Channabasaveshwara Institute of Technology

(Affiliated to VTU, Belgaum & Approved by AICTE, New Delhi)
(**ISO 9001:2015 Certified Institution**)
NH 206 (B.H. Road), Gubbi, Tumkur – 572 216. Karnataka.

System Software Laboratory
[18CSL66]

Department of Computer Science & Engineering

VI Semester

Academic Year : 2022-23

1. 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:

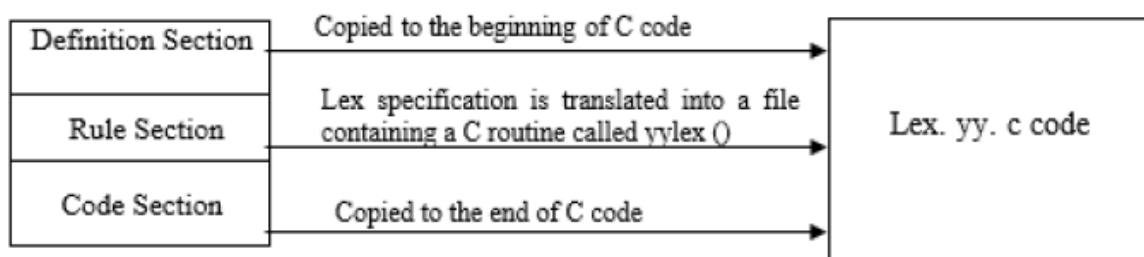**1st Step:** Using gedit create a file with extension l. For example: prg1.l
2ndStep: lex prg1.l
3rdStep: cc lex.yy.c –ll
4thStep: ./a.out

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 lexerwill 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)*	

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.
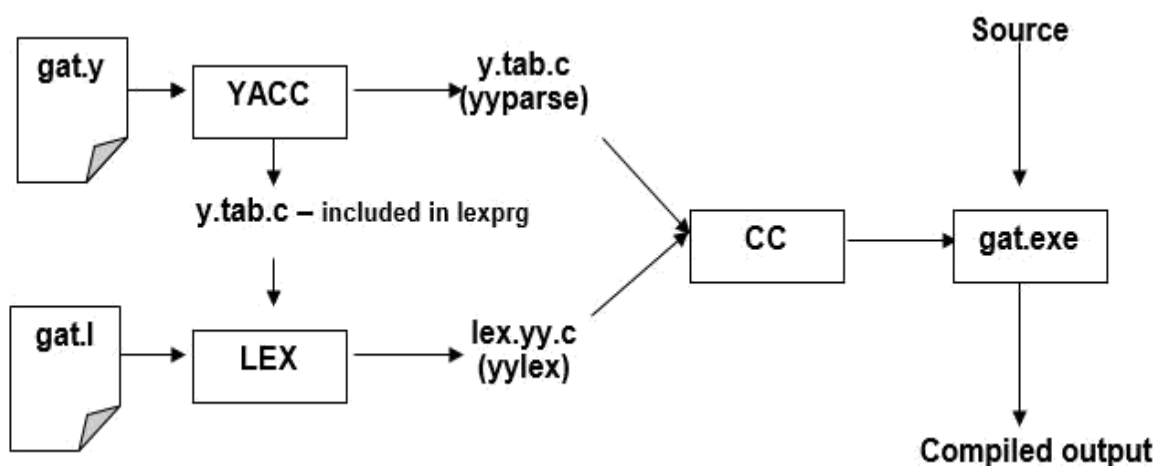
2.1 Steps in writing YACC Program:

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

> {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 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.

3. 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

4. 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
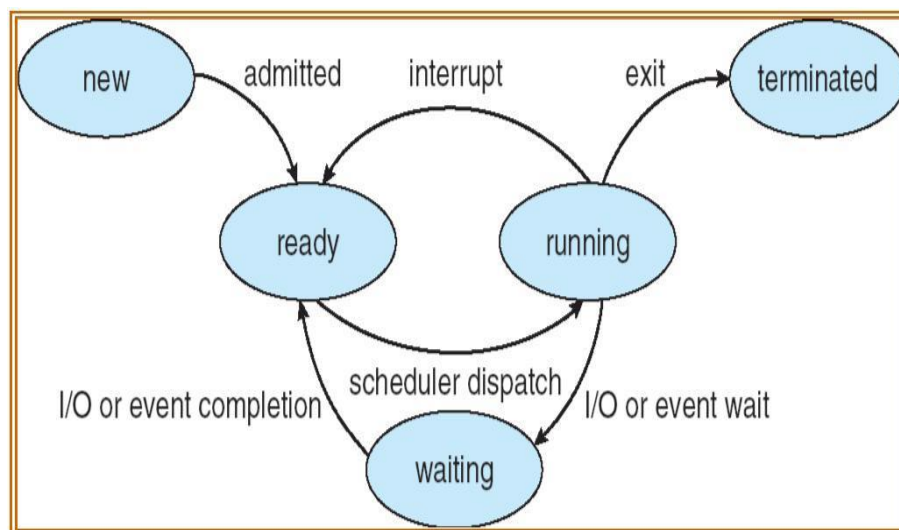
Apart from the program code, it includes the current activity represented by

- ➢ 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 remaining time scheduling algorithm:

Shortest remaining time, also known as shortest remaining time first (SRTF), is a scheduling method that is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with 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 for process 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 EdsgerDijkstrathat 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.

5. 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 low-level machine language that is required by computers.

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 machineindependent 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.
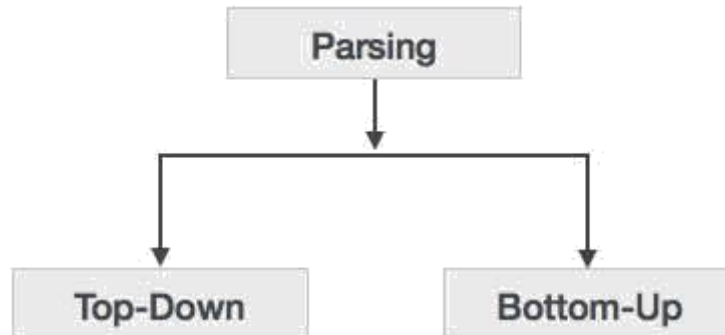
Top-Down Parsing:

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 process using 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 an intermediate code used by optimizing compilers to aid in the implementation of code-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 or processor registers, but rather symbolic addresses that will be translated into actual addresses during register 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

---

6. SAMPLE LEX AND YACC PROGRAMS

Example LEX Program

A Simple program:To count the no of a's in the given string

```
%{
int c=0;
%}
%%
[a]*            {c++;}
.               ;
%%
main()
{
            yylex();
            printf("The no of a's in the given string : %d",c);
}
```

How to run this Program?

$lex filename.l

$cc lex.yy.c –ll

$ ./a.out

aaa

The no of a's in the given string: 3

Example YACC Program

A Simple program: To print whether the input string is accepted or not.

```
%{
        #include<stdio.h>
%}
% token A B
%%
S:
  |A S B
%%
```

```
main()
{
        printf("Enter the string\n");
        yyparse();
        printf("given string is accepted");
}
yyerror()
{
        printf("not accepted");
        exit(0);
}
yylex()
{
        intch;
        ch=getchar();
        if(ch=='a')
        return A;
        if(ch=='b')
        return B;
        if(ch=='\n')
        returnch;
}
```

How to run this program?

$ yaccfilename.y

$cc y.tab.c –ll

$./a.out

Enter the string

aabb

Given string is accepted

7. 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>
int opnd=0, oprt=0,flag=0;
char opnd1[10][10], oprt1[10][10];
%}
%%
[0-9]* {strcpy(opnd1[opnd],yytext); opnd++;}
[\+\*]  { strcpy(oprt1[oprt],yytext); oprt++;}
.{flag=1;}
%%
int main()
{
        inti,j;
        printf("Enter the expression ");
        yylex();
        if(((opnd-oprt)==1)&&flag==0)
        {
                printf("\n Identifiers are:%d\n Operators are:%d\n",opnd,oprt);
                printf("\n Expression is Valid\n");
                for(i=0;i<opnd;i++)
                        printf("\nOperands are %s ",opnd1[i]);
                for(j=0;j<oprt;j++)
                        printf("\nOperators are %s ",oprt1[j]);
        }
        else
                printf("\n Expression is Invalid\n");
        return 1;
}
```

Output:

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

Lex Part

```
%{
#include "y.tab.h"
externyylval;
%}
%%
[0-9]+ {yylval=atoi(yytext);return num;}        /* convert the string to number
                                                    and send the value*/

[\+\-\*\/] {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 error\n");exit(0);}
                else
                $$=$1/$3;}
|'('exp')'      {$$=$2;}
|num            {$$=$1;};
%%
intyyerror()
{
        printf("error");
        exit(0);
}
int main()
{
        printf("Enter an expression:\n");
        yyparse();
}
```

Output:

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"
#include<stdio.h>
#include<string.h>
#define N 3
%}
%%
a     {if(strlen(yytext)==N) return A;}
b     {return B;}
%%
```

YACC Part

```
%{
#include<stdio.h>
%}
%token A B
%%
input:s'\n' {printf("Successful Grammar\n");exit(0);}
s: A s1 B
s1:
| A s1
%%
 main()
{
    printf("Program to check the
    grammar a^nb\n");
    printf("Enter A String\n");
    yyparse();
    printf("\nString is accepted by the
    grammar\n");
    exit(0);
}
intyyerror()
{
    printf("String is not accepted\n");
}
```

Output:

```
😣➖⊡  vidya@vidya-Inspiron-1564: ~/ssoslab
vidya@vidya-Inspiron-1564:~/ssoslab$ lex lab2.l
vidya@vidya-Inspiron-1564:~/ssoslab$ yacc -d lab2.y
vidya@vidya-Inspiron-1564:~/ssoslab$ cc lex.yy.c y.tab.c -ll
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
Enter A String


Invalid string
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
Enter A String
b

Successful Grammar
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
Enter A String
aaaabb

Invalid string
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
Enter A String
aaaaaaaaaaaaaaaaab

Successful Grammar
vidya@vidya-Inspiron-1564:~/ssoslab$ ▮
```

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

```
#include<stdlib.h>
#include<string.h>
#include<stdio.h>
char prod[3][10]={"A->aBa","B->bB","B->@"}, input[10],stack[25];
int top=-1; int j=0,k,l;

void push(char item)
{
stack[++top]=item;
}

void pop()
{
top=top-1;
}

void display()
{
int j; for(j=top;j>=0;j--)
printf("%c",stack[j]);
}

void stackpush(char p)
{
      if(p=='A')
      {
      pop();
      for(j=strlen(prod[0])-1;j>=3;j--)
      push(prod[0][j]);
      }
      else
      {
      pop();
      for(j=strlen(prod[1])-1;j>=3;j--)
      push(prod[1][j]);
      }
}

void main()
{
      char c;
```

```
        int i;

        printf("first(A)={a}\t");
        printf("follow(A)={$}\n");
        printf("first(B)={b,@}\t");
        printf("follow(B)={a}\n\n");

        printf("\t a \t b \t $ \n");
        printf("A\t%s\n",prod[0]);
        printf("B\t%s\t%s\n",prod[2],prod[1]);
        printf("enter the input string terminated with $ to parse:-");
        scanf("%s",input);

for(i=0;input[i]!='\0';i++)
{
        if((input[i]!='a')&&(input[i]!='b')&&(input[i]!='$'))
        {
        printf("invalid string");
        exit(0);
        }
}

if(input[i-1]!='$')
{
        printf("\n\nInput string entered without end marker $");
        exit(0);
}

push('$');
push('A');
i=0;
printf("\n\n");
printf("stack\tInput\taction");
printf("\n...........\n");
while(i!=strlen(input)&&stack[top]!='$')
{
        printf("\n");
        for(l=top;l>=0;l--)
        printf("%c",stack[l]);
        printf("\t");
        for(l=i;l<strlen(input);l++)
        printf("%c",input[l]);
        printf("\t");

if(stack[top]=='A')
```

```
{
        printf("A->aBa");
        stackpush('A');
}

else if(stack[top]=='B')
{
        if(input[i]!='b')
        {
                printf("B->@");
                printf("\t matched @");
                pop();
        }
        else
                {
                        printf("B->bB");
                        stackpush('B');
                }
}
else
        {
                if(stack[top]==input[i])
                {
                printf("pop%c",input[i]);
                printf("\tmatched %c",input[i]);
                pop();
                i++;
                }
        else
        break;
        }
}

if(stack[top]=='$' && input[i]=='$')
{
        printf("\n$\t$");
        printf("\nValid string Accepted\n");
}
else
printf("\nInvalid string rejected\n");
}
```

Output:
first(A)={a}   follow(A)={$}

first(B)={b,@}   follow(B)={a}

a   b $
A   A->aBa
B   B->@ B->bB

enter the input string terminated with $ to parse:-abba$
stack   Input   action

A$ abba$ A->aBa
aBa$ abba$ popa matched a
Ba$ bba$ B->bB
bBa$ bba$ popb matched b
Ba$ ba$ B->bB
bBa$ ba$ popb matched b
Ba$ a$ B->@ matched @
a$ a$ popa matched a
$ $

Valid string Accepted

Output:

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<conio.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();
void main()
  {
    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("enter input string ");

    gets(a);
    c=strlen(a);
    strcpy(act,"SHIFT->");
    puts("stack \t input \t action");

    for(k=0,i=0; j<c; k++,i++,j++)
     {
       if(a[j]=='i' && a[j+1]=='d')
         {
           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
            {
           stk[i]=a[j];
           stk[i+1]='\0';

           a[j]=' ';
           printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
           check();
         }
     }

    getch();
  }
void check()
    {
```

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

         if(stk[z]=='E' &&stk[z+1]=='*' &&stk[z+2]=='E')
           {
            stk[z]='E';
            stk[z+1]='\0';
            stk[z+1]='\0';
            printf("\n$%s\t%s$\t%s",stk,a,ac);
            i=i-2;
           }
        for(z=0; z<c; z++)

         if(stk[z]=='(' &&stk[z+1]=='E' &&stk[z+2]==')')
           {
            stk[z]='E';
            stk[z+1]='\0';
            stk[z+1]='\0';
                printf("\n$%s\t%s$\t%s",stk,a,ac);
            i=i-2;
           }
     }
```

Output:

```
vidya@vidya-Inspiron-1564: ~/ssoslab
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->id
Enter input string
id+id*id
Stack     Input   Action

$id        +id*id$      SHIFT->id
$E         +id*id$      REDUCE TO E
$E+         id*id$      SHIFT->symbols
$E+id        *id$       SHIFT->id
$E+E         *id$       REDUCE TO E
$E           *id$       REDUCE TO E
$E*           id$       SHIFT->symbols
$E*id          $        SHIFT->id
$E*E           $        REDUCE TO E
$E             $        REDUCE TO Evidya@vidya-Inspiron-1564:~/ssoslab$
```

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("input.txt","r"); fp2=fopen("output.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",result);
    }
    }
    fclose(fp1);
    fclose(fp2);
```
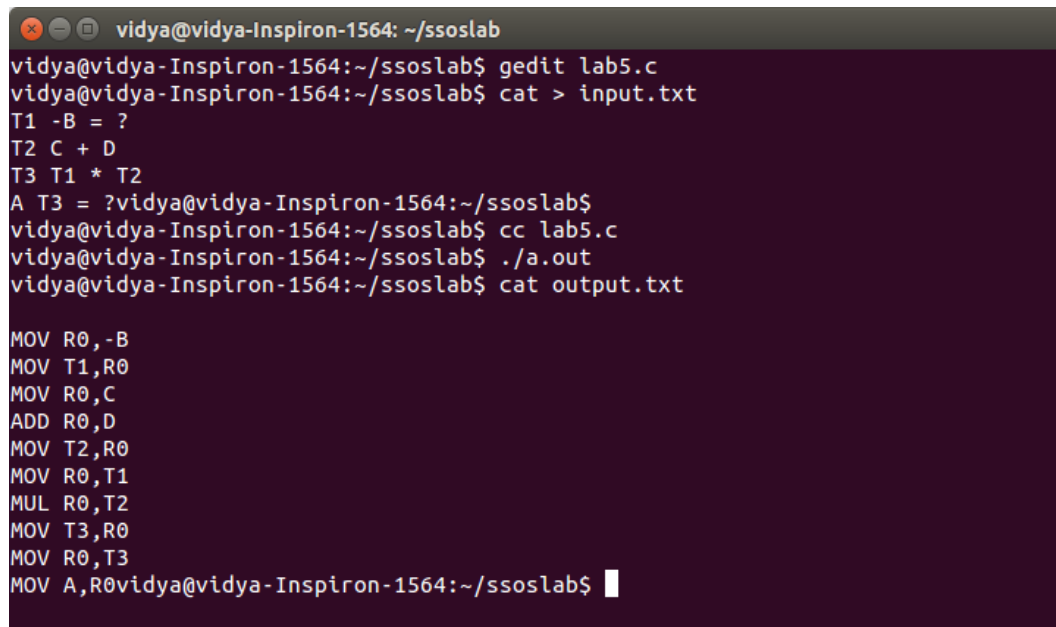
```
    getch();
}
```

Input.txt file

```
T1 − B = ?

T2  C + D

T3 T1 * T2

A T3 = ?
```

Output:

```
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
```

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>
intcc=0;
%}
%%
"/*"[^/*]*"*/" {cc++;}
"/*".*"*/"      {cc++;}
%%
main()
{
        printf("Program to eliminate comment lines\n");
        yyin=fopen("input.c","r");
        yyout=fopen("output.c","w");
        yylex();
        fclose(yyin);
        fclose(yyout);
        printf("\n Number of comment lines are %d\n",cc);
}
```

Output:

```
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
Program to eliminate comment lines

 Number of comment lines are 4
vidya@vidya-Inspiron-1564:~/ssoslab$ cat input.c
/*Program to print sum of two numbers*/
#include<stdio.h>

int main()
{
        int a,b,c;
        printf("Enter two numbers: \n");
        scanf("%d%d",&a,&b); /*Read input*/
        /*compute
                sum*/
        c=a+b;
        /*print result*/
        printf("Sum=%d",c);
}
vidya@vidya-Inspiron-1564:~/ssoslab$ cat output.c

#include<stdio.h>

int main()
{
        int a,b,c;
        printf("Enter two numbers: \n");
        scanf("%d%d",&a,&b);

        c=a+b;

        printf("Sum=%d",c);
}
vidya@vidya-Inspiron-1564:~/ssoslab$ █
```

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

Lex File

```
%{
#include <stdio.h>
#include<string.h>
#include "y.tab.h"
char id[100], key[100], op[100];
intidcount=0,keycount=0,opcount=0,flag=1;
%}
%%
int|char|float|double        {strcat(key,yytext); strcat(key," "); keycount++; flag=1; return
                             key;}
main|if|else|for|while|do    {strcat(key,yytext); strcat(key," "); keycount++; flag=1; return
                             key;}
[\+\-\*\/]                   {strcat(op,yytext); strcat(key," "); opcount++; return op;}
[a-zA-Z_][a-zA-Z0-9_]*[,;=\[] {if(flag==1) {strcat(id,yytext,yyleng-1); strcat(id," ");
                             idcount++; return id;}}
\n                          {flag=0;}
"print"                     ;
.                           ;
%%
```

Yacc File

```
%{
#include <stdio.h>
#include <stdlib.h>
extern char op[100], id[100], key[100];
externintopcount, idcount, keycount;
%}
%token KEY ID OP
%%
input: KEY input
| OP input
|ID input
%%
intmain()
{
    printf("Program to count and print OP, KEY, ID\n");
    yyparse();
    printf("Number of keywords are %d\n",keycount);
    printf("Keywords are %s\n",key);
    printf("Number of identifiers are %d\n",idcount);
    printf("Identifiers are %s\n",id);
    printf("Number of operators are %d\n",opcount);
    printf("Operators are %s\n",op);
}
yyerror()
```

{
      return -1;
}

Output:
Input file

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>
struct proc
{
        int id;
        int arrival;
        int burst;
        int rem;
        int wait;
        int finish;
        int turnaround;
        float ratio;
}process[10]; //structure to hold the process information
struct proc temp;
int no;
int chkprocess(int);
int nextprocess();
 void roundrobin(int, int, int[], int[]);
 void srtf(int);
 main()
{
        intn,tq,choice;
        intbt[10],st[10],i,j,k;
 for(; ;)
 {
 printf("Enter the choice \n");
printf(" 1. Round Robin\n 2.SRT\n 3. Exit \n");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("Round Robin scheduling algorithm\n");
printf("Enter number of processes:\n");
scanf("%d",&n);
printf("Enter burst time for sequences:");
for(i=0;i<n;i++)
  {
    scanf("%d",&bt[i]);
    st[i]=bt[i];          //service time
  }
  printf("Enter time quantum:");
  scanf("%d",&tq);
  roundrobin(n,tq,st,bt);
  break;
case 2:
```

```
                printf("\n \n ---SHORTEST REMAINING TIME NEXT---\n \n
                "); printf("\n \n Enter the number of processes: "); scanf("%d",
                &n);
                srtf(n);
break;
case 3: exit(0);
}// end of switch
```
}// end of for
}//end of main()

```
voidroundrobin(intn,inttq,intst[],intbt[])
 {
        int time=0;
        int tat[10],wt[10],i,count=0,swt=0,stat=0,temp1,sq=0,j,k;
        floatawt=0.0,atat=0.0;
        while(1)
        {
                for(i=0,count=0;i<n;i++)
                {
    temp1=tq;
    if(st[i]==0) // when service time of a process equals zero then//count value is
        incremented
    {
      count++;
      continue;
    }
    if(st[i]>tq) // when service time of a process greater than time//quantum
        then time
                st[i]=st[i]-tq;  //quantum value subtracted from service time
    else
      if(st[i]>=0)
      {
                temp1=st[i];        // temp1 stores the service time of a process
                st[i]=0;            // making service time equals 0
      }
      sq=sq+temp1;          // utilizing temp1 value to calculate turnaround time
      tat[i]=sq;            // turn around time
  } //end of for
  if(n==count)             // it indicates all processes have completed their task
                        because the count value
  break;                   // incremented when service time equals 0
 } //end of while
 for(i=0;i<n;i++)          // to calculate the wait time and turnaround time of each process
 {
  wt[i]=tat[i]-bt[i];       // waiting time calculated from the turnaround time burst time
  swt=swt+wt[i];          // summation of wait time
  stat=stat+tat[i];        // summation of turnaround time
 }
 awt=(float)swt/n;          // average wait time
```

```
   atat=(float)stat/n;            // average turnaround time
   printf("Process_no          Burst time      Wait time       Turn around time\n");
   for(i=0;i<n;i++)
    printf("%d\t\t%d\t\t%d\t\t%d\n",i+1,bt[i],wt[i],tat[i]);
    printf("Avg wait time is %f\n Avgturn around time is %f\n",awt,atat);
 }// end of Round Robin
intchkprocess(int s)            // function to check process remaining time is zero or not
{
int i;
        for(i = 1; i <= s; i++)
        {
           if(process[i].rem != 0)
                  return 1;
        }
        return 0;
} // end of chkprocess
intnextprocess()          // function to identify the next process to be executed
{
        int min, l, i;
        min = 32000;     //any limit assumed
        for(i = 1; i <= no; i++)
        {
                if( process[i].rem!=0 && process[i].rem < min)
                {
                        min = process[i].rem;
                        l = i;
                }
        }
        return l;
} // end of nextprocess

voidsrtf(int n)
{
inti,j,k,time=0;
floattavg,wavg;
for(i = 1; i <= n; i++)
        {
                process[i].id = i;
                printf("\n\nEnter the arrival time for process %d: ", i); scanf("%d",
                &(process[i].arrival));
                printf("Enter the burst time for process %d: ", i);
                scanf("%d", &(process[i].burst));
                process[i].rem = process[i].burst;
        }
        for(i = 1; i <= n; i++)
        {
                for(j = i + 1; j <= n; j++)
                {
if(process[i].arrival > process[j].arrival)     // sort arrival time of aprocess
```

```
                        {
                                temp = process[i];
                                process[i] = process[j];
                                process[j] = temp;
                        }
                }
        }
  no = 0;
  j = 1;
while(chkprocess(n) == 1)
{
        if(process[no + 1].arrival == time)
{
        while(process[no+1].arrival==time)
         no++;
                if(process[j].rem==0)
                        process[j].finish=time;
                j = nextprocess();
        }
        if(process[j].rem != 0)              // to calculate the waiting time of a process
        {
                process[j].rem--;
                for(i = 1; i <= no; i++)
                {
                    if(i != j && process[i].rem != 0)
                            process[i].wait++;

                }
        }
        else
        {
                process[j].finish = time;
                j=nextprocess();
                time--;
                k=j;
        }
        time++;
}
process[k].finish = time;
printf("\n\n\t\t\t---SHORTEST REMAINING TIME FIRST---");
printf("\n\n Process Arrival Burst Waiting Finishing turnaround Tr/Tb\n");
printf("%5s %9s %7s %10s %8s %9s\n\n", "id", "time", "time", "time", "time",
"time");
    for(i = 1; i <= n; i++)
     {
        process[i].turnaround = process[i].wait + process[i].burst; // calc of turnaround
        process[i].ratio = (float)process[i].turnaround / (float)process[i].burst;
```

```
        printf("%5d %8d %7d %8d %10d %9d %10.1f ", process[i].id,
            process[i].arrival, process[i].burst, process[i].wait, process[i].finish,
            process[i].turnaround, process[i].ratio);
            tavg=tavg+ process[i].turnaround;        //summation of turnaround time
            wavg=wavg+process[i].wait;               // summation of waiting time
printf("\n\n");
        }
        tavg=tavg/n;           // average turnaround time
        wavg=wavg/n;           // average wait time
         printf("tavg=%f\t wavg=%f\n",tavg,wavg); }// end of srtf
```

Output:

```
        Enter the choice
         1) Round Robin 2) SRT
         3) Exit
        1

        Round Robin scheduling algorithm
        ********************************
        Enter number of processes:3
        Enter burst time for sequences:24
        3
        3

        Enter time quantum:4
         Process_no    Burst time      Wait time    Turnaround time
            1             24              6              30
            2             3               4              7
            3             3               7              10
        Avg wait time is 5.666667
        Avg turnaround time is 15.666667
        Enter the choice
         1) Round Robin
         2) SRT
         3) Exit
        2

         ---SHORTEST REMAINING TIME NEXT---
         Enter the number of processes: 4
        Enter the arrival time for process 1: 0
        Enter the burst time for process 1: 8
        Enter the arrival time for process 2: 1
        Enter the burst time for process 2: 4
        Enter the arrival time for process 3: 2
        Enter the burst time for process 3: 9
        Enter the arrival time for process 4: 3
        Enter the burst time for process 4: 5
```

```
    1        24       6        30
    2        3        4        7
    3        3        7        10
      ---SHORTEST REMAINING TIME FIRST---
Enter the number of processes: 4
Enter the arrival time for process 1: 0
Enter the burst time for process 1: 8
Enter the arrival time for process 2: 1
Enter the burst time for process 2: 4
Enter the arrival time for process 3: 2
Enter the burst time for process 3: 9
Enter the arrival time for process 4: 3
Enter the burst time for process 4: 5
                ---SHORTEST REMAINING TIME NEXT---
```

Process id	Arrival time	Burst time	Waiting time	Finishing time	turnaround time	Tr/Tb time
1	0	8	9	17	17	2.1
2	1	4	0	5	4	1.0
3	2	9	15	26	24	2.7
4	3	5	2	10	7	1.4

```
tavg=13.000000
wavg=6.500000
Using OpenMP
```

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>
int main()
{
    int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10], safeSequence[10];
    int p, r, i, j, process, count;
    count = 0;
    printf("Enter the no of processes : ");
    scanf("%d", &p);
    for(i = 0; i< p; i++)
        completed[i] = 0;
    printf("\n\nEnter the no of resources : ");
    scanf("%d", &r);
    printf("\n\nEnter the Max Matrix for each process : ");
    for(i = 0; i < p; i++)
    {
        printf("\nFor process %d : ", i + 1);
        for(j = 0; j < r; j++)
            scanf("%d", &Max[i][j]);
}
  printf("\n\nEnter the allocation for each process : ");
  for(i = 0; i < p; i++)
  {
      printf("\nFor process %d : ",i + 1);
      for(j = 0; j < r; j++)
          scanf("%d", &alloc[i][j]);
  }
  printf("\n\nEnter the Available Resources : ");
  for(i = 0; i < r; i++)
      scanf("%d", &avail[i]);
  for(i = 0; i < p; i++)
      for(j = 0; j < r; j++)
          need[i][j] = Max[i][j] - alloc[i][j];
      do
      {
          printf("\n Max matrix:\tAllocation matrix:\n");
          for(i = 0; i < p; i++)
          {
              for( j = 0; j < r; j++)
                  printf("%d ", Max[i][j]);
              printf("\t\t");
              for( j = 0; j < r; j++)
                  printf("%d ", alloc[i][j]);
              printf("\n");
          }
          process = -1;
```

```
    for(i = 0; i < p; i++)
    {
       if(completed[i] == 0)//if not completed
       {
          process = i ;
          for(j = 0; j < r; j++)
          {
             if(avail[j] < need[i][j])
             {
                process = -1;
                break;
                     }
          }
       }
       if(process != -1)
          break;
    }
    if(process != -1)
    {
       printf("\nProcess %d runs to completion!", process + 1);
       safeSequence[count] = process + 1;
       count++;
       for(j = 0; j < r; j++)
       {
          avail[j] += alloc[process][j];
          alloc[process][j] = 0;
          Max[process][j] = 0;
          completed[process] = 1;
       }
    }
}
while(count != p && process != -1);
if(count == p)
{
   printf("\nThe system is in a safe state!!\n");
   printf("Safe Sequence : < ");
   for( i = 0; i < p; i++)
      printf("%d ", safeSequence[i]);
   printf(">\n");
}
else
   printf("\nThe system is in an unsafe state!!");
}
```

Output:

   Enter the no of processes : 5
   Enter the no of resources : 3
   Enter the Max Matrix for each process :
   For process 1 : 7
   5
   3
  For process 2 : 3
  2
  2
  For process 3 : 7
  0
  2
  For process 4 : 2
  2
  2
  For process 5 : 4
  3
  3
  Enter the allocation for each process :
  For process 1 : 0
  1
  0
  For process 2 : 2
  0
  0
  For process 3 : 3
  0
  2
  For process 4 : 2
  1
  1
  For process 5 : 0
  0
  2
  Enter the Available Resources : 3
  3
  2
   Max matrix: Allocation matrix:
  7 5 3   0 1 0
  3 2 2   2 0 0
  7 0 2   3 0 2
  2 2 2   2 1 1
  4 3 3   0 0 2

  Process 2 runs to completion!

Max matrix: Allocation matrix:
7 5 3   0 1 0
0 0 0   0 0 0
7 0 2   3 0 2
2 2 2   2 1 1

4 3 3   0 0 2

Process 3 runs to completion!
 Max matrix: Allocation matrix:
7 5 3   0 1 0
0 0 0   0 0 0
0 0 0   0 0 0
2 2 2   2 1 1
4 3 3   0 0 2

Process 4 runs to completion!
 Max matrix: Allocation matrix:
7 5 3   0 1 0
0 0 0   0 0 0
0 0 0   0 0 0
0 0 0   0 0 0
4 3 3   0 0 2

Process 1 runs to completion!
 Max matrix: Allocation matrix:
0 0 0   0 0 0
0 0 0   0 0 0
0 0 0   0 0 0
0 0 0   0 0 0
4 3 3   0 0 2

 Process 5 runs to completion!
 The system is in a safe state!!
Safe Sequence: < 2 3 4 1 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.

```
#include<stdio.h>
#include<stdlib.h>
void FIFO(char [ ],char [ ],int,int);
voidlru(char [ ],char [ ],int,int);
void opt(char [ ],char [ ],int,int);
int main()
{
  intch,YN=1,i,l,f;
  char F[10],s[25];
  printf("\n\n\tEnter the no of empty
  frames: "); scanf("%d",&f);
  printf("\n\n\tEnter the length of the string: ");
  scanf("%d",&l);
  printf("\n\n\tEnter 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\t4: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 4:
        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
voidlru(char s[],char F[],intl,int f)
{
  inti,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
4:EXIT

Enter your choice: 1
PAGE	FRAMES	FAULTS
H	h	Page-fault 0
E	h e	Page-fault 1
L	h e l	Page-fault 2
L	h e l	No page-fault
O	o e l	Page-fault 3

Do u want to continue IF YES PRESS 1
IF NO PRESS 0 : 1

*********** MENU ***********
1:FIFO
2:LRU
4:EXIT
Enter your choice: 2
PAGE	FRAMES	FAULTS
H	h	Page-fault 0
E	h e	Page-fault 1
L	h e l	Page-fault 2
L	h e l	No page fault
O	e l o	Page-fault 3

Do u want to continue IF YES PRESS 1
IF NO PRESS 0 : 1

*********** MENU ***********
1:FIFO
2:LRU
4:EXIT
Enter your choice: 4

8. 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. **What is an Assembler?**
   Assembler for an assembly language, a computer program to translate between lower-level representations of computer programs.
3. **Explain lex and yacc tools**
   a. Lex: - scanner that can identify those tokens
   b. Yacc: - parser.yacc takes a concise description of a grammar and produces a C routine that can parse that grammar.
4. **Explain yyleng?**
   yyleng-contains the length of the string our lexer recognizes.
5. **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.
6. **What is the Syntax of a Language?**
   The Rules which tells whether a string is a valid Program or not are called the Syntax.
7. **What is the Semantics of a Language?**
   The Rules which gives meaning to programs are called the Semantics of a Language.
8. **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.
9. **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.
10. **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.
11. **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.
12. **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.
13. **What are the Advantages of using Context-Free grammars?**
    a. It is precise and easy to understand.
    b. It is easier to determine syntactic ambiguities and conflicts in the grammar.
14. **If Context-free grammars can represent every regular expression, why do one needs R.E at all?**
    a. Regular Expression are Simpler than Context-free grammars.

    b. It is easier to construct a recognizer for R.E than Context-Free grammar.

    c. Breaking the Syntactic structure into Lexical & non-Lexical parts provide better front end for the Parser.

    d. 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.

15. **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.

16. **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.

17. **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.

18. **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.

19. **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.

20. **What is exit status command?**

Exit 0- return success, command executed successfully.

Exit 1 – return failure.

21. **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.