printf("\nEnter the input string: ");
  scanf("%s",ip);
  strcat(ip,"$");
  n=strlen(ip);
  stack[0]='$';
  stack[1]='A';
  i=1;
  j=0;
  printf("PARSING TABLE:\n");
  for(p=0;p<2;p++)
  {
       for(q=0;q<3;q++)
       {
            for(r=0;r<3;r++)
            printf("%c",m[p][q][r]);
            printf("\t");
       }
       printf("\n");
  }
  printf("\nStack\t\tInput\n");
  printf("_____\t_____\n");
  for(k=0;k<=i;k++)
  printf("%c",stack[k]); /* Initial stack*/
  printf("\t\t");
  for(k=j;k<=n;k++)
  printf("%c",ip[k]);
  printf("\n");
  while((stack[i]!='$')&&(ip[j]!='$')) //check stack and input both contains $
  {
       if(stack[i]==ip[j])
       //input and stack matches, discard stack and move to next input symbol
       {
            i--;
            j++;
            for(k=0;k<=i;k++)
            printf("%c",stack[k]); /*stack content */
         printf("\t\t");
```

```
        for(k=j;k<=n;k++)
        printf("%c",ip[k]);
        printf("\n");
    }
    switch(stack[i])
    {
        case 'A':
        row=0;
        break;
        case 'B':
        row=1;
        break;
        default:
        if ((stack[i]=='$')&&(ip[j]=='$'))
        printf("\nSUCCESSFULL PARSING\n");
        else
        {
            printf("\nUNSUCCESSFULL PARSING\n");
            printf("ERROR-NO VALID MATCH\n");
        }
        exit(0);
    }
    switch(ip[j])
    {
        case 'a':
        col=0;
        break;
        case 'b':
        col=1;
        break;
        case '$':
        col=2;
        break;
    }
    if(m[row][col][0]==ip[j]) //to check top of stack and input are equal
    {
        for(k=size[row][col]-1;k>=0;k--)
        // to replace non terminal by its production
        {
            stack[i]=m[row][col][k];
            i++;
        }
        i--;        // points to top of stack
    }
    if(m[row][col][0]=='E')  // to check error entry
    {
        if(i>0)
        printf("\nERROR....\n");
        else
        flag=1;
        exit(0);
```

```
            if(m[row][col][0]=='n') /*to check for epsilon*/
            i--;
            for(k=0;k<=i;k++)
            printf("%c",stack[k]); /* stack*/
            printf("\t\t");
            for(k=j;k<=n;k++)
            printf("%c",ip[k]); /* input*/
            printf("\n");
        }
        return 0;
}
```

Output 1 :

Enter the input string: abba$
PARSING TABLE:

aBa	E	E
n	bB	E

Stack	Input
$A	abba$$
$aBa	abba$$
$aB	bba$$
$aBb	bba$$
$aB	ba$$
$aBb	ba$$
$aB	a$$
$a	a$$
$	$$

SUCCESSFULL PARSING

Output 2 :

Enter the input string: aabba$
PARSING TABLE:

aBa	E	E
n	bB	E

Stack	Input
$A	aabba$$
$aBa	aabba$$
$aB	abba$$
$a	abba$$
$	bba$$

UNSUCCESSFULL PARSING
ERROR-NO VALID MATCH

4. Design, develop and implement YACC/C program to demonstrate *Shift Reduce Parsing* technique for the grammar rules: $E \rightarrow E+T \mid T$, $T \rightarrow T*F \mid F$, $F \rightarrow (E) \mid id$ and parse the sentence: *id + id * id*.

```
#include<stdio.h>
#include<string.h>
int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
{
   printf("GRAMMAR is:\nE->E+T|T \nT->T*F|F\nF->(E)|id\n");
   printf("Enter input string\n");
   scanf("%s",&a);
   c=strlen(a);
   printf("stack \t input \t action\n");
   printf("------\t ------\t ------\n");
   printf("\n$%s\t%s$\t",stk,a);
   for(k=0,i=0; j<c; k++,i++,j++)
   {
        if(a[j]=='i'&& a[j+1]=='d')
        {
             strcpy(act,"SHIFT ");
             stk[i]=a[j];
             stk[i+1]=a[j+1];
             stk[i+2]='\0';
             a[j]=' ';
             a[j+1]=' ';
             printf("\n$%s\t%s$\t%sid",stk,a,act);
             check();
        }
        else if(a[j]=='+'||a[j]=='*')
        {
             strcpy(act,"SHIFT ");
             stk[i]=a[j];
             stk[i+1]='\0';
             a[j]=' ';
             printf("\n$%s\t%s$\t%s%c",stk,a,act,stk[i]);
             check();
        }
        else
        {
             printf("\nERROR IN INPUT\n");
             break;
        }
   }
   if(stk[0]=='E'&& j==c)
   printf("\n****SUCCESSFULL PARSING****\n");
   else
   printf("\n****FAILURE IN PARSING****\n");
```

```
void check()
{
   strcpy(ac,"REDUCE F->id");
   for(z=0; z<c; z++)
   if(stk[z]=='i'&&stk[z+1]=='d')
   {
        stk[z]='F';
        stk[z+1]='\0';
        printf("\n$%s\t%s$\t%s",stk,a,ac);
        j++;
   }
   strcpy(ac,"REDUCE T->T*F");
   for(z=0; z<c; z++)
   if(stk[z]=='T'&&stk[z+1]=='*'&&stk[z+2]=='F')
   {
        stk[z]='T';
        stk[z+1]='\0';
        stk[z+2]='\0';
        printf("\n$%s\t%s$\t%s",stk,a,ac);
        i=i-2;
   }
   strcpy(ac,"REDUCE T->F");
   for(z=0; z<c; z++)

   if(stk[z]=='F' )
   {
        stk[z]='T';
        stk[z+1]='\0';
        printf("\n$%s\t%s$\t%s",stk,a,ac);
   }
   strcpy(ac,"REDUCE E->E+T");
   for(z=0; z<c; z++)
   {
   if(stk[z]=='E'&&stk[z+1]=='+'&&stk[z+2]=='T'&&stk[z+3]!='*'&& a[j+1]!='*')
   {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+2]='\0';
        stk[z+3]='\0';
        printf("\n$%s\t%s$\t%s",stk,a,ac);
        i=i-2;
   }
   else if(stk[z]=='E'&&stk[z+1]=='+'&&stk[z+2]=='T'&& j==c)
   {
        stk[z]='E';
        stk[z+1]='\0';
        stk[z+2]='\0';
        printf("\n$%s\t%s$\t%s",stk,a,ac);
        i=i-2;
   }
   else break;
   }
```

```
        strcpy(ac,"REDUCE E->T");
        for(z=0; z<c; z++)
        if(stk[z]=='T' )
        {
              if(a[j+1]=='+'||a[j+1]=='\0')
              {
                    stk[z]='E';
                    stk[z+1]='\0';
                    printf("\n$%s\t%s$\t%s",stk,a,ac);
              }
              else
              break;
        }
        strcpy(ac,"REDUCE F->(E)");
        for(z=0; z<c; z++)
        if(stk[z]=='('&&stk[z+1]=='E'&&stk[z+2]==')')
        {
              stk[z]='F';
              stk[z+1]='\0';
              stk[z+1]='\0';
              printf("\n$%s\t%s$\t%s",stk,a,ac);
              i=i-2;
        }
}
```

Output 1:

GRAMMAR is:
E->E+T|T
T->T*F|F
F->(E)|id
Enter input string
id+id*id

stack	input	action
------	------	------
S	id+id*idS	
Sid	+id*idS	SHIFT id
SF	+id*idS	REDUCE F->id
ST	+id*idS	REDUCE T->F
SE	+id*idS	REDUCE E->T
SE+	id*idS	SHIFT +
SE+id	*idS	SHIFT id
SE+F	*idS	REDUCE F->id
SE+T	*idS	REDUCE T->F
SE+T*	idS	SHIFT *
SE+T*id	S	SHIFT id
SE+T*F	S	REDUCE F->id
SE+T	S	REDUCE T->T*F
SE	S	REDUCE E->E+T

****SUCCESSFULL PARSING****

Output 2:

GRAMMAR is:
E->E+T|T
T->T*F|F
F->(E)|id
Enter input string
id+id/id

stack	input	action
------	------	------
S	id+id/idS	
Sid	+id/idS	SHIFT id
SF	+id/idS	REDUCE F->id
ST	+id/idS	REDUCE T->F
SE	+id/idS	REDUCE E->T
SE+	id/idS	SHIFT +
SE+id	/idS	SHIFT id
SE+F	/idS	REDUCE F->id
SE+T	/idS	REDUCE T->F
SE	/idS	REDUCE E->E+T

ERROR IN INPUT

****FAILURE IN PARSING****

5. Design, develop and implement a C/Java program to generate the machine code using *Triples* for the statement $A = -B * (C + D)$ whose intermediate code in three-address form:

T1 = -B
T2 = C + D
*T3 = T1 * T2*
A = T3

```
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
#include<string.h>

char tset[4][3][3]=   {     {"-","B","?"}, {"+","C","D"},
                      {"*","0","1"}, {"=","A","2"}
                      };

int main()

    introw,col;
    for(row=0;row<4;row++)
    {
        col=2;
        if (tset[row][col][0]=='?')
        {
            printf("\nLD R0,%s%s",tset[row][0],tset[row][1]);
        }
        else
        {
            if(tset[row][0][0]=='+')
            {
                printf("\nLD R1,%s",tset[row][1]);
                printf("\nLD R2,%s",tset[row][2]);
                printf("\nADD R1,R1,R2");
            }
            else
            {
                if(tset[row][0][0]=='*')
                {
                    printf("\nMUL R1,R1,R0");
                }
                else
                {
                    printf("\nST  %s,R1",tset[row][1]);
                }
            }
        }
    }
    printf("\n");
    return 0;
```

Output:
LD R0,-B
LD R1,C
LD R2,D
ADD R1,R1,R2
MUL R1,R1,R0
ST A,R1

6 a) Write a LEX program to eliminate *comment lines* in a *C* program and copy the resulting program into a separate file.

```
%%
"//".*
"/*"[^*/]*"*/"   ;              /*for single line comments*/
%%                              /*for multiple line comments*/

int main( )
{
    yyin=fopen("input.c","r");
    yyout=fopen("output.c","w");
    yylex();
}
```

input.c

```
/* C Program to add two integers */
#include <stdio.h>
void main()
{
int number1, number2, sum;

// inputting two integers
printf("Enter two integers: ");
scanf("%d %d", &number1, &number2);

// calculating sum
sum = number1 + number2;
printf(sum = %d",sum);
}
```

Output:

```
cat output.c

#include <stdio.h>
void main()
{
int number1, number2, sum;


printf("Enter two integers: ");
scanf("%d %d", &number1, &number2);


sum = number1 + number2;
printf(sum = %d",sum);
}
```

b) Write YACC program to recognize valid *identifier*, *operators* and *keywords* in the given text (C program) file.

Lex File

```
%{
    #include "y.tab.h"
    extern yylval;
%}

%%
[ \t]    ;
[+|-|*|/|=|<|>]  { printf("operator is %s\n",yytext);return OP; }
[0-9]+           { yylval = atoi(yytext); printf("numbers is %d\n",yylval);
                 return DIGIT; }
int|char|bool|float|void|for|do|while|if|else|return|void                    {
printf("keyword is%s\n",yytext);return KEY; }
[a-zA-Z0-9]+             { printf("identifier is %s\n",yytext);return ID; }
.      ;
%%
```

YACC File

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    int id=0, dig=0, key=0, op=0;
%}
%token DIGIT ID KEY OP
%%
input:
DIGIT input { dig++; }
| ID input { id++; }
| KEY input { key++; }
| OP input {op++;}
| DIGIT { dig++; }
| ID { id++; }
| KEY { key++; }
| OP { op++;}
;
%%


extern FILE *yyin;
int main()
{
    yyin=fopen("demo.c","r");
    if(!yyin)
    {
        printf("File not found");
        exit(0);
    }
    yyparse();
```

```
    printf("Identifier Count=%d, Digit Count=%d,Keyword    Count=%d,Operator
    Count=%d\n",id,dig,key,op);
    return 0;
}

void yyerror()
{
    printf("EEK, parse error! Message: ");
    exit(0);
}
```

demo.c

```
void main()
{
        int a=10;
        if(a==10)
        printf("True");
        else printf("False");
}
```

Output:

```
keyword is void
identifier is main


keyword is int
identifier is a
operator is =
numbers is 10

keyword is if
identifier is a
operator is =
operator is =
numbers is 10

identifier is printf
identifier is True

keyword is else
identifier is printf
identifier is False

Identifier Count=7, Digit Count=2,Keyword Count=4,Operator Count=3
```

7. Design, develop and implement a C/C++/Java program to simulate the working of *Shortest remaining time* and *Round Robin (RR)* scheduling algorithms. Experiment with different quantum sizes for RR algorithm.

Round-robin (RR) is one of the algorithms employed by process and network schedulers in computing.As the term is generally used, time slices (also known as time quanta) are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks. It is an operating system concept.

The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

```
#include<stdio.h>
intn,qtum,pid[10],a[10],b[10],tt[10],avg[10];
intswt=0,stat=0;
float f_avg_turn_arnd_time=0.0,f_avg_wait_time=0.0;

intrr()
{
    intch, st[10],i, count=0,temp,sq=0;
    printf("\n Enter the no of processes\n");
    scanf("%d",&n);
    printf("Enter burst time\n");
    for(i=0;i<n;i++)
    {
    pid[i]=i+1;
    scanf("%d",&b[i]);
  }
  printf("Enter the quantum:");
  scanf("%d",&qtum);
  for(i=0;i<n;i++)
  {
    st[i]=b[i];
    }
    while(1)
    {
          for(i=0;i<n;i++)
          {
                temp=qtum;
                if(st[i]==0)
                {
                      count++;
                      continue;
                }
                if(st[i]>qtum)
```

```
                st[i]=st[i]-qtum;
                else
                if(st[i]>=0)
                {
                      temp=st[i];
                      st[i]=0;
                }
                sq=sq+temp;
                tt[i]=sq;
           }
       if(n==count)
       break;
    }
    printf("\n Process id burst time wait time turnaround time \n");
    for(i=0;i<n;i++)
    {
          avg[i]=tt[i]-b[i];
          printf("%d\t\t%d\t%d\t%d",i+1,b[i],avg[i],tt[i]);
          printf("\n");
    }
    for(i=0;i<n;i++)
    {
          swt=swt+avg[i];
          stat=stat+tt[i];
    }
    f_avg_wait_time=(float)swt/n;
    f_avg_turn_arnd_time = (float)stat/n;
    printf("\n\nAverage waiting time is %f\n Average turnaround time is
    %f",f_avg_wait_time, f_avg_turn_arnd_time);
    return 1;
}

intsrtf()
{
    intch, a[10],b[10],x[10],i,smallest, count=0,time=0,n;
    double avg=0,tt=0,end;
    printf("Enter the number of processes:\n");
    scanf("%d",&n);
    printf("Enter arrival time\n");
    for(i=0;i<n;i++)
    scanf("%d",&a[i]);
    printf("Enter the burst time \n");
    for(i=0;i<n;i++)
    scanf("%d",&b[i]);
    for(i=0;i<n;i++)
    x[i]=b[i];
    b[9]=9999;
```

```
    for(time=0;count!=n;time++)
    {
        smallest=9;
        for(i=0;i<n;i++)
        {
            if(a[i]<=time && b[i]<b[smallest] && b[i]>0)
            smallest=i;
        }
        b[smallest]--;
        if(b[smallest]==0)
        {
            count++;
            end=time+1;
            avg=avg+end-a[smallest]-x[smallest];
            tt=tt+end-a[smallest];
        }
    }
    printf("\n Average waiting time = %lf\n",avg/n);
    printf("Average turnaround time = %lf",tt/n);
    return 0;
}

int main()
{
    inti, ch;
    printf("1:SRTF\n");
    printf("2:RR\n");
    printf("3:exit\n");
    printf("Enter the choice\n");
    scanf("%d",&ch);
    switch(ch)
    {
        case 1: srtf();break;
        case 2: rr(); break;
        case 3: return 1;
    }
    return 1;
}
```

Output :

1: SRTF
2: RR
3: Exit
Enter the choice
1
Enter the number of processes:
4
Enter arrival time
0
1
2
3
Enter the burst time
8
4
9
5

8. Design, develop and implement a C/C++/Java program to implement *Banker's algorithm*. Assume suitable input required to demonstrate the results.

The **Banker's algorithm**, sometimes referred to as the **detection algorithm**, is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an "s-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

The algorithm was developed in the design process for the operating system and originally described in EWD108. When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

```
#include<stdio.h>
struct process
{
inti_all[6],i_max[6],i_need[6],i_finished;
} p[10];

inti_avail[6], i_sseq[10], i_ss=0, i_check1=0, i_check2=0, n, i_work[6];
inti_nor;

void main()
{
  intsafeseq(void);
  inttj,ch,i=0,j=0,k,ch1;
  printf("Enter number of processes : ");
  scanf("%d",&n);
  printf("Enter the Number of Resources : ");
  scanf("%d", &i_nor);
  printf("Enter the Available Resources : ");
  for(j=0;j<n;j++)
  {
       for(k=0; k<i_nor; k++)
       {
            if(j==0)
            {
                 printf("\n For Resource type %d : ",k);
                 scanf("%d", &i_avail[k]);
            }
           p[j].i_max[k]=0;
           p[j].i_all[k]=0;
           p[j].i_need[k]=0;
           p[j].i_finished=0;
           p[j].i_request[k]=0;
       }
  }
```

```
printf("\n Enter Max resources for all processes\n");
for(i=0;i<n;i++)
{
      for(j=0; j<i_nor; j++)
      {
            scanf("%d",&p[i].i_max[j]);
      }
}
printf("\n Enter Allocated resources for all processes\n");
for(i=0;i<n;i++)
{
      for(j=0;j<i_nor;j++)
      {
            scanf("%d",&p[i].i_all[j]);
            if(p[i].i_all[j]>p[i].i_max[j])
            {
                  printf("\n Allocation should be less < or == max");
                  j--;
            }
            else
            p[i].i_need[j]=p[i].i_max[j]-p[i].i_all[j];
      }
}
if(safeseq()==1)
{
      printf("\n The System is in Safe state\n ");
}
else
printf("\n The System is Not in safe state\n ");
printf("\n Need\n");
for(i=0;i<n;i++)
{
      for(j=0;j<i_nor;j++)
      printf(" %d ",p[i].i_need[j]);
      printf("\n");
}


ntsafeseq()

{
inttk,tj,i,j,i_ss;
i_ss=0;
for(j=0; j<i_nor; j++)
i_work[j] = i_avail[j];
for(j=0;j<n;j++)
p[j].i_finished=0;
for(tk=0; tk<i_nor; tk++)
{
      for(j=0;j<n;j++)
      {
            if(p[j].i_finished==0)
            {
```

```
                    i_check1=0;
                    for(k=0; k<i_nor; k++)
                    if(p[j].i_need[k]<=i_work[k])
                    i_check1++;
                    if(i_check1== i_nor)
                    {
                          for(k=0;k<i_nor;k++)
                          {
                                i_work[k]= i_work[k]+p[j]. i_all[k];
                                p[j]. i_finished=1;
                          }
                          i_sseq[i_ss]=j;
                          i_ss++;
                    }
               }
          }
     }
     i_check2=0;
     for(i=0;i<n;i++)
     if(p[i].i_finished==1)
     i_check2++;
     if(i_check2>=n)
     {
        printf("The Safe Sequence is\t :");
        for(tj=0;tj<n;tj++)
        printf("%d ", i_sseq[tj]);
        return 1;
     }
     return 0;
}
```

Output :

```
Enter number of processes: 5
Enter the number of Resources: 3
Enter the Available Resources:
For Resource type 0: 3
For Resource type 1: 3
For Resource type 2: 2
Enter Max resources for all processes
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
Enter Allocated resources for all processes
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
The safe sequence is : 1,3,4,0,2
The System is in Safe state
Need
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1
```

9. Design, develop and implement a C/C++/Java program to implement *page replacement algorithms LRU* and *FIFO*. Assume suitable input required to demonstrate the results.

In a computer operating system that uses paging for virtual memory management, **page replacement algorithms** decide which memory pages to page out, sometimes called swap out, or write to disk, when a page of memory needs to be allocated. Page replacement happens when a requested page is not in memory (page fault) and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.
When the page that was selected for replacement and paged out is referenced again it has to be paged in (read in from disk), and this involves waiting for I/O completion. This determines the *quality* of the page replacement algorithm: the less time waiting for page-ins, the better the algorithm. A page replacement algorithm looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself.
The page replacing problem is a typical online problem from the competitive analysis perspective in the sense that the optimal deterministic algorithm is known.

```
#include<stdio.h>
#include<stdlib.h>
void FIFO(char [],char [],int,int);
void lru(char [],char [],int,int);
int main()
{
   intch,YN=1,i,l,f;
   char F[10],s[25];
   printf("\n\nEnter the no of empty frames: ");
   scanf("%d",&f);
   printf("\n\nEnter the length of the string: ");
   scanf("%d",&l);
   printf("\n\nEnter the string: ");
   scanf("%s",s);
   for(i=0;i<f;i++)
   F[i]=-1;
   do
   {
        printf("\n\n\t********** MENU **********");
        printf("\n\n\t1:FIFO\n\n\t2:LRU \n\n\t3:EXIT");
        printf("\n\n\tEnter your choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
             case 1:
             for(i=0;i<f;i++)
             {
             F[i]=-1;
        }
```

```
        FIFO(s,F,l,f);
        break;
        case 2:
        for(i=0;i<f;i++)
        {
             F[i]=-1;
        }
        lru(s,F,l,f);
        break;
        case 3:
        exit(0);
    }
    printf("\n\n\tDo u want to continue IF YES PRESS 1\n\n\tIF NO PRESS 0:");
    scanf("%d",&YN);
    }while(YN==1);
    return(0);
}

//FIFO
void FIFO(char s[],char F[],intl,int f)
{
    inti,j=0,k,flag=0,cnt=0;
    printf("\n\tPAGE\t FRAMES\t FAULTS");
    for(i=0;i<l;i++)
    {
        for(k=0;k<f;k++)
        {
            if(F[k]==s[i])
            flag=1;
        }
        if(flag==0)
        {
            printf("\n\t%c\t",s[i]);
            F[j]=s[i];
            j++;
            for(k=0;k<f;k++)
            {
                printf(" %c",F[k]);
            }
            printf("\tPage-fault%d",cnt);
            cnt++;
        }
        else
        {
            flag=0;
            printf("\n\t%c\t",s[i]);
            for(k=0;k<f;k++)
            {
                printf(" %c",F[k]);
            }
            printf("\tNo page-fault");
        }
```

```
        if(j==f)
        j=0;
        }

}


//LRU
void lru(char s[],char F[],int l,int f)
{
    int i,j=0,k,m,flag=0,cnt=0,top=0;
    printf("\n\tPAGE\t FRAMES\t FAULTS");
    for(i=0;i<l;i++)
    {
        for(k=0;k<f;k++)
        {
            if(F[k]==s[i])
            {
                flag=1;
                break;
            }
        }
        printf("\n\t%c\t",s[i]);
        if(j!=f && flag!=1)
        {
            F[top]=s[i];
            j++;
            if(j!=f)
            top++;
        }
        else
        {
            if(flag!=1)
            {
                for(k=0;k<top;k++)
                {
                    F[k]=F[k+1];
                }
                F[top]=s[i];
            }
            if(flag==1)
            {
                for(m=k;m<top;m++)
                {
                    F[m]=F[m+1];
                }
                F[top]=s[i];
            }
        }
        for(k=0;k<f;k++)
        {
            printf("%c",F[k]);
        }
```

```
        if(flag==0)
        {
              printf("\tPage-fault%d",cnt);
              cnt++;
        }
        else
        printf("\tNo page fault");
        flag=0;
    }
}
```

Output :

```
Enter the no of empty frames: 3
Enter the length of the string: 5
Enter the string: hello
********** MENU **********
1:FIFO
2:LRU
3:EXIT
Enter your choice: 1
PAGE FRAMES FAULTS
H h Page-fault0
E h e Page-fault1
L h e l Page-fault2
L h e l No page-fault
O o e l Page-fault3
Do u want to continue IF YES PRESS 1
IF NO PRESS 0 : 1
********** MENU **********
1:FIFO
2:LRU
3:EXIT
Enter your choice: 2
PAGE FRAMES FAULTS
h h Page-fault0
e h e Page-fault1
l h e l Page-fault2
l h e l No page fault
o e l o Page-fault3
Do u want to continue IF YES PRESS 1
IF NO PRESS 0 : 1
********** MENU **********
1:FIFO
2:LRU
3:EXIT
Enter your choice: 3
```

CHAPTER 5

Additional Experiments

Lex program to count the number of words,small and capital letters, digits and special characters in a C File

```
%{
  #include<stdio.h>
  int lines=0, words=0,s_letters=0,c_letters=0, num=0, spl_char=0,total=0;
%}

%%
\n { lines++; words++;}
[\t ' '] words++;
[A-Z] c_letters++;
[a-z] s_letters++;
[0-9] num++;
. spl_char++;
%%

main(void)
{
yyin= fopen("abc.txt","r");
yylex();
total=s_letters+c_letters+num+spl_char;
printf(" This File contains ...");
printf("\n\t%d lines", lines);
printf("\n\t%dwords",words);
printf("\n\t%d small letters", s_letters);
printf("\n\t%d capital letters",c_letters);
printf("\n\t%d digits", num);
printf("\n\t%d special characters",spl_char);
printf("\n\tIn total %d characters.\n",total);
}

intyywrap()
{
return(1);
}
```

LEX program to count the no of "scanf" and "printf" statement in a c program. Replace them with "writf" and "readf" respectively.

```
%{
#include<stdio.h>
int sf=0,pf=0;
%}

%%
"scanf"    { sf++; fprintf(yyout,"readf");}
// replace scanf  withreadf
"printf"           { pf++; fprintf(yyout,"writef");}
// replace printf  withwritef

%%
int main()
{
     yyin=fopen("open.c","r");
// input file open.c
     yyout=fopen("new.c","w");
// output file new.c with replace
     yylex();
//no of printf and scanf in the file
     printf("Number of scanfs=%d\nNumber of Printf's=%d\n",sf,pf);

return 0;
}
```

CHAPTER 6

Viva Questions

1. **Define system software.**
   System software is computer software designed to operate the computer hardware and to provide a platform for running application software. Eg: operating system, assembler, and loader.
2. **Define compiler and interpreter.**
   Compiler is a set of programs which converts the whole high level language program to machine language program. Interpreter is a set of programs which converts high level language program to machine language program line by line.
3. **What is an Assembler?**
   Assembler for an assembly language, a computer program to translate between lower-level representations of computer programs.
4. **Define the basic functions of assembler.**
   Translating mnemonic operation codes to their machine language equivalents. Assigning machine addresses to symbolic labels used by the programmer.
5. **What is the need of SYMTAB (symbol table) in assembler?**
   The symbol table includes the name and value for each symbol in the source program, together with flags to indicate error conditions. Sometimes it may contain details about the data area. SYMTAB is usually organized as a hash table for efficiency of insertion and retrieval.
6. **What is the need of OPTAB (operation code table) in assembler?**
   The operation code table contains the mnemonic operation code and its machine language equivalent. Some assemblers it may also contain information about instruction format and length. OPTAB is usually organized as a hash table, with mnemonic operation code as the key.
7. **Define relocatable program.**
   An object program that contains the information necessary to perform required modification in the object code depends on the starting location of the program during load time is known as relocatable program.
8. **What is meant by external references?**
   Assembler program can be divided into many sections known as control sections and each control section can be loaded and relocated independently of the others. If the instruction in one control section need to refer instruction or data in another control section. The assembler is unable to process these references in normal way. Such references between control are called external references.
9. **Define control section.**
   A control section is a part of the program that maintains its identity after assembly; each control section can be loaded and relocated independently of the others. Control sections are most often used for subroutines. The major benefit of using control sections is to increase flexibility.
10. **Define absolute loader.**
    The loader, which is used only for loading, is known as absolute loader. e.g. Bootstrap loader
11. **What is meant by bootstrap loader?**
    This is a special type of absolute loader which loads the first program to be run by the computer.

(Usually an operating system)

12. What are relative (relocative) loaders?

Loaders that allow for program relocation are called relocating (relocative) loaders.

13. Explain lex and yacc tools.

Lex: - scanner that can identify those tokens.

Yacc: - parser.yacc takes a concise description of a grammar and produces a C routine that can parse that grammar.

14. Explain yyleng?

Yyleng-contains the length of the string our lexer recognizes.

15. What Is A Symbol Table?

A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier. The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly. Whenever an identifier is detected by a lexical analyser, it is entered into the symbol table. The attributes of an identifier cannot be determined by the lexical analyser.

16. Mention Some of the cousins of a Compiler.

- Cousins of the compiler are:
- Preprocessors
- Assemblers
- Loaders and Link-Editors

17. What is a Parser?

A Parser for a Grammar is a program which takes in the Language string as it's input and produces either a corresponding Parse tree or an Error.

18. What is the Syntax of a Language?

The Rules which tell whether a string is a valid Program or not are called the Syntax.

19. What is the Semantics of a Language?

The Rules which give meaning to programs are called the Semantics of a Language.

20. What are tokens?

When a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the Language.

21. What is the Lexical Analysis?

The Function of a lexical Analyzer is to read the input stream representing the Source program, one character at a time and to translate it into valid tokens.

22. How can we represent a token in a language?

The Tokens in a Language are represented by a set of Regular Expressions. A regular expression specifies a set of strings to be matched. It contains text characters and operator characters. The Advantage of using regular expression is that a recognizer can be automatically generated.

23. How are the tokens recognized?

The tokens which are represented by a Regular Expressions are recognized in an input string by means of a state transition Diagram and Finite Automata.

24. Are Lexical Analysis and Parsing two different Passes?

These two can form two different passes of a Parser. The Lexical analysis can store all the recognized tokens in an intermediate file and give it to the Parser as an input. However, it is more convenient to have the lexical Analyzer as a co routine or a subroutine which the Parser calls whenever it requires a token.

25. How do we write the Regular Expressions?

The following are the most general notations used for expressing a R.E.

Symbol	Description
|	OR (alternation)
()	Group of Subexpression
*	0 or more Occurrences
?	0 or 1 Occurrence
+	1 or more Occurrences

26. What are the Advantages of using Context-Free grammars?

It is precise and easy to understand.

It is easier to determine syntactic ambiguities and conflicts in the grammar.

27. If Context-free grammars can represent every regular expression, why do one needs R.E at all?

Regular Expression are Simpler than Context-free grammars. It is easier to construct a recognizer for R.E than Context-Free grammar. Breaking the Syntactic structure into Lexical & non-Lexical parts provide better front end for the Parser. R.E are most powerful in describing the lexical constructs like identifiers, keywords etc while Context-free grammars in representing the nested or block structures of the Language.

28. What are the Parse Trees?

Parse trees are the Graphical representation of the grammar which filters out the choice for replacement order of the Production rules.

29. What are Terminals and non-Terminals in a grammar?

Terminals: - All the basic symbols or tokens of which the language is composed of are called Terminals. In a Parse Tree the Leaf's represents the Terminal Symbol.

Non-Terminals: - These are syntactic variables in the grammar which represents a set of strings the grammar is composed of. In a Parse tree all the inner nodes represent the Non-Terminal symbols.

30. What are Ambiguous Grammars?

A Grammar that produces more than one Parse Tree for the same sentences or the Production rules in a grammar is said to be ambiguous.

31. What is bottom-up Parsing?

The Parsing method is which the Parse tree is constructed from the input language string beginning from the leaves and going up to the root node.

Bottom-Up parsing is also called shift-reduce parsing due to its implementation. The YACC supports shift-reduce parsing.

32. What is the need of Operator precedence?

The shift reduce Parsing has a basic limitation. Grammars which can represent a left-sentential parse tree as well as right-sentential parse tree cannot be handled by shift reduce parsing. Such a grammar ought to have two non-terminals in the production rule. So, the Terminal sandwiched between these two non-terminals must have some associability and precedence. This will help the parser to understand which non-terminal would be expanded first.

33. What is exit status command?

Exit 0- return success, command executed successfully.

Exit 1 – return failure.

34. Define API's

An application programming interface (API) is a source code-based specification intended to be used as an interface by software components to communicate with each other.

35. Explain the main purpose of an operating system?

Operating systems exist for two main purposes. One is that it is designed to make sure a computer system performs well by managing its computational activities. Another is that it provides an environment for the development and execution of programs.

36. What is demand paging?

Demand paging is referred when not all of a process's pages are in the RAM, then the OS brings the missing (and required) pages from the disk into the RAM.

37. Briefly explain FCFS.

FCFS stands for First-come, first-served. It is one type of scheduling algorithm. In this scheme, the process that requests the CPU first is allocated the CPU first. Implementation is managed by a FIFO queue.

38. What is RR scheduling algorithm?

RR (round-robin) scheduling algorithm is primarily aimed for time-sharing systems. A circular queue is a setup in such a way that the CPU scheduler goes around that queue, allocating CPU to each process for a time interval of up to around 10 to 100 milliseconds.

39. What are necessary conditions which can lead to a deadlock situation in a system?

Deadlock situations occur when four conditions occur simultaneously in a system: Mutual exclusion; Hold and Wait; No preemption; and Circular wait.

40. Describe Banker's algorithm

Banker's algorithm is one form of deadlock-avoidance in a system. It gets its name from a banking system wherein the bank never allocates available cash in such a way that it can no longer satisfy the needs of all of its customers.

41. Give an example of a Process State.

- New State – means a process is being created.
- Running – means instructions are being executed.
- Waiting – means a process is waiting for certain conditions or events to occur.
- Ready – means a process is waiting for an instruction from the main processor.
- Terminate – means a process is stopped abruptly.