CSL411 COMPILER LAB

Experiment list

1. Design and implement a lexical analyzer using C language to recognize all valid tokens in the input program. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments.

2..Write a lex program to find out total number of vowels and consonants from the given input sting.

3. Write a lex program to display the number of lines, words and characters in an input text.

4. Write a LEX Program to convert the substring abc to ABC from the given input string.

5. Implement a Lexical Analyzer for a given program using Lex Tool

6. Generate a YACC specification to recognize a valid arithmetic expression that uses operators +, – , *,/ and parenthesis.

7. Generate a YACC specification to recognize a valid identifier which starts with a letter followed by any number of letters or digits.

8. Implementation of Calculator using LEX and YACC

9. Write a program to find ε – closure of all states of any given NFA with ε transition.

10. Write a program to find First and Follow of any given grammar.

11. Design and implement a recursive descent parser for a given grammar.

12. Construct a Shift Reduce Parser for a given language.

13. Write a program to perform constant propagation.

14. Implement Intermediate code generation for simple expressions.

15. Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc

Exercise 1.

Aim: Design and implement a lexical analyzer using C language to recognize all valid tokens in the input program. The lexical analyzer should ignore redundant spaces, tabs and newlines. It should also ignore comments.

Algorithm

1Read the given input) program code) from the file.

2 Check whether the string is identifier/ keyword /symbol by using the rules of identifier and keyword

3. print appropriate message

Program

```
#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<ctype.h>

int isKeyword(char buffer[]){

char keywords[32][10] =

{"auto","break","case","char","const","continue","default",

"do","double","else","enum","extern","float","for","goto",

"if","int","long","register","return","short","signed",

"sizeof","static","struct","switch","typedef","union",

"unsigned","void","volatile","while"};

int i, flag = 0;

for(i = 0; i < 32; ++i){

if(strcmp(keywords[i], buffer) == 0){

flag = 1;

break;

}

}

return flag;

}

int main(){
```

```
char ch, buffer[15], operators[] = "+-
*/%=",specialch[]=",;[]{}",num[]="1234567890",buf[10];
FILE *fp;
int i,j=0,k=0;
fp = fopen("program.txt","r");
if(fp == NULL){
printf("error while opening the file\n");
exit(0);
}
while((ch = fgetc(fp)) != EOF)
{
 for(i = 0; i < 6; ++i)
 {
 if(ch == operators[i])
 {
 printf("%c is operator\n", ch);
 }
 if(ch == specialch[i])
 {
 printf("%c is special character\n", ch);
 }


 }
 if(isalpha(ch)){
 buffer[j++] = ch;}
if(isdigit(ch)){
 buf[k++]=ch;
 }
 else if((ch == ' ' || ch == '\n') && (j != 0)){
buffer[j] = '\0';
 j = 0;

 if(isKeyword(buffer) == 1)
 printf("%s is keyword\n", buffer);
 else{
 printf("%s is identifier\n", buffer);
 printf("%s is constant\n", buf);
 }}

}
fclose(fp);
return 0;
}
```

output

program.txt

```
int a,b,v=1
```

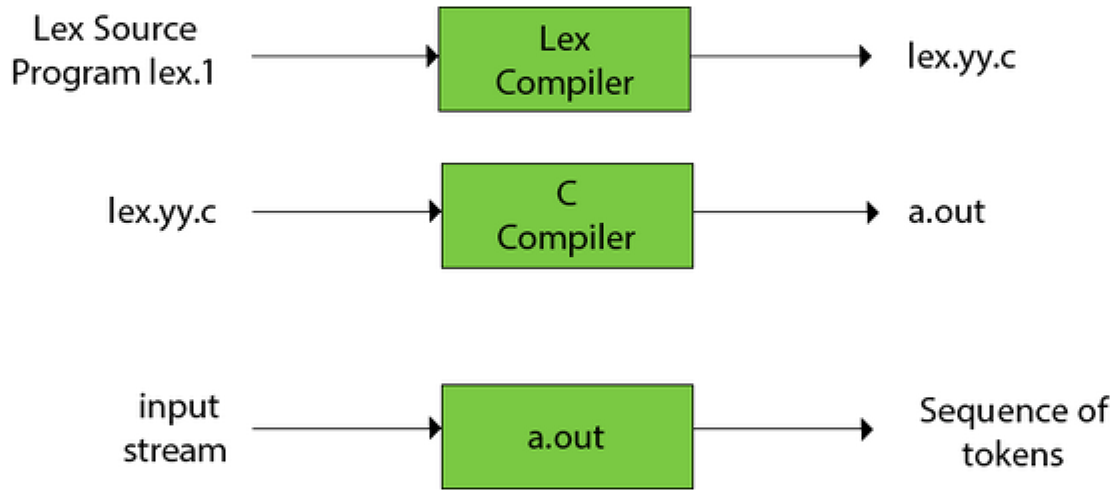
```
 gcc lexical2.c -o test
 ./test
int is keyword
, is special character
, is special character
= is operator
abv is identifier
1 is constant
```

Introduction to LEX

Lex is a program that generates lexical analyzers. It is used with the YACC parser generator.The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

The function of Lex is as follows:

- Firstly, a lexical analyzer creates a program programname.l in the Lex language. Then the Lex compiler runs the programname.l program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is a lexical analyzer that transforms an input stream into a sequence of tokens.



Lex file format

A Lex program is separated into three sections by %% delimiters. The formal of Lex source is as follows:Syntax Of Lex Program:

%{

definition section

%}

```
%%
rules section
/*pattern action */
%%

/*user subroutine section*/
main()
{
yylex();
}
```

<u>Commands To create executable program:</u>

To create C file:

```
lex <pgmname.l>
```

To compile & to create executable file:

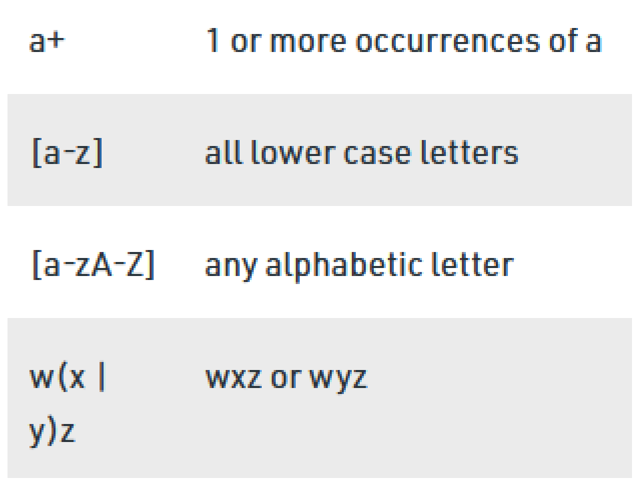
```
cc lex.yy.c –ll
```

To execute program:

```
./a.out
```

Descriptions of expected inputs

Pattern	It can match with
[0-9]	all the digits between 0 and 9
[0+9]	either 0, + or 9
[0, 9]	either 0, ‘,’ or 9
[0 9]	either 0, ‘ ’ or 9
[-09]	either -, 0 or 9
[-0-9]	either − or all digit between 0 and 9
[0-9]+	one or more digit between 0 and 9
[^a]	all the other characters except a
[^A-Z]	all the other characters except the upper case letters
a{2, 4}	either aa, aaa or aaaa
a{2, }	two or more occurrences of a
a{4}	exactly 4 a’s i.e, aaaa
.	any character except newline
a*	0 or more occurrences of a



a+	1 or more occurrences of a
[a-z]	all lower case letters
[a-zA-Z]	any alphabetic letter
w(x | y)z	wxz or wyz

ACTIONS

- yytext - contains the actual characters recognised from input
- yyleng - the number of characters in yytext
- yywrap - is called **whenever the scanner reaches the end of file EOF**. If yywrap() returns 1, the scanner continues with normal wrapup on the end of input.
- The yylex() - this is the main flex function which runs the Rule Section. It reads characters from a FILE * file pointer called yyin. If you do not set yyin, it defaults to standard input. It outputs to yyout, which if unset defaults to stdout.

SAMPLE PROGRAMS

1. HELLO WORLD

```
%{

%}

%%

%%
void main()
{
   yylex();
printf("Hello world\n");
}
```

/*lex code to count alphabets and non alphabets*/

```
%{
```

```
int  alpha=0,nonalpha=0; /*Global variables*/
%}

/*Rule Section*/
%%
[a-zA-Z] alpha++;
. nonalpha++;

%%

int main()
{
   // The function that starts the analysis
   yylex();


   printf("\nNo. of  alpahbets=%d", alpha);
   printf("\nNo. of non-alphabets=%d", nonalpha);

}

/*lex code to count the number of lines,
   tabs and spaces used in the input*/

%{
#include<stdio.h>
int lc=0, sc=0, tc=0, ch=0, nwords=0; /*Global variables*/
%}

/*Rule Section*/
%%
\n lc++; //line counter
([ ])+ sc++; //space counter
\t tc++; //tab counter
[^ \n\t]+ {nwords++, c    h=ch+yyleng;}

%%

int main()
{
   // The function that starts the analysis
   yyin=fopen("input.txt","r");
   yylex();
   fclose(yyin);
   printf("\nNo. of lines=%d", lc);
```

```
    printf("\nNo. of spaces=%d", sc);
    printf("\nNo. of tabs=%d", tc);
    printf("\nNo. of other characters=%d", ch);
    printf("\nNo. of other words=%d", nwords);

}
```

Exercise 2

Aim. Write a lex program to find out total number of vowels and consonants from the given input sting.

Algorithm

Algorithm

Procedure:

1. In definition section, declare and initiate the variables, which are used to count the total number of vowels and consonants.

2. In rule section,

(i)Define the pattern which is used to recognize vowels. (In action part) , If a character from the input string matches with this pattern then increment the vowel count.

(ii)Define the pattern which is used to recognize consonants. (In action part)If a character from the input string matches with this pattern then increment the consonant count.

3.In user subroutine part,

(i) In main() ,Call yylex(). Then print the total number of vowels and consonants in the input string.

Program

```
%{

 int vowels=0,cons=0;    /* Initialize counters */

%}

%%

[aeiouAEIOU] { vowels++; }    /* Count vowels */

[a-zA-Z]     { cons++; }    /* Count consonants */
```

```
%%

main()

{

yylex();

printf("\nThe number of vowels       = %d ",vowels);

printf("\nThe number of consonants = %d\n",cons);

}
```

Sample Input & output:

```
Enter the input string: Bangalore

The number of vowels      =  4

The number of consonants=  5
```

Exercise 3

Aim . Write a lex program to display the number of lines, words and characters in an input text.Algorithm

Procedure:

1. In definition section, declare and initiate the variables, which are used to count the total number of characters, words, and lines in a given input file.

2. In rule section,

(i)Define the pattern which is used to recognize words by specifying any of the word delimiters (one or more occurrences) except tab and new line . (In action part) , If a character from the input string matches with this pattern then increment the word count by 1 and the character count. by yyleng.(yylength is the length of yytext array.)

(ii)Define the pattern which is used to recognize space. (In action part)If a character from the input string matches with this pattern then total character count by 1.

}

(iii)Define the pattern which is used to recognize new line. (In action part) if a character from the input string matches with this pattern then increment the total line count as well as the total character count by 1.

3.In user subroutine part,

(i) In main(),open the input text file in read mode.

(ii) Call yylex(). Then print the total nqumber of words, characters,   and lines in the given input file.

Program

```
%{
 int chars=0,words=0,lines=0;    /* Initialize counters */
%}
%%
[^ \t\n]+     { words++; chars = chars + yyleng; }
[ ]        {  chars++; }
[\n]        { lines++; chars++; }
%%


main()
{
 yyin=fopen("input.txt","r");
 yylex(); AA ll
Pujjprintf("\nCharacters = %d",chars);
  printf("\nLines = %d",lines);
}
```

Sample Input & output:

Input.txt

This is bangalore

Words =3

Characters=17

Lines=1

Exercise 4.

Aim . Implement a Lexical Analyzer for a given program using Lex Tool.

Algorithm

Algorithm for Lexical analyzer using Lex Tool

1 Lex program contains three sections : definitions , rules , and user subroutines . Each section must be separated from the others by a line containing only the delimiter , %%. The format is as follows : definitions %% rules %% user_subroutines

2 In definition section , the variables make up the left column , and their definitions make up the right column . Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric .

3 In rules section , the left column contains the pattern to be recognized in an input file to yylex () . The right column contains the C program fragment executed when that pattern is recognized . The various patterns are keywords , operators , new line character , number , string , identifier , beginning and end of block , comment statements , preprocessor directive statements etc .

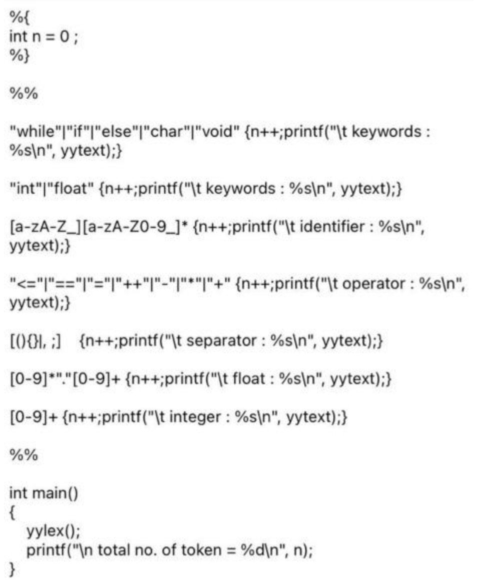
4 Each pattern may have a corresponding action , that is , a fragment of C source code to execute when the pattern is matched .

5 When yylex () matches a string in the input stream , it copies the matched text to an external character array , yytext , before it executes any actions in the rules section .

6 In user subroutine section , main routine calls yylex () . yywrap () is used to get more input .

7 The lex command uses the rules and actions contained in file to generate a program , lex . yy .c , which can be compiled with the gcc command . That program can then receive input , break the input into the logical pieces defined by the rules in file , and run program fragments contained in the actions in file .

Program



```
%{
int n = 0 ;
%}

%%

"while"|"if"|"else"|"char"|"void" {n++;printf("\t keywords :
%s\n", yytext);}

"int"|"float" {n++;printf("\t keywords : %s\n", yytext);}

[a-zA-Z_][a-zA-Z0-9_]* {n++;printf("\t identifier : %s\n",
yytext);}

"<="|"=="|"="|"++"|"-"|"*"|"+" {n++;printf("\t operator : %s\n",
yytext);}

[(){}|, ;]    {n++;printf("\t separator : %s\n", yytext);}

[0-9]*"."[0-9]+ {n++;printf("\t float : %s\n", yytext);}

[0-9]+ {n++;printf("\t integer : %s\n", yytext);}

%%

int main()
{
   yylex();
   printf("\n total no. of token = %d\n", n);
}
```

Output

```
if(a==b)

keywords:if

separator:(

identifier: a

operator:==

identifier: b

separator:)

total no. of token = 6
```

// **Another code**

```
%{

int COMMENT =0;

%}

identifier [a - zA - Z ][ a - zA - Z0 -9]*
```

```
%%

#.* { printf ("\n%s is a preprocessor directive ", yytext ) ;}

int |

float |

char |

double |

while |

for |

struct |

typedef |

do |

if |

break |

continue |

void |

switch |

return |

else |

goto { printf ("\n\t%s is a keyword ", yytext ) ;}

"/*" { COMMENT =1;}{ printf ("\n\t %s is a COMMENT ", yytext ) ;}

{ identifier }\( {if (! COMMENT ) printf ("\ nFUNCTION \n\t%s", yytext ) ;}

\{ {if (! COMMENT ) printf ("\n BLOCK BEGINS ") ;}

\} {if (! COMMENT ) printf (" BLOCK ENDS ") ;}

{ identifier }(\[[0 -9]*\]) ? {if (! COMMENT ) printf ("\n %s IDENTIFIER ",

yytext ) ;}

\" .*\" {if (! COMMENT ) printf ("\ n \ t % s is a STRING ",yytext );}

[0 -9]+ {if (! COMMENT ) printf ("\ n % s is a NUMBER ",yytext );}
```

```
\) (\:) ? {if (! COMMENT ) printf ("\ n \ t"); ECHO ; printf ("\ n");}

\( ECHO ;

= {if (! COMMENT ) printf ("\ n \ t % s is an ASSIGNMENT OPERATOR ",yytext );}

\ <= |

\ >= |

\ < |

== |

\ > {if (! COMMENT ) printf ("\ n \ t % s is a RELATIONAL OPERATOR ",yytext );}

%%

int main (int argc , char ** argv )

{

FILE * file ;

file = fopen ("var . c","r");

if (! file )

{

printf (" could not open the file ");

exit (0) ;

}

yyin = file ;

yylex ();

printf ("\ n");

return (0) ;

}

int yywrap ()

{

return (1) ;

}
```

Input File:

```
var.c

#include<stdio.h>

#include<conio.h>

void main()

{

int a,b,c;

a=1;

b=2;

c=a+b;

printf("Sum:%d",c);

}
```

Output:

Exercise 5.

Aim Write a LEX Program to convert the substring abc to ABC from the given input string

Algorithm

1Lex program contains three sections: definitions, rules, and user subroutines. Each section must be separated from the others by a line containing only the delimiter, %%.

The format is as follows: definitions %% rules %% user_subroutines

2In definition section, the variables make up the left column, and their definitions make up the right column. Any C statements should be enclosed in %{..}%. Identifier is defined such that the first letter of an identifier is alphabet and remaining letters are alphanumeric.

3In rules section, the left column contains the pattern to be recognized in an input file to yylex(). The right column contains the C program fragment executed when that pattern is recognized. The various patterns are keywords, operators, new line character, number, string, identifier, beginning and end of block, comment statements, preprocessor directive statements etc.

4Each pattern may have a corresponding action, that is, a fragment of C source code to execute when the pattern is matched.

5When yylex() matches a string in the input stream, it copies the matched text to an external character array, yytext, before it executes any actions in the rules section.

6In user subroutine section, main routine calls yylex(). yywrap() is used to get more input.

7The lex command uses the rules and actions contained in file to generate a program, lex.yy.c, which can be compiled with the cc command. That program can then receive input, break the input into the logical pieces defined by the rules in file, and run program fragments contained in the actions in file.

Program

```
%{

int i;

%}


%%

[a-z A-Z]* {

for(i=0;i<=yyleng;i++)

{

if((yytext[i]=='a')&&(yytext[i+1]=='b')&&(yytext[i+2]=='c'))

{

yytext[i]='A';

yytext[i+1]='B';

yytext[i+2]='my

}

}

printf("%s",yytext);

}


[\t]* return;
```

```
.* {ECHO;}

\n {printf("%s",yytext);}


%%


main()

{

yylex();

}


int yywrap()

{

return 1;

}
```

Output

```
rabcdrgabchh

rABCdrgABChh
```

Introduction to YACC

A parser generator is a program that takes as input a specification of a syntax, and produces as output a procedure for recognizing that language They are called compiler compilers.

YACC (yet another compiler-compiler) is an LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator.

Yacc is a compiler compiler.It is the standard parser generator for the Unix operating system. An open source program, yacc generates code for the parser in the C programming language

To install Yacc package:

Type the command

```
sudo apt-get install bison
```

Type 'y' when it asks for confirmation

Input File:

YACC input file is divided into three parts.

```
/* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

The auxiliary routines part is only C code. It can also contain the main() function definition if the parser is going to be run as a program.

The main() function must call the function yyparse().

If yylex() is not defined in the auxiliary routines sections, then it should be included:

```
#include "lex.yy.c"
```

Syntax Of Yacc Program:

```
%{    /* definition section  */    %}

% token  symbolic-token-list (i.e) terminals used in grammar

%%
   /* rules section : Grammar rules */
%%

/*user subroutine section*/
main()
{
  yyparse();
}
```

Syntax Of Lex Program:

```
%{
   # include "y.tab.h"
definition section
%}

%%
   rules section

   /*pattern       action     */
```

```
%%
```

Commands To create executable program:

To generate both y.tab.c & y.tab.h (token definition –d):

```
yacc –d  <pgmname.y>
```

To generate C file from the Lex program:

```
lex   <pgmname.l>
```

To compile and to link C files

```
cc  y.tab.c  lex.yy.c  –ll
```

To Execute the program:

```
./a.out
```

First, we need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y).

Commands to create our compiler, bas.exe, are listed below:

```
yacc –d bas.y
```

This will create y.tab.h, y.tab.c

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.

```
lex bas.l
```

This will create lex.yy.c

cc lex.yy.c y.tab.c –o bas.exe # compile/link

Yacc reads the grammar descriptions in bas.y and generates a syntax analyzer (parser), that

includes function yyparse, in file y.tab.c. Included in file bas.y are token declarations. The –d

option causes yacc to generate definitions for tokens and place them in file y.tab.h. Lex reads the pattern descriptions in bas.l, includes file y.tab.h, and generates a lexical analyzer, that includes function yylex, in file lex.yy.c.

Finally, the lexer and parser are compiled and linked together to create executable bas.exe.

From main we call yyparse to run the compiler. Function yyparse automatically calls yylex to

obtain each token.

INPUT SPECIFICATION FOR YACC

%left, for left-associative or %right for right associative.

The last definition listed has the highest precedence.

E.g.

```
%token INTEGER VARIABLE

%left '+' '-'

%left '*' '/'
```

The tokens for INTEGER and VARIABLE are utilized by yacc to create #defines in y.tab.h for use in lex.

Exercise 6.

Aim Generate a YACC specification to recognize a valid arithmetic expression that uses operators +, – , *,/ and parenthesis.

Algorithm :

Algorithm to recognize valid arithmetic expression using YACC

1 START

2 Read the given input expression .

3 Check and correspondingly validate the given expression according

to the rulee using yacc .

4 Using the corresponding expression rules print if the input

expression is valid or not .

5 STOP

Program

Program to recognize a valid arithmetic expression that uses operators +, -, * and /.

LEX

```
%{
#include"y.tab.h"
extern yylval;
%}
%%
[0-9]+ {yylval=atoi(yytext); return NUMBER;}
[a-zA-Z]+ {return ID;}
[\t]+ ;
\n {return 0;}
. {return yytext[0];}
%%
```

YACC

```
%{
#include<stdio.h>
%}
%token NUMBER ID
```

```
}

%left '+' '-'

%left '*' '/'

%%

expr: expr '+' expr

     |expr '-' expr

     |expr '*' expr

     |expr '/' expr

     |'-'NUMBER

     |'-'ID

     |'('expr')'

     |NUMBER

     |ID

     ;

%%

main()

{

printf("Enter the expression\n");

yyparse();

printf("\nExpression is valid\n");

exit(0);

}

int yyerror(char *s)

{

printf("\nExpression is invalid");

exit(0);

}
```

Output

```
Enter the expression

a+b

Expression is valid
```

Exercise 7.

Aim Generate a YACC specification to recognize a valid identifier which starts with a letter followed by any number of letters or digits.

Algorithm

1START

2 Read the given input expression .

3 According to the definition of rules , validate the given expression

to see if it is a valid variable .

4 Using expression rules , print if the corresponding expression is a

valid variable or not .

5 STOP

Program

Aim Generate a YACC specification to recognize a valid identifier which starts with a letter

Procedure:

Yacc

1. Define the symbolic token NUMBER,LETTER.

2. In Rules section, Write the grammar rules to recognize the valid variable which starts with a letter, followed by any number of letters or digits.

Grammar G:

id id LETTER | LETTER

id id NUMBER | NUMBER

3.In user subroutine section, call yyparse() to check the syntax of the input viable name. If it is valid then print the expression as valid, else if it is invalid the yyerror can be called automatically by the parser. User can also redefine the yyerrror();

Lex

1.Lexer will provide the tokens for our parser. In definition section include the “y.tab.h” which are the generated token codes from the parser.

2.In rules section write the patterns for the following:

(a) If the input variable is started with Digit [0-9] (i.e) except alphabets a-z, then it will be return the token code for digit (which is invalid variable name).

(b) If the variable name is started with alphabet [a-z] return the token code as LETTER.

Yacc Program

```
%{

%}

%token LETTER NUMBER
```

```
%%
         validid : iden { printf("\n valid!");}
iden  : LETTER
         |iden NUMBER
         |iden LETTER
                    ;

%%

int main()
{
 printf("\nEnter a string : ");
 yyparse();
 printf("\nThe entered string is a valid identifier.\n");
 return 0;
}

int yyerror(char *s)
{
 printf("\n%s\n",s);
 exit(1);
}
```

Lex Program

```
%{
# include "y.tab.h"
%}
%%
 [a-zA-Z]  { return LETTER; }
 [0-9]+  {return NUMBER;}
%%
```

Sample I/O:

```
Enter a string : name
The entered string is a valid identifier.
```

Exercise 8.

Aim Implementation of Calculator using LEX and YACC

Algorithm

```
 1START
2 Input the set of all statements .
3 Identify assignment statements and map their values into those
variables .
4 After assigning , for every 'print ' call , detect the expression to
be printed .
5 If the argument is a variable , print its value and if it is an
expression , evaluate it and print its value
6 STOP
```

Program

Yacc program

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

 int flag=0;



%}

%token NUMBER



%left '+' '-'

%left '*' '/' '%'

%left '(' ')'

%%

ArithmeticExpression: E{

        printf("\nResult=%d\n",$$);

        return 0;

       };
```

```
E:E'+'E {$$=$1+$3;}

 |E'-'E {$$=$1-$3;}

 |E'*'E {$$=$1*$3;}

 |E'/'E {$$=$1/$3;}

 |E'%'E {$$=$1%$3;}

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

 | NUMBER {$$=$1;}

;

%%



void main()

{

   printf("\nEnter Any Arithmetic Expression:\n");
   yyparse();

  if(flag==0)

   printf("\n\n");




}

void yyerror()

{

   printf("\nInvalid\n\n");

   flag=1;

}
```

Lex program

```
%{
/* Definition section */
#include<stdio.h>
#include "y.tab.h"
extern int yylval;
%}
```

```
/* Rule Section */

%%

[0-9]+ {

                        yylval=atoi(yytext);

                        return NUMBER;


            }

[\t] ;


[\n] return 0;


. return yytext[0];


%%


int yywrap()

{

return 1;

}
```

Output

```
yacc –d  calc.y
```

To generate C file from the Lex program:

```
lex   calc.l
```

To compile and to link C files

```
gcc  y.tab.c  lex.yy.c  –ll
```

To Execute the program:

```
./a.out

Enter Any Arithmetic Expression:

2+3

Result =5
```

Exercise 9

Aim Write a program to find ε – closure of all states of any given NFA with ε transition.

Algorithm

Algorithm 1: Algorithm for - closure of NFA

```
1 START
2 Input the epsilon NFA with all states .
3 For each state repeat the following steps of function .
4 function EPSILONCLOSURE ( enfa , k )
5 Initialize a list t containing only state k
6 Initialize an iterator to the first element of the list t
7 While iterator has not crossed the last element of the list t
8 Append all states in the i pair in the transition table of enfa
which is not previously present in list t to t
9 Set the iterator to the next element of the list t
10 Return list t as the epsilon closure for state k in epsilon NFA ,
enfa
11 end function
12 STOP
```

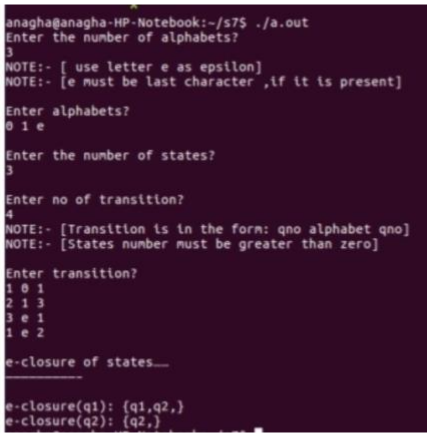
Program

Epsilon - closure

```
# include < stdio .h >

# include < stdlib .h >

struct node

{

int st ;

struct node * link ;

};

void findclosure (int ,int ) ;

void insert_trantbl (int ,char , int ) ;

int findalpha ( char ) ;

void print_e_closure (int ) ;

static int set [20] , nostate , noalpha ,s , notransition ,r , buffer [20];

char alphabet [20] , c ;

static int e_closure [20][20]={0};

struct node * transition [20][20]={ NULL };

void main ()

{

int i ,j ,k ,m ,t , n ;

struct node * temp ;

printf (" Enter the number of alphabets ?\n") ;

scanf ("%d" ,& noalpha ) ;

getchar () ;

printf (" NOTE : - [ use letter e as epsilon ]\n") ;

printf (" NOTE : - [e must be last character ,if it is present

]\n") ;

printf ("\ nEnter alphabets ?\n") ;
```

```
for ( i =0; i < noalpha ; i ++)

{

alphabet [ i ]= getchar () ;

getchar () ;

}

printf ("\ nEnter the number of states ?\n") ;

scanf ("%d" ,& nostate ) ;

printf ("\ nEnter no of transition ?\n") ;

scanf ("%d" ,& notransition ) ;

printf (" NOTE : - [ Transition is in the form : qno alphabet qno

]\n", notransition ) ;

printf (" NOTE : - [ States number must be greater than zero ]\n"

) ;

printf ("\ nEnter transition ?\n") ;

for ( i =0; i < notransition ; i ++)

{

scanf ("%d %c%d" ,&r ,& c ,& s ) ;

insert_trantbl (r ,c , s ) ;

}

printf ("\n")

printf ("e- closure of staes[U+FFFD][U+FFFD]\n") ;

for ( i =1; i <= nostate ; i ++)

{

c =0;

for( j =0; j <20; j ++)

{

buffer [ j ]=0;
e_closure [ i ][ j ]=0;

}

findclosure (i , i ) ;

printf ("\ne - closure (q%d): ",i ) ;

print_e_closure ( i ) ;

}

}

void findclosure (int x ,int sta )

{

struct node * temp ;

int i ;

if( buffer [ x ])

return ;

e_closure [ sta ][ c ++]= x ;

buffer [ x ]=1;

if( alphabet [ noalpha -1]== ’e’ && transition [ x ][ noalpha -1]!=

NULL )

{

temp = transition [ x ][ noalpha -1];

while ( temp != NULL )

{

findclosure ( temp - > st , sta ) ;

temp = temp - > link ;

}

}

}

void insert_trantbl (int r , char c ,int s )
```

```
{

int j ;

struct node * temp ;

j = findalpha ( c ) ;

if( j ==999)

{

printf (" error \n") ;

exit (0) ;

}

temp =( struct node *) malloc ( sizeof ( struct node ) ) ;

temp - > st = s ;

temp - > link = transition [ r ][ j ];

transition [ r ][ j ]= temp ;

}

int findalpha ( char c )

{

int i ;

for ( i =0; i < noalpha ; i ++)

if( alphabet [ i ]== c )

return i ;

return (999) ;

}

void print_e_closure (int i )

{

int j ;

printf ("{") ;

for ( j =0; e_closure [ i ][ j ]!=0; j ++)
printf ("q%d,", e_closure [ i ][ j ]) ;

printf ("}") ;

}
```



```
anagha@anagha-HP-Notebook:~/s7$ ./a.out
Enter the number of alphabets?
3
NOTE:- [ use letter e as epsilon]
NOTE:- [e must be last character ,if it is present]

Enter alphabets?
0 1 e

Enter the number of states?
3

Enter no of transition?
4
NOTE:- [Transition is in the form: qno alphabet qno]
NOTE:- [States number must be greater than zero]

Enter transition?
1 0 1
2 1 3
3 e 1
1 e 2

e-closure of states.....
------------

e-closure(q1): {q1,q2,}
e-closure(q2): {q2,}
```

Exercise 10

.**Aim** Write a program to find First and Follow of any given grammar.

Algorithm

1 Start

2 FIRST (X) for all grammar symbols X

3 1. If X is terminal , FIRST (X) = { X }.

4 2. If X -> e is a production , then add e to FIRST (X)

5 3. If X is a non - terminal , and X -> Y1 Y2 ... Yk is a production , and e is in all of FIRST (Y1) , ... , FIRST (Yk) , then add e to FIRST (X) .

6 4. If X is a non - terminal , and X -> Y1 Y2 ... Yk is a production , then add a to FIRST (X) if for some i , a is in FIRST (Yi) , and e is in all of FIRST (Y1) , ... , FIRST (Yi -1) .

7 FOLLOW (A) for all non - terminals A

8 1. If $ is the input end - marker , and S is the start symbol , $ element of FOLLOW (S) .

9 2. If there is a production , A -> aBb , then (FIRST (b) - e) subset of FOLLOW (B) .

10 3. If there is a production , A -> aB , or a production A -> aBb , where e element of FIRST (b) , then FOLLOW (A) subset of FOLLOW (B) .

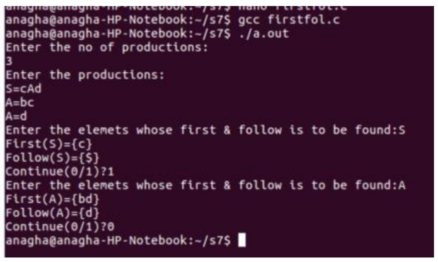
11 Stop

Program

```
#include<stdio.h>

#include<math.h>

#include<string.h>

#include<ctype.h>

#include<stdlib.h>

int n,m=0,p,i=0,j=0;

char a[10][10],f[10];

void follow(char c);

void first(char c);

int main(){



int i,z;

char c,ch;

//clrscr();

printf("Enter the no of prooductions:\n");

scanf("%d",&n);

printf("Enter the productions:\n");

for(i=0;i<n;i++)

scanf("%s%c",a[i],&ch);

do{

m=0;

printf("Enter the elemets whose fisrt & follow is to be found:");

scanf("%c",&c);

first(c);

printf("First(%c)={",c);

for(i=0;i<m;i++)
```

```
printf("%c",f[i]);

printf("}\n");

strcpy(f," ");

//flushall();

m=0;

follow(c);

printf("Follow(%c)={",c);

for(i=0;i<m;i++)

printf("%c",f[i]);

printf("}\n");

printf("Continue(0/1)?");

scanf("%d%c",&z,&ch);

}while(z==1);

return(0);

}

void first(char c)

{

int k;

if(!isupper(c))

f[m++]=c;

for(k=0;k<n;k++)

{

if(a[k][0]==c)

{

if(a[k][2]=='$')

follow(a[k][0]);

else if(islower(a[k][2]))
```

```
f[m++]=a[k][2];

else first(a[k][2]);

}

}

}

void follow(char c)

{

if(a[0][0]==c)

f[m++]='$';

for(i=0;i<n;i++)

{

for(j=2;j<strlen(a[i]);j++)

{

if(a[i][j]==c)

{

if(a[i][j+1]!='\0')

first(a[i][j+1]);

if(a[i][j+1]=='\0' && c!=a[i][0])

follow(a[i][0]);

}

}

}

}
```



```
anagha@anagha-HP-Notebook:~/s7$ nano firstfol.c
anagha@anagha-HP-Notebook:~/s7$ gcc firstfol.c
anagha@anagha-HP-Notebook:~/s7$ ./a.out
Enter the no of productions:
3
Enter the productions:
S=cAd
A=bc
A=d
Enter the elemets whose first & follow is to be found:S
First(S)={c}
Follow(S)={$}
Continue(0/1)?1
Enter the elemets whose first & follow is to be found:A
First(A)={bd}
Follow(A)={d}
Continue(0/1)?0
anagha@anagha-HP-Notebook:~/s7$ 
```

Exercise 11

Aim Design and implement a recursive descent parser for a given grammar.

Algorithm

Algorithm 13: Algorithm for Recursive Descent Parser

1 START

2 Input the set of productions , symbols and expressions .

3 Read each symbol of the expression .

4 Parse method () is called for each non terminal symbol in the

productions .

5 A non terminal in the right hand side of rewrite rule leads to a

call to parse method for that non - terminal .

6 A terminal symbol on the right hand side of a rewrite rule leads to

consuming that token from input token string .

7 l in the CFG leads to "If else " in the parser .

8 If symbol is not expanded correctly are to input expression ,

backtrack .

9 STOP

10 Procedure parser

11 from j =1 to t , repeat

12 choose a production A -> x1 , x2 , ... , xi ;

13 from i -1 to k repeat

14 if(xi is a non - terminal)

15 call procedure xi () ;

16 else if(xi equals current input a)

17 advance input to next symbol

18 else backtrack input and reset pointer .

Program

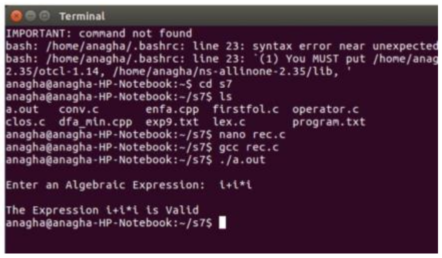
Recursive Descent Parser

```
# include < stdio .h >

# include < ctype .h >

# include < string .h >

void Tprime () ;

void Eprime () ;

void E () ;

void check () ;

void T () ;

char expression [10];

int count , flag ;

int main ()

{

count = 0;

flag = 0;

printf ("\ nEnter an Algebraic Expression :\t") ;

scanf ("%s", expression ) ;

E () ;

if (( strlen ( expression ) == count ) && ( flag == 0) )

{

printf ("\ nThe Expression %s is Valid \n", expression ) ;

}

else

{

printf ("\ nThe Expression %s is Invalid \n", expression ) ;

}
```

```
}

void E ()

{

T () ;

Eprime () ;

}

void T ()

{

check () ;

Tprime () ;

}

void Tprime ()

{

if( expression [ count ] == ’*’)

{

count ++;

check () ;

Tprime () ;

}

}

void check ()

{

if( isalnum ( expression [ count ]) )

{

count ++;

}

else if( expression [ count ] == ’(’)
{
count ++;
E () ;
if( expression [ count ] == ’)’)
{
count ++;
}
else
{
flag = 1;
}
}
else
{
flag = 1;
}
}
void Eprime ()
{
if( expression [ count ] == ’+’)
{
count ++;
T () ;
Eprime () ;
}
}
```

Output

Input String:
i+i*i
Output:



```
IMPORTANT: command not found
bash: /home/anagha/.bashrc: line 23: syntax error near unexpected
bash: /home/anagha/.bashrc: line 23: `(1) You MUST put /home/anag
2.35/otcl-1.14, /home/anagha/ns-allinone-2.35/lib, '
anagha@anagha-HP-Notebook:~$ cd s7
anagha@anagha-HP-Notebook:~/s7$ ls
a.out   conv.c       enfa.cpp  firstfol.c  operator.c
clos.c  dfa_min.cpp  exp9.txt  lex.c       program.txt
anagha@anagha-HP-Notebook:~/s7$ nano rec.c
anagha@anagha-HP-Notebook:~/s7$ gcc rec.c
anagha@anagha-HP-Notebook:~/s7$ ./a.out

Enter an Algebraic Expression:  i+i*i

The Expression i+i*i is Valid
anagha@anagha-HP-Notebook:~/s7$ 
```

Exercise 12

Aim Construct a Shift Reduce Parser for a given language.

```
1 START

2 Input the expression .

3 Loop forever .

4 For top of the stack symbol , s and next input symbol a

5 case action o T [s , a ]

6 Shift x :( x is a STATE number ) .

7 Push a , then x on the top of the stack .

8 Advance input to point to the next input symbol .

9 Reduce y :( y is a PRODUCTION number ) .

10 Assume that the production is of the form A == beta .

11 Pop 2+[ beta ] symbols of the stack . At this point , top of the stack

should be a state symbol says ; push A , then go to T [S , A ] on the

top of the stack output the production .

12 Accept return ... a successful parse .

13 Default error ... input string is not in the language .

14 STOP
```

Program

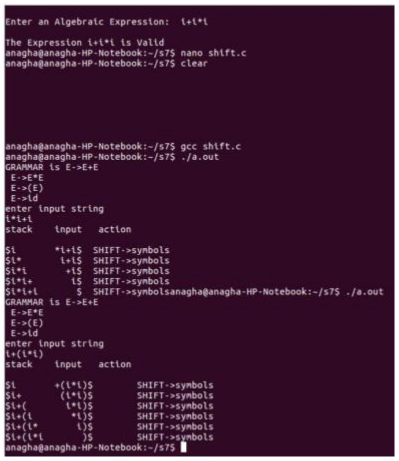
Shift Reduce Parser

```
# include < stdlib .h >
# 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 (" enter input string ") ;

scanf (" %s", a ) ;

c = strlen ( a ) ;

strcpy ( act ,"SHIFT - >") ;

puts (" stack \t input \t action ") ;

for ( k =0 , i =0; j < c ; k ++ , i ++ , j ++)

{

if( a [ j ]== ’i’ && a [ j +1]== ’d’)

{

stk [ i ]= a [ j ];

stk [ i +1]= a [ j +1];

stk [ i +2]= ’\0 ’;

a [ j ]= ’ ’;

a [ j +1]= ’ ’;

printf ("\n$%s\t%s$\t%sid ",stk ,a , act ) ;

check () ;

}

else

{
```

```
stk [ i ]= a [ j ];

stk [ i +1]= ’\0 ’;

a [ j ]= ’ ’;

printf ("\n$%s\t%s$\t% ssymbols ",stk ,a , act ) ;

check () ;

}

}

}

void check ()

{

strcpy ( ac ," REDUCE TO E") ;

for( z =0; z < c ; z ++)

if( stk [ z ]== ’i’ && stk [ z +1]== ’d’)

{

stk [ z ]= ’E’;

stk [ z +1]= ’\0 ’;

printf ("\n$%s\t%s$\t%s",stk ,a , ac ) ;

j ++;

}

for( z =0; z < c ; z ++)

if( stk [ z ]== ’E’ && stk [ z +1]== ’+’ && stk [ z +2]== ’E’)

{

stk [ z ]= ’E’;

stk [ z +1]= ’\0 ’;

stk [ z +2]= ’\0 ’;

printf ("\n$%s\t%s$\t%s",stk ,a , ac ) ;

i =i -2;
```

```
}

for( z =0; z < c ; z ++)

if( stk [ z ]== ’E’ && stk [ z +1]== ’*’ && stk [ z +2]== ’E’)

{

stk [ z ]= ’E’;

stk [ z +1]= ’\0 ’;

stk [ z +1]= ’\0 ’;

printf ("\n$%s\t%s$\t%s",stk ,a , ac ) ;

i =i -2;

}

for( z =0; z < c ; z ++)

if( stk [ z ]== ’(’ && stk [ z +1]== ’E’ && stk [ z +2]== ’)’)

{

stk [ z ]= ’E’;

stk [ z +1]= ’\0 ’;

stk [ z +1]= ’\0 ’;

printf ("\n$%s\t%s$\t%s",stk ,a , ac ) ;

i =i -2;

}

}
```

Input String:
i+i*i
Output:



```
Enter an Algebraic Expression:  i+i*i

The Expression i+i*i is Valid
anagha@anagha-HP-Notebook:~/s7$ nano shift.c
anagha@anagha-HP-Notebook:~/s7$ clear







anagha@anagha-HP-Notebook:~/s7$ gcc shift.c
anagha@anagha-HP-Notebook:~/s7$ ./a.out
GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->id
enter input string
i*i+i
stack    input   action

$i       *i+i$  SHIFT->symbols
$i*       i+i$  SHIFT->symbols
$i*i       +i$  SHIFT->symbols
$i*i+       i$  SHIFT->symbols
$i*i+i       $  SHIFT->symbolsanagha@anagha-HP-Notebook:~/s7$ ./a.out
GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->id
enter input string
i+(i*i)
stack    input   action

$i       +(i*i)$        SHIFT->symbols
$i+       (i*i)$        SHIFT->symbols
$i+(       i*i)$        SHIFT->symbols
$i+(i       *i)$        SHIFT->symbols
$i+(i*       i)$        SHIFT->symbols
$i+(i*i       )$        SHIFT->symbols
anagha@anagha-HP-Notebook:~/s7$
```

Exercise 13

Aim Write a program to perform constant propagation.

Algorithm

```
1 Start
2 For all statement in the program do begin
3 for each output v of s do valout (v , s ) = unknown
4 for each input w of s do
5 if w is a variable then valin (w , s ) = unknown
6 else valin (w , s ) = constant value of w
7 end
```

Program

```
# include < stdio .h >

# include < string .h >

# include < ctype .h >

# include < stdlib .h >

void input () ;

void output () ;
void change (int p , char * res ) ;

void constant () ;

struct expr

{

char op [2] , op1 [5] , op2 [5] , res [5];

int flag ;

} arr [10];

int n ;

void main ()

{

input () ;

constant () ;

output () ;

}

void input ()

{

int i ;

printf ("\n\ nEnter the maximum number of expressions : ") ;

scanf ("%d" ,& n ) ;

printf ("\ nEnter the input : \n") ;

for ( i =0; i < n ; i ++)

{

scanf ("%s", arr [ i ]. op ) ;

scanf ("%s", arr [ i ]. op1 ) ;

scanf ("%s", arr [ i ]. op2 ) ;

scanf ("%s", arr [ i ]. res ) ;

arr [ i ]. flag =0;
```

```
}
}
void constant ()
{
int i ;
int op1 , op2 , res ;
char op , res1 [5];
for ( i =0; i < n ; i ++)
{
if( isdigit ( arr [ i ]. op1 [0]) && isdigit ( arr [ i ]. op2 [0]) || strcmp ( arr [ i]. op ,"=") ==0)
 /* if both digits , store them in variables */
{
op1 = atoi ( arr [ i ]. op1 ) ;
op2 = atoi ( arr [ i ]. op2 ) ;
op = arr [ i ]. op [0];
switch ( op )
{
case ’+’:
res = op1 + op2 ;
break ;
case ’-’:
res = op1 - op2 ;
break ;
case ’*’:
res = op1 * op2 ;
break ;
case ’/’:
}

res = op1 / op2 ;

break ;

case ’=’:

res = op1 ;

break ;

}

sprintf ( res1 ,"%d", res ) ;

arr [ i ]. flag =1; /* eliminate expr and replace any operand below that

uses result of this expr */

change (i , res1 ) ;

}

}

}

void output ()

{

int i =0;

printf ("\ nOptimized code is : ") ;

for ( i =0; i < n ; i ++)

{

if (! arr [ i ]. flag )

{

printf ("\n%s %s %s %s", arr [ i ]. op , arr [ i ]. op1 , arr [ i ]. op2 , arr [ i ]. res ) ;

}

}

}

void change (int p , char * res )

{
```

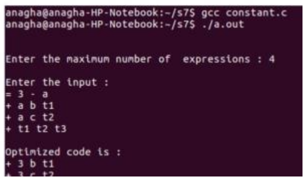
```
int i ;

for ( i = p +1; i < n ; i ++)

{

if( strcmp ( arr [ p ]. res , arr [ i ]. op1 ) ==0)

strcpy ( arr [ i ]. op1 , res ) ;

else if( strcmp ( arr [ p ]. res , arr [ i ]. op2 ) ==0)

strcpy ( arr [ i ]. op2 , res ) ;

}

}
```

Output

Input expressions:

```
= 3 - a
+ a b t1
+ a c t2
+ t1 t2 t3
```

Output:



```
anagha@anagha-HP-Notebook:~/s7$ gcc constant.c
anagha@anagha-HP-Notebook:~/s7$ ./a.out


Enter the maximum number of  expressions : 4

Enter the input :
= 3 - a
+ a b t1
+ a c t2
+ t1 t2 t3

Optimized code is :
+ 3 b t1
```

Exercise 14

Aim . Implement Intermediate code generation for simple expressions.

Algorithm

1START

2 Enter the expression .

3 If there is an ’() ’ compute the expression inside in it fast .

4 Else find ’/’ operator and the operands on the left and right side of that operator , compute it and assign it to a new variable .

5 Repeat the above step for all the operators in the order ’*’, ’+’,’-’

6 Assign the last variable to another new variable .

7 STOP

Program

Intermediate Code Generation

```
# include < stdio .h >

# include < string .h >

# include < stdlib .h >

int i =1 , j =0 , no =0 , tmpch =90;

char str [100] , left [15] , right [15];

void findopr () ;

void explore () ;

void fleft (int ) ;

void fright (int ) ;

struct exp

{

int pos ;

char op ;

} k [15];

void main ()

{

printf (" Enter the Expression :") ;

scanf ("%s", str ) ;

printf ("The intermediate code :\t\ tExpression \n") ;

findopr () ;

explore () ;

}
```

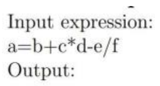
```
void findopr ()

{

for( i =0; str [ i ]!= ’\0 ’; i ++)

if( str [ i ]== ’:’)

{

k [ j ]. pos = i ;

k [ j ++]. op =’:’;

}

for( i =0; str [ i ]!= ’\0 ’; i ++)

if( str [ i ]== ’/’)

{

k [ j ]. pos = i ;

k [ j ++]. op =’/’;

}

for( i =0; str [ i ]!= ’\0 ’; i ++)

if( str [ i ]== ’*’)

{

k [ j ]. pos = i ;

k [ j ++]. op =’*’;

}

for( i =0; str [ i ]!= ’\0 ’; i ++)

if( str [ i ]== ’+’)

{

k [ j ]. pos = i ;

k [ j ++]. op =’+’;

}

for( i =0; str [ i ]!= ’\0 ’; i ++)
```

```
if( str [ i ]== ’-’)

{

k [ j ]. pos = i ;

k [ j ++]. op =’-’;

}

}

void explore ()

{

i =1;

while ( k [ i ]. op != ’\0 ’)

{

fleft ( k [ i ]. pos ) ;

fright ( k [ i ]. pos ) ;

str [ k [ i ]. pos ]= tmpch - -;

printf ("\t%c := %s%c%s\t\t", str [ k [ i ]. pos ] , left , k [ i ]. op , right ) ;

for ( j =0; j < strlen ( str ) ; j ++)

if( str [ j ]!= ’$’)

printf ("%c", str [ j ]) ;

printf ("\n") ;

i ++;

}

fright ( -1) ;

if( no ==0)

{

fleft ( strlen ( str ) ) ;

printf ("\t%s := %s\n", right , left ) ;

exit (0) ;
```

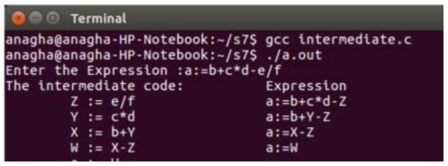
```
}

printf ("\t%s := %c\n", right , str [ k [ - - i ]. pos ]) ;

}

void fleft (int x )

{

int w =0 , flag =0;

x - -;

while ( x != -1 && str [ x ]!= ’+’ && str [ x ]!= ’*’&& str [ x ]!= ’=’&& str [ x ]!= ’

\0 ’&& str [ x ]!= ’-’&& str [ x ]!= ’/’&& str [ x ]!= ’:’)

{

if( str [ x ]!= ’$’&& flag ==0)

{

left [ w ++]= str [ x ];

left [ w ]= ’\0 ’;

flag =1;

}

x - -;

}

}

void fright (int x )

{

int w =0 , flag =0;

x ++;

while ( x != -1 && str [ x ]!= ’+’&& str [ x ]!= ’*’&& str [ x ]!= ’\0 ’&& str [ x ]!= ’

=’&& str [ x ]!= ’:’&& str [ x ]!= ’-’&& str [ x ]!= ’/’)

{

if( str [ x ]!= ’$’&& flag ==0)
```

```
{
right [ w ++]= str [ x ];
right [ w ]= ’\0 ’;
str [ x ]= ’$’;
flag =1;
}
x ++;
}
}
```

Output



Input expression:
a=b+c*d-e/f
Output:

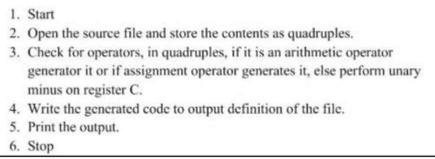


```
anagha@anagha-HP-Notebook:~/s7$ gcc intermediate.c
anagha@anagha-HP-Notebook:~/s7$ ./a.out
Enter the Expression :a:=b+c*d-e/f
The intermediate code:          Expression
        Z := e/f                a:=b+c*d-Z
        Y := c*d                a:=b+Y-Z
        X := b+Y                a:=X-Z
        W := X-Z                a:=W
```

Exercise 15

Aim Implement the back end of the compiler which takes the three address code and produces the 8086 assembly language instructions that can be assembled and run using an 8086 assembler. The target assembly instructions can be simple move, add, sub, jump etc

Algorithm



1. Start
2. Open the source file and store the contents as quadruples.
3. Check for operators, in quadruples, if it is an arithmetic operator generator it or if assignment operator generates it, else perform unary minus on register C.
4. Write the generated code to output definition of the file.
5. Print the output.
6. Stop

Program

Lex Program Exercises

Syntax Of Lex Program:

```
%{
   definition section
%}

%%
   rules section
   /*pattern       action     */
%%

/*user subroutine section*/
```

```
main()

{

  yylex();

}
```

Commands To create executable program:

To create C file:

```
lex   <pgmname.l>
```

To compile & to create executable file:

```
cc  lex.yy.c  –ll
```

To execute program:

```
./a.out
```

1. Aim: Program to count the number of vowels and consonants in a given string.

Procedure:

1. In definition section, declare and initiate the variables, which are used to count the total number of vowels and consonants.

2. In rule section,

(i)Define the pattern which is used to recognize vowels. (In action part) , If a character from the input string matches with this pattern then increment the vowel count.

(ii)Define the pattern which is used to recognize consonants. (In action part)If a character from the input string matches with this pattern then increment the consonant count.

3.In user subroutine part,

(i) In main() ,Call yylex(). Then print the total number of vowels and consonants in the input string.

Progra m:

```
%{
 int vowels=0,cons=0;    /* Initialize counters */
%}
%%
[aeiouAEIOU] { vowels++; }    /* Count vowels */
[a-zA-Z]      { cons++; }    /* Count consonants */
%%
main()
{
yylex();
printf("\nThe number of vowels       = %d ",vowels);
printf("\nThe number of consonants = %d\n",cons);
}
```

Sample Input & output:

```
Enter the input string: Bangalore

The number of vowels      =  4

The number of consonants=  5
```

2. Aim: Program to count the number of characters, words, spaces and lines in a given input file.

Procedure:

1. In definition section, declare and initiate the variables, which are used to count the total

   number of characters, words, spaces and lines in a given input file.

2. In rule section,

(i)Define the pattern which is used to recognize words by specifying any of the word delimiters (one or more occurrences) except tab and new line . (In action part) , If a character from the input string matches with this pattern then increment the word count by 1 and the character count. by yyleng.(yylength is the length of yytext array.)

(ii)Define the pattern which is used to recognize space. (In action part)If a character from the input string matches with this pattern then increment the space count as well as the total character count by 1.

(iii)Define the pattern which is used to recognize new line. (In action part) if a character from the input string matches with this pattern then increment the total line count as well as the total character count by 1.

3.In user subroutine part,

(i) In main(),open the input text file in read mode.

(ii) Call yylex(). Then print the total number of words, characters, spaces and lines in the given input file.

Program:

```
%{
 int chars=0,words=0,spaces=0,lines=0;    /* Initialize counters */
%}
%%
[^ \t\n]+     { words++; chars = chars + yyleng; }
[ ]        { spaces++; chars++; }
[\n]        { lines++; chars++; }
%%

main()
{
 yyin=fopen(“input.txt”,”r”);
 yylex();
 fclose(yyin);
 printf("\nWords=%d",words);
 printf("\nCharacters = %d",chars);
 printf("\nSpaces = %d",spaces);
 printf("\nLines = %d",lines);
}
```

Sample Input & output:

Input.txt

```
This is bangalore


Words =3

Characters=17

Spaces=2

Lines=1
```

3 (a) Aim: Program to count number of Positive and negative integers

Procedure:

1. In definition section, declare and initiate the variables, which are used to count the total

   number of positive and negative integers.

2. In rule section,

(i)Define the pattern which is used to recognize positive integers (i.e) '+' sign(optional) and 0-9 digits (1 or more occurrences). (In action part) , If a number from the input data matches with this pattern then increment the positive count.

(ii) Define the pattern which is used to recognize positive integers (i.e) '-' sign and 0-9 digits (1 or more occurrences). (In action part) , If a number from the input data matches with this pattern then increment the negative count.

3.In user subroutine part,

(i) In main() ,Call yylex(). Then print the total number of positive and negative integers in the input string.

Program:

```
%{
 int pos_ints=0,neg_ints=0;
%}
%%
"+"?[0-9]+ { pos_ints++; }
"-"[0-9]+ { neg_ints++; }
%%
main()
{
    printf("\n Enter the numbers:");
 yylex();
 printf("\n The number of +ve integers = %d",pos_ints);
 printf("\n The number of -ve integers = %d",neg_ints);
}
```

Sample Input & output:

```
Enter the numbers : 12  -12  -14  31
The number of +ve integers = 2
The number of -ve integers = 2
```

3(b) Aim : Program to count number of Positive and negative fractions

Procedure:

1. In definition section, declare and initiate the variables, which are used to count the total number of positive and negative fractions.

2. In rule section,

(i)Define the pattern which is used to recognize positive fractions (i.e) ‘+’ sign(optional) and digits (0 or more occurrences) followed by decimal point and digits(0-9)one or more occurrences. (In action part) , If a number from the input data matches with this pattern then increment the positive count.

(ii) Define the pattern which is used to recognize positive fractions (i.e) ‘-’ sign and digits (0 or more occurrences)followed by decimal point and digits (0-9) one or more occurrences. (In action part) , If a number from the input data matches with this pattern then increment the negative count.

3.In user subroutine part,

(i) In main() ,Call yylex(). Then print the total number of positive and negative fractions in the input string.

Program:

```
%{
 int pos_fracts=0,neg_fracts=0;
%}

%%
"+"?[0-9]*[.][0-9]+ { pos_fracts++; }
"-"[0-9]*[.][0-9]+  { neg_fracs++;  }
%%
main()
{
 printf(“\n Enter the numbers:”);
 yylex();
 printf("\n The number of +ve fractions = %d",pos_fracts);
 printf("\n The number of -ve fractions = %d",neg_fracts);
}
```

Sample Input & output:

```
Enter the numbers : 12.5  -12.7  -14.3  31.23
The number of +ve fractions = 2
The number of -ve fractions = 2
```

4. Aim: Program to count the numbers of comment lines in a given C program. Also eliminate them and copy that program into separate file.

Procedure:

1. In definition section, declare and initiate the variables, which are used to count the total

   number of comment lines. Define a state as COMMENT.

2. In rule section,

(i)If the input string has “/*” ,change the state of LEX to COMMENT.

(ii)In COMMENT state ,

if “*/” encountered then go to the default state of LEX(0).And increment the comment count.

Else if any other character or new line encountered then retain in the COMMENT state and ,don’t write the output on the output file.

3. If any other character or new line in LEX’s default state encountered then print that on output file.

4.In user subroutine part,

(i) In main() ,open the input file using yyin pointer and output file using yyout pointer. Call yylex(). Then print the total number of comment lines.

Program:

```
%{
 int c=0;
%}
%s COMMENT
%%
“/*” {BEGIN COMMENT;}
<COMMENT>“*/” {BEGIN 0;c++;}
< COMMENT>.  ;
< COMMENT>\n ;
. ECHO;
\n ECHO;
%%
main()
{
    yyin=fopen(“input..c”,”r”);
yyout=fopen(“output.c”,”r”);
yylex();
fclose(yyin);
fclose(yyout);
printf("\nNumber of comment lines = %d",c);
}
```

Sample I/O:

Input.c:

```
#include<stdio.h>/* header file*/
main()
{
 int a; /* declaration */
 a=10;
 printf("\n a=%d",a);
}
```

Output.c:

```
#include<stdio.h>
main()
{
 int a;
 a=10;
 printf("\n a=%d",a);
}
```

Number of comment lines = 2

5. Aim : Program to count the number of ‘scanf’ and ‘printf’ statements in a C program. Replace them with ‘readf’ and ‘writef’ statements respectively.

Procedure:

1. In definition section, declare and initiate the variables, which are used to count the total

number of scanf and printf statements in a C program. Define the file pointer also to access the C program.

2. In rule section,

(i)Define the pattern which is used to recognize scanf. (In action part) , If the input file consists of scanf then increment the count for scanf. And also printing it in the output file by replacing scanf with readf.

(ii) Define the pattern which is used to recognize printf. (In action part) , If the input file consists of printf then increment the count for printf. And also printing it in the output file by replacing printf with writef.

(iii)Define the pattern to read any other characters except printf and scanf and write them into the output file without changes.

3.In user subroutine part,

(i) Define the main() to get the I/O file names in command line arguments. n main() ,read the input file using yyin pointer and write the output file using yyout then Call yylex(). Then print the total number of scanf and printf .

Program:

```
%{
 int s=0,p=0;
%}
%%
scanf    { s++; strcpy(yytext,”readf”);ECHO;}
printf    { p++; strcpy(yytext,”writef”);ECHO;}
.|\n     { ECHO; }
%%
main(int argc,char* argv[])
{
        yyin=fopen(argv[1],"r");
       yyout=fopen(argv[2],”w”);
        yylex();
        printf("\nNumber of printf statements = %d",p);
printf("\nNumber of scanf statements = %d",s);
}
```

Sample I/O:

Input.c:

```
#include<stdio.h>
main()
{
   int a,b;
   scanf(“%d”,&a);
scanf(“%d”.,&b);
printf(“%d”,a);
printf(“%d”,b);
}
```

Output.c:

```
#include<stdio.h>

main()

{

   int a,b;

   readf(“%d”,&a);

readf(“%d”.,&b);

writef(“%d”,a);

writef(“%d”,b);

}
```

```
Number of printf statements = 2

Number of scanf statements = 2
```

6. Aim: Program to recognize a valid arithmetic expression and identify the identifiers and operators present. Print them separately.

Procedure:

1. In rule section,

(i)Define the pattern, which is used to recognize identifiers (which are named using alphabets and digits) for one or more occurrences. (In action part) , If the input expression consists of identifier name then print the yytext(which is currently having the identifier name.)

(ii) Define the pattern which is used to recognize operator (+,-,*,/,++,--,=,%). (In action part) , If the input expression consists of operator then print the yytext.(which is currently having the operator).

(iii)Define the pattern to recognize digits (one or more occurrences) In action part print it as number or constant.

}

(iv)Define the pattern to identify the invalid input such as identifier name started with a digit (78df – invalid identifier). In action part set the valid flag as enabled.

(v)Define the pattern to recognize invalid expression which is ended with operator.(a+f+ -invalid expression). In action part set the valid flag as enabled.

2.In user subroutine part,

In main() , Call yylex().

Program:

```
%{

%}

%%

[a-zA-Z][a-zA-Z0-9]+?    {printf("\n id=%s",yytext);}

[-+*/]                {printf("\n operator=%s",yytext);}

[0-9]+                {printf("\n number=%s",yytext);}

[0-9]+[a-zA-Z][a-zA-Z0-9]+     {valid=1;}

[-+*/]\n            {valid=1;}

\n              return;

%%

main()

{

printf("\n enter an expression:");

yylex();

if(valid==1) printf("\n Invalid!");

else printf("\n valid!");

}
```

Sample I/O:

```
enter an expression: a+40

id = a

operator=+

number=40

valid!
```

7.Aim : Program to recognize whether a given sentence is simple or compound.

Procedure:

1. In definition section, declare and initiate the variable, which is to be used as flag to denote whether an entered sentence is simple or compound.

2. In rule section,

(i)Define the pattern which is used to recognize any number of characters, followed by space ,followed by “and” or “or” or “because”, followed by space ,followed by any number of characters. (In action part) , from the input statement if there any match to be found with this pattern then set the flag.

(ii)Define the pattern which will read any character or new line which are all not matched with the previous pattern.

3.In user subroutine part,

(i) In main() ,Call yylex(). If the flag is set then print the sentence given by the user as compound sentence, else print the given sentence is a simple sentence.

Program:

```
%{

    # include <stdio.h>

 int c=0;
%}
%%
.*[ ]"and"[ ].* { c=1; }
.*[ ]"or"[ ].* { c=1; }
.*[ ]"because"[ ].* { c=1; }
. | \n ;
%%
int main()
{
 printf("\nEnter a Sentence : \n");
 yylex();
 if(c==0)
  printf("\nSimple Sentence.");
 else
  printf("\nCompound Sentence.");
 return 0;
}
```

Sample I/O:

```
Enter a Sentence : She is a computer science and engineering student.


Compound statement
```

8.Aim: Program to recognize and count the number of identifiers in a given input file.

Procedure:

1. In definition section, declare and initiate the variable, which is used to count the total number of identifiers.

2.Define a new state IDENTIFIER

3 In rule section,

(i)Define the pattern which is used to recognize identifiers (in the declaration part of the program itself).In the given input program if a statement has int, char, float, double, long or unsigned then change the state of LEX from default state to IDENTIFIER state. (In action part), If a character from the input string matches with this pattern then increment the vowel count.

(ii)Define the following patterns in the IDENTIFIER state of LEX.

(a)(Data type is already identified in previous step (i.e) int a,b;). Define a pattern to recognize one or more spaces, followed by alphabet (identifier name should be started with alphabet), followed by zero or more occurrences of alphabet or digit, followed by cama (,).If this pattern matched with the input given by the user then, increment the total number of identifier count.

(b)Define a pattern, which is having the same patterns to recognize as the previous step to count identifier, but if it is ended with semicolon (;), then go to the default state (0) of the LEX.

(c) Define a pattern which is used to read the input from the file, which are all not matched with the previous 2 patterns.(i.e) pattern to recognize any character or new line character.

3.In user subroutine part,

In main() ,open the input file using yyin pointer and then Call yylex(). Then print the total number of identifiers.

Program:

```
%{
 int c=0;
%}

%s     IDEN
%%
“int”[  ]+  {BEGIN IDEN;}
“char”[  ]+  {BEGIN IDEN;}
<IDEN>[a-zA-Z][a-zA-Z0-9]+? {c++;}
<IDEN>”[“[0-9]+”]”    ;
<IDEN>; {BEGIN 0;}
<IDEN> \n ;
. | \n ;
%%

main()
{
    yyin=fopen(“input.c”,”r”);
yylex();
fclose(yyin);
printf("\nTotal number of identifiers = %d",c);
}
```

Sample I/O:

Input.c:

```
#include<stdio.h>
main()
```

```
{
 int a[10],b,c;
c=a=10;
b=a+c;
}
```

```
Total number of identifiers= 3
```

Yacc Program Exercises

Syntax Of Yacc Program:

```
%{    /* definition section  */    %}

% token  symbolic-token-list (i.e) terminals used in grammar

%%
   /* rules section : Grammar rules */
%%

/*user subroutine section*/
main()
{
   yyparse();
}
```

Syntax Of Lex Program:

```
%{
   # include "y.tab.h"
definition section
%}

%%
   rules section

   /*pattern       action     */
%%
```

Commands To create executable program:

To generate both y.tab.c & y.tab.h (token definition –d):

```
yacc –d  <pgmname.y>
```

To generate C file from the Lex program:

```
lex   <pgmname.l>
```

To compile and to link C files

```
cc  y.tab.c  lex.yy.c  –ll
```

To Execute the program:

```
./a.out
```

Aim : Program to test the validity of a simple expression involving operators +, -, * and /.

Procedure:

Yacc

1. Define the symbolic token NUMBER & ID.

2. Define the precedence of the operators (+,-,*,/) to be used in expressions as left associative.

3. In Rules section, Write the grammar rules to recognize the expression involving addition(+), subtraction(-), multiplication(*) & division(/).

Grammar G:

E E+E | E-E | E*E | E/E | id

4.In user subroutine section, call yyparse() to check the syntax of the input expression. If it is valid then print the expression as valid, else if it is invalid the yyerror can be called automatically by the parser. User can also redefine the yyerror();

Lex

1.Lexer will provide the tokens for our parser. In definition section include the “y.tab.h” which are the generated token codes from the parser.

2.In rules section write the patterns for the following :

(a) if the input expression has Digit [0-9], then it will be matched with the token NUMBER (terminal defined in grammar for parser).

(b) If the input has identifier name it will match with pattern to recognize identifier.(terminal defined in the grammar)

(c) If the expression has white space then ignore. (do no action)

(d) If new line, it will denote the end of input token, which will tell the parser to not read more.

(e) Write the last rule with pattern to return any character, otherwise not handled as a single character token to the parser.

Program:

Yacc Program:

```
%{
%}

%token NUMBER ID
%left '+''-'
%left '*' '/'

%%
        valid : expression {printf("\n valid exprn!");}
expression : expression '+' expression
     | expression '-' expression
                 | expression '*' expression
               | expression '/' expression
                  | NUMBER
            | ID
               ;
%%
```

```
int main()

{

 printf("\nEnter an expression: ");

 yyparse();

}


int yyerror(char *s)

{

 printf("\n %s\n",s);

 exit(1);

}
```

Lex Program

```
%{

 # include "y.tab.h"

%}


%%

            [a-zA-Z][a-zA-Z0-9]+? {return ID;}

[0-9]+     { return NUMBER; }

[ \t] ;

\n     { return 0; }

.    { return yytext[0]; }

%%
```

Sample I/O:

Enter an Expression : 5+3

valid exprn!2.Program to recognize nested IF control statements and display the number of levels of nesting.

Procedure:

Yacc:

1. In definition section define an extern variable to count the total number of levels of if condition. Then define the symbolic tokens EXPR,STMT.

2. In Rules section, Write the grammar rules to recognize the if condition

Grammar G:

stmt if cond ob Stmt1 cb

Stmt1 if cond ob stmt1 cb stmt1 | ξ

3.In user subroutine section, call yyparse() to check the syntax of the input statements. If it is valid then print the statement as valid, else if it is invalid the yyerror can be called.User can also redifine the yyerrror();

Lex :

1.Lexer will provide the tokens for our parser. In definition section include the “y.tab.h” which are the generated token codes from the parser.

2.In rules section write the patterns for the following :

(a) in the input if there is a “if” return the token code for “if”.

}

(a)if the input statement has logical condition with in ( ) , then it will be matched with the token cond.(terminal defined in grammar for parser).

(b) if the expression has white space then ignore .(do no action)

(c)if any other arithmetic expression with in { },then it will be matched with the STMT1.

Yacc Program

```
%{
extern int l2;
%}

%token IF  COND OB CB

%%
statement : IF COND OB statement1 CB    ;
statement1 : IF COND OB statement1 CB statement1              |           ;
%%
int main()
{
 printf("\nEnter Conditions  : \n");
 yyparse();
 printf("\nThe Maximum Level of Nesting is %d\n",l2);
 return 0;
}
```

```
int yyerror(char *s)
{
 printf("\n%s\n",s);
 exit(1);
}
```

Lex Program

```
%{
#include "y.tab.h"
 int l1=0,l2=0;
%}
%s condition
%%
[\n]*[ ]*if       { BEGIN condition;  l1++;      if(l2<l1) { l2=l1; }    return IF; }
<condition>[ ]*"(".+")"    { BEGIN 0;return COND; }
[\n]*[ ]*"{"     { return OB; }
[\n]*[ ]*"}"     { l1--; return CB; }
[^"if""("")""{""}"]* ;
%%
```

3.Aim: Program to recognize a valid arithmetic expression that uses operators +, -, * and /.

Procedure:

Yacc

1. Define the symbolic token NUMBER.

2. Define the precedence of the operators (+,-,*,/) to be used in expressions as left or non associative .

3. In Rules section, Write the grammar rules to recognize the expression involving addition(+), subtraction(-), multiplication(*) & division(/).

Grammar G:

E ⍰ E+E | E-E | E*E | (E) | -E | id

4.In user subroutine section, call yyparse() to check the syntax of the input expression. If it is valid then print the expression as valid, else if it is invalid the yyerror can be called automatically by the parser. User can also redefine the yyerrror();

Lex

1.Lexer will provide the tokens for our parser. In definition section include the “y.tab.h” which are the generated token codes from the parser.

2.In rules section write the patterns for the following :

(a) if the input expression has identifier name started with alphabet it will be matched with the terminal id so return that token code for id.

(b) if the input expression has Digit [0-9], then it will be matched with the token NUMBER (terminal defined in grammar for parser).

(c) if the expression has white space then ignore .(do no action)

(d) if new line ,it will denote the end of input tokens, which will tell the parser to not read more.

(e) Write the last rule with pattern to return any character, which are not matched with the above rules. Return the character to yacc(to return operators).

Yacc Program

```
%{

%}


%token NUMBER

%left '+''-'

%left '*' '/'

%nonassoc UNARY


%%

expression : expression '+' expression

                |  expression '-' expression

                          |  expression '*' expression

                          |  expression '/' expression

      | '(' expression ')'

                          | '-' expression %prec UNARY
```

```
                              | NUMBER

                              ;

%%


int main()

{

 printf("\nEnter an expression: ");

 yyparse();

 printf("\nThe entered expression is valid.\n");

 return 0;

}


int yyerror(char *s)

{

 printf("\n %s\n",s);

 exit(1);

}
```

Lex Program

```
%{

# include "y.tab.h"

%}


%%

[0-9]+ { return NUMBER; }

[ \t] ;
```

```
\n    { return 0; }

.    { return yytext[0]; }

%%
```

Sample I/O:

```
Enter an Expression : 2+3-(5/4)

The entered expression is valid
```

4.Aim: Program to recognize a valid variable, which starts with a letter, followed by any number of letters or digits.

Procedure:

Yacc

1. Define the symbolic token NUMBER,LETTER.

2. In Rules section, Write the grammar rules to recognize the valid variable which starts with a letter, followed by any number of letters or digits.

   Grammar G:

id id LETTER | LETTER

id id NUMBER | NUMBER

3.In user subroutine section, call yyparse() to check the syntax of the input viable name. If it is valid then print the expression as valid, else if it is invalid the yyerror can be called automatically by the parser. User can also redefine the yyerrror();

Lex

1.Lexer will provide the tokens for our parser. In definition section include the “y.tab.h” which are the generated token codes from the parser.

2.In rules section write the patterns for the following:

(a) If the input variable is started with Digit [0-9] (i.e) except alphabets a-z, then it will be return the token code for digit (which is invalid variable name).

(b) If the variable name is started with alphabet [a-z] return the token code as LETTER.

Yacc Program

```
%{
%}

%token LETTER NUMBER



%%
    validid : iden { printf("\n valid!");}
iden  : LETTER
    : iden NUMBER
    : iden LETTER
          ;


%%

int main()
```

```
{
 printf("\nEnter a string : ");
 yyparse();
 printf("\nThe entered string is a valid identifier.\n");
 return 0;
}

int yyerror(char *s)
{
 printf("\n%s\n",s);
 exit(1);
}
```

Lex Program

```
%{
# include "y.tab.h"
%}
%%
 [a-zA-Z]  { return LETTER; }
 [0-9]+  {return NUMBER;}
%%
```

Sample I/O:

```
Enter a string : name
```

The entered string is a valid identifier.5.Program to evaluate an arithmetic expression involving operators +, -, * and /.

Procedure:

Yacc

1. Define the symbolic token NUMBER.

2. Define the precedence of the operators (+,-,*,/) to be used in expressions as left or non associative .

3. In Rules section, Write the grammar rules along with the actions to recognize the expression involving addition(+), subtraction(-), multiplication(*) & division(/). In action subsection write the code to evaluate the value according to the values of the symbols in the right hand side of the rule.

Grammar G:

```
    Stat ⍰ E        Print the value of E
E    ⍰ E+E        calculate as E3.val=E1.val+E2.val
| E-E       calculate as E3.val=E1.val-E2.val
| E*E       calculate as E3.val=E1.val*E2.val
| ( E )        calculate as E2.val=E1.val
| -E       calculate as E2.val=-E1.val
| E / E       if E2.val=0 then Print "divide by zero error"
| id
```

4.In user subroutine section, call yyparse() to check the syntax of the input expression. If it is valid then print the expression as valid, else if it is invalid the yyerror can be called automatically by the parser. User can also redefine the yyerrror();

Lex

1.Lexer will provide the tokens for our parser. In definition section include the “y.tab.h” which are the generated token codes from the parser. Define the variable yylval which will return the token associated value to parser.

2.In rules section write the patterns for the following :

(a) if the input expression has Digits [0-9], then it will be matched with the token NUMBER ,then yylval=numeric value of (yytext))(terminal defined in grammar for parser).

(b) if the expression has white space then ignore .(do no action)

(c) if new line ,it will denote the end of input tokens, which will tell the parser to not read more.

(d) write the last rule with pattern to recognize any character which are not matched with above rules. And return the character . (tp return operators in the input string to yacc grammar)

Yacc Program

```
%{

%}


%token NUMBER

%left '+''-'

%left '*' '/'

%nonassoc UNARY


%%

statement : expression       { printf("\nThe Value of the expression is %d\n",$1); }

    ;
```

```
      expression : expression '+' expression { $$=$1+$3; }
      |  expression '-' expression { $$=$1-$3; }
                        |  expression '*' expression { $$=$1*$3; }
                        |  expression '/' expression  {  if($3==0)
      {  printf("\nDivide By Zero Error!");
                                                       exit(0);
                  }
                                                     else     { $$=$1/$3; }
                                                   }
                        | '(' expression ')'                { $$=$2; }
                        | '-' expression %prec UNARY { $$=-$2; }
                        | NUMBER
                        ;
%%

int main()
{
 printf("\nEnter an expression: ");
 yyparse();
 return 0;
}
void yyerror(char *s)
{
 printf("\n %s\n",s);
 exit(1);
}
```

Lex Program

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

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

Sample I/O:

Enter an expression : 2+3-5

The value of the expression is 06.Program to recognize strings ‘aaab’, ‘abbb’, ‘ab’ and ‘a’ using the grammar   (an bn , n>=0).

Note: The problem statement is incorrect . It may be ((an bm , n,m>=0). Or ab,aabb,aaabbb. Our solution is for equal number of a’s and b’s.

Procedure:

Yacc

1. Define the symbolic tokens a & b .

2. In Rules section, Write the grammar rules to recognize the expression involving

Grammar G:

Exp expa expb | expa | expb

expa expa a | ξ

expb expb b | ξ

3.In user subroutine section, call yyparse() to check the syntax of the input expression. If it is valid then print the expression as valid, else if it is invalid the yyerror can be called automatically by the parser. User can also redefine the yyerror();

Lex

1.Lexer will provide the tokens for our parser. In definition section include the “y.tab.h” which are the generated token codes from the parser.

2. In rules section write the patterns for the following :

(a)If the input has ‘a’ then return the token code for a.

(b) if the input has ‘b’ then return the token code for b.

(c)if it is new line character then return the token code for new line character.

Yacc Program

```
%{

%}

%token a b END

%%
```

```
      validexpn : exp END {printf("\n valid!");}
exp : A1 B1   |   A1       |    B1  |     ;
A1  : A1 A |     ;
           B1  :  B1 B |     ;
%%
int main()
{
 printf("\nEnter a string : ");
 yyparse();
 printf("\nString recognized!\n");
}

int yyerror(char *s)
{
 printf("\n%s\n",s);
 exit(1);
}
```

Lex Program

```
%{
# include "y.tab.h"
%}
%%
a       { return a; }
b        { return b; }
. {return 0;}
```

```
\n {return END;}
%%
```

Sample I/O:

```
Enter a string: aaaaabbbbb
String recognized!
```

7.Program to recognize the grammar (an b, n>=10).

Procedure:

Yacc

1. Define the symbolic tokens X, Z & Y.

2. In Rules section, Write the grammar rules to recognize the expression involving

Grammar G:

```
Exp   a a a a a a a a a a   Exp1   b
Exp1   exp1  a  |   ξ
```

4.In user subroutine section, call yyparse() to check the syntax of the input expression. If it is valid then print the expression as valid, else if it is invalid the yyerror can be called automatically by the parser. User can also redefine the yyerror();

Lex

1.Lexer will provide the tokens for our parser. In definition section include the “y.tab.h” which are the generated token codes from the parser.

2.In rules section write the patterns for the following :

(a)If the input has ‘a’ then return the token code for a.

(b) if the input has ‘b’ then return the token code for b.

(c)if it is new line character then return the token code for new line character.

Yacc Program

```
%{
%}

%token a b END



%%
           validexpn : exp END;
 exp : a a a a a a a a a a  exp1   b   ;
 exp1 : exp1     a    |     ;

%%

int main()
{
 printf("\nEnter a string : ");
 yyparse();
 printf("\nString recognized!\n");
}

int yyerror(char *s)
{
 printf("\n%s\n",s);
 exit(1);
}
```

Lex Program

```
%{
# include "y.tab.h"
%}

%%
a        { return a; }
b       { return b; }
.            {return 0;}
\n        { return END; }
%%
```

Sample I/O:

```
Enter a string: aaaaaaaaaab
String recognized!
```

List of Mini Projects

2-pass Assembler for the working model of 8086.

Text Editor.

Linux shell for a set of commands.

Simple Lexical Analyzer.

The mini projects should be preferably implemented in C/C++ language. However, the program may be run under MS Windows, Unix or Linux environment. The students have to follow the following process activities/Steps.

Process Activities/Steps

Software Engineering processes are composed of many activities, notably the following. They are considered sequential steps in the Waterfall process, but other processes may rearrange or combine them in different ways.

Requirements Analysis

Extracting the requirements of a desired software product is the first task in creating it. While customers probably believe they know what the software is to do, it may require skill and experience in software engineering to recognize incomplete, ambiguous or contradictory requirements.

Specification

Specification is the task of precisely describing the software to be written, in a mathematically rigorous way. In practice, most successful specifications are written to understand and fine-tune applications that were already well-developed, although safety-critical software systems are often carefully specified prior to application development. Specifications are most important for external interfaces that must remain stable.

Software architecture

The architecture of a software system refers to an abstract representation of that system. Architecture is concerned with making sure the software system will meet the requirements of the product, as well as ensuring that future requirements can be addressed. The architecture step also addresses interfaces between the software system and other software products, as well as the underlying hardware or the host operating system.

Coding

Reducing a design to code may be the most obvious part of the software engineering job, but it is not necessarily the largest portion.

Testing

Testing of parts of software, especially where code by two different engineers must work together, falls to the software engineer.

Documentation

An important (and often overlooked) task is documenting the internal design of software for the purpose of future maintenance and enhancement. Documentation is most important for external interfaces.

Two Pass Assembler

Assembler Pseudo code:

Pass 1:

```
BEGIN
 initialize Scnt, Locctr, ENDval, and Errorflag to 0
 WHILE Sourceline[Scnt] is a comment
  BEGIN
   increment Scnt
  END {while}
 Breakup Sourceline[Scnt]
 IF Opcode = 'START' THEN
  BEGIN
   convert Operand from hex and save in Locctr and ENDval
   IF Label not NULL THEN
    Insert (Label, Locctr) into Symtab
   ENDIF
   increment Scnt
   Breakup Sourceline[Scnt]
  END
 ENDIF
```

```
WHILE Opcode <> 'END'
  BEGIN
    IF Sourceline[Scnt] is not a comment THEN
      BEGIN
        IF Label not NULL THEN
          Xsearch Symtab for Label
          IF not found
            Insert (Label, Locctr) into Symtab
          ELSE
            set errors flag in Errors[Scnt]
          ENDIF
        ENDIF
        Xsearch Opcodetab for Opcode
        IF found THEN
          DO CASE
            1. Opcode is 'RESW' or 'RESB'
              BEGIN
                increment Locctr by Storageincr
                IF error THEN
                  set errors flag in Errors[Scnt]
                ENDIF
              END {case 1 (RESW or RESB)}
            2. Opcode is 'WORD' or 'BYTE' THEN
              BEGIN
                increment Locctr by Storageincr
                IF error THEN
                  set errors flag in Errors[Scnt]
          ENDIF
         END {case 2 (WORD or BYTE)}
        3. OTHERWISE
         BEGIN
          increment Locctr by Opcodeincr
          IF error THEN
           set errors flag in Errors[Scnt]
          ENDIF {case 3 (default)}
         END
       ENDCASE
      ELSE
       /* directives such as BASE handled here or */
       set errors flag in Errors[Scnt]
      ENDIF
    END {IF block}
   ENDIF
  increment Scnt
  Breakup Sourceline[Scnt]
END {while}
IF Label not NULL THEN
  Xsearch Symtab for Label
  IF not found
   Insert (Label, Locctr) into Symtab
  ELSE
   set errors flag in Errors[Scnt]
  ENDIF
ENDIF
```

```
 IF Operand not NULL
  Xsearch Symtab for Operand
  IF found
   install in ENDval
  ENDIF
 ENDIF
END {of Pass 1}
```

Pass 2:

```
BEGIN
 initialize Scnt, Locctr, Skip, and Errorflag to 0
 write assembler report headings
 WHILE Sourceline[Scnt] is a comment
  BEGIN
   append to assembler report
   increment Scnt
  END {while}
 Breakup Sourceline[Scnt]
```

```
IF Opcode = 'START' THEN
  BEGIN
    convert Operand from hex and save in Locctr
    append to assembler report
    increment Scnt
    Breakup Sourceline[Scnt]
  END
ENDIF
format and place the load point on object code array
format and place ENDval on object code array, index ENDloc
WHILE Opcode <> 'END'
  BEGIN
    IF Sourceline[Scnt] is not a comment THEN
      BEGIN
        Xsearch Opcodetab for Opcode
        IF found THEN
          DO CASE
            1. Opcode is 'RESW' or 'RESB'
              BEGIN
                increment Locctr by Storageincr
                place '!' on object code array
                replace the value at index ENDloc with loader address
                format and place Locctr on object code array
                format and place ENDval on object code array, index ENDloc
                set Skip to 1
              END
            2. Opcode is 'WORD' or 'BYTE'
              BEGIN
                increment Locctr by Storageincr
                Dostorage to get Objline
                IF error THEN
                  set errors flag in Errors[Scnt]
                ENDIF
              END
            3. OTHERWISE
              BEGIN
                increment Locctr by Opcodeincr
                Doinstruct to get Objline
                IF error THEN
                  set errors flag in Errors[Scnt]
                ENDIF
              END
          ENDCASE
        ELSE
          /* directives such as BASE handled here or */
          set errors flag in Errors[Scnt]
        ENDIF
      END
    ENDIF
    append to assembler report
    IF Errors[Scnt] <> 0 THEN
      BEGIN
        set Errorflag to 1
        append error report to assembler report
```

```
      END
     ENDIF
     IF Errorflag = 0 and Skip = 0 THEN
      BEGIN
        place Objline on object code array
      END
     ENDIF
     IF Skip = 1 THEN
      set Skip to 0
     ENDIF
     increment Scnt
     Breakup Sourceline[Scnt]
   END {while}
  place '!' on object code array
  IF Errorflag = 0 THEN
   transfer object code array to file
  ENDIF
END {of Pass 2}
```

Text Editor

An interactive editor is a computer program that allows a user to create and revise a target document. The document-editing process is an interactive user-computer dialogue designed to accomplish four tasks:

1.Select the part of the target document to be viewed and manipulated.

Selection of the part of the document to be viewed and edited involves first traveling through the document to locate the area of interest. This search is accomplished with operations such as next screen full, bottom, and find pattern. Traveling specifies where the area of interest is. The selection of what is to be viewed and manipulated there is controlled by filtering. Filtering extracts the relevant subset of the target document at the point of interest , such as the next screenful of text or the next statement.

2.Determine how to format this view online and how to display it.

Formatting then determines how the result of the filtering will be seen as a visible representation (the view) on a display screen or other device.

3.Specify and execute operations that modify the target documents.

In the actual editing phase , the target document is created or altered with a set of operations such as insert, delete, replace, move and copy. The editing functions are often specialized to operate on elements meaningful to the type of editor. For example, a manuscript-oriented editor might operate on elements such as single characters, words, lines, sentences and paragraphs.

4.Update the view appropriately.

In a simple scenario, then the user might travel to the end of the document. A screenful of text would be filtered, this segment would be formatted, and the view would be displayed on an output device. The user could then for example, delete the first three words of this view.

Note: The student has to design the Editor with workspace, title bar and menu bar with all the above mentioned characteristics.

Linux Shell

Computer understand the language of 0's and 1's called binary language. In early days of computing, instruction are provided using binary language, which is difficult for all , to read and write. So in Os there is special program called Shell. Shell accepts instruction or commands in English (mostly) and if its a valid command, it is pass to kernel.

Shell is a user program or it's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file. Shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Note : The student has to develop the shell with minimum 25 commands.

4.Lexical Analyzer

Lexical analysis is the processing of an input sequence of characters (such as the source code of a computer program) to produce, as output, a sequence of symbols called "lexical tokens", or just "tokens". For example, lexers for many programming languages convert the character sequence 123 abc into two tokens: 123 and abc (whitespace is not a token in most languages). The purpose of producing these tokens is usually to forward them as input to another program, such as a parser.

A lexical analyzer, or lexer for short, can be thought of having two stages, namely a scanner and an evaluator. (These are often integrated, for efficiency reasons, so they operate in parallel.)

The first stage, the scanner, is usually based on a finite state machine. It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are known as a lexemes). For instance, an integer token may contain any sequence of numerical digit characters. In many cases the first non-whitespace character can be used to deduce the kind of token that follows, the input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is known as the maximal munch rule). In some languages the lexeme creation rules are more complicated and may involve backtracking over previously read characters.

A lexeme, however, is only a string of characters known to be of a certain type. In order to construct a token, the lexical analyzer needs a second stage, the evaluator, which goes over the characters of the lexeme to produce a value. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser. (Some tokens such as parentheses do not really have values, and so the evaluator function for these can return nothing. The evaluators for integers, identifiers, and strings can be considerably more complex. Sometimes evaluators can suppress a lexeme entirely, concealing it from the parser, which is useful for whitespace and comments.)

Note: The student has to develop the LEX for scanning a specific Programming language. Eg. C, C++

References:

1. System Software –An introduction to system programming by Leland L.Beck.

[3rd edition]

2.Principles of Compiler design by Aho-Ullman

3.System programming & Operating system –2nd edition by Dhamdhere

4.Systems Programming by John J.Donavan

5. John R. Levine, Tony Mason and Doug Brown, Lex and Yacc, O’Reilly, SPD, 1999,