Thadomal Shahani Engineering College

Bandra (W.), Mumbai- 400 050.

CERTIFICATE

Certify that Mr./~~Miss~~ Om Anirudha Shete of COMPUTER ENGG. Department, Semester VI with Roll No. 2103163 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

10/04/24

Teacher In-Charge

Head of the Department

Date 10/4/24

Principal

THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

CONTENTS

Om Shet
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
TE C32
2103163

Experiment No : 1

Aim : Write a program to implement Lexical Analyzer

Theory :

- **Lexical Analysis**

⇒ Lexical analysis is the first phase of a compiler. The main task accomplished by lexical analysis is to identify the set of a valid word of the language that occurs in an input stream.

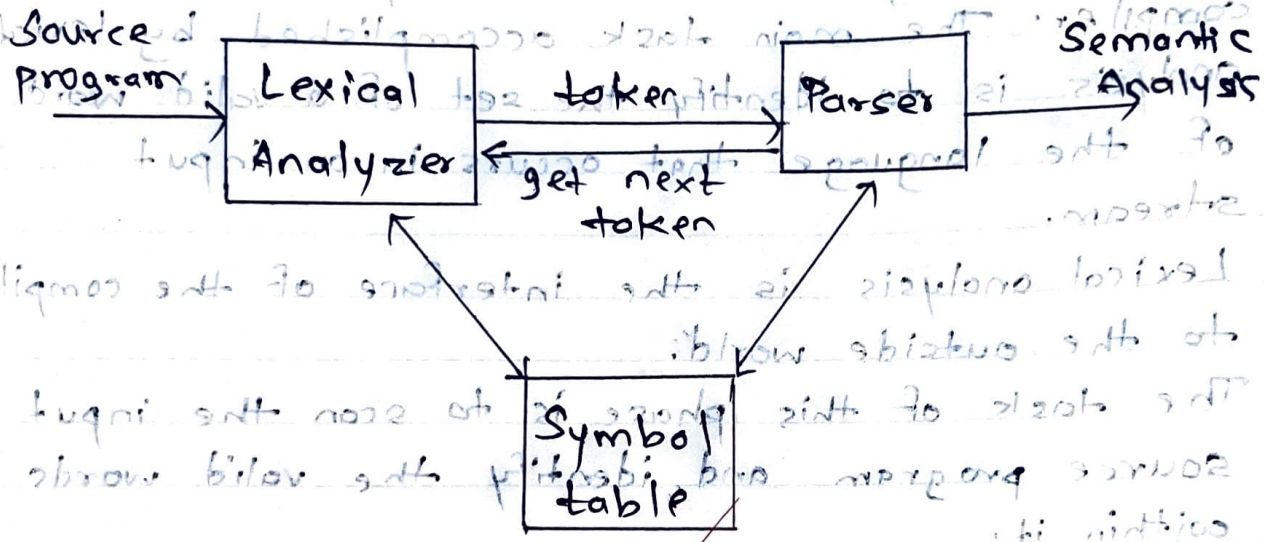
- Lexical analysis is the interface of the compiler to the outside world.
- The task of this phase is to scan the input source program and identify the valid words within it.
- It also does the additional job of removing cosmetics like

i) Extra white spaces, comments added by the user etc from the program.

ii) Expanding the user defined macros like #include and # define in C, reporting of foreign words that are the words, not belonging to the language of the source program if any.

- In computer science, lexical analysis is the first stage of processing the language.
- It is the process of converting a sequence of characters into a sequence of tokens.
- Stream of characters making up the source program or other input is read one at a

Role of Lexical Analyzer :



time and grouped into lexemes.
- A program function which performs lexical analysis is called a lexical analyzer, lexer or scanner.
- A lexical/lexer often exists as a single entity which is called by the parser.

• **Lexeme**

⇒ A lexeme is a basic unit of a language consisting of one word or several words. That corresponds to a set of words that are different forms of the same word
- For e.g., run, runs, running, ran are example or terms of same lexeme.

• **Tokens**

⇒ Lexical token/tokens are sequence of a characters that can be treated as a unit in the grammar of programming languages.
- For e.g.,

1) Type token ("id" {a-z, A-Z, 0-9}, "num" {0-9}...)

2) Punctuation tokens ("if", "void", "return", "begin", "call", "const", "do", "end", ...)

3) Non-tokens

- **Pattern**

⇒ A pattern is a set of a string in input for which the same token is produced as output. These set of strings is described by a rule called "pattern".

- For e.g.,

Token	Lexeme	Pattern
Literal	"hello"	chars with in " "
Number	0.31, 0.5	List of numbers.
if	If	if
const	const	const

- **Conclusion :**

In this experiment, we learnt about the basics of lexical analysis.

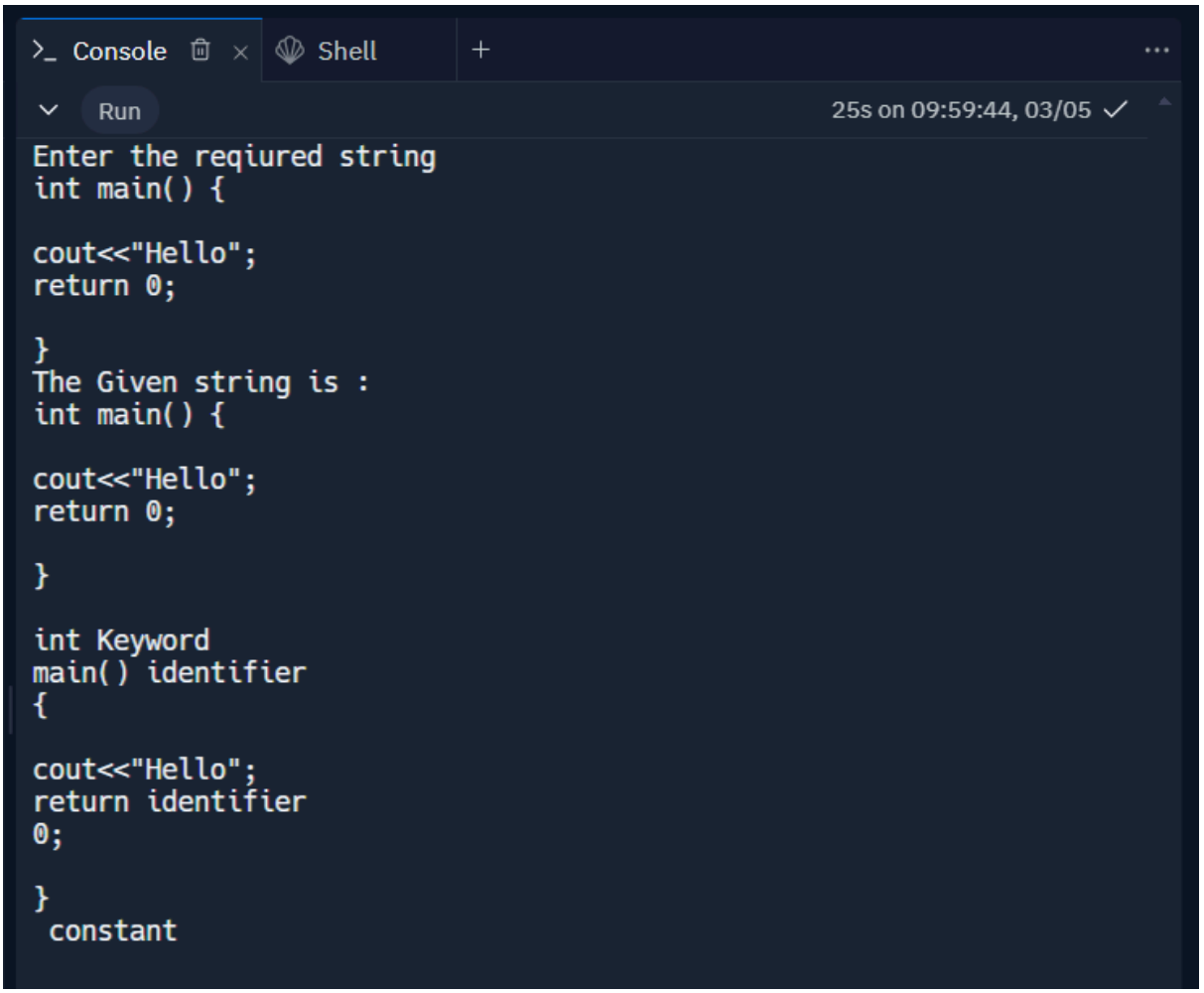
✓A
22/02/2024

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  string x;
  cout << "Enter the reqiured string" << endl;
  getline(cin, x, static_cast<char>(EOF));
  cout << "The Given string is :" << endl;
  cout << x << endl;

  set<string> sep = {"\n", ";", "\t"};
  set<string> key = {"int", "float", "cin", "cout", "for", "if",
"else"};
  set<string> op = {"=", "+", "-", "*", "%", "/", "<=", ">=", "=="};
  vector<int> token;

  stringstream ss(x);
  string word;
  vector<string> all;
  while (getline(ss, word, ' ')) {
    all.push_back(word);
  }

  for (int i = 0; i < all.size(); i++) {
    if (sep.find(all[i]) != sep.end()) {
      cout << all[i] << " "
           << "seperator" << endl;
    } else if (op.find(all[i]) != op.end()) {
      cout << all[i] << " "
           << "operator" << endl;
    } else if (key.find(all[i]) != key.end()) {
      cout << all[i] << " "
           << "Keyword" << endl;
    } else if (all[i][0] >= '0' and all[i][0] <= '9') {
      cout << all[i] << " "
           << "constant" << endl;
    } else {
      cout << all[i] << " "
           << "identifier" << endl;
    }
  }
```

```
  return 0;
}
```

Output:



Experiment NO: 2

Aim: To study and implement programs LEX and YACC tool.

Theory:

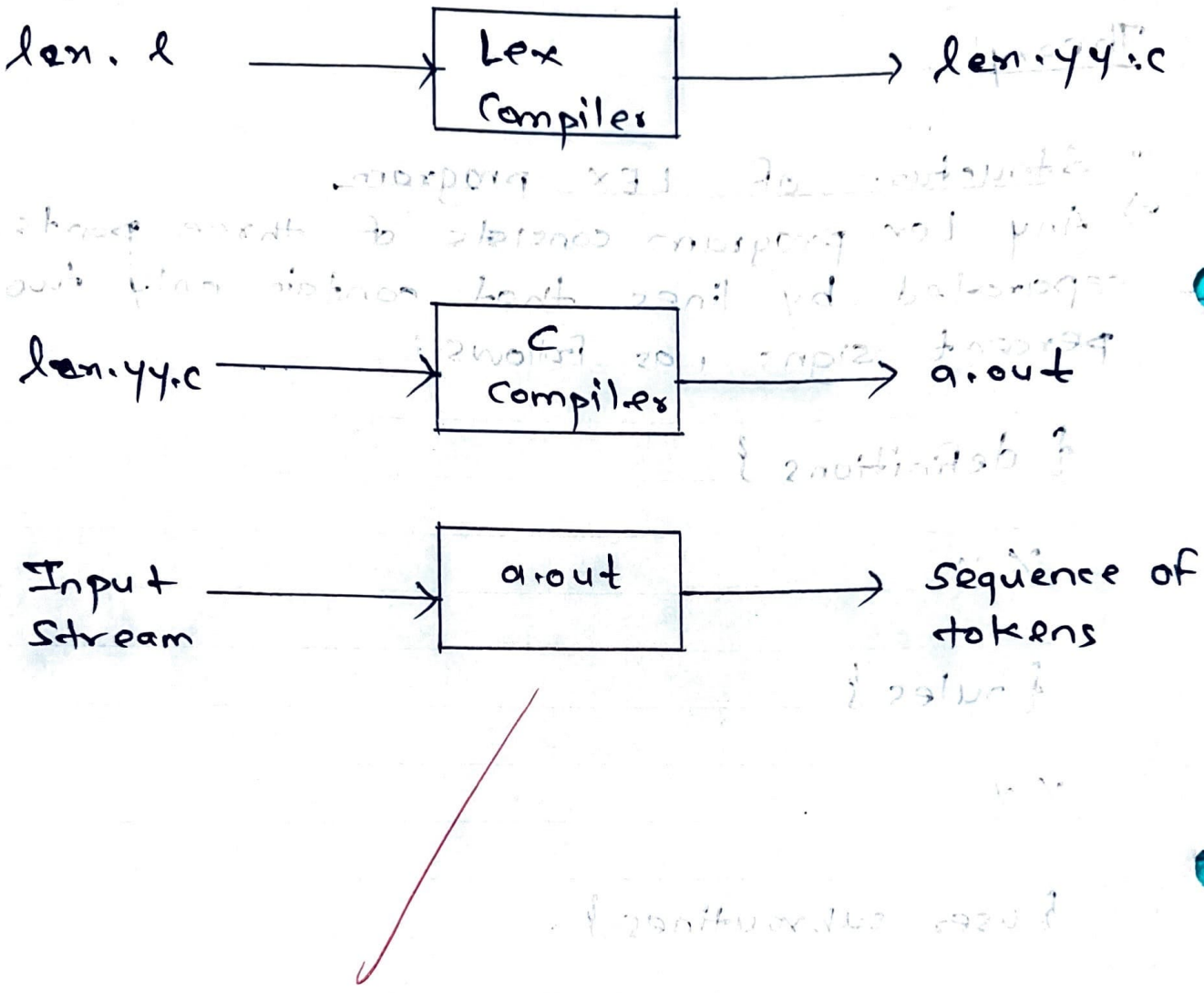
- **Structure of LEX program**

⇒ Any Lex program consists of three parts, separated by lines that contain only two percent signs, as follows:

```
{ definitions }

%%

{ rules }

%%

{ user subroutines }
```

- The definition section is the place to define macros and to import header files written in C.
- It includes variable declarations, constant declarations, and regular definitions.
- The rules section is the most important section; it associates patterns with C statements.

LEX Compiler :



THADOMAL SHAHANI TSEC ENGINEERING COLLEGE

- The rule section contains the specification of token and the associated actions.
- This section is of form:

```
P1      {action 1}
P2      {action 2}
P3      {action 3}
.  .  .  .  .  .  .
Pn      {action n}
```

- Here, Pi is a regular expression.
- The user subroutines contains code called by the ~~until~~ rules in the rules section.

- Compilation steps:

1) flex program.l

2) gcc lex.yy.c -lfl

3) ./a.out

- Conclusion:

In this experiment, we learnt about the data structure of the lex program

DA
22/02/2024

Code-Output:

```
%{
/* Program to show the message when an ENTER key is pressed*/
%}

%%

[\n] {
        printf("\n\nHi.....Good Morning...\n");
        return;
        }

%%

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

```
root@dellaio304: /home/complab304pc4/Desktop/163
root@dellaio304:/home/complab304pc4/Desktop/163# flex program4.l
root@dellaio304:/home/complab304pc4/Desktop/163# flex program1.l
root@dellaio304:/home/complab304pc4/Desktop/163# gcc lex.yy.c -lfl
program1.l: In function ‘yylex’:
program1.l:9:9: warning: ‘return’ with no value, in function returning non-void
    9 |         return;
      |         ^~~~~~
lex.yy.c:607:21: note: declared here
  607 | #define YY_DECL int yylex (void)
      |                     ^~~~~
lex.yy.c:627:1: note: in expansion of macro ‘YY_DECL’
  627 | YY_DECL
      | ^~~~~~~
program1.l: At top level:
program1.l:14:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   14 | main()
      | ^~~~
root@dellaio304:/home/complab304pc4/Desktop/163# ./a.out


Hi.....Good Morning...
root@dellaio304:/home/complab304pc4/Desktop/163#
```

```
%{

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');
}
```

```
root@dellaio304: /home/complab304pc4/Desktop/163
root@dellaio304:/home/complab304pc4/Desktop/163# ./a.out

What is your name?Om

 Hi........Om......Good Morning

Press y to continue
```

```
%{
void display(int, char *);
int flag;
%}

%%
[a|e|i|o|u]+ {
        flag = 1;
        display(flag, yytext);
        }
.       {
                flag = 0;
                display(flag, yytext);
}

%%

void main()
{
        printf("\nEnter a character to check whether it is vowel or NOT\n");
        yylex();
}

void display(int flag, char *t)
{
        if(flag==1)
                printf("\nThe given character %s is a vowel\n", t);
        else
                printf("\nThe given character %s not is a vowel\n", t);
}
```

```
root@dellaio304: /home/complab304pc4/Desktop/163
root@dellaio304:/home/complab304pc4/Desktop/163# flex program3.l
root@dellaio304:/home/complab304pc4/Desktop/163# gcc lex.yy.c -lfl
root@dellaio304:/home/complab304pc4/Desktop/163# ./a.out

Enter a character to check whether it is vowel or NOT
i

The given character i is a vowel

k

The given character k not is a vowel
```

```
%{
void display(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("\nThe string\n\t%s\t\t is a word\n", string);
        else if(flag==0)
                printf("\nThe given string\"%d\"is a digit\n", atoi(string));
        else
                printf("\nThe given string is either a word or a digit\n");
}

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

```
root@dellaio304: /home/complab304pc4/Desktop/163
root@dellaio304:/home/complab304pc4/Desktop/163# ./a.out

Enter a string to check whether it is word or digit
564

The given string"564"is a digit
root@dellaio304:/home/complab304pc4/Desktop/163#
```

```
%{
#include<stdio.h>
%}

%%

[0-9]+          {printf("NUMBER: %s\n", yytext);}
[ ]             {printf("WHITE SPACE: %s\n", yytext);}
("n"|"t")       {printf("ESCAPE SEQUENCE: %s\n", yytext);}
[,;]            {printf("SEPERATOR: %s\n", yytext);}
[()]            {printf("PARANTHESIS: %s\n", yytext);}
[{}]            {printf("BRACKETS: %s\n", yytext);}
[-+/*%=]        {printf("OPERATORS: %s\n", yytext);}
("else"|"if"|"int"|"void"|"main"|"return"|"for"|"while") {printf("KEYWORDS: %s\n", yytext);}
[a-zA-z]+       {printf("IDENTIFIER: %s\n", yytext);}

%%

int main(){
        yylex();
        return 0;
}
```

```
root@dellaio304: /home/complab304pc4/Desktop/163
root@dellaio304:/home/complab304pc4/Desktop/163# ./a.out
(Om) {Shete} 0550
PARANTHESIS: (
IDENTIFIER: Om
PARANTHESIS: )
WHITE SPACE:
BRACKETS: {
IDENTIFIER: Shete
BRACKETS: }
WHITE SPACE:
NUMBER: 0550
```

Om Shele
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
TE C32
2103163

Experiment No. 3

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

Theory:

- In compiler design, "FIRST" and "FOLLOW" are sets used in the context of parsing techniques, particularly for constructing predictive parser.

- **FIRST set**

⇒ The First set of a non-terminal in a grammar consists of the terminal symbols that can begin the strings derivable from that non-terminal.

- Algorithm:

1) If X is a terminal, then FIRST(X) = {X}.
2) If X is a non-terminal and X → ε is a production, then add ε to FIRST(X).
3) If X is a non-terminal and X → $Y_1 Y_2 \dots Y_k$ is a production, add FIRST($Y_1, Y_2, \dots Y_k$) - {ε} to FIRST(X)
4) Repeat steps 2-3 until no more additions can be made.

For e.g.,

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

⇒

	FIRST	FOLLOW
$E \rightarrow E+T \mid T$ $T \rightarrow T*F \mid F$ $F \rightarrow (E) \mid id$	{ (, id }	

We have left recursion in above grammar

⇒ After removing left recursion,

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E'	{ +, ε }	{ $,) }
T	{ (, id }	{ +, $,) }
T'	{ *, ε }	{ +, $,) }
F	{ (, id }	{ *, +, $,) }

- **FOLLOW set**

⇒ The FOLLOW set of a non-terminal in a grammar consists of the terminal symbols that can appear immediately to the right of that non-terminal in some sentimental form.

- Algorithm:

1) Place $ in FOLLOW(S), where S is the start symbol.
2) IF there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ε is in FOLLOW(B)
3) IF there is production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ where First(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).
4) Repeat steps 2-3 until no more additions can be made.

- **Conclusion**:

In this experiment, we learnt about FIRST and FOLLOW of grammar, how to calculate it.

29/02/2024

Code:

```
#include <iostream>
#include <string.h>
#define max 20

using namespace std;

char prod[max][10];
char ter[10], nt[10];
char first[10][10], follow[10][10];
int eps[10];
int count_var = 0;

int findpos(char ch) {
  int n;
  for (n = 0; nt[n] != '\0'; n++)
    if (nt[n] == ch)
      break;
  if (nt[n] == '\0')
    return 1;
  return n;
}

int IsCap(char c) {
  if (c >= 'A' && c <= 'Z')
    return 1;
  return 0;
}

void add(char *arr, char c) {
  int i, flag = 0;
  for (i = 0; arr[i] != '\0'; i++) {
    if (arr[i] == c) {
      flag = 1;
      break;
    }
  }
  if (flag != 1)
    arr[strlen(arr)] = c;
}

void addarr(char *s1, char *s2) {
```

```
  int i, j, flag = 99;
  for (i = 0; s2[i] != '\0'; i++) {
    flag = 0;
    for (j = 0;; j++) {
      if (s2[i] == s1[j]) {
        flag = 1;
        break;
      }
      if (j == strlen(s1) && flag != 1) {
        s1[strlen(s1)] = s2[i];
        break;
      }
    }
  }
}

void addprod(char *s) {
  int i;
  prod[count_var][0] = s[0];
  for (i = 3; s[i] != '\0'; i++) {
    if (!IsCap(s[i]))
      add(ter, s[i]);
    prod[count_var][i - 2] = s[i];
  }
  prod[count_var][i - 2] = '\0';
  add(nt, s[0]);
  count_var++;
}

void findfirst() {
  int i, j, n, k, e, n1;
  for (i = 0; i < count_var; i++) {
    for (j = 0; j < count_var; j++) {
      n = findpos(prod[j][0]);
      if (prod[j][1] == (char)238)
        eps[n] = 1;
      else {
        for (k = 1, e = 1; prod[j][k] != '\0' && e == 1; k++) {
          if (!IsCap(prod[j][k])) {
            e = 0;
            add(first[n], prod[j][k]);
          } else {
            n1 = findpos(prod[j][k]);
```

```
            addarr(first[n], first[n1]);
            if (eps[n1] == 0)
              e = 0;
          }
        }
        if (e == 1)
          eps[n] = 1;
      }
    }
  }
}

void findfollow() {
  int i, j, k, n, e, n1;
  n = findpos(prod[0][0]);
  add(follow[n], '#');
  for (i = 0; i < count_var; i++) {
    for (j = 0; j < count_var; j++) {
      k = strlen(prod[j]) - 1;
      for (; k > 0; k--) {
        if (IsCap(prod[j][k])) {
          n = findpos(prod[j][k]);
          if (prod[j][k + 1] == '\0') {
            n1 = findpos(prod[j][0]);
            addarr(follow[n], follow[n1]);
          }
          if (IsCap(prod[j][k + 1])) {
            n1 = findpos(prod[j][k + 1]);
            addarr(follow[n], first[n1]);
            if (eps[n1] == 1) {
              n1 = findpos(prod[j][0]);
              addarr(follow[n], follow[n1]);
            }
          } else if (prod[j][k + 1] != '\0')
            add(follow[n], prod[j][k + 1]);
        }
      }
    }
  }
}

int main() {
  char s[max], i;
```

```
  cout << "Enter the productions\n";
  cin >> s;
  while (strcmp("end", s)) {
    addprod(s);
    cin >> s;
  }
  findfirst();
  findfollow();

  // Displaying the title labels
  cout << "Production\tFirst\tFollow\n";

  for (i = 0; i < strlen(nt); i++) {
    cout << nt[i] << "\t\t\t" << first[i];
    if (eps[i] == 1)
      cout << ((char)238) << "\t\t";
    else
      cout << "\t\t";
    cout << follow[i] << "\n";
  }
  return 0;
}
```

Output:

```
/tmp/MO8WSvh02h.o
Enter the productions
E->TB
B->+TB
T->FC
C->*FC
F->(E)
F->i
B->
C->
end
Production  First   Follow
E           (i      #)
B           +       #)
T           (i      +#)
C           *       +#)
F           (i      *+#)
```

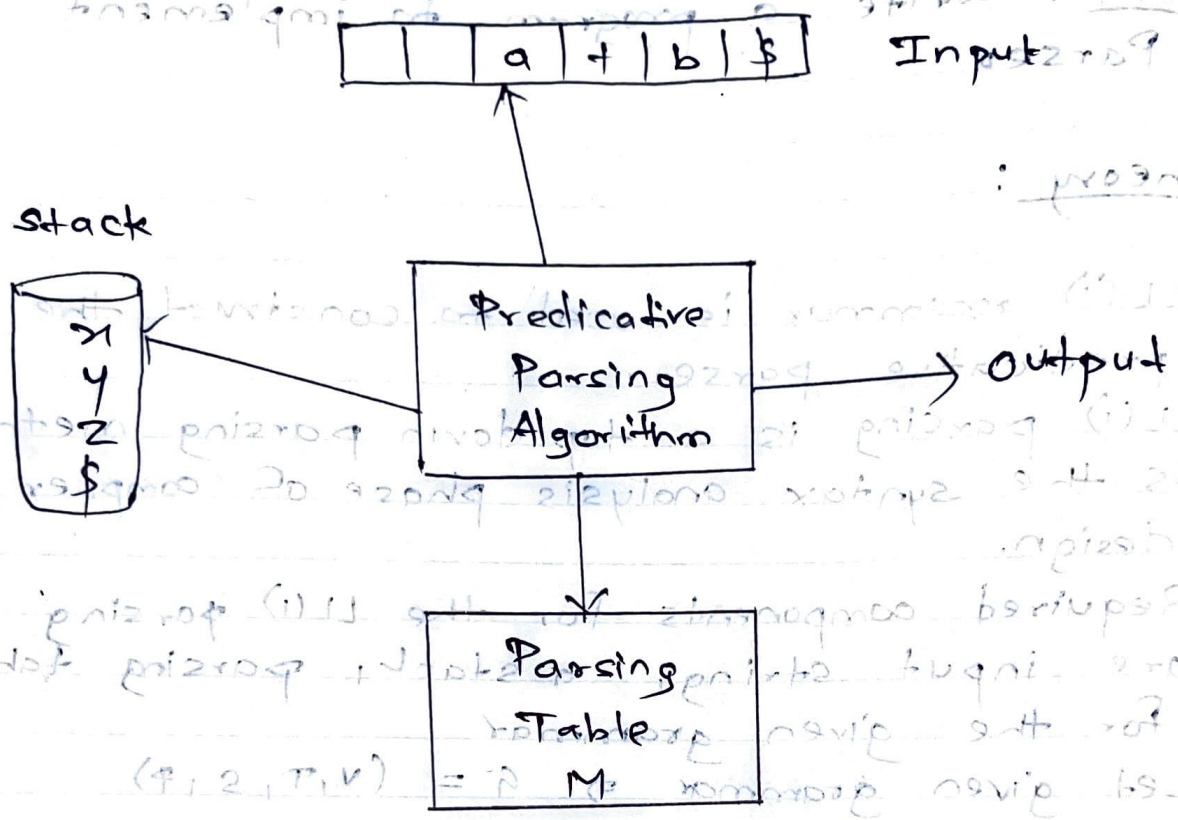
Experiment No: 4

Aim: Write a program to implement Parser.

Theory:

- LL(1) grammar is used to construct the predicative parser.
- LL(1) parsing is a top down parsing method is the syntax analysis phase of compiler design.
- Required components for the LL(1) parsing are input string, a stack, parsing table for the given grammar
- Let given grammar => $G = (V, T, S, P)$ where,
  - V = set of variables
  - T = terminal symbols set
  - S = start symbol
  - P = production set.
- Non-recursive predictive parser is also known as LL(1) parser.
- The LL(1) parser stands for :
  - "L" : left to right scan of input
  - "L" : Leftmost derivation
  - An LL parser is called an LL(K) parser if uses.
  - "K" : It uses k tokens of look ahead when parsing sentence.

Model of Non-Recursive Predicative Parser:



THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

- Algorithm :

Input : Grammar G

Output : Parsing Table M

Step 1 : For each terminal 'a' in FIRST(α)
ADD A → α to M[A, a]

Step 2 :

CASE 1

⇒ If ε is in FIRST(α) then
for each terminal b in FOLLOW(A)
ADD A → α to M[A, b]

CASE 2 :

⇒ If ε is in FIRST(α) and $ in FOLLOW(A)
then for each terminal b in FOLLOW(A)
Add A → α to M[A, b]

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

- Conclusion : In this we have implemented LL(1) parser algorithm successfully.

28/03/2024

For e.g.,

$E \to TE'$, $E' \to +TE' \mid \varepsilon$, $T \to FT'$,

$T' \to *FT' \mid \varepsilon$, $F \to (E) \mid id$

$\Rightarrow$

	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$			$F \to (E)$		

Code:

```
from collections import OrderedDict

def isterminal(char):
  if (char.isupper() or char == "`"):
    return False
  else:
    return True

def insert(grammar, lhs, rhs):
  if (lhs in grammar and rhs not in grammar[lhs] and grammar[lhs] !=
"null"):
    grammar[lhs].append(rhs)
  elif (lhs not in grammar or grammar[lhs] == "null"):
    grammar[lhs] = [rhs]
  return grammar

def first(lhs, grammar, grammar_first):
  rhs = grammar[lhs]
  for i in rhs:
    k = 0
    flag = 0
    current = []
    confirm = 0
    flog = 0
    if (lhs in grammar and "`" in grammar_first[lhs]):
      flog = 1
    while (1):
      check = []
      if (k >= len(i)):
        if (len(current) == 0 or flag == 1 or confirm == k or flog ==
1):
          grammar_first = insert(grammar_first, lhs, "`")
        break
      if (i[k].isupper()):
        if (grammar_first[i[k]] == "null"):
          grammar_first = first(i[k], grammar, grammar_first)
        for j in grammar_first[i[k]]:
          grammar_first = insert(grammar_first, lhs, j)
          check.append(j)
      else:
        grammar_first = insert(grammar_first, lhs, i[k])
```

```
        check.append(i[k])
      if (i[k] == "`"):
        flag = 1
      current.extend(check)
      if ("`" not in check):
        if (flog == 1):
          grammar_first = insert(grammar_first, lhs, "`")
        break
      else:
        confirm += 1
        k += 1
        grammar_first[lhs].remove("`")
  return (grammar_first)

def rec_follow(k, next_i, grammar_follow, i, grammar, start,
grammar_first,
               lhs):
  if (len(k) == next_i):
    if (grammar_follow[i] == "null"):
      grammar_follow = follow(i, grammar, grammar_follow, start)
    for q in grammar_follow[i]:
      grammar_follow = insert(grammar_follow, lhs, q)
  else:
    if (k[next_i].isupper()):
      for q in grammar_first[k[next_i]]:
        if (q == "`"):
          grammar_follow = rec_follow(k, next_i + 1, grammar_follow, i,
                                      grammar, start, grammar_first,
lhs)
        else:
          grammar_follow = insert(grammar_follow, lhs, q)
    else:
      grammar_follow = insert(grammar_follow, lhs, k[next_i])

  return (grammar_follow)

def follow(lhs, grammar, grammar_follow, start):
  for i in grammar:
    j = grammar[i]
    for k in j:
      if (lhs in k):
        next_i = k.index(lhs) + 1
```

```
        grammar_follow = rec_follow(k, next_i, grammar_follow, i,
grammar,
                                    start, grammar_first, lhs)
  if (lhs == start):
    grammar_follow = insert(grammar_follow, lhs, "$")
  return (grammar_follow)

def show_dict(dictionary):
  for key in dictionary.keys():
    print(key + "  :  ", end="")
    for item in dictionary[key]:
      if (item == "`"):
        print("Epsilon, ", end="")
      else:
        print(item + ", ", end="")
    print("\b\b")

def get_rule(non_terminal, terminal, grammar, grammar_first):
  for rhs in grammar[non_terminal]:
    #print(rhs)
    for rule in rhs:
      if (rule == terminal):
        string = non_terminal + "~" + rhs
        return string

      elif (rule.isupper() and terminal in grammar_first[rule]):
        string = non_terminal + "~" + rhs
        return string


def generate_parse_table(terminals, non_terminals, grammar,
grammar_first,
                         grammar_follow):
  parse_table = [[""] * len(terminals) for i in
range(len(non_terminals))]

  for non_terminal in non_terminals:
    for terminal in terminals:
      if terminal in grammar_first[non_terminal]:
        rule = get_rule(non_terminal, terminal, grammar, grammar_first)
      elif ("`" in grammar_first[non_terminal]
            and terminal in grammar_follow[non_terminal]):
        rule = non_terminal + "~`"
```

```
      elif (terminal in grammar_follow[non_terminal]):
        rule = "Sync"

      else:
        rule = ""

      parse_table[non_terminals.index(non_terminal)][terminals.index(
          terminal)] = rule

  return (parse_table)


def display_parse_table(parse_table, terminal, non_terminal):
  print("\t\t\t\t", end="")
  for terminal in terminals:
    print(terminal + "\t\t", end="")
  print("\n\n")

  for non_terminal in non_terminals:
    print("\t\t" + non_terminal + "\t\t", end="")
    for terminal in terminals:

print(parse_table[non_terminals.index(non_terminal)][terminals.index(
          terminal)] + "\t\t",
            end="")
    print("\n")

def parse(expr, parse_table, terminals, non_terminals):
  stack = ["$"]
  stack.insert(0, non_terminals[0])

  print("\t\t\tMatched\t\t\tStack\t\t\tInput\t\t\tAction\n")
  print("\t\t\t-\t\t\t", end="")
  for i in stack:
    print(i, end="")
  print("\t\t\t", end="")
  print(expr + "\t\t\t", end="")
  print("-")

  matched = "-"
  while (True):
    action = "-"
```

```
    if (stack[0] == expr[0] and stack[0] == "$"):
      break

    elif (stack[0] == expr[0]):
      if (matched == "-"):
        matched = expr[0]
      else:
        matched = matched + expr[0]
      action = "Matched " + expr[0]
      expr = expr[1:]
      stack.pop(0)

    else:
      action =
parse_table[non_terminals.index(stack[0])][terminals.index(
          expr[0])]
      stack.pop(0)
      i = 0
      for item in action[2:]:
        if (item != "`"):
          stack.insert(i, item)
        i += 1

    print("\t\t\t" + matched + "\t\t\t", end="")
    for i in stack:
      print(i, end="")
    print("\t\t\t", end="")
    print(expr + "\t\t\t", end="")
    print(action)

grammar = OrderedDict()
grammar_first = OrderedDict()
grammar_follow = OrderedDict()

f = open('grammar.txt')
for i in f:
  i = i.replace("\n", "")
  lhs = ""
  rhs = ""
  flag = 1
  for j in i:
    if (j == "~"):
```

```
      flag = (flag + 1) % 2
      continue
    if (flag == 1):
      lhs += j
    else:
      rhs += j
  grammar = insert(grammar, lhs, rhs)
  grammar_first[lhs] = "null"
  grammar_follow[lhs] = "null"

print("Grammar\n")
show_dict(grammar)

for lhs in grammar:
  if (grammar_first[lhs] == "null"):
    grammar_first = first(lhs, grammar, grammar_first)

print("\n")
print("First\n")
show_dict(grammar_first)

start = list(grammar.keys())[0]
for lhs in grammar:
  if (grammar_follow[lhs] == "null"):
    grammar_follow = follow(lhs, grammar, grammar_follow, start)

print("\n")
print("Follow\n")
show_dict(grammar_follow)

non_terminals = list(grammar.keys())
terminals = []

for i in grammar:
  for rule in grammar[i]:
    for char in rule:

      if (isterminal(char) and char not in terminals):
        terminals.append(char)

terminals.append("$")

print("Parse Table\n\n")
```

```
parse_table = generate_parse_table(terminals, non_terminals, grammar,
                                   grammar_first, grammar_follow)
display_parse_table(parse_table, terminals, non_terminals)

expr = "i+i*i$"

print("\n\n")
print("Parsing Expression\n\n")
parse(expr, parse_table, terminals, non_terminals)
```

Output:

```
Grammar

E  :  TL,
L  :  +TL, Epsilon,
T  :  FK,
K  :  *FK, Epsilon,
F  :  i, (E),


First

E  :  i, (,
L  :  +, Epsilon,
T  :  i, (,
K  :  *, Epsilon,
F  :  i, (,


Follow

E  :  ), $,
L  :  ), $,
T  :  +, ), $,
K  :  +, ), $,
F  :  *, +, ), $,
```

Parse Table

	+	*	i	(	)	$
E			E~TL	E~TL	Sync	Sync
L	L~+TL				L~`	L~`
T	Sync		T~FK	T~FK	Sync	Sync
K	K~`	K~*FK			K~`	K~`
F	Sync	Sync	F~i	F~(E)	Sync	Sync

Parsing Expression

Matched	Stack	Input	Action
-	E$	i+i*i$	-
-	TL$	i+i*i$	E~TL
-	FKL$	i+i*i$	T~FK
-	iKL$	i+i*i$	F~i
i	KL$	+i*i$	Matched i
i	L$	+i*i$	K~`
i	+TL$	+i*i$	L~+TL
i+	TL$	i*i$	Matched +
i+	FKL$	i*i$	T~FK
i+	iKL$	i*i$	F~i
i+i	KL$	*i$	Matched i
i+i	*FKL$	*i$	K~*FK
i+i*	FKL$	i$	Matched *
i+i*	iKL$	i$	F~i
i+i*i	KL$	$	Matched i
i+i*i	L$	$	K~`
i+i*i	$	$	L~`

Om Shete
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
TE C32
2103163

Experiment No : 5

Aim : WAP to implement Three Address Code

Theory :

- Three address code (TAC) is an intermediate type of code which is easy to generate and can be easily converted to machine code.
- It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler.
- The compiler decides the order of operation given by three address code.
- Three address code is used in compiler applications :-

1) **Optimization**
⇒ TAC is often used as an intermediate representation of code during optimization phases of compilation process.

2) **Code generation**
⇒ TAC code can also be used as an intermediate representation of code during the code generation phase of the compilation process.

3) **Debugging**
⇒ TAC can be helpful in debugging the code

1) Quadraple Field

⇒

operator
source 1
source 2
Destination

$p = -q * r + s$

$t1 = -q$

$t2 = r + s$

$t3 = t1 * t2$

$p = t3$

2) Triple Field

⇒

Operator
Source 1
Source 2

$p = -q * r + s$

$t1 = -q$

$t2 = r + s$

generated by the compiler. Since three address code is low-level language, it is often easier to read and understand the final generated code.

4) Language translation

→ TAC can also be used to translate code from one programming language to another.

- Three representation technique :-
  - → Quadruple
  - → Triples
  - → Indirect tripler

- General Illustration :

  a = b op c

  a, b, c → operands

  op → operator

- Conclusion : In this program I learned about the TAC and how to implement it.

28/03/2024

Code:

```
import re


print("enter your choice 1 for assignment 2 for arithmetic 3 for
relational 4 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
    elif choice == 3:
        print("enter the relational expression")
```

```
        exp = input().split(" ")
        operators = ["<", ">", "<=", ">=", "==", "!="]
        if exp[1] in operators:
            print("100 IF ", exp[0], exp[1], exp[2], " GOTO 103")
            print("101 T:=0")
            print("102 GOTO 104")
            print("103 T:=1")
            print("104")
    else:
        print("invalid operator")
    print(
        "enter your choice 1 for assignment 2 for arithmetic 3 for
relational 4 to exit"
    )
    choice = int(input())
```

Output:

```
Run                                          Ask AI   41s on 00:46:36, 03/27
enter your choice 1 for assignment 2 for arithmetic 3 for relational 4 to exit
1
enter the variable
a=d
['a', 'd']
t1 =  d
a = t1
enter your choice 1 for assignment 2 for arithmetic 3 for relational 4 to exit
2
enter the expression
a=b+c*d
t1 =  c * d
t2 =  b + t1
enter your choice 1 for assignment 2 for arithmetic 3 for relational 4 to exit
3
enter the relational expression
a > b
100 IF  a > b  GOTO 103
101 T:=0
102 GOTO 104
103 T:=1
104
enter your choice 1 for assignment 2 for arithmetic 3 for relational 4 to exit
4
```

Om Shete
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
TE C32
2103163

Experiment No : 8

Aim : Write a program to implement Code Generation Optimization.

Theory :

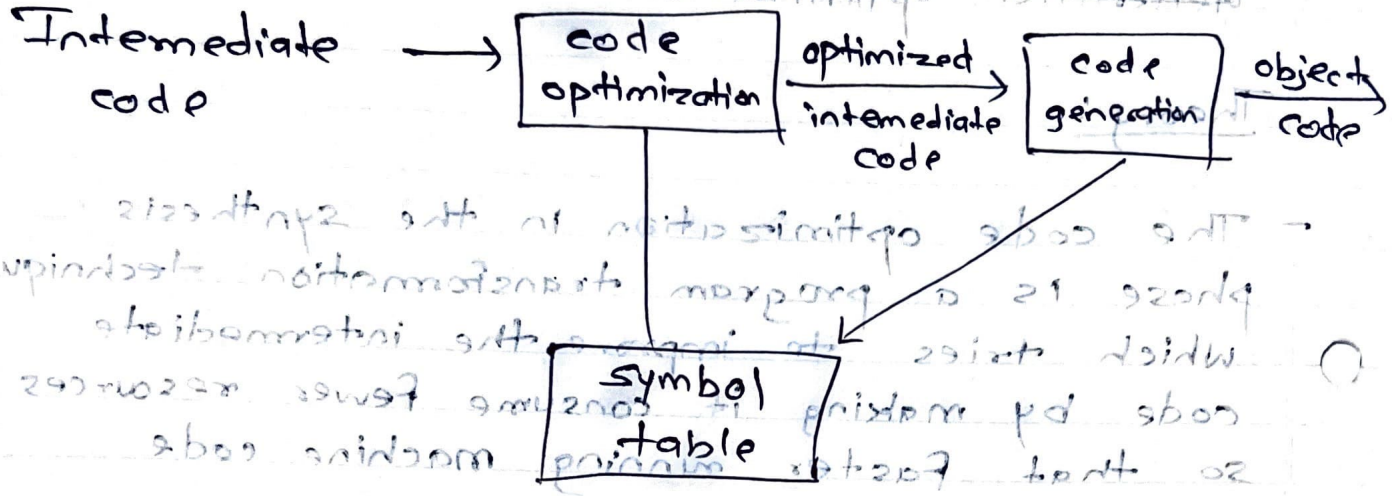
- The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources so that faster running machine code will result.
- Compiler optimizing process should meet the following objective:
  1) The optimization must be correct, it must not, in any way, change the meaning of the program.
  2) Optimization should increase the speed and the performance of program.
- Types of Code Optimization :-

1) Machine Independent Optimization

⇒ This code optimization phase attempts to improve the optimization phase at intemediate code to get better target code as output.

2) Machine dependent Optimization

⇒ Machine Dependent optimization is done after the target code has been generated and when the code is transformed according to


Intemediate code
code optimization
optimized intemediate code
code generation
object code
symbol table

target machine architecture.

- Ways to optimize code :-

i) Compile time evaluation

⇒

A = 2 * (22.0 / 7.0) * r

Perform 2 * (22.0 / 7.0) * r at compiler time.

ii) Variable propagation

⇒

```
c = a * b          |          c = a * b
x = a              |          x = a
till               | optimized  till
d = x * b + 4      |          d = a * b + 4
```

iii) Constant propagation

⇒ If variable is a constant, then replace variable constant.

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

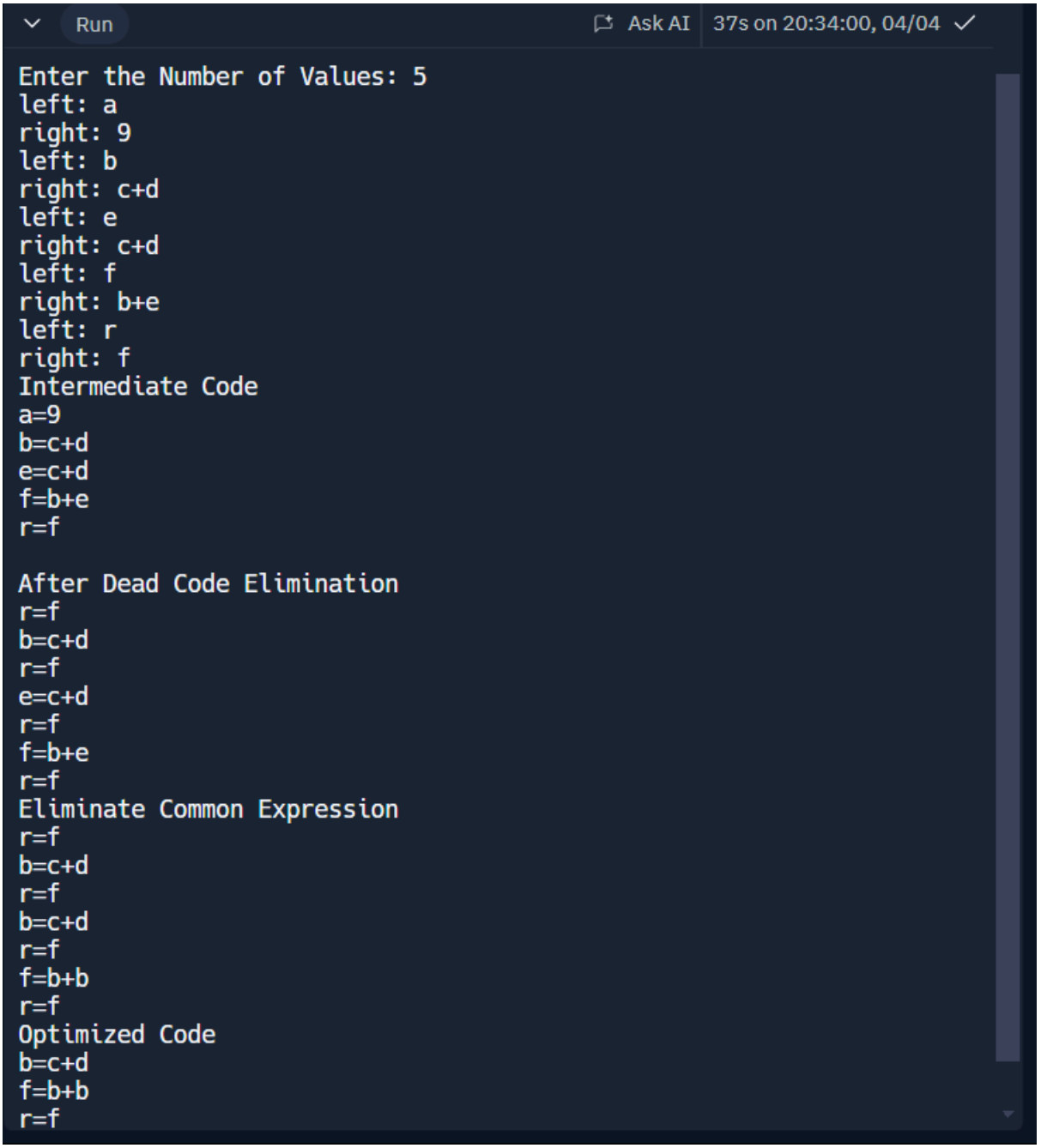
28/03/2024

Code:

```
class OP:

  def __init__(self, l, r):
    self.l = l
    self.r = r


op = []
pr = []


def main():
  n = int(input("Enter the Number of Values: "))
  for i in range(n):
    l = input("left: ")
    r = input("right: ")
    op.append(OP(l, r))

  print("Intermediate Code")
  for item in op:
    print(item.l + "=" + item.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.append(OP(op[i].l, op[i].r))
        z += 1
    pr.append(OP(op[n - 1].l, op[n - 1].r))
    z += 1

  print("\nAfter Dead Code Elimination")
  for item in pr:
    print(item.l + "=" + item.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[:l] + pr[m].l + pr[i].r[l + 1:]

  print("Eliminate Common Expression")
  for item in pr:
    print(item.l + "=" + item.r)

  for i in range(z):
    for j in range(i + 1, z):
      if pr[i].l == pr[j].l and pr[i].r == pr[j].r:
        pr[i].l = '\0'

  print("Optimized Code")
  for item in pr:
    if item.l != '\0':
      print(item.l + "=" + item.r)


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

Output:


Run
Ask AI
37s on 20:34:00, 04/04
Enter the Number of Values: 5
left: a
right: 9
left: b
right: c+d
left: e
right: c+d
left: f
right: b+e
left: r
right: f
Intermediate Code
a=9
b=c+d
e=c+d
f=b+e
r=f
After Dead Code Elimination
r=f
b=c+d
r=f
e=c+d
r=f
f=b+e
r=f
Eliminate Common Expression
r=f
b=c+d
r=f
b=c+d
r=f
f=b+b
r=f
Optimized Code
b=c+d
f=b+b
r=f

Om Shete
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
TE C32
2103163

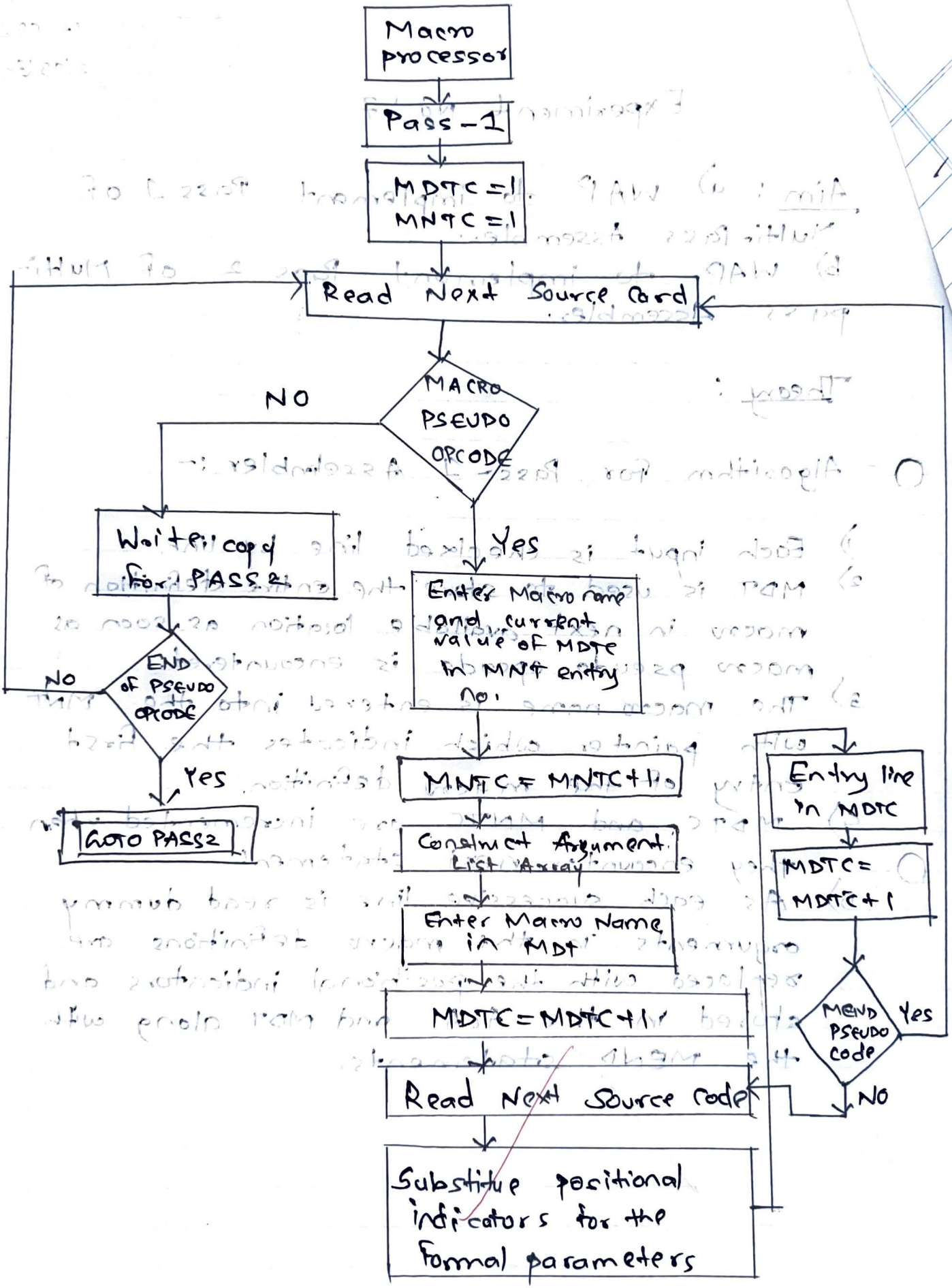
Experiment No :7

Aim : a) WAP to implemant Pass 1 of Multi-Pass Assembler.
b) WAP to implement Pass 2 of Multi-pass Assembler.

Theory :

- Algorithm for Pass-1 Assembler :-

1) Each input is checked line by line.
2) MDT is used to store the entire definition of macro in next available location as soon as macro pseudo opcode is encountered.
3) The macro name is entered into the MNT with pointer which indicates the first entry of the macro definition.
4) MDTC and MNTC are incremented when they encounter next statement.
5) As each successive line is read dummy arguments in the macro definitions are replaced with the positional indicators and stored in the ALA and MDT along with the MEND statements.


Macro processor
Pass-1
MDTC=1
MNTC=1
Read Next Source Card
MACRO
PSEUDO
OPCODE
NO
Yes
Write copy for PASS2
Enter Macro name and current value of MDTC in MNT entry no.
END of PSEUDO OPCODE
NO
Yes
GOTO PASS2
MNTC = MNTC+1
Construct Argument List Array
Enter Macro Name in MDT
MDTC=MDTC+1
Read Next Source code
Substitute positional indicators for the formal parameters
Entry line in MDTC
MDTC = MDTC+1
MEND PSEUDO code
Yes
NO

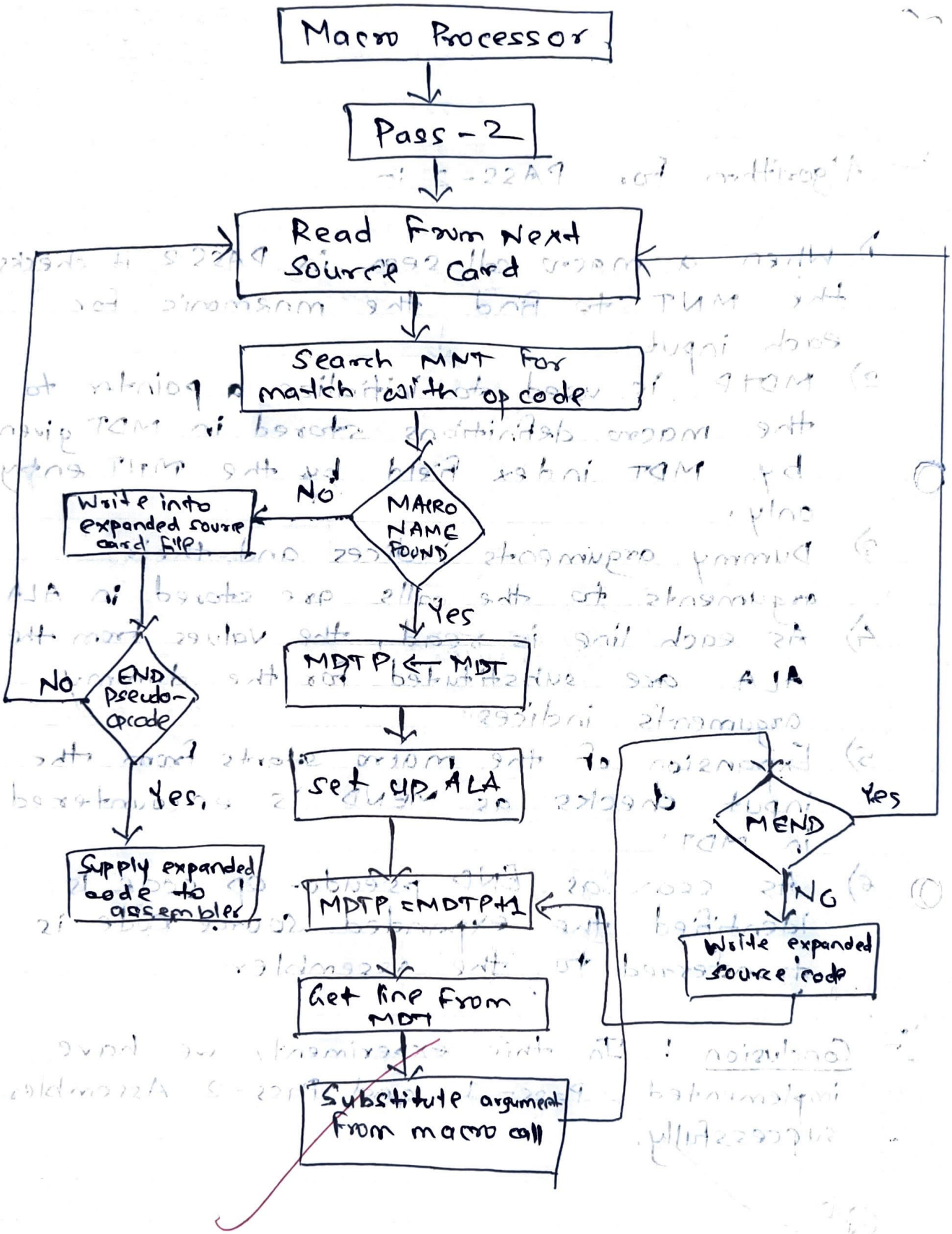
THADOMAL SHAHANI
TSEC
ENGINEERING COLLEGE

- Algorithm for PASS-2 :-

1) When a macro call seen in PASS 2 it checks the MNT to find the mnemonic for each input.
2) MDTP is used to initialize a pointer to the macro definitions stored in MDT given by MDT index field by the MNT entry only.
3) Dummy arguments indices and their arguments to the calls are stored in ALA
4) As each line is read, the values from the ALA are substituted for the dummy arguments indices.
5) Expansion of the macro starts from the input checks as MEND is encountered in MDT.
6) As soon as END pseudo-op code is identified the expanded source code is transferred to the assembler.

- Conclusion : In this experiment, we have implemented Pass-1 and Pass-2 Assembler successfully.

10/04/2024.


Macro Processor
Pass - 2
Read From Next Source Card
Search MNT for match with opcode
MACRO NAME FOUND
No
Yes
Write into expanded source card file
MDTP ← MDT
END pseudo-opcode
NO
Yes
set up ALA
Supply expanded code to assembler
MDTP = MDTP+1
Get line from MDT
Substitute argument from macro call
MEND
Yes
NO
Write expanded source code

Code:

```
#include <fstream>
#include <iostream>
#include <map>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

map<string, pair<int, vector<string>>> MNT; // Macro Name Table
map<int, pair<string, vector<string>>> MXT; // Macro Expansion Table
map<string, string> ALA;                    // Argument List Array

void passOne(ifstream &inputFile);
void passTwo(ifstream &inputFile, ofstream &outputFile);

int main() {
  ifstream inputFile("input.txt");

  if (!inputFile) {
    cerr << "Error: Could not open input file." << endl;
    return 1;
  }

  passOne(inputFile);
  inputFile.close();

  cout << "MNT after pass one:" << endl;
  for (const auto &entry : MNT) {
    cout << entry.first << " -> ";
    cout << "Index: " << entry.second.first << ", ";
    cout << "Args: ";
    for (const auto &arg : entry.second.second) {
      cout << arg << " ";
    }
    cout << endl;
  }
  cout << endl;

  ofstream outputFile("output.txt");
  if (!outputFile) {
```

```
    cerr << "Error: Could not create output file." << endl;
    return 1;
  }

  inputFile.open("input.txt");
  passTwo(inputFile, outputFile); // Second pass
  inputFile.close();
  outputFile.close();

  cout << "MXT after pass two:" << endl;
  for (const auto &entry : MXT) {
    cout << entry.first << " -> ";
    cout << "Macro: " << entry.second.first << ", ";
    cout << "Args: ";
    for (const auto &arg : entry.second.second) {
      cout << arg << " ";
    }
    cout << endl;
  }
  cout << endl;

  cout << "ALA after pass two:" << endl;
  for (const auto &entry : ALA) {
    cout << entry.first << " -> " << entry.second << endl;
  }

  cout << "Assembly process complete. Output written to output.txt" <<
endl;

  return 0;
}

void passOne(ifstream &inputFile) {
  string line;
  int macroIndex = 0;

  while (getline(inputFile, line)) {
    stringstream ss(line);
    string token;
    ss >> token;

    if (token == "MEND") {
      continue;
```

```
    } else if (token == "&LAB") {
      string macroName;
      ss >> macroName;

      MNT[macroName].first = macroIndex++;

      vector<string> arguments;
      while (ss >> token) {
        if (token != ",")
          arguments.push_back(token);
      }

      MNT[macroName].second = arguments;
    }
  }

  cout << "MNT after pass one:" << endl;
  for (const auto &entry : MNT) {
    cout << entry.first << " -> ";
    cout << "Index: " << entry.second.first << ", ";
    cout << "Args: ";
    for (const auto &arg : entry.second.second) {
      cout << arg << " ";
    }
    cout << endl;
  }
  cout << endl;
}

void passTwo(ifstream &inputFile, ofstream &outputFile) {
  string line;
  int macroIndex = 0;

  while (getline(inputFile, line)) {
    stringstream ss(line);
    string token;

    while (ss >> token) {
      if (token == "MEND") {
        continue;
      } else if (token == "&LAB") {
        string macroName;
        ss >> macroName;
```

```
    vector<string> arguments;
    while (ss >> token) {
      if (token != ",")
        arguments.push_back(token);
    }

    MXT[macroIndex].first = macroName;
    MXT[macroIndex].second = arguments;

    macroIndex++;
  } else {

    if (MNT.find(token) != MNT.end()) {

      outputFile << "loop " << macroIndex + 1 << " "
                 << MXT[macroIndex].first << endl;

      int argNum = 1;
      for (const auto &arg : MNT[token].second) {
        if (arg[0] == '%') {

          string argValue = ALA[arg];
          outputFile << "  " << argNum << " " << argValue << " ";
        } else {
          outputFile << "  " << argNum << " " << arg << " ";
        }

        ss >> token;
        outputFile << " " << argNum << ", " << token << endl;
        argNum++;
      }
    } else {

      string macroName = MXT[macroIndex].first;
      vector<string> args = MXT[macroIndex].second;
      for (size_t i = 0; i < args.size(); ++i) {
        ALA[args[i]] = token;

        ss >> token;
      }
    }
  }
```

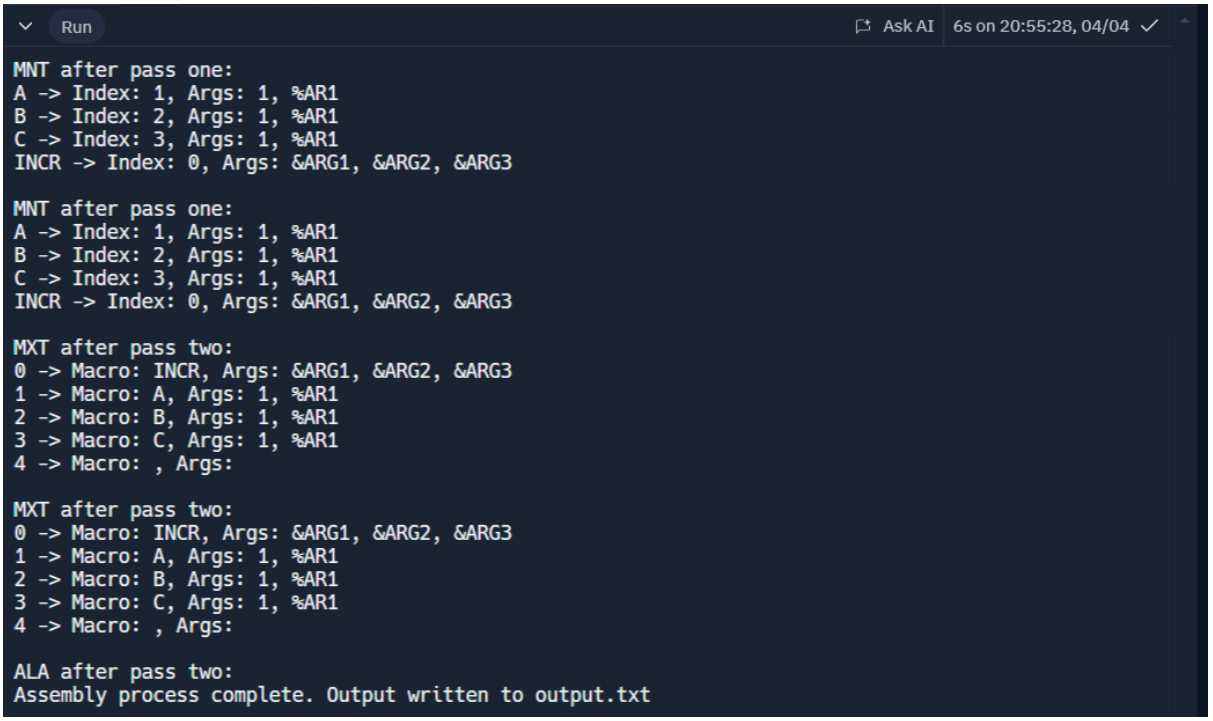
```
    }
  }

  cout << "MXT after pass two:" << endl;
  for (const auto &entry : MXT) {
    cout << entry.first << " -> ";
    cout << "Macro: " << entry.second.first << ", ";
    cout << "Args: ";
    for (const auto &arg : entry.second.second) {
      cout << arg << " ";
    }
    cout << endl;
  }
  cout << endl;

  // cout << "ALA after pass two:" << endl;
  // for (const auto& entry : ALA) {
  //     cout << entry.first << " -> " << entry.second << endl;
  // }
}
```

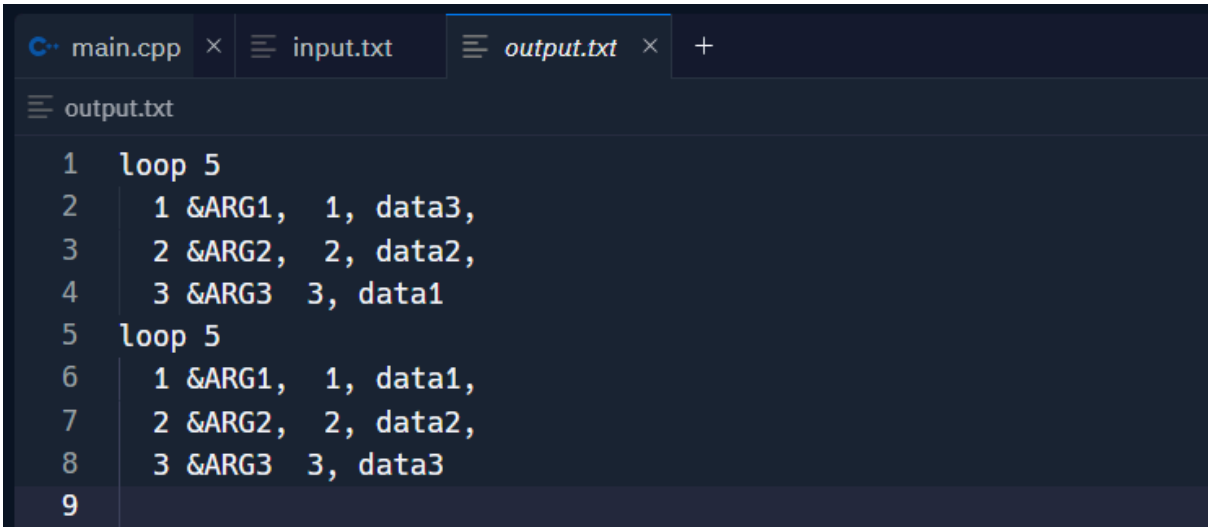
main.cpp × input.txt × output.txt +

input.txt

```
1  &LAB INCR &ARG1, &ARG2, &ARG3
2  &LAB A 1, %AR1
3  &LAB B 1, %AR1
4  &LAB C 1, %AR1
5  MEND
6
7  Loop1 INCR data3, data2, data1
8  Loop2 INCR data1, data2, data3
```

Output:


Run
Ask AI
6s on 20:55:28, 04/04
MNT after pass one:
A -> Index: 1, Args: 1, %AR1
B -> Index: 2, Args: 1, %AR1
C -> Index: 3, Args: 1, %AR1
INCR -> Index: 0, Args: &ARG1, &ARG2, &ARG3
MNT after pass one:
A -> Index: 1, Args: 1, %AR1
B -> Index: 2, Args: 1, %AR1
C -> Index: 3, Args: 1, %AR1
INCR -> Index: 0, Args: &ARG1, &ARG2, &ARG3
MXT after pass two:
0 -> Macro: INCR, Args: &ARG1, &ARG2, &ARG3
1 -> Macro: A, Args: 1, %AR1
2 -> Macro: B, Args: 1, %AR1
3 -> Macro: C, Args: 1, %AR1
4 -> Macro: , Args:
MXT after pass two:
0 -> Macro: INCR, Args: &ARG1, &ARG2, &ARG3
1 -> Macro: A, Args: 1, %AR1
2 -> Macro: B, Args: 1, %AR1
3 -> Macro: C, Args: 1, %AR1
4 -> Macro: , Args:
ALA after pass two:
Assembly process complete. Output written to output.txt


main.cpp
input.txt
output.txt
output.txt
1 loop 5
2 1 &ARG1, 1, data3,
3 2 &ARG2, 2, data2,
4 3 &ARG3 3, data1
5 loop 5
6 1 &ARG1, 1, data1,
7 2 &ARG2, 2, data2,
8 3 &ARG3 3, data3
9

Experiment No: 8

Aim: Write a program to implement Multi Pass Macroprocessor.

Theory:

- A multi-pass compiler is a type of compiler that processes the source code or abstract syntax tree of a program multiple time.
- In multi-pass compiler, we divide phases into two passes as :-

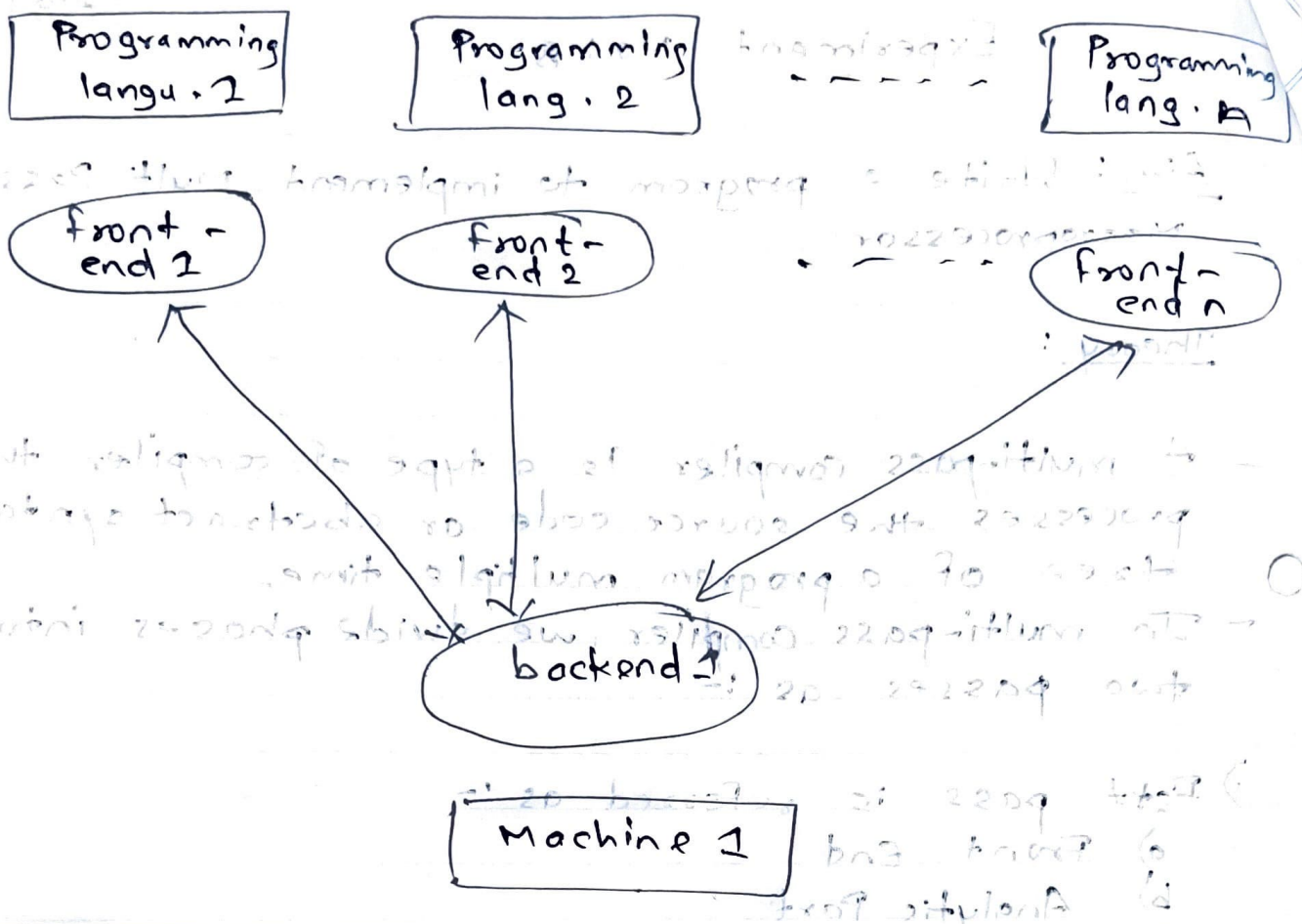
1) First pass is referred as :-
   a) Front End
   b) Analytic Part
   c) Platform Independent

2) Second pass is referred as :-
   a) Back End
   b) Synthesis Part
   c) Platform Dependent

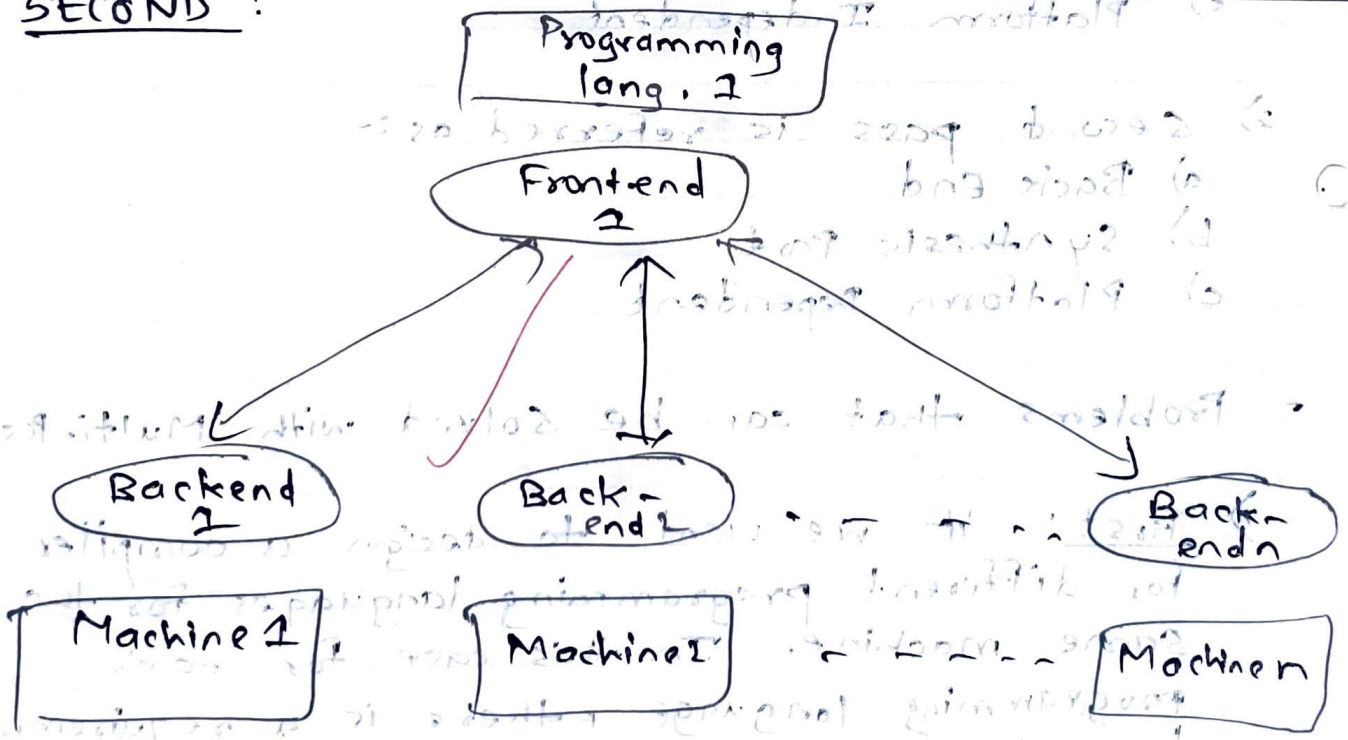
- Problems that can be solved with Multi-Pass:

1) **First:** If we want to design a compiler for different programming languages for the same machine. In this case for each programming language, there is a requirement to make the Front end for each of them and only Backend pass

FIRST :



SECOND :



2) Second : If we want to design a compiler for the same programming language for different machines / system. In this case, we make different Back End for different Machines and make only one Front End for same programming language.

- Conclusion : In conclusion, the choice between a single pass and a two-pass compiler depends on specific requirements and trade-offs. Multi-pass compilers offers greater flexibility for different programming languages and machines systems but comp at cost of additional processing.

10/04/2024.

Code:

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

class twopassmacro {
    static String mnt[][] = new String[5][3];

    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: All 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[mntc++][2] = (++mdtc) + "";

                            st = new StringTokenizer(str[1], ",");
                            for (i = 0; i < string.length; i++) {
                                string[i] = st.nextToken();
                                ala[alac][0] = alac + "";
                                if (index != -1)
                                    ala[alac++][1] =
string[i].substring(0, index);
                                else
                                    ala[alac++][1] = string[i];
                            }
                        } else {
                            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("Unable to find 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++) {
                            index = mdt[i][0].indexOf("#");
                            temp = mdt[i][0].substring(0, index);
                            temp += ala[Integer.parseInt("" +
mdt[i][0].charAt(index + 1))][1];
                            output.write(temp);
                            output.newLine();
```

```
                        }

                        flag = 1;
                    }
                }

                if (flag == 0) {
                    output.write(s);
                    output.newLine();
                }
            }

            output.close();
        } catch (FileNotFoundException ex) {
            System.out.println("Unable to find 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:

```
C:\Program Files (x86)\Java\jdk1.6.0_2\bin>java twopassmacro
Macro Name Table(MNT)
1  INCR1  1
2  INCR2  5
Argument List Array(ALA) for Pass1
0  &FIRST
1  &SECOND
2  &ARG1
3  &ARG2
Macro Definition 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 List Array(ALA) for Pass2
0  DATA1
1  DATA2
2  DATA3
3  DATA4
```

Om Shete
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
TE C32
2103163

Experiment No : 8

Aim : Write a program to eliminate left recursion from the given grammar.

Theory :

- A production of the 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 recursive because the left production in the form

$$A \longrightarrow A\alpha \mid \beta$$

- 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 a pair of production with

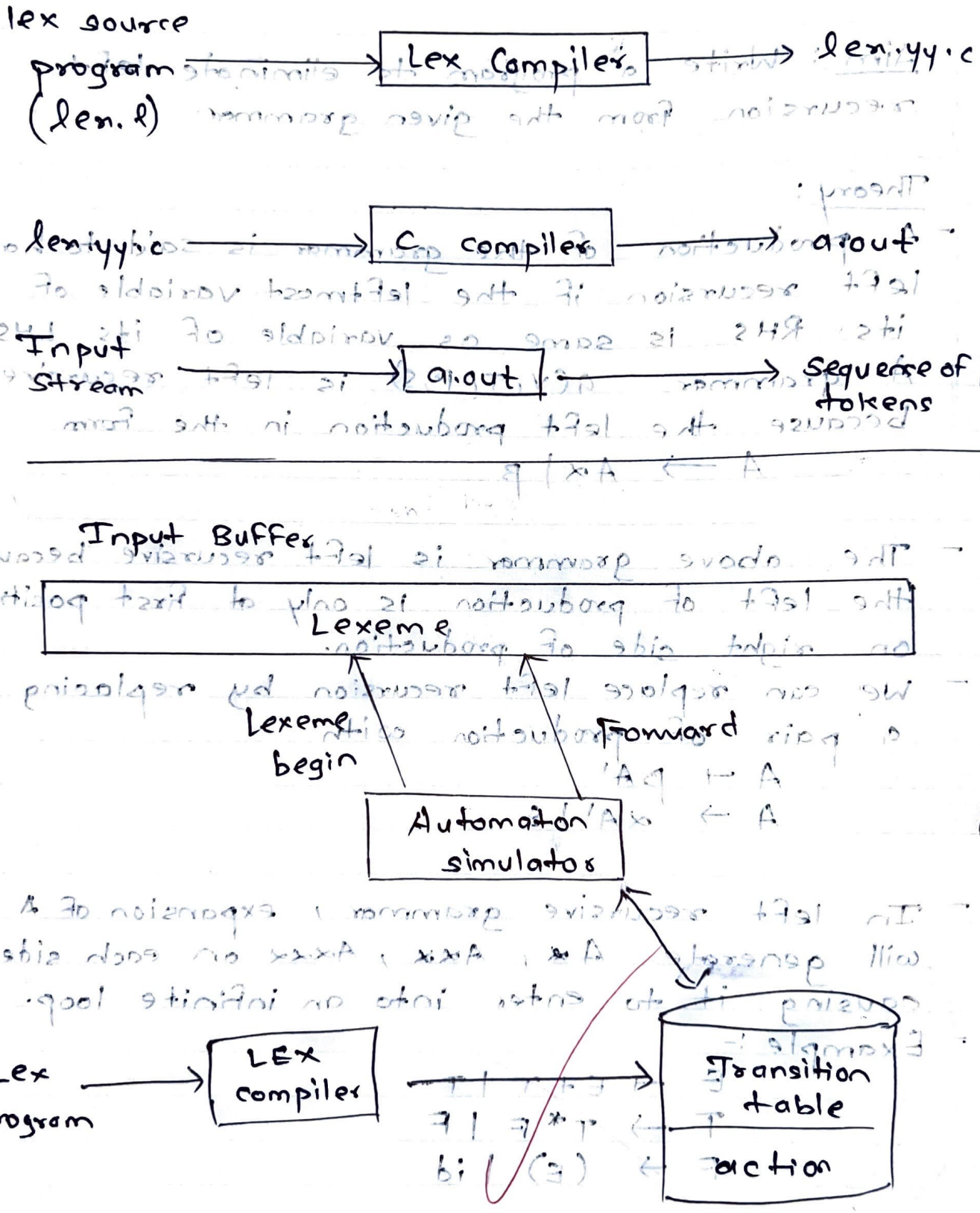
$$A \rightarrow \beta A'$$
$$A \rightarrow \alpha A' \mid \varepsilon$$

- In left recursive grammar, expansion of A will generate $A\alpha$, $A\alpha\alpha$, $A\alpha\alpha\alpha$ on each side, causing it to enter into an infinite loop.
- 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 $A \rightarrow A\alpha \mid \beta$


lex source program (lex.l)
Lex Compiler
lex.yy.c
lex.yy.c
C compiler
a.out
Input Stream
a.out
Sequence of tokens
Input Buffer
Lexeme
Lexeme begin
Forward
Automaton simulator
Lex program
LEX compiler
Transition table
action

$\therefore A = E \quad , \alpha = +T \quad , \beta = T$

$A \rightarrow A\alpha \mid \beta \longrightarrow$ change to $\rightarrow A' \rightarrow \alpha A' \mid \varepsilon$

$\therefore A \rightarrow \beta A'$ means $E \rightarrow TE'$

Compare $T \rightarrow T * F \mid F$

$A = T \quad , \alpha = +F \quad , \beta = F$

$A = BA'$ means $T' \rightarrow *FT' \mid \varepsilon$

Production $F \rightarrow (E) \mid id$ does not have any left recursion.

$\therefore E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

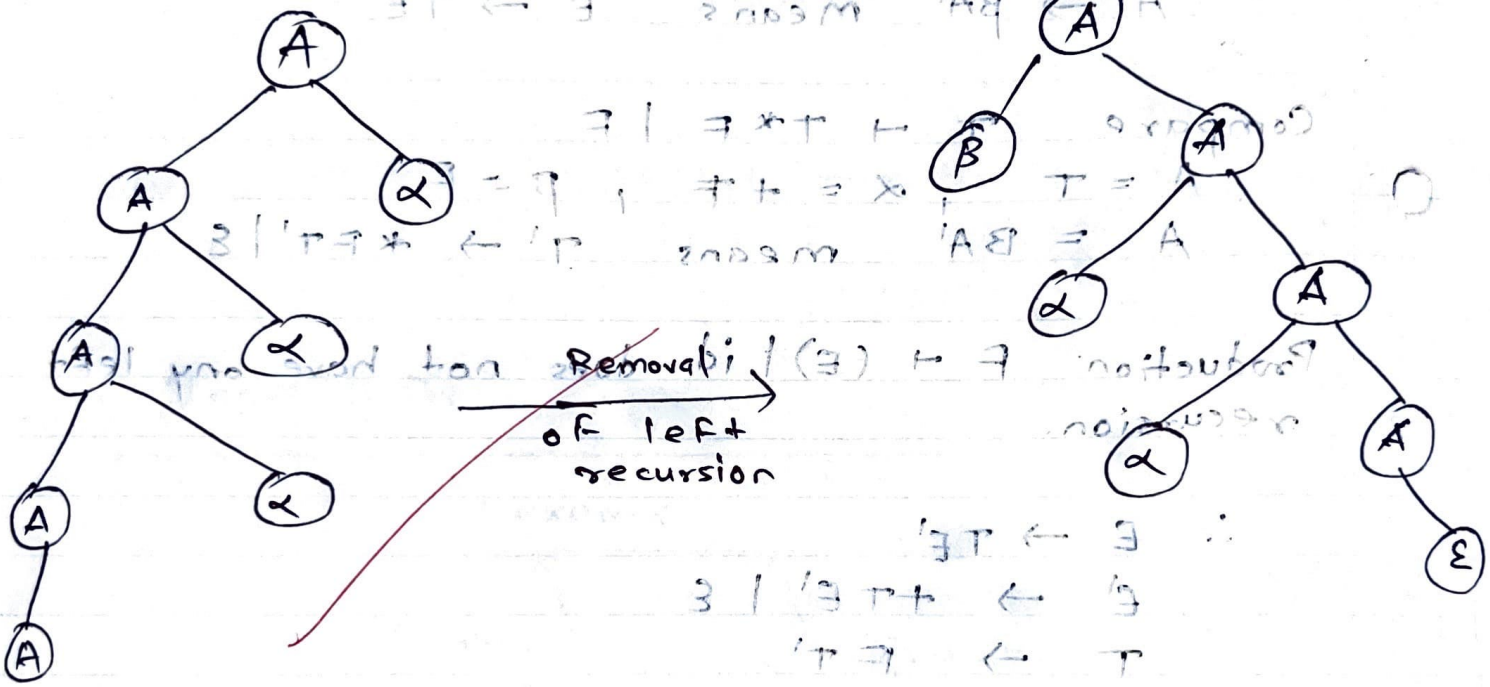
$T' \rightarrow +FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

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

VA
28/03/2024

$A \rightarrow A\alpha \mid \beta$ — Removal of left recursion → $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon$

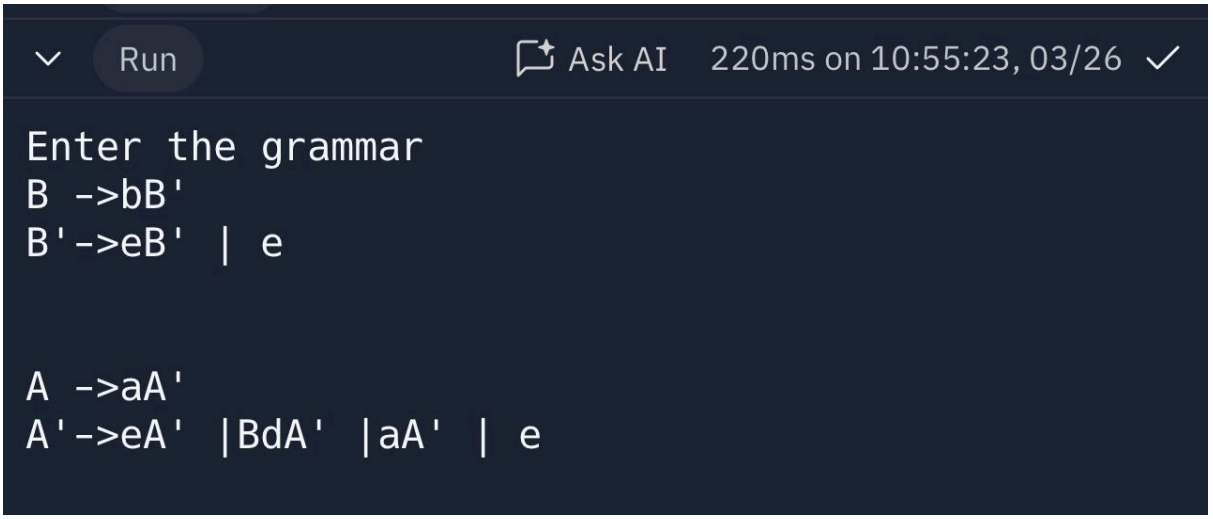


Conclusion: In this experiment, we learnt about left recursion and how to remove left recursion

Code:

```
import re

print("Enter the grammar")
gm = {"A->ABd|Aa|a", "B->Be|b"}
aplha = []
beta = []
for i in gm:
    # print(re.split(r'->|\|', i))
    exp = re.split(r"->|\|", i)
    nt = exp[0]
    cnt = 0
    for i in exp:
        if cnt == 0:
            cnt += 1
            continue
        else:
            if i[0] == exp[0]:
                aplha.append(i[1:])

            else:
                beta.append(i)
    # print('alpha',aplha)
    # print('beta',beta)
    # use left recursion
    print(exp[0], "->", end="")
    for i in beta:
        # print(i+exp[0]+'\'','|', end='')
        # if last beta dont add | else add |
        if i == beta[-1]:
            print(i + exp[0] + "'", end="")
        else:
            print(i + exp[0] + "'", "|", end="")

    print("\n", exp[0] + "'", "->", end="", sep="")
    for i in aplha:
        if i == aplha[-1]:
            print(i + exp[0] + "'", "|", "e", end="")
        else:
            print(i + exp[0] + "'", "|", end="")

    alpha = []
    beta = []
    print("\n\n")
```

Output:


Run
Ask AI
220ms on 10:55:23, 03/26
Enter the grammar
B ->bB'
B'->eB' | e
A ->aA'
A'->eA' |BdA' |aA' | e

Om Shete
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
TE C32
2103163

Experiment No : 10

Aim : WAA to implement code generation.

Theory :

- Code generation is need to produce the target code for three address statements. It uses register to store the operands of the additional statements.
- Consider 3 additional statements like x = y + z, It can have following sequence of codes.

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

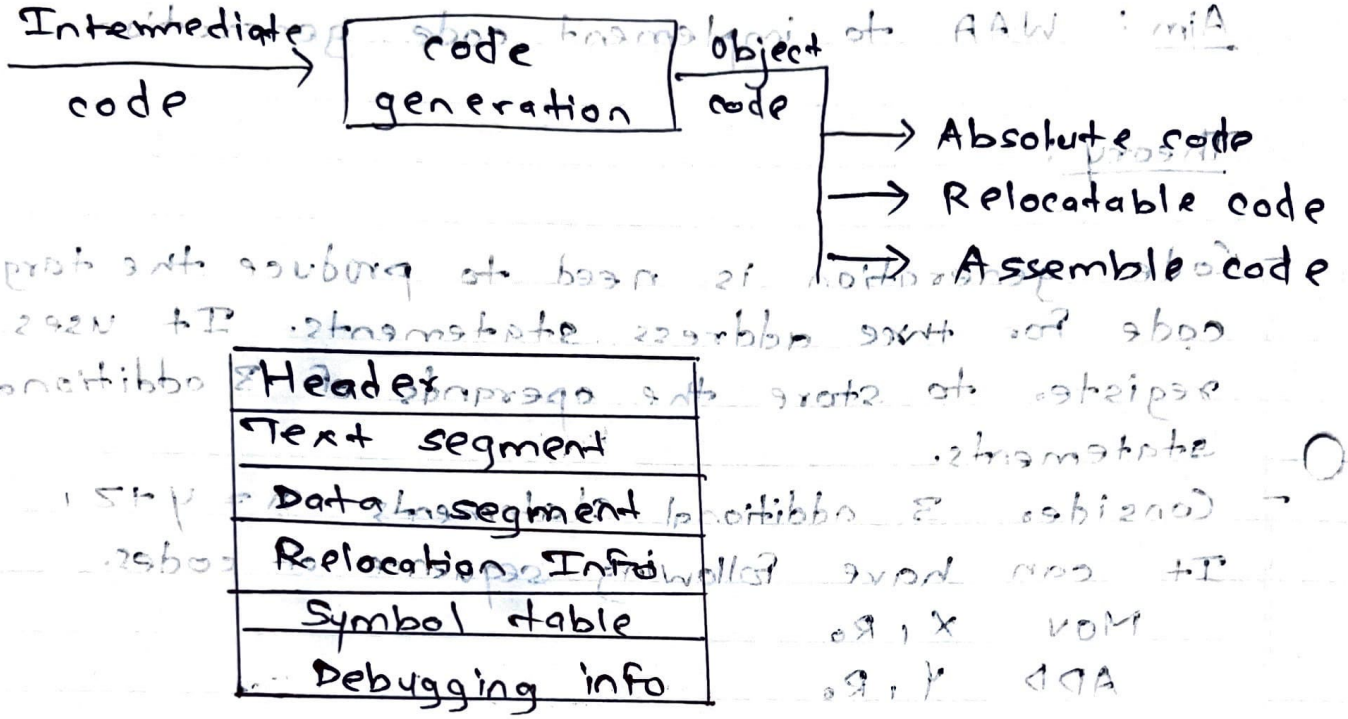
- Register description contains the track of what is in currently each register.
- An address descriptor is need to store the location where current value of name can be found at run time.

- **Code Generation Algorithm :**

The algorithm takes a sequence of three address statements as input. For each three address statement of the form a : b op c perform the various actions.

These actions are as follows :

1) Invoke a function getreg to find out the location where the result of computation b op c should be stored.



Statement	Code generated	Register Description	Address descriptor
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	~~Mov t, R2~~ ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R1
d = v + u	ADD R1, R0 Mov R0, d	R0 contained	d in R0 d in R0 d mem

2) Consult the address description for y to determine y'. If the value of y currently in memory and register both then prefer the register y'. If the value of y currently in memory and register both then prefer the register y'. If the value of y is not already in L there generates the information MOV y', L to place a copy of y in L.

3) Generate the information OP z', L where z' is used to show the current location of z. If z is in both then prefer a register to a memory location. Update the address descriptor to x to indicate that x is in location L. If x is in L then update its descriptor and remove x from all other descriptor.

4) If the current value of y have no next uses or not have an exit from block or in register then allows the register descriptor to indicate that after execution of x := y op z those register with no longer contains y on x.

- Conclusion: In this, I learned and implemented code generation and its algorithm.

QA
28/03/2024.

Code:

```
op1, op2, op3, op4 = "", "", "", ""
print("Enter operation and operands (e.g., + o1 o2 o3): ")

while True:
  line = input().split()
  if not line:
    break
  op1, op2, op3 = line[0], line[1], line[2]
  if len(line) > 3:
    op4 = line[3]
  else:
    op4 = "Result"
  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")
```

Output:

```
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
```

QA
10/04/2024.

Om Shete
THADOMAL SHAHANI TSEC ENGINEERING COLLEGE
TE C32
2103163

Assignment No: 1

Q.1 With references to assembler, consider any assembly program and generate Pass-1 and Pass-2 output. Also show contents of Database table involved in it.

⇒

- Assembly program:

```
START   100
        LDA  ALPHA
        STA  BETA
        HLT
ALPHA   RES  1
BETA    RES  2
        END  START
```

- Pass-1 output (Symbol table):

Symbol	Address
START	100
ALPHA	101
BETA	102

- Pass-2 output (Machine code):

100	0000	1010
101	0001	1010
102	0010	1010

- **Database table:**

Address	Label	Opcode	Operand
200	START		
201		LDA	ALPHA
202		STA	BETA
203		HLT	
204	ALPHA	RES	1
205	BETA	RES	1

- In Pass-1, the assembler reads the assembly code, assigns addresses to symbols, and creates a symbol table.
- In Pass-2, the assembler translates assembly instruction into machine code using the symbol table generated in Pass-1.
- The database table shows the address, label, opcode and operand for each instruction or data declaration in the program.

Q.2. Write short note on : YACC

⇒ Each translation rule input to YACC has a string specification that resembles a production of grammar it has a non-terminal on the LHS and a few alternatives on the RHS.

- For simplicity, we will refer to a string specification as a 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 shif action, it would invoke the scanner to obtain the next token and continue the parse by using 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 achieve the building of an intemediate representation or target code as follows :

1) Every non-terminal symbol in the parser has an attribute.
2) The semantic action uses the values of these attributes for building the intermediate representation or target code.

- A parser generator is a program that takes as input a specification of a syntax and produces as a output procedure for recognizing that language. Historically, they are also called compile compilers

- Input File: YACC input file is divided into three parts.

```
/* definitions */
%%
/* rules */
%%
/* auxiliary routines */
```

- Definition Part: The definition part includes the information about the tokens used in syntax definition. The definition part can include C code external to the definition of the parser and variable declarations, within %{ and %} in the first column.

- Rule part: The rule part contains grammar definitions in a modified BNF form. Actions are C code in { } and can be embedded inside.

- Auxiliary Routine Part: The auxiliary routine part is only C code. It includes function definition for every function needed in the rules part.

10/04/2024

Assignment 2

Q.1 Explain working of Direct Linking loader with example showing the 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 memory.
- Function of Loader :-

1) Allocation

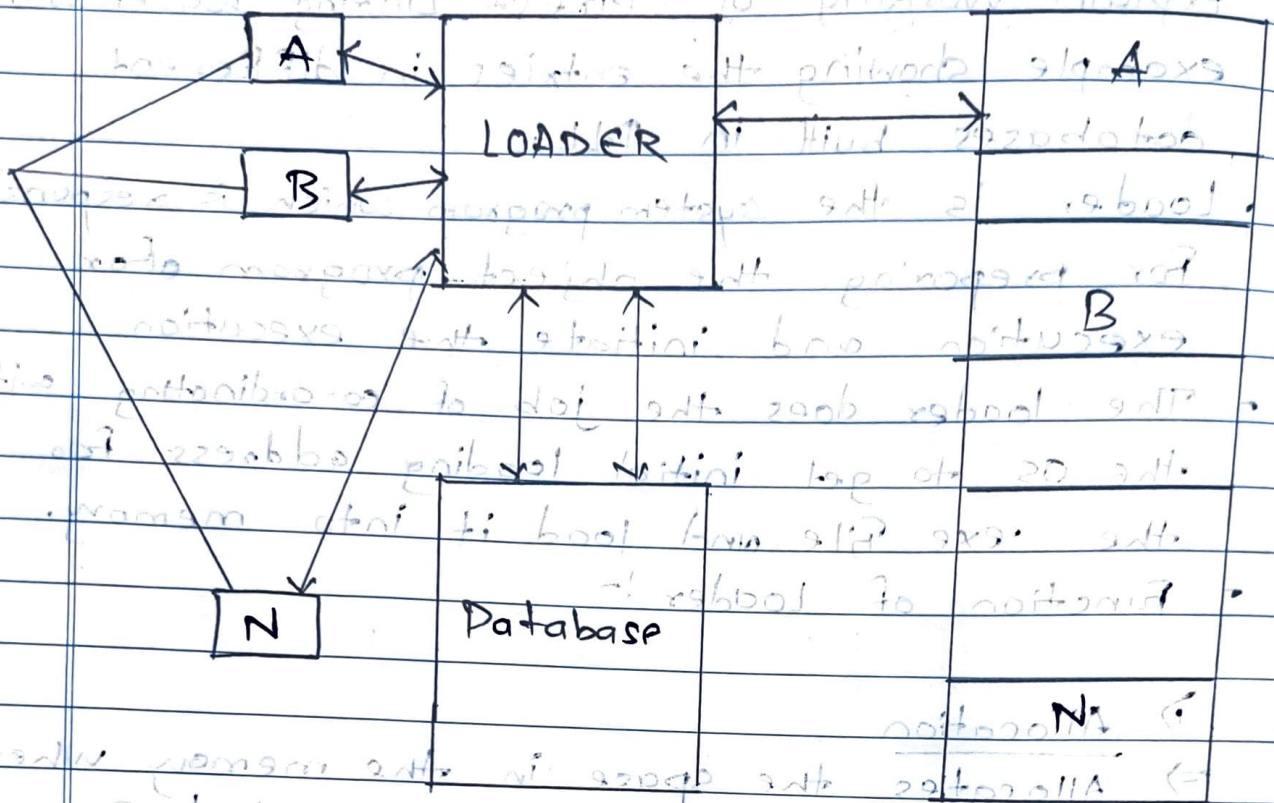
⇒ Allocates the space in the memory where the object program would be loaded for Execution.

2) Linking

⇒ It links two or more object codes and provides the information needed to allow references between them.

3) Loading

⇒ It brings the object program into the memory for execution.



- Dynamic Linking loader

⇒ Dynamic Linking Loader is a general re-locable loader.

- Allowing the 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 loader following information with each procedure or data segments.
- Dynamic linked shared libraries are easier to create than static linked shared libraries.

- Now, let's consider the entries in different databases built in this DLL:

- Function Entry Points Database

⇒ This database contains the memory addresses of all the functions exported by DLL, such as add, substract, multiply, and divide.

- Example Program for DLL

⇒

Source	Relative		Sample Program
1	0	LOOP1	START
2			ENTRY LOSTART0, LOSTART1
3			EXTRN L1START0, L1START1
4	8	LOSTART0	
5	12	LOSTART1	
6	16		DC B (LOSTART0)
7	20		DC B (LOSTART1 + 10)
8	24		DC B (LOSTAR1 - LOSTART0 - 2)
9	28		DC B (L1START1)
10	32		DC B (L1START0 + L1START1 + 5)
11			END
12	0	L1START1	START
13			ENTRY L1START0
14			EXTRN LOSTART0, LOSTART1

Source	Relative		Sample Program
15	4	LASTARTO	
16	14	DC	A(LOSTARTO)
17	18	DC	A(LOSTART1 + 10)
18	22	DC	A(LOSTART2 + LOSTART0 - 5)
19	END		

THADOMAL SHAHANI ENGINEERING COLLEGE
TSEC
Om Shete
TE C32
2103153

Assignment 3

Q. Write short notes 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, the term editor usually refers to the source code editors that include many special features for writing and editing code.
- Features normally associated with text editors are moving the cursor, deleting, replacing, pasting, finding, saving, etc.

- Types of Editors :-

1) Line Editors

⇒ In this, you can only edit one line at time or an integral number of lines. You cannot have a free-flowing sequence of characters. It will take care of only one line.

- For e.g., Teleprinter, edlin, teco

2) Stream Editors

⇒ In this type of editors, the file is treated as continuous flow or sequence of characters instead of line numbers, which means have you can type paragraphs.

- For e.g., Sed Editor in UNIX

3) Screen Editors

⇒ In this type of editors, the user is able to cursor on the screen and can make a copy, cut, paste operation easily.

- It is very easy to use mouse pointer.
- For eg., vi, emacs, Notepad.

4) Word Processor

⇒ Overcoming the limitation of screen editors, it allows one to use some format to insert images, files, videos, use font, size, style features.

- It majorly focuses on Natural Language.

5) Structure Editor

⇒ Structure editor focuses on programming languages. It provides features to write and edit source code.

- For e.g., Netbeans IDE, gEdit

b) Backpatching in Intermediate Code Generation

⇒ - Backpatching is basically a process of fulfilling unspecified information. This information is of labels.
- It basically uses the appropriate semantic actions during the process of code generation.
- It may indicate the address of the label in goto statements while producing TACs for the given expressions.
- Here basically two passes are used because assigning the position 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 the second round. Backpatching is the process of filling up gaps in incomplete transformations and information.
- Backpatching is mainly used for two purposes:

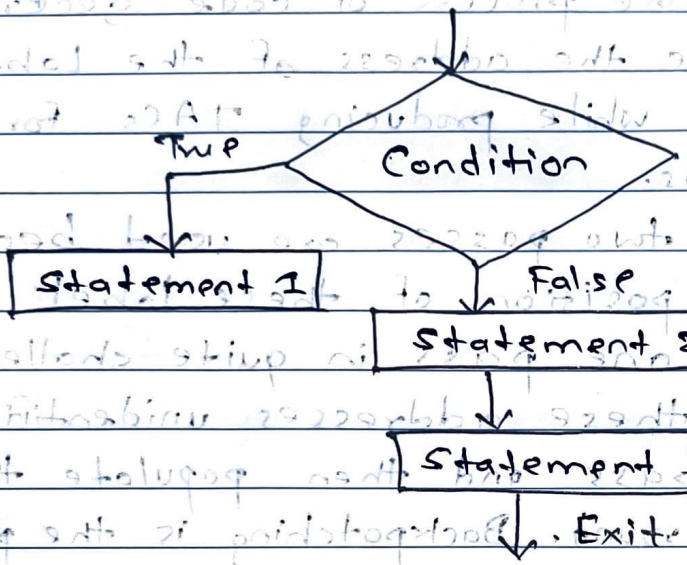
1) Boolean expression

⇒ Boolean expressions are statements whose results can be either true or false.
- A boolean expression which is named for mathematician George Boole is an expression that evaluates to either true or false.
- Let's look at some common language examples:
  - My favorite color is blue → true
  - I am afraid of mathematics → false.

2) Flow of control statements

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

- For e.g.,



3) Labels and Gotos :

⇒ The most elementary programming language construct for changing the flow of control in a program is a label and goto.

- When a compiler encounters a statement like goto L, it must check that there is exactly one statement with label L in the scope of this goto statements.