Thadomal Shahani Engineering College

Bandra (W.), Mumbai- 400 050.

CERTIFICATE

Certify that Mr./Miss Parth Dabholkar

of Computer Department, Semester VI with

Roll No. 2103032 has completed a course of the necessary

experiments in the subject SPCC under my

supervision in the **Thadomal Shahani Engineering College**

Laboratory in the year 2023 - 2024

13/4/24

Teacher In-Charge

Head of the Department

Date 13/4/24

Principal

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

CONTENTS

3/4/24

Parth D.
C12
2103032

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

EXPERIMENT NO : 1

AIM : Write a program to implement lexical analyser

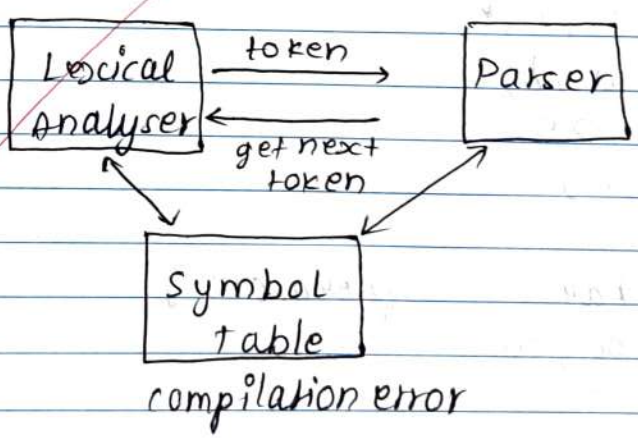
THEORY : Lexical analysis is the starting phase of the compiler. It gathers modified source code that is written in the form of sentences from the language processor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace in the source code. If the lexical analyzer gets any invalid token, it generates an error. The stream of character is read by it and it seeks the legal tokens, and then the data is passed to the syntax analyzer, when it is asked for.

Token : It is a sequence of characters that represent a unit of information in the source code.

Pattern : the description used by token is known as pattern.

Lexeme : A sequence of characters in source code, as per matching pattern of a token, is known as lexeme.

* Architecture of LA:



Functions: (i) Lexical analyzer is responsible for removing whitespaces and comments from source program.

(ii) It corresponds to the error messages with source program.

(iii) It helps to identify tokens.

(iv) The input characters are read by the lexical analyzer from the source code.

Example of Tokens: keywords: while, if, for etc.
operators: '+', '++', '--', ... etc..
seperators: ';' ',' ';' etc..

Input: int a = b + c * 25;
char ac = 'w';

['int', 'a', '=', 'b', '+', 'c', '*', '25', ';', 'char', 'ac', '=', 'w', ';']

keyword: int
operator: = , + , *
variables: a , b, c
numbers: 25
seperators: ;

keyword: char seperator: ';'
variables: ac , w
operator: =

24 | 1 | 24
24
A+

```
"C12_2103032 Parth D. Copyright@2023-2024"

"Code by 2103032"
import string
file = open('test1', 'r')
content = file.read()

print(type(content))

keywords = ["int","char","float", "double", "long", "String"]
variables = list(string.ascii_letters)
numbers_list = list(range(-9999,10000))
operators = ["+","-","*","/","="]
seperators = [",",";"]
count = 0

word = content
str_list = word.split()
print(str_list)

for i in str_list:
    if i in keywords:
        print(f"keyword : {i}")
    if i in variables:
        print(f"variables : {i}")
    if i in map(str,numbers_list):
        print(f"numbers : {i}")
    if i in operators:
        print(f"operators : {i}")
    if i in seperators:
        print(f"seperators : {i}")
    count = count + 1
print("Total numbers of tokens are: ",count)
```

```
<class 'str'>
['int', 'z', '=', 'x', '+', 'y', '-', '10', ';', 'int', 'a', '=', 'b']
keyword : int
variables : z
operators : =
variables : x
operators : +
variables : y
operators : -
numbers : 10
seperators : ;
keyword : int
variables : a
operators : =
variables : b
Total numbers of tokens are:  13
```

Parth s. Dabholkar
C12
2103032

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

EXPERIMENT NO: 2

AIM: To study and implement LEX and YACC tool.

THEORY: There are various way of developing a lexical analyzer, however use of an automatic lexical analyzer generator tool like (LEX) is common way to develop lexical analyzer.

LEX has an input specification file which contains definitions of token in form of regular expressions, which represents the patterns. These files are saved with .lex or .l extension. Each regular expression is associated with a statement in a programming language which will evaluate lexemes that match the regular expression. The LEX then constructs a state table for the appropriate finite state machine and creates program code which contains the table, the evaluation phases, and a routine which uses them approximately i.e. yy.lex.c.

The lex program has a specific structure as given below:

```
%{
   Declarations and Definitions
%}
%%
   Translational Rules
%%
Auxilliary Procedure.
```

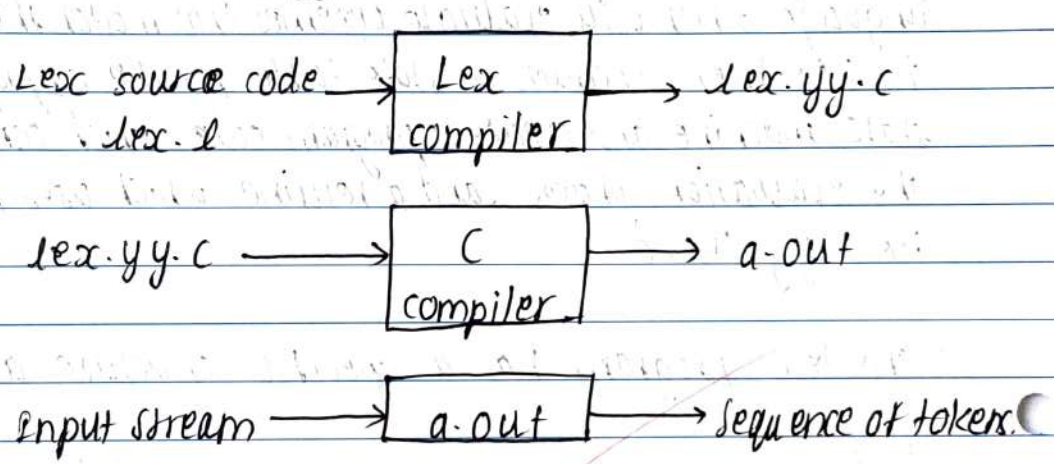
Eg:

```
%{ %/* Program to print*/
%}
%%
[\n] {
   printf("Hellow\n");
   }
%%
main(){
   yylex(); }
```

Declaration section: This section includes declarations, function definitions, start conditions and translations. It can also contain comments.

Translation Rules: The rules section contains pattern lines and action (code in C). If lexeme matches the pattern then respective action will be performed.

Auxiliary Procedure: LEX generates C code for the rules specified in the Rules section and places the code into a single function called yylex().

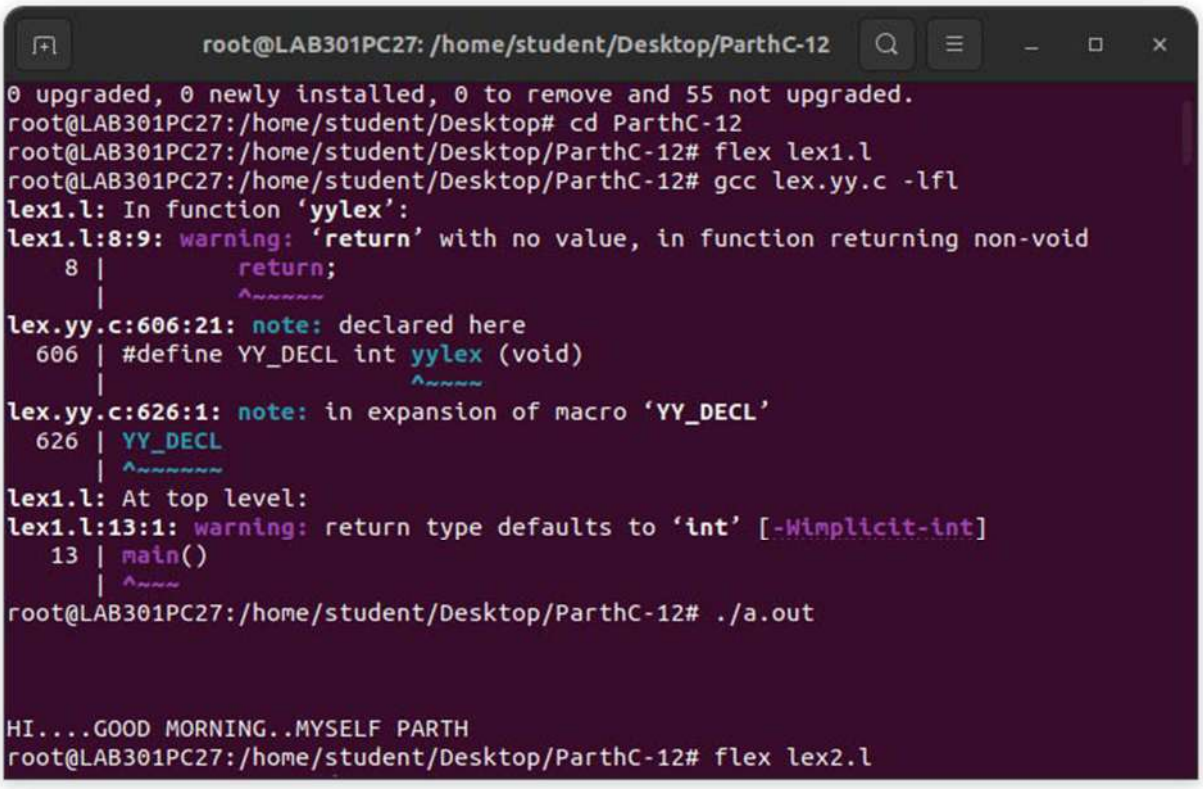


Execution of program: (i) $sudo su root // To access the root
(ii) # Enter Password //To get access to the root.
(iii) # apt-get install flex // Install FLEX tool.
(iv) Write a program and save with (.l) extension
(v) # flex prog1.l // Converts Lex to lex.yy.c
(vi) # gcc lex.yy.c -lfl // Compile C-file
(vii) # ./a.out // sequence of tokens.

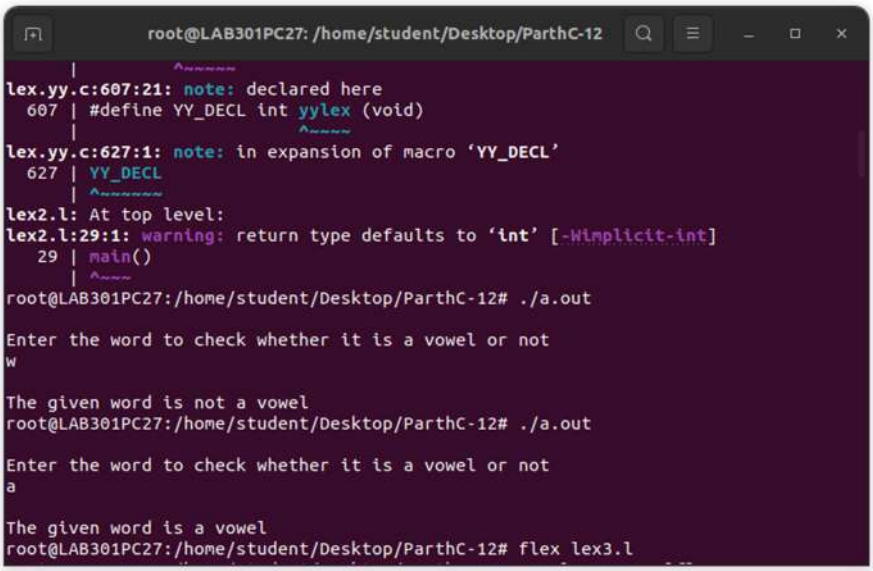
21/9/24
A+

LEX variables: yyin, yytext, yyleng, yylex(), yywrap() — Functions

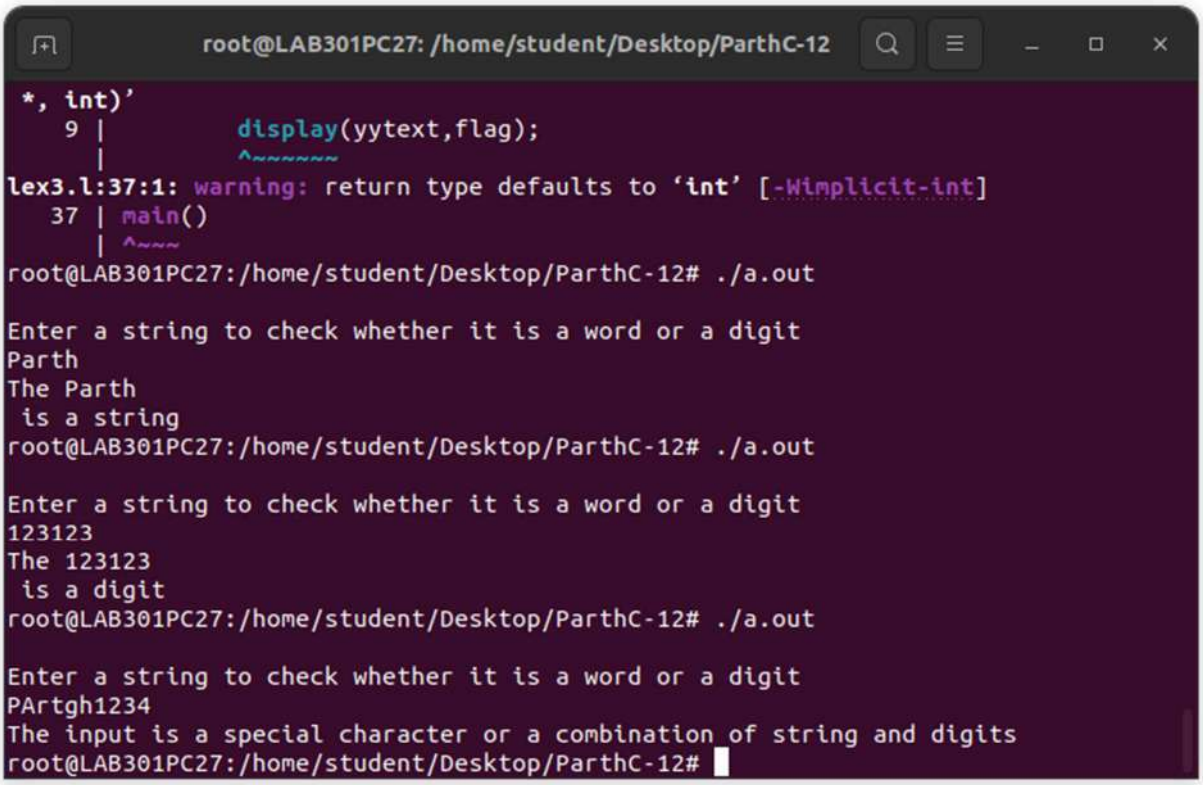
```
/* Code by Parth D. C12-2103032 */

%{
%}

%%

[\n] {
        printf("\n\nHI....GOOD MORNING..MYSELF PARTH \n");
        return;
        }

%%

main()
{
        yylex();
}
```



```
/* Code by Parth D. C12-2103032 */

%{
void display(int);
%}

%%

[a|e|i|o|u]+ {
        int flag=1;
        display(flag);
        return;
}

.+ {
        int flag=0;
        display(flag);
        return;
}

%%

void display(int flag)
{
        if(flag==0)
                printf("\nThe given word is not a vowel\n");
        else
                printf("\nThe given word is a vowel\n");
}

main()
{
        printf("\nEnter the word to check whether it is a vowel or not\n");
        yylex();
}
```



```
/* Code by Parth D. C12-2103032 */

%{
void dispay(char[],int);
%}

%%

[a-zA-Z]+[\n] {
        int flag=1;
        display(yytext,flag);
        return;
}

[0-9]+[\n] {
        int flag=0;
        display(yytext,flag);
        return;
}

.+ {
        int flag=-1;
        display(yytext,flag);
        return;
}

%%

void display(char string[],int flag)
{
        if(flag==1)
                printf("The %s is a string\n",string);
        else if(flag==0)
                printf("The %s is a digit\n",string);
        else
                printf("The input is a special character or a combination of string and digits\n");
}

main()
{
        printf("\nEnter a string to check whether it is a word or a digit\n");
        yylex();
}
```


root@LAB301PC27: /home/student/Desktop/ParthC-12
*, int)'
9 | display(yytext,flag);
| ^~~~~~~
lex3.l:37:1: warning: return type defaults to 'int' [-Wimplicit-int]
37 | main()
| ^~~~
root@LAB301PC27:/home/student/Desktop/ParthC-12# ./a.out
Enter a string to check whether it is a word or a digit
Parth
The Parth
is a string
root@LAB301PC27:/home/student/Desktop/ParthC-12# ./a.out
Enter a string to check whether it is a word or a digit
123123
The 123123
is a digit
root@LAB301PC27:/home/student/Desktop/ParthC-12# ./a.out
Enter a string to check whether it is a word or a digit
PArtgh1234
The input is a special character or a combination of string and digits
root@LAB301PC27:/home/student/Desktop/ParthC-12#

```
%{
void display(char[],int);
%}
%%
[a-zA-Z]+[\n] {
int flag=1;
display(yytext,flag);
return 1;
}
[0-9]+[\n] {
int flag=0;
display(yytext,flag);
return 1;
}
.+ {
int flag=1;
display(yytext,flag);
return 1;
}
%%
void display( char string[],int flag)
{
}
if(flag==1)
printf("\nThe given string %s is a word \n",string);
else if(flag==0)
printf("\nthe given string %d is a digit\n",atoi(string));
else
printf("\nthe given string is neither a word nor digit\n");
int main()
{
printf("\nenter a string\n");
yylex();
}
```

```
enter a string
Alphabets

The given string Alphabets is a word
admini@admini-OptiPlex-3070:~/Desktop/lex $ ./a.out

enter a string
421005

the given string 421005 is a digit
admini@admini-OptiPlex-3070:~/Desktop/lex $
```

```
%{
int lc=0, ch=0;
%}
%%
[\n] {lc++;}
[^\t] {ch+=yyleng;}
[^\t\n]+ {ch+=yyleng;}
%%
int main(int argc ,char* argv[ ])
{
printf("enter the sentence: ");
yylex();
printf("number of lines = %d\n", lc);
printf("number of characters = %d\n", ch);
}
int yywrap()
{
return 1;
}
```

```
admini@admini-OptiPlex-3070:~/Desktop/lex $ lex lex5.l
admini@admini-OptiPlex-3070:~/Desktop/lex $ gcc lex.yy.c -lfl
admini@admini-OptiPlex-3070:~/Desktop/lex $ ./a.out
Enter the sentence: Parth Dabholkar
this is second line
number of lines = 2
number of characters = 33
```

EXPERIMENT NO : 3

AIM: Write a program to implement FIRST and FOLLOW set for a given grammar.

THEORY: First and Follow are concepts used in compiler construction, specifically in parsing techniques like LL(1) and LR(1) parser (top down and bottom up respectively). They help in constructing predictive parsing tables, which are used to parse the input string efficiently.

FIRST SET : First(x) for a grammar symbol x is set of terminals that begin the strings derivable from x.

First set is a concept used in syntax analysis, specifically in the context of LL and LR parsing algorithms. It is a set of terminals that can appear immediately after a non terminal in a grammar.

The first set is used to determine which production rule should be used to expand a non-terminal in a LL or LR parser. For example, in a LL parser, if the next symbol in input stream is in the FIRST SET of a non-terminal, then that non-terminal can be safely expanded using the production rule that starts with a symbol.

Rules for FIRST SET:

(i) If 'a' is a terminal, then First(a) = {a}

(ii) If $A \rightarrow \epsilon$, then First(A) = {ϵ}

(iii) If $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_n$ is a production,

a) First(X) = First(Y_1) b) If First(Y_i) contains ϵ then First($Y_1 - \epsilon$) U First(Y_2)

c) If all Y_i contain ϵ then add ϵ to set.

FOLLOW SET: follow set in compiler design are used to identify the terminal symbol immediately after a non-terminal in a given language. follow set is used to avoid backtracking same as the first set. The only difference is follow set works on vanishing non-terminal on the right hand side so that decision making gets easier for the compiler while parsing.

Rules for FOLLOW:

(i) Follow(S), where S is start symbol is {$}
(ii) If A → pBq is a production, then follow(A) = First(q) = {q}
(iii) If A → pB is production then everything in Follow(A) is in Follow(B).
(iv) If A → pBq is a production and First(q) contains ε, then Follow(B) contains {First(q) − ε} ∪ Follow(A).

Eg: S → Aa
A → BD
B → b | ε
D → d | ε

First(D) = {d, ε}
First(B) = {b, ε}
First(A) = First(B) ∪ First(D) = {b, d, ε}
First(S) = First(A) ∪ First(a) = {a, b, d, ε}

Follow(S) = {$}
Follow(A) = {a}
Follow(B) = {d, a}
Follow(D) = {a}

28/2/24 A+

EXPERIMENT NO: 3

CODE:

```
import re
def cal_follow(s, productions, first):
    follow = set()
    if len(s)!=1 :
        return {}
    if(s == list(productions.keys())[0]):
        follow.add('$')

    for i in productions:
        for j in range(len(productions[i])):
            if(s in productions[i][j]):
                idx = productions[i][j].index(s)

                if(idx == len(productions[i][j])-1):
                    if(productions[i][j][idx] == i):
                        break
                    else:
                        f = cal_follow(i, productions, first)
                        for x in f:
                            follow.add(x)
                else:
                    while(idx != len(productions[i][j]) - 1):
                        idx += 1
                        if(not productions[i][j][idx].isupper()):
                            follow.add(productions[i][j][idx])
                            break
                        else:
                            f = cal_first(productions[i][j][idx], productions)

                            if('ε' not in f):
```

```
                    for x in f:
                        follow.add(x)
                    break
                elif('ε' in f and idx != len(productions[i][j])-1):
                    f.remove('ε')
                    for k in f:
                        follow.add(k)

                elif('ε' in f and idx == len(productions[i][j])-1):
                    f.remove('ε')
                    for k in f:
                        follow.add(k)

                    f = cal_follow(i, productions, first)
                    for x in f:
                        follow.add(x)
                 return follow

def cal_first(s, productions):
    first = set()

  for i in range(len(productions[s])):
     for j in range(len(productions[s][i])):
          c = productions[s][i][j]
          if(c.isupper()):
          f = cal_first(c, productions)
          if('ε' not in f):
             for k in f:
                first.add(k)
             break
          else:
             if(j == len(productions[s][i])-1):
```

```
                for k in f:
                    first.add(k)
            else:
                f.remove('ε')
                for k in f:
                    first.add(k)
        else:
            first.add(c)
            break

    return first

def main():
    productions = {}
    grammar = open("grammar9.txt", "r")

    first = {}
    follow = {}

    for prod in grammar:
        l = re.split("( /->/\n/)*", prod)
        m = []
        for i in l:
            if (i == "" or i == None or i == '\n' or i == " " or i == "-" or i == ">"):
                pass
            else:
                m.append(i)

        left_prod = m.pop(0)
        right_prod = []
        t = []
```

```
        for j in m:
            if(j != '|'):
                t.append(j)
            else:
                right_prod.append(t)
                t = []

        right_prod.append(t)
        productions[left_prod] = right_prod

    for s in productions.keys():
        first[s] = cal_first(s, productions)

    print("*****FIRST*****")
    for lhs, rhs in first.items():
        print(lhs, ":" , rhs)

    print("")

    for lhs in productions:
        follow[lhs] = set()

    for s in productions.keys():
        follow[s] = cal_follow(s, productions, first)

    print("*****FOLLOW*****")
    for lhs, rhs in follow.items():
        print(lhs, ":" , rhs)

    grammar.close()
if __name__ == "__main__":
    main()
```

INPUT FILE:

S -> aBDh

B -> cC

C -> bC | ε

D -> EF

E -> g | ε

F -> f | ε

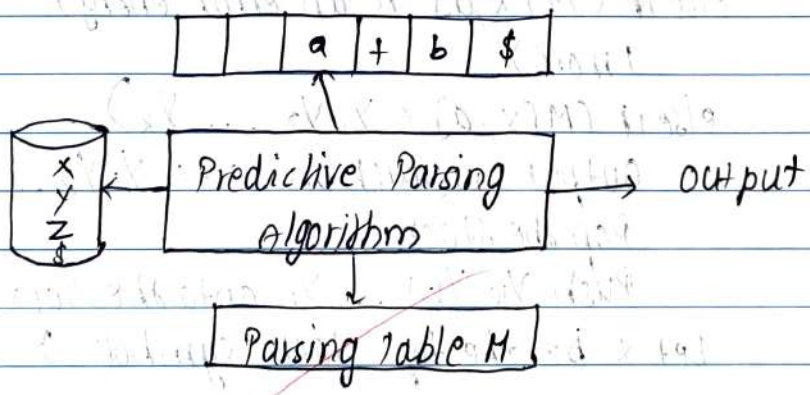
OUTPUT:

```
*****FIRST*****
S : {'a'}
B : {'c'}
C : {'ε', 'b'}
D : {'ε', 'f', 'g'}
E : {'ε', 'g'}
F : {'ε', 'f'}

*****FOLLOW*****
S : {'$'}
B : {'h', 'f', 'g'}
C : {'h', 'f', 'g'}
D : {'h'}
E : {'h', 'f'}
F : {'h'}
```

EXPERIMENT NO: 4

AIM: Write a program to implement LL(1) Parser.

THEORY: LL(1) Parser is a top down parser in syntax analysis. The first 'L' in LL(1) stands for scanning input from left to right, the second 'L' stands for producing a leftmost derivation, and the '1' stands for using one input symbol of lookahead at each step to make parsing action decisions. LL(1) grammar follows Top-down parsing method. For a class of grammars called LL(1) we construct grammar predictive parser.



* Algorithm for construction of parsing table:

Step 1: For each terminal 'a' in First(α)
Add $A \longrightarrow \alpha$ to $M[A, a]$

Step 2: Case 1: If ϵ is in First(α) than for each terminal b in follow(A) Add $A \longrightarrow \alpha$ to $M[A, b]$

Case 2: If ϵ is in First(α) and $ is in follow(A) then for each terminal b in follow(A). Add $\longrightarrow \alpha$ to $M[A, b]$

Step 3: Make each undefined entry of M be an error.

* Algorithm for parsing string using LL(1)

- Let a be the first symbol of w
- Let x be to top of the stack symbol.

```
while (x != $) {
    if (x == a)
        pop the stack and let 'a' be the next symbol of w
    else if (x is a terminal)
        Error()
    else if (M[x,a] is an error entry)
        Error()
    else if (M[x,a] = Y1 Y2 ... Yk)
        Output production X → Y1 Y2 ... Yk
        Pop the stack
        Push Yk Yk-1 ... Y1 onto the stack Y1 on top
    Let x be the top stack symbol }
```

Eg: $S \rightarrow (L) \mid a$
$L \rightarrow SL'$
$L' \rightarrow)SL' \mid \epsilon$

	First	Follow
S	(, a	$,)
L	(, a	)
L'	), ε	)

Parse Table:

	(	)	a	$
S	S → (L)		S → a	
L	L → SL'		L → SL'	
L'		L' →)SL'		
		L' → ε		

CODE:

```
import re
import pandas as pd


def cal_follow(s, productions, first):
    follow = set()
    if len(s)!=1 :
        return {}
    if(s == list(productions.keys())[0]):
        follow.add('$')


    for i in productions:
        for j in range(len(productions[i])):
            if(s in productions[i][j]):
                idx = productions[i][j].index(s)


                if(idx == len(productions[i][j])-1):
                    if(productions[i][j][idx] == i):
                        break
                    else:
                        f = cal_follow(i, productions, first)
                        for x in f:
                            follow.add(x)
                else:
                    while(idx != len(productions[i][j]) - 1):
                        idx += 1
                        if(not productions[i][j][idx].isupper()):
                            follow.add(productions[i][j][idx])
                            break
                        else:
                            f = cal_first(productions[i][j][idx], productions)


                            if('ε' not in f):
                                for x in f:
```

```
                follow.add(x)
            break
        elif('ε' in f and idx != len(productions[i][j])-1):
            f.remove('ε')
            for k in f:
                follow.add(k)

        elif('ε' in f and idx == len(productions[i][j])-1):
            f.remove('ε')
            for k in f:
                follow.add(k)

            f = cal_follow(i, productions, first)
            for x in f:
                follow.add(x)
            return follow
def cal_first(s, productions):
 first = set()

 for i in range(len(productions[s])):

    for j in range(len(productions[s][i])):

        c = productions[s][i][j]

        if(c.isupper()):
            f = cal_first(c, productions)
            if('ε' not in f):
                for k in f:
                    first.add(k)
                break
            else:
                if(j == len(productions[s][i])-1):
                    for k in f:
```

```
                    first.add(k)
                else:
                    f.remove('ε')
                    for k in f:
                        first.add(k)
            else:
                first.add(c)
                break

    return first

def parsing_table(productions, first, follow):

    print("\nParsing Table\n")

    table = {}
    for key in productions:
        for value in productions[key]:
            val = ''.join(value)
            if val != 'ε':
                for element in first[key]:
                    if(element != 'ε'):
                        if(not val[0].isupper()) :
                            if(element in val):
                                table[key, element] = val
                            else:
                                pass
                        else:
                            table[key, element] = val
            else:
                for element in follow[key]:
                    table[key, element] = val

    for key,val in table.items():
```

```
        print(key,"=>",val)

    new_table = {}
    for pair in table:
        new_table[pair[1]] = {}

    for pair in table:
        new_table[pair[1]][pair[0]] = table[pair]



    print("\n")
    print("\nParsing Table in matrix form\n")
    print(pd.DataFrame(new_table).fillna('-'))
    print("\n")

    return table


def check_validity(string, start, table):

    accepted = True

    input_string = string + '$'
    stack = []

    stack.append('$')
    stack.append(start)

    idx = 0
    print("Stack\t\tInput\t\tMoves")
    while (len(stack) > 0):

        top = stack[-1]
        print(f"Top => {top}")
```

```
        curr_string = input_string[idx]
        print(f"Current input => {curr_string}")

        if top == curr_string:
            stack.pop()
            idx += 1

        else:
            key = (top, curr_string)
            print(f"Key => {key}")
            if key not in table:
                accepted = False
                break

            value = table[key]
            if value != 'ε':
                value = value[::-1]
                value = list(value)

                stack.pop()

                for ele in value:
                    stack.append(ele)

            else:
                stack.pop()

    if accepted:
        print("String accepted")
    else:
        print("String not accepted")

def main():
```

```
productions = {}
grammar = open("grammar4.txt", "r")

first = {}
follow = {}
table = {}

start = ""
 for prod in grammar:
   l = re.split("( /->/\n/)*", prod)
   m = []
   for i in l:
      if (i == "" or i == None or i == '\n' or i == " " or i == "-" or i == ">"):
         pass
      else:
         m.append(i)

   left_prod = m.pop(0)
   right_prod = []
   t = []

   for j in m:
      if(j != '|'):
         t.append(j)
      else:
         right_prod.append(t)
         t = []

   right_prod.append(t)
   productions[left_prod] = right_prod

   if(start == ""):
      start = left_prod
```

```
    print("*****GRAMMAR*****")
    for lhs, rhs in productions.items():
        print(lhs, ":" , rhs)
    print("")

    for s in productions.keys():
        first[s] = cal_first(s, productions)

    print("*****FIRST*****")
    for lhs, rhs in first.items():
        print(lhs, ":" , rhs)

    print("")

    for lhs in productions:
        follow[lhs] = set()

    for s in productions.keys():
        follow[s] = cal_follow(s, productions, first)

    print("*****FOLLOW*****")
    for lhs, rhs in follow.items():
        print(lhs, ":" , rhs)

    table = parsing_table(productions, first, follow)
    string = input("Enter a string to check its valididty : ")
    check_validity(string, start, table)
    grammar.close()

if __name__ == "__main__":
    main()
```

INPUT FILE:

E -> TX

X -> +TX | ε

T -> FY

Y -> *FY | ε

F -> id | (E)

OUTPUT:

```
*****GRAMMAR*****
E : [['T', 'X']]
X : [['+', 'T', 'X'], ['ε']]
T : [['F', 'Y']]
Y : [['*', 'F', 'Y'], ['ε']]
F : [['i', 'd'], ['(', 'E', ')']]

*****FIRST*****
E : {'(', 'i'}
X : {'+', 'ε'}
T : {'(', 'i'}
Y : {'ε', '*'}
F : {'(', 'i'}

*****FOLLOW*****
E : {')', '$'}
X : {')', '$'}
T : {'+', ')', '$'}
Y : {'+', ')', '$'}
F : {'+', '*', ')', '$'}
```

```
Parsing Table in matrix form


     (   i    +  )  $    *
E   TX  TX    -  -  -    -
T   FY  FY    -  -  -    -
F  (E)  id    -  -  -    -
X    -   -  +TX  ε  ε    -
Y    -   -    ε  ε  ε  *FY
```

```
Enter a string to check its valididty :
id*id
Stack           Input           Moves
Top => E
Current input => i
Key => ('E', 'i')
Top => T
Current input => i
Key => ('T', 'i')
Top => F
Current input => i
Key => ('F', 'i')
Top => i
Current input => i
Top => d
Current input => d
Top => Y
Current input => *
Key => ('Y', '*')
Top => *
```

```
Current input => *
Top => F
Current input => i
Key => ('F', 'i')
Top => i
Current input => i
Top => d
Current input => d
Top => Y
Current input => $
Key => ('Y', '$')
Top => X
Current input => $
Key => ('X', '$')
Top => $
Current input => $
String accepted
```

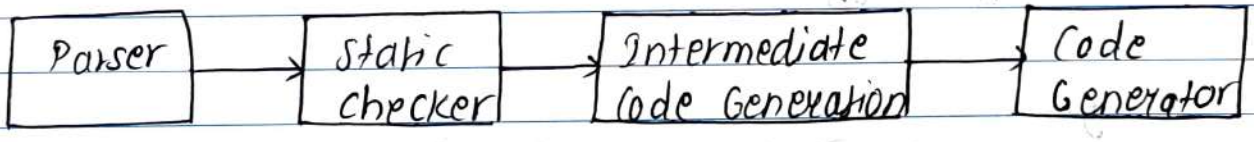
EXPERIMENT NO: 5

AIM: Write a program to implement Three Address Code(TAC).

THEORY:

In the analysis model of compiler, the frontend of a compiler translates a source program into an independent intermediate code, then the backend of the compiler uses this intermediate code to generate the target code. Because of the machine-independent intermediate code, portability will be enhanced. The commonly used intermediate code representation are:

1. Postfix Notation
2. Three Address Code.



Three Address Code:

- Three address code is a sequence of statement of the general form:

  $x := y \ op \ z$

  where x, y and z are names, constants or compiler-generated temporaries.

- Thus, a source language expression like x+y*z might be translated into a sequence.

$t_1 := y * z$
$t_2 := x + t_1$

where t_1 and t_2 are compiler generated temporary names.

For eg:
Given Expression: $a = b * -c + b * -c$

For Syntax Tree

$t_1 := -c$
$t_2 := b * t_1$
$t_3 := -c$
$t_4 := b * t_3$
$t_5 := t_2 * + t_4$
$a := t_5$

For directed acyclic graph

$t_1 := -c$
$t_2 := b * t_1$
$t_5 := t_2 + t_2$
$a := t_5$

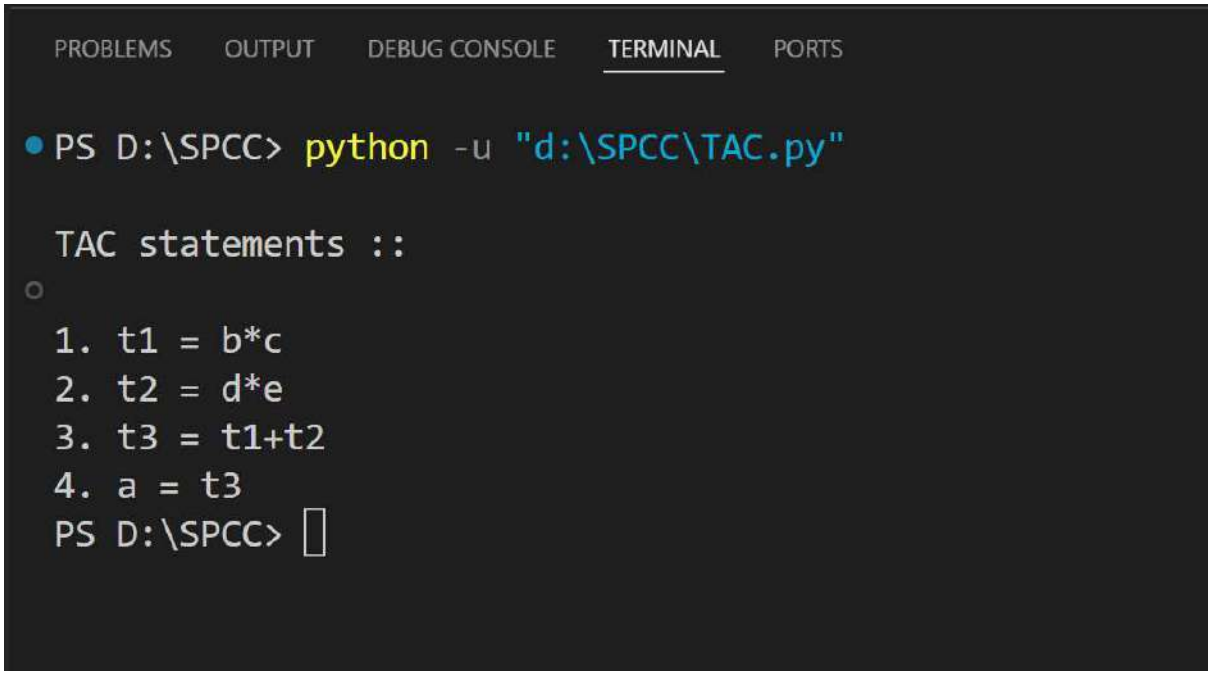
Eg:
Given expression: $a * (b + c)$

Three Address Code (TAC)

$t_1 = b + c$
$t_2 = \text{uminus } t_1$
$t_3 = a * t_2$

```
def get_precedence(c):
    if c == '^':
        return 3
    elif c == '*' or c == '/':
        return 2
    return 1

def infix_to_postfix(ip):
    res = []
    st = []
    n = len(ip)
    for i in range(n):
        if ip[i].isalpha():
            res.append(ip[i])
        elif ip[i] == '(':
            st.append('(')
        elif ip[i] == ')':
            while st and st[-1] != '(':
                res.append(st.pop())
            st.pop()
        else:
            while st and get_precedence(ip[i]) <= get_precedence(st[-1]):
                res.append(st.pop())
            st.append(ip[i])

    while st:
        res.append(st.pop())

    return ''.join(res)
```

```
def is_operator(c):
    return c in ['+', '-', '*', '/', '^']

def tac(postfix, input_str):
    s = []
    print("\nTAC statements :: \n")
    ct = 1
    for i in range(len(postfix)):
        if not is_operator(postfix[i]):
            s.append(postfix[i])
        else:
            op2 = s.pop()
            op1 = s.pop()
            intr_rhs = op1 + postfix[i] + op2
            intr_lhs = 't' + str(ct)
            print(f"{ct}. {intr_lhs} = {intr_rhs}")
            ct += 1
            s.append(intr_lhs)

    print(f"{ct}. {input_str[0]} = {s[-1]}")
    s.pop()

def main():
    ip = "a=b*c+d*e"
    infix = ip[2:]
    postfix = infix_to_postfix(infix)
    tac(postfix, ip)

if __name__ == "__main__":
    main()
```


PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
PS D:\SPCC> python -u "d:\SPCC\TAC.py"
TAC statements ::
1. t1 = b*c
2. t2 = d*e
3. t3 = t1+t2
4. a = t3
PS D:\SPCC>

EXPERIMENT NO: 6

AIM: Write a program to implement code optimization.

THEORY: The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resource (i.e CPU, Memory) so that faster-running machine code will result. Compiler optimization process should meet the following objectives:

- The optimization must be correct, it must not, in any way change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

Types of Code optimization: This optimization phase can be broadly classified into two types:

1. Machine Independent Optimization:
   This code optimization phase attempts to improve the intermediate code to get better target code as the output. The part of the intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

2. Machine Dependent Optimization:
   Machine-dependent optimization is done after the target code has been generated. and when the code is transformed

according to the target machine. It involves CPU registers and may have absolute memory reference rather than relative reference. Machine-dependent optimizers put efforts to take maximum advantage of the memory hierarchy.

Advantages of Code Optimization:

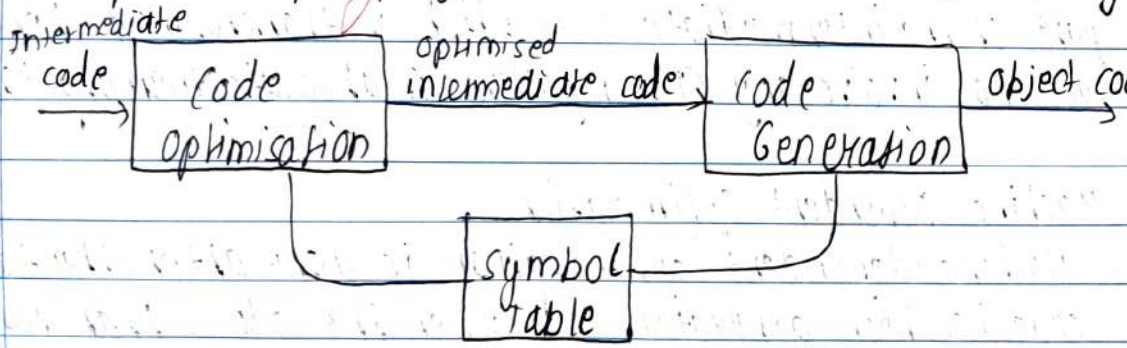
i) Improved performance: Code optimization can help reduce the size of generate code, making it easier to distribute and deploy.

ii) Reduction in code size: Code optimization can help reduce the size of the generated code, making it easier to distribute and deploy.

Disadvantages of Code Optimization:

i) Increased Compilation time: Code optimization can significantly increase the compilation time, which can be a significant drawback when developing large software systems.

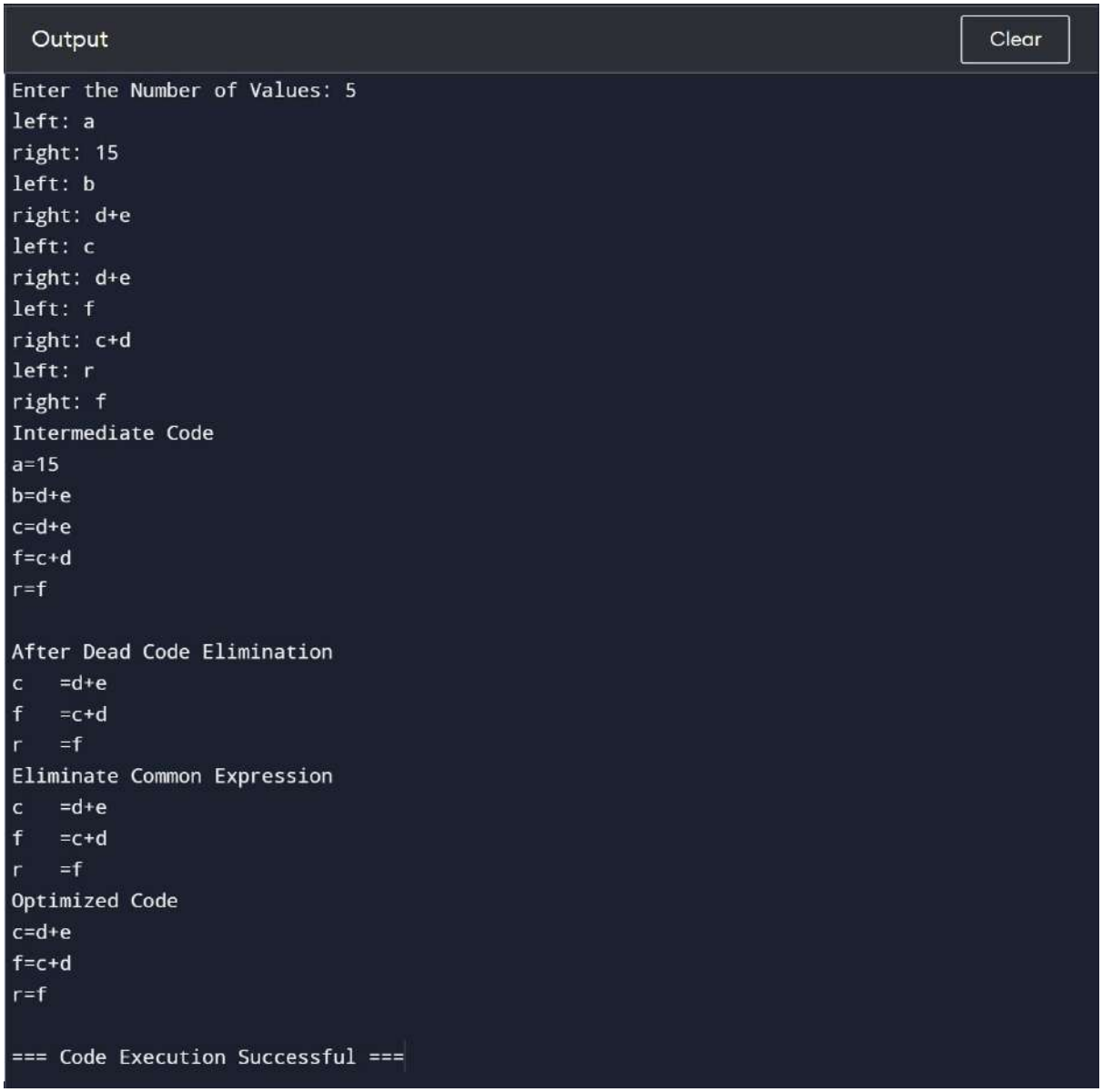
ii) Increased Complexity: Code optimization can result in more complex code, making it harder to understand and debug.



```
class Operation:
    def __init__(self):
        self.l = ''
        self.r = ''

op = [Operation() for _ in range(10)]
pr = [Operation() for _ in range(10)]

def main():
    global op, pr
    n = int(input("Enter the Number of Values: "))
    for i in range(n):
        op[i].l = input("left: ")[0]
        op[i].r = input("right: ")

    print("Intermediate Code")
    for i in range(n):
        print(f"{op[i].l}={op[i].r}")

    z = 0
    for i in range(n - 1):
        temp = op[i].l
        for j in range(n):
            p = op[j].r.find(temp)
            if p != -1:
                pr[z].l = op[i].l
                pr[z].r = op[i].r
                z += 1

    pr[z].l = op[n - 1].l
```

```
pr[z].r = op[n - 1].r
z += 1

print("\nAfter Dead Code Elimination")
for k in range(z):
    print(f"{pr[k].l}\t={pr[k].r}")

for m in range(z):
    tem = pr[m].r
    for j in range(m + 1, z):
        p = tem.find(pr[j].r)
        if p != -1:
            t = pr[j].l
            pr[j].l = pr[m].l
            for i in range(z):
                l = pr[i].r.find(t)
                if l != -1:
                    pr[i].r = pr[i].r.replace(pr[m].l, "", 1)

print("Eliminate Common Expression")
for i in range(z):
    print(f"{pr[i].l}\t={pr[i].r}")

for i in range(z):
    for j in range(i + 1, z):
        q = pr[i].r == pr[j].r
        if pr[i].l == pr[j].l and not q:
            pr[i].l = ''

print("Optimized Code")
```

```
    for i in range(z):
        if pr[i].l != '':
            print(f"{pr[i].l}={pr[i].r}")

if __name__ == "__main__":
    main()
```



EXPERIMENT NO: 7

AIM: Write a program to implement Pass1 and Pass 2 of Multi-pass Assembler.

THEORY:

Assembler is a program for converting instructions written in low level assembly code into relocatable machine code and generating along information for the loader. It is necessary to convert written programs into machinery code. This called as translation of the high level language to low level that is machinary code. This type of translation is done with the help of the assembler.

Pass 1:

1. **Input Processing:** In pass 1, the assembler reads the entire assembly language program line by line.

2. **SymbolTable Creation:** As assembler encounters lables, it adds them to a symbol table along with their corresponding memory address.

3. **location counter Management (LC):** The assembler maintains a location counter(LC) that helps track of memory address addressed to instructions and data.

4. **Error Detection:** Pass1 performes basic error checks such as syntax errors, undefined symbols and duplicate labels.

5. Output: Pass 1 generates an intermediate file containing the symbol table and modified assembly code with resolved addresses.

Pass 2:

1) Input Processing: Pass 2 takes the intermediate file generated by Pass1 as input, it reads the modified assembly code and symbols.

2) Instruction Translation: For each instruction, Pass 2 translates the mnemonic opcode and operands into corresponding machine code.

3) Object Code Generation: As Pass 2 processes each instruction, it generates the object code in binary or hexadecimal format.

4) Error Handling: Pass 2 performs additional error checks such as undefined symbols (not resolved in Pass1) invalid opcodes and operand mismatches.

5) Final Output: Upon successful completion, Pass 2 produces the final machine code or object program ready for execution on the target computer.

8/4/24

EXPERIMENT NO. 7

AIM: a) Write a program to implement Pass 1 of Multi-Pass Assembler
b) Write a program to implement Pass 2 of Multi-Pass Assembler

input.txt

```
START 100
A DS 3
L1 MOVEM AREG, B
ADD AREG, C
MOVER AREG, ='12'
D EQU A+1
LTORG
L2 PRINT D
ORIGIN A-1
MOVER AREG, ='5'
C DC '5'
ORIGIN L2+1
STOP
B DC '19'
END
```

```
START 501
READ A
READ B
MOVER AREG A
ADD AREG B
MOVEM AREG C
PRINT C
A DS 1
B DS 1
C DS 1
END
```

Pass1.java

```
import java.io.*;
import java.util.*;
public class PassOne {
  Hashtable < String, MnemonicTable > is = new Hashtable < > ();
  ArrayList < String > symtab = new ArrayList < > ();
  ArrayList < Integer > symaddr = new ArrayList < > ();
  ArrayList < String > littab = new ArrayList < > ();
  ArrayList < Integer > litaddr = new ArrayList < > ();
  ArrayList < Integer > pooltab = new ArrayList < > ();
  int LC = 0;
  public void createIS() {
    MnemonicTable m = new MnemonicTable("STOP", "00", 0);
    is.put("STOP", m);
    m = new MnemonicTable("ADD", "01", 0);
    is.put("ADD", m);
    m = new MnemonicTable("SUB", "02", 0);
    is.put("SUB", m);
    m = new MnemonicTable("MULT", "03", 0);
    is.put("MULT", m);
    m = new MnemonicTable("MOVER", "04", 0);
    is.put("MOVER", m);
    m = new MnemonicTable("MOVEM", "05", 0);
    is.put("MOVEM", m);
    m = new MnemonicTable("COMP", "06", 0);
    is.put("COMP", m);

    m = new MnemonicTable("BC", "07", 0);
    is.put("BC", m);
    m = new MnemonicTable("DIV", "08", 0);
    is.put("DIV", m);
    m = new MnemonicTable("READ", "09", 0);
    is.put("READ", m);
    m = new MnemonicTable("PRINT", "10", 0);
    is.put("PRINT", m);
  }
  public void generateIC() throws Exception {
    BufferedWriter wr = new BufferedWriter(new FileWriter("intermediate.txt"));
    BufferedReader br = new BufferedReader(new FileReader("input.txt"));
```

```
String line = " ";
pooltab.add(0, 0);
while ((line = br.readLine()) != null) {
  String[] split = line.split("\\s+");
  if (split[0].length() > 0) {
    //it is a label
    if (!symtab.contains(split[0])) {
      symtab.add(split[0]);
      symaddr.add(LC);
    } else {
      int index = symtab.indexOf(split[0]);
      symaddr.remove(index);
      symaddr.add(index, LC);
    }
  }
  if (split[1].equals("START")) {
    LC = Integer.parseInt(split[2]);
    wr.write("(AD,01)(C," + split[2] + ") \n");
  } else if (split[1].equals("ORIGIN")) {
    if (split[2].contains("+") || split[2].contains("-")) {
      LC = getAddress(split[2]);
    } else {
      LC = symaddr.get(symtab.indexOf(split[2]));
    }
  } else if (split[1].equals("EQU")) {
    int addr = 0;
    if (split[2].contains("+") || split[2].contains("-")) {
      addr = getAddress(split[2]);
    } else {
      addr = symaddr.get(symtab.indexOf(split[2]));

    }
    if (!symtab.contains(split[0])) {
      symtab.add(split[0]);
      symaddr.add(addr);
    } else {
      int index = symtab.indexOf(split[0]);
      symaddr.remove(index);
      symaddr.add(index, addr);
    }
```

```
} else if (split[1].equals("LTORG") || split[1].equals("END")) {
  if (litaddr.contains(0)) {
    for (int i = pooltab.get(pooltab.size() - 1); i < littab.size(); i++) {
      if (litaddr.get(i) == 0) {
        litaddr.remove(i);
        litaddr.add(i, LC);
        LC++;
      }
    }
    if (!split[1].equals("END")) {
      pooltab.add(littab.size());
      wr.write("(AD,05) \n");
    } else
      wr.write("(AD,04) \n");
  }
} else if (split[1].contains("DS")) {
  LC += Integer.parseInt(split[2]);
  wr.write("(DL,01) (C," + split[2] + ") \n");
} else if (split[1].equals("DC")) {
  LC++;
  wr.write("(DL,02) (C," + split[2].replace("'", "").replace("'", "") + ") \n");
} else if (is.containsKey(split[1])) {
  wr.write("(IS," + is.get(split[1]).opcode + ") ");
  if (split.length > 2 && split[2] != null) {
    String reg = split[2].replace(",", "");
    if (reg.equals("AREG")) {
      wr.write("(1) ");
    } else if (reg.equals("BREG")) {
      wr.write("(2) ");
    } else if (reg.equals("CREG")) {

      wr.write("(3) ");
    } else if (reg.equals("DREG")) {
      wr.write("(4) ");
    } else {
      if (symtab.contains(reg)) {
        wr.write("(S," + symtab.indexOf(reg) + ") ");
      } else {
        symtab.add(reg);
        symaddr.add(0);
```

```
          wr.write("(S," + symtab.indexOf(reg) + ") ");
        }
      }
    }
    if (split.length > 3 && split[3] != null) {
      if (split[3].contains("=")) {
        String norm = split[3].replace("=", "").replace("'", "").replace("'", "");
        if (!littab.contains(norm)) {
          littab.add(norm);
          litaddr.add(0);
          wr.write("(L," + littab.indexOf(norm) + ") \n");
        } else {
          wr.write("L," + littab.indexOf(norm) + ") \n");
        }
      } else if (symtab.contains(split[3])) {
        wr.write("(S," + symtab.indexOf(split[3]) + ")");
      } else {
        symtab.add(split[3]);
        symaddr.add(0);
        wr.write("(S," + symtab.indexOf(split[3]) + ")");
      }
    }
    LC++;
  }
}
wr.flush();
wr.close();
BufferedWriter br1 = new BufferedWriter(new FileWriter("symtab.txt"));
BufferedWriter br2 = new BufferedWriter(new FileWriter("littab.txt"));
BufferedWriter br3 = new BufferedWriter(new FileWriter("pool.txt"));
for (int i = 0; i < symtab.size(); i++)
  br1.write(symtab.get(i) + " " + symaddr.get(i) + "\n");

for (int i = 0; i < littab.size(); i++)
  br2.write(littab.get(i) + " " + litaddr.get(i) + "\n");
for (int i = 0; i < pooltab.size(); i++)
  br3.write(pooltab.get(i) + "\n");
br1.flush();
br2.flush();
br3.flush();
```

```
    br1.close();
    br2.close();
    br3.close();
  }
  private int getAddress(String string) {
    int temp = 0;
    if (string.contains("+")) {
      String sp[] = string.split("\\+");
      int ad = symaddr.get(symtab.indexOf(sp[0]));
      temp = ad + Integer.parseInt(sp[1]);
    } else if (string.contains("-")) {
      String sp[] = string.split("\\-");
      int ad = symaddr.get(symtab.indexOf(sp[0]));
      temp = ad - Integer.parseInt(sp[1]);
    }
    return temp;
  }
  public static void main(String[] args) throws Exception {
    PassOne p = new PassOne();
    p.createIS();
    p.generateIC();
  }
}
```

OUTPUT

intermediate.txt:

```
(AD, 01) (C, 501)
(IS, 09) (S, 01)
(IS, 09) (S, 02)
(IS, 04) (1) (S, 01)
(IS, 01) (1) (S, 02)
(IS, 05) (1) (S, 03)
(IS, 10) (S, 03)
(DL, 02) (C, 1)
(DL, 02) (C, 1)
(DL, 02) (C, 1)
(AD, 02)
```

symtab.txt:

A 507 1
B 508 1
C 509 1

Pass2.java

```
import java.io.*;
import java.util.*;
class Pass2 {
  public static void main(String[] Args) throws IOException {
    BufferedReader b1 = new BufferedReader(new FileReader("intermediate.txt"));
    BufferedReader b2 = new BufferedReader(new FileReader("symtab.txt"));
    BufferedReader b3 = new BufferedReader(new FileReader("littab.txt"));
    FileWriter f1 = new FileWriter("Pass2.txt");
    HashMap < Integer, String > symSymbol = new HashMap < Integer, String > ();
    HashMap < Integer, String > litSymbol = new HashMap < Integer, String > ();
    HashMap < Integer, String > litAddr = new HashMap < Integer, String > ();
    String s;
    int symtabPointer = 1, littabPointer = 1, offset;
    while ((s = b2.readLine()) != null) {
      String word[] = s.split("\t");
      symSymbol.put(symtabPointer++, word[1]);
    }
    while ((s = b3.readLine()) != null) {
      String word[] = s.split("\t");
      litSymbol.put(littabPointer, word[0]);
      litAddr.put(littabPointer++, word[1]);
    }
    while ((s = b1.readLine()) != null) {
      if (s.substring(1, 6).compareToIgnoreCase("IS,00") == 0) {
        f1.write("+ 00 0 000\n");
      } else if (s.substring(1, 3).compareToIgnoreCase("IS") == 0) {
        f1.write("+ " + s.substring(4, 6) + " ");
        if (s.charAt(9) == ')') {
          f1.write(s.charAt(8) + " ");
          offset = 3;
        } else {
```

```
          f1.write("0 ");
          offset = 0;
        }
        if (s.charAt(8 + offset) == 'S')
          f1.write(symSymbol.get(Integer.parseInt(s.substring(10 + offset, s.length() - 1))) +
"\n");
        else
          f1.write(litAddr.get(Integer.parseInt(s.substring(10 + offset, s.length() - 1))) + "\n");
      } else if (s.substring(1, 6).compareToIgnoreCase("DL,01") == 0) {
        String s1 = s.substring(10, s.length() - 1), s2 = "";
        for (int i = 0; i < 3 - s1.length(); i++)
          s2 += "0";
        s2 += s1;
        f1.write("+ 00 0 " + s2 + "\n");
      } else {
        f1.write("\n");
      }
    }
    f1.close();
    b1.close();
    b2.close();
    b3.close();
  }
}
```

OUTPUT

Pass2.txt: (machine code):

```
+ 09 0 507
+ 09 0 508
+ 04 1 507
+ 01 1 508
+ 05 1 509
+ 10 0 509
```

EXPERIMENT NO: 8

AIM: Write a program to implement Multipass macroprocessor

THEORY:

A multipass macroprocessor in system programming and compiler construction involves a process where source code is passed multiple times to handle macro expansion effectively. In this context, a multipass approach allow for more intricate processing of code, especially when dealing with macros and their expansion. Unlike single pass system that go through the code only once, multipass systems revisit the code to ensure accurate handling of macros and other language construct.

The concept of multipass macroproccesors is significant in system programming as it enables a more comprehensive analysis of source code, particularly when dealing with complex macro definition and invocation. By pasing code multiple times, the system can manage macro expansion effectively and efficiently, ensuring all necessary replacement and transformation are accurately executed.

for eg:

```
c program
# define square(x) x*x
# define cube(x) x*x*x

int main(){
```

```
int a=2;
int b= square(a) + cube(a);
return 0;
}
```

Pass 1: Macro expansion:

```
int main(){
 int a=2;
 int b= a* a + a* a *a;
 return 0;
 }
```

Pass 2: Code optimization:

```
int main(){
 int a=2;
 int b= a* (a+a*a);
 return 0;
 }
```

Pass 3: Code Generation:

LOAD	2, R_1	Load value 2 in R_1
MUL	R_1, R_1, R_2	Square R_1 and store in R_2
ADD	R_2, MUL R_1, R_3	Add square of R_1 to cube of R_1 and store in R_3

8/4/24

Pass 1:

```
class MDT_row:
    def __init__(self, index, code):
        self.index = index
        self.code = code

class MNT_row:
    def __init__(self, index, name, mdt_index):
        self.index = index
        self.name = name
        self.mdt_index = mdt_index

def get_macro_name_init_ALA(line, ALA):
    pos = 0
    name = ""
    while pos < len(line):
        while pos < len(line) and (line[pos] == ' ' or line[pos] == ','):
            pos += 1
        if pos < len(line) and line[pos] == '&':
            arg = ""
            while pos < len(line) and line[pos] != ' ' and line[pos] != ',':
                arg += line[pos]
                pos += 1
            ALA.append(arg)
        else:
            while pos < len(line) and line[pos] != ' ':
                name += line[pos]
                pos += 1
    return name
```

```
def search_ALA(ALA, arg):
    for i in range(len(ALA)):
        if ALA[i] == arg:
            return i
    return -1

def replace_dummy_args(line, ALA):
    res = ""
    pos = 0
    while pos < len(line):
        while pos < len(line) and (line[pos] == ' ' or line[pos] == ','):
            res += line[pos]
            pos += 1
        if pos < len(line) and line[pos] == '&':
            arg = ""
            while pos < len(line) and line[pos] != ' ' and line[pos] != ',':
                arg += line[pos]
                pos += 1
            index = search_ALA(ALA, arg)
            res = res + '#' + str(index)
        else:
            while pos < len(line) and line[pos] != ' ' and line[pos] != ',':
                res += line[pos]
                pos += 1
    return res

def print_MDT(MDT):
    print("---- Macro Definition Table ----")
    print("{:<7s}|{:<50s}".format("Index", "Card"))
    print("-------------------------------")
```

```
    for row in MDT:
        print("{:<7d}|{:<50s}".format(row.index, row.code))

def print_MNT(MNT):
    print("---- Macro Name Table ----")
    print("{:<7s}|{:<10s}|{:<7s}".format("Index", "Name", "MDT Index"))
    print("--------------------------------")
    for row in MNT:
        print("{:<7d}|{:<10s}|{:<7d}".format(row.index, row.name, row.mdt_index))

def print_ALA(ALA):
    print("---- Argument List Array ----")
    print("{:<7s}|{:<10s}".format("Index", "Arguments"))
    print("--------------------------------")
    for i, arg in enumerate(ALA):
        print("{:<7d}|{:<10s}".format(i, arg))

def print_output(output):
    print("---- Output of Pass 1 ----")
    for line in output:
        print(line)

if __name__ == "__main__":
    inputfile = open("macro.asm", "r")
    output = []
    MDT = []
    MNT = []
    ALA = []
    inside_macro = False
    macro_name_line = False
```

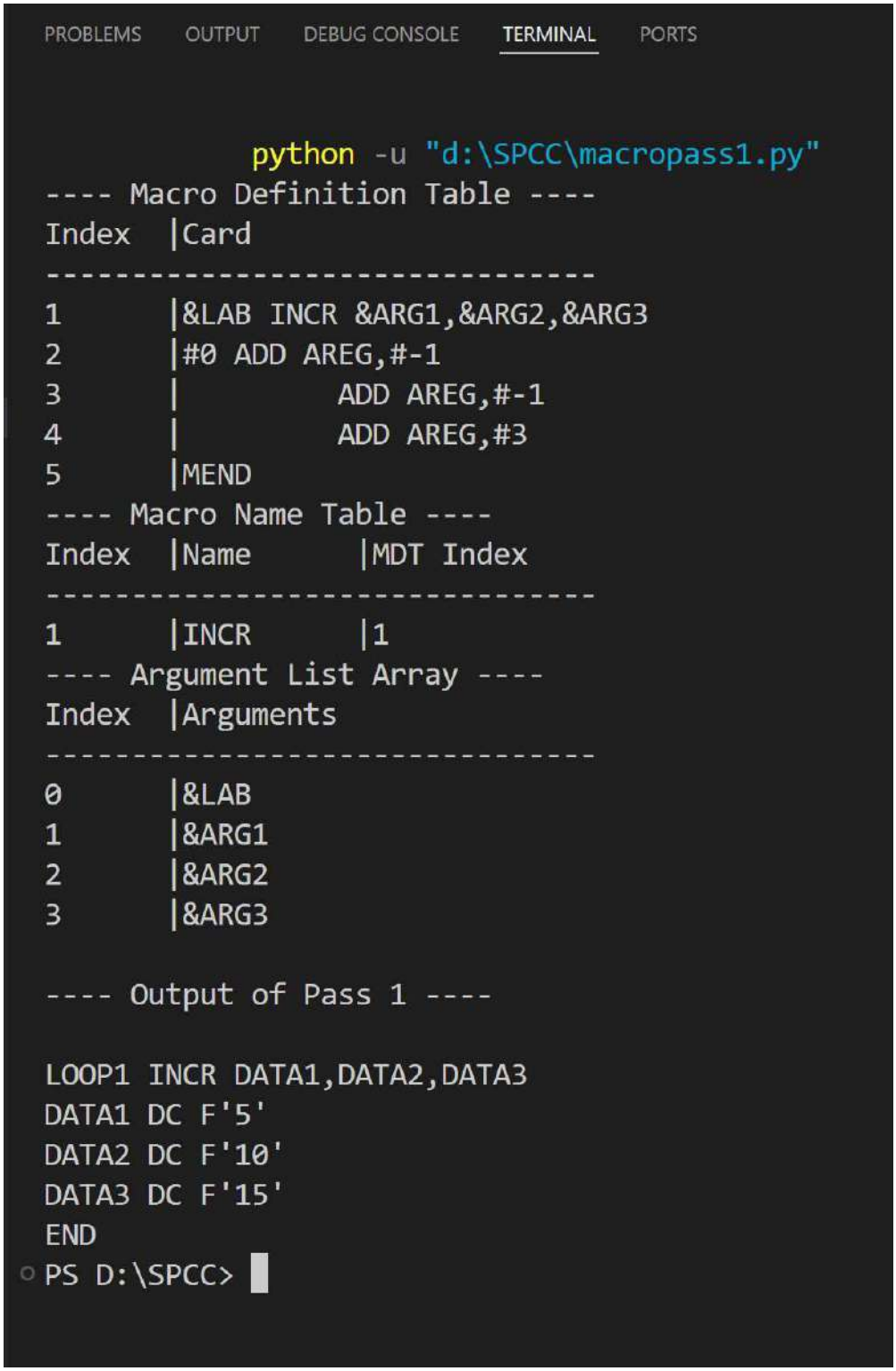
```
mdtc = 1
mntc = 1
for line in inputfile:
    if "MACRO" in line.upper():
        inside_macro = True
        macro_name_line = True
    elif "MEND" in line.upper():
        row = MDT_row(mdtc, line.strip())
        mdtc += 1
        MDT.append(row)
        inside_macro = False
    elif not inside_macro:
        output.append(line.strip())
    elif inside_macro:
        if macro_name_line:
            row = MNT_row(mntc, "", mdtc)
            mntc += 1
            row.name = get_macro_name_init_ALA(line, ALA)
            MNT.append(row)

            entry = MDT_row(mdtc, line.strip())
            mdtc += 1
            MDT.append(entry)
            macro_name_line = False
        else:
            row = MDT_row(mdtc, replace_dummy_args(line, ALA))
            mdtc += 1
            MDT.append(row)

inputfile.close()
```

print_MDT(MDT)

print_MNT(MNT)

print_ALA(ALA)

print_output(output)



Pass 2:

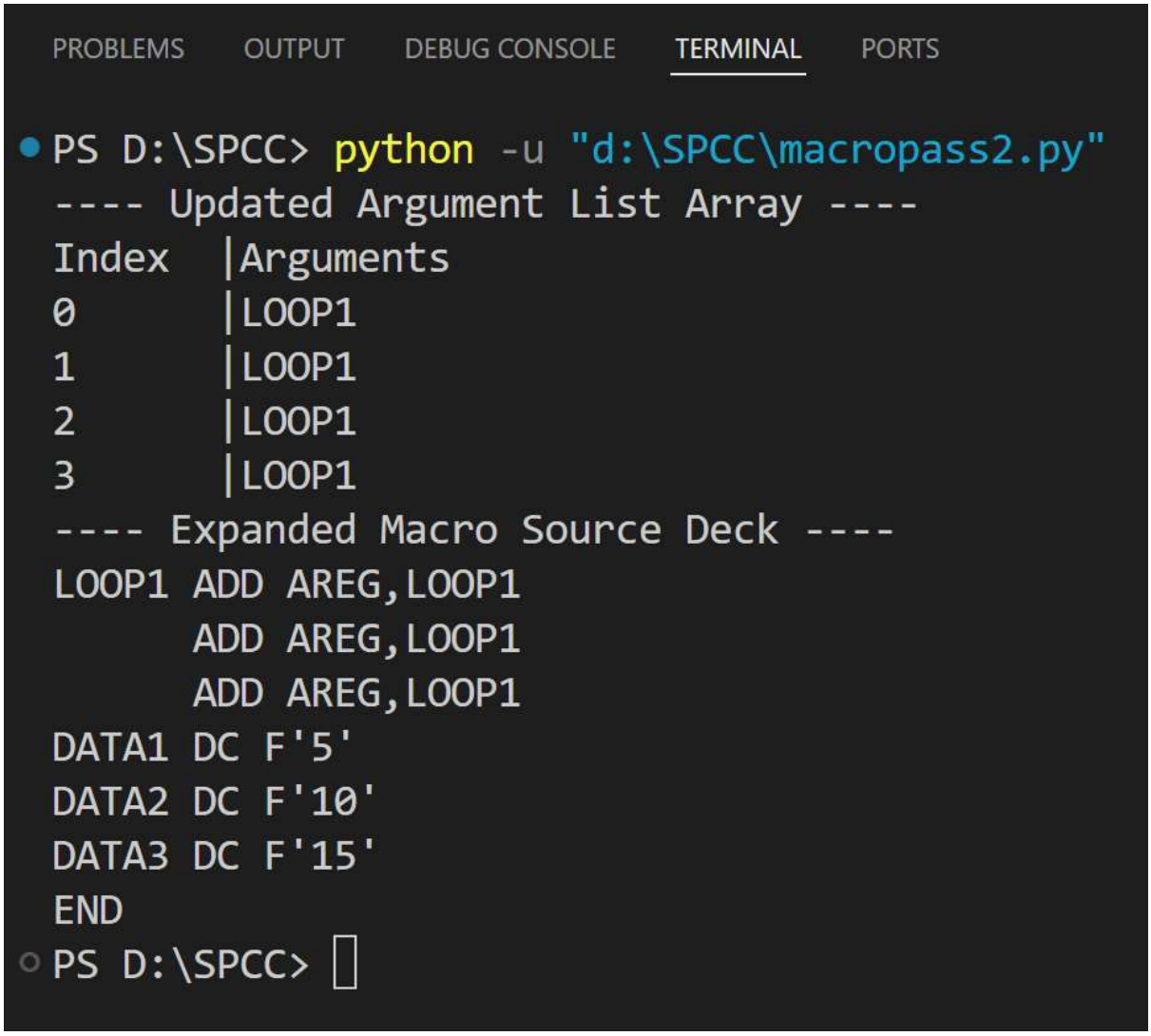
```
class MDT_row:
    def __init__(self, index, code):
        self.index = index
        self.code = code

class MNT_row:
    def __init__(self, index, name, mdt_index):
        self.index = index
        self.name = name
        self.mdt_index = mdt_index

def init_MDT(MDT):
    mdtc = 1
    code_list = [
        "&LAB INCR &ARG1,&ARG2,&ARG3",
        "#0 ADD AREG,#1",
        "      ADD AREG,#2",
        "      ADD AREG,#3",
        "      MEND"
    ]
    for code in code_list:
        row = MDT_row(mdtc, code)
        MDT.append(row)
        mdtc += 1

def init_MNT(MNT):
    mntc = 1
    row = MNT_row(mntc, "INCR", 1)
    MNT.append(row)
```

```
def init_ALA(ALA):
    ALA.extend(["&LAB", "&ARG1", "&ARG2", "&ARG3"])

def search_mnt(word, MNT):
    for i, row in enumerate(MNT):
        if row.name == word:
            return i
    return -1

def get_mnt_row(line, MNT):
    pos = 0
    while pos < len(line):
        while pos < len(line) and (line[pos] == ' ' or line[pos] == ','):
            pos += 1
        word = ""
        while pos < len(line) and line[pos] != ',' and line[pos] != ' ':
            word += line[pos]
            pos += 1
        index = search_mnt(word, MNT)
        if index != -1:
            return index
    return -1

def expand_macro(row, MDT, output, ALA, line):
    mdtp = row.mdt_index
    pos = 0
    dummy_params = []
    actual_params = []
    while pos < len(line):
```

```
    while pos < len(line) and (line[pos] == ' ' or line[pos] == ','):
        pos += 1
    word = ""
    while pos < len(line) and line[pos] != ',' and line[pos] != ' ':
        word += line[pos]
        pos += 1
    if word != row.name:
        actual_params.append(word)
line = MDT[mdtp - 1].code
mdtp += 1
pos = 0
while pos < len(line):
    while pos < len(line) and (line[pos] == ' ' or line[pos] == ','):
        pos += 1
    if line[pos] == '&':
        arg = ""
        while pos < len(line) and line[pos] != ',' and line[pos] != ' ':
            arg += line[pos]
            pos += 1
        dummy_params.append(arg)
    else:
        pos += 1
for actual_param in actual_params:
    for j, dummy_param in enumerate(dummy_params):
        if dummy_param in ALA:
            ALA[ALA.index(dummy_param)] = actual_param
while True:
    pos = 0
    line = MDT[mdtp - 1].code
    if "MEND" in line.upper():
```

```
            break
        res = ""
        while pos < len(line):
            while pos < len(line) and (line[pos] == ' ' or line[pos] == ','):
                res += line[pos]
                pos += 1
            if line[pos] == '#':
                pos += 1
                index = ""
                while pos < len(line) and line[pos] != ',' and line[pos] != ' ':
                    index += line[pos]
                    pos += 1
                res += ALA[int(index)]
            else:
                res += line[pos]
                pos += 1
        output.append(res)
        mdtp += 1

def print_ALA(ALA):
    print("---- Updated Argument List Array ----")
    print("{:<7s}|{:<10s}".format("Index", "Arguments"))
    for i, arg in enumerate(ALA):
        print("{:<7d}|{:<10s}".format(i, arg))

def print_output(output):
    print("---- Expanded Macro Source Deck ----")
    for line in output:
        print(line)
```

```
if __name__ == "__main__":
    MDT = []
    MNT = []
    ALA = []
    output = []
    init_MDT(MDT)
    init_MNT(MNT)
    init_ALA(ALA)

    with open("input.asm", "r") as inputfile:
        for line in inputfile:
            mnt_index = get_mnt_row(line, MNT)
            if mnt_index != -1:
                expand_macro(MNT[mnt_index], MDT, output, ALA, line)
            else:
                output.append(line.strip())

    print_ALA(ALA)
    print_output(output)
```



Input to Pass 1: (macro.asm)

```
MACRO
&LAB INCR &ARG1,&ARG2,&ARG3
&LAB ADD AREG,&ARG1
        ADD AREG,&ARG2
        ADD AREG,&ARG3
        MEND

LOOP1 INCR DATA1,DATA2,DATA3
DATA1 DC F'5'
DATA2 DC F'10'
```

```
DATA3 DC F'15'
END
```

Input to Pass 2: (input.asm)

```
LOOP1 INCR DATA3,DATA2,DATA1
        DATA1 DC F'5'
        DATA2 DC F'10'
        DATA3 DC F'15'
        END
```

Additional Experiment

AIM: Write a program to implement Code Generation

THEORY:

Code Generation can be considered the final phase of compilation. Though post code optimization process, can be applied on the code but that can be seen as a part of code generation phase itself. The code generated by compiler is an object code of some low level programming language for eg: assembly language.

Regist and Address Description:
→ Register description contains track of what is currently in each register. It is used to store the operands of three Address Statements.

→ An address description is used to store the location where current value of registers are initially empty.

ALGORITHM:

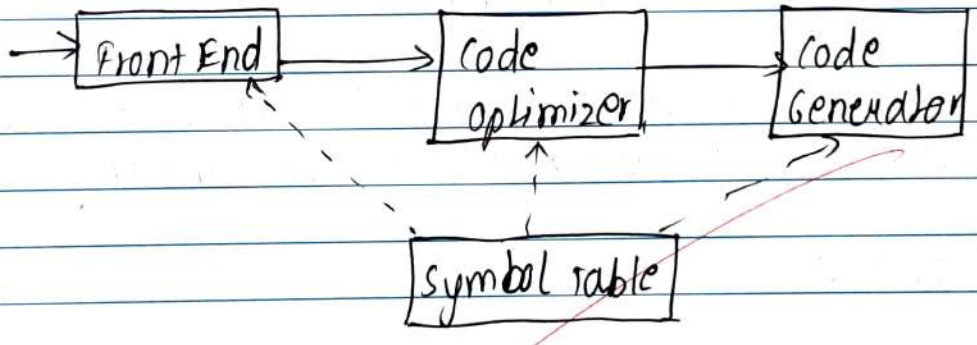
1. Involve a function getReg to find out the location, the result of compilation b op c should be stored.

2. Consult the address description for y to determine 'y'. If the value of y currently in memory and registers both prefer 'y'. If 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 of z', L where z' is used to show the current location of z. If z is in both then prefer a register to memory location in. If x in L then update its descriptor and remove x from all descriptors.

4. If current value of y or z have no next uses or not live on exit from the block or register then after the regista to indicate that after execution of x:= y op z those register will no ogner contain y or z.



13/4/24

```
def generate_code_3addr(statements):
    register_descriptor = {}
    address_descriptor = {}

    for statement in statements:
        parts = statement.split(" := ")
        destination = parts[0].strip()
        expression = parts[1].strip()

        # Code generation
        code_generated = []
        operands = expression.split()
        op = operands[1]
        if op == '-':
            opr = 'SUB'
        elif op == '+':
            opr = 'ADD'
        elif op == '*':
            opr = 'MUL'
        elif op == '/':
            opr = 'DIV'
        code_generated.append(f"MOV {operands[0]}, R{len(register_descriptor)}")
        code_generated.append(f"{opr} {operands[2]}, R{len(register_descriptor)}")

        # Register descriptor
        register_descriptor[destination] = f"R{len(register_descriptor)}"

        # Address descriptor
        address_descriptor[destination] = f"{destination} in R{len(register_descriptor)}"
```

```
        print(f"Statement: {statement}\n")
        print("Code Generated:")
        for code_line in code_generated:
            print("    " + code_line)
        print("\nRegister Descriptor:")
        print(f"    {register_descriptor[destination]} contains {destination}\n")
        print("Address Descriptor:")
        print(f"    {address_descriptor[destination]}\n")
        print("-" * 50)

# Example usage:
three_address_code = [
    "t := a - b",
    "u := a - c",
    "v := t + u",
    "d := v + u"
]

generate_code_3addr(three_address_code)
```

Output

```
Statement: t := a - b

Code Generated:
    MOV a, R0
    SUB b, R0

Register Descriptor:
    R0 contains t

Address Descriptor:
    t in R1

-------------------------------------------------
Statement: u := a - c

Code Generated:
    MOV a, R1
    SUB c, R1

Register Descriptor:
    R1 contains u

Address Descriptor:
    u in R2

-------------------------------------------------
Statement: v := t + u

Code Generated:
    MOV t, R2
    ADD u, R2

Register Descriptor:
    R2 contains v

Address Descriptor:
    v in R3

-------------------------------------------------
Statement: d := v + u

Code Generated:
    MOV v, R3
    ADD u, R3

Register Descriptor:
    R3 contains d

Address Descriptor:
    d in R4

-------------------------------------------------
```

Additional
EXPERIMENT NO: 2

AIM: write a program to eliminate left recursion from the given grammar.

THEORY:

Left recursion is a common problem that occurs in grammar during parsing in the syntax analysis part of compilation. It is important to remove left recursion from grammar because it can create an infinite loop, leading to errors and a significant decrease in performance.

- **Ambiguous Grammar:**
  For a given grammar, we can generate atleast one string which can be presented using more than one parse tree than such grammar is called ambiguous grammar.

- **Algorithm for Removing Left Recursion:**

A grammar is left recursive if it has a non terminal A such that there is a derivation

$$A \longrightarrow A\alpha \text{ for some string } \alpha$$

Top down parsing methods cannot handle left-recursive grammar.

Hence left recursion can be eliminated as follows:

Left Recursion: $A \longrightarrow A\alpha \mid \beta$

Then, $A \longrightarrow \beta A'$
$A' \longrightarrow \alpha A' \mid \epsilon$

Arrange non-terminals in some order $A_1, A_2 \ldots, A_n$

```
for i=1 to n do begin
    for j=1 to i-1 do begin
        replace each production Ai → Ajy by Ai → δ1y | δ2y ... | δky
        where Aj → δ1 | δ2 | ... | δk are all current Aj productions;
    end
end
```

for eg: $S \longrightarrow A$
$A \longrightarrow Ad \mid Ae \mid aB \mid ac$
$B \longrightarrow bBc \mid f$

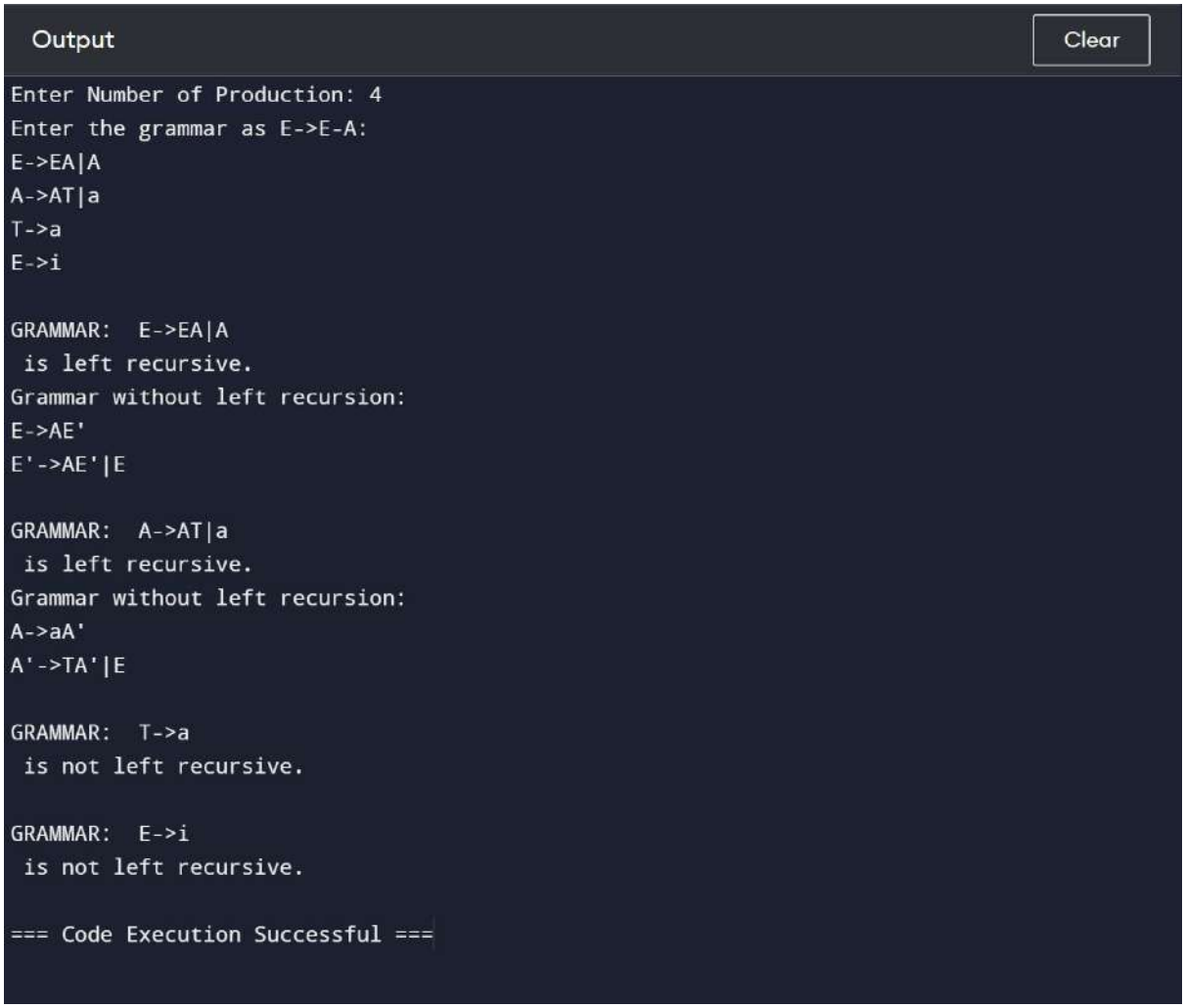
Left Recursion in:
$A \longrightarrow Ad$ $\qquad$ $A \longrightarrow Ae$

then,
$A \longrightarrow aBA' \mid acA'$
$A' \longrightarrow dA' \mid eA' \mid \epsilon$

then,
$A \longrightarrow aBA' \mid acA'$
$A' \longrightarrow dA' \mid eA' \mid \epsilon$

8/4/24

```
num = int(input("Enter Number of Production: "))
print("Enter the grammar as E->E-A:")
productions = []
for i in range(num):
    productions.append(input())

for production in productions:
    print("\nGRAMMAR: ", production)
    non_terminal = production[0]
    index = 3  # starting of the string following "->"
    if non_terminal == production[index]:
        alpha = production[index + 1]
        print(" is left recursive.")
        while index < len(production) and production[index] != '|' and production[index]
!= '\0':
            index += 1
        if index < len(production):
            beta = production[index + 1]
            print("Grammar without left recursion:")
            print(f"{non_terminal}->{beta}{non_terminal}'")
            print(f"{non_terminal}'->{alpha}{non_terminal}'|E")
        else:
            print(" can't be reduced")
    else:
        print(" is not left recursive.")
```


Output
Clear
Enter Number of Production: 4
Enter the grammar as E->E-A:
E->EA|A
A->AT|a
T->a
E->i
GRAMMAR: E->EA|A
is left recursive.
Grammar without left recursion:
E->AE'
E'->AE'|E
GRAMMAR: A->AT|a
is left recursive.
Grammar without left recursion:
A->aA'
A'->TA'|E
GRAMMAR: T->a
is not left recursive.
GRAMMAR: E->i
is not left recursive.
=== Code Execution Successful ===

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Path ID. C12 32

ASSIGNMENT NO:1

Q1] With reference to assembler, consider any assembly program and generate Pass1 and Pass2 output. Also show content of DB table involved in it.

⇒ Assembly language is a low level programming language that is specific to compiler architecture or processor. Assembly language assembler is a software tool that translates assembly language code into machine code, which can be executed by Machine CPU. It performs this translation process in two main phases, phase 1 and phase 2.

Two passes assembly process:

Two pass assembly program is a common method used by assembler to translate assembly language source code into machine code. It involves two main process over into source code.

Pass-1 → Scans the code, validates the tokens create symbol table.

i) It keeps track of location under (LC).
ii) Determines the length of machine instructions. (MOT).
iii) keeps track of value of symbol until Pass-2 is done. (ST).
iv) Process some pseudocode eg: EQU, DS, DC (POT)
v) Stores the literals (LT).

Pass-2 → Convert the code to machine code, solves the forward reference problem.

i) It looks up values of symbol (ST).

Pass I Output

Source Code	Intermediate Code (IC)
START 200	(AD,01) - (C,200)
MOVER AREG,A	(200) (IS,01) 01 (S,01)
L1: ADD BREG,B	(201) (IS,03) 02 (S,03)
PRINT B	(202) (IS,07) - (S,03)
READ='5'	(203) (IS,08) - (L,01)
PRINT='15'	(204) (IS,07) - (L,02)
LTORG	(205) (AD,05) - 005
MVL CREG,X	(206) (AD,05) - 015
SUB BREG='5'	(207) (IS,05) 03 (S,04)
DIV CREG LOOP	(208) (IS,04) 02 (L,03)
LOOP: STORE X	(209) (IS,06) 03 (S,05)
ORIGIN L1+30	(210) (IS,10) - (S,04)
A DS 3	(231) (D,01) - 003
LTORG	(234) (AD,05) - 005
MOVER BREG,X	(235) (IS,01) 02 (S,04)
X DC 2	(236) (DL,2) - 002
B DS 2	(237) (DL,01) - 002
END	(239) (AD,02) - -

Symbol Table

Sym_no	Symbol Name	Address
01	A	231
02	L1	201
03	B	237
04	X	236
05	LOOP	210

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

Literal Table

Literal No	Literal name	Address
01	='5'	205
02	='15'	206
03	='5'	234

POOL TABLE

01
03

MOT

opcode	Mnemonic
1	MOVER
2	MOVEM
3	ADD
4	SUB
5	MUL
6	DIV
7	PRINT
8	READ
9	BC
10	STORE

POT

opcode	Mnemonic
1	START
2	END
3	ORIGIN
4	EQU
5	LTORG

Registers

No	Name
1	AREG
2	BREG
3	CREG
4	DREG

DL

1	DS
2	DC

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

Pass - II

IC	Machine code/ Target Code		
(AD,01) - (C,200)	-	-	-
200) (IS,01) 01 (S,01)	200) 01	01	231
201) (IS,03) 02 (S,03)	201) 03	02	237
202) (IS,07) - (S,03)	202) 07	-	237
203) (IS,08) - (L,01)	203) 08	-	205
204) (IS,07) - (L,02)	204) 07	-	206
205) (AD,05) - (005)	205) -	-	005
206) (AD,05) - (015)	206) -	-	015
207) (IS,05) 03 (S,04)	207) 05	03	236
208) (IS,04) 02 (L,03)	208) 04	02	237
209) (IS,06) 03 (S,05)	209) 06	03	210
210) (IS,10) - (S,04)	210) 10	-	236
231) (DL,01) - 003	231) -	-	-
234) (AD,05) - 005	234) -	-	005
235) (IS,01) 02 (S,04)	235) 01	02	236
236) (DL,2) - 002	236) -	-	002
237) (DL,01) - 002	237) -	-	-
239) (AD,02) - -	239) -	-	-

8/4/12 A+

Pass-2 Output:

```
PRG       START     0
LOOP 1    A         1, DATA 1
          A         2, DATA 2
          A         3, DATA 3

DATA 1    DC        F'5'
DATA 2    DC        F'10'
DATA 3    DC        F'15'
```

(1) Argument List Array (ALA):

Index	Argument
0	LOOP 1
1	DATA 1
2	DATA 2
3	DATA 3

Q2] Write a short note on YACC.

⇒ YACC or Yet Another Computer Compiler is a powerful used in field of compiler construction to automate the process of generating parser for programming language. It plays crucial role in transforming high level language specification, typically written in formal grammar notation such as BNF into efficient parser capable of recognizing and analyzing the structure of input program YACC functionality and how its fits into compiler construction process:-

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

1) Formal Grammar specification:
YACC token input a formal specification of syntax rules for programming language. This specification define the language syntax in terms of production rules which describe how valid sentence or program can be considered constructed from language symbol.

2) Lexer Integration:
YACC work in conjugation with lexical tool such as lex flex. Lexer are responsible for breaking down the input token or lexemes, which are smallest meaningful unit the language.

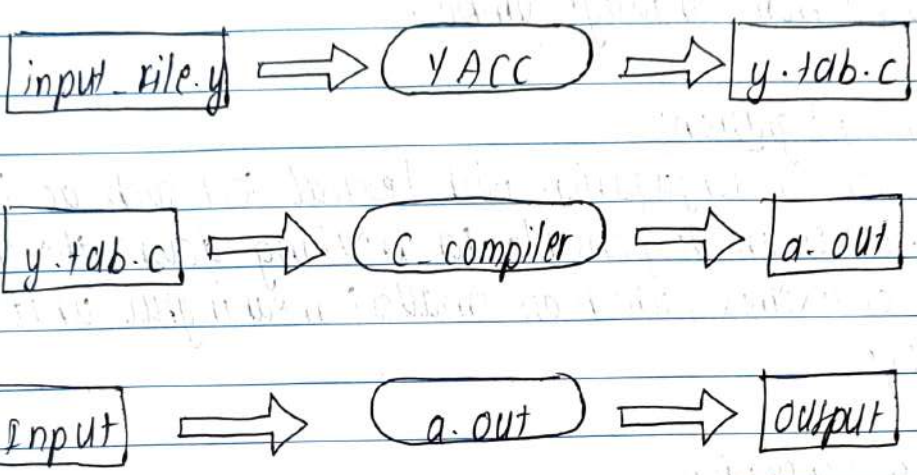
3) Parser Generation:
YACC generates a parser for specified grammar based on principles of LR Parsing. LR parsing is a bottom up parsing technique that builds a parse tree for input string starting from leaves and working up to root.

4) Parsing Algorithm:
YACC parser implement an efficient parsing algorithm that navigate through grammar and input token whenever input confirm to language syntax.

5) Error Handling:

YACC parser can be augmented with error handling mechanism to detect and report syntax error in input. Error recovery strategies can also be determined to

help the parse continue its parsing after encountering error, ensuring that multiple syntax error in program do not halt the parsing process prematurely.



YACC Compiler Process.

8/4/24

ASSIGNMENT NO: 2

Q]- Explain working of Direct Linking Loader with example showing entries in different databases built in DLL.

⇒ The Direct Linker Loader (DLL) is a re-located loader. The DLL is a general relocating loader and the most popular loading scheme presently used. The scheme has an advantage that it allows the programmer to use multiple procedures and multiple data segments. In addition, the programmer is free to reference data or instructions that are contained in other segments. The direct linking loader provide flexible intersegment referencing and accessing ability.

Working:

1] Loading DLLs:
when a program starts execution, the DLLs it depends on are not loaded into the memory immediately. Instant only the code necessary to start are loaded dynamically.

2) Loading Process:
The program requests the operating system to load a DLL. The OS checks if the DLL is already else, it is loaded. The linker resolves the symbol that refer to function of variables in DLL.

3) Symbol Resolution:
The program loads a function or reference a variable

Parth D. C12

defined in a DLL, the OS linker resolves the symbol to corresponding address in DLL.

4. Binding Address :
The loader binds the address of internal symbols directly into the program's code or data sections. This binding involves replacing placeholder addresses with the actual addresses of the symbols. Since the addresses are between known at the compile time, the loader can directly embed them into the program.

DLL Structure:

- database.dll :→ contains functions for database operations.
- add_entry (database, entry) :- Add entry to specified DB.
- update_entry (database, new_data) :- updates entry in specified DB.
- delete_entry (database, entry_id) : Deletes entry from specified DB.

Main Program:

- Main program.exe :- use database for database operations.
- It dynamically loads dll at runtime.
- It calls function from database.dll to perform operation on different database. DLL to perform operation on different database.

Usage:

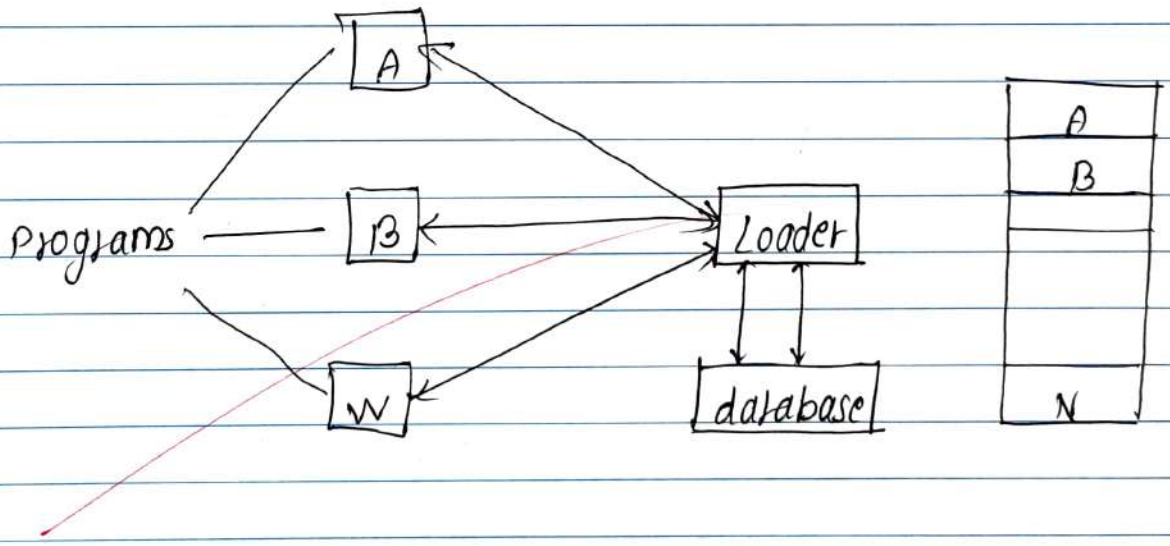
THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

- When main-program.exe start, it does load database.dll immediately.

- When program needs to perform a database operation such as adding or entry to MySQL, it dynamically loads database.dll.

eg: // main_program.exe

```
load library ("database.dll");
add_entry ("MySQL", new_entry);
update_entry ("MongoDB", entry_id, updated_data)
delete_entry ("SQLite", entry_id_to_delete);
FreeLibrary ("data.dll");
```



8/4/24 A+

Out of Syllabus Assignment

Q 1]. Editors and type of editors.

Editors or text editors are software programs that enable the user to create and edit text files. In the field of programming, the term editor usually refers to source code or editors that include many special features for writing and editing code. Notepad, Wordpad are some common editors used on Windows OS and vi, emacs, jed, pico are the editors on UNIX OS. Features normally associated with text editors are- moving the cursor, deleting, replacing, pasting, finding, finding and replacing, saving etc.

Types of editors:

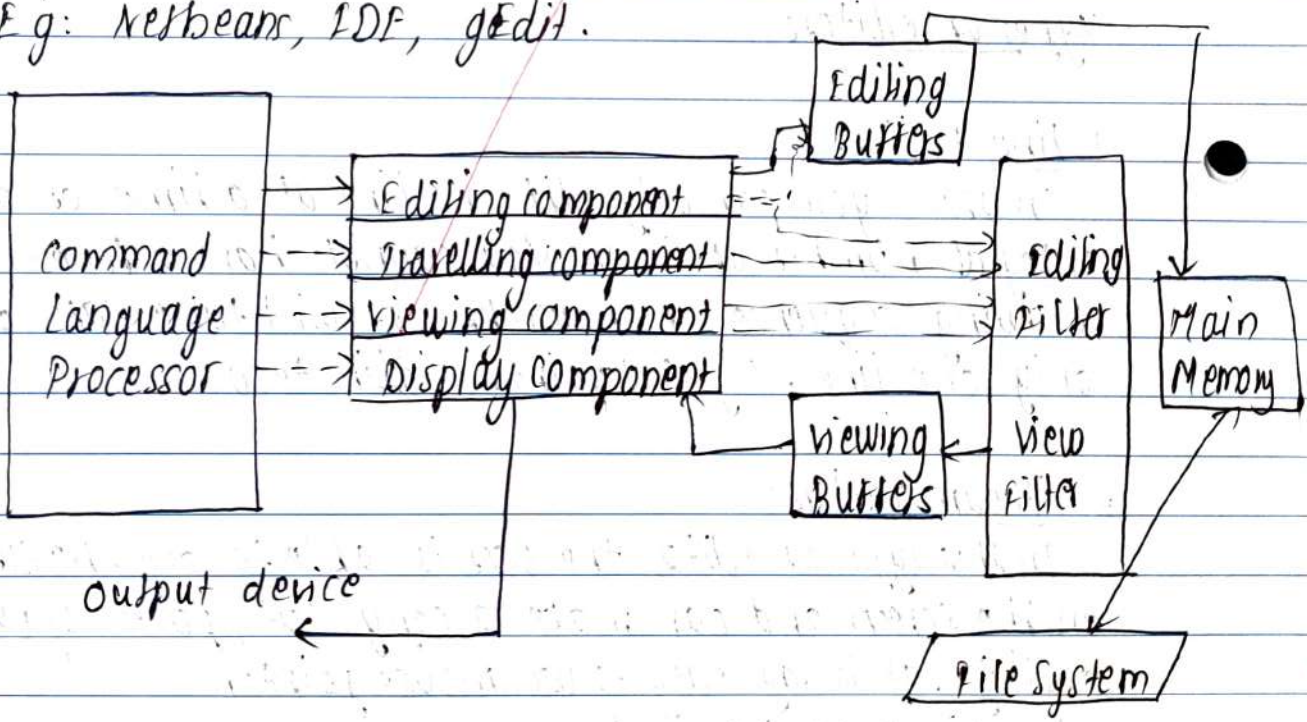
1. Line Editor:
In this, you can only edit one line at a time or an integral number of lines. You cannot have a free-flowing sequence of characters. It will take care of only one line. Eg: Teleprinter, edlin, teco.

2. Stream editors:
In this type of editors, the user is able to see the cursor on the screen and can make a copy, cut, paste operation easily. It is very easy to use mouse pointer.
Eg: vi, emacs, Notepad.

3. Screeneditors:
In this type of editors, the file is created and treated as continuos flow or sequence of characters instead of line numbers, which means here you can type paragraphs.
Eg: sed Editors in UNIX.

4. Word Processor:
Overcoming the limitations of screen editors, it allows one to use some format to insert images, files, videos usefont, size, style features. It majorly focusses on natural language.

5. Structure Editor:
Structure editor focusses on programming languages. It provide features to write and edit source code.
Eg: Netbeans, IDE, gEdit.



Q2] Backpatching in Intermediate Code Generation.

Backpatch is basically a process of palfilling unspecified information. This information is of labels. It basically uses the appropriate semantic actions during the process of code generation. It may indicate the address of the label in goto statements while producing TACs.
For the given expressions. Here basically the passes are used because alligning the positions of these labels statements in one pass is quite challenging. It can leave these addresses unidentified in the first pass and then populate them in the second round.

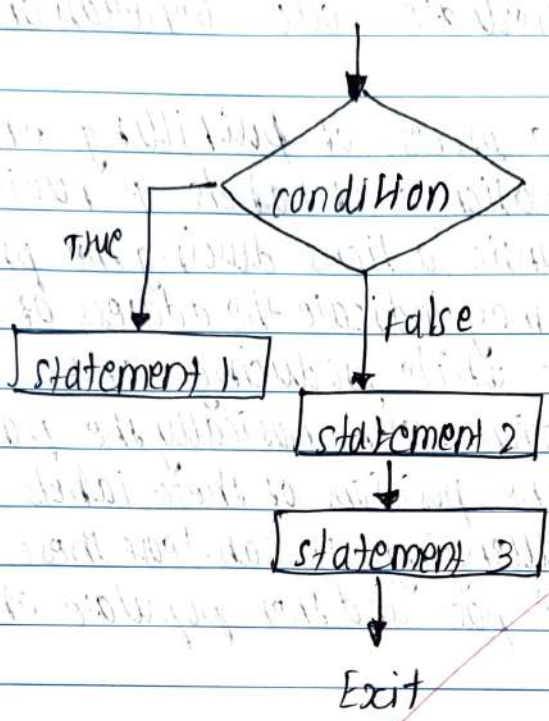
Need for backtracking:

a) Boolean Expression:
statement whole results are either True or False. A boolean expression which is named after Mathematician George Boole, is an expression that evaluates to either True or False.

b) How to control statements:

The flow of control statements needs to be controlled during the execution of statement in a program.

Eg:


condition
True
false
statement 1
statement 2
statement 3
Exit

8/1/24 (A+)