_____

INTRODUCTION TO LEX

Lex and YACC helps you write programs that transforms structured input. Lex generates C code for lexical analyzer whereas 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:

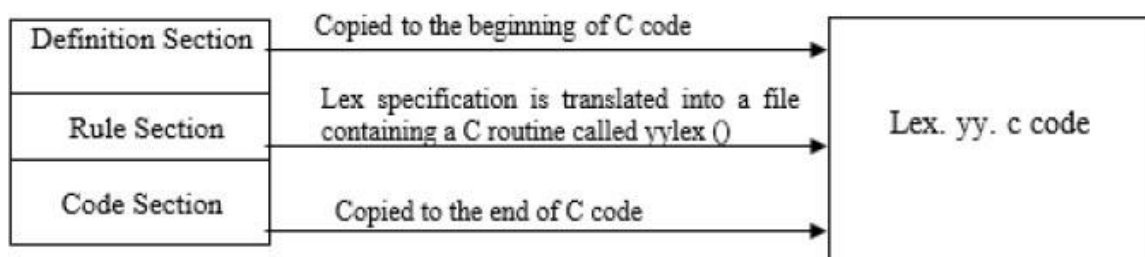**1st Step:** Using gedit create a file with extension l. For example: prg1.l

2nd Step: lex prg1.l

3rd Step: cc lex.yy.c –ll

4th Step: ./a.out

1.2 Structure of LEX source program:

```
{definitions}
%%
{rules}
%%
{user subroutines/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 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. 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)*	

_____

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.
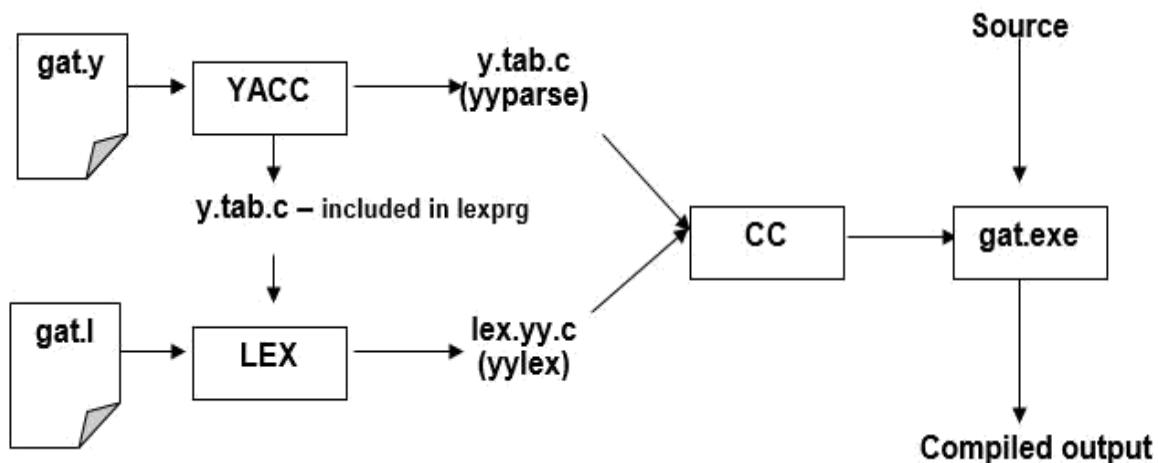
2.1 Steps in writing YACC Program:

1st Step: Using gedit editor create a file with extension y. For example: gedit prg1.y

2nd Step: YACC –d prg1.y

3rd Step: lex prg1.l

4th Step: cc y.tab.c lex.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.

> {definitions}
> %%
> {rules}
> %%
> {user subroutines}

%% 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 nonterminal 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.

_____

INTRODUCTION TO UNIX

Basic UNIX commands

Folder/Directory Commands and Options

Action	UNIX options & filespec
Check current Print Working Directory	**Pwd**
Return to user's home folder	**Cd**
Up one folder	**cd ..**
Make directory	**mkdir** proj1
Remove empty directory	**rmdir**/usr/sam
Remove directory-recursively	**rm –r**

File Listing Commands and Options

Action	UNIX options & filespec
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 number of lines in folder	**wc -l *.xtumlsed -n '$='**
List files with x anywhere in 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)-r copy recursively through directory structure -a archive, combines the flags-p – R and-d	**cp** old.filenew.file
Move old.file(**-i** interactively flag prompts before overwriting files)	**mv** –i old.file/tmp
Remove file(-intention)	**rm** –i sam.txt
View a file	**vi file.txt**
Concatenate files	**cat file1file2 to standard output.**
Counts-lines,-words, and- characters in a file	**wc** -l

_____

_____

INTRODUCTION TO OPERATING SYSTEMS

Introduction

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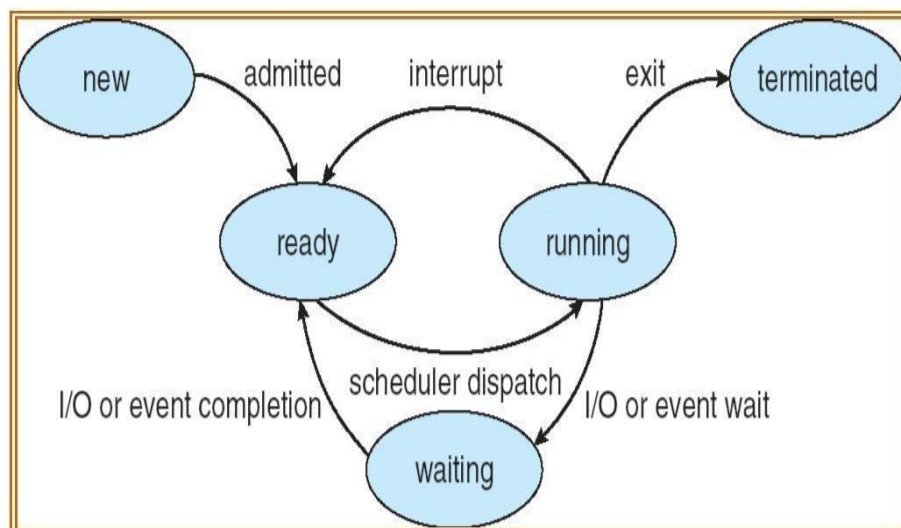
Apart from the program code, it includes the current activity represented by

➢ 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 time 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

Shortest remainingtime **scheduling algorithm:**

Shortest remaining time, also known as shortest remaining time first (SRTF), is aschedulingmethod that is apreemptiveversion ofshortest job nextscheduling. In this scheduling algorithm, theprocesswith the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute.

Like shortest job first, it has the potential forprocess starvation; long processes may be held off indefinitely if short processes are continually added.

Round Robin (RR) **scheduling algorithm:**

Round-robin (RR) is one of the algorithms employed by process and networkschedulers 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.

Banker's algorithm:

_____

_____

The **Banker's algorithm**, sometimes referred to as the **detection algorithm**, is a resourceallocation 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 Dutch) 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.

Page replacement algorithms LRU and *FIFO:*

In a computer operating system that uses paging for virtual memory management, **pagereplacement algorithms** decide which memory pages to page out, sometimes called swap out, or writeto 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.

_____

INTRODUCTION TO COMPILER DESIGN

A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the lowlevel machine language that is required by computers.

The phases of a compiler:

Lexical analysis: This is the initial part of reading and analysing the program text: The text is read and divided into tokens, each of which corresponds to a symbol in the programming language, e.g., a variable name, keyword or number.

Syntax analysis: This phase takes the list of tokens produced by the lexical analysis and arranges these in a tree-structure (called the syntax tree) that reflects the structure of the program. This phase is often called parsing.

Type checking: This phase analyses the syntax tree to determine if the program violates certain consistency requirements, e.g., if a variable is used but not declared or if it is used in a context that does not make sense given the type of the variable, such as trying to use a boolean value as a function pointer.

Intermediate code generation: The program is translated to a simple machine independent intermediate language.

Register allocation: The symbolic variable names used in the intermediate code are translated to numbers, each of which corresponds to a register in the target machine code.

Machine code generation: The intermediate language is translated to assembly language (a textual representation of machine code) for a specific machine architecture.

Assembly and linking: The assembly-language code is translated into binary representation and addresses of variables, functions, etc., are determined.

Parsing:

A parser is a compiler or interpreter component that breaks data into smaller elements for easy translation into another language. A parser takes input in the form of a sequence of tokens or program instructions and usually builds a data structure in the form of a parse tree or an abstract syntax tree.

A parser's main purpose is to determine if input data may be derived from the start symbol of the grammar.

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing.
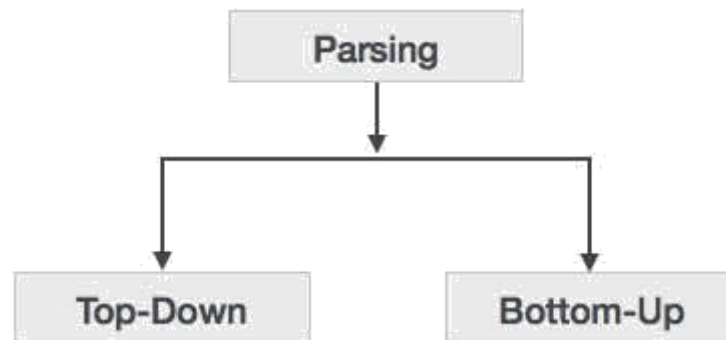
Top-Down Parsing:

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

- **Recursive descent parsing**: It is a common form of top-down parsing. It is called recursive as it usesrecursive procedures to process the input. Recursive descent parsing suffers from backtracking.

- **Backtracking**: It means, if one derivation of a production fails, the syntax analyzer restarts the processusing different rules of same production. This technique may process the input string more than once to determine the right production.

Shift Reduce Parsing/Bottom up parsing:

Bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol. Bottom up parsing can be defined as an attempt to reduce the input string „w" to the start symbol of a grammar by tracing out the rightmost derivations of „w" in reverse.

Shift-reduce Parsing (Bottom-up Parsing)

Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. In other words, it is a process of "reducing" (opposite of deriving a symbol using a production rule) a string w to the start symbol of a grammar. At every (reduction) step, a particular substring matching the RHS of a production rule is replaced by the symbol on the LHS of the production.

A general form of shift-reduce parsing is **LR** (scanning from **L**eft to right and using **R**ight-most derivation in reverse) parsing, which is used in a number of automatic parser generators like Yacc, Bison, etc.

Intermediate code/ three address code:

Three-address code(often abbreviated to TAC or 3AC) is anintermediate codeused byoptimizing compilersto aid in the implementation ofcode-improving transformations. Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator. For example, t1 = t2 + t3. The name derives from the use of three operands in these statements even though instructions with fewer operands may occur.

Since three-address code is used as an intermediate language within compilers, the operands will most likely not be concrete memory addresses orprocessor registers, but rather symbolic addresses that will be translated into actual addresses duringregister allocation. It is also not uncommon that operand names are numbered sequentially since three-address code is typically generated by the compiler.

Example: One solution to the quadratic equation using three address code is as below.
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a) t1 = b * b t2 = 4 * a t3 = t2 * c t4 = t1 - t3 t5 = sqrt(t4) t6 = 0 - b t7 = t5 + t6 t9 = t7 / t8 x = t9 t8 = 2 * a

LAB PROGRAMS

1(a)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.

```
%{
#include<stdio.h>
#include<string.h>
int nooper=0,nooperand=0,top=0,i=0,j=0,tnooper=0,tnoopnd=0,valid=0;
char opnd[10][10],opert[10][10];
%}
%%
"(" { valid =1;}
")" { valid=0;}
"+"|"*" {nooper++; strcpy(opert[i],yytext);i++;}
[0-9]+ {nooperand++; strcpy(opnd[j],yytext);j++;}
[^+*]"("[^0-9]   YYFAIL();
%%

int main()
{
        int k;
        printf("Enter the expression\n");
        yylex();
        if(valid==0 &&(nooperand-nooper)==1)
        {
                printf("The exp is valid\n");
                printf("The operator are\n");
                for(k=0;k<i;k++)
                printf("%s\n",opert[k]);
                printf("The operands are\n");
                for(k=0;k<j;k++)
                printf("%s\n",opnd[k]);
        }
        else{
                printf("The exp is invalid");
                return 0;
            }
}

int YYFAIL()
{
        printf("Invalid");
        exit(0);
}
```

/* __OUTPUT__

[root@localhost ss]# lex 1a.l
[root@localhost ss]# cc lex.yy.c -ll
[root@localhost ss]# ./a.out
Enter the expression
2+3*4

The exp is valid
The operator are
+
*
The operands are
2
3
4

[root@localhost ss]# ./a.out
enter the expression
(2+3)()
Invalid[root@localhost ss]# ./a.out
Enter the expression
(2+3)*()

The exp is invalid

[root@localhost ss]# ./a.out
Enter the expression
((2+3)*(4+5))

The exp is valid
The operator are
+
*
+
The operands are
2
3
4
5
*/

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

LEX PART

```
%{
#include "y.tab.h"
#include <stdlib.h>
extern int yylval;
%}

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

YACC PART

```
%{
#include<stdio.h>
int valid=1;
%}
%token NUM
%left '+' '-'
%left '*' '/'
%%
Stmt:expr{if(valid)
                {
                        printf("Result=%d\n",$$);}
                }
expr:expr'+'expr {$$=$1+$3;}
   |expr'-'expr {$$=$1-$3;}
   |expr'*'expr {$$=$1*$3;}
   |expr'/'expr {if($3==0){
                        valid=0;printf("Divide by zero error\n");}
                else
                        $$=$1/$3;}
   |'('expr')' {$$=$2;}
   |NUM {$$=$1;}
   ;
%%

main()
{
        printf("Enter arithmetic exression:\n");
        yyparse();
        if(valid==1)
                printf("Expression is valid\n");
```

```
}

int yyerror()
{
        printf("Invalid expression\n");
        exit(0);
}


/*  OUTPUT

[root@localhost ss]# lex 1b.l
[root@localhost ss]# yacc -d 1b.y
[root@localhost ss]# cc lex.yy.c y.tab.c -ll
[root@localhost ss]# ./a.out
Enter arithmetic exression:
2+3*6
Result=20
Expression is valid

[root@localhost ss]# ./a.out
Enter arithmetic exression:
2+3/0
Divide by zero error

[root@localhost ss]# ./a.out
Enter arithmetic exression:
(2*8)/2
Result=8
Expression is valid
*/
```

2. **Develop, Implement and Execute a program using YACC tool to recognize all strings ending with *b* 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 0;}
. {return yytext[0];}
%%
```

YACC PART

```
%{
#include <stdio.h>
int aCount=0,n;
%}
%token A
%token B
%%
s : X B {   if (aCount<n || aCount>n)
            {
        YYFAIL();
}
 }
X : X T | T
T : A { aCount++;}
 ;
%%

int main()
{       printf("Enter the value of n \n");
scanf("%d",&n);
        printf("Enter the string\n");
        yyparse();
        printf("Valid string\n");
}

int YYFAIL()
{
        printf("Invalid count of 'a'\n");
        exit(0);
}

int yyerror()
{
```

```
        printf("Invalid string\n");
        exit(0);
}
```

/* **OUTPUT**

```
[root@localhost ss]# lex 2.l
[root@localhost ss]# yacc -d 2.y
[root@localhost ss]# cc lex.yy.c y.tab.c -ll
[root@localhost ss]# ./a.out
Enter the value of n
2Enter the string
aab
Valid string

[root@localhost ss]# ./a.out
Enter the value of n
2Enter the string
aaab
Invalid Count of 'a'

[root@localhost ss]# ./a.out
Enter the value of n
2Enter the string
aabb
Invalid string
```

*/

3. **Design, develop and implement YACC/C program to construct** *Predictive / LL(1)* *ParsingTable* **for the grammar rules:** *A →aBa , B →bB | ε*. **Use this table to parse the sentence:** *abba$*.

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int n,i,j,k,count;
char grm[10][20], fst[10][20], fol[10][20], tble[3][4], inp[20], inpt[20],
          mch[20], stk[20];
void firstSet();
void followSet();
void parsingTable();
void parseInput();
void print(char* s);
void main() {
      setbuf(stdout,NULL);
      printf("The Given Grammar is : \n");
      printf("A->aBa\nB-bB|@\n");
      printf("Enter The Number Of Rules : ");
      scanf("%d", &n);
      printf("Enter The Rules : \n");
      for (i = 0; i < n; i++)
            scanf("%s", grm[i]);
      firstSet();
      followSet();
      parsingTable();
      parseInput();
}
void firstSet() {
      printf("The First Set Is : \n");
      for (i = 0; i < n; i++) {
            count = 0;
            j = 3;
            printf("FIRST[%c]={", grm[i][0]);
            while (grm[i][j] != '\0') {
                  if (!(grm[i][j] >= 65 && grm[i][j] <= 90)) {
                        fst[i][count++] = grm[i][j];
                        printf("%c,", grm[i][j]);
                  }
                  while (grm[i][j] != '|' && grm[i][j] != '\0')
                        j++;
                  j++;
            }
            printf("\b}\n");
      }
}
void followSet() {
      printf("The Follow Set Is : \n");
      for (k = 0; k < n; k++) {
            count = 0;
            printf("FOLLOW[%c]={", grm[k][0]);
            if (k == 0) {
                  printf("$,");
                  fol[k][count++] = '$';
            }
            for (i = 0; i < n; i++) {
                  for (j = 3; grm[i][j] != '\0'; j++) {
```

```c
                            if (grm[i][j] == grm[k][0] && grm[i][j + 1] != '\0'
                                    && grm[i][j + 1] != '|') {
                                if (!(grm[i][j + 1] >= 65 && grm[i][j + 1] <= 90))
{
                                    printf("%c,", grm[i][j + 1]);
                                    fol[k][count++] = grm[i][j + 1];
                                }
                            }
                        }
                    }
                }
                printf("\b}\n");
            }
        }
}
void parsingTable() {
        char p[10], q[10], r[10], f;
        strcpy(p, "A->aBa");
        strcpy(q, "B->bB");
        strcpy(r, "B->@");
        tble[1][0] = 'A';
        tble[2][0] = 'B';
        tble[0][1] = 'a';
        tble[0][2] = 'b';
        tble[0][3] = '$';
        for (i = 0; i < n; i++) {
                for (j = 0; fst[i][j] != '\0'; j++) {
                        f = fst[i][j];
                        if (f == 'a')
                                tble[i + 1][1] = 'p';
                        else if (f == 'b')
                                tble[i + 1][2] = 'q';
                        else if (f == '@') {
                                for (k = 0; fol[i][k] != '\0'; k++)
                                        if (fol[i][k] == 'a')
                                                tble[i + 1][1] = 'r';
                        }
                }
        }
        printf("The Parsing Table is : \n");
        for (i = 0; i < 3; i++) {
                for (j = 0; j < 4; j++) {
                        if (tble[i][j] == 'p')
                                printf("%s\t\t", p);
                        else if (tble[i][j] == 'q')
                                printf("%s\t\t", q);
                        else if (tble[i][j] == 'r')
                                printf("%s\t\t", r);
                        else
                                printf("%c\t\t", tble[i][j]);
                }
                printf("\n");
        }
}
void parseInput() {
        printf("Enter The String : ");
        scanf("%s", inp);
        strcpy(inpt, inp);
        strcat(inpt, "$");
        strcpy(stk, "A$");
        i = 0;
```

```c
        j = 0;
        k = 0;
        printf("Matched\t\tStack\t\tInput\t\tActio\n");
        while (1) {
                if (stk[i] == inpt[j]) {
                        if (stk[i] == '$') {
                                print("Accepted\n");
                                break;
                        }
                        print("POP");
                        printf(" %c\n", stk[i]);
                        mch[k++] = stk[i];
                        stk[i++] = inpt[j++] = ' ';
                        continue;
                } else if (stk[i] == 'A') {
                        print("A->aBa\n");
                        strcpy(stk, "aBa$");
                } else if (stk[i] == 'B' && inpt[j] == 'b') {
                        print("B->bB\n");
                        strcpy(stk, "bBa$");
                        i = 0;
                } else if (stk[i] == 'B' && inpt[j] == 'a') {
                        print("B->@\n");
                        stk[i++] = ' ';
                } else {
                        print("ERROR\n");
                        exit(0);
                }
        }
}
void print(char* s) {
        printf("%s\t\t%s\t\t%s\t\t%s", mch, stk, inpt, s);
}
```

```
/*   OUTPUT
1)
The Given Grammar is :
A->aBa
B-bB|@
Enter The Number Of Rules : 2
Enter The Rules :
A->aBa
B->bB|@
The First Set Is :
FIRST[A]={a}
FIRST[B]={b,@}
The Follow Set Is :
FOLLOW[A]={$}
FOLLOW[B]={a}
The Parsing Table is :
             a              b              $
A                          A->aBa
B           B->@           B->bB


Enter The String : abba
Matched      Stack        Input       Actio
             A$           abba$       A->aBa
             aBa$         abba$       POP a
a            Ba$          bba$        B->bB
a            bBa$         bba$        POP b
ab           Ba$          ba$         B->bB
ab           bBa$         ba$         POP b
abb          Ba$          a$          B->@
abb          a$           a$          POP a
abba         $            $           Accepted
2)
The Given Grammar is :
A->aBa
B-bB|@
Enter The Number Of Rules : 2
Enter The Rules :
A->aBa
B->bB|@
The First Set Is :
FIRST[A]={a}
FIRST[B]={b,@}
The Follow Set Is :
FOLLOW[A]={$}
FOLLOW[B]={a}
The Parsing Table is :
             a              b              $
A                          A->aBa
B           B->@           B->bB
Enter The String : abb
Matched          Stack        Input        Actio
             A$           abb$        A->aBa
             aBa$         abb$        POP a
a            Ba$          bb$         B->bB
a            bBa$         bb$         POP b
ab           Ba$          b$          B->bB
ab           bBa$         b$          POP b
abb          Ba$          $           ERROR                  */
```

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

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
char G[6][10]={"E+T","T*F","(E)","T","F","id"};
char L[6]={'E','T','F','E','T','F'};
char stk[20]="$";
char inpt[20];
char pr[20],temp[20];
int i,j,k,f,len,lens,lenr,top=1,index1;
void extract()
{
    for(k=0;k<lenr;k++)
        temp[k]=stk[lens-lenr+k+1];
    temp[k]='\0';
}
void print()
{
    printf("%s\t\t%s$\t\t%s\n",stk,inpt,pr);
}
void shift()
{
        if(inpt[len-1]==' ')
        {
                    sprintf(pr,"ERROR");
                    print();
                    exit(0);
        }
    stk[top++]=inpt[i];
    if(inpt[i]=='i' && inpt[i+1]=='d')
    {
       stk[top++]=inpt[i+1];
       inpt[i+1]=' ';
       inpt[i]=' ';
       sprintf(pr,"SHIFT id");
       i++;
    }
    else
    {
       sprintf(pr,"SHIFT %c",inpt[i]);
       inpt[i]=' ';
    }
    print();
}
void reduce()
{
    int l=strlen(stk);
    stk[l-lenr]=L[index1];
    top=l-lenr+1;
    for(k=top;k<20;k++)
        stk[k]='\0';
    sprintf(pr,"REDUCE %c->%s",L[index1],G[index1]);
    print();
}
```

```c
int main1()
{
    printf("The given GRAMMAR is \nE->E+T|T\nT->T*F|F\nF->(E)|id\n");
    printf("Enter The String : ");
    scanf("%[^\n]",inpt);
    len=strlen(inpt);
    printf("STACK\t\tINPUT\t\t\tACTION\n");
    print(pr);
    i=0;
    lens=strlen(stk)-1;
    while(i<len || lens>0)
    {
        f=0;
        lens=strlen(stk)-1;
        if(stk[1]=='E' && lens==1 && inpt[len-1]==' ')
        {
                sprintf(pr,"ACCEPT");
                print();
                break;
        }
        if(lens==0)
        {
            shift();
            i++;
            continue;
        }
        for(j=0;j<6;j++)
        {
            lenr=strlen(G[j]);
            if(lens<lenr)
             continue;
            extract();
            if(strcmp(temp,G[j])==0)
            {
             if((j==0 || j==3) && inpt[i]=='*')
                    break;
                f=1;
                index1=j;
                break;
            }
        }
        if(f==1)
        {
            reduce();
        }
        else
        {
            shift();
            i++;
            continue;
        }
    }
}
```

/* __OUTPUT__

Note: Do Not give spaces in between the operator in the input

1)
```
The given GRAMMAR is
E->E+T|T
T->T*F|F
F->(E)|id
Enter The String : id++id
STACK           INPUT               ACTION
$           id++id$
$id            ++id$               SHIFT id
$F             ++id$               REDUCE F->id
$T             ++id$               REDUCE T->F
$E             ++id$               REDUCE E->T
$E+             +id$               SHIFT +
$E++            id$                SHIFT +
$E++id            $                SHIFT id
$E++F             $                REDUCE F->id
$E++T             $                REDUCE T->F
$E++E             $                REDUCE E->T
$E++E             $                ERROR
```

2)
```
The given GRAMMAR is
E->E+T|T
T->T*F|F
F->(E)|id
Enter The String : id+id*id
STACK           INPUT               ACTION
$           id+id*id$
$id            +id*id$             SHIFT id
$F             +id*id$             REDUCE F->id
$T             +id*id$             REDUCE T->F
$E             +id*id$             REDUCE E->T
$E+             id*id$             SHIFT +
$E+id            *id$              SHIFT id
$E+F             *id$              REDUCE F->id
$E+T             *id$              REDUCE T->F
$E+T*            id$               SHIFT *
$E+T*id                   $             SHIFT id
$E+T*F            $                REDUCE F->id
$E+T              $                REDUCE T->T*F
$E                $                REDUCE E->E+T
$E                $                ACCEPT
```

*/

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

```c
#include<stdio.h>
#include<stdlib.h>
#include<ctype.h>
char op[2],arg1[5],arg2[5],result[5];
void main()
{
        FILE *fp1,*fp2;
        fp1=fopen("5input.txt","r");
        fp2=fopen("5output.txt","w");
        while(!feof(fp1))
        {
                fscanf(fp1,"%s%s%s%s",result,arg1,op,arg2);
                if(strcmp(op,"+")==0)
                {
                        fprintf(fp2,"\nMOV R0,%s",arg1);
                        fprintf(fp2,"\nADD R0,%s",arg2);
                        fprintf(fp2,"\nMOV %s,R0",result);
                }
                if(strcmp(op,"*")==0)
                {
                        fprintf(fp2,"\nMOV R0,%s",arg1);
                        fprintf(fp2,"\nMUL R0,%s",arg2);
                        fprintf(fp2,"\nMOV %s,R0",result);
                }
                if(strcmp(op,"-")==0)
                {
                        fprintf(fp2,"\nMOV R0,%s",arg1);
                        fprintf(fp2,"\nSUB R0,%s",arg2);
                        fprintf(fp2,"\nMOV %s,R0",result);
                }
                if(strcmp(op,"/")==0)
                {
                        fprintf(fp2,"\nMOV R0,%s",arg1);
                        fprintf(fp2,"\nDIV R0,%s",arg2);
                        fprintf(fp2,"\nMOV %s,R0",result);
                }
                if(strcmp(op,"=")==0)
                {
                        fprintf(fp2,"\nMOV R0,%s",arg1);
                        fprintf(fp2,"\nMOV %s,R0\n",result);
                }
        }
```

fclose(fp1);
fclose(fp2);
}

/* **OUTPUT**

[root@localhost ss]# gedit 5input.txt
T1 -B = ?
T2 C + D
T3 T1 * T2
A T3 = ?

[root@localhost ss]# cc 5.c
[root@localhost ss]# ./a.out 5input.txt
[root@localhost ss]# cat 5output.txt

MOV R0,-B
MOV T1,R0
MOV R0,C
ADD R0,D
MOV T2,R0
MOV R0,T1
MUL R0,T2
MOV T3,R0
MOV R0,T3
MOV A,R0
MOV R0,T3
MOV A,R0

*/

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

```
%{
#include<stdio.h>
int com=0;
%}
%%
"//".* {com++;}
"/*"([^*]|\*+[^*/])*\*+"/" {com++;}
.|\n {fprintf(yyout,"%s",yytext);}
%%

void main(int argc, char *argv[])
{
yyin=fopen(argv[1],"r");
yyout=fopen(argv[2],"w");
yylex();
printf("No of comment lines=%d\n",com);
}
```

/* **OUTPUT**

[root@localhost ss]# lex 6a.l
[root@localhost ss]# cc lex.yy.c -ll
[root@localhost ss]# ./a.out comin.c comout.c

No of comment lines=5

[root@localhost ss]# cat comout.c
```
#include<stdio.h>
void main()
{
int a,b,c;
a= b+c;
}
```

[root@localhost ss]# gedit comin.c
```
//welcome to BIT
#include<stdio.h>
/*   Main begins
*/
void main()
{
// declaration
int a,b,c;  //  integer types
a= b+c;
/* *************
********nmm,,*/
}
```

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

LEX PART

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
int|char|bool|float|void|for|do|while|if|else|return|void|main {printf("keyword is
%s\n",yytext);return KEY;}
[+|-|*|/|=|<|>] {printf("operator is %s\n",yytext);return OP;}
[a-zA-Z][_a-zA-Z0-9]* {printf("identifier is %s\n",yytext);return ID;}
. ;
%%
```

YACC PART

```
%{
 #include <stdio.h>
 #include <stdlib.h>
int id=0, key=0, op=0;
 %}

%token ID KEY OP
%%
input: ID input { id++; }
| KEY input { key++; }
| OP input {op++;}
| ID { id++; }
| KEY { key++; }
| OP { op++;}
 ;
 %%

extern FILE *yyin;

void main(int argc ,char** argv)
{
        yyin = fopen(argv[1],"r");
        yyparse();
        printf("Keywords = %d\nIdentifiers = %d\noperators = %d\n", key,id, op);
}

void yyerror()
{
        printf("Not valid");
}
```

/* **OUTPUT**

[root@localhost ss]# gedit 6bin.c
```
void main()
{
int a,b,c;
p= q+r;
m = k + t;
}
```

[root@localhost ss]# lex 6b.l
[root@localhost ss]# yacc -d 6b.y
[root@localhost ss]# cc lex.yy.c y.tab.c -ll
[root@localhost ss]# ./a.out 6bin.c
keyword is void
keyword is main

keyword is int
identifier is a
identifier is b
identifier is c

identifier is p
operator is =
identifier is q
operator is +
identifier is r

identifier is m
operator is =
identifier is k
operator is +
identifier is t

Keywords = 3
Identifiers = 9
operators = 4

*/

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.

```c
#include<stdio.h>
#include<stdlib.h>
int bt[10], rbt[10], at[10] = { 0 }, ct[10], wt[10], tat[10];
int choice, tq, n;
void roundRobin();
void srtf();
void readBT() {
        printf("Enter Burst Time : ");
        for (int i = 0; i < n; i++) {
                scanf("%d", &bt[i]);
                rbt[i] = bt[i];
        }
}
void readAT() {
        printf("Enter Arrival Time : ");
        for (int i = 0; i < n; i++)
                scanf("%d", &at[i]);
}
void display() {
        int swt = 0, stat = 0;
        for (int i = 0; i < n; i++) {
                tat[i] = ct[i] - at[i];
                wt[i] = tat[i] - bt[i];
                swt += wt[i];
                stat += tat[i];
        }
        printf("PNO\tAT\tBT\tCT\tTAT\tWT\n");
        for (int i = 0; i < n; i++)
                printf("%d\t%d\t%d\t%d\t%d\t%d\n", i, at[i], bt[i], ct[i], tat[i],
                        wt[i]);
        printf("Average TAT : %f\n", (float) stat / n);
        printf("Average WT : %f\n", (float) swt / n);
}
int main() {
        setbuf(stdout,NULL);
        for (;;) {
                printf("1)RR\n2)SRTF\n3)EXIT\n");
                printf("Enter Choice : ");
                scanf("%d", &choice);
                switch (choice) {
                case 1:
                        printf("ROUND ROBIN\n");
                        printf("Enter The Number Of Processes : ");
                        scanf("%d", &n);
                        readBT();
                        printf("Enter Time Quantum : ");
                        scanf("%d", &tq);
                        roundRobin();
                        break;
                case 2:
                        printf("SRTF\n");
                        printf("Enter The Number Of Processes : ");
                        scanf("%d", &n);
                        readBT();
```

```
                        readAT();
                        srtf();
                        break;
                case 3:
                        exit(0);
                }
        }
        return 0;
}
void roundRobin() {
        int count = 0, i, time = 0;
        while (1) {
                for (i = 0; i < n; i++) {
                        if (rbt[i] > tq) {
                                rbt[i] -= tq;
                                time += tq;
                        } else if (rbt[i] != 0) {
                                time += rbt[i];
                                count++;
                                rbt[i] = 0;
                                ct[i] = time;
                        }
                }
                if (count == n)
                        break;
        }
        display();
}
void srtf() {
        int count = 0, i, time;
        rbt[9] = 999;
        for (time = 0; count != n; time++) {
                int smallest = 9;
                for (i = 0; i < n; i++) {
                        if (at[i] <= time && rbt[i] < rbt[smallest] && rbt[i] > 0)
                                smallest = i;
                }
                rbt[smallest]--;
                if (rbt[smallest] == 0) {
                        count++;
                        ct[smallest] = time + 1;
                }
        }
        display();
}
```

/***<u>OUTPUT</u>**

```
1)RR
2)SRTF
3)EXIT
Enter Choice : 1
ROUND ROBIN
Enter The Number Of Processes : 4
Enter Burst Time : 4 2 5 2
Enter Time Quantum : 15
PNO     AT      BT      CT      TAT     WT
0       0       4       4       4       0
1       0       2       6       6       4
2       0       5       11      11      6
3       0       2       13      13      11
Average TAT : 8.500000
Average WT : 5.250000
1)RR
2)SRTF
3)EXIT
Enter Choice : 2
SRTF
Enter The Number Of Processes : 5
Enter Burst Time : 5 4 4 2 4
Enter Arrival Time : 3 0 5 7 5
PNO     AT      BT      CT      TAT     WT
0       3       5       9       6       1
1       0       4       4       4       0
2       5       4       15      10      6
3       7       2       11      4       2
4       5       4       19      14      10
Average TAT : 7.600000
Average WT : 3.800000
1)RR
2)SRTF
3)EXIT
Enter Choice : 3
```

*/

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

```c
#include <stdio.h>
#include <stdlib.h>
struct process
{
    int alloc[5], max[5], need[5], finished;
} p[10];
int avail[5], req[5], work[5], sseq[10];
int np, nr;
void input()
{
    int i, j, chk = 0;
    printf("Enter The No Of Processes : ");
    scanf("%d", &np);
    printf("Enter The No Of Resources : ");
    scanf("%d", &nr);
    printf("Enter Availability Matrix : \n");
    for (i = 0; i < nr; i++)
        scanf("%d", &avail[i]);
    printf("Enter Allocated Matrix : \n");
    for (i = 0; i < np; i++)
        for (j = 0; j < nr; j++)
            scanf("%d", &p[i].alloc[j]);
    printf("Enter Max Matrix : \n");
    for (i = 0; i < np; i++)
        for (j = 0; j < nr; j++)
        {
            scanf("%d", &p[i].max[j]);
            p[i].need[j] = p[i].max[j] - p[i].alloc[j];
            if (p[i].need[j] < 0)
                chk = 1;
        }
    if (chk)
        printf("Allocation must be Less than Max\n");
}
int safe()
{
    int flag, sp = 0, i, j;
    for (i = 0; i < nr; i++)
        work[i] = avail[i];
    for (i = 0; i < np; i++)
        p[i].finished = 0;
    while (sp != np)
    {
        flag = 0;
        for (i = 0; i < np; i++)
        {
            if (p[i].finished)
                continue;
```

```c
        int less = 1;
        for (j = 0; j < nr; j++)
           if (p[i].need[j] > work[j])
              less = 0;
        if (less)
        {
           p[i].finished = 1;
           flag = 1;
           sseq[sp++] = i;
           for (j = 0; j < nr; j++)
              work[j] += p[i].alloc[j];
        }
     }
     if (!flag)
     {
        printf("No Safe Sequence\n");
        return 0;
     }
  }
  printf("Safe Sequence \n");
  for (i = 0; i < np; i++)
     printf("P%d ", sseq[i]);
  printf("\n");
  return 1;
}
void newReq()
{
  int pid, i, j, chk1 = 0, chk2 = 0;
  printf("Enter Process ID : ");
  scanf("%d", &pid);
  printf("Enter Request Matrix : \n");
  for (j = 0; j < nr; j++)
  {
     scanf("%d", &req[j]);
     if (req[j] > p[pid].need[j])
        chk1 = 1;
     if (req[j] > avail[j])
        chk2 = 1;
  }
  if (chk1)
  {
     printf("Process Exceeds Max Need\n");
     return;
  }
  if (chk2)
  {
     printf("Lack Of Resources\n");
     return;
  }
  for (j = 0; j < nr; j++)
```

```c
   {
      avail[j] -= req[j];
      p[pid].alloc[j] += req[j];
      p[pid].need[j] -= req[j];
   }
   if (!safe())
   {
      for (j = 0; j < nr; j++)
      {
         avail[j] += req[j];
         p[pid].alloc[j] -= req[j];
         p[pid].need[j] += req[j];
      }
   }
   else
      printf("Request Committed\n");
}
void display()
{
   int i, j;
   printf("Number of Process : %d\n", np);
   printf("Number of Resources : %d\n", nr);
   printf("PID\tMax\tAllocated\tNeed\n");
   for (i = 0; i < np; i++)
   {
      printf("P%d\t", i);
      for (j = 0; j < nr; j++)
         printf("%d ", p[i].max[j]);
      printf("\t");
      for (j = 0; j < nr; j++)
         printf("%d ", p[i].alloc[j]);
      printf("\t");
      for (j = 0; j < nr; j++)
         printf("%d ", p[i].need[j]);
      printf("\n");
   }
   printf("Available\n");
   for (i = 0; i < nr; i++)
      printf("%d ", avail[i]);
   printf("\n");
}
void main1()
{
   int ch;
   for (;;)
   {
      printf("1)Input 2)NewRequest 3)Safe 4)Display 5)Exit\n");
      printf("Enter Choice : ");
      scanf("%d", &ch);
      switch (ch)
```

```
        {
          case 1:
             input();
             break;
          case 2:
             newReq();
             break;
          case 3:
             safe();
             break;
          case 4:
             display();
             break;
          case 5:
             exit(0);
        }
      }
}
```

/***OUTPUT**

1)Input 2)NewRequest 3)Safe 4)Display 5)Exit
Enter Choice : 1
Enter The No Of Processes : 5
Enter The No Of Resources : 3
Enter Availability Matrix :
3 3 2
Enter Allocated Matrix :
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2
Enter Max Matrix :
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3
1)Input 2)NewRequest 3)Safe 4)Display 5)Exit
Enter Choice : 4
Number of Process : 5
Number of Resources : 3
PID Max   Allocated Need
P0 7 5 3 0 1 0   7 4 3
P1 3 2 2 2 0 0   1 2 2
P2 9 0 2 3 0 2   6 0 0
P3 2 2 2 2 1 1   0 1 1
P4 4 3 3 0 0 2   4 3 1
Available

3 3 2
1)Input 2)NewRequest 3)Safe 4)Display 5)Exit
Enter Choice : 3
Safe Sequence
P1 P3 P4 P0 P2
1)Input 2)NewRequest 3)Safe 4)Display 5)Exit
Enter Choice : 2
Enter Process ID : 1
Enter Request Matrix :
1 0 2
Safe Sequence
P1 P3 P4 P0 P2
Request Committed
1)Input 2)NewRequest 3)Safe 4)Display 5)Exit
Enter Choice : 4
Number of Process : 5
Number of Resources : 3
PID Max   Allocated Need
P0 7 5 3 0 1 0   7 4 3
P1 3 2 2 3 0 2   0 2 0
P2 9 0 2 3 0 2   6 0 0
P3 2 2 2 2 1 1   0 1 1
P4 4 3 3 0 0 2   4 3 1
Available
2 3 0
1)Input 2)NewRequest 3)Safe 4)Display 5)Exit
Enter Choice : 2
Enter Process ID : 4
Enter Request Matrix :
3 3 0
Lack Of Resources
1)Input 2)NewRequest 3)Safe 4)Display 5)Exit
Enter Choice : 5
*/

___

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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int S[100];
int A[10][100];
int len, nf, i, j, v, f, h, k, ctn;
void read()
{
   int i;
   printf("Enter The No Of Frames : ");
   scanf("%d", &nf);
   printf("Enter The Length : ");
   scanf("%d", &len);
   printf("Enter The String : ");
   for (i = 0; i < len; i++)
      scanf("%d", &S[i]);
}
void display()
{
   for (i = 0; i < len; i++)
      printf("%d ", S[i]);
   printf("\n\n");
   for (i = 0; i <= nf; i++)
   {
      for (j = 0; j < len; j++)
      {
         if (A[i][j] == -1)
            printf(" ");
         else
            printf("%d ", A[i][j]);
      }
      if (i == nf - 1)
         printf("\n");
      printf("\n");
   }
   printf("No of Hits : %d\n", h);
   printf("No of Faults : %d\n", (len - h));
}
void setInit()
{
   for (i = 0; i < nf; i++)
      A[i][0] = -1;
   A[0][0] = S[0];
}
void alg(int type)
{
```

___

```c
     h = 0;
   for (j = 1; j < len; j++)
   {
      f = 0;
      int C[10] = { 0 };
      for (i = 0; i < nf; i++)
      {
        A[i][j] = A[i][j - 1];
        if (A[i][j] == S[j])
           f = 1;
      }
      if (f)
      {
        A[nf][j] = 1;
        h++;
      }
      else
      {
        if (type == 1) // LRU
        {
           ctn = 0;
           for (i = j - 1; i >= 0 && ctn < nf - 1; i--)
              for (k = 0; k < nf; k++)
                if (A[k][j] == S[i])
                {
                   C[k] = 1;
                   ctn++;
                }
           v = 0;
           for (i = 0; i < nf; i++)
              if (C[i] == 0)
              {
                 v = i;
                 break;
              }
           A[v][j] = S[j];
        }
        else // FIFO
        {
           A[v][j] = S[j];
           v = (v + 1) % nf;
        }
      }
   }
}
int main()
{
   int ch;
   while (1)
   {
```

```
        printf("1)FIFO 2)LRU 3)EXIT\nEnter The Choice : ");
        scanf("%d", &ch);
        v = 1;
        switch (ch)
        {
            case 1:
                read();
                setInit();
                alg(0);
                display();
                break;
            case 2:
                read();
                setInit();
                alg(1);
                display();
                break;
            case 3:
                exit(0);
        }
    }
}
```

OUTPUT

```
1)FIFO 2)LRU 3)EXIT
Enter The Choice : 1
Enter The No Of Frames : 3
Enter The Length : 20
Enter The String : 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7 7 7 2 2 2 2 4 4 4 0 0 0 0 0 0 0 7 7 7
  0 0 0 0 3 3 3 2 2 2 2 2 1 1 1 1 1 0 0
    1 1 1 1 0 0 0 3 3 3 3 3 2 2 2 2 2 1

0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 1 1 0 0 0
No of Hits : 5
No of Faults : 15
1)FIFO 2)LRU 3)EXIT
Enter The Choice : 2
Enter The No Of Frames : 3
Enter The Length : 10
Enter The String : 2 1 4 3 5 4 3 2 1 5
2 1 4 3 5 4 3 2 1 5

2 2 2 3 3 3 3 3 3 5
  1 1 1 5 5 5 2 2 2
    4 4 4 4 4 4 1 1

0 0 0 0 1 1 1 0 0 0
No of Hits : 2
No of Faults : 8
1)FIFO 2)LRU 3)EXIT
Enter The Choice : 3
```

VIVA QUESTIONS

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.

What is an Assembler?
Assembler for an assembly language, a computer program to translate between lowerlevel representations of computer programs.

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.

Explain yyleng?
yyleng-contains the length of the string our lexer recognizes.

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.

What is the Syntax of a Language?
The Rules which tells whether a string is a valid Program or not are called the Syntax.

What is the Semantics of a Language?
The Rules which gives meaning to programs are called the Semantics of a Language.

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.

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.

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.

How are the tokens recognized?

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

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.

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.

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.

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.

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 Leafs 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 represents the Non-Terminal symbols.

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.

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.

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.

What is exit status command?

Exit 0- return success, command executed successfully.
Exit 1 – return failure.

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.