



Compiler Practical

Compiler design (Gujarat Technological University)



Scan to open on Studocu

Practical-1

AIM: Implementation of Finite Automata and String Validation.

```
#include<stdio.h>

int main() {
    int length;
    printf("Enter length of a string: ");
    scanf("%d",&length);
    char str[length];
    int i,flag='0';
    printf("Enter a string to be checked: ");
    scanf("%s",str);
    for(i=0;str[i]!='\0';i++) {
        if(str[i]=='\0' || str[i]=='\1') {
            flag=1;
        }
        else {
            flag=0;
            break;
        }
    }
    if(flag==1) {
        if(str[length-1]=='1' && str[length-2]=='\0') {
            printf("String is accepted");\
```

```
}  
else {  
    printf("String is not accepted");  
}  
}  
else{  
    {  
        printf("String is not accepted");  
    }  
}
```

Output:

```
ubuntu@ubuntu:~$ gcc pr1.c
ubuntu@ubuntu:~$ ./a.out
Enter length of a string: 6
Enter a string to be checked: 101001
String is not acceptedubuntu@ubuntu:~$
```

Practical-2

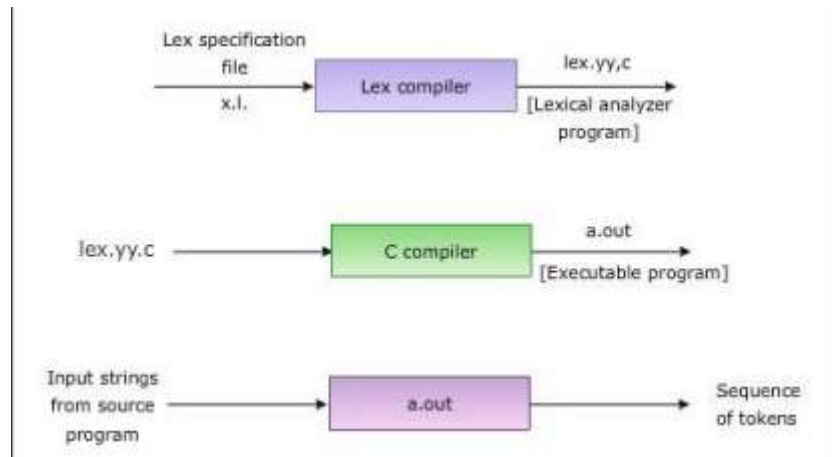
AIM: Introduction to Lex tools

What is Lex?

- Lex is a program that generates lexical analyzer. It is used with 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

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 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 lexical analyzer that transforms an input stream into a sequence of tokens.



The structure of Lex programs

A Lex program is separated into three sections by %% delimiters. A Lex program consists of

three sections : Declarations, Rules and Auxiliary functions. The formal of Lex source is as

follows:

DECLARATIONS

%%

RULES

%%

AUXILIARY FUNCTIONS

Declarations

- The declarations section consists of two parts, auxiliary declarations and regular definitions.

- The auxiliary declarations are copied as such by Lex to the output lex.yy.c file. This C

code consists of instructions to the C compiler and are not processed by the Lex tool.

The auxiliary declarations are written in C language and are enclosed within ' %{ ' and ' %} '. It is generally used to declare functions, include header files, or define global

variables and constants.

- Lex allows the use of short-hands and extensions to regular expressions for the regular definitions.

Example:

```
/*Declarations section start here*/  
/* Auxiliary declarations start here*/  
%{  
    #include <stdio.h>  
    int global_variable;  
}%  
/*Auxiliary declarations end & Regular definitions start  
here*/  
    number [0-9]+ //Regular definition  
    op [-|+|*|/|^|=] //Regular definition  
/*Declarations section ends here*/  
%%  
  
/* Rules */  
%%  
  
/* Auxiliary functions */
```

Rules

- Rules in a Lex program consist of two parts :

1. The pattern to be matched
 2. The corresponding action to be executed
- The pattern to be matched is specified as a regular expression.

Example:

```
/* Declarations */  
  
%%  
  
{number} {printf(" number");}  
  
{op} {printf(" operator");}  
  
%%  
  
/* Auxiliary functions */
```

Auxiliary functions

- Lex generates C code for the rules specified in the Rules section and places this code into

a single function called `yylex()`. (To be discussed in detail later). In addition to this Lex

generated code, the programmer may wish to add his own code to the `lex.yy.c` file. The

auxiliary functions section allows the programmer to achieve this.

Example:

```
/* Declarations */  
  
%%  
  
/* Rules */  
  
%%  
  
int main()  
{
```



```
yylex();  
return 1;  
}
```

The yyvariables

The following variables are offered by Lex to aid the programmer in designing sophisticated

lexical analyzers. These variables are accessible in the Lex program and are automatically

declared by Lex in lex.yy.c.

1. yyin is a variable of the type FILE* and points to the input file. yyin is defined by Lex

automatically. If the programmer assigns an input file to yyin in the auxiliary functions

section, then yyin is set to point to that file. Otherwise Lex assigns yyin to stdin(console input).

2. yytext is of type char* and it contains the lexeme currently found. A lexeme is a sequence

of characters in the input stream that matches some pattern in the Rules Section. (In fact, it

is the first matching sequence in the input from the position pointed to by yyin.) Each

invocation of the function yylex() results in yytext carrying a pointer to the lexeme found in

the input stream by yylex(). The value of yytext will be overwritten after the next yylex()

invocation.

3. `yyleng` is a variable of the type `int` and it stores the length of the lexeme pointed to by `yytext`.

The **yyfunctions**

1. `yylex()`

- `yylex()` is a function of return type `int`. Lex automatically defines `yylex()` in `lex.yy.c` but

does not call it. The programmer must call `yylex()` in the Auxiliary functions section of

the Lex program. Lex generates code for the definition of `yylex()` according to the rules

specified in the Rules section.

- `yylex()` need not necessarily be invoked in the Auxiliary Functions Section of Lex

program when used with YACC.

2. `yywrap()`

- Lex declares the function `yywrap()` of return-type `int` in the file `lex.yy.c`. Lex does not

provide any definition for `yywrap()`. `yylex()` makes a call to `yywrap()` when it encounters

the end of input.

- If `yywrap()` returns zero (indicating false) `yylex()` assumes there is more input and it

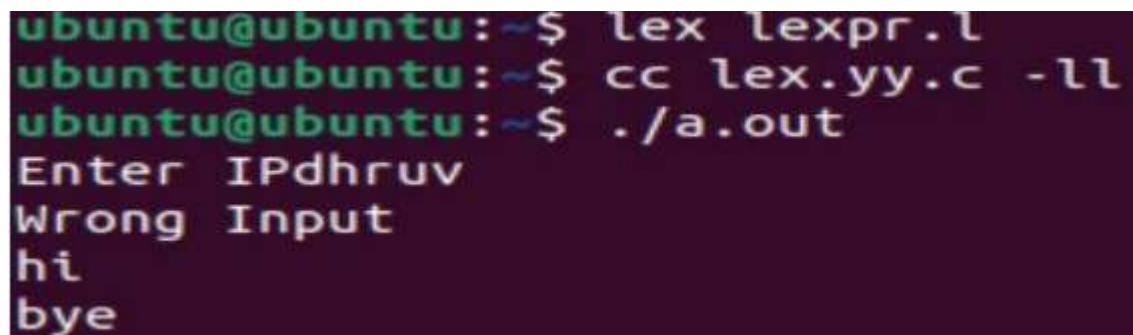
continues scanning from the location pointed to by `yyin`. If `yywrap()` returns a non-zero

value (indicating true), `yylex()` terminates the scanning process and returns 0.

Example of Lex Tool

```
%{  
#include<stdio.h>  
%}  
%%  
"hi" {printf("bye");}  
.* {printf("Wrong Input");}  
%%  
void main()  
{  
printf("Enter IP");  
yylex();  
}
```

Output:



```
ubuntu@ubuntu:~$ lex lexpr.l  
ubuntu@ubuntu:~$ cc lex.yy.c -ll  
ubuntu@ubuntu:~$ ./a.out  
Enter IPdhruv  
Wrong Input  
hi  
bye
```

Practical-3

AIM: Implement following Programs Using Lex.

(a) Generate Count of words

Code:

```
%{  
#include<stdio.h>  
#include<string.h>  
  
int i = 0;  
%}  
%%  
([a-zA-Z0-9])* {i++;}  
"\n" {printf("%d\n", i); i = 0;}  
%%  
int yywrap(void){}  
int main()  
{  
    yylex();  
    return 0;  
}
```

output:

```
ubuntu@ubuntu:~$ gedit pr3a.l
^C
ubuntu@ubuntu:~$ lex pr3a.l
ubuntu@ubuntu:~$ cc lex.yy.c -ll
ubuntu@ubuntu:~$ ./a.out
i am dhruv moradiya
4
```

(b) Ceasor Cypher

Code:

```
%{  
#include<stdio.h>  
%}  
%%  
[a-z] {char ch = yytext[0];  
ch += 3;  
if (ch> 'z') ch -= ('z'+1- 'a');  
printf ("%c" ,ch );  
}  
[A-Z] { char ch = yytext[0] ;  
ch += 3;  
if (ch> 'Z') ch -= ('Z'+1- 'A');  
printf("%c",ch);  
}  
%%  
void main()  
{  
printf("Enter Input Message: ");  
yylex();  
}
```

Output:

```
ubuntu@ubuntu:~$ lex pr3b.l
ubuntu@ubuntu:~$ cc lex.yy.c -ll
ubuntu@ubuntu:~$ ./a.out
Enter Input Message: Dhruv Moradiya
Gkuxy Prudglbd
```

(c) Extract single and multiline comments from C Program.

Input file:

```
#include<stdio.h>

void main()
{
//Single-line comment
/*Multiline
comment
*/
printf("Dhruv Moradiya");
}
```

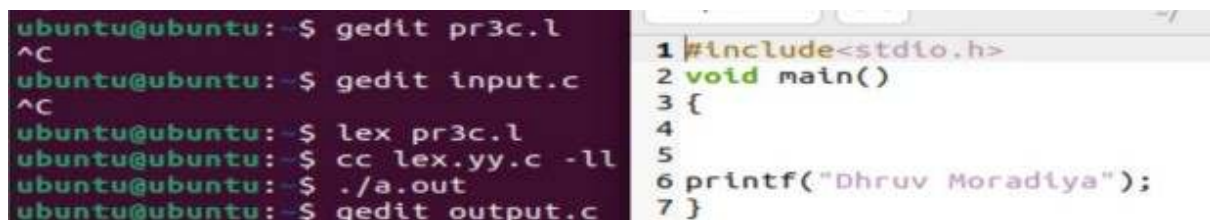
Code

```
%{
#include<stdio.h>
%}
%%
\\(.*) {};
\\*(.*\\n)*.*\\*V {};
%%
int yywrap()
{
return 1;
}
```



```
void main()
{
yyin=fopen("input.c","r");
yyout=fopen("output.c","w");
yylex();
}
```

Output :



The image shows a terminal window on the left and a code editor on the right. The terminal window displays the following commands and their outputs:

```
ubuntu@ubuntu:~$ gedit pr3c.l
^C
ubuntu@ubuntu:~$ gedit input.c
^C
ubuntu@ubuntu:~$ lex pr3c.l
ubuntu@ubuntu:~$ cc lex.yy.c -ll
ubuntu@ubuntu:~$ ./a.out
ubuntu@ubuntu:~$ gedit output.c
```

The code editor on the right shows the following C code:

```
1 #include<stdio.h>
2 void main()
3 {
4
5
6 printf("Dhruv Moradiya");
7 }
```

Practical-4

AIM: Implement following Programs Using Lex.

(a) Convert Roman to Decimal

Code:

```
WS [ \t ]+
```

```
%{
```

```
int total=0;
```

```
%}
```

```
%%
```

```
I total += 1;
```

```
IV total += 4;
```

```
V total += 5;
```

```
IX total += 9;
```

```
X total += 10;
```

```
XL total += 40;
```

```
L total += 50;
```

```
XC total += 90;
```

```
C total += 100;
```

```
CD total += 400;
```

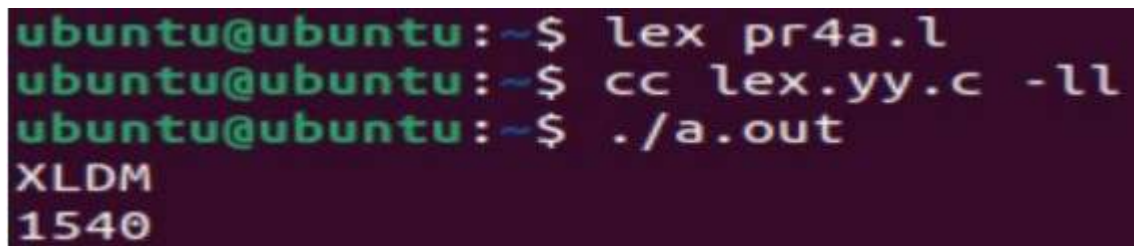
```
D total += 500;
```

```
CM total += 900;
```

```
M total += 1000;
```

```
{WS} |  
\n return total;  
%%  
int main (void) {  
int number;  
number = yylex ();  
printf ("%d\n", number);  
return 0;  
}
```

Output:



```
ubuntu@ubuntu:~$ lex pr4a.l  
ubuntu@ubuntu:~$ cc lex.yy.c -ll  
ubuntu@ubuntu:~$ ./a.out  
XLDM  
1540
```

(b) Check whether given statement is compound or simple.

Code:

```
%{  
#include<stdio.h>  
  
int flag=0;  
%}  
%%  
and |  
or |  
but |  
because |  
if |  
then |  
nevertheless { flag=1; }  
.;  
\n { return 0; }  
%%  
  
int main()  
{  
printf("Enter your sentence:\n");  
yylex();  
if(flag==0)  
printf("Simple sentence\n");
```

else

printf("compound sentence\n");

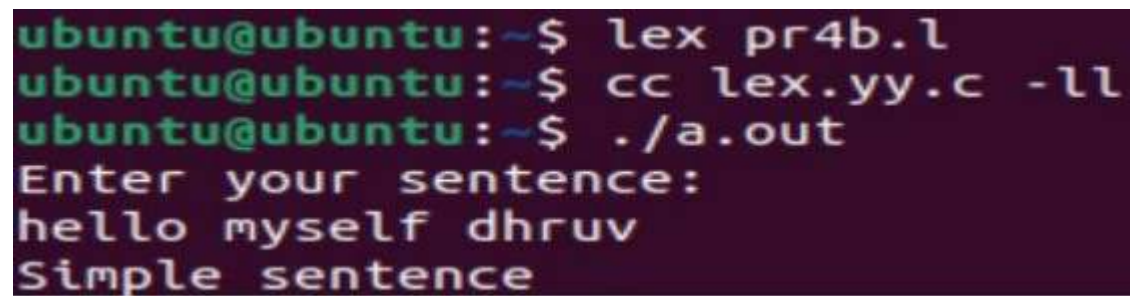
}

int yywrap()

{

return 1;

output:



```
ubuntu@ubuntu:~$ lex pr4b.l
ubuntu@ubuntu:~$ cc lex.yy.c -ll
ubuntu@ubuntu:~$ ./a.out
Enter your sentence:
hello myself dhruv
Simple sentence
```

(c) Extract html tags from .html file.

39.html:

```
<html>
<head>
</head>
<body>
<p>
<a href="https://www.coursera.org/">Dhruv Moradiya</a>
</p>
</body>
</html>
```

Code:

```
%{
#include<stdio.h>
%}
%%
"<"[^>]*> {printf("%s\n", yytext); }
.;
%%
int yywrap(){}
int main(int argc, char*argv[])
{
yyin = fopen("39.html","r");
```

```
yylex();  
return 0;  
}
```

Output:

```
ubuntu@ubuntu:~$ gedit pr4c.l  
ubuntu@ubuntu:~$ lex pr4c.l  
ubuntu@ubuntu:~$ cc lex.yy.c -ll  
ubuntu@ubuntu:~$ ./a.out  
<html>  
  
<head>  
  
</head>  
  
<body>  
  
<p>  
  
<a href="https://www.coursera.org/">  
Dhruv Moradiya</a>  
  
</p>  
  
</body>  
  
</html>
```

Practical-5

AIM: Implementation of Recursive Descent Parser without backtracking.

Input: The string to be parsed.

Output: Whether string parsed successfully or not.

Explanation: Students have to implement the recursive procedure for

RDP for a typical grammar. The production no. are displayed as they

are used to derive the string.

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

```
char str[20];
```

```
int p=0;
```

```
char l;
```

```
int flag = 0;
```

```
void getString(){
```

```
printf("The Production rules are as follows: - \nE -> iE'\nE' ->  
+tE/^\\nEnter a String : ");
```

```
scanf("%[^\\n]%"c", str);
```

```
}
```

```
void error(){
```

```
flag = 0;
```



```

}
int match(char t){
if(l == '$'){
flag = 1;
return 0;
}
if(l == t){
p++;
l = str[p];
}
else{
error();
return 0;
}
}
void Edesh(){
if(l == '+'){
match('+');
match('i');
E();
}
else{
error();
}
}

```

```
}  
void E(){  
    l = str[p];  
    match('i');  
    if(l != '$'){  
        Edesh();  
    }  
}  
  
void main(){  
    getString();  
    E();  
    match('$');  
    if(flag == 1){  
        printf("ACCEPTED\n");  
    }else{  
        printf("NOT ACCEPTED\n");  
    }  
}
```

Output:

```
ubuntu@ubuntu:~$ ./a.out
The Production rules are as follows: -
E -> iE'
E' -> +tE/^
Enter a String : 
```

Practical-6

Aim: Finding “First” set Input: The string consists of grammar symbols.

Output: The First set for a given string.

```
For grammar:  
S -> A  
A -> aB | d  
B -> bBC | f  
C -> g
```

```
#include<stdio.h>  
#include<string.h>  
char first[100];  
char c;  
void FiS()  
{  
    FiA();  
}  
void FiA()  
{  
    c = '{';  
    strncat(first,&c,1);  
    c = 'a';  
    strncat(first,&c,1);
```

```
c = ',';
strncat(first,&c,1);
c = ' ';
strncat(first,&c,1);
c = 'd';
strncat(first,&c,1);
c = '}';
strncat(first,&c,1);
}
void FiB()
{
c = '{';
strncat(first,&c,1);
c = 'b';
strncat(first,&c,1);
c = ',';
strncat(first,&c,1);
c = ' ';
strncat(first,&c,1);
c = 'f';
strncat(first,&c,1);
c = '}';
strncat(first,&c,1);
}
```

```

void FiC()
{
    c = '{';
    strncat(first,&c,1);
    c = 'g';
    strncat(first,&c,1);
    c = '}';
    strncat(first,&c,1);
}

void main()
{
    char r;
    printf("Enter your Non Terminal:\n");
    scanf("%c",&r);
    if(r=='S')
    {
        FiS();
    }
    else if(r=='A')
    {
        FiA();
    }
    else if(r=='B')
    {

```

```

FiB();
}
else if(r=='C')
{
FiC();
}
else
{
printf("INVALID NON TERMINAL");
}
printf("First of \'%c\' is:",r);
puts(first);
}

```

Output:

```

ubuntu@ubuntu:~$ ./a.out
Enter your Non Terminal:
S
First of 'S' is:{a, d}
ubuntu@ubuntu:~$ ./a.out
Enter your Non Terminal:
B
First of 'B' is:{b, f}
ubuntu@ubuntu:~$ ./a.out
Enter your Non Terminal:
A
First of 'A' is:{a, d}
ubuntu@ubuntu:~$ ./a.out
Enter your Non Terminal:
C
First of 'C' is:{g}

```

Practical-7

Aim: Generate 3-tuple intermediate code for given infix expression

```
#include<stdio.h>
#include<string.h>
void pm();
void plus();
void div();
int i,ch,j,l,addr=100;
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
void main()
{
printf("\nEnter the expression with arithmetic operator:");
scanf("%s",ex);
strcpy(exp,ex);
l=strlen(exp);
exp1[0]='\0';
for(i=0;i<l;i++)
{
if(exp[i]=='+' | exp[i]=='-')
{
if(exp[i+2]=='/' | exp[i+2]=='*')
{
```



```

pm();
break;
}
else
{
plus();
break;
}
}
else if(exp[i]=='/' || exp[i]=='*')
{
div();
break;
}
}
}
void pm()
{
strrev(exp);
j=l-i-1;
strncat(exp1,exp,j);
strrev(exp1);
printf("Three address code:\ntemp=%s\ntemp1=%c
%ctemp\n",exp1,exp[j+1],exp[j]);

```

```

}

void div()

{
    strncat(exp1,exp,i+2);

    printf("Three address code:\ntemp=%s\ntemp1=temp%c\n",exp1,exp[i+2],exp[i+3]);
}

void plus()

{
    strncat(exp1,exp,i+2);

    printf("Three address code:\ntemp=%s\ntemp1=temp%c\n",exp1,exp[i+2],exp[i+3]);
}

```

Output:

```
C:\CD Practicals\CD_P_7>gcc CD_P_7.c
CD_P_7.c:7:14: warning: built-in function 'exp' declared as non-function
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
      ^~~~~
CD_P_7.c:7:32: warning: built-in function 'exp2' declared as non-function
char ex[10], exp[10] ,exp1[10],exp2[10],id1[5],op[5],id2[5];
      ^~~~~~

C:\CD Practicals\CD_P_7>a

Enter the expression with arithmetic operator:a+b=c
Three address code:
temp=a+b
temp1=temp+c

C:\CD Practicals\CD_P_7>a

Enter the expression with arithmetic operator:a*b+c
Three address code:
temp=a*b
temp1=temp+c
```

Practical-8

Aim: Extract Predecessor and Successor from given
Control Flow Graph

```
package cd_p_8;
class CD_P_8{
    static class Node
    {
        int key;
        Node left, right;
        public Node()
        {}
        public Node(int key)
        {
            this.key = key;
            this.left = this.right = null;
        }
    };
    static Node pre = new Node(), suc = new Node();
    static void findPreSuc(Node root, int key)
    {
        if (root == null)
            return;
        if (root.key == key)
```

```
{
if (root.left != null)
{
Node tmp = root.left;
while (tmp.right != null)
tmp = tmp.right;
pre = tmp;
}
if (root.right != null)
{
Node tmp = root.right;
while (tmp.left != null)
tmp = tmp.left;
suc = tmp;
}
return;
}
if (root.key > key)
{
suc = root;
findPreSuc(root.left, key);
}
else
{

```

```

pre = root;
findPreSuc(root.right, key);
}
}

static Node insert(Node node, int key)
{
    if (node == null)
        return new Node(key);
    if (key < node.key)
        node.left = insert(node.left, key);
    else
        node.right = insert(node.right, key);
    return node;
}

public static void main(String[] args)
{
    int key = 32;
    Node root = new Node();
    root = insert(root, 10);
    insert(root, 70);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);
    insert(root, 90);
}

```

```
insert(root, 20);
findPreSuc(root, key);
if (pre != null)
    System.out.println("Predecessor is " + pre.key);
else
    System.out.println("No Predecessor");
if (suc != null)
    System.out.println("Successor is " + suc.key);
else
    System.out.println("No Successor");
}
}
```

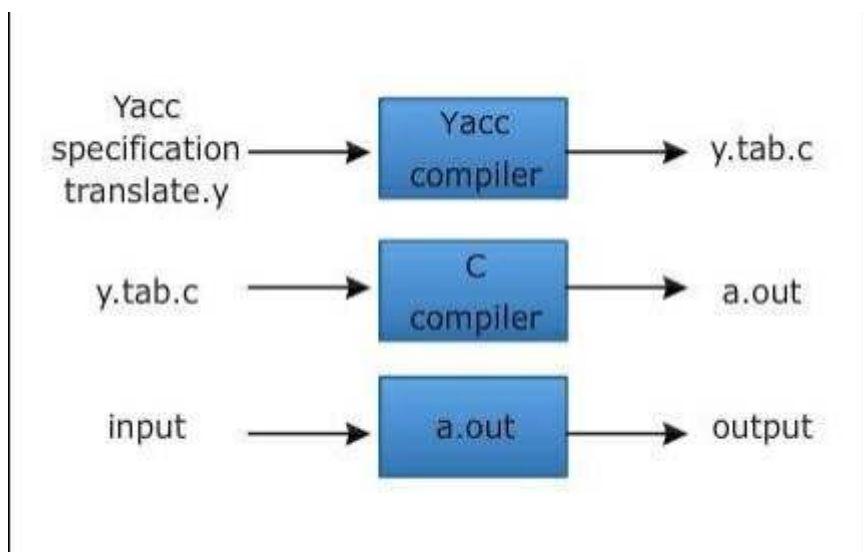
Output:

```
java -cp /tmp/9JSC4SYEH0 P8
Predecessor is 30Successor is 40
```

Practical-9

Aim: Introduction to YACC and generate Calculator Program

1. YACC (Yet Another Compiler Compiler) is a tool used to generate a parser.
2. This document is a tutorial for the use of YACC to generate a parser for ExpL.
3. 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).
4. This C program when compiled, yields an executable parser.



How yacc works?

- The input to yacc describes the rules of a grammar.
- yacc uses these rules to produce the source code for a program that parses the grammar.

- You can then compile this source code to obtain a program that reads input, parses it according to the grammar, and takes action based on the result.
- The source code produced by yacc is written in the C programming language.
- It consists of a number of data tables that represent the grammar, plus a C function named `yyparse()`.
- By default, yacc symbol names used begin with `yy`.
- This is an historical convention, dating back to yacc's predecessor, UNIX yacc.
- You can avoid conflicts with yacc names by avoiding symbols that start with `yy`.

The structure of YACC programs:

- A YACC program consists of three sections: Declarations, Rules and Auxiliary functions.

(Note the similarity with the structure of LEX programs).

```
DECLARATIONS
```

```
%%
```

```
RULES
```

```
%%
```

```
AUXILIARY FUNCTIONS
```

CD_P_9.l

```
%{  
/* 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;  
}
```

CD_P_9.y

```

%{
/* Definition section */
#include<stdio.h>

int flag=0;

%}

%token NUMBER

%left '+' '-'
%left '*' '/' '%'
%left '(' ')'

/* Rule Section */

%%

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;};

%%

//driver code

```

```
void main()
{
printf("\nEnter Any Arithmetic Expression which can have operations
Addition, Subtraction,
Multiplication, Division, Modulus and Round brackets:\n");
yyvsparse();
if(flag==0)
printf("\nEnter arithmetic expression is Valid\n\n");
}
void yyerror()
{
printf("\nEnter arithmetic expression is Invalid\n\n");
flag=1;
}
```

Output:

```

ubuntu@ubuntu:~$ flex pr9.l
ubuntu@ubuntu:~$ bison -dy pr9.y
pr9.y:27 parser name defined to default : "parse"
ubuntu@ubuntu:~$ gcc lex.yy.c y.tab.c
/usr/share/bison+/bison.cc:198:24: warning: implicit declaration of function 'yyerror'; did you mean 'yyerror'? [-Wimplicit-function-declaration]
198 | #define YY_ERROR yyerror
    |                        ^~~~~
/usr/share/bison+/bison.cc:667:11: note: in expansion of macro 'YY_parse_ERROR'
667 |     YY_ERROR("parser stack overflow");
    |     ^~~~~~
/usr/share/bison+/bison.cc:180:22: warning: implicit declaration of function 'yylex' [-Wimplicit-function-declaration]
180 | #define YY_LEX yylex
    |                ^~~~~
/usr/share/bison+/bison.cc:465:25: note: in expansion of macro 'YY_parse_LEX'
465 | #define YYLEX      YY_LEX()
    |                ^~~~~~
/usr/share/bison+/bison.cc:730:23: note: in expansion of macro 'YYLEX'
730 |     YY_CHAR = YYLEX;
    |             ^~~~~

```

```

ubuntu@ubuntu:~$ a.exe
a.exe: command not found
ubuntu@ubuntu:~$ ./a.out

Enter Any Arithmetic Expression which can have operations Addition, Subtraction, Multiplication, Division, Modulus and Round brackets:
2+8

Result=10

Entered arithmetic expression is Valid

```

Practical-10

Aim: Finding “Follow” set Input: The string consists of grammar symbols.

Output: The Follow set for a given string.

```

For grammar:
S -> A
A -> aB | d
B -> bBC | f
C -> g

```

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

```
char follow[100];
```

```
char c;
```

```
void FoS()
```

```
{
```

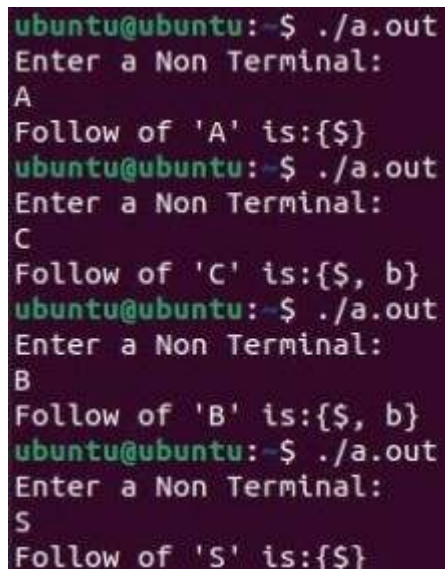
```
c = '{';
```

```
strncat(follow,&c,1);
c = '$';
strncat(follow,&c,1);
}
void FoA()
{
FoS();
}
void FoB()
{
FoA();
FiC();
}
void FoC()
{
FoB();
}
void FiC()
{
c = ',';
strncat(follow,&c,1);
c = ' ';
strncat(follow,&c,1);
c = 'b';
```

```
strncat(follow,&c,1);  
}  
void main()  
{  
char r;  
printf("Enter your Non Terminal:\n");  
scanf("%c",&r);  
if(r=='S')  
{  
FoS();  
}  
else if(r=='A')  
{  
FoA();  
}  
else if(r=='B')  
{  
FoB();  
}  
else if(r=='C')  
{  
FoC();  
}  
else
```

```
{  
printf("INVALID NON TERMINAL");  
}  
c = '}';  
strncat(follow,&c,1);  
printf("Follow of \'%c\' is:",r);  
puts(follow);  
}
```

Output:



```
ubuntu@ubuntu:~$ ./a.out  
Enter a Non Terminal:  
A  
Follow of 'A' is:{$}  
ubuntu@ubuntu:~$ ./a.out  
Enter a Non Terminal:  
C  
Follow of 'C' is:{$, b}  
ubuntu@ubuntu:~$ ./a.out  
Enter a Non Terminal:  
B  
Follow of 'B' is:{$, b}  
ubuntu@ubuntu:~$ ./a.out  
Enter a Non Terminal:  
S  
Follow of 'S' is:{$}
```


Practical-11

Aim: Implement a C program for constructing LL (1) parsing.

```
For grammar:  
S -> AaAb | BbBa  
A -> E  
B -> E
```

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

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

```
int stackPointer = 0;
```

```
int inputPointer = 0;
```

```
int stackLength = 0;
```

```
char stack[20], input[10];
```

```
int LL1Parse()
```

```
{
```

```
int s = 0;
if(stack[stackPointer]=='S')
{
if(input[inputPointer]=='a')
{
stack[stackPointer] = 'b';
stackPointer++;
stack[stackPointer] = 'A';
stackPointer++;
stack[stackPointer] = 'a';
stackPointer++;
stack[stackPointer] = 'A';
}
else if(input[inputPointer]=='b')
{
stack[stackPointer] = 'a';
stackPointer++;
stack[stackPointer] = 'B';
stackPointer++;
stack[stackPointer] = 'b';
stackPointer++;
stack[stackPointer] = 'B';
}
}
```

```
else if(stack[stackPointer]=='A')
{
if(input[inputPointer]=='a')
{
stack[stackPointer] = 'x';
stackPointer--;
}
else if(input[inputPointer]=='b')
{
stack[stackPointer] = 'x';
stackPointer--;
}
}
else if(stack[stackPointer]=='B')
{
if(input[inputPointer]=='a')
{
stack[stackPointer] = 'x';
stackPointer--;
}
else if(input[inputPointer]=='b')
{
stack[stackPointer] = 'x';
stackPointer--;
```

```

}
}
else if(stack[stackPointer]=='a' && input[inputPointer]=='a')
{
stack[stackPointer] = 'x';
stackPointer--;
inputPointer++;
}
else if(stack[stackPointer]=='b' && input[inputPointer]=='b')
{
stack[stackPointer] = 'x';
stackPointer--;
inputPointer++;
}
else if(stack[stackPointer]=='b' && input[inputPointer]=='a')
{
printf("Input rejected");
s = 2;
}
else if(stack[stackPointer]=='a' && input[inputPointer]=='b')
{
printf("Input rejected");
s = 2;
}
}

```

```
else if(input[inputPointer]=='$')
{
if(stack[stackPointer]=='$')
{
printf("Input accepted");
s = 1;
}
else
{
printf("Input rejected");
s = 2;
}
}
else if(stack[stackPointer]=='$')
{
if(input[inputPointer]=='$')
{
printf("Input accepted");
s = 1;
}
else
{
printf("Input rejected");
s = 2;
```

```

}
}
return s;
}
void main()
{
int i,j,status=0;
stack[stackPointer] = '$';
stackPointer++;
stack[stackPointer] = 'S';
printf("Enter your string: ");
gets(input);
input[strlen(input)] = '$';
while(status!=1)
{
status = LL1Parse();
if(status==2)
break;
}
}

```

Output:

```

ubuntu@ubuntu:~$ gcc pr11.c
pr11.c: In function 'main':
pr11.c:118:1: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
  118 |   gets(input);
      |   ^~~~~
      |   fgets
/usr/bin/ld: /tmp/cces9BR8.o: in function 'main':
pr11.c:(.text+8x538): warning: the 'gets' function is dangerous and should not be used.
ubuntu@ubuntu:~$ ./a.out
Enter string: aa
Rejectedubuntu@ubuntu:~$ ./a.out
Enter string: bbba
Rejectedubuntu@ubuntu:~$ ./a.out
Enter string: ba

```

Practical-12

Aim: Implement a C program for constructing LALR parsing.

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

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

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

```
#include<conio.h>
```

```
void push(char *, int *, char);
```

```
char stacktop(char *);
```

```
void isproduct(char, char);
```

```
int ister(char);
```

```
int isnter(char);
```

```

int isstate(char);
void error();
void isreduce(char, char);
char pop(char *, int *);
void printt(char *, int *, char[], int);
void rep(char[], int);
struct action{
    char row[6][5];
};
const struct action A[12] = {
    {"sf", "emp", "emp", "se", "emp", "emp"},
    {"emp", "sg", "emp", "emp", "emp", "acc"},
    {"emp", "rc", "sh", "emp", "rc", "rc"},
    {"emp", "re", "re", "emp", "re", "re"},
    {"sf", "emp", "emp", "se", "emp", "emp"},
    {"emp", "rg", "rg", "emp", "rg", "rg"},
    {"sf", "emp", "emp", "se", "emp", "emp"},
    {"sf", "emp", "emp", "se", "emp", "emp"},
    {"emp", "sg", "emp", "emp", "sl", "emp"},
    {"emp", "rb", "sh", "emp", "rb", "rb"},
    {"emp", "rb", "rd", "emp", "rd", "rd"},
    {"emp", "rf", "rf", "emp", "rf", "rf"}
};
struct goto{

```



```

char r[3][4];

};

const struct goto1 G[12] = {
    {"b", "c", "d"},
    {"emp", "emp", "emp"},
    {"emp", "emp", "emp"},
    {"emp", "emp", "emp"},
    {"i", "c", "d"},
    {"emp", "emp", "emp"},
    {"emp", "j", "d"},
    {"emp", "emp", "k"},
    {"emp", "emp", "emp"},
    {"emp", "emp", "emp"},
};

char ter[6] = {'i', '+', '*', ')', '(', '$'};

char nter[3] = {'E', 'T', 'F'};

char states[12] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'm', 'j', 'k', 'l'};

char stack[100];

int top = -1;

char temp[10];

struct grammar{
    char left;
    char right[5];
};

```

```

const struct grammar rl[6] = {
    {'E', "e+T"}, {'E', "T"}, {'T', "T*F"}, {'T', "F"}, {'F', "(E)"}, {'F', "i"},
};

void main(){
    char inp[80], x, p, dl[80], y, bl = 'a';
    int i = 0, j, k, l, n, m, c, len;
    printf(" Enter the input :");
    scanf("%s", inp);
    len = strlen(inp);
    inp[len] = '$';
    inp[len + 1] = '\0';
    push(stack, &top, bl);
    printf("\n stack \t\t\t input");
    printt(stack, &top, inp, i);
    do{
        x = inp[i];
        p = stacktop(stack);
        isproduct(x, p);
        if (strcmp(temp, "emp") == 0)
            error();
        if (strcmp(temp, "acc") == 0)
            break;
        else{
            if (temp[0] == 's') {

```

```

push(stack, &top, inp[i]);
push(stack, &top, temp[1]);

i++;
}

else{
if (temp[0] == 'r'){
j = isstate(temp[1]);
strcpy(temp, rl[j - 2].right);
dl[0] = rl[j - 2].left;
dl[1] = '\0';
n = strlen(temp);
for (k = 0; k < 2 * n; k++)
pop(stack, &top);
for (m = 0; dl[m] != '\0'; m++)
push(stack, &top, dl[m]);
l = top;
y = stack[l - 1];
isreduce(y, dl[0]);
for (m = 0; temp[m] != '\0'; m++)
push(stack, &top, temp[m]);
}
}
}

printt(stack, &top, inp, i);

```

```

} while (inp[i] != '\0');
if (strcmp(temp, "acc") == 0)
printf(" \n accept the input ");
else
printf(" \n do not accept the input ");
getch();
}

void push(char *s, int *sp, char item) {
if (*sp == 100)
printf(" stack is full ");
else{
*sp = *sp + 1;
s[*sp] = item;
}
}

char stacktop(char *s) {
char i;
i = s[top];
return i;
}

void isproduct(char x, char p) {
int k, l;
k = ister(x);
l = isstate(p);

```

```

strcpy(temp, A[l - 1].row[k - 1]);
}

int ister(char x) {
    int i;
    for (i = 0; i < 6; i++)
        if (x == ter[i])
            return i + 1;
    return 0;
}

int isnter(char x) {
    int i;
    for (i = 0; i < 3; i++)
        if (x == nter[i])
            return i + 1;
    return 0;
}

int isstate(char p) {
    int i;
    for (i = 0; i < 12; i++)
        if (p == states[i])
            return i + 1;
    return 0;
}

void error(){

```

```

printf(" error in the input ");
exit(0);
}
void isreduce(char x, char p) {
    int k, l;
    k = isstate(x);
    l = isnter(p);
    strcpy(temp, G[k - 1].r[l - 1]);
}
char pop(char *s, int *sp) {
    char item;
    if (*sp == -1)
        printf(" stack is empty ");
    else{
        item = s[*sp];
        *sp = *sp - 1;
    }
    return item;
}
void printt(char *t, int *p, char inp[], int i) {
    int r;
    printf("\n");
    for (r = 0; r <= *p; r++)
        rep(t, r);
}

```

```
printf("\t\t\t");  
for (r = i; inp[r] != '\0'; r++)  
    printf("%c", inp[r]);  
}  
void rep(char t[], int r) {  
    char c;  
    c = t[r];  
    switch (c) {  
        case 'a':  
            printf("0");  
            break;  
        case 'b':  
            printf("1");  
            break;  
        case 'c':  
            printf("2");  
            break;  
        case 'd':  
            printf("3");  
            break;  
        case 'e':  
            printf("4");  
            break;  
        case 'f':
```

```
printf("5");  
break;  
case 'g':  
printf("6");  
break;  
case 'h':  
printf("7");  
break;  
case 'm':  
printf("8");  
break;  
case 'j':  
printf("9");  
break;  
case 'k':  
printf("10");  
break;  
case 'l':  
printf("11");  
break;  
default:  
printf("%c", t[r]);  
break;  
}
```


}

Output:

```
ubuntu@ubuntu:~$ gcc pr12.c
ubuntu@ubuntu:~$ ./a.out
Enter the input :i*i+i

stack          input
0              i*i+i$
0i5            *i+i$
0F3            *i+i$
0T2            *i+i$
0T2*7          i+i$
0T2*7i5        +i$
0T2*7F10       +i$
0E1            +i$
0E1+6          i$
0E1+6i5        $
0E1+6F3        $
0E1+6T9        $
0E1            $
accept the input ubuntu@ubuntu:~$
```

Practical-13

Aim: Implement a C program to implement operator precedence parsing.

```
#include<stdio.h>

#include<string.h>

#include<stdlib.h>

char *input;

int i=0;

char lasthandle[6],stack[50],handles[]
[5]={"")E(", "E*E", "E+E", "i", "E^E"};

//(E) becomes )E( when pushed to stack

int top=0,l;
```

```

char prec[9][9]={
    /*input*/
    /*stack + - * / ^ i ( ) $ */
    /* + */ '>','>','<','<','<','<','<','>','>',
    /* - */ '>','>','<','<','<','<','<','>','>',
    /* * */ '>','>','>','>','<','<','<','>','>',
    /* / */ '>','>','>','>','<','<','<','>','>',
    /* ^ */ '>','>','>','>','<','<','<','>','>',
    /* i */ '>','>','>','>','>','>','e','e','>','>',
    /* ( */ '<','<','<','<','<','<','<','>','e',
    /* ) */ '>','>','>','>','>','>','e','e','>','>',
    /* $ */ '<','<','<','<','<','<','<','<','>',
};

```

```

int getindex(char c)

```

```

{
    switch(c)
    {
        case '+':return 0;
        case '-':return 1;
        case '*':return 2;
        case '/':return 3;
        case '^':return 4;
        case 'i':return 5;
        case '(':return 6;
    }
}

```

```

case ')':return 7;
case '$':return 8;
}
}
int shift()
{
stack[++top]=*(input+i++);
stack[top+1]='\0';
}
int reduce()
{
int i,len,found,t;
for(i=0;i<5;i++)//selecting handles
{
len=strlen(handles[i]);
if(stack[top]==handles[i][0]&&top+1>=len)
{
found=1;
for(t=0;t<len;t++)
{
if(stack[top-t]!=handles[i][t])
{
found=0;
break;

```

```

}
}
if(found==1)
{
    stack[top-t+1]='E';
    top=top-t+1;
    strcpy(lasthandle,handles[i]);
    stack[top+1]='\0';
    return 1;//successful reduction
}
}
}
return 0;
}

void dispstack()
{
    int j;
    for(j=0;j<=top;j++)
        printf("%c",stack[j]);
}

void dispinput()
{
    int j;
    for(j=i;j<l;j++)

```

```

    printf("%c",*(input+j));
}

void main()
{
    int j;
    input=(char*)malloc(50*sizeof(char));
    printf("\nEnter the string\n");
    scanf("%s",input);
    input=strcat(input,"$");
    l=strlen(input);
    strcpy(stack,"$");
    printf("\nSTACK\tINPUT\tACTION");
    while(i<=l)
    {
        shift();
        printf("\n");
        dispstack();
        printf("\t");
        dispinput();
        printf("\tShift");
        if(prec[getIndex(stack[top])][getIndex(input[i])]=='>')
        {
            while(reduce())
            {

```

```

printf("\n");
dispstack();
printf("\t");
dispinput();
printf("\tReduced: E->%s",lasthandle);
}
}
}
if(strcmp(stack,"$E$")==0)
    printf("\nAccepted;");
else
    printf("\nNot Accepted;");}

```

output:

```

Enter the string
i+i

STACK   INPUT   ACTION
$i      +i$      Shift
$E      +i$      Reduced: E->i
$E+     i$      Shift
$E+i    $       Shift
$E+E    $       Reduced: E->i
$E      $       Reduced: E->E+E
$E$     $       Shift
$E$     $       Shift
Accepted;ubuntu@ubuntu:~$

```

