Thadomal Shahani Engineering College

Bandra (W.), Mumbai - 400 050.

CERTIFICATE

Certify that Mr./~~Miss~~ Shirish Shetty

of COMPS Department, Semester VI with

Roll No. 2103164 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

28/03/2024

Teacher In- Charge

Head of the Department

Date 28/03/24

Principal

THADOMAL SHAHANI TSEC ENGINEERING SOLLEGE

CONTENTS

TSEC
ENGINEERING COLLEGE

Experiment No 1

Aim : Lexical Analysis program using Python

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 preprocessor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by moving whitespaces 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 data is passed to the syntax analyzer, when it is asked for

There are three terminologies :-

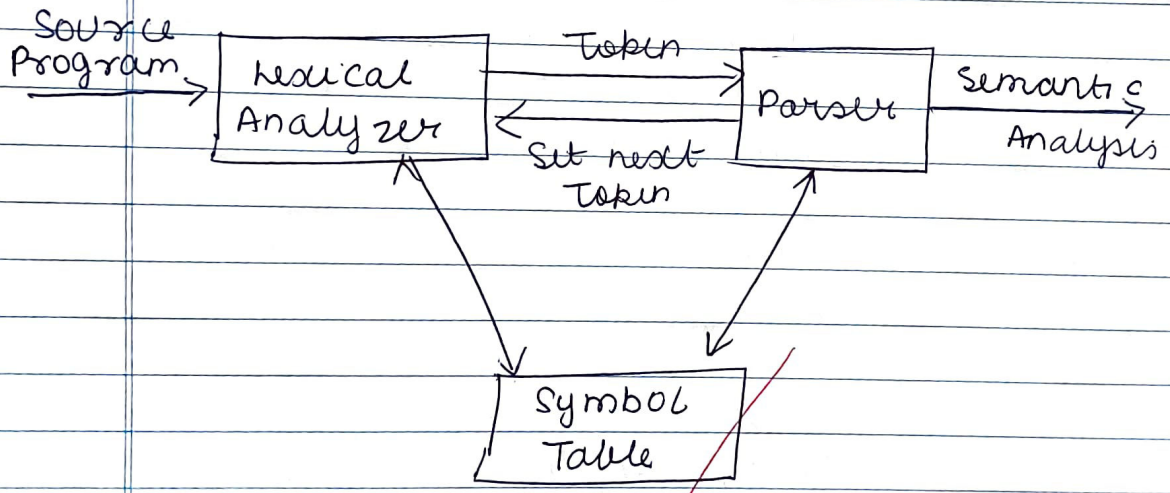
A) TOKEN :- It is sequence of characters that represent a unit of information in source code.

B) PATTERN :- The description used by the token is known as a pattern.

C) LEXEME :- A sequence of character in source code, as per the matching pattern of a token, is known as lexeme. It is also the instance of a token

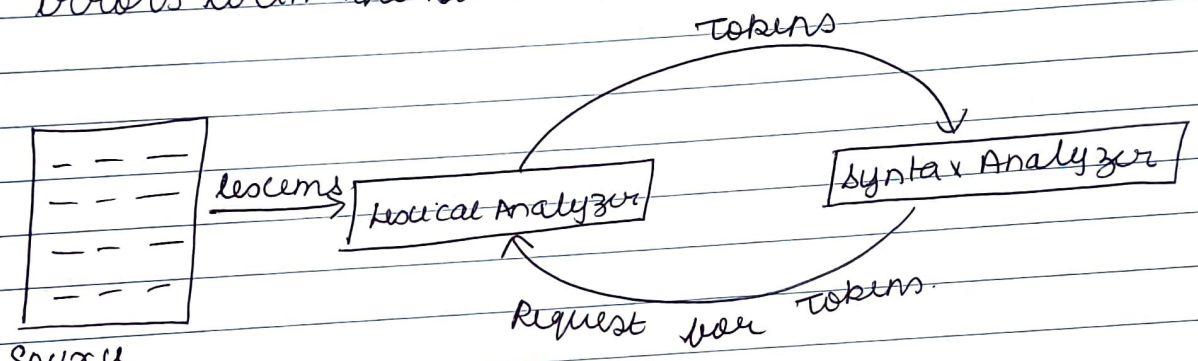
Example

Token	Lexeme	Pattern
1.) Keyword	while	wh-i-l-e
~~2.) Relop~~	~~<~~	~~<, >, =, <=, !=, ==~~
3.) Integer	7	(0-9)* ⇒ Sequence digit with at least one digit.
4.) String	"Hi"	characters enclosed by " "
5.) Punctuation	;	; , . ! etc.
6.) Identifier	number	A-Z, a-z



Architecture of Lexical Analyzer.

To read the input character in the source code and produce a token is the most important task of a lexical analyzer. The lexical analyzer goes through with the entire source code and indentifies each token one by one. The scanner is responsible to produce tokens when it is requested by parser. The lexical analyzer avoids the white-space and comments while creating these tokens. If any error occurs, the analyzer correlates these errors with the source file and line number.



Roles of lexical Analyzer

- The lexical analyzer is responsible for removing the white spaces and comments from the source program.
- It corresponds to error messages with the source program.
- It help to identify the tokens.
- The input characters are read by lexical analyzer from the source code

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Advantages

- It help the browser to format and display a web page with help of parsed data
- It help to create a more efficient and specialized processor for the task.

Disadvantages

- It requires additional runtime overhead to generate the lexer table & construct the tokens.

Conclusion :-

The lexical Analyzer is crucial for misunderstanding and organizing code by breaking it into smaller segments or parts as token. It carefully examins each characters to help compiler proces the code correctly.

QA
22/02/2024

```
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
```

Experiment No 2.

Aim :- To study and implement program LEX and YACC tool.

Theory :-

Lex is a lexical analyzer generator used to create programs that recognize and process tokens in input text. It takes a set of regular expressions as input and generates a code for lexical analyzer. This tool simplifies the creation of parsers by automatically generating code to identify and categorize lexical element in a given language. Lex is often paired with Yacc parser generator to build complete language processing system.

→ INSTALLATION AND COMPILATION OF LEX & YACC

1) Sudo apt-get update
2) Sudo apt-get install flex
3) sudo apt-get install bison

```
>> lex program_name.l
>> gcc lex.yy.c -lfl
>> ./a.out
```

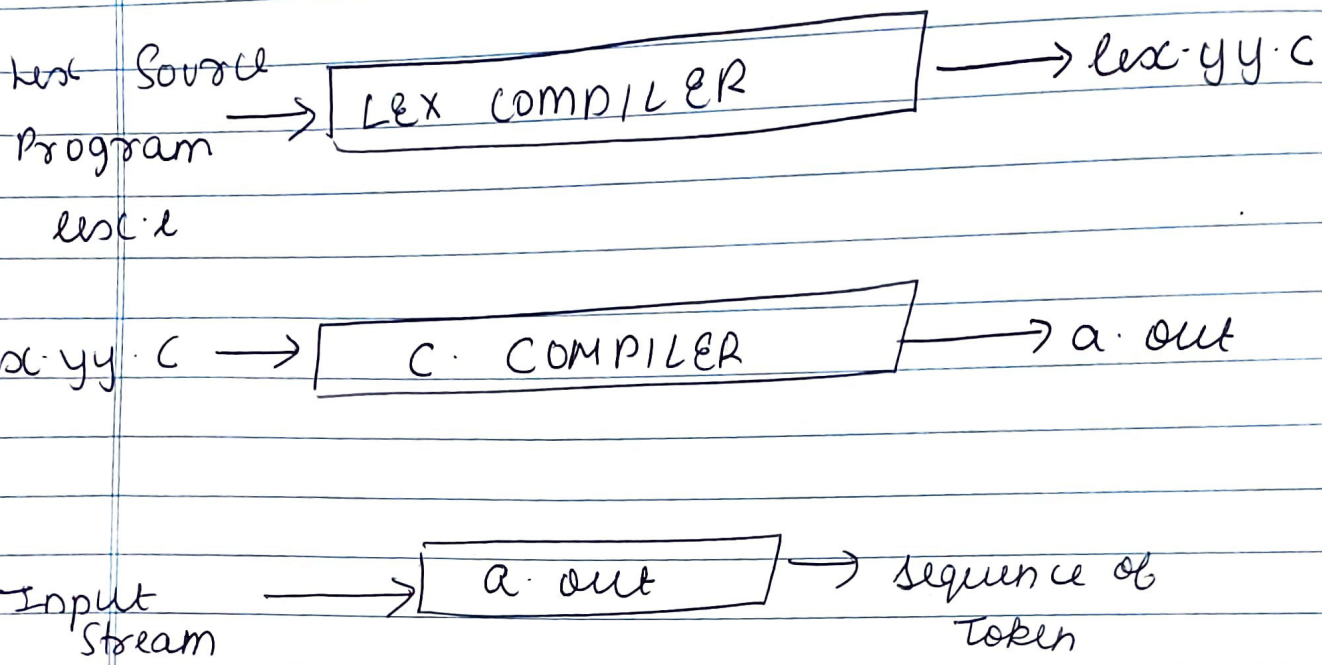


Fig) LEX COMPILER

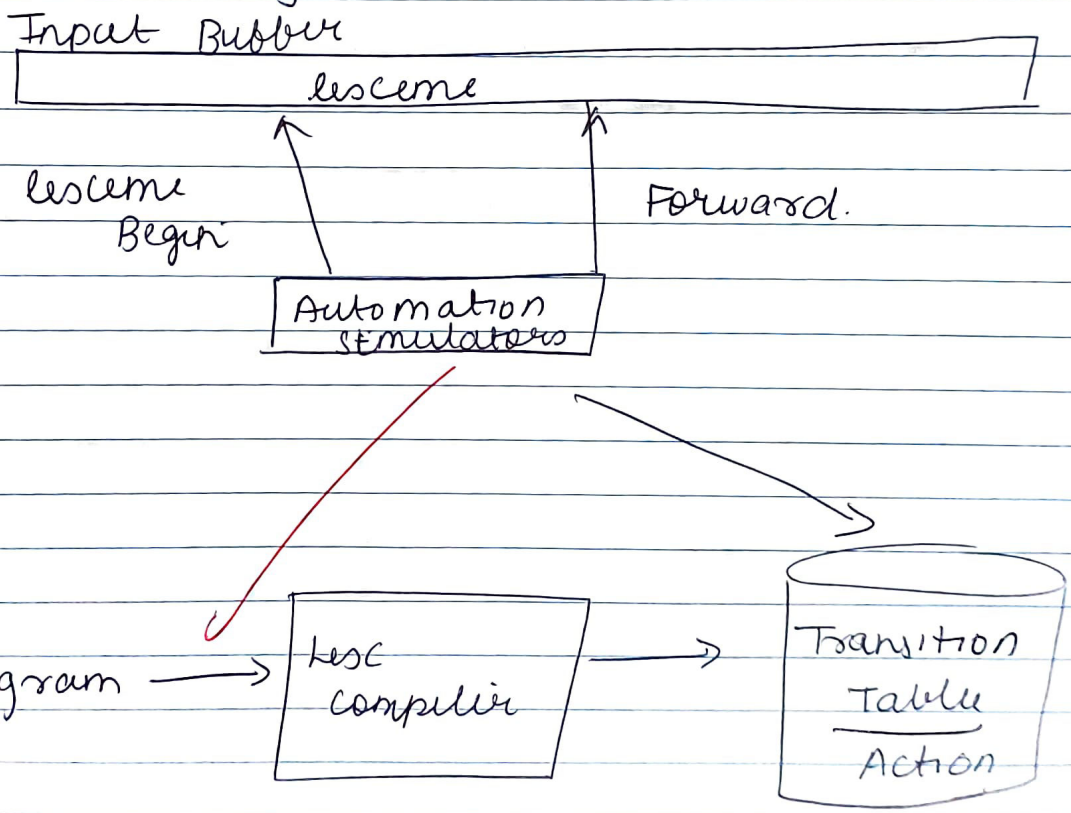


Fig) → Lexical Analyzer generator

Design of lexical Analyzer Generator.

- A lex program is turned into transition table & action which are used by finite automation simulator.
- The program that serves as the lexical analyzer includes a fixed program that simulates an automaton.
- The rest of lexical analyzer consist of following component.
  1.) A transition table of the automation
  2.) Those function that are passed directly through Lex to input.
  3.) The action from input program which is to be invoked at appropriate time by automation simulator

- To construct the automation take each regular expression pattern in LEX program and convert it using algorithm to NFA.
- We need a single automation that will recognize lexemes matching any of pattern pi

→ Structure of LEX program

lex program consist of three parts

```
%{                          Rule section
Declaration                 %%
%}
                            Auxiliary Procedure
%%
```

Program Description.

- In first esp, we need to print HelloWorld so in Rule section, we need to define on receiving \n, it should perform action of printf("Hello World")

- In 2nd esp, we need to take input the name of user and print it, so in declaration we need to declaration string and then in Rule section we used to print ("Hello user %s", c); syntax is same as language.

- 3rd program, we need to output number of uni & char in input. so in rule section, ib we execute it will give the required output.

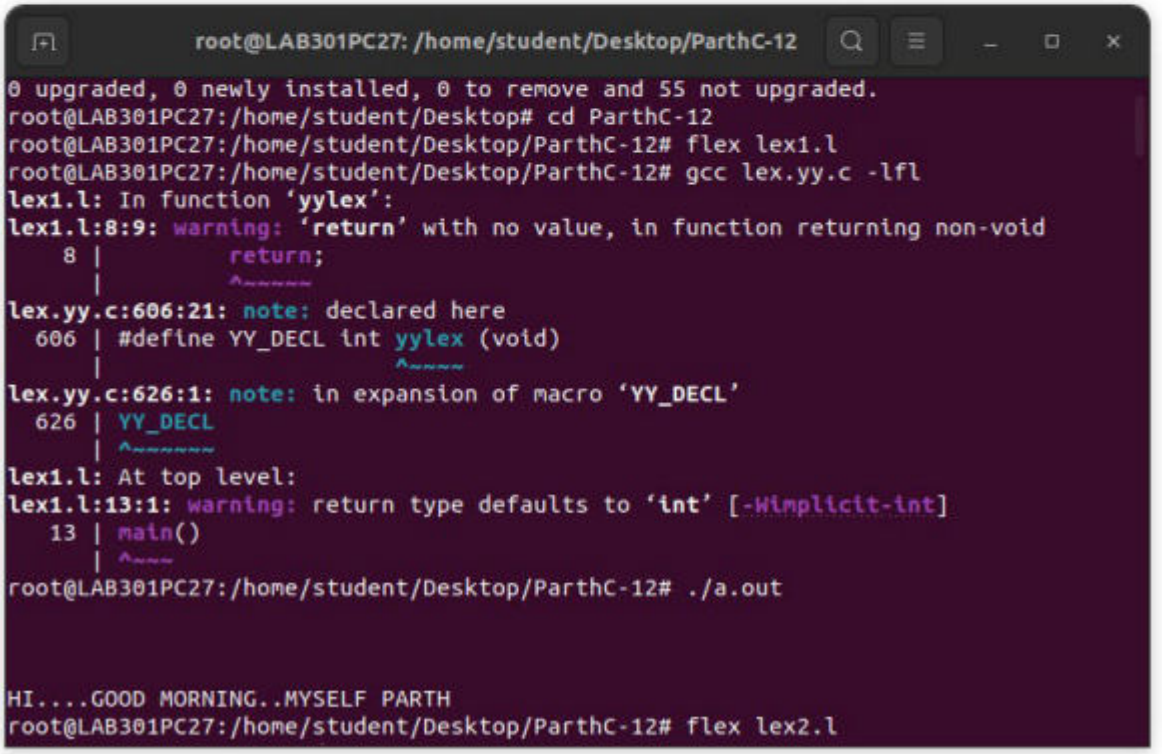
- 4th program, ib we execute a vowel (a, e, i, o, u) then we need to display vowel occured, so we used regex (a, e, i, o, u) & rule y

- In 5th program, we implemented code generation program, so ib we encountered int or keyword, we print keyword & so on. So we print keyword, identifier, constant & seperator.

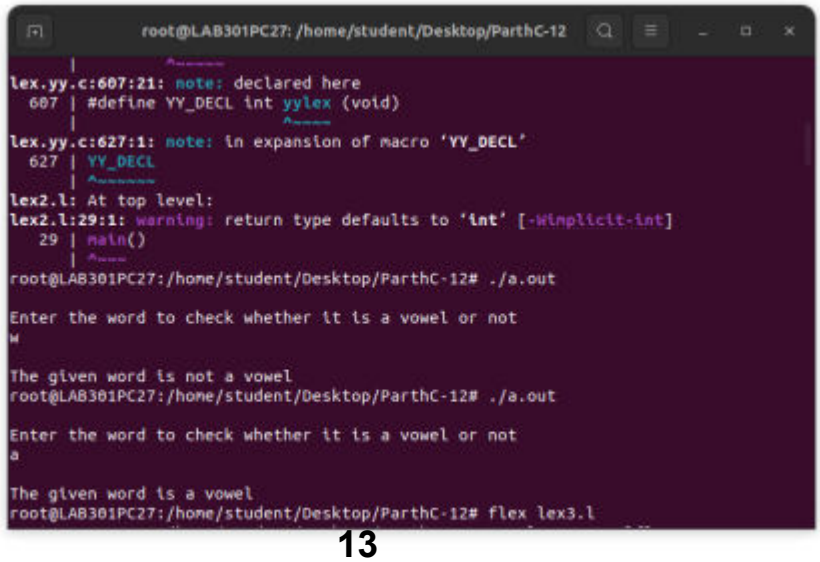
Conclusion :- In this experiment we learned how to install and compile the lex programs and implemented 5 codes.

22/02/2024

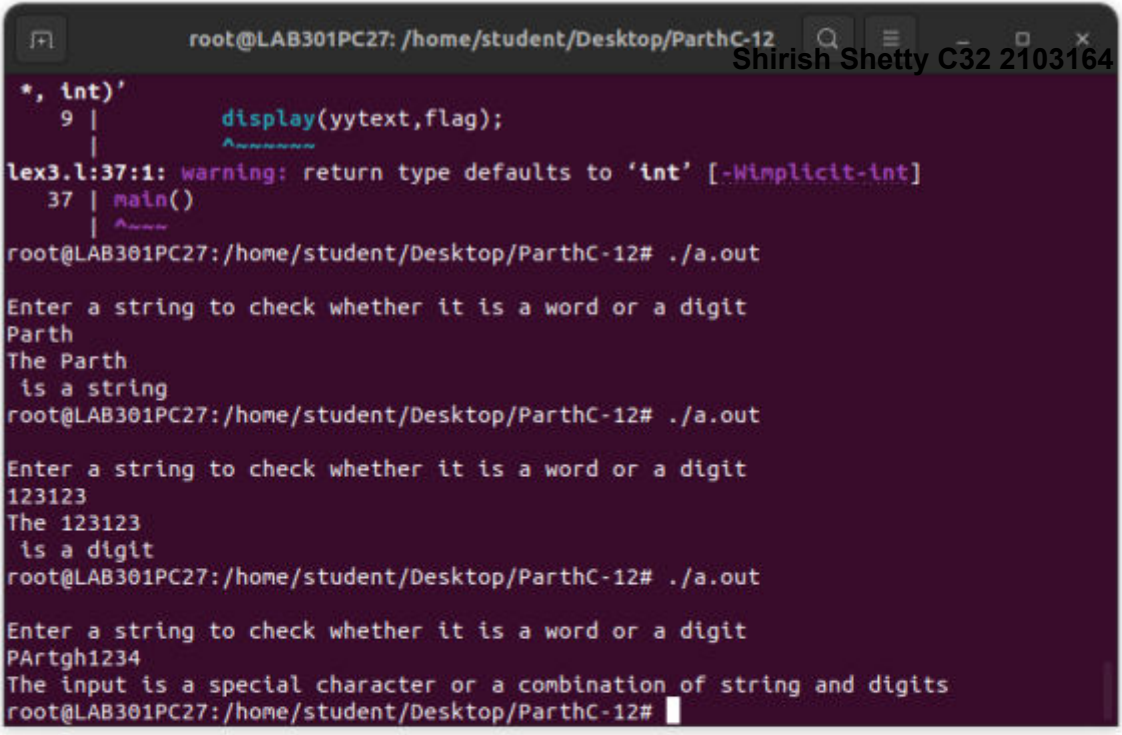
```
%{
%}

%%

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

%%

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



```
%{
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();
}
```



```
%{
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
Shirish Shetty C32 2103164
*, 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#

Experiment No - 3

Aim :- To write a program to implement First and Follow

Theory :-

First and Follow in compiler design are 2 grammatical Functions that help you inter table Entries. If compiler knew area of time what the 'initial character and follow up of string produced when a production rule is applied, "it might carefully choose rule is applied, Which production rule to apply by comp it to current character or token in input string it sees.

First ()

→ First () is a function that specifies the set of terminal that start a string derived from production rule. It's the first terminal that appears on RHS of production.

for eg :-

If input string is

$T \rightarrow * F T' / \epsilon$

Here we find out that T has 2 productions like $T \rightarrow * F T'$ and $T \rightarrow \epsilon$, after viewing this we found the first of T in both the production statement which is * and ϵ

Then first of string will be { *, ∈ }

Rules for finding First() :-

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

- if x is terminal, then first(x) = { x }
- if x is non terminal and x tends to aα is production, then add 'a' to first of x
- if X → ∈, then add null to first(X)
- if X → Y2 then if first(Y) = ∈, the first(X) = { first(Y) − ∈ } ∪ first(Z)
- if X → Y2, then if first(X) = Y, then first(Y) = terminal but null then first(X) = First(Y) = terminal.

* Follow :-

It's a set of terminal of symbol that can be displayed just to the right of non-terminal symbol in any sintence format

Rules to find Follow() :-

- $ 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) then everything in following = ∈ (A) is a follow of B

TSEC
ENGINEERING COLLEGE

Example of First and Follow :-

$E \rightarrow TE'$
$E' \rightarrow +TE' / \in$
$T \rightarrow FT'$
$T' \rightarrow *FT' / \in$
$F \rightarrow (E) / id$

Non terminal	First ( )	Follow( )
E	{ (, id }	{ $,) }
E'	{ +, ∈ }	{ $,) }
T	{ (, id }	{ $, +,) }
T'	{ *, ∈ }	{ +,), $ }
F	{ (, id }	{ +,), $, * }

First (E) = First(T) = First(F) = { (, id }
First (E') = { +, ∈ }
First T ⇒ First(F) = { (, id }
First T' ⇒ { *, ∈ }
First F = { (, id }

Conclusion :- learnt about First & Follow concept

25/02/24

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 :- To implement Parser

Theory :-

A parser is a software component that takes input data and builds a data structure after some kind of parse tree, abstract syntax tree or other hierarchical structure - giving a structural representation of input while checking for correct syntax according to formal grammar. The parsing process is a key phase in compiler, but it is also widely used in various applications like data interchange formats, document processing and NLP

⇒ for simple arithmatic expression parser, we will focus on the following :-

1] Tokenization :- This process of breathing the input string into meangiful units called token, In our cases, token would be numbers addition (+) and subtraction (-)

2] grammar :- It define the syntax rules for our arithmatic expression. A simple grammar for our purpose could be

- An expression can be another expression followed by operator & control number.

iii] Parsing Technique :- We will use a recursive decent parsing technique, which is top down method of syntax analyses. The method involves writing a set of recursive function whereas each function attempt to parse a portion of input according to a rule of grammar.

Conclusion :-

The developed parser successfully evaluates simple arithmatic expression involving addition and substraction, demonstrating the fundamental concepts of tokenization & recursive descent pasing.

28/03/2024

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 :-

Components of TAC :-

- Operators :- Represent basic operation like arthmatic, logical assignment etc
- Operands :- Variable, constants and temprory variable used in operators
- Instructions :- Basic units of TAC containing three operand (result, source 1, source 2)

→ Example TAC

Instructions :-

i) Temp1 = a + b : Addition operation
ii) temp2 = c * d : Multiplication operation
iii) e = temp1 - temp2 : subtraction
iv) if x < y goto L1 : Conditional jump instruction
v) print e : Print statement.

→ Implementation steps :-

(i) Lexical Analysis;

(ii) Syntax analysis;
(iii) Intermediate code generation;
(iv) Code Optimization
(v) Code execution

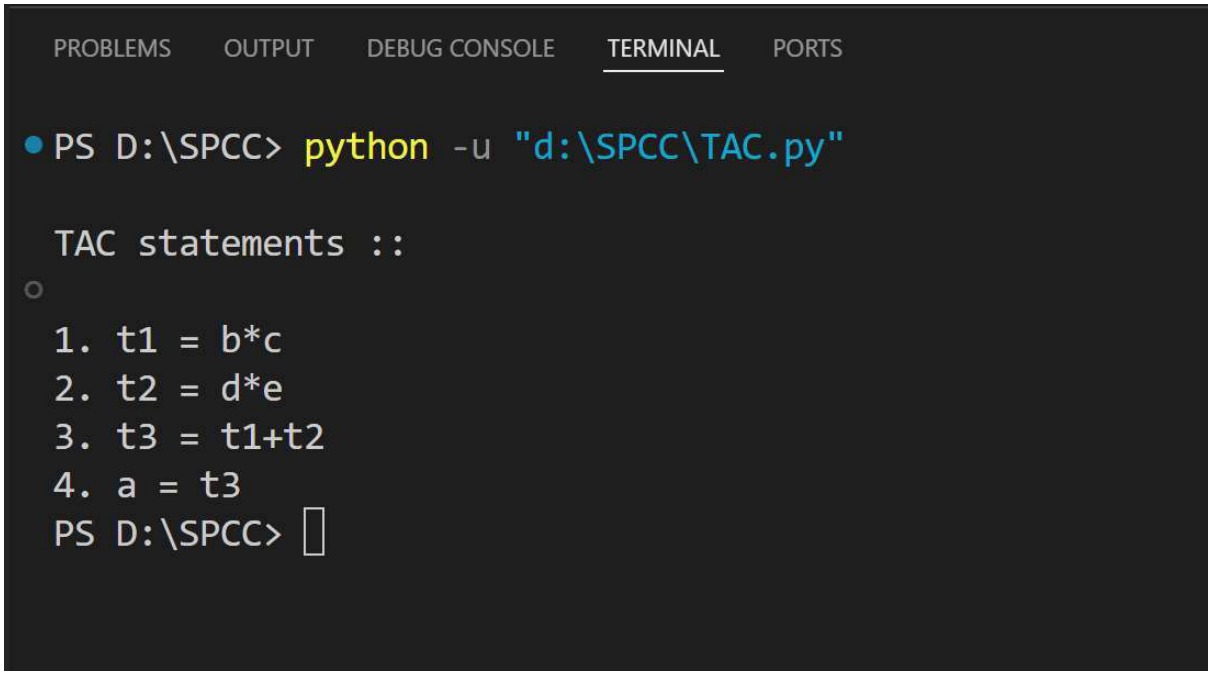
Conclusion :-

The experiment help in gaining a practical understanding of TAC, its generation & execution

28/03/2024.

```
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 :- WAP to implement Code Optimization

Theory :-

Code optimization is a crucial aspect of compiler designer aimed at improving the efficiency, speed and resource utilization of generated code.

⇒ Code Optimization Techniques

1. Constant Folding

```
# define K 5
x = 5 * K      ⟶  x = 10
y = K + 5          y = 10
```

2. Compile Time Evaluation

A = 2 * (22/7) * r

Perform 2 * (22/7) * r at compile time

3. Variable propagation

```
c = a * b                    c = a * b
x = a          OPTIMISED     x = a
till       ⟶                 till
d = x * b + y                d = a * b + y
```

(iii) Constant propogation

If variable is a constant, then replace variable by constant

(iv) Copy propogation

$c = a * b$	$c = a * b$
$x = a$	$x = a$
till $d = x * b + y$	till $d = a * b + y$

When to apply optimization

Source program
Intermediate code
Target code
Local Optimization.

Conclusion :- In this, I learned the code optimization and its technique.

28/03/2024.

```
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()
```

Output Clear

```
Enter the Number of Values: 5
left: a
right: 15
left: b
right: d+e
left: c
right: d+e
left: f
right: c+d
left: r
right: f
Intermediate Code
a=15
b=d+e
c=d+e
f=c+d
r=f

After Dead Code Elimination
c   =d+e
f   =c+d
r   =f
Eliminate Common Expression
c   =d+e
f   =c+d
r   =f
Optimized Code
c=d+e
f=c+d
r=f

=== Code Execution Successful ===
```

Experiment No 7

Aim :- WAP to implement Pass1, Pass2 of Multi-Pass Assembler.

Theory :-

PASS 1

1] Input Processing
- In PASS1, the assembler reads the entire assembly language program line by line

2] Symbol Table Creation
- As assembler encounter labels, it adds them to a symbol table along with that corresponding memory address.

3] Location counter [LC] Management :-
- The assembler maintain a location counter that keeps track of memory address addressed to instructions and data.

4] Error Detection :-
- PASS1 also perform basic error checks such as syntax error, undefined symbol and dulicpate labels.

5] Output :-

→ PASS1 generate a intermediate file containing the symbol table and modified assembly code with issolved addres

PASS2 :-

1] I/p Processing :-

→ PASS2 takes the intermediate file genrated by PASS1 as i/p, it reads the modified assembly code & symbol.

2] Instruction Translation :-

→ For each instruction, PASS 2 translate the mnemonic opcode and operand into corresponding machine code.

3] Object code generation :-

→ As PASS 2 process each instruction, it generate the object code in binary or hexadecimal format

4] Error Handling :-

→ PASS2 perform additional error check such as undefined symbol invalid opcode & operand mismatches

v] Final o/p

- Upon succesful compilation, PASS 2 produces the final machine code or object program ready for execution on target computer.

Conclusion :-

The implementation of PASS I and PASS 2 of a multi-pass assembler involves efficient i/p processing, address resolution, error detection and obj code generation.

28/03/2024

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 - 8

Aim:- Write a program to implement Multi Pass Macroprocessor.

Theory:-

A multi pass macroprocessor in system programming and compiler construction involves a process where source code is parsed multiple times to handle macro expansion effectively. In this context, a multiple pass approach allows 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, multi pass systems revisit the code to ensure accurate handling of macros and other language construct.

The concept of multi pass macroprocessors is significant in system programming as enable a more comprehensive analysis of source code, particularly when dealing with complex macro definations & invocations. By parsing code multiple times, these system can manage macro expansion efficiently, ensuring all necessary replacement & transformation are accurately executed.

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,R1      Load value 2 in R1
MUL R1,R1,R2    Square R1 and store in R2
ADD R2,MUL R1,R3  Add square of R1 to
                  cube of R1 and
                  store in R3
```

Conclusion :- Multi pass macroprocessor in SPCC involves a sophisticated approach to handling macros by source code parsing multiple time.
This method allow for more intricate procesing of macros & other ~~intricate processing of macro~~ language element, contributing to effective design & implement ation of compiler & system programming tools.

28/03/2024.

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
```

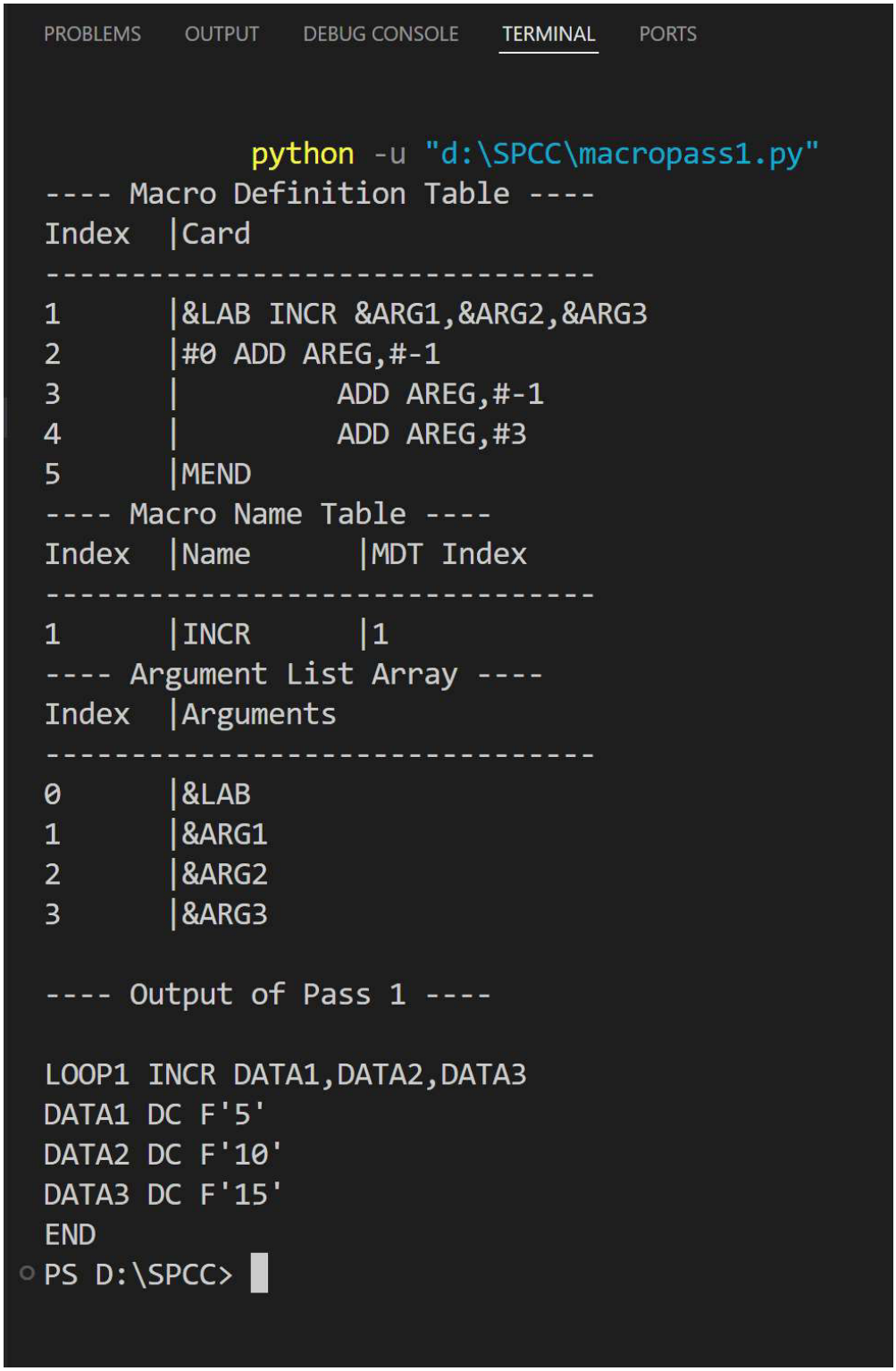
```
mdtc = 1
mntc = 1
for line in inputfile:
    if "MACRO" in line.upper():
        inside_macro = True
        macro_name_line = True
    elif "MEND" in line.upper():
        row = MDT_row(mdtc, line.strip())
        mdtc += 1
        MDT.append(row)
        inside_macro = False
    elif not inside_macro: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

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\SPCC> python -u "d:\SPCC\macropass2.py"
---- Updated Argument List Array ----
Index   |Arguments
0       |LOOP1
1       |LOOP1
2       |LOOP1
3       |LOOP1
---- Expanded Macro Source Deck ----
LOOP1 ADD AREG,LOOP1
      ADD AREG,LOOP1
      ADD AREG,LOOP1
DATA1 DC F'5'
DATA2 DC F'10'
DATA3 DC F'15'
END
PS D:\SPCC>
```

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
```

Experiment No 9

Aim - WAP to implement code generation

Theory :- Code generation is used to produce the target code for the 3 address statement. It uses registration to store the operandents of 3 add structs

Consider 3 Add Statement $x = y + z$. It can have following sequence of codes.

```
MOV x, R0
MOV y, R0
```

- Register description contain the track of what is currently in each register.
- An address description is used to store location where current value of name can be found at runtime

Code generation algo :-

1.) Involve the getting to find at location L where result. If value of y currently in memory and register both then prefer the register. If value y is not already in L and generate the instruction MOV y, L to place copy of y in L.

2.) Generate of z, L where 'z' is used to show current location of z. If z is in both, the register to memory location update the add description of x is in location L. If x is in L then update its desc and reserve x from all other description

3.) If current value of y or z have do no next use or not live on exit from block in register then all register desc to indicate that after execution of x = y then register while longer contain y or z.

Statement :-

$$d = (a-b) + (a-c) + (a-c)$$

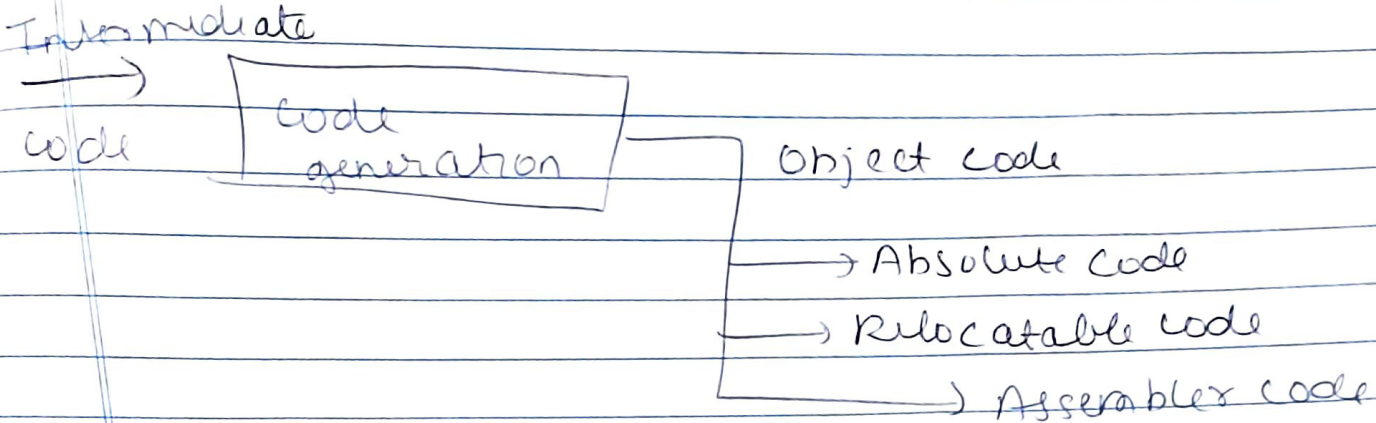
$$t = a - b$$

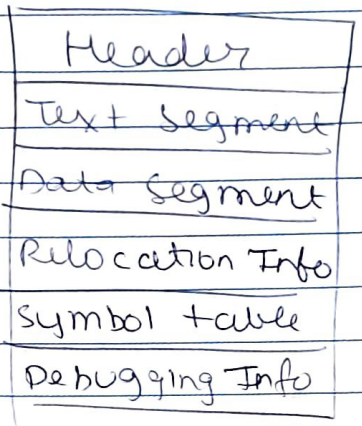
$$u = a - c$$

$$v = t + u$$

$$d = v + u$$

Stmt	Code gn	Register Desc	Address Desc
t = a - b	mov a, R0	R0 contains t	t in R0
	sub b, R0	R0 contain t	t in R0
		R1 contain u	u in R1
u = a - c	mov a, R1	R0 contain t	u in R1
	sub c, R1	R1 contain u	v in R2
v = t + u	Add R1, R0		
d = v + u	Add R1, R0	R0 contains d	d in R0
	mov R0, d		d in R0 and memory





Conclusion :- - learnt about code generation in SPCC.

28/03/2024

Code :

Shirish Shetty C32 2103164

```
op1, op2, op3, op4 = "", "", "", ""


print("Enter operation and operands (e.g., + o1 o2 o3): ")
while True:
    line = input().split()
    if not line:
        break

    # Assuming there's always an operation and at least two operands provided
    op1, op2, op3 = line[0], line[1], line[2]
    if len(line) > 3:  # Check if there's a fourth operand
        op4 = line[3]
    else:
        op4 = "Result"  # Default name for the result if not provided

    if op1 == "+":
        print(f"MOV AX, {op2}")
        print(f"MOV BX, {op3}")
        print("ADD AX, BX")
        print(f"MOV {op4}, AX")
    elif op1 == "-":
        print(f"MOV AX, {op2}")
        print(f"MOV BX, {op3}")
        print("SUB AX, BX")
        print(f"MOV {op4}, AX")
    elif op1 == "*":
        print(f"MOV AX, {op2}")
        print(f"MOV BX, {op3}")
        print("MUL AX, BX")
        print(f"MOV {op4}, AX")
    elif op1 == "/":
        print(f"MOV AX, {op2}")
        print(f"MOV BX, {op3}")
        print("DIV AX, BX")
        print(f"MOV {op4}, AX")
    elif op1 == "=":
        print(f"MOV {op2}, {op3}")
    else:
        print("Invalid operation")

print("\nCode generation successful")
```

Ouput:


```
PS C:\Users\Jagjeet\Desktop\CODEFORCES\spcc> python exp7.py
Enter operation and operands (e.g., + o1 o2 o3):
- a b t
MOV AX, a
MOV BX, b
SUB AX, BX
MOV t, AX
- a c u
MOV AX, a
MOV BX, c
SUB AX, BX
MOV u, AX
+ t u v
MOV AX, t
MOV BX, u
ADD AX, BX
MOV v, AX
+ v u d
MOV AX, v
MOV BX, u
ADD AX, BX
MOV d, AX
* v u k
MOV AX, v
MOV BX, u
MUL AX, BX
MOV k, AX


Code generation successful
```

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Experiment No 10

Aim :- Write program to eliminate left Recursion from given grammar

Theory :-

A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.

A grammar G(V, T, P, S) is left Recursion because the left of production in form.

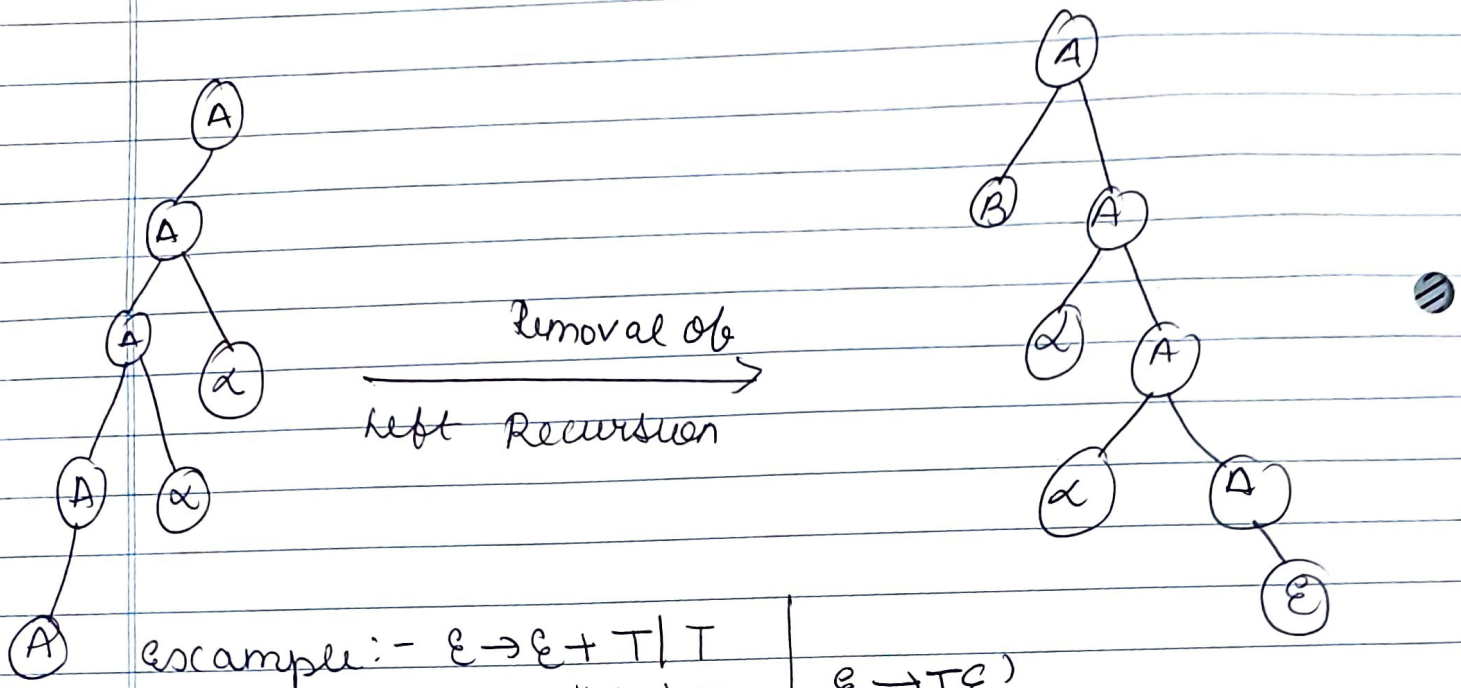
$$A \rightarrow A\alpha \mid B$$

The above grammar is left Recursive because the left of production is only at first position on right side of production.
We can replace left Recursion by replacing by replacing a pair of production with

$$A \rightarrow BA'$$
$$A \rightarrow \alpha A' \mid \varepsilon$$

In left Recursive grammar, expansion of A will generate AX, AXX, AXXX on each side, causing it to enter into an infinite loop

$A \rightarrow A\alpha \mid B$ —Removal of Left Recursion→ $A \rightarrow BA'$, $A' \rightarrow \alpha A' \mid \varepsilon$



Example:- $E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$

Compare $E \rightarrow E+T \mid T$
with $A \rightarrow A\alpha \mid B$
$\therefore A = E, \alpha = +T, B = T$
$A \rightarrow A\alpha \mid B$ then it changes to
$A' \rightarrow \alpha A' \mid \varepsilon$
$\therefore A \rightarrow BA'$ means $E \rightarrow TE'$

Compare $T \rightarrow T*F \mid F$
$A = T, \alpha = +F, B = F$
Production $F \rightarrow (E) \mid id$ does not have any left Recursion

$E \rightarrow TE'$
$E' \rightarrow +TE \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow +FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Conclusion:- In this experiment, we learn about left Recursion and how to remove left Recursion.

28/03/2024.

```
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 ===
```