

RAJIV GANDHI INSTITUTE OF TECHNOLOGY
CHOLANAGAR ,R.T NAGAR, BEGALURU-560022



Department of Computer Science and Engineering

6th SEMESTER

**SYSTEM SOFTWARE & OPERATING
SYSTEM LABORATORY**

(18CSL66)

FACULTY INCHARGE

Prof. Swetha V

HOD Dept. of CSE

Dr. Arudra A

Table of Contents

Sl. No.	Particulars	Page No.
1	Introduction to LEX	4
2	Introduction to YACC	7
3	Introduction to UNIX	10
4	Introduction to Operating Systems	11
5	Introduction to Compiler Design	15
6	Sample LEX and YACC programs	16
7	Lab Programs	
	Program 1 a) Write a LEX program to recognize valid <i>arithmetic expression</i> . Identifiers in the expression could be only integers and operators could be + and *. Count the identifiers & operators present and print them separately. b) Write YACC program to evaluate <i>arithmetic expression</i> involving operators: +, -, *, and /.	18
	Program 2 Develop, Implement and Execute a program using YACC tool to recognize all strings ending with <i>b</i> preceded by <i>n</i> <i>a</i> 's using the grammar $a^n b$ (note: input <i>n</i> value).	22
	Program 3 Design, develop and implement YACC/C program to construct <i>Predictive / LL(1) Parsing Table</i> for the grammar rules: $A \rightarrow aBa, B \rightarrow bB / \epsilon$. Use this table to parse the sentence: <i>abba\$</i> .	24
	Program 4 Design, develop and implement YACC/C program to demonstrate <i>Shift Reduce Parsing</i> technique for the grammar rules: $E \rightarrow E+T / T, T \rightarrow T * F / F, F \rightarrow (E) / id$ and parse the sentence: <i>id + id * id</i> .	29
	Program 5 Design, develop and implement a C/Java program to generate the machine code using <i>Triples</i> for the statement $A = -B * (C + D)$ whose intermediate code in three-address form: $T1 = -B$ $T2 = C + D$ $T3 = T1 + T2$ $A = T3$	32
	Program 6 a) Write a LEX program to eliminate <i>comment lines</i> in a C program and copy the resulting program into a separate file. b) Write YACC program to recognize valid <i>identifier, operators and keywords</i> in the given text (C program) file.	34
	Program 7 Design, develop and implement a C/C++/Java program to simulate the working of Shortest remaining time and Round Robin (RR) scheduling algorithms and SRTF. with different quantum sizes for RR algorithm.	38
	Program 8 Design, develop and implement a C/C++/Java program to implement Banker's algorithm. Assume suitable input required to demonstrate the results.	41

8	Program 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.	44
	Viva Questions		48

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
- 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 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.
yyomore()	This function tells the lexer to append the next token to the current token.

1.1 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

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 .31417 But won't match 0 or 2

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 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.

Steps in writing yacc program

- Step 1 Using gedit editor create a file with extension y. For ex. gedit pgm1.y
- Step 2 lex pgm1.l
- Step 3 yacc -d pgm1.y
- Step 4 gcc lex.yy.c y.tab.c
- Step 5 ./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.

Structure of Yacc source program

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 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 associativity of a terminal symbol.
%left	Specifies the left associativity of a Terminal Symbol
%right	Specifies the right associativity 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 non 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.

INTRODUCTION YO UNIX

Basic UNIX commands

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 copyrecursivelythroughdirectory 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 SYSTEM

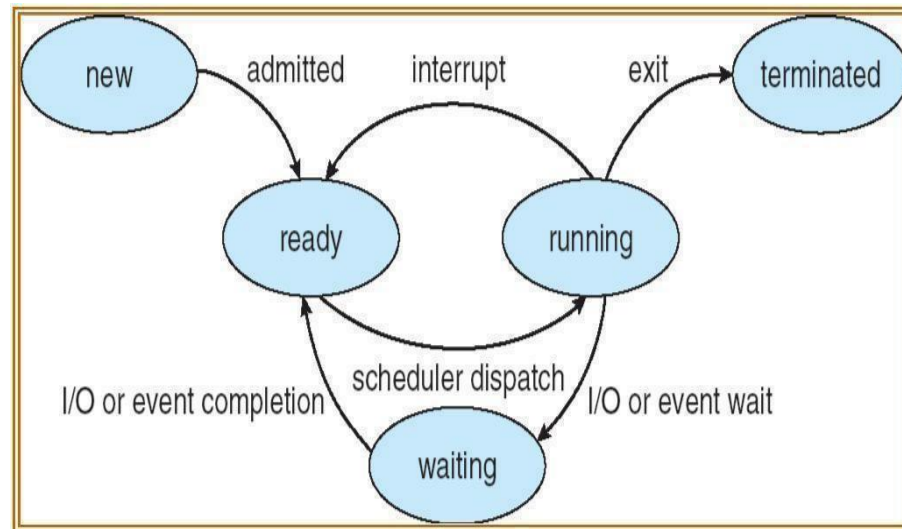
An Operating System is a program that manages the Computer hardware. It controls and coordinates the use of the hardware among the various application programs for the various users.

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

New: The process is being created

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.

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

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

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

Banker's algorithm:

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

The algorithm was developed in the design process for the operating system and originally described (in 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, **page replacement algorithms** decide which memory pages to page out, sometimes called swap out, or write to disk, when a page of memory needs to be allocated. Page replacement happens when a requested page is not in memory (page fault) and a free page cannot be used to satisfy the allocation, either because there are none, or because the number of free pages is lower than some threshold.

When the page that was selected for replacement and paged out is referenced again it has to be paged in (read in from disk), and this involves waiting for I/O completion. This determines the *quality* of the page replacement algorithm: the less time waiting for page-ins, the better the algorithm. A page replacement algorithm looks at the limited information about accesses to the pages provided by hardware, and tries to guess which pages should be replaced to minimize the total number of page misses, while balancing this with the costs (primary storage and processor time) of the algorithm itself. The page replacing problem is a typical online problem from the competitive analysis perspective in the sense that the optimal deterministic algorithm is known.

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.

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 uses recursive 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 **r**ight 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.

Steps to compile and Run the Lex program

To Compile Lex Program: `lex file_name.l`

After the successful compilation of the lex program the `lex.yy.c` file is generated automatically

To Run Lex Program : `gcc lex.yy.c`

After the successful Run of the lex program the `./a.out` file is generated automatically

To see the output of Lex program type `./a.out` in the command prompt

Execute: `./a.out`

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

LEX Program

```
%{
#include<stdio.h>
#include<string.h>
int i=0,o=0,k,flag=0;
char id[10][10], op[10][10];
}%

%%
[0-9]+      {flag++;strcpy(id[i],yytext);i++;}
[+*]        {flag--;strcpy(op[o],yytext);o++;}
.|\n        {return 0;}
%%


int main()
{
printf("enter the expression: \n");
yylex();
if(flag!=1)
{
printf("\n Invalid expression: \n");
}
else
{
printf("\n Valid expression \n");
printf("\n Operators are: \n");
for(k=0;k<o;k++)
{
printf("%s\t",op[k]);
}
printf("\n Identifiers are: \n");
for(k=0;k<i;k++)
{
printf("%s\t",id[k]);
}
}
}

int yywrap()
{
return 1;
}
```

Execution commands

```
gedit 1a.1
lex 1a.1
gcc lex.yy.c
./a.out
```

OUTPUT



```
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
Type the arithmetic Expression
1+6
Valid Arithmetic Expression
No. of Operands/Identifiers : 2
No. of Additions : 1
No. of Multiplications : 0
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
Type the arithmetic Expression
(1+6)+(7*56)
Valid Arithmetic Expression
No. of Operands/Identifiers : 4
No. of Additions : 1
No. of Multiplications : 2
```

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

LEX Program

```
#include<stdio.h>

%{

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

%%

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

[(]          {return yytext[0];}
```

```
.                {;}

\n                {return 0;}

%%

yywrap()

}
```

YACC Program

```
%{
#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. Invalid
expression.\n");exit(0);}

else $$=$1/$3;}

| '(' exp ')' {$$=$2;}

| num {$$=$1;};

%%

int yyerror()

{
printf("Error. Invalid Expression.\n"); exit(0);
}

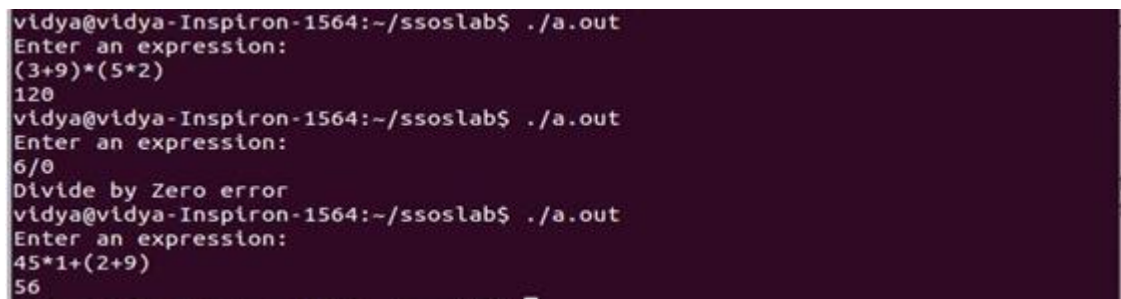
int main()
{
```

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

Execution commands

```
gedit 1b.1  
lex 1b.1  
gedit 1b.y  
yacc -d lex.yy.c  
gcc lex.yy.c y.tab.c  
./a.out
```

OUTPUT



```
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out  
Enter an expression:  
(3+9)*(5+2)  
120  
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out  
Enter an expression:  
6/0  
Divide by Zero error  
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out  
Enter an expression:  
45*1+(2+9)  
56
```

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 aⁿb

LEX Program

```
%{
#include<stdio.h>
#include<stdlib.h>
%{
#include "y.tab.h"

%}

%%

a          {return A;}

b          {return B;}
[\\n]      return '\\n';
%%
```

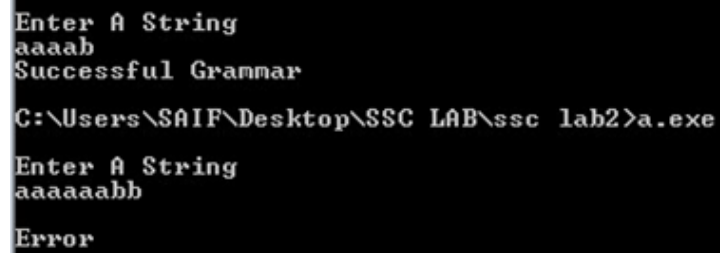
YACC Program

```
%{
#include<stdio.h>
#include<stdlib.h>
%}
%token A B
%%
input: s'\\n' {printf("Successful Grammar\\n");exit(0);}
      s: A s1
      B| B s1:
      ; | A s1
%%
int main()
{
    printf("\\nEnter A
String\\n"); yyparse();
}
int yyerror()
{
    printf("\\nError
\\n"); exit(0);
}
```

Execution commands

```
gedit 2.1  
lex 2.1  
gedit 2.y  
yacc -d lex.yy.c  
gcc lex.yy.c y.tab.c  
./a.out
```

OUTPUT



```
Enter A String  
aaaab  
Successful Grammar  
  
C:\Users\SAIF\Desktop\SSC LAB\ssc lab2>a.exe  
  
Enter A String  
aaaaaabb  
Error
```

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

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

char prod [3][10]={"A->aBa", "B->bB", "B->@"};
char first[3][10]={"a", "b", "@"};
char follow[3][10]={"$", "a", "a"};
char table[3][4][10];

char input[10];
int top=-1;
char stack[25];
char curp[20];

void push(char item)
{
    stack[++top]=item;
}
void pop()
{
    top=top-1;
}
void display()
{
    int i;
    for(i=top;i>=0;i--)
        printf("%c",stack[i]);
}

int numr(char c)
{
    switch(c)
    {
        case'A':return 1;
        case'B':return 2;
        case'a':return 1;
        case'b':return 2;
        case'@':return 3;
    }
    return 1;
}

int main()
```



```

{
    char c;
    int i,j,k,n;
    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            strcpy(table[i][j],"EMPTY");
        }
    }
    printf("\nGrammar\n");

    for(i=0;i<3;i++)
        printf("%s\n",prod[i]);

    printf("\nfirst={%s,%s,%s}",first[0],first[1],first[2]);
    printf("\nfollow={%s,%s}\n",follow[0],follow[1]);
    printf("\nPredictive parsing table for the given grammar :\n");

    strcpy(table[0][0],"");
    strcpy(table[0][1],"a");
    strcpy(table[0][2],"b");
    strcpy(table[0][3],"$");
    strcpy(table[1][0],"A");
    strcpy(table[2][0],"B");

    for(i=0;i<3;i++)
    {
        if(first[i][0]!='@')
            strcpy(table[numr(prod[i][0])][numr(first[i][0])],prod[i]);
        else
            strcpy(table[numr(prod[i][0])][numr(follow[i][0])],prod[i]);
    }
    printf("\n-----\n");
    for(i=0;i<3;i++){
        for(j=0;j<4;j++){
            {
                printf("%-30s",table[i][j]);
                if(j==3)
printf("\n-----\n");
            }
        }
    }

    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\t Input\tAction");
printf("\n-----\n");

while(input[i]!='$' && stack[top]!='$')
{
    display();
    printf("\t\t%s\t", (input+i));
    if(stack[top]==input[i])
    {
        printf("\tMatched %c\n", input[i]);
        pop();
        i++;
    }
    else
    {
        if(stack[top]>=65 && stack[top]<92)
        {
            strcpy(curp, table[numr(stack[top])][numr(input[i])]);
            if(!(strcmp(curp, "e")))
            {
                printf("\nInvalid String - Rejected\n");
                exit(0);
            }
        }
        else
        {
            printf("\tApply production %s\n", curp);
            if(curp[3]=='@')
            pop();
            else
            {
                pop();
                n=strlen(curp);
                for(j=n-1; j>=3; j--)
                    push(curp[j]);
            }
        }
    }
}

```

```
        }
    }
}

display();
printf("\t\t%s\t", (input+i));
printf("\n-----\n");
if(stack[top]=='$' && input[i]=='$')
{
    printf("\nValid String - Accepted\n");
}
else
{
    printf("Invalid String - Rejected\n");
}
}
```

Execution commands

```
cc 3.c
./a.out
```

OUTPUT

```
[test5@localhost ~]$ ./a.out

Grammar
A->aBa
B->bB
B->ε

first={a,b,ε}
follow={$,a}

Predictive parsing table for the given grammar :

-----
                a                b                $
-----
A                A->aBa                EMPTY                EMPTY
-----
B                B->ε                B->bB                EMPTY
-----

Enter the input string terminated with $ to parse:-abba$

Stack   Input   Action
-----
A$      abba$   Apply production A->aBa
aBa$    abba$   Matched a
Ba$     bba$    Apply production B->bB
bBa$    bba$    Matched b
Ba$     ba$     Apply production B->bB
bBa$    ba$     Matched b
Ba$     a$      Apply production B->ε
a$      a$      Matched a
$        $
-----

Valid String - Accepted
```

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

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

int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();

int main()
{
    puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
    puts("\nEnter 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]=='(' && 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();
        }
    }
}

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

Execution commands

```
cc 4.c
./a.out
```

OUTPUT

```

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@vict

```

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

T1 = -B

T2 = C + D

T3 = T1 * T2

A = T3

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

char op[2],arg1[5],arg2[5],result[5];
int 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);
}
```


Execution commands

```
gedit 5.c
cc 5.c
gedit input.txt
./a.out
cat input.txt
cat output.txt
```

Create a separate input.txt file

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

OUTPUT



```
vidya@vidya-Inspiron-1564:~/ssoslab$ ./a.out
vidya@vidya-Inspiron-1564:~/ssoslab$ cat output.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,R0vidya@vidya-Inspiron-1564:~/ssoslab$ █
```

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

LEX Program

```
%{
#include<stdio.h>
int sl=0;
int ml=0;
}%
%%
"/*" [a-zA-Z0-9' '\t\n]+ "*" ml++;
"//" .* sl++;
%%

main()
{
    yyin=fopen("f1.c","r");
    yyout=fopen("f2.c","w");
    yylex();
    fclose(yyin);
    fclose(yyout);
    printf("\n Number of single line comments are = %d\n",sl);
    printf("\nNumber of multiline comments are =%d\n",ml);
}
```

Execution commands

```
gedit 6a.1
lex 6a.1
gcc lex.yy.c
cat f1.c
cat f2.c
```

```
void main()
{
    int a;
    float bc;
    char c;
    char ch;

    if(a == 80)
        printf("Good");
    else
        printf("Bad");
}
```

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

LEX Program

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

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

YACC Program

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

%token DIGIT ID KEY OP

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

#include <stdio.h>
```

```
extern int yylex();
extern int yyparse();
extern FILE *yyin;
main()
{
    FILE *myfile = fopen("inputfile.c", "r");
    if (!myfile)
    {
        printf("I can't open inputfile.c!");
        return -1;
    }
    yyin = myfile;
    do{
        yyparse();
    }while (!feof(yyin));
    printf("numbers = %d\nKeywords = %d\nIdentifiers = %d\noperators\n\n",dig, key,id, op);
}

void yyerror()
{
    printf(" parse error! Message: ");
    exit(-1);
}
```

Execution commands

```
gedit 6a.1
gedit 6b.y
lex 6a.1
yacc -d lex.yy.c
cc lex.yy.c y.tab.c -ll
./a.out
```

OUTPUT

```
keyword is void  
identifier is main
```

```
keyword is int  
identifier is a
```

```
keyword is float  
identifier is bc
```

```
keyword is char  
identifier is c
```

```
keyword is char  
identifier is ch
```

```
keyword is if  
identifier is a  
operator is =  
operator is =  
numbers is 80
```

```
identifier is printf  
identifier is Good
```

```
keyword is else
```

```
identifier is printf  
identifier is Bad
```

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.

```
#include<stdio.h>
int main()
{
    int count,j,n,time,flag=0,time_quantum,ch=0;
    int wait_time=0,turnaround_time=0,at[10],bt[10],rt[10];
    int endTime,i,smallest;
    int remain=0,sum_wait=0,sum_turnaround=0;
    printf("1.Round Robin \n2.SRTF \n");
    scanf("%d",&ch);
    printf("Enter no of Processes : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter arrival time for Process P%d : ",i+1);
        scanf("%d",&at[i]);
        printf("Enter burst time for Process P%d :",i+1);
        scanf("%d",&bt[i]);
        rt[i]=bt[i];
    }
    switch(ch)
    {
        case 1:
            printf("Enter Time Quantum:\t");
            scanf("%d",&time_quantum);
            remain=n;
            printf("\nProcess time|Turnaround Time|Waiting Time\n");
            for(time=0,count=0;remain!=0;)
            {
                if(rt[count]<=time_quantum && rt[count]>0)
                {
                    time+=rt[count];
                    rt[count]=0;
                    flag=1;
                }
                else if(rt[count]>0)
                {
                    rt[count]-=time_quantum;
                    time+=time_quantum;
                }
                if(rt[count]==0 && flag==1)
                {
                    remain--;
                }
            }
        }
    }
```

```

        printf("P[%d]\t|\t%d\t|\t%d\n", count+1, time-at[count], time-
at[count]-bt[count]);

        wait_time+=time-at[count]-bt[count];
        turnaround_time+=time-at[count];
        flag=0;
    }
    if(count==n-1)
        count=0;
    else if(at[count+1]<=time)
        count++;
    else
        count=0;
}
printf("\nAverage Waiting Time=
%.2f\n", wait_time*1.0/n);
printf("Avg Turnaround Time =
%.2f\n", turnaround_time*1.0/n);
break;
case 2:
    remain=0;
    printf("\nProcesst|Turnaround Time| Waiting
Timen\n");
    rt[9]=9999;
    for(time=0; remain!=n; time++)
    {
        smallest=9;
        for(i=0; i<n; i++)
            if(at[i]<=time && rt[i]<rt[smallest] &&
rt[i]>0)
                smallest=i;
        rt[smallest]--;
        if(rt[smallest]==0)
        {
            remain++;
            endTime=time+1;

            printf("\nP[%d]\t|\t%d\t|\t%d", smallest+1, endTime-
at[smallest], endTime-bt[smallest]-at[smallest]);
            printf("\n");
            sum_wait+=endTime-bt[smallest]-
at[smallest];
            sum_turnaround+=endTime-at[smallest];
        }
    }
    printf("\nAverage waiting time =
%f\n", sum_wait*1.0/n);

```

```

        printf("Average Turnaround time =
%f",sum_turnaround*1.0/n);
        break;
    default:
        printf("Invalid\n");
    }
    return 0;
}

```

Execution commands

cc 7.c

./a.out

OUTPUT

```

1.Round Robin
2.SRTF
2
Enter no of Processes : 1
Enter arrival time for Process P1 : 1
Enter burst time for Process P1 :5
Process time!Turnaround Time!Waiting Time
P[1] : 5 : 0
Average waiting time = 0.000000
Average Turnaround time = 5.000000
C:\Users\SAIF\Desktop\SSC LAB\ssc lab7>a.exe
1.Round Robin
2.SRTF
1
Enter no of Processes : 1
Enter arrival time for Process P1 : 1
Enter burst time for Process P1 :5
Enter Time Quantum: 4
Process time!Turnaround Time!Waiting Time
P[1] : 4 : -1
Average Waiting Time= -1.00
Avg Turnaround Time = 4.00
C:\Users\SAIF\Desktop\SSC LAB\ssc lab7>a.exe
1.Round Robin
2.SRTF
1
Enter no of Processes : 3
Enter arrival time for Process P1 : 1
Enter burst time for Process P1 :5
Enter arrival time for Process P2 : 2
Enter burst time for Process P2 :7
Enter arrival time for Process P3 : 2
Enter burst time for Process P3 :5
Enter Time Quantum: 4
Process time!Turnaround Time!Waiting Time
P[1] : 12 : 7
P[2] : 14 : 7
P[3] : 15 : 10
Average Waiting Time= 8.00
Avg Turnaround Time = 13.67

```


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

```
#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("Enter the no of resources : ");
    scanf("%d", &r);
    printf("Enter 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("Enter 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("Enter 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("Max matrix:\t\nAllocation 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("Process %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("The system is in a safe state!!\n");
    printf("Safe Sequence : < ");
    for( i = 0; i < p; i++)
        printf("%d ", safeSequence[i]);
    printf(">\n");
}
else
    printf("The system is in an unsafe state!!");
}

```

Execution commands

```
cc 8.c
```

```
./a.out
```

OUTPUT

```
Enter the no of processes : 3
Enter the no of resources : 2
Enter the Max Matrix for each process :
For process 1 : 2 1
For process 2 : 3 1
For process 3 : 4 7
Enter the allocation for each process :
For process 1 : 2 1
For process 2 : 2 2
For process 3 : 2 5
Enter the Available Resources : 2 7
Max matrix:
Allocation matrix:
2 1      2 1
3 1      2 2
4 7      2 5
Process 1 runs to completion!Max matrix:
Allocation matrix:
0 0      0 0
3 1      2 2
4 7      2 5
Process 2 runs to completion!Max matrix:
Allocation matrix:
0 0      0 0
0 0      0 0
4 7      2 5
Process 3 runs to completion!The system is in a safe state!!
Safe Sequence : < 1 2 3 >
```

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);
void lru(char [ ],char [ ],int,int);
void opt(char [ ],char [ ],int,int);

int main()
{
    int ch,YN=1,i,l,f;
    char F[10],s[25];
    printf("\nEnter the no of empty frames: ");
    scanf("%d",&f);
    printf("\nEnter the length of the string: ");
    scanf("%d",&l);
    printf("\nEnter the string: ");
    scanf("%s",s);
    for(i=0;i<f;i++)
        F[i]=-1;

    do
    {
        printf("\n***** MENU *****");
        printf("\n1:FIFO\n2:LRU \n3:EXIT");
        printf("\nEnter your choice: ");
        scanf("%d",&ch);

        switch(ch)
        {
            case 1: for(i=0;i<f;i++)
                    F[i]=-1;
                    FIFO(s,F,l,f);
                    break;

            case 2: for(i=0;i<f;i++)
                    F[i]=-1;
                    lru(s,F,l,f);
                    break;

            case 3: exit(0);
        }
    }
```

```

        printf("\n\nDo u want to continue IF YES PRESS 1\nIF NO
PRESS 0 : ");
        scanf("%d",&YN);
    } while(YN==1);
    return(0);
}

//FIFO
void FIFO(char s[],char F[],int l,int f)
{
    int i,j=0,k,flag=0,cnt=0;
    printf("\n\tPAGE\t\t\tFRAMES\t\t\t\tFAULTS");
    for(i=0;i<l;i++)
    {
        for(k=0;k<f;k++)
        {
            if(F[k]==s[i])
                flag=1;
        }

        if(flag==0)
        {
            printf("\n\t%c\t",s[i]);
            F[j]=s[i];
            j++;
            for(k=0;k<f;k++)
                printf(" %c",F[k]);
            printf("\tPage-fault%d",cnt);
            cnt++;
        }

        else
        {
            flag=0;
            printf("\n\t%c\t",s[i]);
            for(k=0;k<f;k++)
                printf(" %c",F[k]);
            printf("\tNo page-fault");
        }
        if(j==f)
            j=0;
    }
}

//LRU
void lru(char s[],char F[],int l,int f)
{
    int i,j=0,k,m,flag=0,cnt=0,top=0;

```

```
printf("\n\tPAGE\t\t\t\t\tFRAMES\t\t\t\t\tFAULTS");  
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;
```

```
}  
}
```

Execution commands

```
cc 9.c
```

```
./a.out
```

OUTPUT

```
Enter the no of empty frames: 2  
Enter the length of the string: 5  
Enter the string: hello  
***** MENU *****  
1:FIFO  
2:LRU  
3:EXIT  
Enter your choice: 1  


| PAGE | FRAMES | FAULTS        |
|------|--------|---------------|
| h    | h      | Page-fault0   |
| e    | h e    | Page-fault1   |
| l    | l e    | Page-fault2   |
| l    | l e    | No page-fault |
| o    | l o    | Page-fault3   |

  
Do u want to continue IF YES PRESS 1  
IF NO PRESS 0 : 1  
***** MENU *****  
1:FIFO  
2:LRU  
3:EXIT  
Enter your choice: 2  


| PAGE | FRAMES | FAULTS        |
|------|--------|---------------|
| h    | h      | Page-fault0   |
| e    | h e    | Page-fault1   |
| l    | e l    | Page-fault2   |
| l    | e l    | No page fault |
| o    | l o    | Page-fault3   |

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

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 lower-level 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 its input and produces either a corresponding Parse tree or an Error.

What is the Syntax of a Language?

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

What is the Semantics of a Language?

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

What are tokens?

When a string representing a program is broken into a 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 we need 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 Leaf represents the Terminal Symbol.

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

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 in 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.

