Name : Rishabh Singh Sahil
Register No. : RA2011003011334
Course Code : 18CSC304J
Course name : Compiler Design
Date of Submission : 14 May 2023

TABLE OF CONTENTS

1. ASSIGNMENTS

1.1 UNIT 1 CROSSWORD

1.2 UNIT 2 CROSSWORD

1.3 UNIT 3 CROSSWORD

1.4 UNIT 1 ASSIGNMENT

2. LAB EXPERIMENTS

2.1 EXPERIMENT 1 - LEXICAL ANALYSER

2.2 EXPERIMENT 2 - RE TO NFA

2.3 EXPERIMENT 3 - NFA TO DFA CONVERSION

2.4 EXPERIMENT 4 - LEFT RECURSION & LEFT FACTORING

2.5 EXPERIMENT 5 - FIRST AND FOLLOW

2.6 EXPERIMENT 6 - PREDICTIVE PARSING

2.7 EXPERIMENT 7 - SHIFT REDUCE PARSER

2.8 EXPERIMENT 8 - LEADING AND TRAILING

2.9 EXPERIMENT 9 - LR(0)

2.1 EXPERIMENT 10 - INTERMEDIATE CODE GENERATION

2.11 EXPERIMENT 11 - QUADRUPLES, TRIPLES, INDIRECT TRIPLES

2.12 EXPERIMENT 12 - SIMPLE CODE GENERATOR

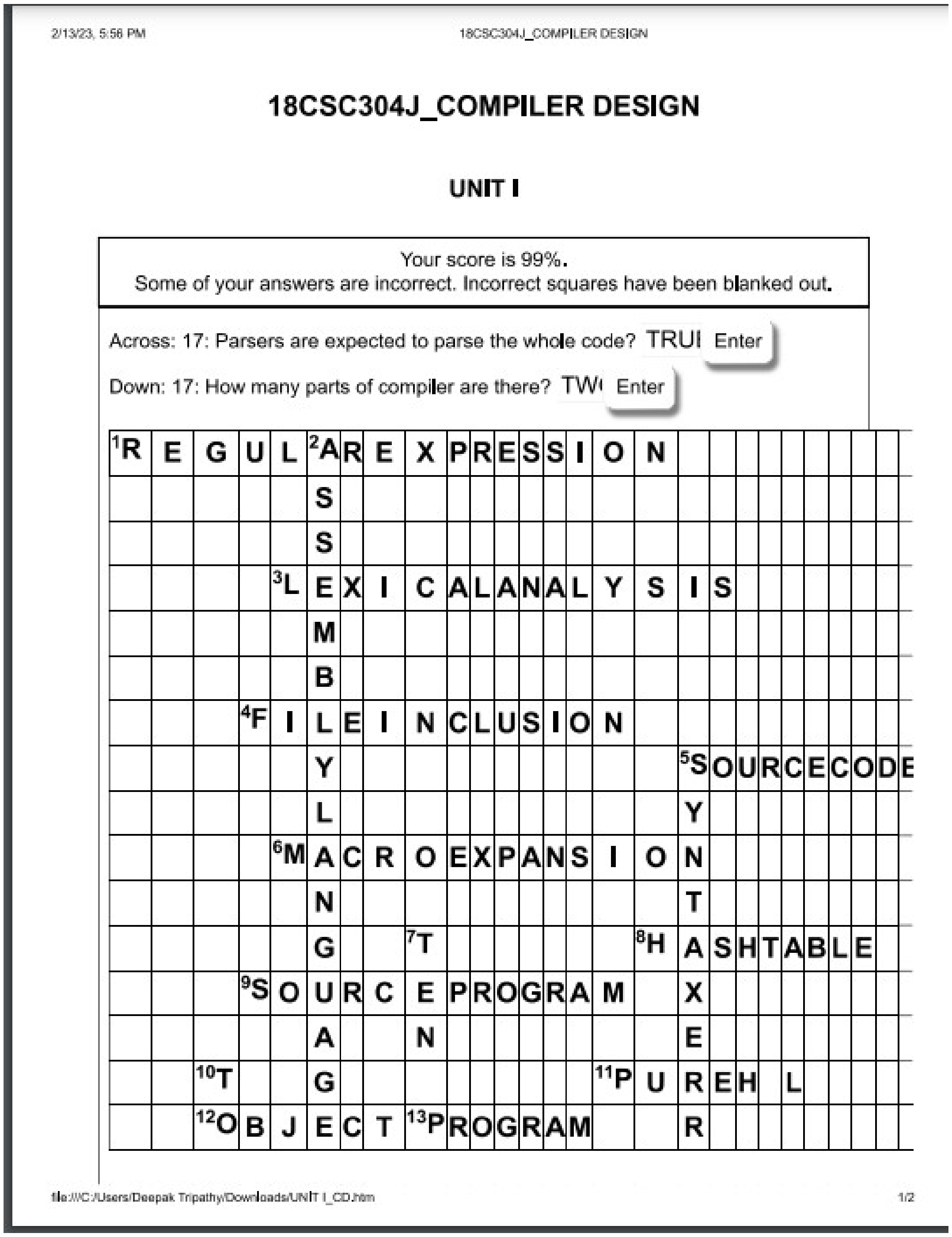
2.13 EXPERIMENT 13 - DAG

3. PROJECT

3.1 MINI PROJECT - MINI C++ COMPILER

ASSIGNMENTS

ASSIGNMENT 1: UNIT 1 CROSSWORD



2/13/23, 5:56 PM 18CSC304J_COMPILER DESIGN

18CSC304J_COMPILER DESIGN

UNIT I

Your score is 99%.
Some of your answers are incorrect. Incorrect squares have been blanked out.

Across: 17: Parsers are expected to parse the whole code? TRUI Enter

Down: 17: How many parts of compiler are there? TW(Enter

^{1}R	E	G	U	L	^{2}A	R	E	X	P	R	E	S	S	I	O	N										
					S																					
					S																					
				^{3}L	E	X	I	C	A	L	A	N	A	L	Y	S	I	S								
					M																					
					B																					
			^{4}F	I	L	E	I	N	C	L	U	S	I	O	N											
					Y												^{5}S	O	U	R	C	E	C	O	D	E
					L												Y									
				^{6}M	A	C	R	O	E	X	P	A	N	S	I	O	N									
					N												T									
					G			^{7}T								^{8}H	A	S	H	T	A	B	L	E		
			^{9}S	O	U	R	C	E	P	R	O	G	R	A	M		X									
					A			N									E									
		^{10}T			G										^{11}P	U	R	E	H		L					
		^{12}O	B	J	E	C	T	^{13}P	R	O	G	R	A	M			R									

file:///C:/Users/Deepak Tripathy/Downloads/UNIT I_CD.htm 1/2

		K						A								^{14}C	O	M	P	I	L	E	R			
	^{15}S	E	C	O	N	D		R									R									
		N						S																		
^{16}P	A	S	S	1				E																		
							^{17}T	R	U	E																
							W																			
							O																			

Check

Across:

1. Input of lex is set of ________.
3. In a compiler, keywords of a language are recognized during the_________of the program
4. Role of preprocessor
5. which is the permanent data base in the geneal model of Compiler ?
6. Function of preprocessor
8. The symbol table implementation is based on the property of locality of reference is______.
9. A compiler program written in a high level language is called_________.
11. Preprocessor converts high level language to_________________
12. ______________is a Translation of high-level language into machine language
14. By whom is the symbol table created?
15. Which phase of compiler is Syntax Analysis?
16. When is the symbol table created?
17. Parsers are expected to parse the whole code?

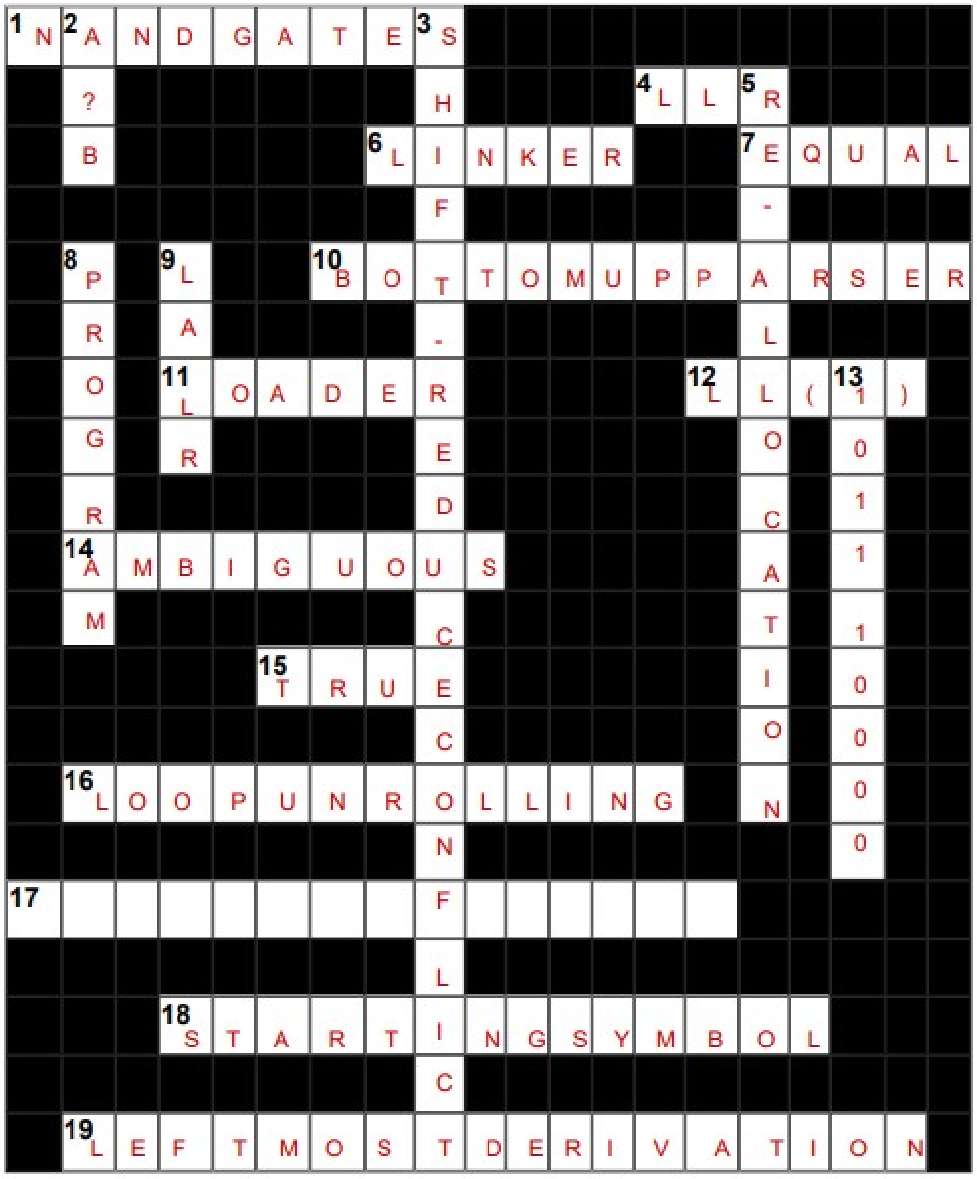
Down:

2. Assembler converts ______________to machine language,
5. Compiler can check
7. The number of tokens in the following C statement is printf("i = %d, &i = %x", i, &i);
10. The output of lexical analyzer is set of ________.
13. Which concept of grammar is used in the compiler?
17. How many parts of compiler are there?

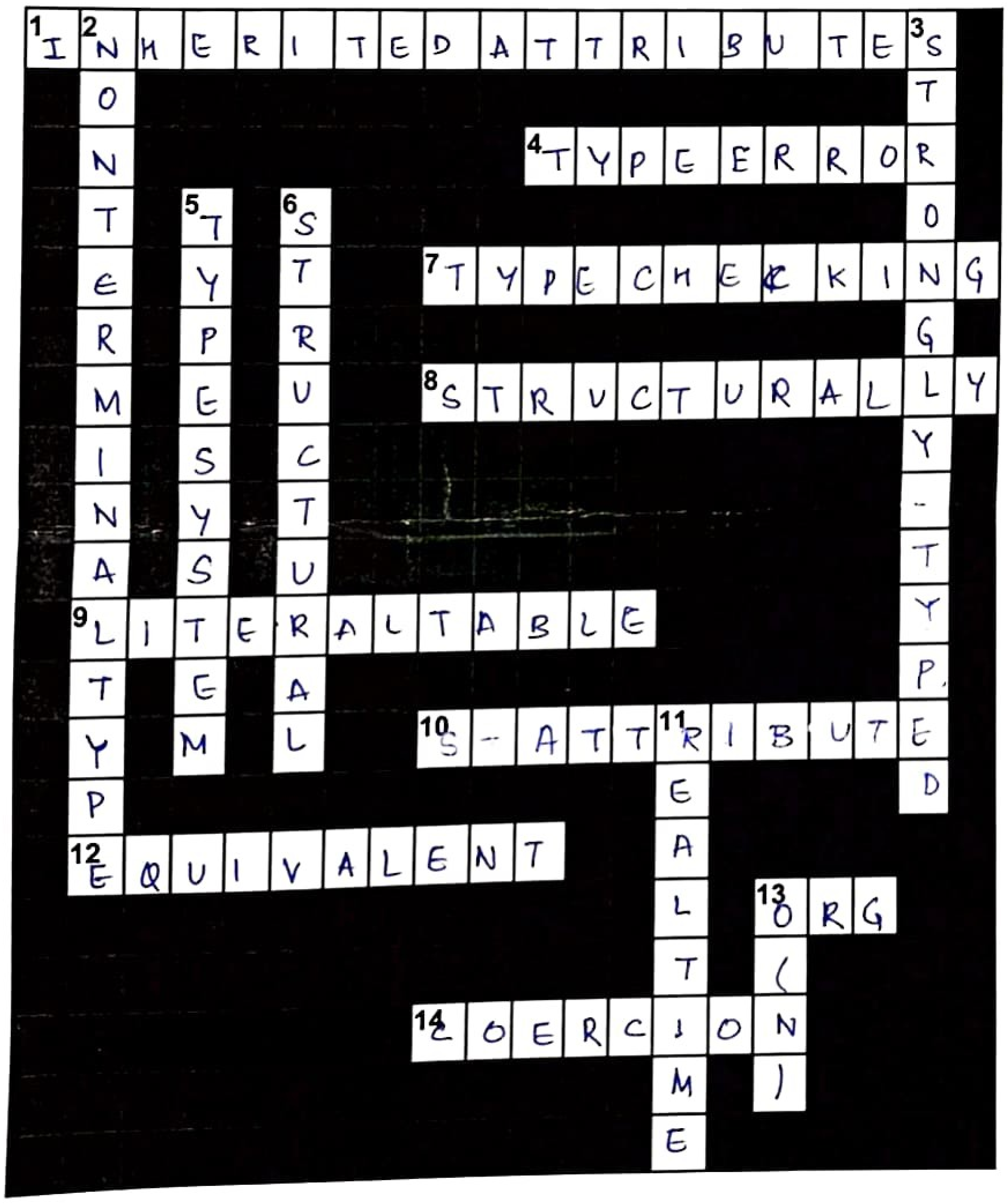
ASSIGNMENT 2: UNIT 2 CROSSWORD

18CSC304J/Compiler Design/Dr.Shiny Irene D

UNIT II


1 N 2 A N D G A T E 3 S
4 L L 5 R
6 L I N K E R
7 E Q U A L
8 P 9 L 10 B O T T O M U P P A R S E R
11 L O A D E R
12 L L (13 1)
14 A M B I G U O U S
15 T R U E
16 L O O P U N R O L L I N G
17
18 S T A R T I N G S Y M B O L
19 L E F T M O S T D E R I V A T I O N

UNIT III


1 INHERITEDATTRIBUTES
2 NONTERMINALITYP
3 STRONGLY-TYPED
4 TYPEERROR
5 TYPESYSTEM
6 STRUCTURAL
7 TYPECHECKING
8 STRUCTURALLY
9 LITERALTABLE
10 S-ATTRIBUTE
11 REALTIME
12 EQUIVALENT
13 ORG
14 COERCION
(N)

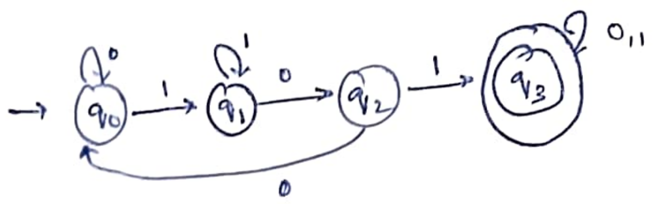
Name: Rishabh Singh Sahil

Reg. No. : RA2011003011334

CSE - J2

Assignment:

L = {101, 1101, 0101, . . .}



DFA are finite state machines that accept or reject strings of character by passing them through a sequence of character it uses a finite tape to store the string.

	0	1
$\rightarrow q_0$	q_0	q_1
q_1	q_2	q_1
q_2	q_0	q_3
q_3^*	q_3	q_3

To check acceptance of 110100

$\delta(q_0, 1) = q_1$ $\delta(q_3, 0) = q_3$

$\delta(q_1, 1) = q_1$ $\delta(q_3, 0) = q_3$

$\delta(q_1, 0) = q_2$ acceptance state

$\delta(q_2, 1) = q_3$ ∴ above string is acceptance

Final state

q_1, q_2, q_3

Non-Final States

q_0, q_3, q_5

$P_0 = \{\{q_1, q_2, q_4\}\}$

for $\{q_1, q_2, q_4\}$

$\delta(q_1, 0), \delta(q_2, 0) - q_2$

$\delta(q_1, 1), \delta(q_2, 1) - q_5$

$\delta(q_1, 0) = \delta(q_4, 0) = q_2$

$\delta(q_1, 1) = \delta(q_4, 1) = q_5$

$\{q_1, q_2, q_4\}$ will not be partioned

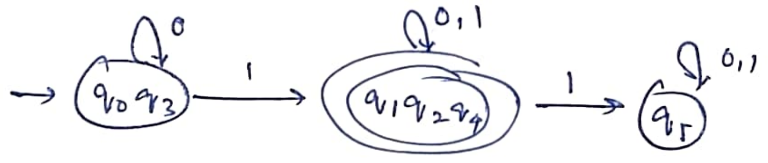
$\{q_0, q_3, q_5\}$

$\delta(q_0, 0) = q_3$ $\delta(q_3, 0) = q_0$ $\delta(q_0, 1) = q_1$

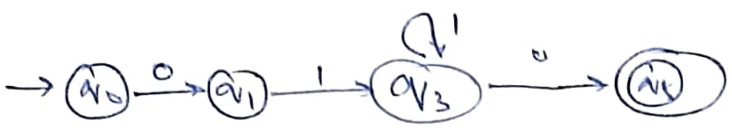
$\delta(q_3, 1) = q_4$ $\delta(q_0, 0) = q_3$ $\delta(q_5, 0) = q_5$

$\delta(q_0, 1) = q_1$ $\delta(q_5, 1) = q_5$

$P_1 = \{\{q_1, q_2, q_4\} \{q_0\ q_3\} \{q_5\}\}$

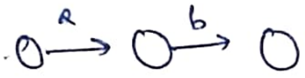


After applying the required changes

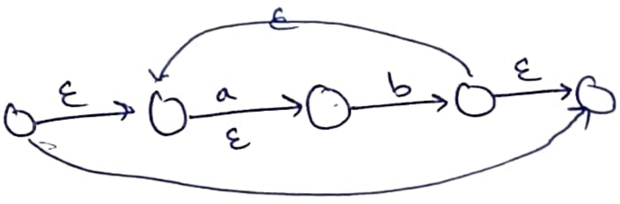


5. a (ab)* b

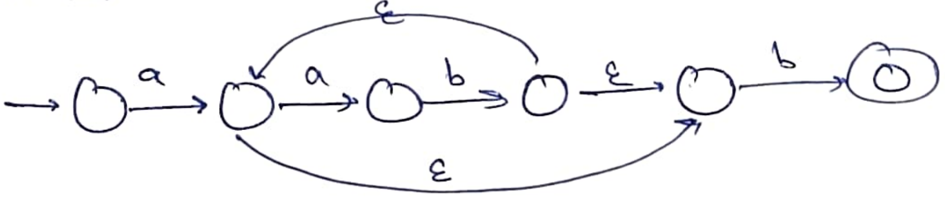
ab

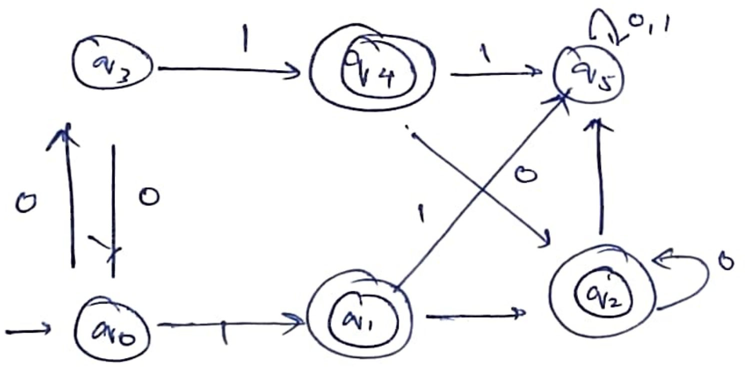


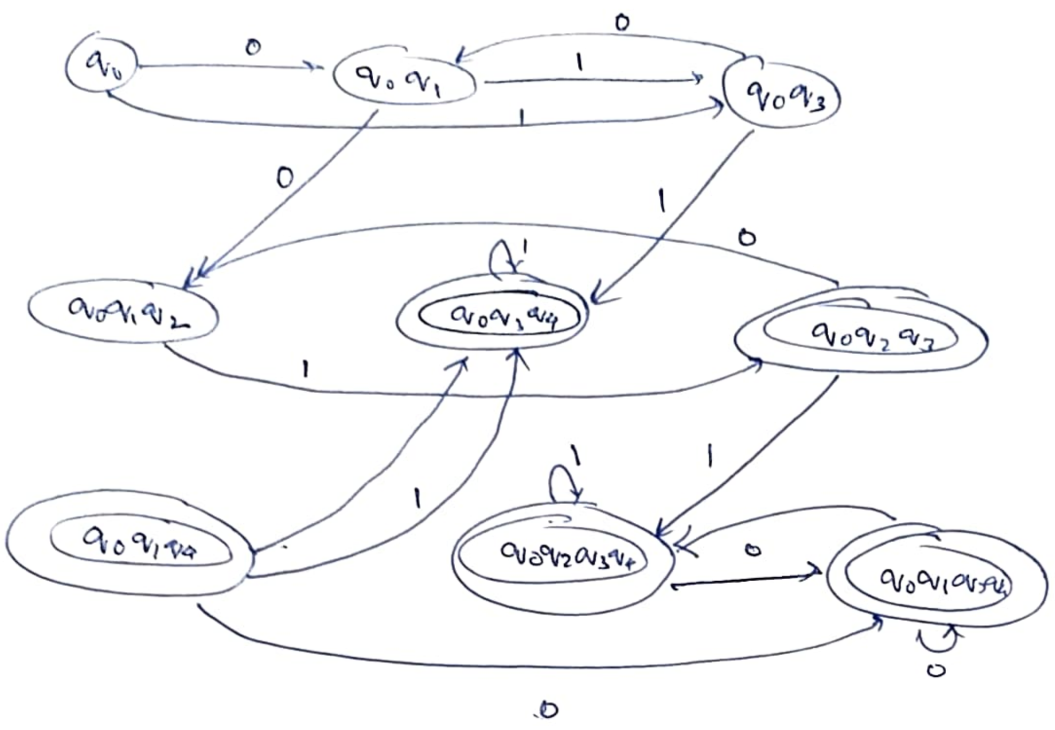
(ab)*



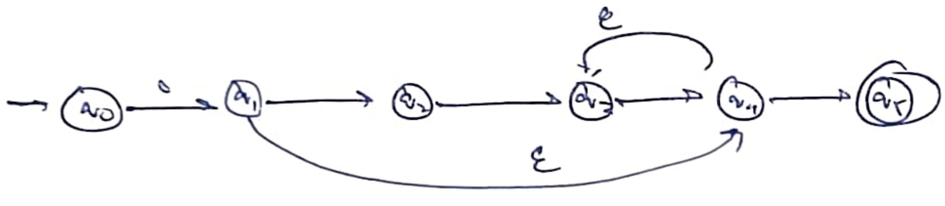
a (ab)* b

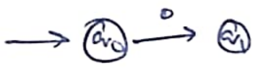




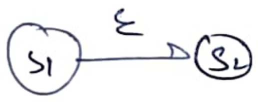


Above is DFA equivalent





Steps to eliminate ε-moves



1. Find all edges starting from S_2
2. Duplicate all edges to S_1 without changing edge labels.
3. Change initial to final start and vice versa
4. Remove lead state.

$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[loop above, "0,1"] \arrow[r, "0"] \arrow[dr, "1"] & q_1 \arrow[r, "0"] & \boxed{q_2} \arrow[loop right, "0,1"] \\
& & q_3 \arrow[r, "1"] & \boxed{q_4} \arrow[loop right, "0,1"]
\end{tikzcd}$$

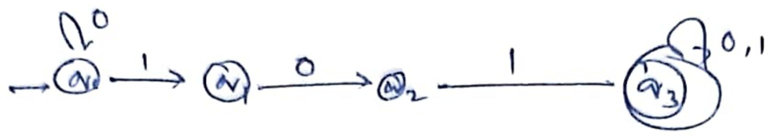
	0	1
$\rightarrow q_0$	$q_2 q_1$	$q_0 q_3$
$\cdot\, q_1$	q_3	Φ
$* q_2$	q_2	q_3
	Φ	q_4
$* q_3$	q_4	q_4

← NFA Transition table

To convert to DFA

	0	1
$\rightarrow q_0$	$q_0 q_1$	$q_0 q_3$
$\cdot\, q_0 q_1$	$q_0 q_1 q_2$	$q_0 q_3$
$q_0 q_3$	$q_0 q_1$	$q_0 q_3 q_4$
$q_0 q_1 q_2$	$q_0 q_1 q_2$	$q_0 q_2 q_3$
$q_0 q_3 q_4$	$q_0 q_1 q_4$	$q_0 q_3 q_4$
$q_0 q_2 q_3$	$q_0 q_1 q_2$	$q_0 q_2 q_3 q_4$
$q_0 q_1 q_4$	$q_0 q_1 q_2 q_4$	$q_0 q_3 q_4$
$q_0 q_2 q_3 q_4$	$q_0 q_1 q_2 q_4$	$q_0 q_2 q_3 q_4$
$q_0 q_1 q_2 q_4$		$q_0 q_2 q_3 q_4$

NFA



NFA is easier to construct than DFA it is called

so when on one input it can go to multiple states

Each NFA can be converted to DFA.

it can be defined using 5 tuples

$(Q, \Sigma, \delta, q_0, F)$

$Q \to$ set of states

$\Sigma \to$ set of inputs

$\delta \to$ transition functions

$q_0 \to$ initial states

$F \to$ set of final states

	0	1
$\to q_0$	q_0	q_1
q_1	q_2	ϕ
q_2	ϕ	q_3
q_3	q_3	q_3

LAB EXPERIMENTS

EXPERIMENT 1
DATE:

LEXICAL ANALYSER

AIM
To write a program to implement Lexical Analysis using C.

ALGORITHM
1. Start the program.
2. Declare all the variables and file pointers.
3. Display the input program.
4. Separate the keyword in the program and display it.
5. Display the header files of the input program
6. Separate the operators of the input program and display it.
7. Print the punctuation marks.
8. Print the constant that are present in input program.
9. Print the identifiers of the input program.

CODE
```
#include <fstream>
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
using namespace std;
bool isPunctuator(char ch) //check if the given
character is a punctuator or not
{
 if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
 ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
 ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
 ch == '[' || ch == ']' || ch == '{' || ch == '}' ||
 ch == '&' || ch == '|')
 {
 return true;
 }
 return false;
}
bool validIdentifier(char* str) //check if the given
identifier is valid or not
{
 if (str[0] == '0' || str[0] == '1' || str[0] == '2' ||
 str[0] == '3' || str[0] == '4' || str[0] == '5' ||
 str[0] == '6' || str[0] == '7' || str[0] == '8' ||
 str[0] == '9' || isPunctuator(str[0]) == true)
 {
 return false;
 } //if first character of string
is a digit or a special character, identifier is not valid
 int i,len = strlen(str);
 if (len == 1)
 {
 return true;
 } //if length is one,
validation is already completed, hence return true
 else
 {
 for (i = 1 ; i < len ; i++) //identifier cannot
contain special characters
 {
 if (isPunctuator(str[i]) == true)
 {
 return false;
 }
 }
 }
 return true;
}
bool isOperator(char ch) //check if the given
character is an operator or not
{
 if (ch == '+' || ch == '-' || ch == '*' ||
 ch == '/' || ch == '>' || ch == '<' ||
 ch == '=' || ch == '|' || ch == '&')
 {
 return true;
 }
 return false;
}
bool isKeyword(char *str) //check if the given
substring is a keyword or not
{
 if (!strcmp(str, "if") || !strcmp(str, "else") ||
 !strcmp(str, "while") || !strcmp(str, "do") ||
 !strcmp(str, "break") || !strcmp(str, "continue")
 || !strcmp(str, "int") || !strcmp(str, "double")
 || !strcmp(str, "float") || !strcmp(str, "return")
 || !strcmp(str, "char") || !strcmp(str, "case")
 || !strcmp(str, "long") || !strcmp(str, "short")
 || !strcmp(str, "typedef") || !strcmp(str, "switch")
 || !strcmp(str, "unsigned") || !strcmp(str, "void")
 || !strcmp(str, "static") || !strcmp(str, "struct")
 || !strcmp(str, "sizeof") || !strcmp(str,"long")
 || !strcmp(str, "volatile") || !strcmp(str, "typedef")
 || !strcmp(str, "enum") || !strcmp(str, "const")
 || !strcmp(str, "union") || !strcmp(str, "extern")
 || !strcmp(str,"bool"))
 {
 return true;
 }
 else
 {
 return false;
 }
}
bool isNumber(char* str) //check if the given
substring is a number or not
{
 int i, len = strlen(str),numOfDecimal = 0;
 if (len == 0)
 {
 return false;
 }
 for (i = 0 ; i < len ; i++)
 {
 if (numOfDecimal > 1 && str[i] == '.')
 {
 return false;
 } else if (numOfDecimal <= 1)
 {
 numOfDecimal++;
 }
 if (str[i] != '0' && str[i] != '1' && str[i] != '2'
 && str[i] != '3' && str[i] != '4' && str[i] != '5'
 && str[i] != '6' && str[i] != '7' && str[i] != '8'
 && str[i] != '9' || (str[i] == '-' && i > 0))
 {
 return false;
 }
 }
 return true;
}
char* subString(char* realStr, int l, int r) //extract the required
substring from the main string
{
 int i;
 char* str = (char*) malloc(sizeof(char) * (r - l + 2));
 for (i = l; i <= r; i++)
 {
 str[i - l] = realStr[i];
 str[r - l + 1] = '\0';
 }
 return str;
}
void parse(char* str) //parse the expression
{
 int left = 0, right = 0;
 int len = strlen(str);
 while (right <= len && left <= right) {
 if (isPunctuator(str[right]) == false) //if character is a digit or an
alphabet
 {
 right++;
 }
 if (isPunctuator(str[right]) == true && left == right) //if character is a
punctuator
```

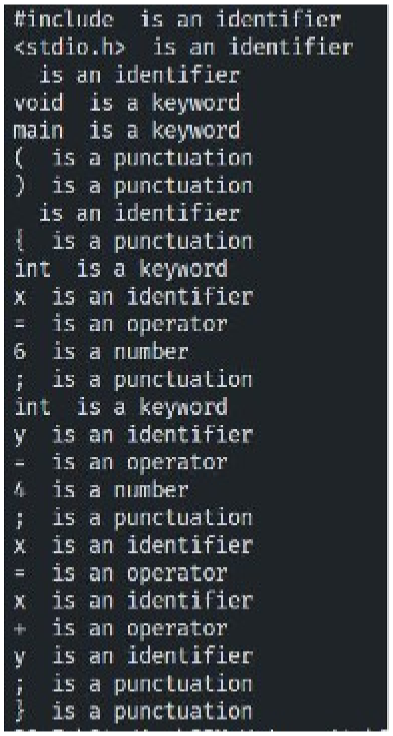
```
 {
 if (isOperator(str[right]) == true)
 {
 std::cout<< str[right] <<" IS AN OPERATOR\n";
 }
 right++;
 left = right;
 } else if (isPunctuator(str[right]) == true && left != right
 || (right == len && left != right)) //check if parsed
substring is a keyword or identifier or number
 {
 char* sub = subString(str, left, right - 1); //extract substring
 if (isKeyword(sub) == true)
 {
 cout<< sub <<" IS A KEYWORD\n";
 }
 else if (isNumber(sub) == true)
 {
 cout<< sub <<" IS A NUMBER\n";
 }
 else if (validIdentifier(sub) == true
 && isPunctuator(str[right - 1]) == false)
 {
 cout<< sub <<" IS A VALID IDENTIFIER\n";
 }
 else if (validIdentifier(sub) == false
 && isPunctuator(str[right - 1]) == false)
 {
 cout<< sub <<" IS NOT A VALID IDENTIFIER\n";
 }
 left = right;
 }
 }
 return;
}
int main()
{
 char c[100] = "int a = b * c";
 parse(c);
 return 0;
```

}

SAMPLE INPUT AND OUTPUT

INPUT :

```
#include <stdio.h>

void main ( )

{
   int x = 6 ;
   int y = 4 ;
   x = x + y ;
}
```



```
#include  is an identifier
<stdio.h>  is an identifier
  is an identifier
void  is a keyword
main  is a keyword
(  is a punctuation
)  is a punctuation
  is an identifier
{  is a punctuation
int  is a keyword
x  is an identifier
=  is an operator
6  is a number
;  is a punctuation
int  is a keyword
y  is an identifier
=  is an operator
4  is a number
;  is a punctuation
x  is an identifier
=  is an operator
x  is an identifier
+  is an operator
y  is an identifier
;  is a punctuation
}  is a punctuation
```

RESULT

Thus, the program for lexical analyzer was executed successfully.

EXPERIMENT 2
DATE:

RE TO NFA

AIM
To write a program for converting Regular Expression to NFA.

ALGORITHM
1. Start
2. Get the input from the user
3. Initialize separate variables and functions for Postfix , Display and NFA
4. Create separate methods for different operators like +,*, .
5. By using Switch case Initialize different cases for the input
6. For ' . ' operator Initialize a separate method by using various stack functions do the same for the other operators like ' * ' and ' + '.
7. Regular expression is in the form like a.b (or) a+b
8. Display the output
9. Stop

CODE
```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct prodn
{
        char p1[10];
        char p2[10];
};
void main()
{
        char input[20],stack[50],temp[50],ch[2],*t1,*t2,*t;
        int i,j,s1,s2,s,count=0;
        struct prodn p[10];
        FILE *fp=fopen("sr_input.txt","r");
        stack[0]='\0';
        printf("\n Enter the input string\n");
        scanf("%s",&input);
        while(!feof(fp))
        {
```

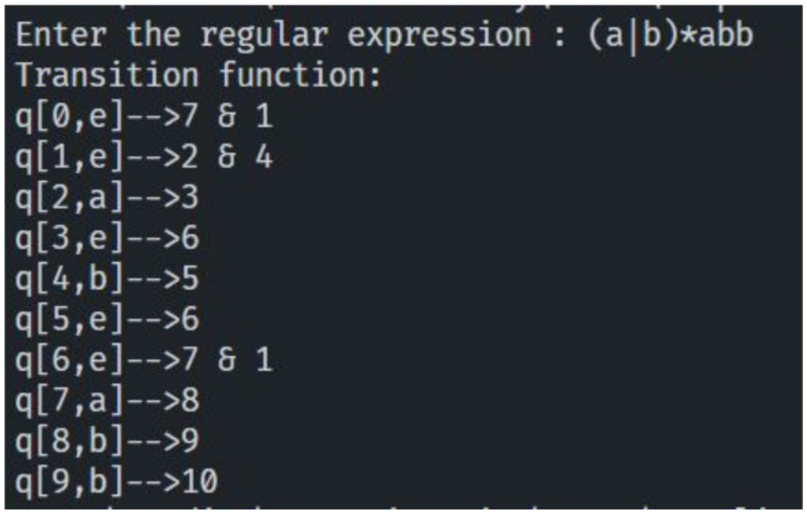
```
		fscanf(fp,"%s\n",temp);
		t1=strtok(temp,"->");
		t2=strtok(NULL,"->");
		strcpy(p[count].p1,t1);
		strcpy(p[count].p2,t2);
		count++;
	}
	i=0;
	while(1)
	{
		if(i<strlen(input))
		{
			ch[0]=input[i];
			ch[1]='\0';
			i++;
			strcat(stack,ch);
			printf("%s\n",stack);
		}
		for(j=0;j<count;j++)
		{
			t=strstr(stack,p[j].p2);
			if(t!=NULL)
			{
				s1=strlen(stack);
				s2=strlen(t);
				s=s1-s2;
				stack[s]='\0';
				strcat(stack,p[j].p1);
				printf("%s\n",stack);
				j=-1;
			}
		}
		if(strcmp(stack,"E")==0&&i==strlen(input))
		{
			printf("\n Accepted");
			break;
		}
		if(i==strlen(input))
		{
			printf("\n Not Accepted");
```

```
                        break;
                }
        }
}
```

SAMPLE INPUT AND OUTPUT

INPUT : (a|b)*abb

OUTPUT :



```
Enter the regular expression : (a|b)*abb
Transition function:
q[0,e]-->7 & 1
q[1,e]-->2 & 4
q[2,a]-->3
q[3,e]-->6
q[4,b]-->5
q[5,e]-->6
q[6,e]-->7 & 1
q[7,a]-->8
q[8,b]-->9
q[9,b]-->10
```

RESULT

Thus, the code for RE to NFA was executed successfully.

EXPERIMENT 3

NFA TO DFA

AIM
To write a program for converting NFA to DFA.

ALGORITHM
1. Start
2. Get the input from the user
3. Set the only state in SDFA to “unmarked”.
4. while SDFA contains an unmarked state do:
5. Let T be that unmarked state
6. b. for each a in % do S = e-Closure(MoveNFA(T,a)) c. if S is not in SDFA already then, add S to SDFA (as an “unmarked” state) d. Set MoveDFA(T,a) to S.
7. For each S in SDFA if any s & S is a final state in the NFA then, mark S an a final state in the DFA
8. Print the result.
9. Stop the program.

CODE
```
#include<stdio.h>
#include<string.h>
#include<math.h>
int ninputs;
int dfa[100][2][100] = {0};
int state[10000] = {0};
char  ch[10],  str[1000];
int go[10000][2] = {0};
int arr[10000] = {0};
int main()
{
 int st, fin, in;
 int f[10];
 int i,j=3,s=0,final=0,flag=0,curr1,curr2,k,l;
 int c;

 printf("\nFollow the one based indexing\n");

 printf("\nEnter the number of states::");
```

```
scanf("%d",&st);

printf("\nGive state numbers from 0 to %d",st-1);

for(i=0;i<st;i++)
state[(int)(pow(2,i))] = 1;
printf("\nEnter number of final states\t");
scanf("%d",&fin);
printf("\nEnter final states::");
for(i=0;i<fin;i++)
{
scanf("%d",&f[i]);
}
int p,q,r,rel;

printf("\nEnter the number of rules according to NFA::");
scanf("%d",&rel);

printf("\n\nDefine transition rule as \"initial state input symbol final state\"\n");


for(i=0; i<rel; i++)
{
scanf("%d%d%d",&p,&q,&r);
if (q==0)
dfa[p][0][r] = 1;
else
dfa[p][1][r] = 1;
}

printf("\nEnter initial state::");
scanf("%d",&in);
in = pow(2,in);
i=0;

printf("\nSolving according to DFA");

int x=0;
for(i=0;i<st;i++)
{
```

```
for(j=0;j<2;j++)
{
int stf=0;
for(k=0;k<st;k++)
{
if(dfa[i][j][k]==1)
stf = stf + pow(2,k);
}


go[(int)(pow(2,i))][j] = stf;
printf("%d-%d-->%d\n",(int)(pow(2,i)),j,stf);
if(state[stf]==0)
arr[x++] = stf;
state[stf] = 1;
}

}



//for new states
for(i=0;i<x;i++)
{
printf("for %d----- ",arr[x]);
for(j=0;j<2;j++)
{
int new=0;
for(k=0;k<st;k++)
{
if(arr[i] & (1<<k))
{
int h = pow(2,k);

if(new==0)
new = go[h][j];
new = new | (go[h][j]);


}
}
if(state[new]==0)
{
arr[x++] = new;
state[new] = 1;
}
}
}

printf("\nThe total number of distinct states are::\n");

printf("STATE 0 1\n");

for(i=0;i<10000;i++)
{
if(state[i]==1)
{
//printf("%d**",i);
int y=0;
if(i==0)
printf("q0 ");

else
for(j=0;j<st;j++)
{
int x = 1<<j;
if(x&i)
{
printf("q%d ",j);
y = y+pow(2,j);
//printf("y=%d ",y);
}
}
//printf("%d",y);
printf(" %d %d",go[y][0],go[y][1]);
printf("\n");
}
}
```

```
 j=3;
 while(j--)
 {
 printf("\nEnter string");
 scanf("%s",str);
 l = strlen(str);
 curr1 = in;
 flag = 0;
 printf("\nString takes the following path-->\n");
 printf("%d-",curr1);
 for(i=0;i<l;i++)
 {
 curr1 = go[curr1][str[i]-'0'];
 printf("%d-",curr1);
 }

 printf("\nFinal state - %d\n",curr1);

 for(i=0;i<fin;i++)
 {
 if(curr1 & (1<<f[i]))
 {
 flag = 1;
 break;
 }
 }

 if(flag)
 printf("\nString Accepted");
 else
 printf("\nString Rejected");

 }


 return 0;
}
```

SAMPLE INPUT AND OUTPUT

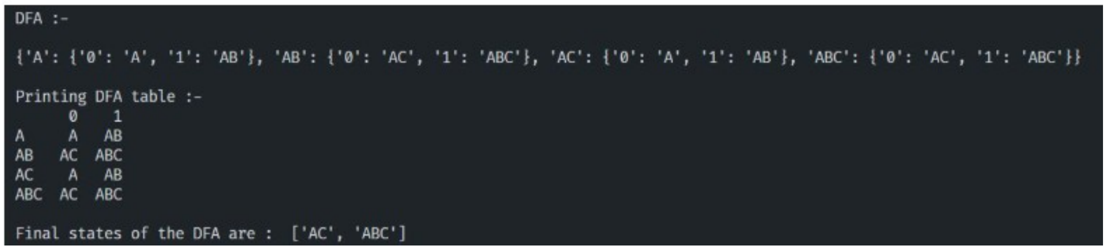
INPUT :

```
No. of states : 3

No. of transitions : 2

state name : A

path : 0
Enter end state from state A travelling through path 0 :

A

path : 1
Enter end state from state A travelling through path 1 :

A B

state name : B

path : 0
Enter end state from state B travelling through path 0 :

C

path : 1
Enter end state from state B travelling through path 1 :

C

state name : C
```

```
path : 0
Enter end state from state C travelling through path 0 :
```

```
path : 1
Enter end state from state C travelling through path 1 :



NFA :-

{'A': {'0': ['A'], '1': ['A', 'B']}, 'B': {'0': ['C'], '1': ['C']}, 'C': {'0': [], '1': []}}

Printing NFA table :-
    0      1
A  [A]  [A, B]
B  [C]     [C]
C   []      []
Enter final state of NFA :

C
```

OUTPUT :



```
DFA :-

{'A': {'0': 'A', '1': 'AB'}, 'AB': {'0': 'AC', '1': 'ABC'}, 'AC': {'0': 'A', '1': 'AB'}, 'ABC': {'0': 'AC', '1': 'ABC'}}

Printing DFA table :-
      0    1
A     A   AB
AB   AC  ABC
AC    A   AB
ABC  AC  ABC

Final states of the DFA are :  ['AC', 'ABC']
```

EXPERIMENT 4

LEFT RECURSION AND LEFT FACTORING

AIM

A program for Elimination of Left Recursion and Left Factoring.

ALGORITHM

LEFT RECURSION

1. Start the program.
2. Initialize the arrays for taking input from the user.
3. Prompt the user to input the no. of non-terminals having left recursion and no. of productions for these non-terminals.
4. Prompt the user to input the production for non-terminals.
5. Eliminate left recursion using the following rules:-

A->Aα1| Aα2 |.......... |Aαm

A->β1| β2|......... | βn

Then replace it by

A-> βi A' i=1,2,3,…. m

A'-> αj A' j=1,2,3,…. n

A'-> Ɛ

6. After eliminating the left recursion by applying these rules, display the productions without left recursion.
7. Stop.

LEFT FACTORING

1. Start
2. Ask the user to enter the set of productions
3. Check for common symbols in the given set of productions by comparing with: A->aB1|aB2
4. If found, replace the particular productions with: A->aA' A'->B1 | B2|ε
5. Display the output
6. Exit

CODE (left factoring)

```
#include <iostream>
#include <vector>
#include  <string>
using namespace std;
int main()
{
 int n;
 cout<<"\nEnter number of non terminals: ";
 cin>>n;
 cout<<"\nEnter non terminals one by one: ";
 int i;
 vector<string> nonter(n);
 vector<int>  leftrecr(n,0);
 for(i=0;i<n;++i) {
 cout<<"\nNon terminal "<<i+1<<" : ";
 cin>>nonter[i];
 }
 vector<vector<string> > prod;
 cout<<"\nEnter '^' for null";
 for(i=0;i<n;++i) {
 cout<<"\nNumber of "<<nonter[i]<<" productions: ";
 int k;
 cin>>k;
 int j;
 cout<<"\nOne by one enter all "<<nonter[i]<<" productions";
 vector<string> temp(k);
 for(j=0;j<k;++j) {
 cout<<"\nRHS of production "<<j+1<<": ";
 string abc;
 cin>>abc;
 temp[j]=abc;
if(nonter[i].length()<=abc.length()&&nonter[i].compare(abc.substr(0,nonter[i].length()))==0)
 leftrecr[i]=1;
 }
 prod.push_back(temp);
 }
 for(i=0;i<n;++i)
 {cout<<leftrecr[i];
 }
 for(i=0;i<n;++i)
 {if(leftrecr[i]==0)
 continue;
 int j;
 nonter.push_back(nonter[i]+"'");
 vector<string> temp;
 for(j=0;j<prod[i].size();++j) {
```

```
if(nonter[i].length()<=prod[i][j].length()&&nonter[i].compare(prod[i][j].substr(0,nonter[i].length
()))==0) {
 string
abc=prod[i][j].substr(nonter[i].length(),prod[i][j].length()-nonter[i].length())+nonter[i]+"'";
 temp.push_back(abc);
 prod[i].erase(prod[i].begin()+j);
 --j;
 }
 else
 { prod[i][j]+=nonter[i]+"'
 ";
 }
 }
 temp.push_back("^");
 prod.push_back(temp);
 }
 cout<<"\n\n";
 cout<<"\nNew set of non-terminals: ";
 for(i=0;i<nonter.size();++i)
 cout<<nonter[i]<<" ";
 cout<<"\n\nNew set of productions: ";
 for(i=0;i<nonter.size();++i) {
 int j;
 for(j=0;j<prod[i].size();++j) {
 cout<<"\n"<<nonter[i]<<" -> "<<prod[i][j];
 }
 }
 return 0;
}
```

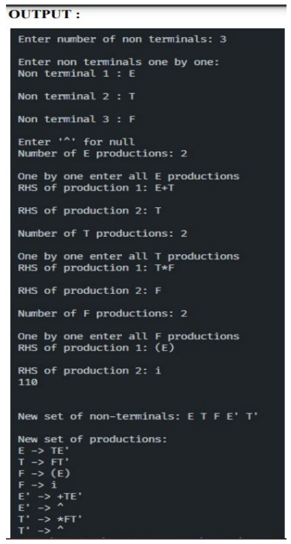
<u>CODE (left factoring)</u>

```
#include <iostream>
#include <math.h>
#include <vector>
#include <string>
#include <stdlib.h>
using namespace std;
int main()
{
 cout<<"\nEnter number of productions: ";
 int p;
```

```
cin>>p;
vector<string> prodleft(p),prodright(p);
cout<<"\nEnter productions one by one: ";
int i;
for(i=0;i<p;++i) {
cout<<"\nLeft of production "<<i+1<<": ";
cin>>prodleft[i];
cout<<"\nRight of production "<<i+1<<": ";
cin>>prodright[i];
}
int  j;
int e=1;
for(i=0;i<p;++i)
{ for(j=i+1;j<p;++j)
{ if(prodleft[j]==prodleft[i])
{int k=0;
string com="";
while(k<prodright[i].length()&&k<prodright[j].length()&&prodright[i][k]==prodright[j][k])
{com+=prodright[i][k];
++k;
}
if(k==0)
continue;
char* buffer;
string comleft=prodleft[i];
if(k==prodright[i].length())
{ prodleft[i]+=string(itoa(e,buffer,10));
prodleft[j]+=string(itoa(e,buffer,10));
prodright[i]="^";
prodright[j]=prodright[j].substr(k,prodright[j].length()-k);
}
else if(k==prodright[j].length())
{ prodleft[i]+=string(itoa(e,buffer,10));
prodleft[j]+=string(itoa(e,buffer,10));
prodright[j]="^";
prodright[i]=prodright[i].substr(k,prodright[i].length()-k);
}
else
{ prodleft[i]+=string(itoa(e,buffer,10));
prodleft[j]+=string(itoa(e,buffer,10));
```

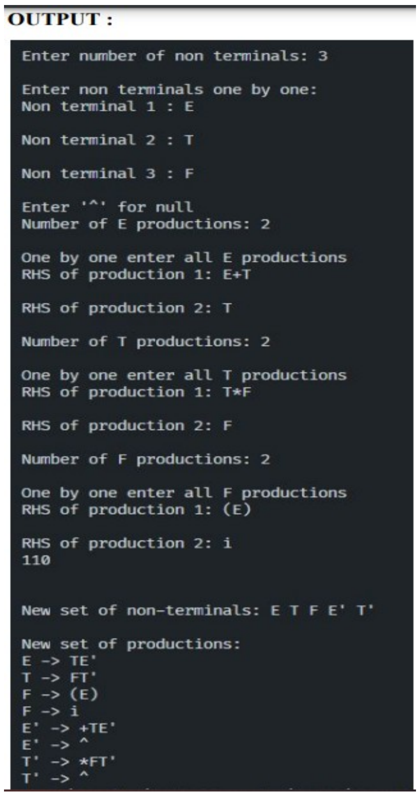
```
 prodright[j]=prodright[j].substr(k,prodright[j].length()-k);
 prodright[i]=prodright[i].substr(k,prodright[i].length()-k);
 }
 int l;
 for(l=j+1;l<p;++l) {
if(comleft==prodleft[l]&&com==prodright[l].substr(0,fmin(k,prodright[l].length())))
 {prodleft[l]+=string(itoa(e,buffer,10));
 prodright[l]=prodright[l].substr(k,prodright[l].length()-k);
 }
 }
 prodleft.push_back(comleft);
 prodright.push_back(com+prodleft[i]);
 ++p;
 ++e;
 }
 }
 }
 cout<<"\n\nNew productions";
 for(i=0;i<p;++i)
 { cout<<"\n"<<prodleft[i]<<"-
>"<<prodright[i];
 }
 return 0; }
```

SAMPLE INPUT AND OUTPUT



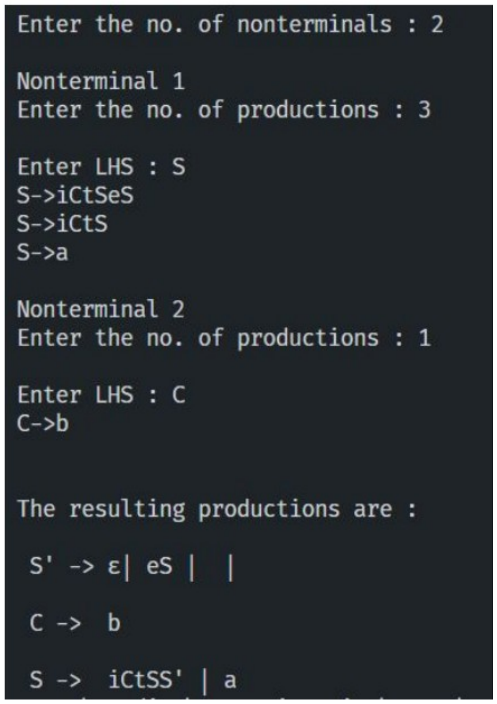
OUTPUT :

```
Enter number of non terminals: 3

Enter non terminals one by one:
Non terminal 1 : E

Non terminal 2 : T

Non terminal 3 : F

Enter '^' for null
Number of E productions: 2

One by one enter all E productions
RHS of production 1: E+T

RHS of production 2: T

Number of T productions: 2

One by one enter all T productions
RHS of production 1: T*F

RHS of production 2: F

Number of F productions: 2

One by one enter all F productions
RHS of production 1: (E)

RHS of production 2: i
110


New set of non-terminals: E T F E' T'

New set of productions:
E -> TE'
T -> FT'
F -> (E)
F -> i
E' -> +TE'
E' -> ^
T' -> *FT'
T' -> ^
```


OUTPUT :

```
Enter number of non terminals: 3

Enter non terminals one by one:
Non terminal 1 : E

Non terminal 2 : T

Non terminal 3 : F

Enter '^' for null
Number of E productions: 2

One by one enter all E productions
RHS of production 1: E+T

RHS of production 2: T

Number of T productions: 2

One by one enter all T productions
RHS of production 1: T*F

RHS of production 2: F

Number of F productions: 2

One by one enter all F productions
RHS of production 1: (E)

RHS of production 2: i
110


New set of non-terminals: E T F E' T'

New set of productions:
E -> TE'
T -> FT'
F -> (E)
F -> i
E' -> +TE'
E' -> ^
T' -> *FT'
T' -> ^
```

OUTPUT :



```
Enter the no. of nonterminals : 2

Nonterminal 1
Enter the no. of productions : 3

Enter LHS : S
S->iCtSeS
S->iCtS
S->a

Nonterminal 2
Enter the no. of productions : 1

Enter LHS : C
C->b


The resulting productions are :

 S' -> ε| eS |  |

 C ->  b

 S ->  iCtSS' | a
```

<u>RESULT</u>

Thus, the programs for left recursion and left factoring elimination were executed successfully.

EXPERIMENT 5

FIRST AND FOLLOW

AIM

To write a program to implement first and follow using C.

ALGORITHM

First:

To find the first() of the grammar symbol, then we have to apply the following set of rules to the given grammar:-

• If X is a terminal, then First(X) is {X}.

• If X is a non-terminal and X tends to aα is production, then add 'a' to the first of X. if X->ε, then add null to the First(X).

• If X_>YZ then if First(Y)=ε, then First(X) = { First(Y)-ε} U First(Z).

• If X->YZ, then if First(X)=Y, then First(Y)=terminal but null then First(X)=First(Y)=terminals.

Follow:

To find the follow(A) of the grammar symbol, then we have to apply the following set of rules to the given grammar:-

• $ is a follow of 'S'(start symbol).

• If A->αBβ,β!=ε, then first(β) is in follow(B).

• If A->αB or A->αBβ where First(β)=ε, then everything in Follow(A) is a Follow(B).

CODE

```
// C program to calculate the First and
// Follow sets of a given grammar
#include<stdio.h>
#include<ctype.h>
#include<string.h>
// Functions to calculate Follow
void followfirst(char, int, int);
void follow(char c);
// Function to calculate First
void findfirst(char, int, int);
int count, n = 0;
// Stores the final result
// of the First Sets
char calc_first[10][100];
// Stores the final result
// of the Follow Sets
```

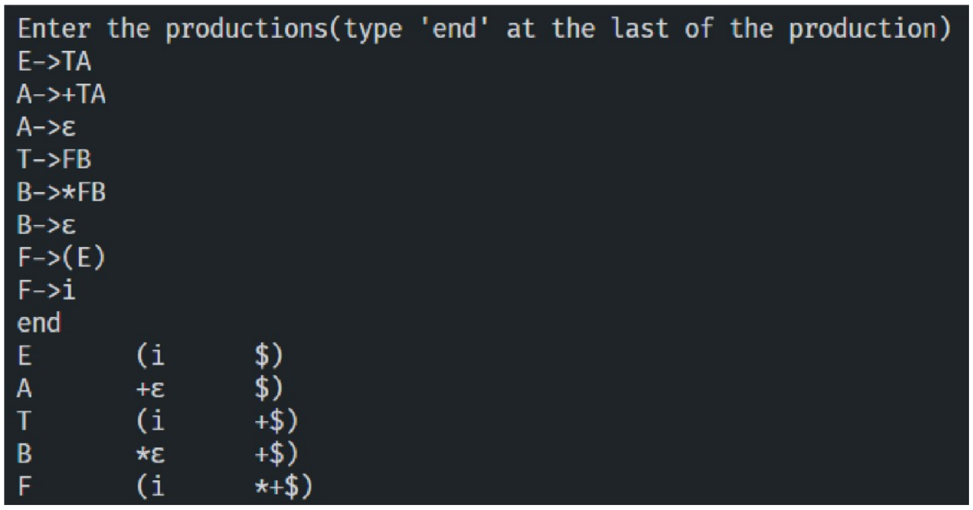
```
char calc_follow[10][100];
int m = 0;
// Stores the production rules
char production[10][10];
char f[10], first[10];
int k;
char ck;
int e;
int main(int argc, char **argv)
{
int jm = 0;
int km = 0;
int i, choice;
char c, ch;
count = 8;
// The Input grammar
strcpy(production[0], "E=TR");
strcpy(production[1], "R=+TR");
strcpy(production[2], "R=#");
strcpy(production[3], "T=FY");
strcpy(production[4], "Y=*FY");
strcpy(production[5], "Y=#");
strcpy(production[6], "F=(E)");
strcpy(production[7], "F=i");
int kay;
char done[count];
int ptr = -1;
// Initializing the calc_first array
for(k = 0; k < count; k++)
{ for(kay = 0; kay < 100; kay++)
{calc_first[k][kay] = '!';
}
}
int point1 = 0, point2, xxx;
for(k = 0; k < count; k++)
{
c = production[k][0];
point2 = 0;
xxx = 0;
// Checking if First of c has
// already been calculated
for(kay = 0; kay <= ptr; kay++)
if(c == done[kay])
xxx = 1;
if (xxx == 1)
continue;
// Function call
findfirst(c, 0, 0);
ptr += 1;
// Adding c to the calculated list
done[ptr] = c;
printf("\n First(%c) = { ", c);
calc_first[point1][point2++] = c;
// Printing the First Sets of the grammar
for(i = 0 + jm; i < n; i++) {
int lark = 0, chk = 0;
for(lark = 0; lark < point2; lark++)
{ if (first[i] ==
calc_first[point1][lark])
{
chk = 1;
break;
}
}
if(chk == 0)
{
printf("%c, ", first[i]);
calc_first[point1][point2++] = first[i];
}
}
printf("}\n");
jm = n;
point1++;
}
printf("\n");
printf("----------------------------------------------- \n\n");
char donee[count];
ptr = -1;
// Initializing the calc_follow array
for(k = 0; k < count; k++)
{ for(kay = 0; kay < 100; kay++) {
```

```
calc_follow[k][kay] = '!';
}
}
point1 = 0;
int land = 0;
for(e = 0; e < count; e++)
{
ck = production[e][0];
point2 = 0;
xxx = 0;
// Checking if Follow of ck
// has alredy been calculated
for(kay = 0; kay <= ptr; kay++)
if(ck == donee[kay])
xxx = 1;
if (xxx == 1)
continue;
land += 1;
// Function call
follow(ck);
ptr += 1;
// Adding ck to the calculated list
donee[ptr] = ck;
printf(" Follow(%c) = { ", ck);
calc_follow[point1][point2++] = ck;
// Printing the Follow Sets of the grammar
for(i = 0 + km; i < m; i++) {
int lark = 0, chk = 0;
for(lark = 0; lark < point2; lark++)
{
if (f[i] == calc_follow[point1][lark])
{
chk = 1;
break;
}
}
if(chk == 0)
{
printf("%c, ", f[i]);
calc_follow[point1][point2++] = f[i];
}
}
printf(" }\n\n");
km = m;
point1++;
}
}
void follow(char c)
{
int i, j;
// Adding "$" to the follow
// set of the start symbol
if(production[0][0] == c)
{f[m++] = '$';
}
for(i = 0; i < 10; i++)
{
for(j = 2;j < 10; j++)
{
if(production[i][j] == c)
{
if(production[i][j+1] != '\0')
{
// Calculate the first of the next
// Non-Terminal in the production
followfirst(production[i][j+1], i, (j+2));
}
if(production[i][j+1]=='\0' && c!=production[i][0])
{
// Calculate the follow of the Non-Terminal
// in the L.H.S. of the production
follow(production[i][0]);
}
}
}
}
}
void findfirst(char c, int q1, int q2)
{
int j;
```

```
// The case where we
// encounter a Terminal
if(!(isupper(c)))
{ first[n++] = c;
}
for(j = 0; j < count; j++)
{
if(production[j][0] == c)
{
if(production[j][2] == '#')
{
if(production[q1][q2] == '\0')
first[n++] = '#';
else if(production[q1][q2] != '\0'
&& (q1 != 0 || q2 != 0))
{
// Recursion to calculate First of New
// Non-Terminal we encounter after epsilon
findfirst(production[q1][q2], q1, (q2+1));
}
else
first[n++] = '#';
}
else if(!isupper(production[j][2]))
{
first[n++] = production[j][2];
}
else
{
// Recursion to calculate First of
// New Non-Terminal we encounter
// at the beginning
findfirst(production[j][2], j, 3);
}
}
}
}
void followfirst(char c, int c1, int c2)
{
int k;
```

```
// The case where we encounter
// a Terminal
if(!(isupper(c)))
f[m++] = c;
else
{
int i = 0, j = 1;
for(i = 0; i < count; i++)
{
if(calc_first[i][0] == c)
break;
}
//Including the First set of the
// Non-Terminal in the Follow of
// the original query
while(calc_first[i][j] != '!')
{
if(calc_first[i][j] != '#')
{
f[m++] = calc_first[i][j];
}
else
{
if(production[c1][c2] == '\0')
{
// Case where we reach the
// end of a production
follow(production[c1][0]);
}
else
{
// Recursion to the next symbol
// in case we encounter a "#"
followfirst(production[c1][c2], c1, c2+1);
}
}
j++;
}
}
}
```

SAMPLE INPUT AND OUTPUT



```
Enter the productions(type 'end' at the last of the production)
E->TA
A->+TA
A->ε
T->FB
B->*FB
B->ε
F->(E)
F->i
end
E       (i      $)
A       +ε      $)
T       (i      +$)
B       *ε      +$)
F       (i      *+$)
```

RESULT

Thus, the code for first and follow was executed successfully.

Experiment -6

Predictive Parsing Table

Aim:

To write a program for a Predictive Parsing table.

Algorithm:

For the production A → α of Grammar G.

- For each terminal, a in FIRST (□) add A → α to M [A, a].
- If ε is in FIRST (α), and b is in FOLLOW (A), then add A → α to M[A, b].
- If ε is in FIRST (α), and $ is in FOLLOW (A), then add A → α to M[A, $].
- All remaining entries in Table M are errors.

Program:

```
#include <stdio.h>
#include <string.h>
char prol[7][10] = { "S", "A", "A", "B", "B", "C", "C" };
char pror[7][10] = { "A", "Bb", "Cd", "aB", "@", "Cc", "@" };
char prod[7][10] = { "S->A", "A->Bb", "A->Cd", "B->aB", "B->@", "C->Cc", "C-
>@" };
char first[7][10] = { "abcd", "ab", "cd", "a@", "@", "c@", "@" };
char follow[7][10] = { "$", "$", "$", "a$", "b$", "c$", "d$" };
char table[5][6][10];
int numr(char c)
{
 switch (c)
 {
 case 'S':
 return 0;
 case 'A':
 return 1;
 case 'B':
 return 2;
 case 'C':
 return 3;
 case 'a':
 return 0;
 case 'b':
```

```
 return 1;
 case 'c':
 return 2;
 case 'd':
 return 3;
 case '$':
 return 4;
 }
 return (2);
}
int main()
{
 int i, j, k;
 for (i = 0; i < 5; i++)
 for (j = 0; j < 6; j++)
 strcpy(table[i][j], " ");
 printf("The following grammar is used for Parsing Table:\n");
 for (i = 0; i < 7; i++)
 printf("%s\n", prod[i]);
 printf("\nPredictive parsing table:\n");
 fflush(stdin);
 for (i = 0; i < 7; i++)
 {
 k = strlen(first[i]);
 for (j = 0; j < 10; j++)
 if (first[i][j] != '@')
 strcpy(table[numr(prol[i][0]) + 1][numr(first[i][j]) + 1], prod[i]);
 }
 for (i = 0; i < 7; i++)
 {
 if (strlen(pror[i]) == 1)
 {
 if (pror[i][0] == '@')
 {
 k = strlen(follow[i]);
 for (j = 0; j < k; j++)
 strcpy(table[numr(prol[i][0]) + 1][numr(follow[i][j]) + 1], prod[i]);
 }
 }
 }
 strcpy(table[0][0], " ");
 strcpy(table[0][1], "a");
 strcpy(table[0][2], "b");
 strcpy(table[0][3], "c");
 strcpy(table[0][4], "d");
 strcpy(table[0][5], "$");
 strcpy(table[1][0], "S");
 strcpy(table[2][0], "A");
 strcpy(table[3][0], "B");
 strcpy(table[4][0], "C");
 printf("\n-------------------------------------------------------- \n");
 for (i = 0; i < 5; i++)
 for (j = 0; j < 6; j++)
 {
 printf("%-10s", table[i][j]);
 if (j == 5)
 printf("\n-------------------------------------------------------- \n");
 }
}
```

Result: The implementation and creation of predictive parse table using c was executed successfully.

Experiment -7

Shift Reduce Parsing

Aim:
To write a program to implement Lexical Analysis using C.

Algorithm:

- Shift reduce parsing is a process of reducing a string to the start symbol of a grammar.
- Shift reduce parsing uses a stack to hold the grammar and an input tape to hold the string.
- Shift reduce parsing performs the two actions: shift and reduce. That's why it is known as shift reduce parsing.
- At the shift action, the current symbol in the input string is pushed to a stack.
- At each reduction, the symbols will be replaced by the non-terminals. The symbol is the right side of the production and non-terminal is the left side of the production.

Program:

```
#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 ");
 gets(a);
 c=strlen(a);
 strcpy(act,"SHIFT->");
 puts("stack \t input \t action");
 for(k=0,i=0; j<c; k++,i++,j++)
 {
 if(a[j]=='i' && a[j+1]=='d')
 {
 stk[i]=a[j];
 stk[i+1]=a[j+1];
 stk[i+2]='\0';
 a[j]=' ';
 a[j+1]=' ';
 printf("\n$%s\t%s$\t%sid",stk,a,act);
 check();
 }
 else
 {
 stk[i]=a[j];
 stk[i+1]='\0';
 a[j]=' ';
 printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
 check();
 }
 }
 }
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;
}
}
```

Result:

The implementation of shift reduce parsing was executed and verified successfully.

EXPERIMENT 8

LEADING AND TRAILING

AIM

To write a program to implement Leading and Trailing using C.

ALGORITHM

1. For Leading, check for the first non-terminal.
2. If found, print it.
3. Look for next production for the same non-terminal.
4. If not found, recursively call the procedure for the single non-terminal present before the comma or End Of Production String.
5. Include it's results in the result of this non-terminal.
6. For trailing, we compute same as leading but we start from the end of the production to the beginning.
7. Stop

CODE

```
#include<iostream>
#include<conio.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
using namespace std;

int vars,terms,i,j,k,m,rep,count,temp=-1;
char var[10],term[10],lead[10][10],trail[10][10];
struct grammar
{
        int prodno;
        char lhs,rhs[20][20];
}gram[50];
void get()
{
        cout<<"\nLEADING AND TRAILING\n";
        cout<<"\nEnter the no. of variables : ";
        cin>>vars;
```

```
		cout<<"\nEnter the variables : \n";
		for(i=0;i<vars;i++)
		{
			cin>>gram[i].lhs;
			var[i]=gram[i].lhs;
		}
		cout<<"\nEnter the no. of terminals : ";
		cin>>terms;
		cout<<"\nEnter the terminals : ";
		for(j=0;j<terms;j++)
			cin>>term[j];
		cout<<"\nPRODUCTION DETAILS\n";
		for(i=0;i<vars;i++)
		{
			cout<<"\nEnter the no. of production of "<<gram[i].lhs<<":";
			cin>>gram[i].prodno;
			for(j=0;j<gram[i].prodno;j++)
			{
				cout<<gram[i].lhs<<"->";
				cin>>gram[i].rhs[j];
			}
		}
}
void leading()
{
		for(i=0;i<vars;i++)
		{
			for(j=0;j<gram[i].prodno;j++)
			{
				for(k=0;k<terms;k++)
				{
					if(gram[i].rhs[j][0]==term[k])
						lead[i][k]=1;
					else
					{
						if(gram[i].rhs[j][1]==term[k])
							lead[i][k]=1;
					}
				}
			}
```

```
        }
        for(rep=0;rep<vars;rep++)
        {
                for(i=0;i<vars;i++)
                {
                        for(j=0;j<gram[i].prodno;j++)
                        {
                                for(m=1;m<vars;m++)
                                {
                                        if(gram[i].rhs[j][0]==var[m])
                                        {
                                                temp=m;
                                                goto out;
                                        }
                                }
                                out:
                                for(k=0;k<terms;k++)
                                {
                                        if(lead[temp][k]==1)
                                                lead[i][k]=1;
                                }
                        }
                }
        }
}
void trailing()
{
        for(i=0;i<vars;i++)
        {
                for(j=0;j<gram[i].prodno;j++)
                {
                        count=0;
                        while(gram[i].rhs[j][count]!='\x0')
                                count++;
                        for(k=0;k<terms;k++)
                        {
                                if(gram[i].rhs[j][count-1]==term[k])
                                        trail[i][k]=1;
                                else
                                {
                                        if(gram[i].rhs[j][count-2]==term[k])
                                                trail[i][k]=1;
                                }
                        }
                }
        }
        for(rep=0;rep<vars;rep++)
        {
                for(i=0;i<vars;i++)
                {
                        for(j=0;j<gram[i].prodno;j++)
                        {
                                count=0;
                                while(gram[i].rhs[j][count]!='\x0')
                                        count++;
                                for(m=1;m<vars;m++)
                                {
                                        if(gram[i].rhs[j][count-1]==var[m])
                                                temp=m;
                                }
                                for(k=0;k<terms;k++)
                                {
                                        if(trail[temp][k]==1)
                                                trail[i][k]=1;
                                }
                        }
                }
        }
}
void display()
{
        for(i=0;i<vars;i++)
        {
                cout<<"\nLEADING("<<gram[i].lhs<<") = ";
                for(j=0;j<terms;j++)
                {
                        if(lead[i][j]==1)
                                cout<<term[j]<<",";
                }
        }
        cout<<endl;
        for(i=0;i<vars;i++)
        {
                cout<<"\nTRAILING("<<gram[i].lhs<<") = ";
                for(j=0;j<terms;j++)
                {
                        if(trail[i][j]==1)
                                cout<<term[j]<<",";
                }
        }
}
int main()
{

        get();
        leading();
        trailing();
        display();

}

#include<iostream>
#include<conio.h>
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
using namespace std;

int vars,terms,i,j,k,m,rep,count,temp=-1;
char var[10],term[10],lead[10][10],trail[10][10];
struct grammar
{
        int prodno;
        char lhs,rhs[20][20];
}gram[50];
void get()
{
        cout<<"\nLEADING AND TRAILING\n";
        cout<<"\nEnter the no. of variables : ";
        cin>>vars;
```

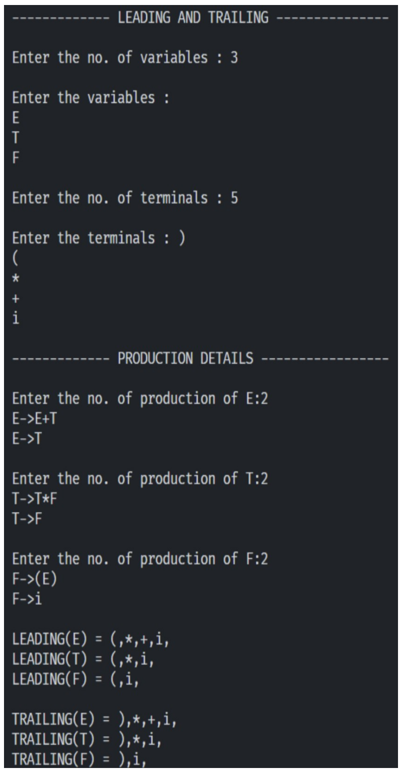
```
	cout<<"\nEnter the variables : \n";
	for(i=0;i<vars;i++)
	{
		cin>>gram[i].lhs;
		var[i]=gram[i].lhs;
	}
	cout<<"\nEnter the no. of terminals : ";
	cin>>terms;
	cout<<"\nEnter the terminals : ";
	for(j=0;j<terms;j++)
		cin>>term[j];
	cout<<"\nPRODUCTION DETAILS\n";
	for(i=0;i<vars;i++)
	{
		cout<<"\nEnter the no. of production of "<<gram[i].lhs<<":";
		cin>>gram[i].prodno;
		for(j=0;j<gram[i].prodno;j++)
		{
			cout<<gram[i].lhs<<"->";
			cin>>gram[i].rhs[j];
		}
	}
}
void leading()
{
	for(i=0;i<vars;i++)
	{
		for(j=0;j<gram[i].prodno;j++)
		{
			for(k=0;k<terms;k++)
			{
				if(gram[i].rhs[j][0]==term[k])
					lead[i][k]=1;
				else
				{
					if(gram[i].rhs[j][1]==term[k])
						lead[i][k]=1;
				}
			}
		}
```

```
        }
        for(rep=0;rep<vars;rep++)
        {
                for(i=0;i<vars;i++)
                {
                        for(j=0;j<gram[i].prodno;j++)
                        {
                                for(m=1;m<vars;m++)
                                {
                                        if(gram[i].rhs[j][0]==var[m])
                                        {
                                                temp=m;
                                                goto out;
                                        }
                                }
                                out:
                                for(k=0;k<terms;k++)
                                {
                                        if(lead[temp][k]==1)
                                                lead[i][k]=1;
                                }
                        }
                }
        }
}
void trailing()
{
        for(i=0;i<vars;i++)
        {
                for(j=0;j<gram[i].prodno;j++)
                {
                        count=0;
                        while(gram[i].rhs[j][count]!='\x0')
                                count++;
                        for(k=0;k<terms;k++)
                        {
                                if(gram[i].rhs[j][count-1]==term[k])
                                        trail[i][k]=1;
                                else
                                {
```

```
                    if(gram[i].rhs[j][count-2]==term[k])
                        trail[i][k]=1;
                }
            }
        }
    }
    for(rep=0;rep<vars;rep++)
    {
        for(i=0;i<vars;i++)
        {
            for(j=0;j<gram[i].prodno;j++)
            {
                count=0;
                while(gram[i].rhs[j][count]!='\x0')
                    count++;
                for(m=1;m<vars;m++)
                {
                    if(gram[i].rhs[j][count-1]==var[m])
                        temp=m;
                }
                for(k=0;k<terms;k++)
                {
                    if(trail[temp][k]==1)
                        trail[i][k]=1;
                }
            }
        }
    }
}
void display()
{
    for(i=0;i<vars;i++)
    {
        cout<<"\nLEADING("<<gram[i].lhs<<") = ";
        for(j=0;j<terms;j++)
        {
            if(lead[i][j]==1)
                cout<<term[j]<<",";
        }
    }
```

```
        cout<<endl;
        for(i=0;i<vars;i++)
        {
                cout<<"\nTRAILING("<<gram[i].lhs<<") = ";
                for(j=0;j<terms;j++)
                {
                        if(trail[i][j]==1)
                                cout<<term[j]<<",";
                }
        }
}
int main()
{       get();
        leading();
        trailing();
        display();}
```

SAMPLE INPUT AND OUTPUT



```
------------ LEADING AND TRAILING --------------

Enter the no. of variables : 3

Enter the variables :
E
T
F

Enter the no. of terminals : 5

Enter the terminals : )
(
*
+
i

------------ PRODUCTION DETAILS ----------------

Enter the no. of production of E:2
E->E+T
E->T

Enter the no. of production of T:2
T->T*F
T->F

Enter the no. of production of F:2
F->(E)
F->i

LEADING(E) = (,*,+,i,
LEADING(T) = (,*,i,
LEADING(F) = (,i,

TRAILING(E) = ),*,+,i,
TRAILING(T) = ),*,i,
TRAILING(F) = ),i,
```

RESULT

Thus, the program for shift reduce parser was executed successfully.

Experiment -9

Computation of LR[0]

Aim:
To write a program to implement LR(0) items.

Algorithm:
1. Start.
2. Create structure for production with LHS and RHS.
3. Open file and read input from file.
4. Build state 0 from extra grammar Law S' -> S $ that is all start symbol of grammar and one Dot (.) before S symbol.
5. If Dot symbol is before a non-terminal, add grammar laws that this nonterminal is in Left Hand Side of that Law and set Dot in before of first part of Right Hand Side.
6. If state exists (a state with this Laws and same Dot position), use that instead.
7. Now find set of terminals and non-terminals in which Dot exist in before.
8. If step 7 Set is non-empty go to 9, else go to 10.
9. For each terminal/non-terminal in set step 7 create new state by using all grammar law that Dot position is before of that terminal/non-terminal in reference state by increasing Dot point to next part in Right Hand Side of that laws.
10. Go to step 5.
11. End of state building.
12. Display the output.
13. End.

Program:
```
#include<iostream>
#include<conio.h>
#include<string.h>
using namespace std;
char prod[20][20],listofvar[26]="ABCDEFGHIJKLMNOPQR";
int novar=1,i=0,j=0,k=0,n=0,m=0,arr[30];
int noitem=0;
struct Grammar
{
char lhs;
char rhs[8];
}g[20],item[20],clos[20][10];
```

```
int isvariable(char variable)
{
for(int i=0;i<novar;i++)
if(g[i].lhs==variable)
return i+1;
return 0;
}
void findclosure(int z, char a)
{
int n=0,i=0,j=0,k=0,l=0;
for(i=0;i<arr[z];i++)
{
for(j=0;j<strlen(clos[z][i].rhs);j++)
{
if(clos[z][i].rhs[j]=='.' && clos[z][i].rhs[j+1]==a)
{
clos[noitem][n].lhs=clos[z][i].lhs;
strcpy(clos[noitem][n].rhs,clos[z][i].rhs);
char temp=clos[noitem][n].rhs[j];
clos[noitem][n].rhs[j]=clos[noitem][n].rhs[j+1];
clos[noitem][n].rhs[j+1]=temp;
n=n+1;}}}
for(i=0;i<n;i++)
{
for(j=0;j<strlen(clos[noitem][i].rhs);j++)
{
if(clos[noitem][i].rhs[j]=='.' && isvariable(clos[noitem]
[i].rhs[j+1])>0)
{
for(k=0;k<novar;k++)
{
if(clos[noitem][i].rhs[j+1]==clos[0][k].lhs)
{
for(l=0;l<n;l++)
if(clos[noitem][l].lhs==clos[0][k].lhs
&& strcmp(clos[noitem][l].rhs,clos[0][k].rhs)==0)
break;
if(l==n)
{
clos[noitem][n].lhs=clos[0][k].lhs;
strcpy(clos[noitem][n].rhs,clos[0][k].rhs);
n=n+1;}}}}}}
arr[noitem]=n;
int flag=0;
for(i=0;i<noitem;i++)
{
if(arr[i]==n)
{
for(j=0;j<arr[i];j++)
{
int c=0;
for(k=0;k<arr[i];k++)
if(clos[noitem][k].lhs==clos[i][k].lhs &&
strcmp(clos[noitem][k].rhs,clos[i][k].rhs)==0)
c=c+1;
if(c==arr[i])
{
flag=1;
goto exit;
}
}
}
}
exit:;
if(flag==0)
arr[noitem++]=n;
}
int main()
{
cout<<"ENTER THE PRODUCTIONS OF THE GRAMMAR(0 TO END) :
\n";
do
{
cin>>prod[i++];
}while(strcmp(prod[i-1],"0")!=0);
for(n=0;n<i-1;n++)
{ m=
0;
j=novar;
g[novar++].lhs=prod[n][0];
```

```
for(k=3;k<strlen(prod[n]);k++)
{
if(prod[n][k] != '|')
g[j].rhs[m++]=prod[n][k];
if(prod[n][k]=='|')
{
g[j].rhs[m]='\0';
m=0;
j=novar;
g[novar++].lhs=prod[n][0];}}}
for(i=0;i<26;i++)
if(!isvariable(listofvar[i]))
break;
g[0].lhs=listofvar[i];
char temp[2]={g[1].lhs,'\0'};
strcat(g[0].rhs,temp);
cout<<"\n\n augumented grammar \n";
for(i=0;i<novar;i++)
cout<<endl<<g[i].lhs<<"->"<<g[i].rhs<<" ";
for(i=0;i<novar;i++)
{
clos[noitem][i].lhs=g[i].lhs;
strcpy(clos[noitem][i].rhs,g[i].rhs);
if(strcmp(clos[noitem][i].rhs,"ε")==0)
strcpy(clos[noitem][i].rhs,".");
else
{
for(int j=strlen(clos[noitem][i].rhs)+1;j>=0;j--)
clos[noitem][i].rhs[j]=clos[noitem][i].rhs[j-1];
clos[noitem][i].rhs[0]='.';
}
}
arr[noitem++]=novar;
for(int z=0;z<noitem;z++)
{
char list[10];
int l=0;
for(j=0;j<arr[z];j++)
{
for(k=0;k<strlen(clos[z][j].rhs)-1;k++)
{
if(clos[z][j].rhs[k]=='.')
{
for(m=0;m<l;m++)
if(list[m]==clos[z][j].rhs[k+1])
break;
if(m==l)
list[l++]=clos[z][j].rhs[k+1];}}}
for(int x=0;x<l;x++)
findclosure(z,list[x]);}
cout<<"\n THE SET OF ITEMS ARE \n\n";
for(int z=0; z<noitem; z++)
{
cout<<"\n I"<<z<<"\n\n";
for(j=0;j<arr[z];j++)
cout<<clos[z][j].lhs<<"->"<<clos[z][j].rhs<<"\n";}}
```

Input:

```
E->E+T
E->T
T->T*F
T->F
F->(E)
F->i
```

Result:

The program for computation of LR[0] was successfully compiled and run.

EXPERIMENT 10

Intermediate code generation – Postfix, Prefix

Aim:

A program to implement Intermediate code generation – Postfix, Prefix.

Algorithm:-

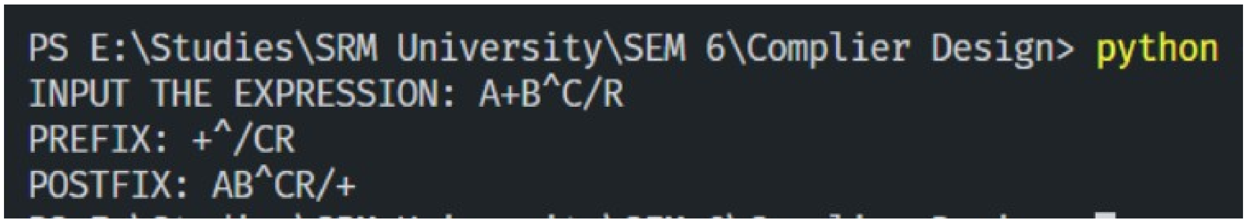
1. Declare a set of operators.
2. Initialize an empty stack.
3. To convert INFIX to POSTFIX follow the following steps
4. Scan the infix expression from left to right.
5. If the scanned character is an operand, output it.
6. Else, If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty or the stack contains a '('), push it.
7. Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack.
8. If the scanned character is an '(', push it to the stack.
9. If the scanned character is an ')', pop the stack and output it until a '(' is encountered, and discard both the parenthesis.
10. Pop and output from the stack until it is not empty.
11. To convert INFIX to PREFIX follow the following steps
12. First, reverse the infix expression given in the problem.
13. Scan the expression from left to right.
14. Whenever the operands arrive, print them.
15. If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
16. Repeat steps 6 to 9 until the stack is empty

Program:

```
OPERATORS = set(['+', '-', '*', '/', '(', ')'])
PRI = {'+': 1, '-': 1, '*': 2, '/': 2}
### INFIX ===> POSTFIX ###
def infix_to_postfix(formula):
stack = [] # only pop when the coming op has priority
output = ''
for ch in formula:
if ch not in OPERATORS:
output += ch
elif ch == '(':
stack.append('(')
elif ch == ')':
while stack and stack[-1] != '(':
output += stack.pop()
stack.pop() # pop '('
else:
while stack and stack[-1] != '(' and PRI[ch] <= PRI[stack[-1]]:
output += stack.pop()
stack.append(ch)
# leftover
while stack:
output += stack.pop()
print(f'POSTFIX:  {output}')
return output
### INFIX ===> PREFIX ###
def infix_to_prefix(formula):
op_stack = []
exp_stack =  []
for ch in formula:
if not ch in OPERATORS:
exp_stack.append(ch)
elif ch == '(':
op_stack.append(ch)
elif ch == ')':
while op_stack[-1] != '(':
op = op_stack.pop()
a = exp_stack.pop()
b = exp_stack.pop()
exp_stack.append(op + b + a)
op_stack.pop() # pop '('
else:
while op_stack and op_stack[-1] != '(' and PRI[ch] <= PRI[op_stack[-1]]:
op = op_stack.pop()
a = exp_stack.pop()
b = exp_stack.pop()
exp_stack.append(op + b + a)
op_stack.append(ch)
#  leftover
while op_stack:
op = op_stack.pop()
a = exp_stack.pop()
```

```
b = exp_stack.pop()
exp_stack.append(op + b + a)
print(f'PREFIX: {exp_stack[-1]}')
return exp_stack[-1]
expres = input("INPUT THE EXPRESSION: ")
pre = infix_to_prefix(expres)
pos = infix_to_postfix(expres)
```

<u>Output:-</u>



```
PS E:\Studies\SRM University\SEM 6\Complier Design> python
INPUT THE EXPRESSION: A+B^C/R
PREFIX: +^/CR
POSTFIX: AB^CR/+
```

<u>Result:-</u>

The program was successfully compiled and run.

EXPERIMENT 11

Intermediate code generation – Quadruple, Triple, Indirect triple

Aim:

Intermediate code generation – Quadruple, Triple, Indirect triple

Algorithm:-

The algorithm takes a sequence of three-address statements as input. For each three address statements of the form a:= b op c perform the various actions. These are as follows: 1. Invoke a function getreg to find out the location L where the result of computation b op c should be stored.
2. Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y' . If the value of y is not already in L then generate the instruction MOV y' , L to place a copy of y in L.
3. Generate the instruction OP z' , L where z' is used to show the current location of z. if z is in both then prefer a register to a memory location. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptors.
4. If the current value of y or z have no next uses or not live on exit from the block or in register then alter the register descriptor to indicate that after execution of x : = y op z those register will no longer contain y or z.

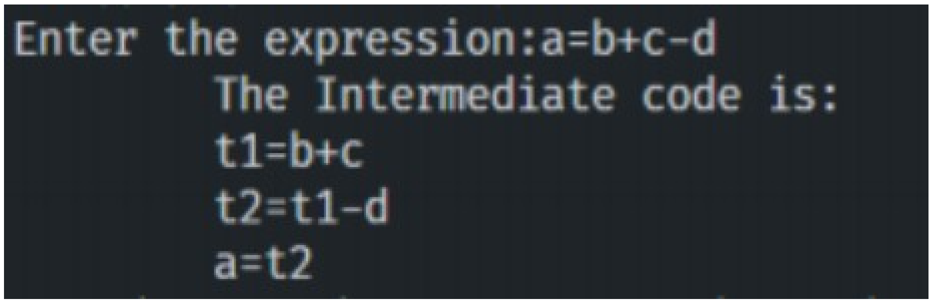
Program:

```
#include<stdio.h>
#include<ctype.h>
#include<stdlib.h>
#include<string.h>
void small();
void dove(int i);
int p[5]={0,1,2,3,4},c=1,i,k,l,m,pi;
char sw[5]={'=','-','+','/','*'},j[20],a[5],b[5],ch[2];
void main()
{
printf("Enter the expression:");
scanf("%s",j);
printf("\tThe Intermediate code is:\n");
small();
}
void dove(int i)
{
```

```
a[0]=b[0]='\0';
if(!isdigit(j[i+2])&&!isdigit(j[i-2]))
{
a[0]=j[i-1];
b[0]=j[i+1];
}
if(isdigit(j[i+2])){
a[0]=j[i-1];
b[0]='t';
b[1]=j[i+2];
}
if(isdigit(j[i-2]))
{ b[0]=j[i+
1];
a[0]='t';
a[1]=j[i-2];
b[1]='\0';
}
if(isdigit(j[i+2]) &&isdigit(j[i-2]))
{ a[0]='
t';
b[0]='t';
a[1]=j[i-2];
b[1]=j[i+2];
sprintf(ch,"%d",c);
j[i+2]=j[i-2]=ch[0];
}
if(j[i]=='*')
printf("\tt%d=%s*%s\n",c,a,b);
if(j[i]=='/')
printf("\tt%d=%s/%s\n",c,a,b);
if(j[i]=='+')
printf("\tt%d=%s+%s\n",c,a,b);if(j[i]=='-')
printf("\tt%d=%s-%s\n",c,a,b); if(j[i]=='=')
printf("\t%c=t%d",j[i-1],--c);
sprintf(ch,"%d",c);
j[i]=ch[0];
c++;
small();
```

```
}
void small()
{
pi=0;l=0;
for(i=0;i<strlen(j);i++)
{
for(m=0;m<5;m++)
if(j[i]==sw[m])
if(pi<=p[m])
{
pi=p[m];
l=1;
k=i;
}
}
if(l==1)
dove(k);
else
exit(0);}
```

Output:-



```
Enter the expression:a=b+c-d
        The Intermediate code is:
        t1=b+c
        t2=t1-d
        a=t2
```

Result:-
The program was successfully compiled and run.

EXPERIMENT 12

Simple Code Generator

Aim:

A program to implement Simple Code Generator

Algorithm:-

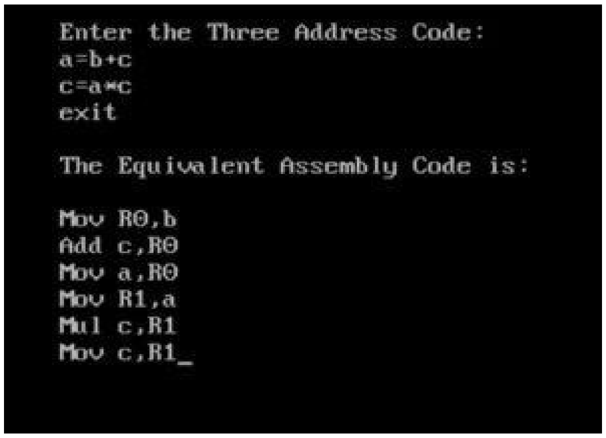
1. Start
2. Get address code sequence.
3. Determine current location of 3 using address (for 1st operand).
4. If the current location does not already exist, generate move (B,O).
5. Update address of A(for 2nd operand).
6. If the current value of B and () is null,exist.
7. If they generate operator () A,3 ADPR.
8. Store the move instruction in memory
9. Stop

Program:

```
#include <stdio.h>
typedef struct
{ char var[10];
int alive;
}
regist;
regist preg[10];
void substring(char exp[],int st,int end)
{ int i,j=0;
char dup[10]="";
for(i=st;i
dup[j++]=exp[i];
dup[j]='0';
strcpy(exp,dup);
} int getregister(char var[])
{ int i;
for(i=0;i<10;i++)
{
if(preg[i].alive==0)
{
strcpy(preg[i].var,var);
break;
}}
```

```
return(i);
}
void getvar(char exp[],char v[])
{ int i,j=0;
char var[10]="";
for(i=0;exp[i]!='\0';i++)
if(isalpha(exp[i]))
var[j++]=exp[i];
else
break;
strcpy(v,var);
}
void main()
{ char basic[10][10],var[10][10],fstr[10],op;
int i,j,k,reg,vc,flag=0;
clrscr();
printf("\nEnter the Three Address Code:\n");
for(i=0;;i++)
{
gets(basic[i]);
if(strcmp(basic[i],"exit")==0)
break;
}
printf("\nThe Equivalent Assembly Code is:\n");
for(j=0;j
{
getvar(basic[j],var[vc++]);
strcpy(fstr,var[vc-1]);
substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
getvar(basic[j],var[vc++]);
reg=getregister(var[vc-1]);
if(preg[reg].alive==0)
{
printf("\nMov R%d,%s",reg,var[vc-1]);
preg[reg].alive=1;
}
op=basic[j][strlen(var[vc-1])];
substring(basic[j],strlen(var[vc-1])+1,strlen(basic[j]));
getvar(basic[j],var[vc++]);
switch(op)
{ case '+': printf("\nAdd"); break;
case '-': printf("\nSub"); break;
case '*': printf("\nMul"); break;
case '/': printf("\nDiv"); break;
}
flag=1;
for(k=0;k<=reg;k++)
{ if(strcmp(preg[k].var,var[vc-1])==0)
{
printf("R%d, R%d",k,reg);
preg[k].alive=0;
flag=0;
break;
}} if(flag)
{
printf(" %s,R%d",var[vc-1],reg);
printf("\nMov %s,R%d",fstr,reg);
}strcpy(preg[reg].var,var[vc-3]);
getch();
}}
```

Output:-



```
Enter the Three Address Code:
a=b+c
c=a*c
exit

The Equivalent Assembly Code is:

Mov R0,b
Add c,R0
Mov a,R0
Mov R1,a
Mul c,R1
Mov c,R1_
```

Result:-
The program was successfully compiled and run.

Experiment -13

Construction of DAG

Aim:
To write a program to implement type checking.

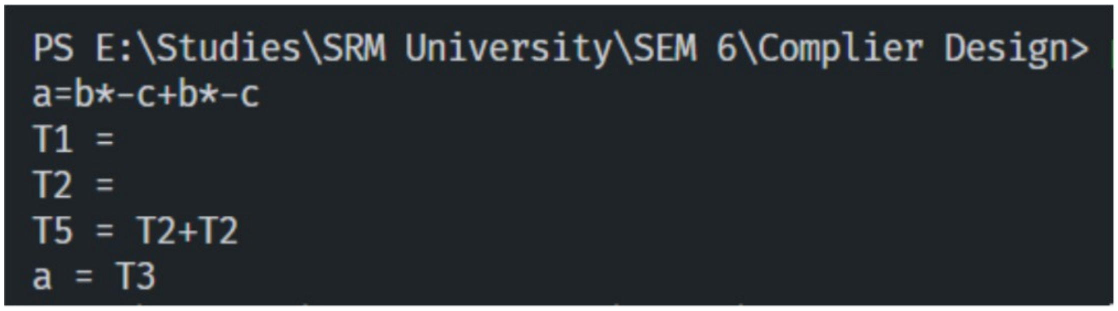
Algorithm:
• Track the global scope type information (e.g. classes and their members)
• Determine the type of expressions recursively, i.e. bottom-up, passing the resulting types upwards.
• If type found correct, do the operation
• Type mismatches, semantic error will be notified

Program:
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
int n,i,k,flag=0;
char vari[15],typ[15],b[15],c;
printf("Enter the number of variables:");
scanf(" %d",&n);
for(i=0;i<n;i++)
{
printf("Enter the variable[%d]:",i);
scanf(" %c",&vari[i]);
printf("Enter the variable-type[%d](float-f,int-i):",i);
scanf(" %c",&typ[i]);
if(typ[i]=='f')
flag=1;
}
printf("Enter the Expression(end with $):");
i=0;
getchar();
while((c=getchar())!='$')
{
b[i]=c;
i++; }
k=i;
for(i=0;i<k;i++)
{
if(b[i]=='/')
{
flag=1;
break; } }
for(i=0;i<n;i++)
{
if(b[0]==vari[i])
{
if(flag==1)
{
if(typ[i]=='f')
{ printf("\nthe datatype is correctly defined..!\n");
break; }
else
{ printf("Identifier %c must be a float type..!\n",vari[i]);
break; } }
else
{ printf("\nthe datatype is correctly defined..!\n");
break; } }
}
return 0;
}
```

Input:

```
Enter the number of variables:4
Enter the variable[0]:A
Enter the variable-type[0](float-f,int-i):i
Enter the variable[1]:B
Enter the variable-type[1](float-f,int-i):i
Enter the variable[2]:C
Enter the variable-type[2](float-f,int-i):f
Enter the variable[3]:D
Enter the variable-type[3](float-f,int-i):i
Enter the Expression(end with $):A=B*C/D
$
Identifier A must be a float type..!
```

SAMPLE INPUT AND OUTPUT



```
PS E:\Studies\SRM University\SEM 6\Complier Design>
a=b*-c+b*-c
T1 =
T2 =
T5 = T2+T2
a = T3
```

Result: The program for implementation of DAG was successfully
compiled and run.

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

Kattankulathur, Chengalpattu District - 603203

18CSC304J/ COMPLIER DESIGN

MINI PROJECT REPORT

CODE GENERATOR

Gudied by:

Dr. M. KOWSIGAN

Submitted By:

Rishabh Singh Sahil (RA2011003011334)
Abhishek Verma (RA2011003011341)
Pranshul Verma (RA2011003011361)

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

(Under Section 3 of UGC Act, 1956)

BONAFIDE CERTIFICATE

Certified that this project report "Mini Python Compiler" is the bonafide work of **RISHABH SINGH SAHIL (RA2011003011334), ABHISHEK VERMA (RA2011003011341)**, and **PRANSHUL VERMA (RA2011003011361)** who carried out the project under my supervision.

SIGNATURE
Compiler Design
FacultyCSE
SRM university of
Science and Technology
Potheri,SRM Nagar,
Kattankulathur
Tamil Nadu 603203

ACKNOWLEDGEMENT

We would like to take this opportunity to express our sincere gratitude to all those who have supported and contributed to the successful completion of our project "**A Website-based Dynamic Visualization of Tokenization and Parsing in the Lexical Analysis Process**".

First and foremost, we extend my heartfelt appreciation to our project supervisors, **Dr. M. Kowsigan** for theirinvaluable guidance, insightful feedback, and unwavering support throughout the project. Her expertise and encouragement have been

instrumental in shaping the project and guiding us in the right direction.

We express our profound gratitude to our Dean (College of Engineering and Technology)

Dr. T. V. Gopal, for bringing out novelty in all executions. We would like to express our

warmth of gratitude to our Registrar **Dr. S. Ponnusamy**, for his encouragement and our

heartfelt thanks to the Chairperson, School of Computing **Dr. Revathi Venkataraman**,

for imparting confidence to complete our course project.

Sincere thanks to all our team members for their commitment, dedication, and hard work in making this project a success. Each team member brought unique skills and perspectives contributing to the project's success. We are grateful for their tireless efforts and support.

Lastly, we express our gratitude to our institution for providing us with the resources and facilities necessary to carry out this project. This project has been a significant learning experience for all of us, and we are grateful for the opportunity to have worked on it.

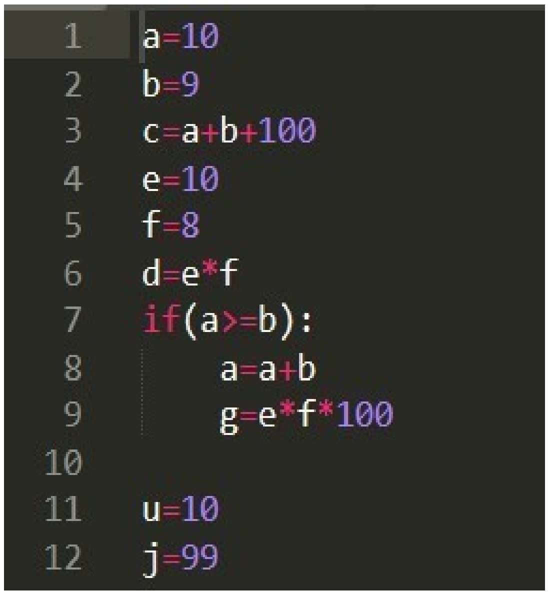
In conclusion, we are grateful to everyone who has been a part of this project, and we look forward to utilizing the knowledge and skills gained during this project in our future endeavors.

ABSTRACT

The Mini-Compiler, contains all phases of compiler has been made for the language Python by using C language (till intermediate code optimization phase) and we used Python language itself for target code generation as well. The constructs that have been focused on are 'if-else' and 'while' statements. The optimizations handled for the intermediate code are 'packing temporaries' and 'constant propagation'. Syntax and semantic errors have been handled and syntax error recovery has been implemented using Panic Mode Recovery in the lexer.

The screenshots of the sample input and target code output are as follows:

Sample Input:

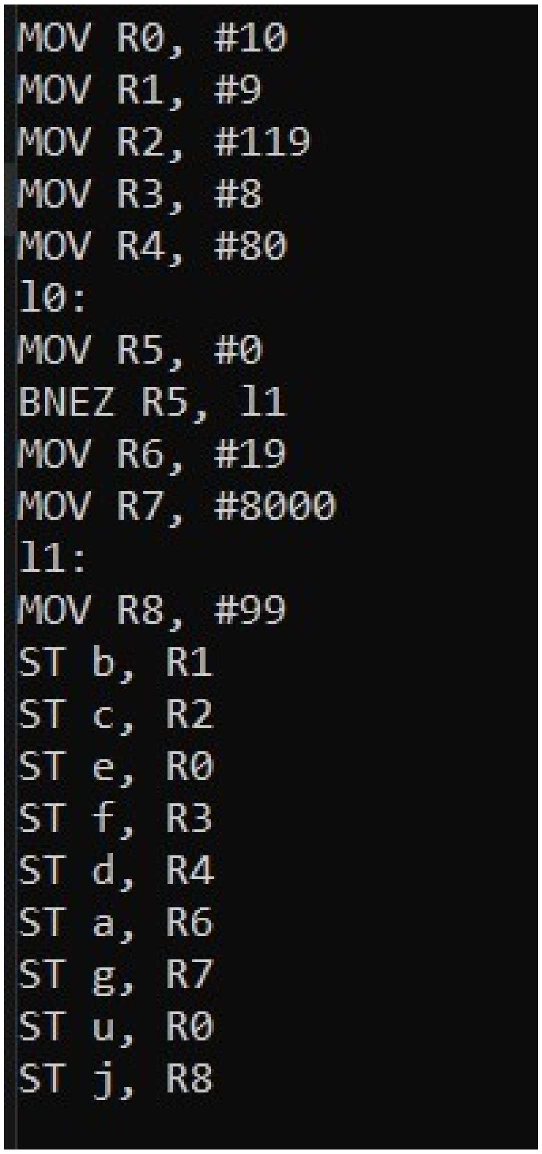


```
a=10
b=9
c=a+b+100
e=10
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Fig. 1.1

Sample Output:

This the target code which is generated after ICG



```
MOV R0, #10
MOV R1, #9
MOV R2, #119
MOV R3, #8
MOV R4, #80
l0:
MOV R5, #0
BNEZ R5, l1
MOV R6, #19
MOV R7, #8000
l1:
MOV R8, #99
ST b, R1
ST c, R2
ST e, R0
ST f, R3
ST d, R4
ST a, R6
ST g, R7
ST u, R0
ST j, R8
```

Fig. 1.2

INTRODUCTION

For this mini-compiler, the following aspects of the Python language syntax have been covered:

- Constructs like ‘if-else’ and ‘while’ and the required indentation for these loops.
- Nested loops
- Integer and float data types

Specific error messages are displayed based on the type of error. Syntax errors are handled using the yyerror () function, while the semantic errors are handled by making a call to a function that searches for a particular identifier in the symbol table. The line number is displayed as part of the error message.

As a part of error recovery, panic mode recovery has been implemented for the lexer. It recovers from errors in variable declaration. In case of identifiers,when the name begins with a digit, the compiler neglects the digit and considers the rest as the identifier name.

Languages used to develop this project:

- C
- YACC
- LEX
- PYTHON

DIFFERENT MODULES OF PROJECT

1. **Different Folders:**
   - **Token and Symbol Table**: This folder contains the code that outputs the tokens and the symbol table.
   - **Abstract Syntax Tree**: This folder contains the code that displays the abstract syntax tree.
   - **Intermediate Code Generation**: This folder contains the code that generates the symbol table before optimizations and the intermediate code.
   - **Optimised ICG:** This folder contains the code that generates the symbol table after optimizations, the quadruples table and the optimized intermediate code.
   - **Target Code**: This folder contains the code that displays the assembly code/target code.

2. **Different Files:**
   - **proj.l**: It is the Lexical analyzer file which defines all the terminals of the productions stated in the yacc file. It contains regular expressions.
   - **proj1.y**: Yacc file is where the productions for the conditional statements like if-else and while and expressions are mentioned. This file also contains the semantic rules defined against every production necessary. Rules for producing three address code is also present.
   - **final.py:** It is the python file which converts the ICG to target code using regex.
   - inp.py: The input python code which will be parsed and checked for semantic correctness by executing the lex and yacc files along with it.

CONTEXT-FREE GRAMMAR

1. REGEX

- digits -> [0-9]
- num -> digits+(\.digits+)?([Ee][+|-]?digits+)?
- id -> [a-zA-Z] [a-zA-Z0-9] *
- integer -> [0-9] +
- string -> [a-z | A-Z | 0-9 | special] *
- special -> [! " # $ % & \ ( ) * + , - . / : ; < = > ? @ [\ \ ] ^ _ ` { | } ~]

2. GRAMMAR

- P ->S
- S ->Simple S | Compound S | epsilon
- Simple ->Assignment LB | Cond LB | Print LB | break | pass | continue Assignment -> id opassgn E1 | id opassgn cond | id listassgnArr | id strassgn Str
- E1 -> E1 op1 E2 | E2
- E2 -> E2 op2 E3 | E3
- E3 -> E4 op3 E3 | E4
- E4 -> num | id | (E1)
- opassgn -> = | /= | *= | += | -= op1 -> + | -
- op2 -> / | *
- op3 -> **
- LB -> \n
- listassgn -> =strassgn -> = | += |-=
- Arr -> [list] | [list] mul | [list] add |mat mat -> [listnum] | [liststr]
- list -> listnum | liststr | Range listnum -> num, listnum | epsilon | num liststr -> Str, liststr | epsilon | Str

- mul -> * integer
- add -> + Arr
- Range -> range (start, stop, step) start -> integer | epsilon
- stop -> integer
- step -> integer | epsilon
- Str -> string | string mul | stringaddstr
- addstr -> + string
- Compound -> if_else LB | while_loop LB
- if_else -> if condition: LB IND else | if condition: LBIND
- | if condition: S | if condition: S else
- else -> else: LB IND | else: S
- while_loop -> while condition: LB IND | while condition: S
- condition -> cond | (cond)
- cond -> cond opor cond1 | cond1 cond1 -> cond1 opand cond2 |cond2 cond2 -> opnot cond2 | cond3
- cond3 -> (cond) | relexp | bool
- relexp -> relexp relop E1 | E1 | id | num relop -> < | > | <= | >= | == |! = | in | not in bool -> True | False
- opor -> || | or
- opand -> && | and
- opnot -> not | ~
- IND -> indent S dedent
- indent -> \t
- dedent -> -\t
- Print -> print (toprint) | print (toprint, sep) | print (toprint,sep,end)
- | print (toprint, end) toprint -> X | X, toprint | epsilon X -> Str | Arr | id | num
- sep -> sep = Str
- end -> end = Str

DESIGN STRATEGY

1. SYMBOL TABLE CREATION

Linked list is being used to create the symbol table. The final output shows the label, value, scope, line number and type. We have created three functions to generate the symbol table. They are:

- **Insert:** It pushes the node onto the linked list.
- **Display:** It displays the symbol table.
- **Search:** It searches for a particular label in the linkedlist.

2. ABSTRACT SYNTAX TREE

This is being implemented using a structure that has three members which hold the data, left pointer and right pointer respectively. The functions that aid in creating and displaying this tree are:

- **Build Tree**: It is used to create a node of this structure and add it to the existing tree.
- **Print Tree**: This function displays the abstract syntax tree using pre-order traversal.

3. INTERMEDIATE CODE GENERATION

We have used the stack data structure to generate the intermediate code that uses some functions, which are called based on some conditions.

4. CODE OPTIMIZATION

A data structure known as quadruple is used to optimize the code. This data structure holds the details of each of the assignment, label and goto statements.

5. ERROR HANDLING

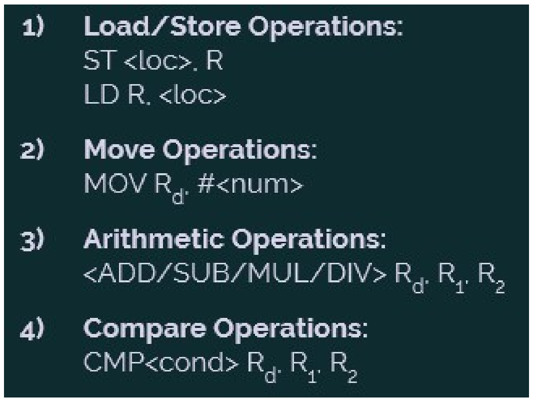
- **Syntax Error**: If the token returned does not satisfy the grammar, then yyerror () is used to display the syntax error along with the line number.

- **Semantic Error:** If there is an identifier in the RHS of an assignment statement, the symbol table is searched for that variable. If the variable does not exist in the symbol table, this is identified as a semantic error and is displayed.
- **Error Recovery:** Panic Mode Recoveryis used asthe error recovery technique, where if the variable declaration has been done with a number at thestart, it ignores the number and considers the rest as the variable name. This has been implemented using regex.

6. TARGET CODE GENERATION

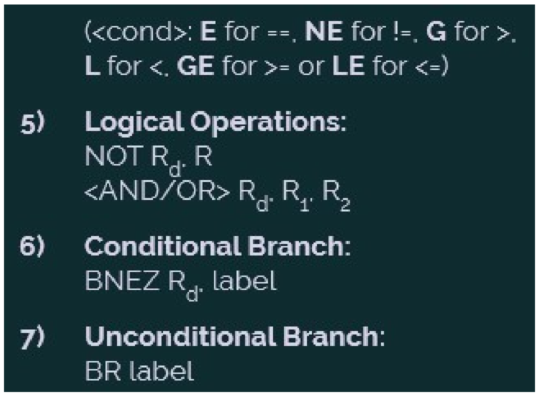
The optimised intermediate code is read from a text file, line after line, and goes through a series of if-else loops to generate the target code. A hypothetical target machine model has been used as the target machine and the limit on the number of reusable registers has been set to 13, numbered from R0 to R12.

A hypothetical machine model has been used that follows the following instruction set architecture:



1) **Load/Store Operations:**
   ST <loc>, R
   LD R, <loc>
2) **Move Operations:**
   MOV R_d, #<num>
3) **Arithmetic Operations:**
   <ADD/SUB/MUL/DIV> R_d, R_1, R_2
4) **Compare Operations:**
   CMP<cond> R_d, R_1, R_2

Fig 2.1



(<cond>: **E** for ==, **NE** for !=, **G** for >, **L** for <, **GE** for >= or **LE** for <=)

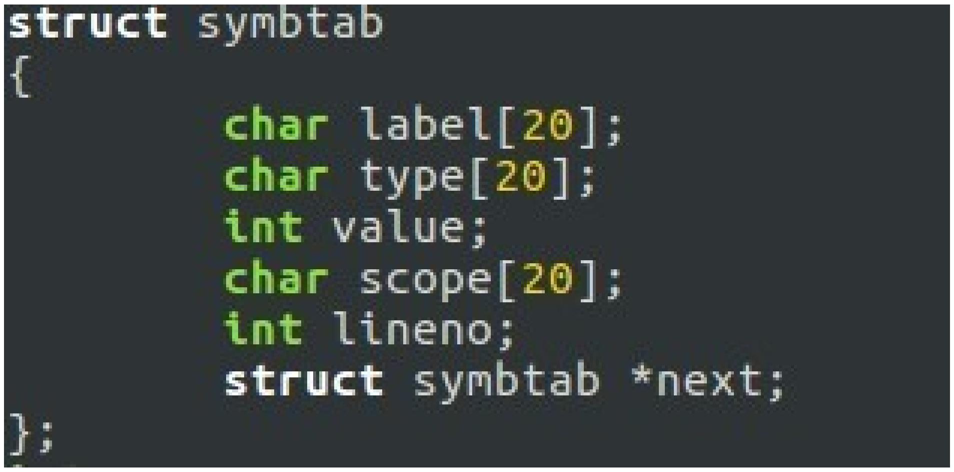
5) **Logical Operations:**
   NOT R_d, R
   <AND/OR> R_d, R_1, R_2
6) **Conditional Branch:**
   BNEZ R_d, label
7) **Unconditional Branch:**
   BR label

Fig 2.2

IMPLEMENTATION DETAILS

SYMBOL TABLE CREATION

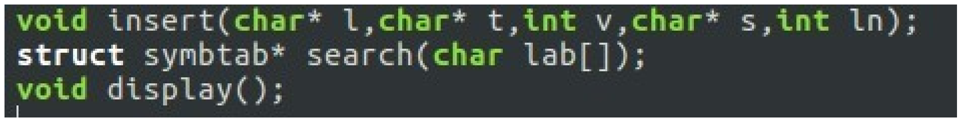
The following snapshot shows the structure declaration for symbol table:



```c
struct symbtab
{
        char label[20];
        char type[20];
        int value;
        char scope[20];
        int lineno;
        struct symbtab *next;
};
```

Fig. 3.1

These are the functions used to generate the symbol table:



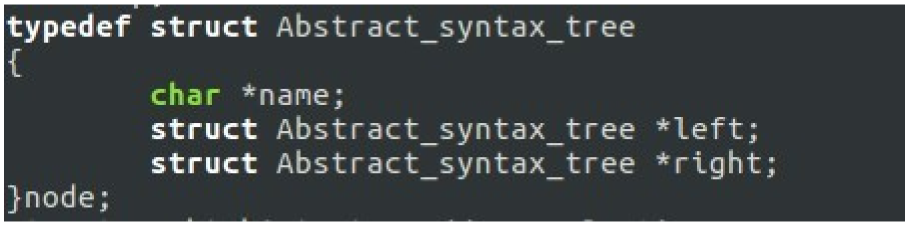
```c
void insert(char* l,char* t,int v,char* s,int ln);
struct symbtab* search(char lab[]);
void display();
```

Fig. 3.2

These snapshots are taken from proj1.y file in Token and Symbol table folder.

ABSTRACT SYNTAX TREE

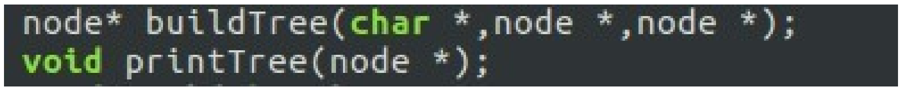
The following data structure is used to represent the abstract syntax tree:



```
typedef struct Abstract_syntax_tree
{
	char *name;
	struct Abstract_syntax_tree *left;
	struct Abstract_syntax_tree *right;
}node;
```

Fig. 3.3

The following functions build and display the syntax tree:



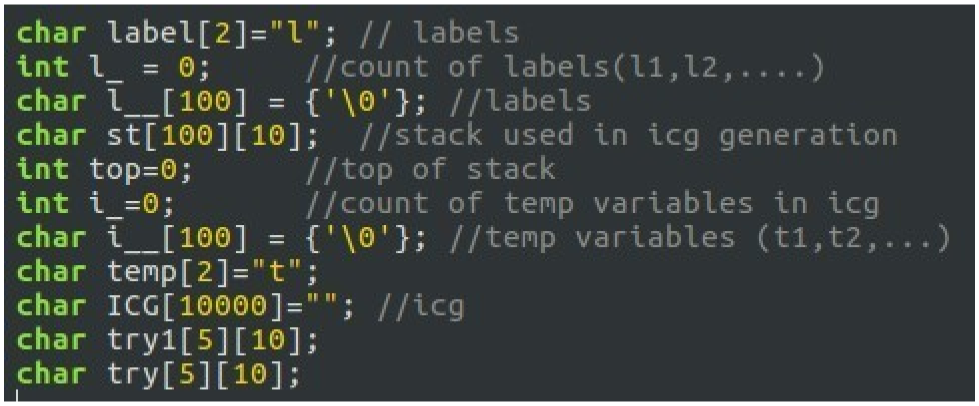
```
node* buildTree(char *,node *,node *);
void printTree(node *);
```

Fig. 3.4

These snapshots are taken from proj1.y file in Abstract syntax tree folder.

INTERMEDIATE CODE GENERATION

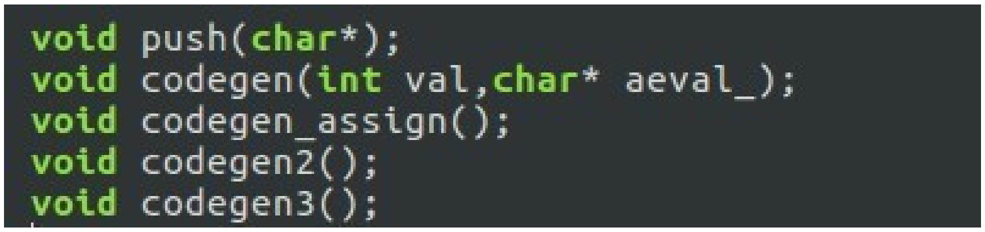
The following arrays act as stacks and are used for the generation of intermediate code:



```
char label[2]="l"; // labels
int l_ = 0;        //count of labels(l1,l2,....)
char l__[100] = {'\0'}; //labels
char st[100][10];  //stack used in icg generation
int top=0;         //top of stack
int i_=0;          //count of temp variables in icg
char i__[100] = {'\0'}; //temp variables (t1,t2,...)
char temp[2]="t";
char ICG[10000]=""; //icg
char try1[5][10];
char try[5][10];
```

Fig. 3.5

The following functions push onto the stack and generate the intermediate code, when called based on various conditions:



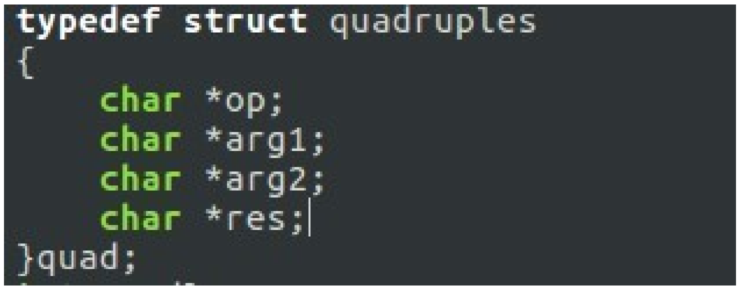
```
void push(char*);
void codegen(int val,char* aeval_);
void codegen_assign();
void codegen2();
void codegen3();
```

Fig. 3.6

These snapshots are taken from proj1.y file in ICG folder.

CODE OPTIMISATION

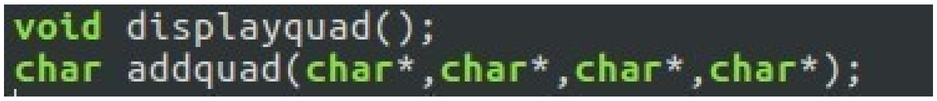
The data structure quadruple declaration has been shown below:



```
typedef struct quadruples
{
    char *op;
    char *arg1;
    char *arg2;
    char *res;
}quad;
```

Fig. 3.7

The following functions are used to add to the quadruples table and display it onto the terminal:



```
void displayquad();
char addquad(char*,char*,char*,char*);
```

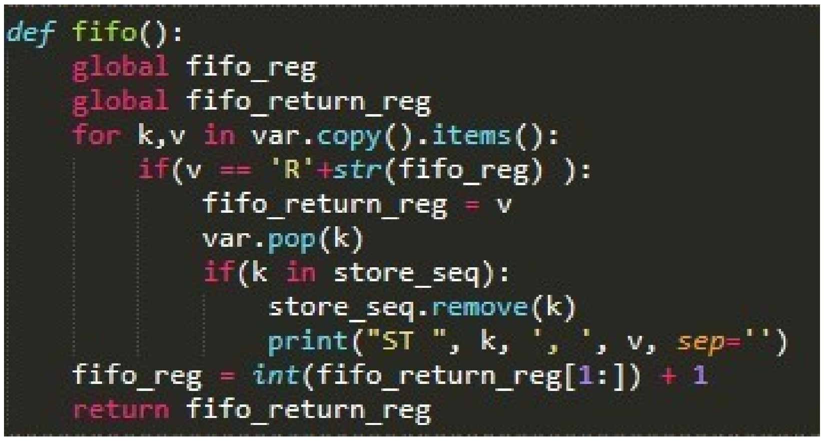
Fig. 3.8

These snapshots are taken from proj1.y file in Optimized ICG folder.

TARGET CODE GENERATION

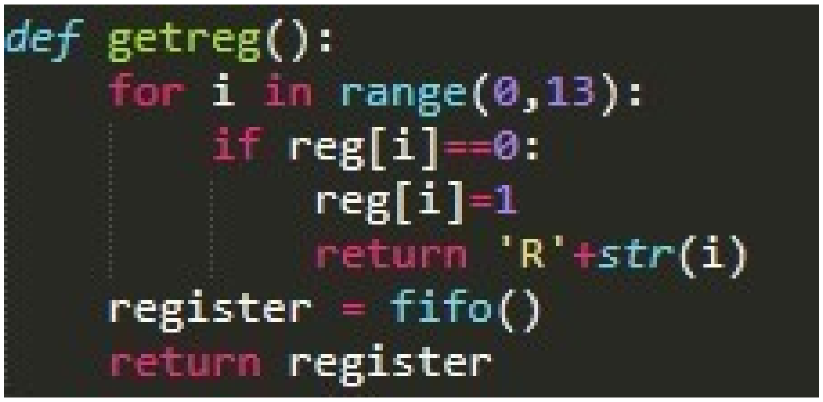
A global dictionary holds the mapping between each constant/identifier and the corresponding register that holds that constant/identifier. There also is a global list that holds the identifiers that need to be stored towards the end of the program.

There are two functions which are used for register allocation. The ‘getreg ()’ function gets the next free/unallocated register and uses the ‘fifo ()’ function in cases when all the registers are used up. The ‘fifo ()’ function uses the ‘First in First Out’ method to free a register and return it to the ‘getreg ()’ function. These functions are as follows:



```
def fifo():
    global fifo_reg
    global fifo_return_reg
    for k,v in var.copy().items():
        if(v == 'R'+str(fifo_reg) ):
            fifo_return_reg = v
            var.pop(k)
            if(k in store_seq):
                store_seq.remove(k)
                print("ST ", k, ', ', v, sep='')
    fifo_reg = int(fifo_return_reg[1:]) + 1
    return fifo_return_reg
```

Fig. 3.9



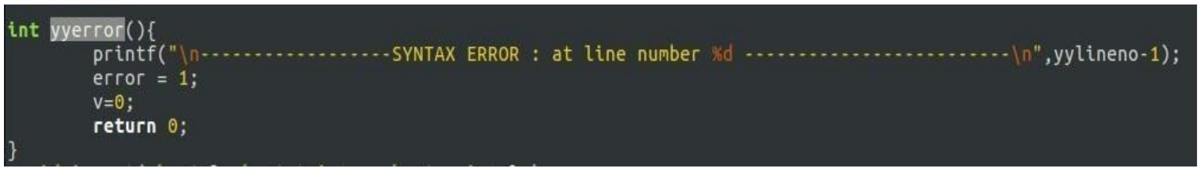
```
def getreg():
    for i in range(0,13):
        if reg[i]==0:
            reg[i]=1
            return 'R'+str(i)
    register = fifo()
    return register
```

Fig. 3.10

These snapshots are taken from final.py file in Target Code folder.

ERROR HANDLING

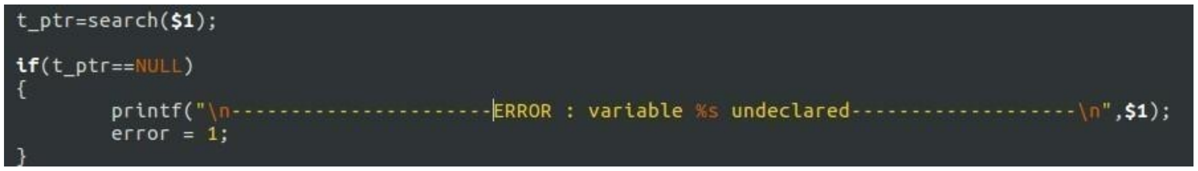
The following snapshot shows the error handling function for syntax errors:



```
int yyerror(){
        printf("\n-----------------SYNTAX ERROR : at line number %d ------------------------\n",yylineno-1);
        error = 1;
        v=0;
        return 0;
}
```

Fig. 3.11

The followingsnapshot shows semantic error handling functionality:

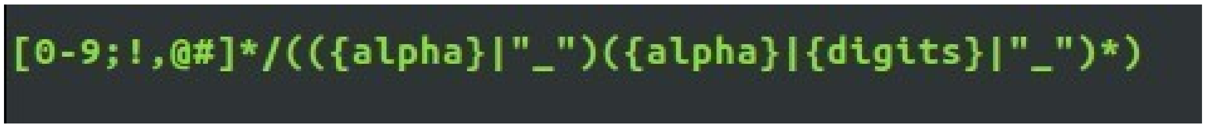


```
t_ptr=search($1);

if(t_ptr==NULL)
{
        printf("\n----------------------ERROR : variable %s undeclared-------------------\n",$1);
        error = 1;
}
```

Fig. 3.12

These above snapshots are taken from proj1.y file in Symbol table folder.

The regex for panic mode recovery implemented in the lexer is as follows:



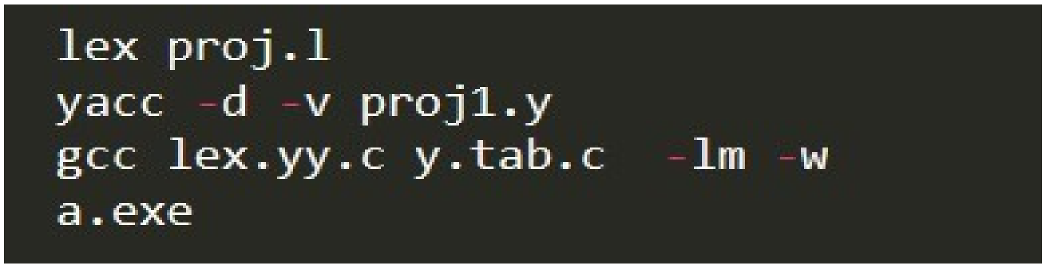
```
[0-9;!,@#]*/(({alpha}|"_")({alpha}|{digits}|"_")*)
```

Fig. 3.13

The above snapshots are taken from proj.l file in Symbol table folder.

BUILD AND RUN THE PROGRAM:

The following screenshot displays what commands need to be executed to build and run the program:



```
lex proj.l
yacc -d -v proj1.y
gcc lex.yy.c y.tab.c  -lm -w
a.exe
```

Fig. 3.14

The above commands need to be executed on the terminal which is inside the project folder that contains the code for the compiler.

RESULTS AND SHORTCOMINGS

The mini-compiler built in this project works perfectly for the ‘if-else’ and ‘while’ constructs of Python language. Our compiler can be executed in different phases by building and running the code separated in the various folders. The final code displays the output of all the phases on the terminal, one after the other. First, the tokens are displayed, followed by a ‘PARSE SUCCESSFUL’ message. Then abstract syntax tree is printed. Next, the symbol table along with the intermediate code is printed without optimization. Finally, the symbol table and the intermediate code after optimization is displayed after the quadruples table. The final output is the target code, written in the instruction set architecture followed by the hypothetical machine model introduced in this project. This is for inputs with no errors. But in case of erroneous inputs, the token generation is stopped on error encounter and the corresponding error message isdisplayed.

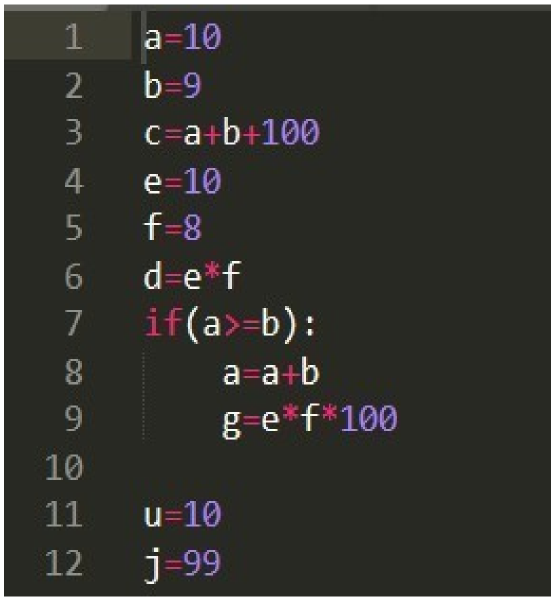
This mini-compiler has the following shortcomings:

- User defined functions are not handled.
- Importing libraries and calling library functions is not taken careof.
- Datatypes other than integer and float, example strings, lists, tuples, dictionaries, etc. have not been considered.
- Constructs other than ‘while’ and ‘if-else’ have not been added in the compiler program.

SNAPSHOTS

TEST CASE 1 (Correct input):

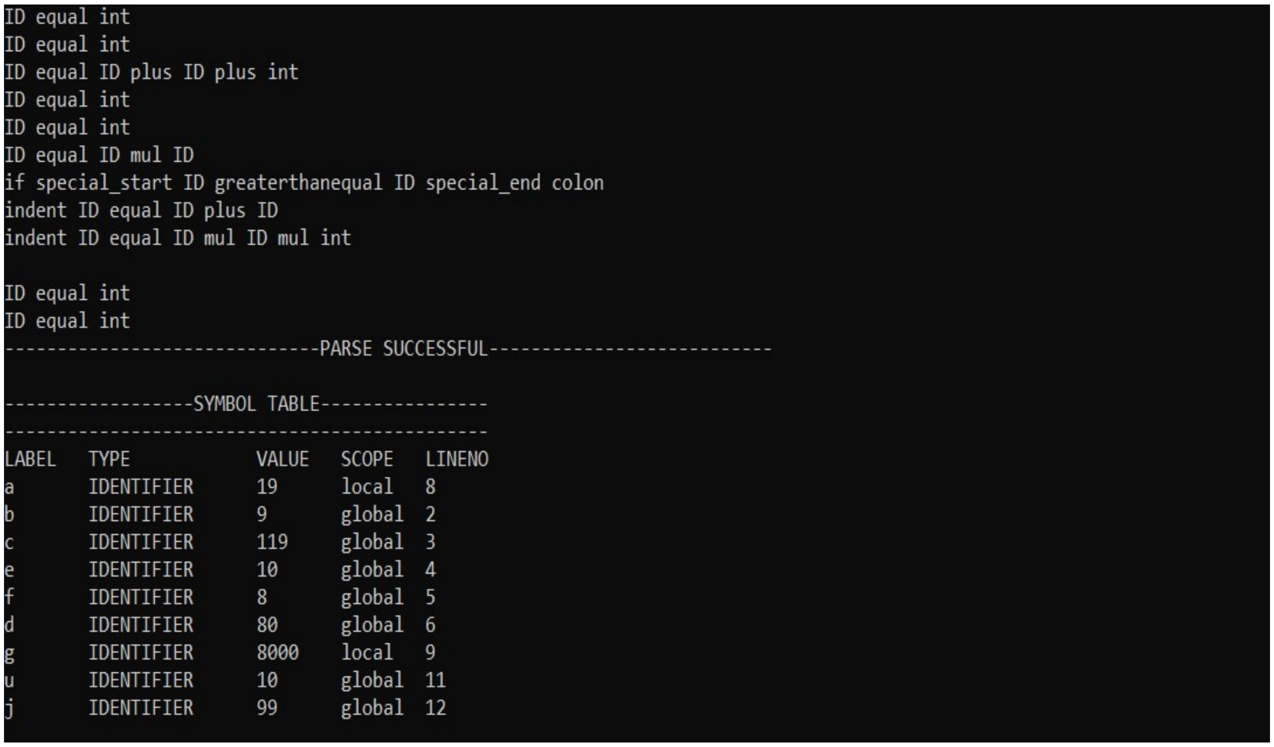
Input:



```
a=10
b=9
c=a+b+100
e=10
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Fig. 4.1

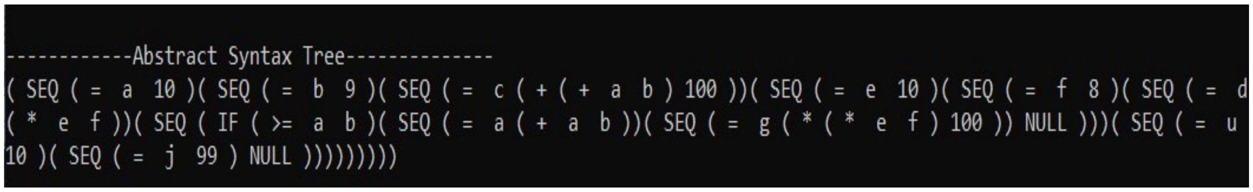
Tokens and Symbol Table:



```
ID equal int
ID equal int
ID equal ID plus ID plus int
ID equal int
ID equal int
ID equal ID mul ID
if special_start ID greaterthanequal ID special_end colon
indent ID equal ID plus ID
indent ID equal ID mul ID mul int

ID equal int
ID equal int
-----------------------------PARSE SUCCESSFUL--------------------------

-----------------SYMBOL TABLE---------------
--------------------------------------------
LABEL   TYPE            VALUE   SCOPE   LINENO
a       IDENTIFIER      19      local   8
b       IDENTIFIER      9       global  2
c       IDENTIFIER      119     global  3
e       IDENTIFIER      10      global  4
f       IDENTIFIER      8       global  5
d       IDENTIFIER      80      global  6
g       IDENTIFIER      8000    local   9
u       IDENTIFIER      10      global  11
j       IDENTIFIER      99      global  12
```

Fig. 4.2

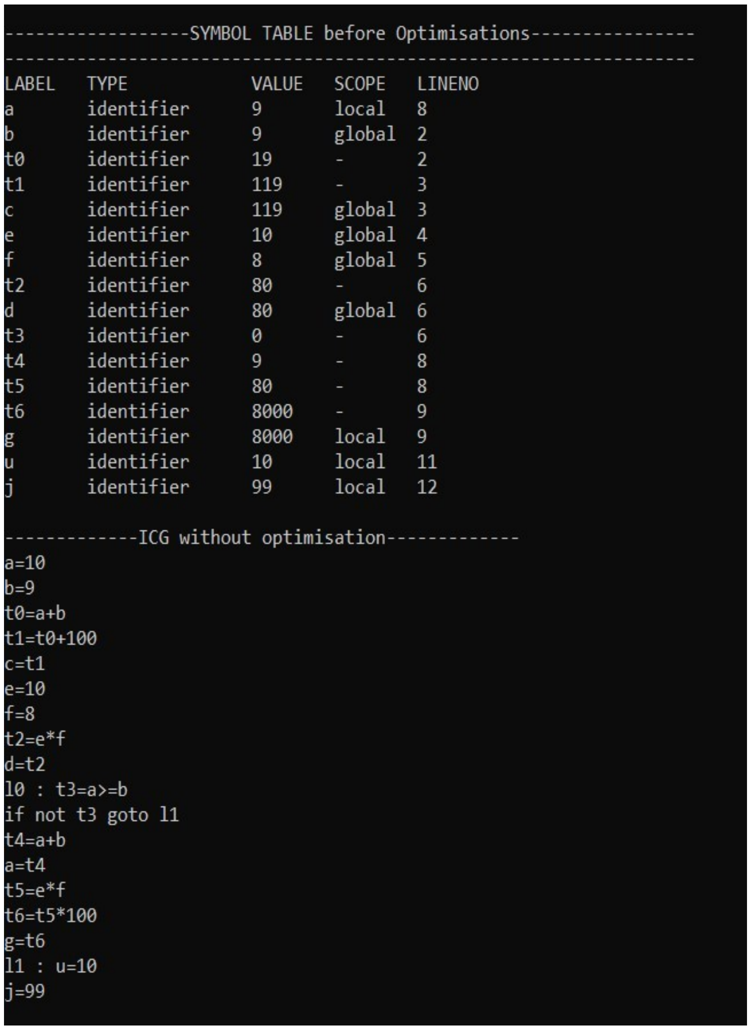
Abstract Syntax Tree:



```
-----------Abstract Syntax Tree-------------
( SEQ ( =  a  10 )( SEQ ( =  b  9 )( SEQ ( =  c ( + ( +  a  b ) 100 ))( SEQ ( =  e  10 )( SEQ ( =  f  8 )( SEQ ( =  d
( *  e  f ))( SEQ ( IF ( >=  a  b )( SEQ ( =  a ( +  a  b ))( SEQ ( =  g ( * ( *  e  f ) 100 )) NULL )))( SEQ ( =  u
10 )( SEQ ( =  j  99 ) NULL ))))))))
```

Fig. 4.3

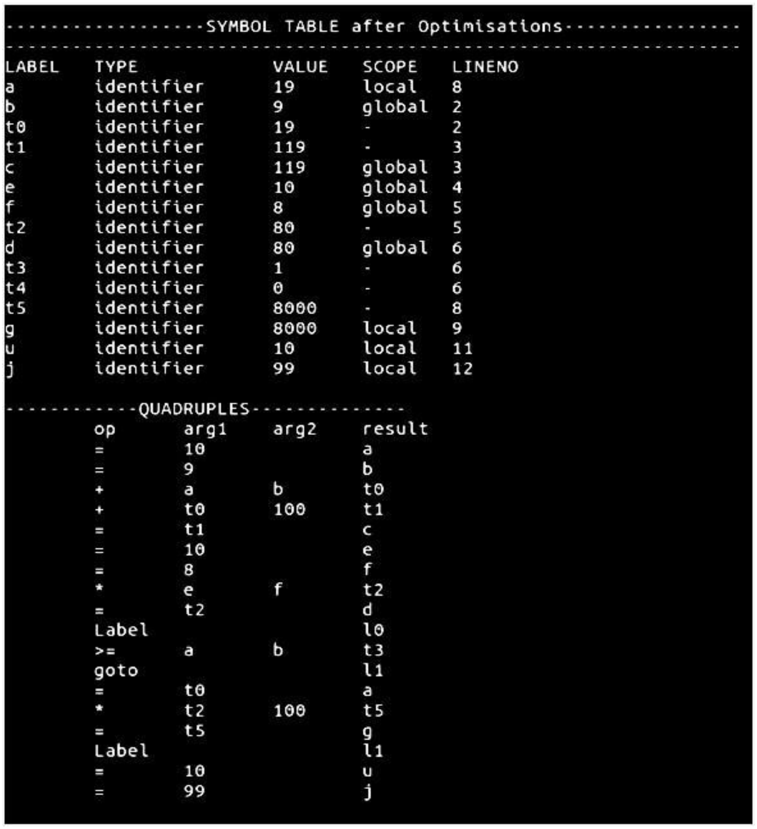
Symbol Table and Unoptimized Intermediate Code:



```
-----------------SYMBOL TABLE before Optimisations---------------
----------------------------------------------------------------
LABEL   TYPE            VALUE   SCOPE   LINENO
a       identifier      9       local   8
b       identifier      9       global  2
t0      identifier      19      -       2
t1      identifier      119     -       3
c       identifier      119     global  3
e       identifier      10      global  4
f       identifier      8       global  5
t2      identifier      80      -       6
d       identifier      80      global  6
t3      identifier      0       -       6
t4      identifier      9       -       8
t5      identifier      80      -       8
t6      identifier      8000    -       9
g       identifier      8000    local   9
u       identifier      10      local   11
j       identifier      99      local   12

------------ICG without optimisation------------
a=10
b=9
t0=a+b
t1=t0+100
c=t1
e=10
f=8
t2=e*f
d=t2
l0 : t3=a>=b
if not t3 goto l1
t4=a+b
a=t4
t5=e*f
t6=t5*100
g=t6
l1 : u=10
j=99
```

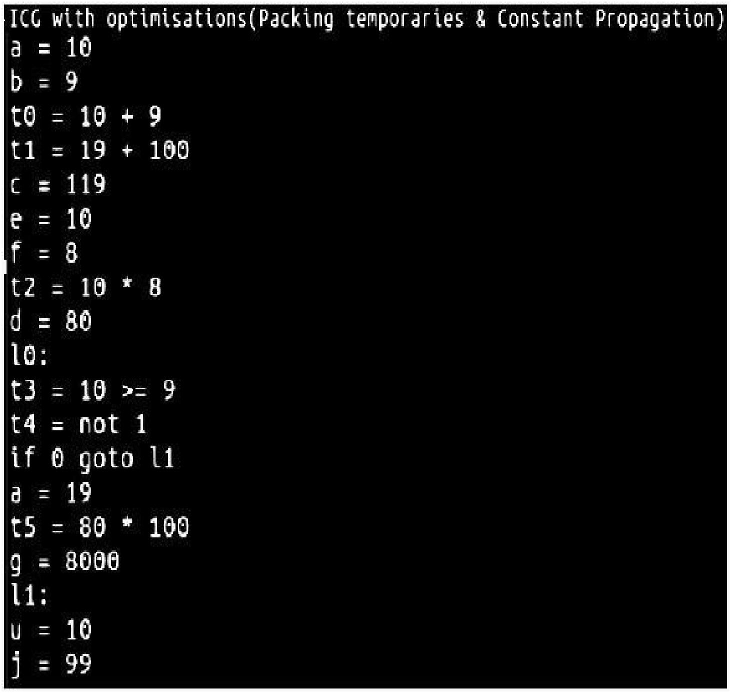
Fig. 4.4

Symbol Table, Quadruples Table and Optimised Intermediate Code:



```
-----------------SYMBOL TABLE after Optimisations---------------
----------------------------------------------------------------
LABEL   TYPE            VALUE   SCOPE   LINENO
a       identifier      19      local   8
b       identifier      9       global  2
t0      identifier      19      -       2
t1      identifier      119     -       3
c       identifier      119     global  3
e       identifier      10      global  4
f       identifier      8       global  5
t2      identifier      80      -       5
d       identifier      80      global  6
t3      identifier      1       -       6
t4      identifier      0       -       6
t5      identifier      8000    -       8
g       identifier      8000    local   9
u       identifier      10      local   11
j       identifier      99      local   12

-----------QUADRUPLES-------------
        op      arg1    arg2    result
        =       10              a
        =       9               b
        +       a       b       t0
        +       t0      100     t1
        =       t1              c
        =       10              e
        =       8               f
        *       e       f       t2
        =       t2              d
        Label                   l0
        >=      a       b       t3
        goto                    l1
        =       t0              a
        *       t2      100     t5
        =       t5              g
        Label                   l1
        =       10              u
        =       99              j
```

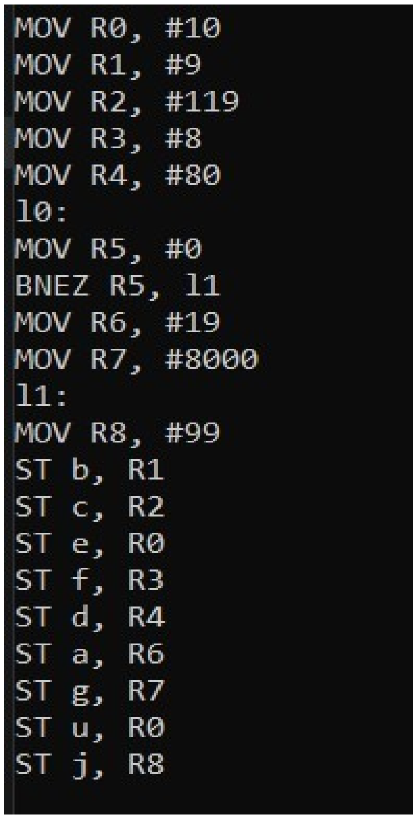
Fig. 4.5



```
ICG with optimisations(Packing temporaries & Constant Propagation)
a = 10
b = 9
t0 = 10 + 9
t1 = 19 + 100
c = 119
e = 10
f = 8
t2 = 10 * 8
d = 80
l0:
t3 = 10 >= 9
t4 = not 1
if 0 goto l1
a = 19
t5 = 80 * 100
g = 8000
l1:
u = 10
j = 99
```

Fig. 4.6

Target Code:

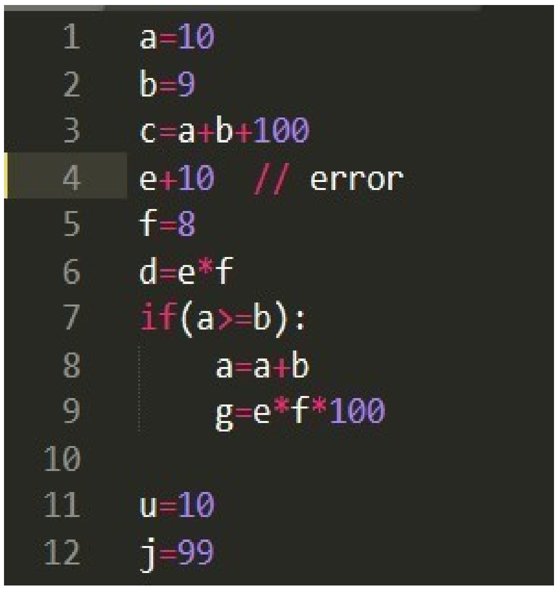


```
MOV R0, #10
MOV R1, #9
MOV R2, #119
MOV R3, #8
MOV R4, #80
l0:
MOV R5, #0
BNEZ R5, l1
MOV R6, #19
MOV R7, #8000
l1:
MOV R8, #99
ST b, R1
ST c, R2
ST e, R0
ST f, R3
ST d, R4
ST a, R6
ST g, R7
ST u, R0
ST j, R8
```

Fig. 4.7

TEST CASE 2 (Syntax Error):

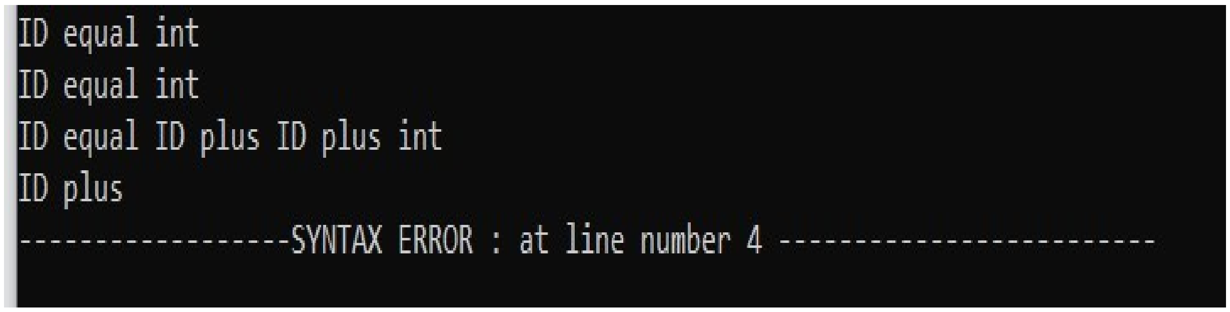
Input:



```
a=10
b=9
c=a+b+100
e+10  // error
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Fig. 5.1

Output:

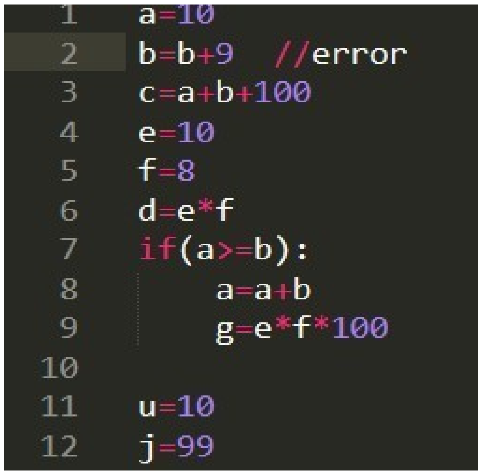


```
ID equal int
ID equal int
ID equal ID plus ID plus int
ID plus
-----------------SYNTAX ERROR : at line number 4 ------------------------
```

Fig. 5.2

TEST CASE 3 (Semantic Error):

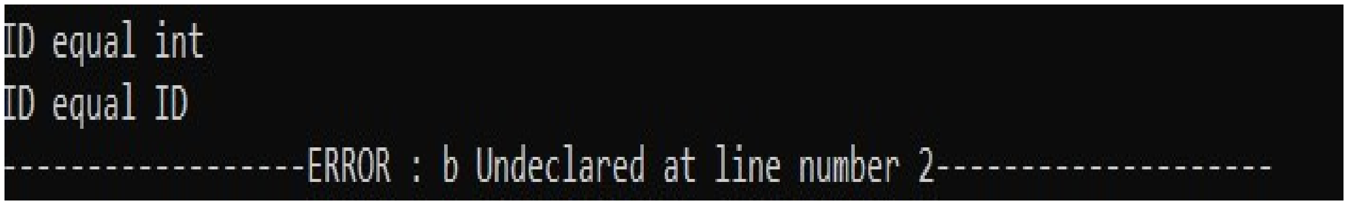
Input:



```
a=10
b=b+9   //error
c=a+b+100
e=10
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Fig. 6.1

Output:

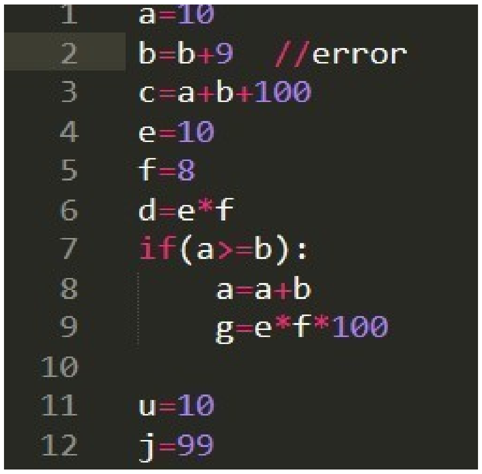


```
ID equal int
ID equal ID
------------------ERROR : b Undeclared at line number 2--------------------
```

Fig. 6.2

TEST CASE 4 (Error Recovery):

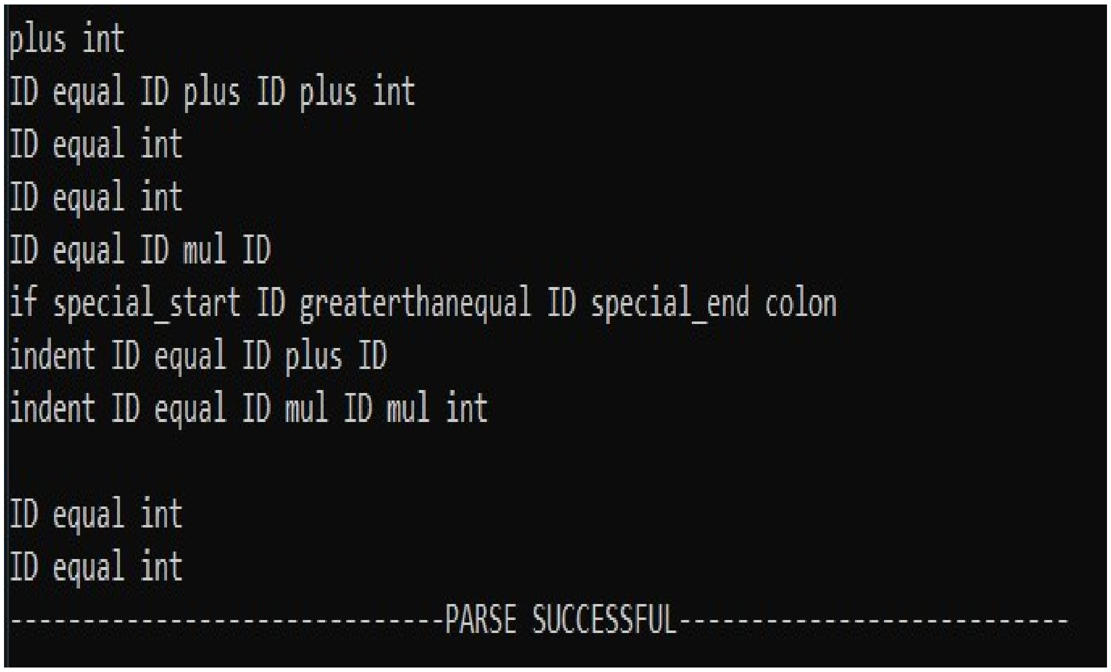
Input:



```
a=10
b=b+9   //error
c=a+b+100
e=10
f=8
d=e*f
if(a>=b):
    a=a+b
    g=e*f*100

u=10
j=99
```

Fig. 7.1

Output:



```
plus int
ID equal ID plus ID plus int
ID equal int
ID equal int
ID equal ID mul ID
if special_start ID greaterthanequal ID special_end colon
indent ID equal ID plus ID
indent ID equal ID mul ID mul int

ID equal int
ID equal int
-----------------------------PARSE SUCCESSFUL--------------------------
```

Fig. 7.2

CONCLUSIONS

Making a full complete compiler is a very difficult task and it takes lots of time to make it. So, we have successfully made a mini compiler which performs following operations:

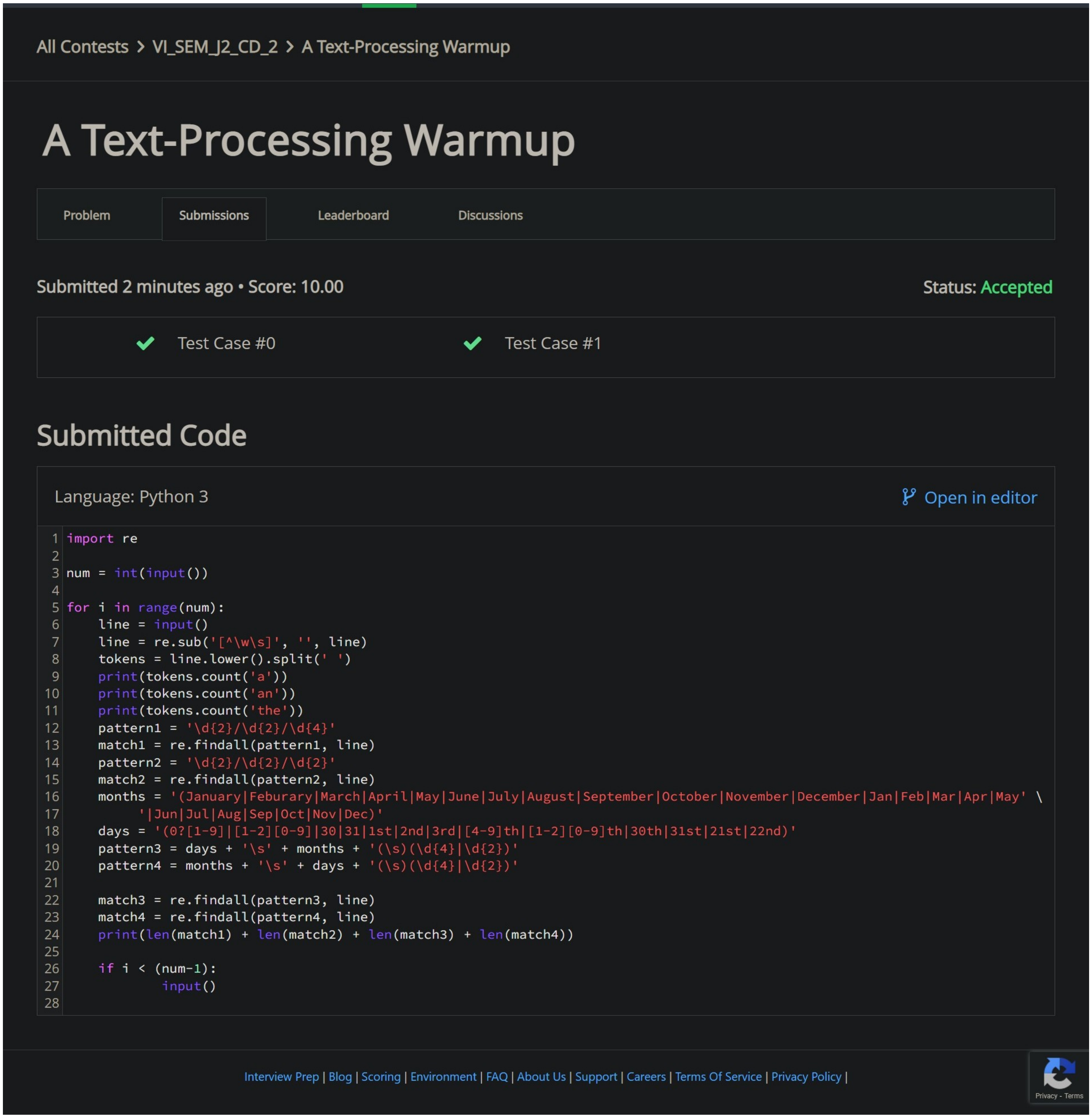
- This is a mini-compiler for python using lex and yacc files which takes in a python program and according to the context free grammar written, the program is validated.
- Regular Expressions are written to generate the tokens.
- Symbol table is created to store the information about theidentifiers.
- Abstract syntax tree is generated and displayed according to thepre-order tree traversal.
- Intermediate code is generated, and the data structure used for optimization is Quadruples. The optimization techniques used are constant propagation and packing temporaries.
- The optimised intermediate code is then converted to the Target code using a hypothetical machine model.
- Error handling and recovery implemented take care of erroneousinputs.

FUTURE ENHANCEMENTS

This mini-compiler can be enhanced to a complete compiler for the Python language by making a few improvements. User defined functions can be handled and the functionality of importing libraries and calling library functions can be taken care of. Datatypes other than integer, example strings, lists, tuples, dictionaries, etc. can be included and constructs other than ‘while’ and ‘if-else’, like ‘for’ can be added in the compiler program. The output can be made to look more enhanced and beautiful. The overall efficiency and speed of the program can be improved by using some other data structures, functions or approaches.

REFERENCES

- **Lex and Yacc**: http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
- **Introduction about Flex and Bison**: http://dinosaur.compilertools.net/
- **Full Grammar Specification**: https://docs.python.org/3/reference/grammar.html
- **Introduction to Yacc**: https://www.inf.unibz.it/~artale/Compiler/intro-yacc.pdf
- **Intermediate Code Generation:** https://2k8618.blogspot.com/2011/06/intermediate-code-generator-for.html?m=0
- **Target Code Generation:**
  https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf
  https://www.javatpoint.com/code-generation
- **A Code Generator to translate Three-Address intermediate code to MIPS assembly code**: https://www2.cs.arizona.edu/~debray/Teaching/CSc453/DOCS/3addr2spim.pdf



All Contests > VI_SEM_J2_CD_2 > A Text-Processing Warmup

A Text-Processing Warmup

Problem | Submissions | Leaderboard | Discussions

Submitted 2 minutes ago • Score: 10.00

Status: Accepted

✓ Test Case #0 ✓ Test Case #1

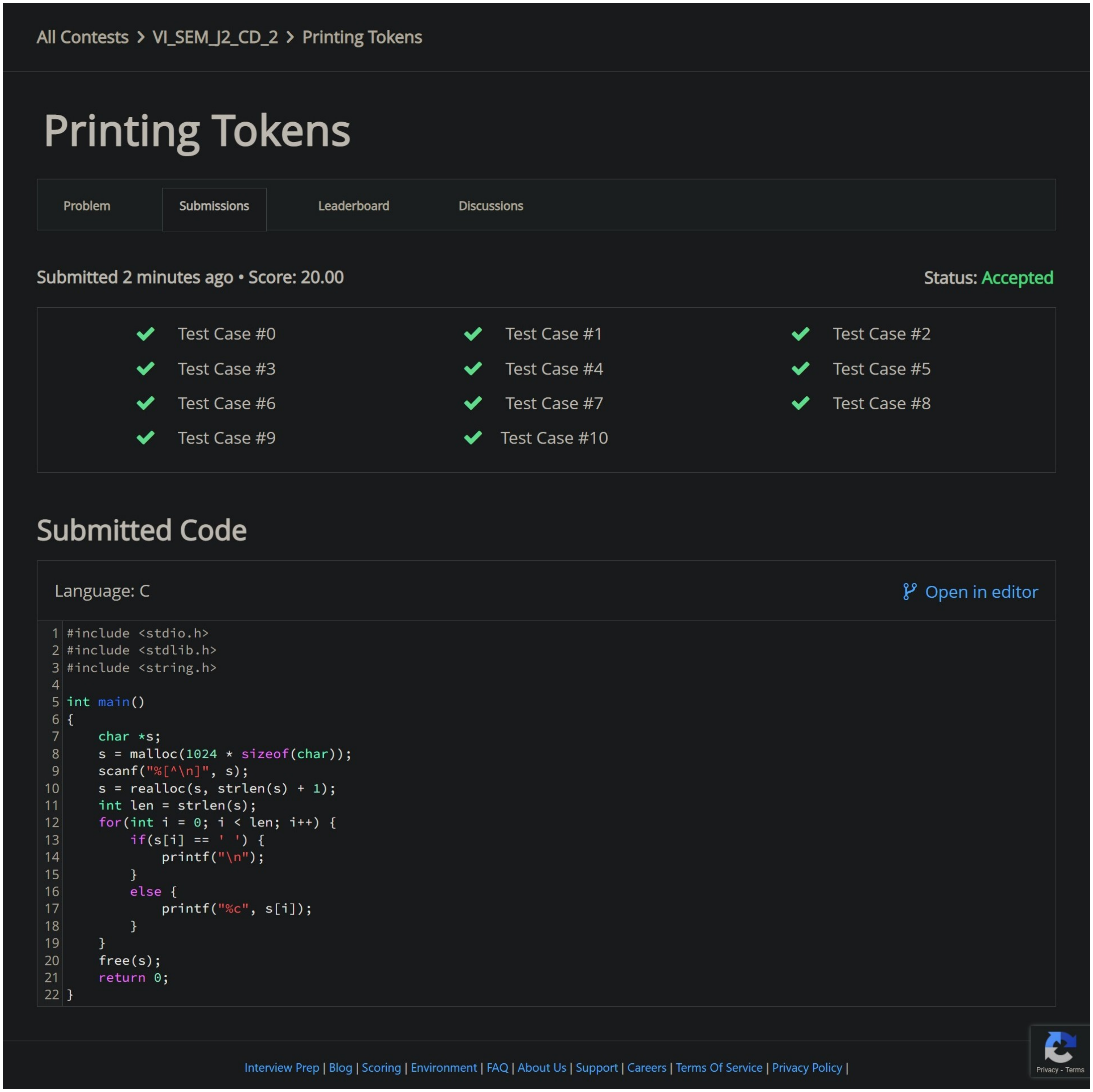
Submitted Code

Language: Python 3

Open in editor

```python
import re

num = int(input())

for i in range(num):
    line = input()
    line = re.sub('[^\w\s]', '', line)
    tokens = line.lower().split(' ')
    print(tokens.count('a'))
    print(tokens.count('an'))
    print(tokens.count('the'))
    pattern1 = '\d{2}/\d{2}/\d{4}'
    match1 = re.findall(pattern1, line)
    pattern2 = '\d{2}/\d{2}/\d{2}'
    match2 = re.findall(pattern2, line)
    months = '(January|Feburary|March|April|May|June|July|August|September|October|November|December|Jan|Feb|Mar|Apr|May' \
         '|Jun|Jul|Aug|Sep|Oct|Nov|Dec)'
    days = '(0?[1-9]|[1-2][0-9]|30|31|1st|2nd|3rd|[4-9]th|[1-2][0-9]th|30th|31st|21st|22nd)'
    pattern3 = days + '\s' + months + '(\s)(\d{4}|\d{2})'
    pattern4 = months + '\s' + days + '(\s)(\d{4}|\d{2})'

    match3 = re.findall(pattern3, line)
    match4 = re.findall(pattern4, line)
    print(len(match1) + len(match2) + len(match3) + len(match4))

    if i < (num-1):
            input()

```

Interview Prep | Blog | Scoring | Environment | FAQ | About Us | Support | Careers | Terms Of Service | Privacy Policy |


All Contests > VI_SEM_J2_CD_2 > Printing Tokens

Printing Tokens

Submitted 2 minutes ago • Score: 20.00

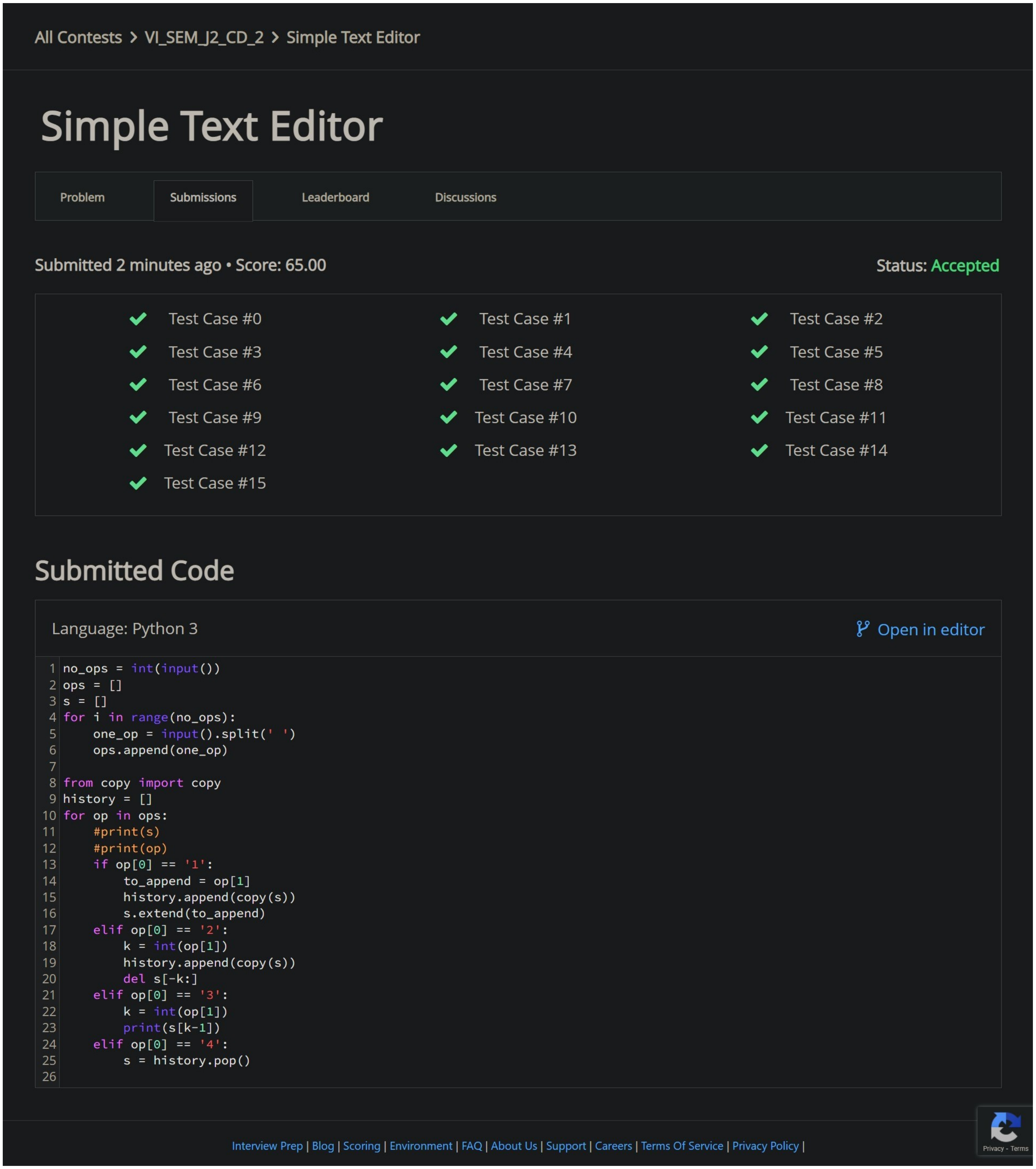
Status: Accepted

✓ Test Case #0	✓ Test Case #1	✓ Test Case #2
✓ Test Case #3	✓ Test Case #4	✓ Test Case #5
✓ Test Case #6	✓ Test Case #7	✓ Test Case #8
✓ Test Case #9	✓ Test Case #10	

Submitted Code

Language: C

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

int main()
{
    char *s;
    s = malloc(1024 * sizeof(char));
    scanf("%[^\n]", s);
    s = realloc(s, strlen(s) + 1);
    int len = strlen(s);
    for(int i = 0; i < len; i++) {
        if(s[i] == ' ') {
            printf("\n");
        }
        else {
            printf("%c", s[i]);
        }
    }
    free(s);
    return 0;
}
```


All Contests > VI_SEM_J2_CD_2 > Simple Text Editor

Simple Text Editor

Submitted 2 minutes ago • Score: 65.00

Status: Accepted

✓ Test Case #0	✓ Test Case #1	✓ Test Case #2
✓ Test Case #3	✓ Test Case #4	✓ Test Case #5
✓ Test Case #6	✓ Test Case #7	✓ Test Case #8
✓ Test Case #9	✓ Test Case #10	✓ Test Case #11
✓ Test Case #12	✓ Test Case #13	✓ Test Case #14
✓ Test Case #15		

Submitted Code

Language: Python 3

```python
no_ops = int(input())
ops = []
s = []
for i in range(no_ops):
    one_op = input().split(' ')
    ops.append(one_op)

from copy import copy
history = []
for op in ops:
    #print(s)
    #print(op)
    if op[0] == '1':
        to_append = op[1]
        history.append(copy(s))
        s.extend(to_append)
    elif op[0] == '2':
        k = int(op[1])
        history.append(copy(s))
        del s[-k:]
    elif op[0] == '3':
        k = int(op[1])
        print(s[k-1])
    elif op[0] == '4':
        s = history.pop()

```