Thadomal Shahani Engineering College

Bandra (W.), Mumbai- 400 050.

CERTIFICATE

Certify that ~~Mr.~~/Miss Niyati Savant of Computer Department, Semester VI with Roll No. 2103156 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

08/04/24

Teacher In-Charge

Head of the Department

Date 08/04/24

Principal

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

CONTENTS

SR. NO.	EXPERIMENTS	PAGE NO.	DATE	TEACHERS SIGN.
1	A Program to implement Lexical Analyzer	1 - 5	16/01/24	
2	To study & implement programs LEX and YACC tool	6 - 16	23/01/24	
3	A Program to implement the FIRST & FOLLOW set of given program.	17 - 27	30/01/24	
4	A Program to implement parser	28 - 40	06/02/24	
5	A program to implement Three address Code.	41 - 46	13/02/24	
6	A program to implement Code Optimization	47 - 54	20/02/24	
7.	a) Write program to implement Pass - 1 of Multi Pass Assembler b) Write program to implement ~~pass~~ Pass - 2 of Multi Pass Assembler	55 - 71	27/02/24	
8.	Write program to implement Multipass Macroprocessor	72 - 81	05/03/24	
9.	Write a program to implement ~~Multipass Macroprocessor~~ Code Generator	82 - 86	12/03/24	

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

CONTENTS

SR. NO.	EXPERIMENTS	PAGE NO.	DATE	TEACHERS SIGN.
10	Write program to eliminate left recursion from given grammar	87-92	19/03/24	
11	Written Assignment -1	93-96	26/03/24	
12	Written Assignment - 2	97-99	30/03/24	
13	Written Assignment -3	100-103	08/04/24	

Niyati Savant
TE - C31
2103156

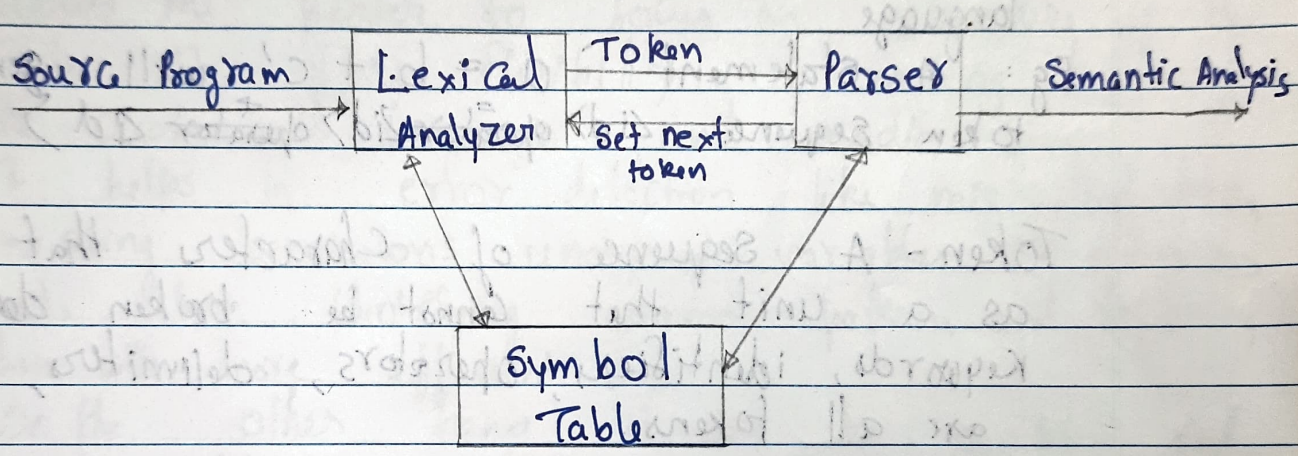
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Experiment - 01

Aim - To implement Lexical Analyzer using Python

Theory -

Lexical Analysis is the first phase of the compiler also called scanner. It converts High level input program into a sequence of Tokens. It can be implemented with the DFA. Its output is a sequence of tokens that is sent to parser for syntax analysis



Working of Lexical Analyzer -
It includes stages -

a) Input preprocessing - Involves cleaning up input text by removing comments, whitespace, and other non-essential characters

b) Tokenization - Breaking input text into a sequence of tokens

Niyati Savant
TE C31
2103156

c) Token classification - Identify the type of each token like keywords, identifiers, operators, etc.

d) Token validation - validates if each token is valid as per the rules of the language.

e) Output generation - Generates output i.e list of tokens to be passed to next stage

Lexical analyzer identifies the error with the help of automation machine and grammar of given language

Eg: A statement 'a = b + c' will generate token sequence <id> ~~operator~~ = <id> ~~operator~~ + <id>

Token - A sequence of characters that are treated as a unit that cannot be broken down further. Keywords, identifiers, operators, delimiters, punctuators are all tokens.

Lexeme - A sequence of characters in source code that are matched by given predefined language rules for every lexeme to be specified as a valid token.

Pattern - Specifies a set of rules that scanner follows to create a token.

Niyati Savant
TE C31
2103156

The output of lexical Analyzer serves as input to Syntax Analyzer as a sequence of tokens and not lexemes as during syntax analysis phase, category of lexeme is considerable and not the individual unit

Conclusion

Lexical analysis improves the efficiency of the parsing process because it breaks down the input into smaller, more manageable chunks. This allows the parser to focus on structure of the code. It offers flexibility by making it easy to create new languages and modify them. It helps in error detection like misspelled words, missing semicolons, undefined variables. It optimizes code by identifying common patterns and replacing them with more efficient code.
On the other hand, it can be complex and need a lot of computing power. It cannot detect logic or type errors. The addition of keywords and reserved words increase code size and also reduce flexibility by limiting words that a programmer may use.

23/01/2024 A+

EXPERIMENT -1

LEXICAL ANALYZER

```
prog_file = open("prog.txt", mode='r')
file_content = prog_file.read()

print("NIYATI's CODE FOR LEXICAL ANALYZER")
print("The program is -- ")
print(file_content)

keyword_list = ['for', 'in', 'range', 'if', 'else', 'int', 'float',
'print', 'open']
identifier_list = re.compile(r'^[a-zA-Z]\w*$')
operator_list = ['+', '-', '*', '/', '=', '==', '!=', '%', '^', '!']
constant_list = re.compile(r"^\d*[.,]?\d*$")
separator_list = [';', '\n', '\t']

op_key = []
op_iden = []
op_opera = []
op_const = []
op_sep = []

for x in file_content.split():
    if x.lower() in map(str.lower, keyword_list) and x.lower() not in
op_key:
        op_key.append(x)
    elif x in operator_list and x not in op_opera:
        op_opera.append(x)
    elif x in separator_list and x not in op_sep:
        op_sep.append(x)
    elif re.match(identifier_list, x) and x not in op_iden:
        op_iden.append(x)
    elif re.match(constant_list, x) and x not in op_const:
        op_const.append(x)

print(f"The keywords are {op_key}")
print(f"The operators are {op_opera}")
print(f"The identifiers are {op_iden}")
print(f"The constants are {op_const}")
print(f"The separators are {op_sep}")
```

OUTPUT -

NIYATI's CODE FOR LEXICAL ANALYZER

The program is --

```
for i in range ( 10 ):
   a_num12 = i + 5 ;
   print ( a_num12 ) ;
```

The keywords are ['for', 'in', 'range', 'print']

The operators are ['=', '+']

The identifiers are ['i', 'a_num12']

The constants are ['10', '5']

The separators are [';']

NIYATI's CODE FOR LEXICAL ANALYZER

The program is --

```
for i in range ( 10 ):
   if i % 2 == 0 :
      print("Even")
   else :
      print("Odd")
```

The keywords are ['for', 'in', 'range', 'if', 'else']

The operators are ['%', '==']

The identifiers are ['i']

The constants are ['10', '2', '0']

The separators are []

Niyati Savart
2103156

C31 Experiment -2

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

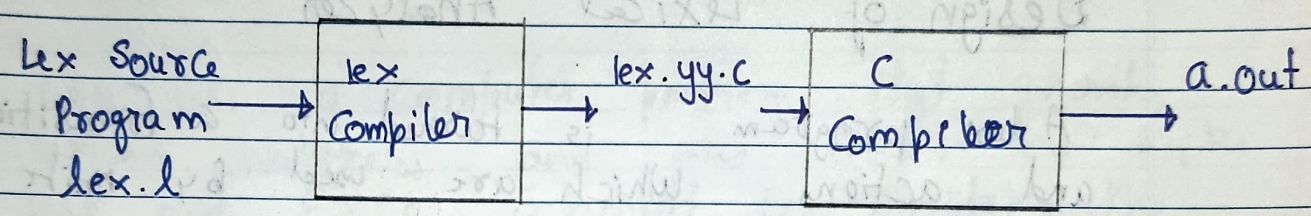
23/1/24

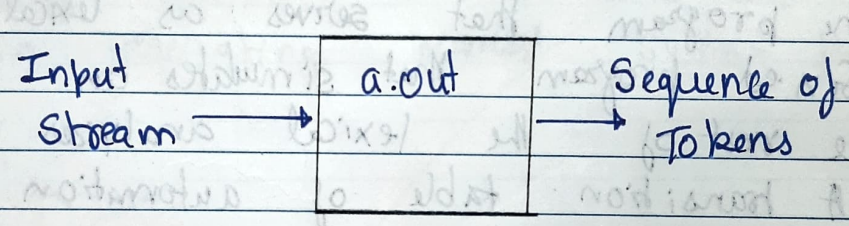
Aim - To study and implement programs LEX and YACC Tool

Theory-

Lexical analysis is aka linear analysis or scanning processes character from left to right and group character stream into significant unit called lexemes which are mapped to tokens.

Lex Compiler





Working of lexical analyzer with Parser-

- When called by parser, lexical analyzer begins reading its remaining input, one character at a time.
- It reads until it finds the longest prefix of input that matches one of the patterns
- It then executes action i which returns

Niyati Savant
2103156
C31

the control to parser.
- But, if it does not then lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser.
- The lexical analyzer returns the token to the parser
- To pass an attribute value with information about the lexeme, a global variable is set known as yylval.

Design of Lexical Analyzer

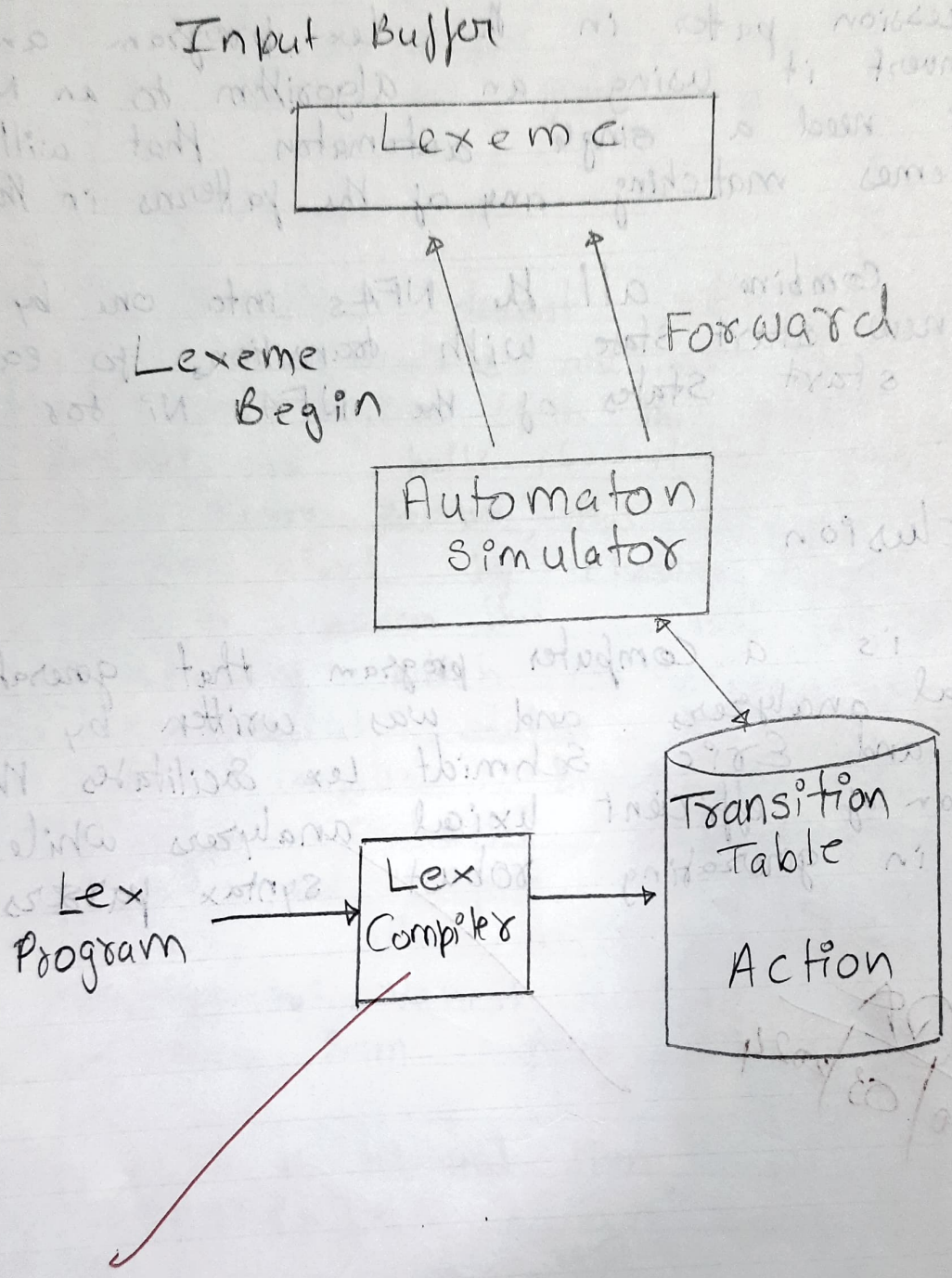
- A lex program is turned into a transition table and actions which are used by finite automaton simulator
- The program that serves as lexical analyzer includes a fixed program that simulates an automaton.
- The rest of the lexical analyzer consists of:

a) A transition table of automation

b) Those functions that are passed directly through lex to the output

c) The action from input program which is to be invoked at appropriate time

- To construct the automaton, take each regular


Input Buffer
Lexeme
Lexeme Begin
Forward
Automaton simulator
Transition Table
Action
Lex Program
Lex Compiler

Nivati Savant
2103156
C31

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

expression patter in the lex program and convert it using an algorithm to an NFA
- We need a single automaton that will recognize lexemes matching any of the patterns in the program

Hence combine all the NFAs into one by introducing a new start state with transitions to each of the start states of the NFAs Ni for pattern pi.

Conclusion

Lex is a computer program that generates lexical analyzers and was written by Mike Lesk and Eric Schmidt. Lex facilitates the creation of efficient lexical analyzers while Yacc aids in generating robust syntax parsers.

DA
20/03/2024

EXPERIMENT – 2

Aim - To study and implement programs LEX and YACC tool

2.a] Hello World

```
%{

%}

%%
\n { printf("\n Hello World");
%%

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

```
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ lex niyati_hello.l
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ gcc lex.yy.c -lfl
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ ./a.out

Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

2.b | Hello User

```
%{
char name[10];
%}

%%
[\n] {printf("\n Hi %s Good Morning\n",name); return 1;}
%%

void main()
{
char opt;
do {
printf("\nWhat is your name?");
scanf("%s",name);
yylex();
printf("\nPress y to continue"); scanf("%c",&opt);
}
while(opt=='y');
}
```

```
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ lex
niyati_hello_user.l
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ gcc lex.yy.c -lfl
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ ./a.out
What is your name ?
Niyati
Hi Niyati Good Morning
Press y to continue
y
What is your name ?
Siya
```

Hi Siya Good Morning

Press y to continue

n

2.c] Count number of characters and lines

```
%{
int linecount = 0, charcount = 0;
%}

%%
. { charcount++; }
\n { linecount++; charcount++; }
%%

void main()
{
yylex();
printf("# of lines = %d, # of chars = %d", linecount, charcount);
}
```

```
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ lex niyati_count.l
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ gcc lex.yy.c -lfl
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ ./a.out
Hello Niyati is typing this.
From lab 401
On a bright Tuesday
# of lines = 3, # of chars = 61
```

2.d] Input is a vowel or not

```
%{
char s;
%}

%%
[aeiouAEIOU] {printf("A vowel");}
. {printf("Not a vowel");}
return 1;
%%

void main()
{
printf("Enter a character: ");
scanf("%c",&s);
yylex();
}
```

```
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ lex niyati_vowel.l
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ gcc lex.yy.c -lfl
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ ./a.out
Enter a character:
F
Not a vowel

admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ ./a.out
Enter a character:
u
A vowel
```

2.e] Identify keywords, identifier, operator, separator and consonants

```
%{
#include <stdio.h>
#include <string.h>
char* str;
%}

%%
range |
break |
else  |
in    |
for   |
if    |
do    |
switch   { printf(" %s is a keyword\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]*          { printf(" %s is an identifier\n", yytext); }
[-+*/=]  { printf(" %s is an operator\n", yytext); }
[0-9]+   { printf(" %s is a constant\n", yytext); }
.   { /* Ignore other characters */ }
%%

int main(){
   printf("Enter a sentence: ");
   yylex();
   return 0;
}
```

```
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ lex niyati_key.l
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ gcc lex.yy.c -lfl
admini@admini-Victus-by-HP-15L-Gaming-Desktop-TG02-1xxx:~/NIYATI C31$ ./a.out
Enter a sentence: int a + b = 93
```

int is an identifier

a is an identifier

+ is an operator

b is an identifier

= is an operator

93 is a constant

Niyati Savant
TE C31
2103156

Experiment 3

20/02/24

Aim - To implement the first and follow set for given grammar

Theory -

A recursive descent parsing program consists of a set of procedures one for each nonterminal. Execution begins with the procedure for start symbol and halts if it has scanned entire input string. However, this is non-deterministic as we start by choosing a production to apply in a manner that is not specified thus require backtracking which is complex.
The easiest solution to this problem is if the compiler knows in advance what is the first character produced when a rule is applied and compare it with current character to take wise decisions. Follow makes a non-terminal vanish out if needed to generate the string from parse tree ~~and~~

1) If X is a terminal then
FIRST(X) = X

2) If X is non terminal &

a) $X \rightarrow Y_1 Y_2 .. Y_k$ is a production for $k >= 1$ then -

Niyati Savant
TE - C31
2103156

Eg - 1

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

First (E) = First (T)
= First (F)
= { (, id }

First (E') = { + , ε }

First (T) = First (F)
= { (, id }

First (T') = { *, ε }

First (F) = { i, id }

Niyati Savant

Niyati Savant
TE C31
2103156

Follow (E) = { $,) } .... E is start symol & rule F → (E)

Follow (E') = Follow (E)
= { $,) } ... rule 3

Follow (T) = { First (E') − ∈ } ∪ { Follow (E') }
= { + } ∪ { $,) }
= { + , $,) }

Follow (T') = Follow (T)
= { + , $,) }

Follow (F) = { First (T') − ∈ } ∪ { Follow (T') }
= { * } ∪ { + , $,) }
= { * , + , $,) }

Non Terminal	First	Follow
E	(, id	$,)
E'	+, ∈	$,)
T	(, id	+, $,)
T'	*, ∈	+, $,)
F	(, id	*, +, $,)

Niyati Savant
Niyati Savant
Niyati Savant
TE - C31
2103156

Case 1 :

i) if for some i, 'a' is in First(Y_i) and
ii) ϵ is in all of first(Y_j) for $j = 1, 2, \ldots j-1$
add 'a' in first(x)

Case 2 :
i) if ϵ in First(Y_j) for all $j = 1, 2, \ldots, k$
add ϵ in first(x)

3) If $x \to \epsilon$ is production add ϵ in first(x)

Follow of grammar

1) Place \$ in Follow(S) if S is start symbol and \$ is input right endmarker

2) If there is a production $A \to \alpha B \beta$ then everything in First(β) except ϵ is in follow(B)

3) For $A \to \alpha B$ or $A \to \alpha B \beta$ where first(β) contains ϵ then everything in FOLLOW(A) is in FOLLOW(B)

Rules are applied till there is no updation in follow list

Conclusion - learnt the rules of first and follow and its use cases

20/09/2024

EXPERIMENT 3

AIM: Implement the FIRST and FOLLOW set for the given grammar

Code:

```
import sys
sys.setrecursionlimit(60)


def first(string):


    first_ = set()
    if string in non_terminals:
        alternatives = productions_dict[string]


        for alternative in alternatives:
            first_2 = first(alternative)
            first_ = first_ |first_2


    elif string in terminals:
        first_ = {string}


    elif string==" or string=='@':
        first_ = {'@'}


    else:
        first_2 = first(string[0])
        if '@' in first_2:
            i = 1
            while '@' in first_2:



                first_ = first_ | (first_2 - {'@'})
                if string[i:] in terminals:
                    first_ = first_ | {string[i:]}
```

```
                break
            elif string[i:] == '':
                first_ = first_ | {'@'}
                break
            first_2 = first(string[i:])
            first_ = first_ | first_2 - {'@'}
            i += 1
        else:
            first_ = first_ | first_2



    return  first_



def follow(nT):


    follow_ = set()


    prods = productions_dict.items()
    if nT==starting_symbol:
        follow_ = follow_ | {'$'}
    for nt,rhs in prods:


        for alt in rhs:
            for char in alt:
                if char==nT:
                    following_str = alt[alt.index(char) + 1:]
                    if following_str==':
                        if nt==nT:
                            continue
                        else:
                            follow_ = follow_ | follow(nt)
```

```
            else:
                follow_2 = first(following_str)
                if '@' in follow_2:
                    follow_ = follow_ | follow_2-{'@'}
                    follow_ = follow_ | follow(nt)
                else:
                    follow_ = follow_ | follow_2
    return follow_

no_of_terminals=int(input("Enter no. of terminals: "))
terminals = []
print("Enter the terminals :")
for _ in range(no_of_terminals):
    terminals.append(input())

no_of_non_terminals=int(input("Enter no. of non terminals: "))
non_terminals = []
print("Enter the non terminals :")
for _ in range(no_of_non_terminals):
    non_terminals.append(input())

starting_symbol = input("Enter the starting symbol: ")
no_of_productions = int(input("Enter no of productions: "))
productions = []
print("Enter the productions:")
for _ in range(no_of_productions):
    productions.append(input())

productions_dict = {}
for nT in non_terminals:
    productions_dict[nT] = []
```

```
for production in productions:
    nonterm_to_prod = production.split("->")
    alternatives = nonterm_to_prod[1].split("/")
    for alternative in alternatives:
        productions_dict[nonterm_to_prod[0]].append(alternative)

FIRST = {}
FOLLOW = {}

for non_terminal in non_terminals:
    FIRST[non_terminal] = set()

for non_terminal in non_terminals:
    FOLLOW[non_terminal] = set()

for non_terminal in non_terminals:
    FIRST[non_terminal] = FIRST[non_terminal] | first(non_terminal)

FOLLOW[starting_symbol] = FOLLOW[starting_symbol] | {'$'}
for non_terminal in non_terminals:
    FOLLOW[non_terminal] = FOLLOW[non_terminal] | follow(non_terminal)

print("{: ^20}{: ^20}{: ^20}".format('Non Terminals','First','Follow'))
for non_terminal in non_terminals:
    print("{: ^20}{: ^20}{:
^20}".format(non_terminal,str(FIRST[non_terminal]),str(FOLLOW[non_terminal])))
```

Output:

```
/usr/bin/python3.10 /home/admini/Desktop/NIYATI/First_Follow.py
Enter no. of terminals: 5
Enter the terminals :
+
*
(
)
a
Enter no. of non terminals: 5
Enter the non terminals :
E
B
T
Y
F
Enter the starting symbol: E
Enter no of productions: 5
Enter the productions:
E->TB
B->+TB/@
T->FY
Y->*FY/@
F->a/(E)
  Non Terminals        First            Follow
      E            {'a', '('}        {')', '$'}
      B            {'@', '+'}        {')', '$'}
      T            {'a', '('}      {')', '+', '$'}
      Y            {'@', '*'}      {')', '+', '$'}
      F            {'a', '('}    {')', '*', '+', '$'}

Enter no. of terminals: 4
```

NIYATI SAVANT TE C31 2103156

Enter the terminals :

a

b

c

d

Enter no. of non terminals: 3

Enter the non terminals :

S

B

C

Enter the starting symbol: S

Enter no of productions: 3

Enter the productions:

S->Bb/Cd

B->aB/@

C->cC/@

Non Terminals	First	Follow
S	{'d', 'b', 'c', 'a'}	{'$'}
B	{'@', 'a'}	{'b'}
C	{'@', 'c'}	{'d'}

Niyati Savant
TE - C31
2103156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Experiment 4

Aim - Write a program to implement Parser

Theory -

The LL(1) parser or the non-recursive descent is a top down parser without backtracking. The first L represents that input is scanned from left to right. The second L shows that left most derivation is used. The 1 represents the number of look-ahead ie how many symbols can you see when you want to make a decision

Essential Conditions -

1. Grammar is free from left recursion
2. Grammar is unambiguous
3. Grammar is deterministic ie left factored.

These conditions are necessary but not sufficient to prove LL(1) parser.

Algorithm to construct parsing table

1. Check all essential conditions mentioned
2. Calculate First() and Follow() for all non-terminals

TE - C31

2105156

3. For each production $A \to a$
find First (α), for each terminal in First (α), make entry $A \to \alpha$ in the table
If First (α) has ϵ as terminal, then for each terminal in Follow(A), make $A \to \epsilon$ entry
If first (α) contains ϵ and follow has \$, make $A \to \epsilon$ entry for \$

The table has Non-terminals as rows and terminals in columns.
If there are two productions in same cell, grammar is not feasible for LL (1)

Approach for String validation
Stack and buffer is used to validate input string. Initially '\$' is added to both stack and buffer. The input string is entered in the reverse order into buffer. The 'start symbol' is added to stack top.
Traverse iteratively over current state of stack and buffer and match entries of Table [x] [y] where x - Top of stack and y is rear of buffer.
- If $x = y$, we pop from both and continue computation.
- If both remain only with '\$' - string is valid and belongs to grammar
- If rules is not present, or $x \neq y$ string does not belong to the grammar

Niyati Savant
TE - C31
2103156

→ Eg - For the rules

$E \rightarrow TE'$ $\quad$ $E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow FT'$ $\quad$ $T' \rightarrow *FT' \mid \epsilon$

$F \rightarrow (E) \mid id$

The first and follow are

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

Parsing table

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→∈	E'→∈
T	T→FT'			T→FT'		
T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F	F→id			F→(E)		

Niyati Savant
TE - C31
2103156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Validate string - (id * id)

Matched	Stack	Buffer	Action
	E$	(id * id)$	E → TE'
	TE'$	(id * id)$	T → FT'
	FT'E'$	(id * id)$	F → (E)
~~matched~~ (	(E) T'E'$	(id * id)$	match (
	E) T'E'$	id * id)$	E → TE'
	TE') E'$	id * id)$	T → FT'
	FT'E') T'E'$	id * id)$	F → id
id	id T'E') T'E'$	id * id)$	match id
	T'E') T'E'$	* id)$	T' → * FT'
*	* FT'E') T'E'$	* id)$	match *
	FT'E') T'E'$	id)$	F → id
id	id T'E') T'E'$	id)$	match id
	T'E') T'E'$	)$	T' → ϵ
	E') T'E'$	)$	E' → ϵ
)	) T'E'$	)$	match)
	T'E'$	$	T' → ϵ
	E'$	$	E' → ϵ
	$	$	Valid

Thus, we can conclude that the entered string is valid.

Conclusion - LL(1) parsers are predictive ~~top~~ parsers capable of parsing a broad class of CFG efficiently with one token look-ahead

20/03/2024

EXPERIMENT 4

CODE:

```
def first(rule):
    global rules, non_Terminals, \
        terminals, diction, firsts
    if len(rule) != 0 and (rule is not None):
        if rule[0] in terminals:
            return rule[0]
        elif rule[0] == '#':
            return '#'

    if len(rule) != 0:
        if rule[0] in list(diction.keys()):
            fres = []
            rhs_rules = diction[rule[0]]
            for itr in rhs_rules:
                indivRes = first(itr)
                if type(indivRes) is list:
                    for i in indivRes:
                        fres.append(i)
                else:
                    fres.append(indivRes)

            if '#' not in fres:
                return fres
            else:

                newList = []
                fres.remove('#')
                if len(rule) > 1:
                    ansNew = first(rule[1:])
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = fres + ansNew
                        else:
                            newList = fres + [ansNew]
                    else:
                        newList = fres
                    return newList
                fres.append('#')
                return fres


def follow(nt):
    global start_symbol, rules, non_Terminals, \
        terminals, diction, firsts, follows

    solset = set()
    if nt == start_symbol:
        solset.add('$')


    for curNT in diction:
        rhs = diction[curNT]
        for subrule in rhs:
            if nt in subrule:
                while nt in subrule:
                    index_nt = subrule.index(nt)
```

```
                subrule = subrule[index_nt + 1:]
                if len(subrule) != 0:
                        res = first(subrule)


                if '#' in res:
                    newList = []
                    res.remove('#')
                    ansNew = follow(curNT)
                    if ansNew != None:
                        if type(ansNew) is list:
                            newList = res + ansNew
                        else:
                            newList = res + [ansNew]
                    else:
                        newList = res
                    res = newList
                else:
                    if nt != curNT:
                        res = follow(curNT)

                if res is not None:
                    if type(res) is list:
                        for g in res:
                            solset.add(g)
                    else:
                        solset.add(res)
    return list(solset)


def computeAllFirsts():
    global rules, non_Terminals, \
        terminals, diction, firsts
    for rule in rules:
        k = rule.split("->")
        k[0] = k[0].strip()
        k[1] = k[1].strip()
        rhs = k[1]
        multirhs = rhs.split('|')
        for i in range(len(multirhs)):
            multirhs[i] = multirhs[i].strip()
            multirhs[i] = multirhs[i].split()
        diction[k[0]] = multirhs


    for y in list(diction.keys()):
        t = set()
        for sub in diction.get(y):
            res = first(sub)
            if res != None:
                if type(res) is list:
                    for u in res:
                        t.add(u)
                else:
                    t.add(res)


        firsts[y] = t
```

```
def computeAllFollows():
    global start_symbol, rules, non_Terminals, \
        terminals, diction, firsts, follows
    for NT in diction:
        solset = set()
        sol = follow(NT)
        if sol is not None:
            for g in sol:
                solset.add(g)
        follows[NT] = solset


# create parse table
def createParseTable():
    import copy
    global diction, firsts, follows, terminals
    print("\nFirsts and Follow Result table\n")

    mx_len_first = 0
    mx_len_fol = 0
    for u in diction:
        k1 = len(str(firsts[u]))
        k2 = len(str(follows[u]))
        if k1 > mx_len_first:
            mx_len_first = k1
        if k2 > mx_len_fol:
            mx_len_fol = k2

    print(f"{{:<{10}}} "
          f"{{:<{mx_len_first + 5}}} "
          f"{{:<{mx_len_fol + 5}}}"
          .format("Non-T", "FIRST", "FOLLOW"))
    for u in diction:
        print(f"{{:<{10}}} "
              f"{{:<{mx_len_first + 5}}} "
              f"{{:<{mx_len_fol + 5}}}"
              .format(u, str(firsts[u]), str(follows[u])))


    ntlist = list(diction.keys())
    terminals = copy.deepcopy(terminals)
    terminals.append('$')


    mat = []
    for x in diction:
        row = []
        for y in terminals:
            row.append('')
        # of $ append one more col
        mat.append(row)


    grammar_is_LL = True
```

```
    for lhs in diction:
        rhs = diction[lhs]
        for y in rhs:
            res = first(y)

            if '#' in res:
                if type(res) == str:
                    firstFollow = []
                    fol_op = follows[lhs]
                    if fol_op is str:
                        firstFollow.append(fol_op)
                    else:
                        for u in fol_op:
                            firstFollow.append(u)
                    res = firstFollow
                else:
                    res.remove('#')
                    res = list(res) + \
                          list(follows[lhs])

            ttemp = []
            if type(res) is str:
                ttemp.append(res)
                res = copy.deepcopy(ttemp)
            for c in res:
                xnt = ntlist.index(lhs)
                yt = terminals.index(c)
                if mat[xnt][yt] == '':
                    mat[xnt][yt] = mat[xnt][yt] \
                                   + f"{lhs}->{' '.join(y)}"
                else:

                    if f"{lhs}->{y}" in mat[xnt][yt]:
                        continue
                    else:
                        grammar_is_LL = False
                        mat[xnt][yt] = mat[xnt][yt] \
                                       + f",{lhs}->{' '.join(y)}"


    print("\nGenerated parsing table:\n")
    frmt = "{:>12}" * len(terminals)
    print(frmt.format(*terminals))

    j = 0
    for y in mat:
        frmt1 = "{:>12}" * len(y)
        print(f"{ntlist[j]} {frmt1.format(*y)}")
        j += 1

    return (mat, grammar_is_LL, terminals)


def validateStringUsingStackBuffer(parsing_table, grammarll1,
                                   table_term_list, input_string,
                                   term_userdef, start_symbol):
```

```
print(f"\nValidate String => {input_string}\n")


if grammarll1 == False:
    return f"\nInput String = " \
           f"\"{input_string}\"\n" \
           f"Grammar is not LL(1)"


stack = [start_symbol, '$']
buffer = []

input_string = input_string.split()
input_string.reverse()
buffer = ['$'] + input_string

print("{:>20} {:>20} {:>20}".
      format("Buffer", "Stack", "Action"))

while True:
    if stack == ['$'] and buffer == ['$']:
        print("{:>20} {:>20} {:>20}"
              .format(' '.join(buffer),
                      ' '.join(stack),
                      "Valid"))
        return "\nValid String!"
    elif stack[0] not in term_userdef:
        x = list(diction.keys()).index(stack[0])
        y = table_term_list.index(buffer[-1])
        if parsing_table[x][y] != '':

            entry = parsing_table[x][y]
            print("{:>20} {:>20} {:>25}".
                  format(' '.join(buffer),
                         ' '.join(stack),
                         f"T[{stack[0]}][{buffer[-1]}] = {entry}"))
            lhs_rhs = entry.split("->")
            lhs_rhs[1] = lhs_rhs[1].replace('#', '').strip()
            entryrhs = lhs_rhs[1].split()
            stack = entryrhs + stack[1:]
        else:
            return f"\nInvalid String! No rule at " \
                   f"Table[{stack[0]}][{buffer[-1]}]."
    else:

        if stack[0] == buffer[-1]:
            print("{:>20} {:>20} {:>20}"
                  .format(' '.join(buffer),
                          ' '.join(stack),
                          f"Matched:{stack[0]}"))
            buffer = buffer[:-1]
            stack = stack[1:]
        else:
            return "\nInvalid String! " \
                   "Unmatched terminal symbols"
```

```
rules = ["E -> T E'",
        "E' -> + T E' | #",
        "T -> F T'",
        "T' -> * F T' | #",
        "F -> ( E ) | id"
        ]
non_Terminals = ['E', 'E\'', 'F', 'T', 'T\'']
terminals = ['id', '+', '*', '(', ')']
print("Niyati's code for LL(1) parser")
print(rules)
sample_input_string = input("Enter a string to validate: ")

diction = {}
firsts = {}
follows = {}

computeAllFirsts()

start_symbol = list(diction.keys())[0]

computeAllFollows()

(parsing_table, result, tabTerm) = createParseTable()

validity = validateStringUsingStackBuffer(parsing_table, result,
                                          tabTerm, sample_input_string,
                                          terminals, start_symbol)
print(validity)
```

OUTPUT:

Niyati's code for LL(1) parser

["E -> T E' "," E' -> + T E' | #", "T -> F T' ", " T' -> * F T' | #", 'F -> (E) | id']

Enter a string to validate: (id) * id + id

Firsts and Follow Result table

Non-T	FIRST	FOLLOW
E	{'id', '('}	{')', '$'}
E'	{'+', '#'}	{')', '$'}
T	{'id', '('}	{'+', ')', '$'}
T'	{'*', '#'}	{'+', ')', '$'}
F	{'id', '('}	{'+', '*', ')', '$'}

Niyati_Savant_TE_C31_2103156

Generated parsing table:

	id	+	*	(	)	$
E	E->T E'			E->T E'		
E'		E'->+ T E'			E'->#	E'->#
T	T->F T'			T->F T'		
T'		T'->#	T'->* F T'		T'->#	T'->#
F	F->id			F->(E)		

Validate String => (id) * id + id

Buffer	Stack	Action
$ id + id *) id (	E $	T[E][(] = E->T E'
$ id + id *) id (	T E' $	T[T][(] = T->F T'
$ id + id *) id (	F T' E' $	T[F][(] = F->(E)
$ id + id *) id (	(E) T' E' $	Matched:(
$ id + id *) id	E) T' E' $	T[E][id] = E->T E'
$ id + id *) id	T E') T' E' $	T[T][id] = T->F T'
$ id + id *) id	F T' E') T' E' $	T[F][id] = F->id
$ id + id *) id	id T' E') T' E' $	Matched:id
$ id + id *)	T' E') T' E' $	T[T'][)] = T'->#
$ id + id *)	E') T' E' $	T[E'][)] = E'->#
$ id + id *)	) T' E' $	Matched:)
$ id + id *	T' E' $	T[T'][*] = T'->* F T'
$ id + id *	* F T' E' $	Matched:*
$ id + id	F T' E' $	T[F][id] = F->id
$ id + id	id T' E' $	Matched:id
$ id +	T' E' $	T[T'][+] = T'->#
$ id +	E' $	T[E'][+] = E'->+ T E'
$ id +	+ T E' $	Matched:+
$ id	T E' $	T[T][id] = T->F T'

Niyati_Savant_TE_C31_2103156

$ id	F T' E' $	T[F][id] = F->id
$ id	id T' E' $	Matched:id
$	T' E' $	T[T'][$] = T'->#
$	E' $	T[E'][$] = E'->#
$	$	Valid

Valid String!

Niyati Savant
TE - C31

2103156 Experiment - 5

Aim - Write a program to implement Three Address Code (TAC)

Theory -

- In analysis - synthesis model of compiler, front end translates source program into independent intermediate code and the backend uses this intermediate code to generate target code.
- The machine independent intermediate code enhances portability, facilitates retargeting, makes it easier to apply source code modification to improve performance of source code by optimization.
- Some common intermediate code representations include
1) Postfix Notation
2) Three Address Code
3) Syntax tree

TAC is the intermediate code which is easy to generate and can be easily converted to machine code. It use atmost 3 address and 1 operator to represent an expression and value computed at each instruction is stored in a temporary variable. The compiler decides the order of operation

General representation

$$a = b \ op \ c$$

Niyati Savant
TE - C31

2103156

where a, b, c represent operands & names, constants, temp variables

It can be implemented in 3 ways -

1) Quadruple
Structure with 4 fields - op, arg1, arg2, result
op : operator
arg1, arg2 : The two operands
result : store result.
. It is easy to rearrange code for global optimization and tempory variable can be quickly accessed using symbol table. However, lot of temporaries increase time and space complexity

Ex- Expression : $a = b * -c + b * -c$.

→ t_1 = uminus c ...(0) $t_2 = b * t_1$ ... (1)
t_3 = uminus c ...(2) $t_4 = b * t_1$ ... (3)
$t_5 = t_2 + t_4$ ...(4) $a = t_5$ ... (5)

#	Op	Arg1	Arg2	Result
(0)	Uminus	c		t_1
(1)	*	t_1	b	t_2
(2)	Uminus	c		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

Niyati Savant
TE - C31
2103156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

2] Triples

Instead of using an extra temporary variable to represent a single operation, a pointer to a triple is used when a reference to another triple value is needed. Thus, it has only 3 fields - op, arg1 and arg2. However, optimization and rearrangement is difficult

Eg expression a = b*-c + b* - c

#	op	arg1	arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

3] Indirect Triples.

Pointer is used for listing of all references to computations which is made and stored separately. Its similar in utility as compared to quadruple representation but needs less space.

Eg: a = b* - c + b* - c

Niyati Savant
TE - C31
2103156

#	Op	Arg 1	Arg 2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Conclusion
TAC is a compact intermediate representation facilitating efficient code generation and optimization in compilers

20/03/2024

EXPERIMENT 5

CODE:

```
print("Enter your choice: 1 for Assignment 2 for Arithmetic  3 to Exit")
choice = int(input())
while choice != 4:
    if choice == 1:
        # assignment
        print("enter the variable")
        var = input()

        print(var.split("="))
        print("t1 = ", var.split("=")[1])
        print(var.split("=")[0], "= t1")

    elif choice == 2:
        print("enter the expression")
        exp = input()
        i = 0
        while i < len(exp):
            if exp[i] == "+" or exp[i] == "-":
                if exp[i + 2] == "*" or exp[i + 2] == "/":
                    print("t1 = ", exp[i + 1], exp[i + 2], exp[i + 3])
                    print("t2 = ", exp[i - 1], exp[i], "t1")
                    break
                else:
                    print("t1 = ", exp[i - 1], exp[i], exp[i + 1])
                    print("t2 = t1", exp[i + 2], exp[i + 3])
                    break
            elif exp[i] == "*" or exp[i] == "/":
                if exp[i + 2] == "+" or exp[i + 2] == "-":
                    print("t1 = ", exp[i - 1], exp[i], exp[i + 1])
```

```
            print("t2 = t1", exp[i + 2], exp[i + 3])
            break
        elif exp[i] == "*" or exp[i] == "/":
            print("t1 = ", exp[i - 1], exp[i], exp[i + 1])
            break
        i += 1

    else:
        print("Invalid Operator")
```

OUTPUT:

```
PS C:\Engineering\3rd Year\Sem VI\PRACTICALS\SPCC\CODES> python -u "
\CODES\Trial.py"
Enter your choice: 1 for Assignment 2 for Arithmetic  3 to Exit
2
enter the expression
a=b-c*d
t1 =  c * d
t2 =  b - t1
enter the expression
```

Experiment. 6

Aim - 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 resources (CPU, Memory) so that faster-running machine code is generated. The optimizing process must meet the following objectives:

- correct optimization without change in meaning
- Increase speed and performance
- Reasonable compilation time
- No delay in overall process

It can be broadly classified into.

- Machine Independent optimization.
  Improve intermediate code for better target code
- Machine Dependent Optimization
  Done after generating target code.

Common Sub-Expression Elimination

Two Operations are common if they produce the same result. In such case, it is likely more efficient to compute result once and reference it to the ~~send~~ second time rather than

Eg:

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

$\Longrightarrow$ Post Transformation

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

re-evaluate it. An expression is alive if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.

Identify common sub-expression present in different expression, compute once, use the result in all places. The definition of variables involved in sub-expression elimination should not change

Conclusion -

Common sub expression elimination optimizes code by identifying and eliminating redundant computations, enhancing performance and reducing code redundancy

QA
20/03/2024

EXPERIMENT 6

CODE:

```
#pragma GCC optimize("Ofast")
#pragma GCC optimize("unroll-loops")

#include <bits/stdc++.h>

using namespace std;

typedef long long int          ll;
typedef unsigned long long int  ull;
typedef long double            ld;

typedef pair<ll,ll>            pll;
typedef vector<ll>             vll;
typedef vector<pll>            vpll;
typedef map<ll, ll>            mll;
typedef map<char, ll>          mcl;

#define inf                1<<30
#define mx9                1000000007
#define test(t)            int t; cin >> t; while(t--)
#define f(i, a, b)         for(int (i) = (a); (i) < (b); ++(i))
#define all(a)             (a).begin(), (a).end()
#define endl               "\n"
#define ff                 first
#define ss                 second
#define pb                 push_back
#define deb(x)             cout << #x << ": " << x << endl;
#define deb2(x,y)          cout << #x << ": " << x << " ~ " <<  #y << ": " << y << endl;
```

```
#define deba(arr)          cout << #arr << " ~ [ "; for(auto n: arr) cout << n << " "; cout << "]" << endl;
#define debap(arr)         cout << #arr << " ~ [ "; for(auto n: arr) cout << n.first << "-" << n.second <<
" "; cout << "]" << endl;
#define fast               ios_base::sync_with_stdio(false); cin.tie(NULL); cout.tie(NULL);
void solve()
{
    ll n;
    cout <<  " Enter no. of statements  "  << endl;
    cin >> n;
    set<string> st;
    map<string, vector<string>> mp;
    map<string, string> dmp;
    cout <<  " Enter the statements  "  << endl;
    f(i, 0, n) {
        string str;
        cin >> str;

        string temp;
        cin >> temp;

        string exp;
        getline(cin, exp);

        st.insert(exp);
        mp[exp].pb(str);
    }

    deba(st)
    for(auto it: mp) {
        deb(it.ff)
        deba(it.ss)
    }
```

```
    for(auto it: mp) {
        f(i, 1, it.ss.size()) {
            dmp[it.ss[i]] = it.ss[0];
        }
    }

    for(auto it: dmp) {
        deb2(it.ff, it.ss)
    }

    for(auto it: mp) {
        string lhs = it.ss[0];
        string rhs = it.ff;
        f(i, 0, rhs.length()) {
            if(rhs[i] == 't') {
                string mapped = "t";
                mapped += rhs[i + 1];
                if(dmp[mapped].length() == 2)
                    rhs[i + 1] = dmp[mapped][1];
            }
        }
        cout << lhs << " = " << rhs << endl;
    }
}

int main()
{
    fast;
    cout <<  " Niyati  "  << endl;
        solve();
    return 0;
}
```

OUTPUT:

```
Niyati
Enter no. of statements  9
Enter the statements
t1 = -c
t2 = a + b
t3 = a + b
t4 = a + b
t5 = d + e
t6 = a + b
t7 = -c
t8 = d + e
t9 = 4 * t4
 Niyati
 Enter no. of statements
 Enter the statements
st ~ [  -c  4 * t4  a + b  d + e ]
it.ff:  -c
it.ss ~ [ t1 t7 ]
it.ff:  4 * t4
it.ss ~ [ t9 ]
it.ff:  a + b
it.ss ~ [ t2 t3 t4 t6 ]
it.ff:  d + e
it.ss ~ [ t5 t8 ]
it.ff: t3 ~ it.ss: t2
it.ff: t4 ~ it.ss: t2
it.ff: t6 ~ it.ss: t2
it.ff: t7 ~ it.ss: t1
it.ff: t8 ~ it.ss: t5
t1 =  -c
```

t9 = 4 * t2

t2 = a + b

t5 = d + e

Niyati Savant
TE C31
2103156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Experiment 7

Aim - (a) Program to implement Pass 1 of Multi-Pass Assembler
(b) To implement Pass 2

Theory -
A two pass assembler can eliminate the forward reference problem and the two passes can process the definition of symbols. The first pass scans code, validate tokens and create symbol table.
Pass 1 looks for label definition and introduces them to symbol table. It defines symbols and literals. It keeps track of LC and determines length of machine instructions (MOT). It also tracks values of symbols until pass 2 is done (ST). It process some pseudo-opcode and stores literals in literal table (LT)

MOT - Machine Opcode Table (TMOT)

Mnemonic op-code (4 byte - characters	Binary op-code (1 byte - hexadecimal)	Instruction length (2 bit)	Instruction format (3 bit)
"Abbb"	5A	10	001
"AHbb"	4A	10	001
"ALbb"	5E	10	001
"ALRb"	1E	01	000
"ARbb"	1A	01	000
⋮	⋮	⋮	⋮
"MVCb"	D2	11	100

Niyati
TE CSI
156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

POT - Pseudo Opcode Table

Pseudo Opcode	Address of routine to process
5 byte	
"DROPb"	P1DROP
"ENDbb"	P1END
"EQUbb"	P1EQU
"START"	P1START
"USING"	P1USING
	∴ Presumed labels used here

Symbol Table (ST)

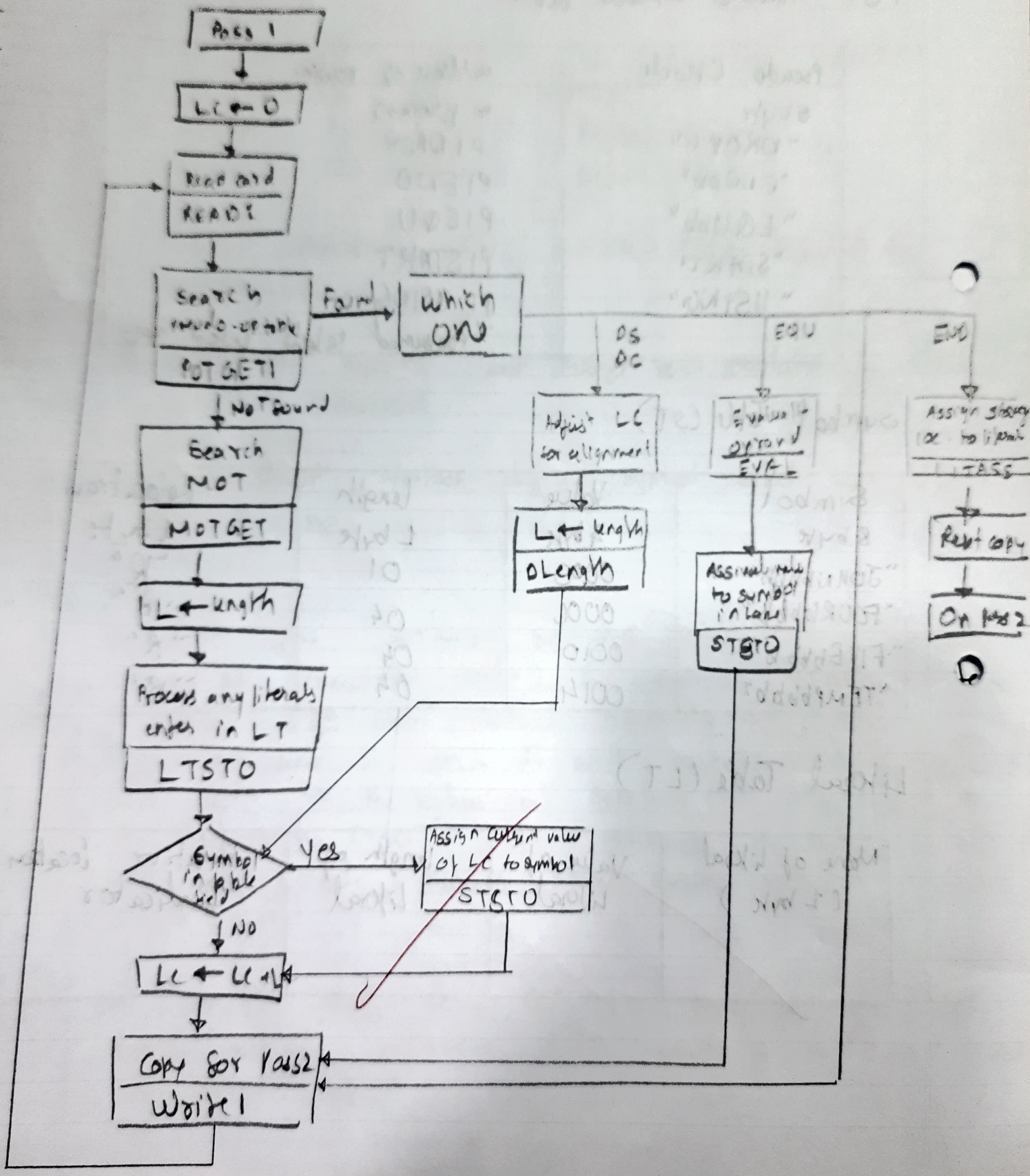
Symbol	Value	Length	Relocation
8 byte	4 byte	1 byte	1 byte
"JOHNbbbb"	0000	01	"R"
"FOURbbbb"	0006	04	"R"
"FIVEbbbb"	0010	04	"R"
"TEMPbbbb"	0014	04	"R"

Literal Table (LT)

Name of literal (1 byte)	Value of literal	length of literal	Relative location indicator

Niyati
156

Pass 1 :



Niyati
TE C31
15C

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

DS & DC
Are assembler directive DC is define constant and DS is define storage.
<label> DC <value> Define Constant
<label> DS <value> Define storage

<label> - symbolic value, <value> - value supplied by programmer.

EQU - define symbol and assign value evaluated by operand field expression

USING & DROP - Neither define symbol label nor affect LC. So no

Pass II
solves the forward reference problem and converts code to machine code. The symbol table completed in Pass 1, it does actual assembly by translating the operations and so on. It will generate object code. It looks up the values of symbols (ST) and generates the instructions (MOT). It generates the data (for DS, DC and literals). Process some pseudo-opcodes eg. USING, DROP (POT).

It uses base Table BT to generate the proper base register reference in machine instructions and its correct offsets

Niyati
TE C31
156

Availability indicator (1-byte-character)	Designated relative-address contents of base register (3 bytes hexadecimal)
1 "N"	-
2 "N"	-
⋮ ⋮	
14 "N"	-
15 "Y"	00 00 00

15 entries

Y ~ register specified in USING pseudo-opcode
N ~ register never specified in USING pseudo opcode or subsequent made unavailable by DROP

It generates object code into appropriate format for later processing by M. loader
- LC is initialized
- Statement is read from copy file created by pass1.
- Check Opcode to find if it is POT code or MOT code. The table in which entry is found gives the instruction length.

a) RR - format instruction - Register to Register Operation
b) RX - Register and Indexed Storage Operations
c) RS - Register and Storage Operations
d) SI - Storage and Immediate Operations
e) SS - Storage and Storage Operations

Niyati
TEC31
156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

: DS and DC
LC must be adjusted so that full words start on a byte whose address is a multiple of four.

· EQU
Pseudo-op only prints EQU card as a part of printed listing

· USING
Evaluate operand fields and mark corresponding BT entry as available

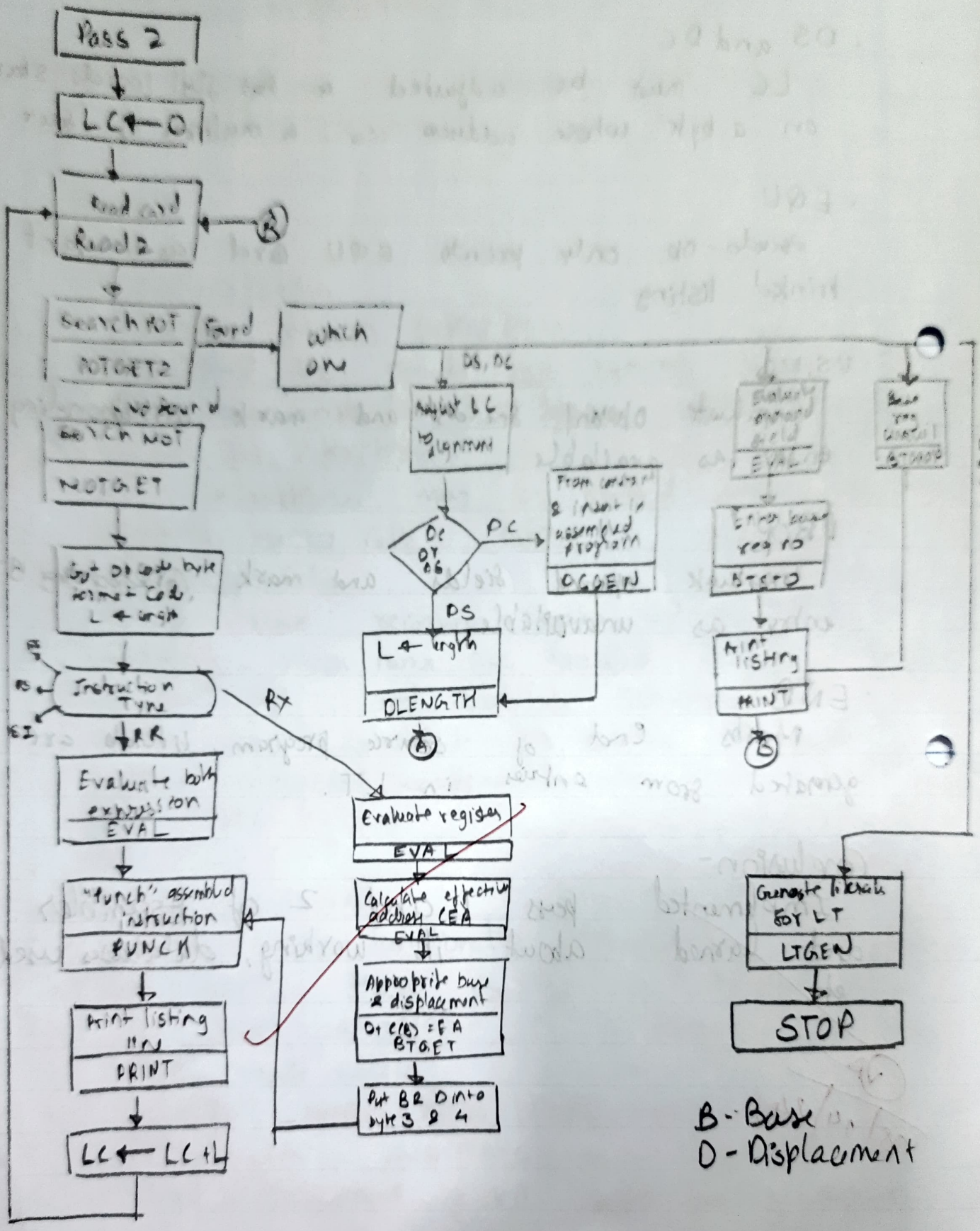
· DROP
Evaluate operand fields and mark corresponding BT entry as unavailable.

· END
Marks end of source program, literals are generated from entries in LT.

Conclusion-
Implemented pass 1 and 2 of Assembler and learned about its working, databases used etc.

08/04/24.

Niyati
TE C31
156



B - Base
D - Displacement

EXPERIMENT 7

CODE:

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
```

```
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

CODE:

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

Niyati Savant
TE - C31

2103156 Experiment 8

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Aim - Program to implement Multipass Macroprocessor

Theory -

→ Implementation

1) Recognize Macro Definition
· Identified by pseudo-ops MACRO and MEND and becomes complicated if definition within Macros.

2) Save the definition
Macro instructions must be stored and used for expanding Macro Calls later

3) Recognize Calls
Processor must recognise Macro call in operation mnemonic which name are handled as type of Opcode.

4) Expand Calls and substitute arguments
Processor must substitute for dummy or macro definition arguments. Resulting symbolic text is substituted for Macro Call

→ Assumptions made

1) Processor is functionally independent of Assembler
2) Output from Macro is fed to Assembler
3) The basic algorithm is implemention (No Calls or macro inside macro)
4) Processor never process EQU statement
5) Makes two systematic scans over the task.
6) Cannot expand a call before finding definition

Nivati
C31
156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

First Pass
- Examine every operation code
- Save all Macro definition in MDT
- Save copy of input, minus macro definition on Secondary storage for second pass.
- Prepare MNT to recognize macro call

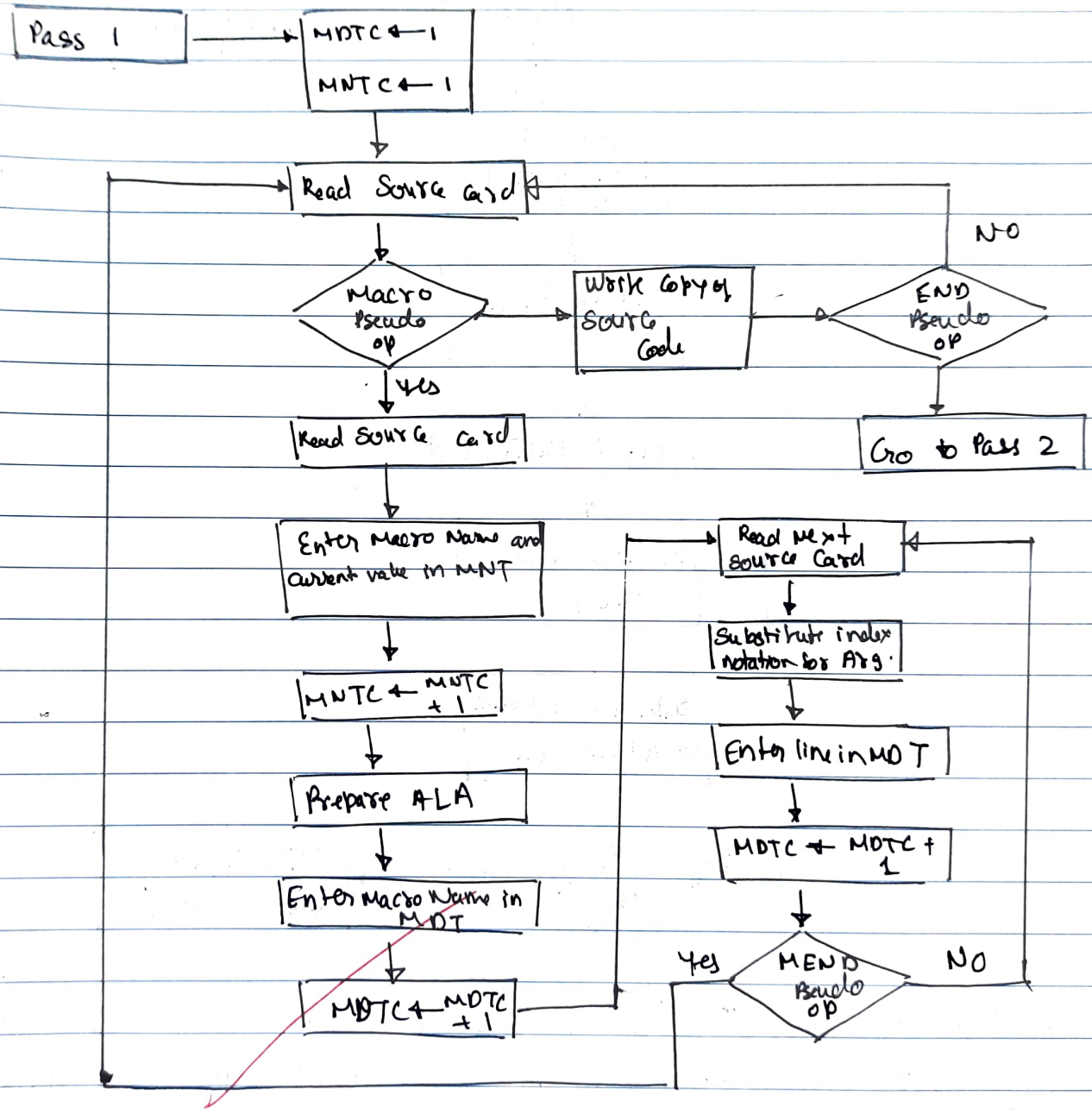
Second Pass
- Examine every operation mnemonic and check whether name is present in MNT.
- Replace each Macro name with text from definition

Data base Used-
Pass 1
- Input source Deck
- Output source deck copy for use
- MDT - Macro Definition Table - store body of definition
- MNT - Macro Name Table - Store name of macro
- MDTC - Indicate next available entry in MDT
- MNT - indicate next available entry in MNT
- ALA - Argument list Array - substitute index markers for dummary argumenty

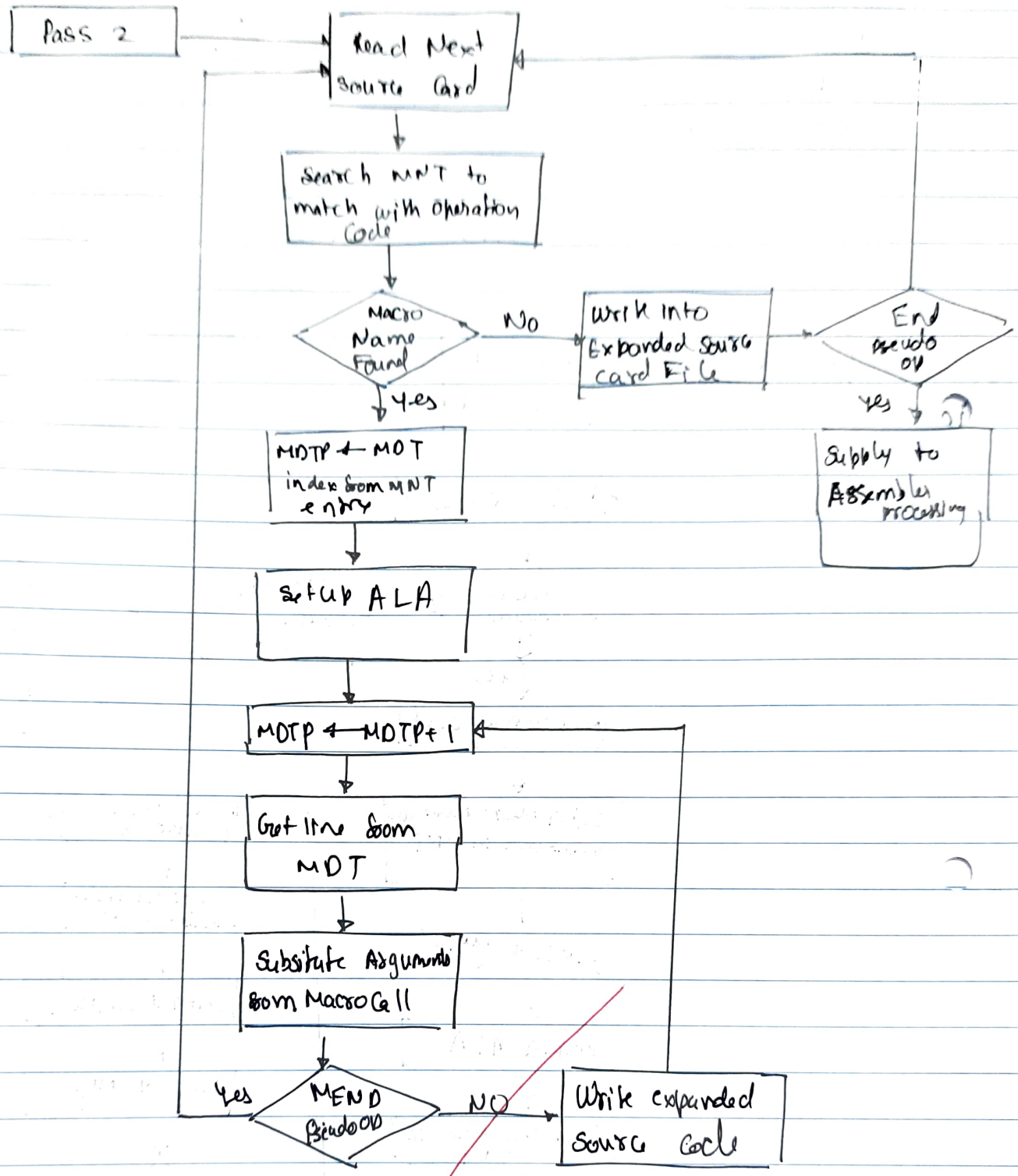
Pass 2
- Copy of input Macro source.
- Output expanded source deck to be used as input to assembler

Niyati
TE- C31
156

- MDT created by Pass 1
- MNT created by Pass 1
- MDTP - MDT Pointer to indicate next line of text to be used during expansion
- ALA - to substitute macro call argument



Niyati
TEC 31
156



Conclusion - ~~Studident~~ Studied and implemented macro processor ~~and~~ with its 2 passes and the databases.

08/04/24.

EXPERIMENT 8

CODE:

```
import java.util.*;
import java.io.*;

class twopassmacro {
   static String mnt[][] = new String[5][3]; // assuming 5 macros in 1 program
   static String ala[][] = new String[10][2]; // assuming 2 arguments in each macro
   static String mdt[][] = new String[20][1]; // assuming 4 LOC for each macro
   static int mntc = 0, mdtc = 0, alac = 0;

   public static void main(String args[]) {
      pass1();
      System.out.println("Macro Table(MNT)");
      display(mnt, mntc, 3);
      System.out.println("Argument Array(ALA) for Pass1");
      display(ala, alac, 2);
      System.out.println("Macronition Table(MDT)");
      display(mdt, mdtc, 1);
      pass2();
      System.out.println("Argument Array(ALA) for Pass2");
      display(ala, alac, 2);
      System.out.println(
            "Note:Alles are displayed here whereas intermediate pass1 output & expanded pass2
output is stored in files");
   }

   static void pass1() {
```

```
int index = 0, i;
String s, prev = "", substring;
try {
    BufferedReader inp = new BufferedReader(new FileReader("input.txt"));
    File op = new File("pass1_output.txt");
    if (!op.exists())
        op.createNewFile();
    BufferedWriter output = new BufferedWriter(new FileWriter(op.getAbsoluteFile()));
    while ((s = inp.readLine()) != null) {
        if (s.equalsIgnoreCase("MACRO")) {
            prev = s;
            for (; !(s = inp.readLine()).equalsIgnoreCase("MEND"); mdtc++, prev = s) {
                if (prev.equalsIgnoreCase("MACRO")) {
                    StringTokenizer st = new StringTokenizer(s);
                    String str[] = new String[st.countTokens()];
                    for (i = 0; i < str.length; i++)
                        str[i] = st.nextToken();
                    mnt[mntc][0] = (mntc + 1) + ""; // mnt formation
                    mnt[mntc][1] = str[0];
                    mnt[mntc++][2] = (++mdtc) + "";
                    st = new StringTokenizer(str[1], ","); // tokenizing the arguments
                    String string[] = new String[st.countTokens()];
                    for (i = 0; i < string.length; i++) {
                        string[i] = st.nextToken();
                        ala[alac][0] = alac + ""; // ala table formation
                        index = string[i].indexOf("=");
                        if (index != -1)
                            ala[alac++][1] = string[i].substring(0, index);
                        else
                            ala[alac++][1] = string[i];
                    }
```

```
                } else // automatically eliminates tagging of arguments in definition
                { // mdt formation
                  index = s.indexOf("&");
                  substring = s.substring(index);
                  for (i = 0; i < alac; i++)
                    if (ala[i][1].equals(substring))
                      s = s.replaceAll(substring, "#" + ala[i][0]);
                }
                mdt[mdtc - 1][0] = s;
              }
              mdt[mdtc - 1][0] = s;
            } else {
              output.write(s);
              output.newLine();
            }
          }
          output.close();
        } catch (FileNotFoundException ex) {
          System.out.println("Unableind file ");
        } catch (IOException e) {
          e.printStackTrace();
        }
      }

      static void pass2() {
        int alap = 0, index, mdtp, flag = 0, i, j;
        String s, temp;
        try {
          BufferedReader inp = new BufferedReader(new FileReader("pass1_output.txt"));
          File op = new File("pass2_output.txt");
          if (!op.exists())
```

```
    op.createNewFile();
  BufferedWriter output = new BufferedWriter(new FileWriter(op.getAbsoluteFile()));
  for (; (s = inp.readLine()) != null; flag = 0) {
    StringTokenizer st = new StringTokenizer(s);
    String str[] = new String[st.countTokens()];
    for (i = 0; i < str.length; i++)
      str[i] = st.nextToken();
    for (j = 0; j < mntc; j++) {
      if (str[0].equals(mnt[j][1])) {
        mdtp = Integer.parseInt(mnt[j][2]);
        st = new StringTokenizer(str[1], ",");
        String arg[] = new String[st.countTokens()];
        for (i = 0; i < arg.length; i++) {
          arg[i] = st.nextToken();
          ala[alap++][1] = arg[i];
        }
        for (i = mdtp; !(mdt[i][0].equalsIgnoreCase("MEND")); i++) // expand till MEND
        {
          index = mdt[i][0].indexOf("#");
          temp = mdt[i][0].substring(0, index);
          temp += ala[Integer.parseInt("" + mdt[i][0].charAt(index + 1))][1]; // converting
                                                    // char->string->integer
                                                    // & appending it
          output.write(temp);
          output.newLine();
        }
        flag = 1;
      }
    }
    if (flag == 0) // when it is not a macro
    {
```

```
                output.write(s);
                output.newLine();
            }
        }
        output.close();
    } catch (FileNotFoundException ex) {
        System.out.println("Unableind file ");
    } catch (IOException e) {
        e.printStackTrace();
    }
  }

  static void display(String a[][], int n, int m) {
    int i, j;
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            System.out.print(a[i][j] + " ");
        System.out.println();
    }
  }
}
```

OUTPUT:

Macro Table(MNT)

1 INCR1 1

2 INCR2 5

Argument Array(ALA) for Pass1

0 &FIRST

1 &SECOND

2 &ARG1

3 &ARG2

Macronition Table(MDT)

```
INCR1     &FIRST, &SECOND=DATA9
A          1,#0
L          2,#1
MEND
INCR2     &ARG1, &ARG2=DATA5
L          3,#2
ST         4,#3
MEND
```

Argument Array(ALA) for Pass2

```
0     DATA1
1     DATA2
2     DATA3
3     DATA4
```

Niyati Savant
2103156
C31

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Experiment - 9

30/1/23

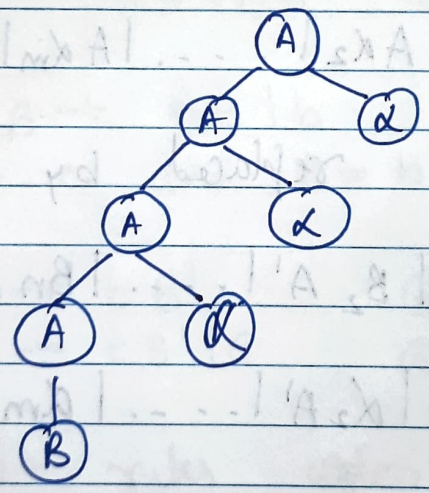
Aim - Program to eliminate left recursion from given grammar

Theory -

A Grammer G (V, T, P, S) is left recursive if it has a production of the form.

$A \rightarrow A\alpha | B$

It is left recursive as the left side of production is occurring at a first position on right side of production. In a left recursive grammar, expansion of A generate $A\alpha$, $A\alpha\alpha$, $A\alpha\alpha\alpha$ at each step causing it to enter an infinite loop



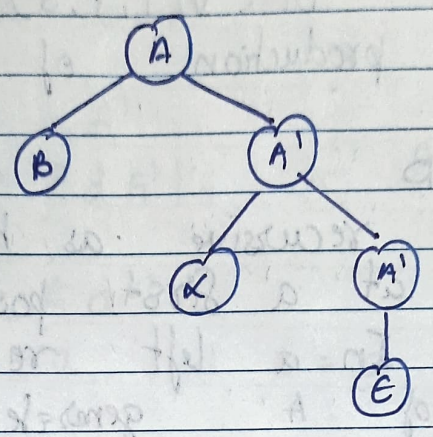
This left recursion can be eliminated by replacing a pair of production as -

Niyati Savant
2103156
C31

$A \rightarrow BA'$

$A' \rightarrow \alpha A' \mid \epsilon$

Thus, we can prevent an infinite loop



A more general form for recursion is

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \beta_n$

which can be replaced by

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

$A \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$

Example-

$A \rightarrow ABd \mid Aa \mid a$

$B \rightarrow Be \mid b$

Niyah Savant
2103156
C31

The left recursion occurs in both production

For i) $A \rightarrow ABd \mid Aa \mid a$

ia) $A \rightarrow ABd \mid a$

here $\alpha = Bd$ and $\beta = a$, thus new rules are

$A \rightarrow aA'$ ① .... $A \rightarrow \beta A'$

$A' \rightarrow BdA' \mid \epsilon$ ② $A' \rightarrow \alpha A' \mid \epsilon$

ib) $A \rightarrow Aa \mid a$

here $\alpha = a$ and $\beta = a$, thus rules are

$A \rightarrow aA'$

$A' \rightarrow aA' \mid \epsilon$ ③

For ii) $B \rightarrow Be \mid b$

$\alpha = e$ and $\beta = b$, thus rules are

$B \rightarrow bB'$ ④

$B' \rightarrow eB' \mid \epsilon$ ⑤

→ Thus, final rules after removing left recursion

i) $A \rightarrow aA'$

ii) $A' \rightarrow aA' \mid BdA' \mid \epsilon$

iii) $B \rightarrow bB'$

iv) $B' \rightarrow eB' \mid \epsilon$

Niyati Savant
2103156
C31

Conclusion

Removing left-recursive grammar is essential for preventing parsing ambiguities and facilitating efficient top-down parsing techniques. It ensures a more straightforward and unambiguous parsing process, contributing to the overall clarity and effectiveness of grammar in language processing.

20/03/2024

EXPERIMENT 3

Aim : LEFT RECURSION

Code:

```
def remove_left_recursion(grammar):
    new_grammar = {}
    for non_terminal, productions in grammar.items():
        alphas = []
        betas = []
        for production in productions:
            if production.startswith(non_terminal):
                alphas.append(production[1:])
            else:
                betas.append(production)
        if alphas:
            new_non_terminal = non_terminal + "'"
            new_grammar[non_terminal] = [beta + new_non_terminal for
beta in betas]
            new_grammar[new_non_terminal] = [alpha + new_non_terminal +
"|" for alpha in alphas] + ["e"]
        else:
            new_grammar[non_terminal] = productions
    return new_grammar

def print_grammar(grammar):
    for non_terminal, productions in grammar.items():
        print(non_terminal + " -> " + " | ".join(productions))


grammar = {}
n = int(input("Enter the number of grammar rules: "))
for _ in range(n):
      rule = input("Enter grammar rule: ").split(" -> ")
      non_terminal = rule[0]
      productions = rule[1].split("|")
      grammar[non_terminal] = productions

updated_grammar = remove_left_recursion(grammar)
print("\nGrammar after left recursion removal:")
print_grammar(updated_grammar)
```

Output:

Enter the number of grammar rules: 3

Enter grammar rule : E -> E+T|T

Enter grammar rule : T -> T*F|F

Enter grammar rule : F -> (E)|id

Grammar after left recursion removal:

E -> TE'

E' -> +TE'| | e

T -> FT'

T' -> *FT'| | e

F -> (E) | id

Niyati Savant
TE - C31

2103156 Experiment - 10

Aim - Program to implement Code Generation

Theory -

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of the code generator should be such that it can be easily implemented, tested and maintained

Some issues arising during code generation phase-
1) Input to code generator
2) Target program
3) ~~Mem~~ Instruction Selection
4) Register Allocation
5) Evaluation order

Thus, code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some design goals include - Correct, easily maintainable, testable and efficient

Code generator is used to produce the target code for three-address statements. It uses register to store the operands of 3 address statement

Niyati Samnt
TE - C31
2103156

Eg :- d := (a-b) + (a-c) + (a-c)
can be translated into the following three-address code sequence:

t := a - b
u := a - c
v := t + u
d := v + u

with d live at the end

Statements	Code Generated	Register Descriptor	Address descriptor
		Register Empty	
t = a-b	Mov a, R0 Sub b, R0	R0 contains t	t in R0
u = a-c	Mov a, R1 Sub c, R1	R0 contains t R1 contains u	t in R0 u in R1
v = t+u	Add R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d = v+u	Add R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

Niyati Savant
TE - C31

2103156

Eg for 3 address statement $x := y + z$ can have -

MOV x, R_0
ADD y, R_0

- A register descriptor contains the track of what is currently in each registers. It shows that all registers are initially empty.
- An address descriptor is used to store the location where current value of the name can be found at run time.

Algorithm -

It takes a sequence of TAC statements as input.
For each statement of form a := b op c perform various actions -

1. Invoke function getreg to find location L to store result
2. Consult address description for y to get y'. If value of y in register and memory both, then prefer y'. If value of y is not in L generate instruction MOV y', L to place copy of y in L
3. Generate instruction OP z', L where z' shows current location of z if z is in both, prefer register. Update the address descriptor of x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all.
4. If current value of y or z have no next uses or not live on exit from block or in register, alter descriptor to indicate that after execution, registers no longer contain y or z.

Conclusion -

The working of code generator was understood and how it takes an input of TAC statements to generate target code

20/03/2024

EXPERIMENT 10

CODE:

```
op1, op2, op3, op4 = "", "", "", ""
print("Niyati Savant")
k = int(input("Enter no of lines"))
print("Enter operation and operands (e.g., + o1 o2): ")
loaded_var= ['a','b','c']
i=0
while k:
    TAC = input().split()
    k -=1
    op1=TAC[0]
    op2=TAC[1]
    op3=TAC[2]

    if op1 == "+" :
        if op2 not in loaded_var:
            print(f"MOV {op2},R{i}")
        print(f"ADD R{i}, R{i+1}")

    elif op1 == "-":
        print(f"MOV {op2},R{i+1}")
        print(f"SUB {op3},R{i+1}")

    elif op1 == "*":
        print(f"MOV {op2},R{i}")
        print(f"MUL R{i}, R{i+1}")

    elif op1 == "/":
        print(f"MOV {op2},R{i}")
        print(f"DIV R{i}, R{i+1}")
```

```
    elif op1 == "=":
        print(f"MOV {op2}, {op3}")
```

OUTPUT:

Niyati Savant

Enter no of lines 4

Enter operation and operands (e.g., + o1 o2):

- a b

MOV a,R0

SUB b,R0

- a c

MOV a,R1

SUB c,R1

+ t u

ADD R1, R0

+ v u

ADD R1, R0

MOV v,R0

Niyati Savant
TE-C31
2103156

Written Assignment - 01

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

08/04/2024.

Q1] With reference to assembler, consider any assembly program and generate Pass 1 and Pass 2 output. Also show contents of Database table involved

→ Sample Program :

```
PRG     START   0
        USING   *,15
        L       1,FOUR
        A       1,FIVE
        A       1,=F'2'
        ST      1,TEMP
FOUR    DC      F'4'
FIVE    DC      F'5'
TEMP    DC      1F
END
```

First Pass :

Relative Address	Mnemonic Instruction
0	L 1,-(0,15)
4	A 1,-(0,15)
8	A 1,-(0,15)
12	ST 1,-(0,15)
16	4
20	5

Niyati
C31
156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

MOT

Mnemonic OpCode	Binary OpCode	length	Format	Not used in this design
L	01011000	4	RX	--
A	01011010	4	RX	--
ST	01010000	4	RX	--

POT

Pseudo-Op	Address of routine
START	PA_START
USING	PA_USING
DC	PA_DC
DS	PA_DS
END	PA_END

Symbol Table

Symbo	Value	length	Relocation
PRG	0	1	R
FOUR	16	4	R
FIVE	20	4	R
TEMP	24	4	R

Literal Table

	Availability Indicator	Base Register Content
1	N	
⋮		
15	Y	00

Niyati
C31
156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Second Pass

Relative Address	Mnemonic Instruction
0	L 1,16 (0,15)
4	A 1, 20 (0,15)
8	A 1, 28 (0,15)
12	ST 1,24 (0,15)
16	4
20	5
24	

Q2] Short Note on YACC

→ Each translation rule input to YACC has a string specification that resembles a production of a grammar. it has a nonterminal on the LHS and a few alternatives on RHS. For simplicity, we will refer to a string specification as production. YACC generates an LALR(1) parser for language L from the productions, which is a bottom-up parser. The parser would operate as follows: For a shift action, it would invoke the scanner to obtain the next token and continue the parse by using that token. While performing a reduced action in accordance with production it would perform the semantic action associated with that production.

The semantic actions associated with productions

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

achieve the building of an intermediate representation or target code as follows:

- Every non terminal symbol in the parser has an attribute
- The semantic action associated with a production can access attributes of nonterminal symbols used in that production - a symbol "$n" in the semantic action, where n is an integer designates the attribute of the nonterminal symbol in the RHS of production and symbol '$$' designates the attribute for building intermediate representation or target code

A parser generator will take syntax specification as input and produce as output a procedure for recognizing that language. Historically, they are also called compiler compilers. YACC (yet another compiler - compiler) is an LALR(1) parser generator. It was originally designed for being complemented by Lex. The definition part includes information about tokens used. Yacc auto assigns numbers for tokens. Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.

Niyati
TE-C31
2108156

Written Assignment -02

08/04/24

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Q1. Explain working of Direct Linking Loader with example showing their entries in different databases built in DLL.

→ Loader is the system program which is responsible for preparing the object program for execution and initiate the execution. The loader does the job of co-ordinating with the OS to get initial loading address for the .EXE file and load it into the memory.

Loader Functions include-

- Allocation - Allocates space in memory where object program would be loaded for Execution
- Linking - It links two or more object codes and provides information needed to allow references between them
- Loading - It brings object program into memory for execution

Dynamic linking ~~took~~ Loader

- It is a general re-locatable loader.
- Allowing programmer multiple procedure segments and multiple data segments and giving programmer complete freedom in referencing data or instruction contained in other segments.
- The assembler must give the loder the following information with each procedure or data segment
- It defers much of linking until a program starts running. It provides a variety of benefits that are hard to get otherwise. Dynamically linked

shared libraries are easier to create and update. The semantics of these libraries can be much close to those of unshared libraries. It permits a program to load and unload routines at run time, a facility that can otherwise be difficult to provide

The databases include

1. External symbol Dictionary (ESD)
   It record combine information about all the symbol that are defined in this program
1. Text Cards (TXT)
   Text card record control the actual object code translated version of the source program.
1. Relocation & Linkage Directory (RLD)
   It contains information about program location
1. END CARD:
   The END card records indicates the end of the object file and specifies the start

Example;

END CARD

RA		
0	PG1	START
4	ENTRY	A, B
8	EXTRN	PG2, C
	|	
20	A	DC
24	B	DC
	|	
40	A+20	

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

44	B-25	
48	C-5	
52	END	

ESD

Symbol	Type	Id	RA	length
PG1	SD	1	0	52
A	LD	1	20	
B	LD	1	24	
PG2	ER	2	-	
C	ER	3	-	

RLD - Relocating linking directory

Esd id	Symbol	Flag	R.A
1	A	+	40
1	B	-	44
3	C	-	48

Niyati Savant
TE - C31
2103156

Written Assignment - 3

08/04/24

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

Q Write Short Note on:

1. Editors and Types of Editors

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

- Line Editor: where user 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.
  Ex: Teleprinter, edlin, teco

- Stream Editor - The File is treated as continuous flow or sequence of characters instead of line numbers, which means you can type paragraphs
  Ex: Sed editor in UNIX

Niyati Sawant
TE - C31
2103156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

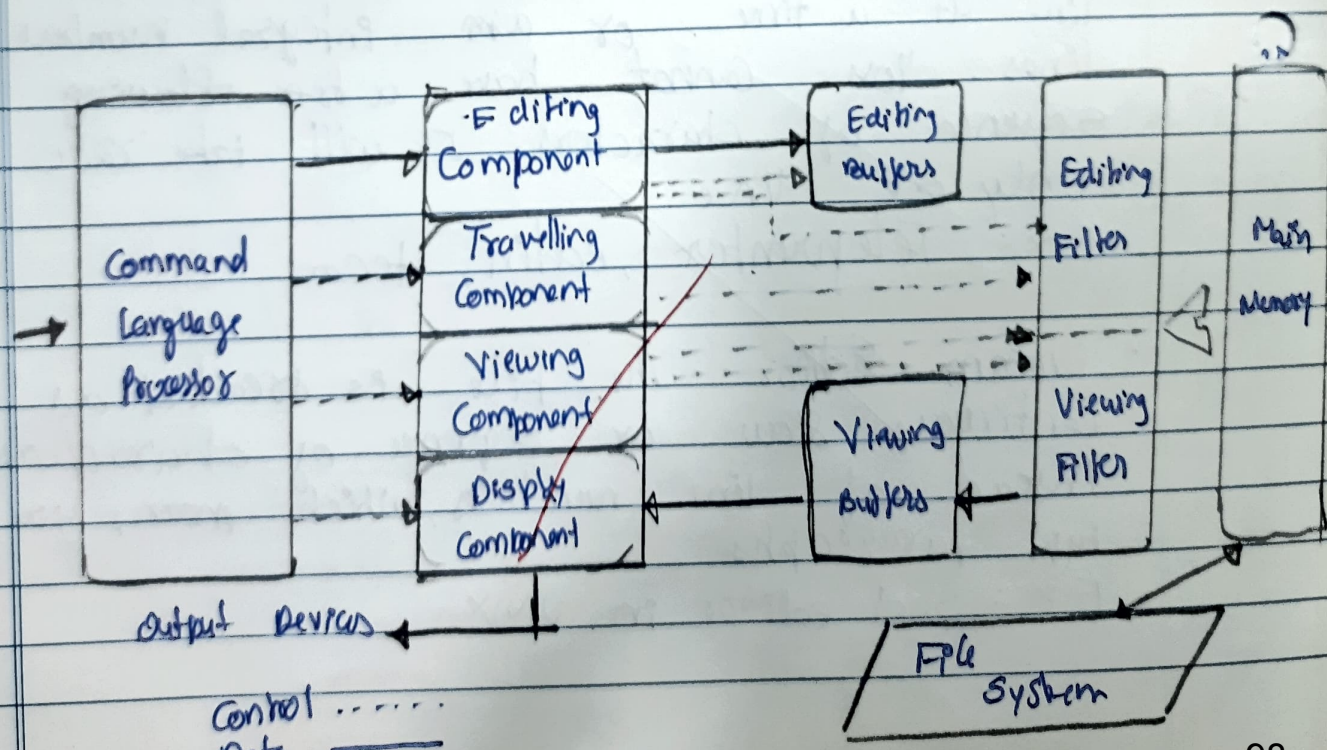
- Screen editors - The user is able to see the cursor on the screen and can make a copy, cut, paste operation easily. It is very easy to use mouse pointer. Eg: Vi, emacs, Notepad.

- Word processor - Overcoming the limitations of screen editors, it allows one to use some format to insert images, tables, videos, use font, size, style features. Majorly focuses on Natural language.

- Structure Editor - Structure editor focuses on programming languages. Allows to write & edit source code. Ex: Netbeans, IDE, gEdit

Some other editors include Full screen Editors, multiple window editor, DOS-Editor, VI editor, Online Editors

Editor Structure



Niyati.
TE C31
156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

2. Backpatching in Intermediate Code Generation

→ It is the process of fulfilling unspecified information of labels. It basically uses the appropriate semantic actions during code generation. It may indicate the address of label in goto statements while producing TACs for the given expressions. Here basically two passes are used because assigning the positions of these label statements in one pass is quite challenging. It can leave these addresses unidentified in the first pass and then populate them in second round. Backpatching is filling up gaps in incomplete transformations and information

It is mostly used for two purposes
- Boolean expression whose results can be either true or false.
- Flow of control statements must be controlled during statements execution
- Labels & Gotos - the most elementary programming construct for changing the

Applications of Backpatching:
- Backpatching is used to translate flow-of-control statements in one pass itself.
- It is used for producing quadruples

Niyati
SE TE-C31
2103156

THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

for boolean expressions during bottom-up parsing.
- It is the activity of filling up unspecified information of labels during generation process.
- It helps to resolve forward branches that have been planted in the code

Advantages
- Simplicity
- Flexibility
- Code Optimization
- Error Handling
- Efficiency

Disadvantages
- Complexity
- Performance Overhead
- Debugging Difficulty
- Potential for Errors
- Limited Applicability.