VISVESVARAYA TECHNOLOGICAL UNIVERSITY

Jnana Sangama, Belagavi - 590 018, Karnataka

RNS Institute of Technology

Dr. Vishnuvardana Road, Channasandra

Bengaluru – 560098

DEPARTMENT
of
COMPUTER SCIENCE & ENGINEERING

LABORATORY MANUAL

System Software Laboratory 18CSL66

Session: April 2022 - Sep 2022

VI Semester (2019-2023 Batch)

Name: _____

USN: _____

Batch:_____ **Section:**_____

SYSTEM SOFTWARE LABORATORY (18CSL66)
INTERNAL EVALUATION SHEET

EVALUATION (MAX MARKS 40)			
TEST A	**REGULAR EVALUATION B**	**RECORD C**	**TOTAL MARKS A+B+C**
10	**10**	**20**	**40**

R1: REGULAR LAB EVALUATION WRITE UP RUBRIC (MAX MARKS 10)				
Sl. No.	**Parameters**	**Good**	**Average**	**Needs improvement**
a.	**Understanding of problem (3 marks)**	Clear understanding of problem statement while designing and implementing the program (3)	Problem statement is understood clearly but few mistakes while designing and implementing program (2)	Problem statement is not clearly understood while designing the program (1)
b.	**Writing program (4 marks)**	Program handles all possible conditions (4)	Average condition is defined and verified. (3)	Program does not handle possible conditions (1)
c.	**Result and documentation (3 marks)**	Meticulousdocumentation and all conditions are taken care (3)	Acceptable documentation shown (2)	Documentation does not take care all conditions (1)

R2: REGULAR LAB EVALUATION VIVA RUBRIC (MAX MARKS 10)					
Sl. No.	**Parameter**	**Excellent**	**Good**	**Average**	**Needs Improvement**
a.	**Conceptual understanding (10 marks)**	Answers 80% of the viva questions asked (10)	Answers 60% of the viva questions asked (7)	Answers 30% of the viva questions asked (4)	Unable to relate the concepts (1)

R3: REGULAR LAB PROGRAM EXECUTION RUBRIC (MAX MARKS 10)				
Sl. No.	**Parameters**	**Excellent**	**Good**	**Needs Improvement**
a.	**Design, implementation, and demonstration (5 marks)**	Program follows syntax and semantics of C programming language. Demonstrates the complete knowledge of the program written (5)	Program has few logical errors, moderately demonstrates all possible concepts implemented in programs (3)	Syntax and semantics of C programming is not clear (1)
b.	**Resultand documentation (5 marks)**	All test cases are successful, all errors are debugged with own practical knowledge and clear documentation according to the guidelines (5)	Moderately debugs the programs , few test case are unsuccessful and Partial documentation (3)	Test cases are not taken care , unable to debug the errors and no proper documentation (1)

R4: RECORD EVALUATION RUBRIC (MAX MARKS 20)					
Sl. No.	**Parameter**	**Excellent**	**Good**	**Average**	**Needs Improvement**
a.	**Documentation (10 marks)**	Meticulous record writing including program, comments and as per the guidelines mentioned (20)	Write up contains program, but comments are not included (18)	Write up contains only program (15)	Program written with few mistakes (10)

Table of Contents

SYLLABUS

SYSTEM SOFTWARE LABORATORY
(Effective from the academic year 2018 -2019)
SEMESTER – VI

Course Code	**18CSL66**	**CIE Marks**	40
Number of Contact Hours/Week	0:2:2	**SEE Marks**	60
Total Number of Lab Contact Hours	36	**Exam Hours**	03

Credits – 2

Course Learning Objectives: This course (18CSL66) will enable students to:

- To make students familiar with Lexical Analysis and Syntax Analysis phases of Compiler Design and implement programs on these phases using LEX & YACC tools and/or C/C++/Java
- To enable students to learn different types of CPU scheduling algorithms used in operating system.
- To make students able to implement memory management - page replacement and deadlock handling algorithms

Descriptions (if any):

Exercises to be prepared with minimum three files (Where ever necessary):

1. Header file.
2. Implementation file.
3. Application file where main function will be present.

The idea behind using three files is to differentiate between the developer and user sides. In the developer side, all the three files could be made visible. For the user side only header file and application files could be made visible, which means that the object code of the implementation file could be given to the user along with the interface given in the header file, hiding the source file, if required. Avoid I/O operations (printf/scanf) and use ***data input file*** where ever it is possible.

Programs List:

Installation procedure of the required software must be demonstrated, carried out in groups and documented in the journal.

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.
b.	Write YACC program to evaluate ***arithmetic expression*** involving operators: +, -, *, and /
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)
3.	Design, develop and implement YACC/C program to construct ***Predictive / LL(1) Parsing Table*** for the grammar rules: $A \rightarrow aBa$, $B \rightarrow bB \mid \varepsilon$. Use this table to parse the sentence: ***abba$***
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***.
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***
6.	

a.	Write a LEX program to eliminate ***comment lines*** in a ***C*** program and copy the resulting program into a separate file.
b.	Write YACC program to recognize valid ***identifier, operators and keywords*** in the given text (*C program*) 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.
8.	Design, develop and implement a C/C++/Java program to implement Banker‟s algorithm. Assume suitable input required to demonstrate the results
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.

Laboratory Outcomes: The student should be able to:

- Implement and demonstrate Lexer‟s and Parser‟s
- Evaluate different algorithms required for management, scheduling, allocation and communication used in operating system.

Conduct of Practical Examination:

- Experiment distribution
  - For laboratories having only one part: Students are allowed to pick one experiment from the lot with equal opportunity.
  - For laboratories having PART A and PART B: Students are allowed to pick one experiment from PART A and one experiment from PART B, with equal opportunity.
- Change of experiment is allowed only once and marks allotted for procedure to be made zero of the changed part only.
- Marks Distribution *(Courseed to change in accoradance with university regulations)*
  m) For laboratories having only one part – Procedure + Execution + Viva-Voce: 15+70+15 = 100 Marks
  n) For laboratories having PART A and PART B
    i. Part A – Procedure + Execution + Viva = 6 + 28 + 6 = 40 Marks
    ii. Part B – Procedure + Execution + Viva = 9 + 42 + 9 = 60 Marks

COURSE OBJECTIVES and COURSE OUTCOMES

Course objectives: This course will enable students to achieve the following:

- To make students familiar with Lexical Analysis and Syntax Analysis phases of Compiler Design and implement programs on these phases using LEX & YACC tools and/or C/C++/Java
- To enable students to learn different types of CPU scheduling algorithms used in operating system
- To make students able to implement memory management - page replacement and deadlock handling algorithms

Course outcomes: At the end of the course the students should be able to:

CO 1: Demonstrate the working of Lexer.

CO 2: Implement various parsing techniques.

CO 3: Implement a code generator

CO 4: Implement and analyze various CPU scheduling algorithms.

CO 5: Implement resource allocation and deadlock avoidance algorithm.

CO 6: Evaluate different algorithms required for management of page allocation techniques.

CO-PO MAPPING MATRIX													CO-PSO MAPPING MATRIX			
CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3	PSO4
CO1	3	1	3	2	3	-	-	-	-	-	-	1	-	-	1	-
CO2	3	3	3	2	3	-	-	-	-	-	1	2	2	1	2	-
CO3	3	3	3	2	1	-	-	-	-	-	1	3	3	1	3	-
CO4	3	2	3	2	1	-	-	-	1	-	-	3	2	1	1	-
CO5	3	2	3	2	1	-	-	-	1	-	-	3	1	1	1	-
CO6	3	2	3	2	1	-	-	-	1	-	-	2	1	1	2	-

INTRODUCTION

LEX

Lex stands for Lexical Analyzer. Lex is a tool for generating scanners. Scanners are programs that recognize lexical patterns in text. These lexical patterns (or regular expressions) are defined in a particular syntax. A matched regular expression may have an associated action. This action may also include returning a token. When Lex receives input in the form of a file or text, it attempts to match the text with the regular expression. It takes input one character at a time and continues until a pattern is matched. If a pattern can be matched, then Lex performs the associated action (which may include returning a token). If, on the other hand, no regular expression can be matched, further processing stops and Lex displays an error message. Lex and C are tightly coupled. A *.lex* file (files in Lex have the extension *.lex*) is passed through the lex utility, and produces output files in C. These file(s) are compiled to produce an executable version of the lexical analyzer.

Regular Expressions in Lex

A regular expression is a pattern description using a meta language. 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.

Defining regular expressions in Lex

Character	Meaning
A-Z, 0-9, a-z	Characters and numbers that form part of the pattern.
.	Matches any character except \n.
-	Used to denote range. Example: A-Z implies all characters from A to Z.
[ ]	
*****	Match *zero* or more occurrences of the preceding pattern.
+	Matches *one* or more occurrences of the preceding pattern.
?	Matches *zero or one* occurrences of the preceding pattern.
$	Matches end of line as the last character of the pattern.
{ }	Indicates how many times a pattern can be present. Example: A{1,3} implies one or three occurrences of A may be present.

\	Used to escape meta characters. Also used to remove the special meaningof characters as defined in this table.
^	Negation.
|	Logical OR between expressions.
"<some symbols>"	Literal meanings of characters. Meta characters hold.
/	Look ahead. Matches the preceding pattern only if followed by the succeeding expression. Example: A0/1 matches A0 only if A01 is the input.
( )	Groups a series of regular expressions.

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. This interaction, summarized schematically in Fig-1.1, is commonly implemented by making the lexical analyzer be a subroutine or a co routine of the parser.

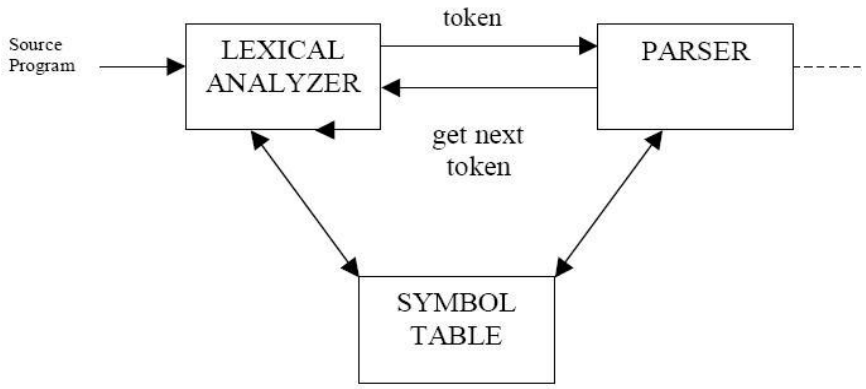


Fig 1.1: Interaction of Lexical Analyzer with Parser

A Language for Specifying Lexical Analyzer

Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expressions. In this section, we describe a particular tool, called *Lex* that has been widely used to specify lexical analyzers for a variety of languages.

We refer to the tool as *Lex compiler*, and its input specification as the *Lex language*. Lex is generally used in the manner depicted in Fig 1.2. First, a specification of a lexical analyzer is prepared by creating a program *lex.l* in the lex language. Then, *lex.l* is run through the Lex compiler to produce a C program *lex.yy.c.* The program *lex.yy.c* consists of a tabular representation of a transition diagram constructed from the regular expression of *lex.l,* together with a standard routine that uses the table to recognize lexemes. The actions associated with regular expression in *lex.l* are pieces of C code and are carried over directly to *lex.yy.c*. Finally*lex.yy.c*is run through the C compiler to produce an object program*a.out*, which is the lexicalanalyzer that transforms an input stream into a sequence of tokens.

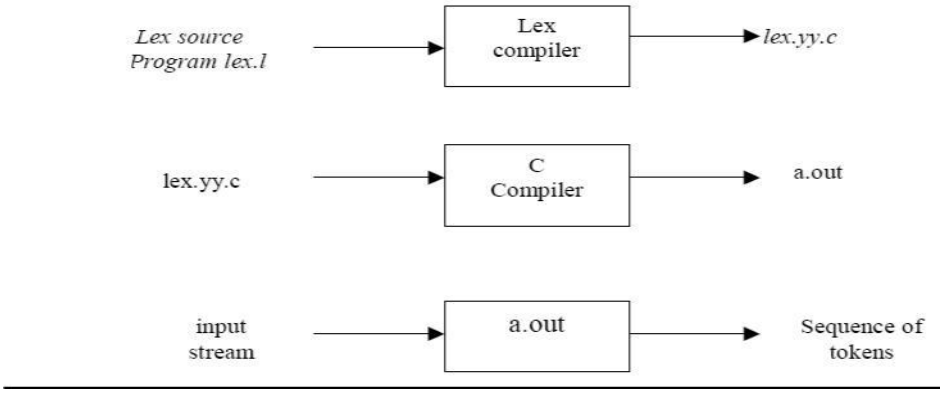


Fig 1.2 Creating a lexical analyzer with Lex

GENERAL STRUCTURE OF LEX

A Lex program consists of three parts:

Declaration Section

%%

Rules Section

%%

auxiliary procedures

The declarations section includes declarations of variables, constants and regular definitions.
The Rules sections of a lex program are statements of the form

Pattern-R1	{*action1*}
Pattern-R2	{*action2*}
....	
Pattern-Rn	{*action n*}

where each R*i* is regular expression and each *action i*, is a program fragment describing what action the lexical analyzer should take when pattern Ri matches lexeme. Typically, *action i* will return control to the parser. In Lex actions are written in C; in general, however, they can be in any implementation language. The third section holds whatever auxiliary procedures are needed by the actions.

The Role of Parser

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input. We know that programs can contain errors at many different levels. For example, errors can be

1. Lexical, such as misspelling an identifier, keyword, or operator
2. Syntactic, such as arithmetic expression with unbalanced parentheses
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as infinitely recursive call.

Often much of the error detection and recovery in a compiler is centered around the syntax analysis phase.

Advanced Lex

Lex has several functions and variables that provide different information and can be used to build programs that can perform complex functions. Some of these variables and functions, along with their uses, are listed in the following tables.

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 thisfunction returns 1, the parsing stops. So, this can be used to parse multiplefiles. Code can be written in the third section, which will allow multiplefiles to be parsed. The strategy is to make yyin file pointer (see thepreceding table) point to a different file until all the files are parsed. Atthe 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 theread token.
yymore()	This function tells the lexer to append the next token to the current token.

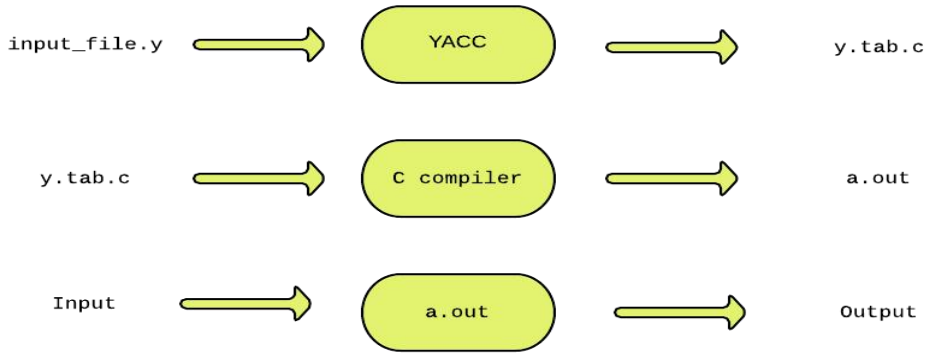
Execution Procedure for LEX programs:

- Edit the lex program using **gedit**with .l extension i.e**$geditfilename.l**
- After editing, save the file.
- Once the program is saved, compile the lex program to get lex.yy.c file using the following command: **lex filename.l**
- If there is any error in the rules section those errors will be displayed on the terminal, to correct those errors edit the program using **gedit**.
- Compile lex.yy.c file generated by the lexer by using cc command as shown: **$ cc lex.yy.c -ll**

- If there is any errors in the program those errors will be displayed, to correct those errors edit the program using gedit.
- To run the compiled program use the command: **$ ./a.out**

YACC

YACC (**Y**et **A**nother **C**ompiler **C**ompiler) is a tool used to generate a parser. This document is a tutorial for the use of YACC to generate a parser for ExpL. YACC translates a given Context Free Grammar (CFG) specifications (input in input_file.y) into a C implementation (y.tab.c) of a corresponding push down automaton (i.e., a finite state machine with a stack). This C program when compiled yields an executable parser.



The source SIL program is fed as the input to the generated parser (a.out). The *parser* checks whether the program satisfies the syntax specification given in the inputfile.y file.

YACC was developed by Stephen C. Johnson at Bell labs.

Parser:
A parser is a program that checks whether its input (viewed as a stream of tokens) meets a given grammar specification. The syntax of SIL can be specified using a Context Free Grammar. As mentioned earlier, YACC takes this specification and generates a parser for SIL.

Structure of a YACC source program

A YACC source program is structurally similar to a LEX one.

Declarations
%%
Rules
%%
Routines

The declaration section may be empty. Moreover, if the routines section is omitted, the second %% mark may be omitted also.

Blanks, tabs, and newlines are ignored except that they may not appear in names.

THE DECLARATIONS SECTION may contain the following items.

Declarations of tokens. YACC requires token names to be declared as such using the keyword **%token.**

Declaration of the start symbol using the keyword **%start**

C declarations: included files, global variables, types.

C code between **%{ and %}**.

RULES SECTION. A rule has the form:

```
nonterminal : sentential form
            | sentential form
            ..................
            | sentential form
            ;
```

Actions may be associated with rules and are executed when the associated sentential form is matched.

LEX-YACC INTERACTION

yyparse() calls yylex() when it needs a new token.

LEX	YACC
return(TOKEN)	%token TOKEN
	TOKEN is used in production

The external variable **yylval** is used in a LEX source program to return values of lexemes,

yylval is assumed to be integer if you take no other action. Changes related to yylval must be made in the definitions section of YACC specification by adding new types in the following way

```
%union {
    (type fieldname)
    (type fieldname)
    ..............
```

```
}
```

and defining which token and non-terminals will use these types

```
%token <fieldname> token

%type <fieldname> non-terminal
```

in LEX specification by using the fieldnames in the assignment as follows

```
yylval.fieldname = ..........
```

If you need a record type, then add it in the union. Example:

```
%union {
   struct s {
      double fvalue;
      int ivalue;
   } t;
}
```

in the LEX specification use the record name and record field in assignments:

```
yylval.t.ivalue = ......
```

in the YACC rules specification use the record field only in the assignment:

```
$1.ivalue = ......
```

assuming that $1 has the appropriate type, whatever it denotes.

Input File:

If yylex() is not defined in the auxiliary routines sections, then it should be included: **#include "lex.yy.c"**

YACC input file generally finishes with: **.y**

Output Files:

- The output of YACC is a file named **y.tab.c**
- If it contains the main() definition, it must be compiled to be executable.

- Otherwise, the code can be an external function definition for the function **int yyparse()**
- If called with the –d option in the command line, Yacc produces as output a header file **y.tab.h**with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the **–v option,** Yacc produces as output a file**y.output**containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves conflicts.

LABORATORY PROGRAMS

1. a) Program to recognize a valid arithmetic expression and to recognize the identifiers and operators present. Print them separately.

```
%{
int a[]={0,0,0,0},i, valid=1,opnd=0;
%}
%x OPER
%%
[a-zA-Z0-9]+ { BEGIN OPER; opnd++;}
<OPER>"+" { if(valid) { valid=0;i=0;} else ext();}
<OPER>"-" { if(valid) { valid=0;i=1;} else ext();}
<OPER>"*" { if(valid) { valid=0;i=2;} else ext();}
<OPER>"/" { if(valid) { valid=0;i=3;} else ext();}
<OPER>[a-zA-Z0-9]+ { opnd++; if(valid==0) { valid=1; a[i]++;}
else ext();}
<OPER>"\n" { if(valid==0) ext(); else return 0;}
.\n ext();
%%
ext()
{ printf(" Invalid Expression \n"); exit(0); }
main()
{
printf(" Type the arithmetic Expression \n");
yylex();
printf(" Valid Arithmetic Expression \n");

printf(" No. of Operands/Identifiers : %d \n ",opnd);

printf(" No. of Additions : %d \n No. of Subtractions : %d
\n",a[0],a[1]);
printf(" No. of Multiplications : %d \n No. of Divisions : %d
\n",a[2],a[3]);
}
```

TEST CASES

Test No	Input Parameters	Expected Output	Obtained Output
1	Type the arithmetic Expression a+b-c*d/e	Valid Arithmetic Expression No. of Operands/Identifiers : 5 No. of Additions : 1 No. of Subtractions : 1 No. of Multiplications : 1 No. of Divisions : 1	Valid Arithmetic Expression No. of Operands/Identifiers : 5 No. of Additions : 1 No. of Subtractions : 1 No. of Multiplications : 1 No. of Divisions : 1
2	Type the arithmetic Expression a-b+c*d	Valid Arithmetic Expression No. of Operands/Identifiers : 4 No. of Additions : 1 No. of Subtractions : 1 No. of Multiplications : 1 No. of Divisions : 0	Valid Arithmetic Expression No. of Operands/Identifiers : 4 No. of Additions : 1 No. of Subtractions : 1 No. of Multiplications : 1 No. of Divisions : 0

b) Program to evaluate an arithmetic expression involving operators +, -, * and /.

Lex Part

```
%{
#include<stdlib.h>
#include "y.tab.h"
extern int yylval;
%}
%%
[0-9]+ { yylval=atoi(yytext); return NUM; }
[\t] ;
\n return 0;
. return yytext[0];
%%
```

Yacc Part

```
%{
#include <stdio.h>
/* Yacc Program to evaluate a valid arithmetic Expression*/
%}
%token NUM
%left '+' '-'
%left '*' '/'
%%
expr : e { printf(" Result : %d\n",$1); return 0;};
e : e '+' e {$$=$1+$3;}
| e '-' e {$$=$1-$3;}
| e '*' e {$$=$1*$3;}
| e '/' e {$$=$1/$3;}
| '('e')' {$$=$2;}
| NUM {$$=$1;}
;
%%
main()
{
printf(" Type the Expression & Press Enter key\n");
yyparse();
printf(" Valid Expression \n");
}
yyerror()
{
printf(" Invalid Expresion!!!!\n");
exit(0); }
```

Test No	Input Parameters	Expected Output	Obtained Output
1	Type the Expression & Press Enter key 10+20*3	Result : 70 Valid Expression	Result : 70 Valid Expression
2	Type the Expression & Press Enter key 50+30-2*12	Result : 56 Valid Expression	Result : 56 Valid Expression

2. Program to recognize the grammar $a^n b^n$.

Lex Part

```
%{
#include "y.tab.h"
%}
%%
a return A;
b return B;
. return yytext[0];
\n return yytext[0];
%%
```

Yacc part

```
%{
/* Yacc program to recognize the grammar AnB */
#include<stdio.h>
#include<stdlib.h>
int yyerror();

%}
%token A B
%%
str: s'\n' {printf("Valid String\n");}
;
s : x B
;
x : x A | A
;
%%
int main()
{
printf(" Type the String ? \n");
if(!yyparse())
printf(" Valid String\n ");
}
int yyerror()
{
printf(" Invalid String.\n");
exit(0);
}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	Type the String ? aaab	Valid String	Valid String
2	Type the String ? aaabb	Invalid String.	Invalid String.
3	Type the String ? aaaaaab	Valid String	Valid String

3.Design, develop and implement C program to construct Predictive / LL(1) Parsing

Table for the grammar rules:
A ->aBa , B ->bB | '\0'. Use this table to parse the sentence: abba$.

LL(1) parsing is a top-down parsing method in the syntax analysis phase of compiler design. Required components for LL(1) parsing are input string, a stack, parsing table for given grammar, and parser. Here, we discuss a parser that determines that given string can be generated from a given grammar(or parsing table) or not.
Let given grammar is G = (V, T, S, P)where V-variable symbol set, T-terminal symbol set, S-start symbol, P- production set.

LL(1) Parser algorithm:
Input- 1. stack = S //stack initially contains only S.
2. input string = w$
where S is the start symbol of grammar, w is given string, and $ is used for the end of string.
3. PT is a parsing table of given grammar in the form of a matrix or 2D array.

Output- determines that given string can be produced by given grammar(parsing table) or not, if not then it produces an error.

Steps:

```
1.    while(stack is not empty) Ein
   {
        // initially it is S
2.      A = top symbol of stack;
     //initially it is first symbol in string, it can be $ also
3.      r = next input symbol of given string;
4.      if (A∈T or A==$) {
5.          if(A==r){
6.              pop A from stack;
7.              remove r from input;
8.          }
9.          else
10.              ERROR();
11.      }
12.      else if (A∈V) {
13.          if(PT[A,r]= A⇢B1B2....Bk) {
14.              pop A from stack;
                 // B1 on top of stack at final of this step
15.              push Bk,Bk-1......B1 on stack
16.          }
17.          else if (PT[A,r] = error())
18.              error();
```

```
19.      }
20. }
// if parser terminate without error()
// then given string can generated by given parsing table.
```

Example –

Let the grammar $G = (V, T, S', P)$ is

$S' \rightarrow \boldsymbol{S\$}$

$S \rightarrow xYzS \mid a$

$Y \rightarrow xYz \mid y$

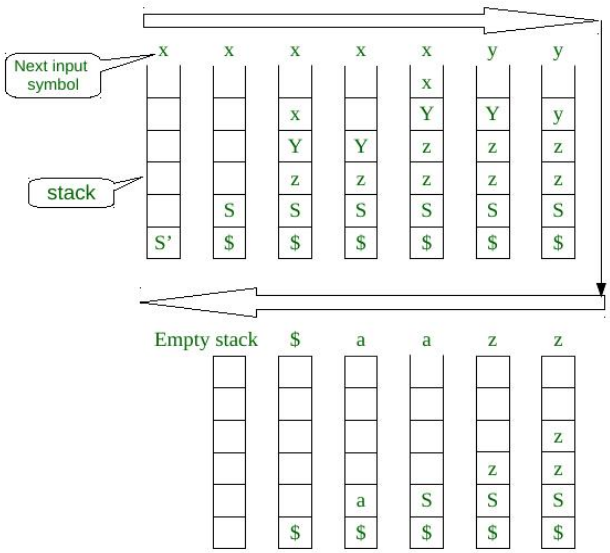
Parsing table(*PT*) for this grammar

	a	***x***	***y***	***z***	***$***
S'	*S' → S$*	*S' → S$*	*error*	*error*	*error*
S	*S → a*	*S → xYzS*	*error*	*error*	*error*
Y	*error*	*Y → xYz*	*Y → y*	*error*	*error*

Let *string1* = *xxyzza*,

We have to add *$* with this string,

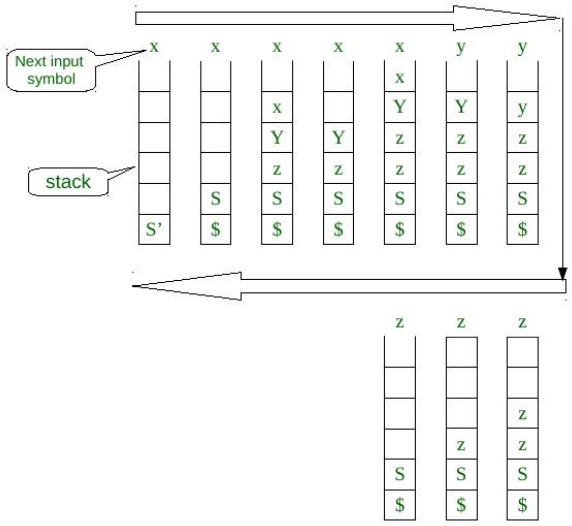
We will use the above parsing algorithm, diagram for the process :



For *string1* we got an empty stack, and while loop or algorithm terminated without error. So, *string1* belongs to language for given grammar *G*.

Let *string2* = *xxyzzz*,

Same way as above we will use the algorithm to parse the string2, here is the diagram



For *string2*, at the last stage as in the above diagram when the top of the stack is *S* and the next input symbol of the string is *z*, but in *PT[S,z] = error*. The algorithm terminated with an error. So, *string2* is not in the language of grammar *G*.

```
/* A ->aBa , B ->bB | @*/
```

```
#include<stdio.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]);
}

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);
}
void main()
{
char c;
int i,j,k,n;
for(i=0;i<3;i++)
for(j=0;j<4;j++)
strcpy(table[i][j],"e");
printf("\n Grammar:\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++)

{
k=strlen(first[i]);
for(j=0;j<k;j++)
if(first[i][j]!='@')

strcpy(table[numr(prod[i][0])][numr(first[i][j])],prod[i]);
else
strcpy(table[numr(prod[i][0])][numr(follow[i][j])],prod[i]);
}
printf("\n-----------------------------------------------------
---\n");
for(i=0;i<3;i++)
for(j=0;j<4;j++)
{
printf("%-10s",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("\n invalid 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("\n valid string - Accepted\n");
}
else
{
printf("\ninvalid string- Rejected\n");
}
}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	abba$	Grammar: A->aBa B->bB B->@ first= {a,b,@} follow ={$ a} Predictive parsing table for the given grammar --- a b $ --- A A->aBa e e --- B B->@ B->bB e --- 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->@ 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 ->E+T | T, T->T*F | F, F -> (E) | id and parse the sentence: id + id * id.

AIM: To write a 'C' Program to implement for the Shift Reduce parser for thegivengrammar,

ALGORITHM/PROCEDURE:

1. **Get the input expression and store it in the input buffer.**
2. **Read the data from the input buffer one at the time.**
3. **Using stack and push & pop operation shift and reduce symbols with respect to production rules available.**
4. **Continue the process till symbol shift and production rule reduce reaches the start symbol.**
5. **Display the Stack Implementation table with corresponding Stack actions with**

input symbols.

Problem-01:

Consider the following grammar-

$$E \rightarrow E - E$$
$$E \rightarrow E \; x \; E$$
$$E \rightarrow id$$

Parse the input string id – id x id using a shift-reduce parser.

Solution-

Stack	Input Buffer	Parsing Action
$	id – id x id $	Shift
$ id	– id x id $	Reduce E → id
$ E	– id x id $	Shift
$ E –	id x id $	Shift
$ E – id	x id $	Reduce E → id
$ E – E	x id $	Shift
$ E – E x	id $	Shift
$ E – E x id	$	Reduce E → id
$ E – E x E	$	Reduce E → E x E
$ E – E	$	Reduce E → E – E
$ E	$	Accept

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char ip_sym[15],stack[15];
int ip_ptr=0,st_ptr=0,len,i;
char temp[2];
char act[15];
void check();
void main()
{
printf("\n\t\t SHIFT REDUCE PARSER\n");
printf("\n GRAMMER\n");
printf("\n E->E+T|T\n T->T* F | F");
printf("\n F-> (E) | id \n ");
printf("\n enter the input symbol:\t");
scanf("%s",ip_sym);
printf("\n\t stack implementation table\n");
```

```
printf("\n stack\t\t input symbol\t\t action");
printf("\n_____\t\t _____\t\t _____\n");
printf("\n $\t\t%s$\t\t--",ip_sym); /*first step empty action
*/
strcpy(act,"shift ");
if (ip_sym[ip_ptr]=='(')
{
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
}
else
{
temp[0]=ip_sym[ip_ptr];
temp[1]=ip_sym[ip_ptr+1];
temp[2]='\0';
}
strcat(act,temp);
len=strlen(ip_sym);
for(i=0;i<=len-1;i++)
{
if(ip_sym[ip_ptr]=='i'&&ip_sym[ip_ptr+1]=='d')
{
stack[st_ptr]=ip_sym[ip_ptr];
st_ptr++;
ip_sym[ip_ptr]='';
ip_ptr++;
stack[st_ptr]= ip_sym[ip_ptr];
stack[st_ptr+1]='\0';
ip_sym[ip_ptr]='';
ip_ptr++;
}
else
{
stack[st_ptr]=ip_sym[ip_ptr];
stack[st_ptr+1]='\0';
ip_sym[ip_ptr]='';
ip_ptr++;
}
printf("\n $%s\t\t%s$\t\t%s",stack,ip_sym,act); /* second print
with action shift*/
strcpy(act,"shift ");
if (ip_sym[ip_ptr]=='(' || ip_sym[ip_ptr]=='*' ||
ip_sym[ip_ptr]=='+' || ip_sym[ip_ptr]==')')
{
temp[0]=ip_sym[ip_ptr];
temp[1]='\0';
}
else
{
temp[0]=ip_sym[ip_ptr];
temp[1]=ip_sym[ip_ptr+1];
temp[2]='\0';
```

```
}
strcat(act,temp);
check();
st_ptr++;
}
st_ptr++;
check();
}
void check()
{
int flag=0;
while(1)
{
if (stack[st_ptr]=='d'&& stack[st_ptr-1]=='i')
{
stack[st_ptr-1]='F';
stack[st_ptr]='\0';
st_ptr--;
flag=1;
printf("\n $%s\t\t%s$\t\tF->id",stack, ip_sym);
}
if (stack[st_ptr]==')'&& stack[st_ptr-1]=='E'&& stack[st_ptr-
2]=='(')
{
stack[st_ptr-2]='F';
stack[st_ptr-1]='\0';
flag=1;
st_ptr=st_ptr-2;
printf("\n $%s\t\t%s$\t\tF->id",stack, ip_sym);
}
if (stack[st_ptr]=='F'&& stack[st_ptr-1]=='*'&& stack[st_ptr-
2]=='T')
{
// stack[st_ptr-2]='T';
stack[st_ptr-1]='\0';
st_ptr= st_ptr-2;
flag=1;
printf("\n $%s\t\t%s$\t\tT->T*F",stack, ip_sym);
}
else
{
if (stack[st_ptr]=='F')
{
stack[st_ptr]='T';
flag=1;
printf("\n $%s\t\t%s$\t\tT->F",stack, ip_sym);
}
}
if( stack[st_ptr]=='T'&& stack[st_ptr-1]=='+'&& stack[st_ptr-
2]=='E'&& ip_sym[ip_ptr]!='*' )
{
//stack[st_ptr-2]='E';
```

```
stack[st_ptr-1]='\0';
st_ptr= st_ptr-2;
flag=1;
printf("\n $%s\t\t%s$\t\tE->E+T",stack, ip_sym);
}
else
if ((stack[st_ptr]=='T'&& ip_sym[ip_ptr]== '+') ||
(stack[0]=='T'&& ip_sym[ip_ptr]== '\0') ||
(stack[st_ptr]=='T'&& ip_sym[ip_ptr]== ')'))
{
stack[st_ptr]='E';
flag=1;
printf("\n $%s\t\t%s$\t\tE->T",stack, ip_sym);
}
if((stack[st_ptr]=='T'&& ip_sym[ip_ptr]== '*') ||
(stack[st_ptr]=='E'&& ip_sym[ip_ptr]==')')||
(stack[st_ptr]=='E'&& ip_sym[ip_ptr]=='+')||
(stack[st_ptr]=='+'&& ip_sym[ip_ptr]=='i'&&
ip_sym[ip_ptr+1]=='d')||
( stack[st_ptr]== '('&& ip_sym[ip_ptr]=='i'&&
ip_sym[ip_ptr+1]=='d') ||
(stack[st_ptr]== '('&& ip_sym[ip_ptr]=='(')||
(stack[st_ptr]=='*'&& ip_sym[ip_ptr]=='i'&&
ip_sym[ip_ptr+1]=='d' ) ||
(stack[st_ptr]=='*'&& ip_sym[ip_ptr]=='(') ||
(stack[st_ptr]=='+'&& ip_sym[ip_ptr]=='(')
)
{
flag=2;
}
if(!strcmp(stack,"E")&& ip_sym[ip_ptr]=='\0')
{
printf("\n $%s\t\t%s$\t\tACCEPT",stack,ip_sym);
exit(0);
}
if(flag==0)
{
printf("\n%s\t\t\t%s\t\treject",stack,ip_sym);
exit(0);
}
if (flag==2)
return;
flag=0;
}
}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	**id+id+id**	SHIFT REDUCE PARSER GRAMMER E->E+T|T T->T* F | F F-> (E) | id enter the input symbol: id+id+id stack implementation table stack input symbol action $ id+id+id$ -- $id +id+id$ shift id $F +id+id$ F->id $T +id+id$ T->F $E +id+id$ E->T $E+ id+id$ shift + $E+id +id$ shift id $E+F +id$ F->id $E+T +id$ T->F $E +id$ E->E+T $E+ id$ shift + $E+id $ shift id $E+F $ F->id $E+T $ T->F $E $ E->E+T $E $ ACCEPT	
2	E->2E2 E->3E3 E->4	stack input action $ 32423$ SHIFT $3 2423$ SHIFT $32 423$ SHIFT $324 23$ REDUCE TO E -> 4 $32E 23$ SHIFT $32E2 3$ REDUCE TO E -> 2E2 $3E 3$ SHIFT $3E3 $ REDUCE TO E -> 3E3 $E $ Accept	

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

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

Three address code

- Three-address code is an intermediate code. It is used by the optimizing compilers.

- In three-address code, the given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.
- Each Three address code instruction has at most three operands. It is a combination of assignment and a binary operator.

Implementation of Three Address Code –
There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple –
It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example – Consider expression a = b * – c + b * – c.
The three address code is:

```
t1 = uminus c
t2 = b * t1
t3 = uminus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

2. Triples –
This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

3. Indirect Triples –
This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = b * – c + b * – c

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Question – Write quadruple, triples and indirect triples for following expression : (x + y) * (y + z) + (x + y + z)

Explanation – The three address code is:

```
t1 = x + y
t2 = y + z
t3 = t1 * t2
t4 = t1 + z
t5 = t3 + t4
```

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

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

char op[2],arg1[5],arg2[5],result[5];
void main()
{
      FILE *fp1,*fp2;
      int count = 1;
      fp1=fopen("input.txt","r");
      fp2=fopen("output.txt","w");

      while(!feof(fp1))
      {
      fscanf(fp1,"%s%s%s%s",result,arg1,op,arg2);
            if(arg2[0] == '?' &&result[0]== 'T')
      {
            fprintf(fp2,"\nLD R0,%s",arg1);
      }
            if(arg1[0] != 'T' && arg2[0] != 'T'
&&strcmp(op,"+")==0)
            {
                  fprintf(fp2,"\nLD R1,%s",arg1);
            fprintf(fp2,"\nLD R2,%s",arg2);
                  fprintf(fp2,"\nADD R1,R1,R2");
      }
            if(arg1[0] == 'T' && arg2[0] == 'T'
&&strcmp(op,"*")==0)
            {
                  fprintf(fp2,"\nMULT R1,R1,R0");
      }
```

```
                if((strcmp(op,"=")==0) && (arg1[0]=='T') &&
count == 1)
          {
                //fprintf(fp2,"\nMOV R0,%s",arg1);
                //fprintf(fp2,"\nMOV %s,R0",result);
                     fprintf(fp2,"\nST  %s,R0",result);
                     count++;
          }
          }
     fclose(fp1);
     fclose(fp2);
     }
```

Test No	Input Parameters	Expected Output	Obtained Output

1	Enter the set of Intermediate code
	T1 = -B
	T2 = C + D
	T3 = T1 + T2
	A = T3
		Load R1,B
		NEG R1
		STORE T1,R1
		
		Load R2,C
		Load R3,D
		ADD R1,R2,R3
		STORE T2,R1
		
		Load R2,T1
		Load R3,T2
		MUL R1,R2,R3
		STORE T3,R1
		
		Load R4,T3
		STORE A,R4
			Same as expected

2	***T1 = -C*** ***T2 = B *T1*** ***T3 = -C*** ***T4 = B*T3*** ***T5 = T2 + T4*** ***A = T5***		

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

```
%x CMNT
%%
"/*" {BEGIN CMNT;}
<CMNT>. ;
<CMNT>\n ;
```

```
<CMNT>"*/" {BEGIN 0; }
%%
main(int argc, char *argv[])
{
if(argc !=3 )
{
printf(" Usage: %s <src file><dst file>\n",argv[0]);
return;
}
yyin=fopen(argv[1],"r");
yyout=fopen(argv[2],"w");
yylex();
}
OR
%{
int cc=0;
%}
%x CMNTML CMNTSL
%%
"/*" {BEGIN CMNTML;cc++;}
<CMNTML>. ;
<CMNTML>\n ;
<CMNTML>"*/" {BEGIN 0;}
"//" {BEGIN CMNTSL;cc++;}
<CMNTSL>. ;
<CMNTSL>\n {BEGIN 0;}
%%
main(int argc,char *argv[])
{
if(argc!=3)
{
printf("usage:%s<src file><dst file>\n",argv[0]);
return;
}
yyin=fopen(argv[1],"r");
yyout=fopen(argv[2],"w");
yylex();
}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	cat a.txt Hello how are you /* kjdsfksjkaldflks */	./a.out a.txt b.txt cat b.txt Hello how are you cat b.txt Hello how are you	

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

LEX PART

```
%{
#include "y.tab.h"
int idc=0,keyc=0;
%}
%x DECL
%%
"int "|"float "|"char "|"double " {BEGIN DECL;keyc++;return
KWORD;}
<DECL>[a-zA-Z_][a-zA-Z_0-9]* { idc++;return ID;}
<DECL>[,;] {return yytext[0];}
<DECL>\n { BEGIN 0;}
.|\n ;
%%
main()
{
yyin=fopen("abc.c","r");
yyparse();
printf("Total No of keyword = %d",keyc);
printf("Total No of identifiers = %d", idc);

}
yyerror()
{
printf(" Invalid Declaration!!!!\n");
exit(0); }
```

YACC PART

```
%{
#include<stdio.h>
%}
%token KWORD ID
%%
start: KWORD  X ';' {printf("Valid Declaration");return 0;}
     ;
X: X ',' X
 | ID
 ;
%%
```

(OR)

LEX PART

```
%{
#include<stdio.h>
int count=0;
```

```
%}
op [\+\-\*\/\=]
nop [\,\;\""]
letter [a-zA-Z]
digitt [0-9]
id {letter}+|({letter}{digitt})+
notid ({digitt}{letter})+
%%
[ \t\n]+ ;
("int")|("float")|("char")|("case")|("default")|("if")|("for")|
("printf")|("scanf") {printf("%s is a keyword\n", yytext);}
{id} {printf("%s is an identifier\n", yytext); count++;}
"+"|"-"|"*"|"/"|"=" {printf( "An operator: %s\n", yytext);}
{notid} {printf("%s is not an identifier\n", yytext);}
","|";"  {;}
%%
int main()
{
FILE *fp;
char file[10];
printf("\nEnter the filename: ");
scanf("%s", file);
fp=fopen(file,"r");
yyin=fp;
yylex();
printf("Total identifiers are: %d\n", count);
return 0;
}
yywrap()
{
return 1;
}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	If	keyword	
2	2,3	Identifiers	
3	+,*	Operator	

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

// ROUND ROBIN:

```
import java.util.Scanner;
```

```
public class TestClass {

    public static void main(String args[]) {
            Scanner s = new Scanner(System.in);

          int wtime[],btime[],rtime[],num,quantum,total;

wtime = new int[10];
btime = new int[10];
rtime = new int[10];

System.out.print("Enter number of processes(MAX 10): ");
num = s.nextInt();
System.out.print("Enter burst time");
for(int i=0;i<num;i++) { System.out.print("\nP["+(i+1)+"]: ");
btime[i] = s.nextInt(); rtime[i] = btime[i]; wtime[i]=0; }
System.out.print("\n\nEnter quantum: "); quantum = s.nextInt();
int rp = num; int i=0; int time=0; System.out.print("0");
wtime[0]=0; while(rp!=0) { if(rtime[i]>quantum)
 {
rtime[i]=rtime[i]-quantum;
System.out.print(" | P["+(i+1)+"] | ");
   time+=quantum;
System.out.print(time);
   }
 else if(rtime[i]<=quantum &&rtime[i]>0)
 {time+=rtime[i];
rtime[i]=rtime[i]-rtime[i];
System.out.print(" | P["+(i+1)+"] | ");
rp--;
System.out.print(time);
 }

i++;
if(i==num)
{
i=0;
}
}       }}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	P[1]: 4 P[2]: 5 P[3]: 8 P[4]: 2 Enter quantum: 4	0 | P[1] | 4 | P[2] | 8 | P[3] | 12 | P[4] | 14 | P[2] | 15 | P[3] | 19	0 | P[1] | 4 | P[2] | 8 | P[3] | 12 | P[4] | 14 | P[2] | 15 | P[3] | 19

2			

// SJF

```
import java.util.Scanner;

public class TEST {

     public static void main(String args[]){
     int
burst_time[],process[],waiting_time[],tat[],i,j,n,total=0,pos,t
emp;
     float wait_avg,TAT_avg;
     Scanner s = new Scanner(System.in);

     System.out.print("Enter number of process: ");
     n = s.nextInt();

     process = new int[n];
     burst_time = new int[n];
     waiting_time = new int[n];
     tat = new int[n];

     System.out.println("\nEnter Burst time:");
     for(i=0;i<n;i++)
     {
     System.out.print("\nProcess["+(i+1)+"]: ");
     burst_time[i] = s.nextInt();;
     process[i]=i+1; //Process Number
     }

     //Sorting
     for(i=0;i<n;i++)
     {
     pos=i;
     for(j=i+1;j<n;j++)
     {
     if(burst_time[j]<burst_time[pos])
     pos=j;
     }

     temp=burst_time[i];
     burst_time[i]=burst_time[pos];
     burst_time[pos]=temp;

     temp=process[i];
```

```
        process[i]=process[pos];
        process[pos]=temp;
        }

        //First process has 0 waiting time
        waiting_time[0]=0;
        //calculate waiting time
        for(i=1;i<n;i++)
        {
        waiting_time[i]=0;
        for(j=0;j<i;j++)
        waiting_time[i]+=burst_time[j];

        total+=waiting_time[i];
        }

        //Calculating Average waiting time
        wait_avg=(float)total/n;
        total=0;

        System.out.println("\nProcess\t Burst Time \tWaiting
Time\tTurnaround Time");
        for(i=0;i<n;i++)
        {
        tat[i]=burst_time[i]+waiting_time[i]; //Calculating
Turnaround Time
        total+=tat[i];
        System.out.println("\n p"+process[i]+"\t\t
"+burst_time[i]+"\t\t "+waiting_time[i]+"\t\t "+tat[i]);
        }

        //Calculation of Average Turnaround Time
        TAT_avg=(float)total/n;
        System.out.println("\n\nAverage Waiting Time: "+wait_avg);
        System.out.println("\nAverage Turnaround Time: "+TAT_avg);

        }
        }
```

<table>
<tr><th>Test No</th><th>Input Parameters</th><th>Expected Output</th><th>Obtained Output</th></tr>
<tr>
<td>1</td>
<td>Enter number of process: 3
Enter Burst time:
Process[1]: 20
Process[2]: 30
Process[3]: 15</td>
<td>
<table>
<tr><th>Process</th><th>Burst Time</th><th>Waiting Time</th><th>TAT</th></tr>
<tr><td>P3</td><td>15</td><td>0</td><td>15</td></tr>
<tr><td>P!</td><td>20</td><td>15</td><td>35</td></tr>
<tr><td>P2</td><td>30</td><td>35</td><td>65</td></tr>
</table>
Average Waiting Time: 16.666666
Average Turnaround Time: 38.333332
BUILD SUCCESSFUL (total time: 12 seconds)
</td>
<td>Same as expected time</td>
</tr>
</table>

2			

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

```
import java.util.Scanner;
public class Bankers{
    private int need[][],allocate[][],max[][],avail[][],np,nr;

    private void input(){
     Scanner sc=new Scanner(System.in);
System.out.print("Enter no. of processes and resources : ");
     np=sc.nextInt();  //no. of process
     nr=sc.nextInt();  //no. of resources
     need=new int[np][nr];  //initializing arrays
     max=new int[np][nr];
     allocate=new int[np][nr];
     avail=new int[1][nr];

System.out.println("Enter allocation matrix -->");
for(int i=0;i<np;i++)
for(int j=0;j<nr;j++)
         allocate[i][j]=sc.nextInt();  //allocation matrix

System.out.println("Enter max matrix -->");
for(int i=0;i<np;i++)
for(int j=0;j<nr;j++)
         max[i][j]=sc.nextInt();  //max matrix

System.out.println("Enter available matrix -->");
for(int j=0;j<nr;j++)
         avail[0][j]=sc.nextInt();  //available matrix

sc.close();
    }

    private int[][] calc_need(){
for(int i=0;i<np;i++)
for(int j=0;j<nr;j++)  //calculating need matrix
```

```
            need[i][j]=max[i][j]-allocate[i][j];

        return need;
    }

    private booleancheck(int i){
        //checking if all resources for ith process can be
allocated
for(int j=0;j<nr;j++)
        if(avail[0][j]<need[i][j])
            return false;

    return true;
    }

    public void isSafe(){
input();
calc_need();
booleandone[]=new boolean[np];
        int j=0;

        while(j<np){  //until all process allocated
boolean allocated=false;
for(int i=0;i<np;i++)
         if(!done[i] && check(i)){  //trying to allocate
for(int k=0;k<nr;k++)
             avail[0][k]=avail[0][k]-need[i][k]+max[i][k];
System.out.println("Allocated process : "+i);
          allocated=done[i]=true;
j++;
               }
            if(!allocated) break;  //if no allocation
        }
        if(j==np)  //if all processes are allocated
System.out.println("\nSafely allocated");
        else
System.out.println("All process cant be allocated safely");
    }

    public static void main(String[] args)
{
        new Bankers().isSafe();
    }
}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	Enter no. of processes and resources : 3 4 Enter allocation matrix --> 1 2 2 1 1 0 3 3 1 2 1 0 Enter max matrix --> 3 3 2 2 1 1 3 4 1 3 5 0 Enter available matrix --> 3 1 1 2	Allocated process : 0 Allocated process : 1 Allocated process : 2 Safely allocated	Same as expected output
2			

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.

```
import java.io.*;
class LRU
{
public static int sort(int c[])
{
int max=-1;
int temp=-1;
for(int k=0;k<3;k++)
{
if(c[k]>max)
{
max=c[k];
temp=k;
}
}
return temp;
}
public static void main(String args[])throws IOException
{
int z,m=0,hit=0,faults=0;
```

```
InputStreamReaderisr=new InputStreamReader(System.in);
BufferedReaderbr=new BufferedReader(isr);
System.out.println("enter the size of the array");
int n=Integer.parseInt(br.readLine());
int a[]=new int[n];
int flag[]=new int[n];
System.out.println("enter the elements");
for(int i=0;i<n;i++)
{
a[i]=Integer.parseInt(br.readLine());
flag[i]=0;
}
int b[]=new int[3];
int c[]=new int[3];
for(int i=0;i<3;i++)
{
b[i]=-1;
c[i]=0;
}
for(int i=0;i<n;i++)
{
z=a[i];
for(int j=0;j<3;j++)
{
if(z==b[j])
{
flag[i]=1;
hit=hit+1;
break;
}
}
if (flag[i]==0 && b[2]==-1)
{
for(int j=0;j<3;j++)
{
if(b[j]==-1)
{
b[j]=z;
faults=faults+1;
flag[i]=1;
break;
}
}
}
if(flag[i]==0)
{
```

```
m=sort(c);
b[m]=z;
faults=faults+1;
flag[i]=1;
for(int k=0;k<3;k++)
c[k]=c[k]+1;
c[m]=0;
}
}
System.out.println("no of hits"+hit);
System.out.println("no of faults"+faults);
System.out.println("hit ratio"+(hit*100)/(hit+faults));

}
}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	enter the size of the array 10 enter the elements 2 3 5 4 2 5 7 3 8 7	no of hits 2 no of faults 8 hit ratio 20	Same as expected output
2			

//FIFO

```
import java.io.*;
class FIFO
{
public static void main(String args[]) throws IOException
{
int fifo[]=new int[3];
int n;
BufferedReaderbr=new BufferedReader(new
InputStreamReader(System.in));
System.out.println("Enter the number of inputs :");
n=Integer.parseInt(br.readLine());
int inp[]=new int[n];
System.out.println("Enter the inputs:");
for(int i=0;i<n;i++)
inp[i]=Integer.parseInt(br.readLine());
```

```
System.out.println("------");
for(int i=0;i<3;i++)
fifo[i]=-1;
int Hit=0;
int Fault=0;
int j=0;
boolean check;
for(int i=0;i<n;i++)
{
check=false;

for(int k=0;k<3;k++)
if(fifo[k]==inp[i])
{
check=true;
Hit=Hit+1;
}
if(check==false)
{
fifo[j]=inp[i];
j++;
if(j>=3)
j=0;
Fault=Fault+1;
}

}
System.out.println(" FAULT:"+Fault);
}
}
```

Test No	Input Parameters	Expected Output	Obtained Output
1	Enter the number of inputs : 10 Enter the inputs: 2,3,5,4,2,5,7,3,8,7	FAULT: 8	Same as expected output
2			

Appendix-A

This section lists the additional exercise problems related to each program which should be designed, implemented and executed by all students in each laboratory session.

Program 1

Additional Questions:

1. Lex Program to count no of vowels and consonants in a given string
2. Lex Program to check whether the sentence is simple or compound
3. Lex Program for checking a valid URL
4. Lex program to check if a Date is valid or not
5. Lex program to check valid Mobile Number

Program 2

Additional Questions:

1. YACC program which accept strings that starts and ends with 0 or 1
2. YACC program to recognize string with grammar { a^nb^n | n≥0 }
3. YACC program to recognize strings of { a^nb | n≥5 }
4. YACC program for Binary to Decimal Conversion
5. YACC program for Conversion of Infix to Postfix expression

Program 3

Additional Questions:

1. Write a C program to Construct of recursive descent parsing for the following grammar.
E->TE'
E'->+TE/@
T->FT'
T`->*FT'/@
F->(E)/ID where"@ represents null character"

2. Write a C program to Construct of recursive descent parsing for the following grammar.
S -> a S a| T
T->bT | @ where"@ represents null character"

3. Implement a predictive parser for an expression that generates arithmetic expressions with digits, +,*

4. Write a C Program To Find First of a Given Grammar using Array

5. Write a C Program To Find Follow of a Grammar using Array

Program 4

Additional Questions:

1. Design a LALR Bottom Up Parser for the given grammar

   S → AA

   A → aA

   A → b

2. Consider the grammar

E –> 2E2
E –> 3E3
E –> 4

Perform Shift Reduce parsing for input string "32423".

3. Consider the following grammar-

S →(L) | a
L →L , S | S

Parse the input string (a , (a , a)) using a shift-reduce parser.

Program 5

Additional Questions:

1. Aprogramtogeneratemachinecodefromtheabstractsyntaxtreegeneratedbythep arser.
2. Implement a C/Java program to generate the machine code using Triples for the statement a = b * – c + b * – c
3. Generate three address code for the following code-

```
c = 0
do
{
if (a < b) then
x++;
else
x–;
c++;
} while (c < 5)
```

Program 6

Additional Questions:

1. Lex program to copy the content of one file to another file
2. Lex program to count the frequency of the given word in a file
3. Lex program to check whether given number is armstrong number or not
4. Lex program to add line numbers to a given file
5. YACC program to check whether given string is Palindrome or not

Program 7

Additional Questions:

1. C program for FCFS algorithm
2. C program for priority scheduling algorithm
3.

Program 8

Additional Questions:

1. Implement the deadlockfree solution to producer consumer Problem (Using Semaphore)
2. Implement the deadlockfree solution to Dining Philosophers Problem (Using Mutex)

Program 9

Additional Questions:

1. write a C program to implement memory management using segmentation.
2. Write a C/Java program to implement optimal page replacement algorithm.
3. If all page frames are initially empty, and a process is allocated 3 page frames in real memory and references its pages in the order 1 2 3 2 4 5 2 3 2 4 1 and the page replacement is FIFO, LRU and Optimal. Find the total number of page faults caused by the process.

END

Appendix-B

Viva Questions

This section lists few viva questions related to each program for which students have to come prepare and answer them in each laboratory session.

Program 1

1. Explain lex and yacc tools.
2. Give the structure of the lex program:-
3. The lexer produced by lex in a 'c' routine is called _____
4. Explain yytext.
5. Why we have to include 'y.tab.h' in lex?

Program 2

1. What type of data structures is used by shift/reduce parsing?
2. Define grammar?
3. Explain the structure of a YACC program?
4. The YACC produced by parser is called _____
5. What does $$ represents?

General questions for program 3,4,5

1. What Is A Compiler?
2. What Are The Two Parts Of A Compilation? Explain Briefly.
3. List The Sub Parts Or Phases Of Analysis Part ?
4. List The Various Phases Of A Compiler ?
5. List The Phases That Constitute The Front End Of A Compiler.
6. Mention The Back-end Phases Of A Compiler.
7. List The Various Compiler Construction Tools.
8. Mention The Various Notational Short Hands For Representing Regular Expressions.
9. Define Ambiguous Grammar.

Program 3

1. Which derivation does a top-down parser use while parsing an input string?
2. Given the following expression grammar:
   E -> E * F | F+E | F
   F -> F-F | id
3. Which operator is having higher precedence?
4. Difference between

Recursive Predictive Descent Parser	Non-Recursive Predictive Descent Parser
It is a technique which may or may not	It is a technique that does not require any kind

require backtracking process.	of back tracking.
It uses procedures for every non terminal entity to parse strings.	It finds out productions to use by replacing input string.
It is a type of top-down parsing built from a set of mutually recursive procedures where each procedure implements one of non-terminal s of grammar.	It is a type of top-down approach, which is also a type of recursive parsing that does not uses technique of backtracking.
It contains several small small functions one for each non- terminals in grammar.	The predictive parser uses a look ahead pointer which points to next input symbols to make it parser back tracking free, predictive parser puts some constraints on grammar.
It accepts all kinds of grammars.	It accepts only a class of grammar known as LL(k) grammar.

Program 4

1. A bottom up parser generates _____
2. A grammar that produces more than one parse tree for some sentence is called _____
3. Shift reduce parsers are _____

Program 5

1. What are the benefits of intermediate code generation?
2. What are the various types of intermediate code generation?
3. What is the intermediate code representation for the expression a or b and not c?
4. What are the various methods of implementing three address statements?

Program 6

1. Explain yyleng?
2. What are tokens or terminal symbols?
3. what is symbol table?
4. What is pseudo token
5. what is lexical analyzer?

Program 7

1. What are the different scheduling algorithms
2. What is FCFS?

3. Three process P1, P2 and P3 arrive at time zero. The total time spent by the process in the system is 10ms, 20ms, and 30ms respectively. They spent first 20% of their execution time in doing I/O and the rest 80% in CPU processing. What is the percentage utilization of CPU using FCFS scheduling algorithm?
4. Three process P1, P2 and P3 arrive at time zero. Their total execution time is 10ms, 15ms, and 20ms respectively. They spent first 20% of their execution time in doing I/O, next 60% in CPU processing and the last 20% again doing I/O. For what percentage of time was the CPU free? Use Round robin algorithm with time quantum 5ms.

Program 8

1. What is deadlock?
2. What are the necessary conditions for deadlock?
3. Why Banker's algorithm is named so?
4. List various dead lock avoidance, dead lock detection mechanism.

Program 9

1. If all page frames are initially empty, and a process is allocated 3 page frames in real memory and references its pages in the order 1 2 3 2 4 5 2 3 2 4 1 and the page replacement is FIFO, the total number of page faults caused by the process will be _____.
2. When does page fault occurs
3. Reference bit is used for implementing _____ algorithm
4. The optimal page replacement algorithm will select the page that_____
5. What is page cannibalizing?