**Symbol table**

def generate\_symbol\_table(assembly\_code):

    symbol\_table = {}

    # First Pass: Collect labels and their addresses

    current\_address = None

    for line in assembly\_code:

        parts = line.split()

        if parts[0] == "START":

            current\_address = int(parts[1])

        elif parts[0] != "END":

            if(len(parts)==2):

              if(parts[0]=="READ"):

                symbol\_table[parts[1]]="Not assigned"

            elif(len(parts)==4):

              symbol\_table[parts[0]]=current\_address

            elif(len(parts)==3 and parts[1]=="DS"):

              symbol\_table[parts[0]]=current\_address

              current\_address+=int(parts[2])-1

            elif(len(parts)==3 and parts[1]=="DC"):

              symbol\_table[parts[0]]=current\_address

            current\_address+=1

    # Second Pass: Generate Symbol Table

    print("Symbol Table:")

    print("Label\t|\tAddress")

    print("----------------------")

    for label, address in symbol\_table.items():

        if label=="READ":

          continue

        print(f"{label}\t|\t{address}")

# Assembly code

assembly\_code = [

    "START 100",

    "READ A",

    "READ B",

    "LOOP MOVER AREG, A",

    "MOVER BREG, B",

    "COMP BREG, ='2'",

    "BC GT, LOOP",

    "BACK SUB AREG, B",

    "COMP AREG, ='5'",

    "BC LT, BACK",

    "STOP",

    "A DS 1",

    "print suyash",

    "B DS 1",

    "END"

]

# Generate symbol table

generate\_symbol\_table(assembly\_code)

**literal table**

def generate\_literal\_table(assembly\_code):

    ans=[]

    literal\_table = {}

    # First Pass: Collect literals and their values

    current\_address = None

    for line in assembly\_code:

        parts = line.split()

        if parts[0] == "START":

            current\_address = int(parts[1])

        elif parts[0] != "END" and parts[0]!='LTORG':

            for part in parts:

                if part.startswith("='"):

                    literal = part.strip("='")

                    literal\_table[literal] = None

            current\_address += 1

        elif parts[0]=='LTORG':

           for literal in literal\_table:

              literal\_table[literal] = current\_address

              current\_address += 1

           ans.append(literal\_table)

           literal\_table={}

    for literal in literal\_table:

        if(literal\_table[literal] == None):

          literal\_table[literal] = current\_address

          current\_address += 1

    ans.append(literal\_table)

    literal\_table={}

    # Second Pass: Generate Literal Table

    print("Literal Table:")

    print("Literal\t|\tValue")

    print("----------------------")

    for i in range(len(ans)):

      for literal, value in ans[i].items():

          print(f"{literal}\t|\t{value}")

# Assembly code

assembly\_code = [

    "START 200",

    "READ X",

    "READ Y",

    "MOVER AREG, ='5'",

    "MOVER BREG, ='6'",

    "ADD AREG, BREG",

    "LOOP MOVER CREG, X",

    "ADD CREG, ='1'",

    "COMP CREG, Y",

    "BC LT, LOOP",

    "LTORG",

    "NEXT SUB AREG, ='1'",

    "COMP AREG, Y",

    "BC GT, NEXT",

    "STOP",

    "X DS 1",

    "Y DS 1",

    "END"

]

# Generate literal table

generate\_literal\_table(assembly\_code)

**pool table**

def pool\_table(assembly\_code):

    ans=[]

    idx=0

    ans.append(idx)

    literal\_table={}

    # First Pass: Collect literals and their values

    current\_address = None

    for line in assembly\_code:

        parts = line.split()

        if parts[0] == "START":

            current\_address = int(parts[1])

        elif parts[0] != "END" and parts[0]!='LTORG':

            for part in parts:

                if part.startswith("='"):

                    literal = part.strip("='")

                    literal\_table[literal] = None

            current\_address += 1

        elif parts[0]=='LTORG':

           for literal in literal\_table:

              literal\_table[literal] = current\_address

              current\_address += 1

              idx+=1

           ans.append(idx)

           literal\_table={}

    for literal in literal\_table:

        if(literal\_table[literal] == None):

          literal\_table[literal] = current\_address

          current\_address += 1

          idx+=1

    ans.append(idx)

    literal\_table={}

    # Second Pass: Generate Literal Table

    print("Pool Table:")

    print("Index")

    print("----------------------")

    for i in range(len(ans)):

      print(ans[i])

# Assembly code

assembly\_code = [

    "START 200",

    "READ X",

    "READ Y",

    "MOVER AREG, ='5'",

    "MOVER BREG, ='6'",

    "ADD AREG, BREG",

    "LOOP MOVER CREG, X",

    "ADD CREG, ='1'",

    "COMP CREG, Y",

    "BC LT, LOOP",

    "LTORG",

    "NEXT SUB AREG, ='1'",

    "COMP AREG, Y",

    "BC GT, NEXT",

    "STOP",

    "X DS 1",

    "Y DS 1",

    "END"

]

# Generate literal table

pool\_table(assembly\_code)

**intermediate code**

def IS(word):

  answer="("

  if(word=="STOP"):

    answer+=("IS,00)")

  elif(word=="ADD"):

    answer+=("IS,01)")

  elif(word=="SUB"):

    answer+=("IS,02)")

  elif(word=="MUL"):

    answer+=("IS,03)")

  elif(word=="MOVER"):

    answer+=("IS,04)")

  elif(word=="MOVEM"):

    answer+=("IS,05)")

  elif(word=="COMP"):

    answer+=("IS,06)")

  elif(word=="BC"):

    answer+=("IS,07)")

  elif(word=="DIV"):

    answer+=("IS,08)")

  elif(word=="READ"):

    answer+=("IS,09)")

  elif(word=="PRINT"):

    answer+=("IS,10)")

  else:

    return DL(word)

  return answer

def DL(word):

  answer="("

  if(word=="DS"):

    answer+=("DL,01)")

  elif(word=="DC"):

    answer+=("DL,02)")

  else:

    return ""

  return answer

def AD(word):

  answer="("

  if(word=="START"):

    answer+=("AD,01)")

  elif(word=="END"):

    answer+=("AD,01)")

  elif(word=="ORIGIN"):

    answer+=("AD,01)")

  elif(word=="EQU"):

    answer+=("AD,01)")

  elif(word=="LTORG"):

    answer+=("AD,01)")

  return answer

def RG(word):

  answer="(RG,"

  if(word=="AREG,"):

    answer+=("1)")

  elif(word=="BREG,"):

    answer+=("2)")

  elif(word=="CREG,"):

    answer+=("3)")

  else:

    return CC(word)

  return answer

def CC(word):

  answer=""

  if(word=="EQ,"):

    answer+=("(CC,1)")

  elif(word=="LT,"):

    answer+=("(CC,2)")

  elif(word=="GT,"):

    answer+=("(CC,3)")

  elif(word=="NE,"):

    answer+=("(CC,4)")

  elif(word=="LE,"):

    answer+=("(CC,5)")

  elif(word=="GE,"):

    answer+=("(CC,6)")

  elif(word=="ANY,"):

    answer+=("(CC,7)")

  return answer

def IC(assembly\_code):

  ST=[]

  LT=[]

  PT=0

  for line in assembly\_code:

    parts=line.split()

    answer=""

    if(len(parts)==1):

        if(parts[0]=="STOP"):

          answer+=(IS(parts[0]))

        else:

          answer+=(AD(parts[0]))

    elif(len(parts)==2):

          if(parts[0]=="START"):

            answer+=(DL(parts[0]))

            answer+=("(C,")

            answer+=(parts[1])

            answer+=(")")

            ADD=int(parts[1])

          else:

            answer+=(IS(parts[0]))

            if parts[1] in ST:

              answer+=("(S,")

              answer+=(ST.index(parts[1]))

            else:

              answer+=("(")

              answer+=(parts[1])

              ST+=(parts[1])

              answer+=(")")

    elif(len(parts)==3):

         answer+=(IS(parts[0]))

         answer+=(RG(parts[1]))

         if(parts[2]=="AREG" or parts[2]=="BREG" or parts[2]=="CREG"):

          parts[2]+=","

          answer+=RG(parts[2])

         else:

          if(parts[2].startswith("='")):

           if( parts[2] in LT):

              answer+=("(L,")

              answer+=(LT.index(parts[2]))

              answer+=(")")

           else:

              LT+=(parts[2])

              PT+=1

              answer+=("(L,")

              answer+=(str(PT))

              answer+=(")")

          elif(parts[1]=="DS" or parts[1]=="DC"):

              answer+=("(S,")

              answer+=(str(ST.index(parts[0])))

              answer+=(")")

              answer+=(DL(parts[1]))

              answer+=("(C,")

              answer+=(parts[2])

              answer+=(")")

          else:

           if parts[2] in ST:

              answer+=("(S,")

              answer+=(str(ST.index(parts[2])))

              answer+=(")")

           else:

              answer+=("(")

              answer+=(parts[2])

              ST.append(parts[2])

              answer+=(")")

    elif(len(parts)==4):

         ST.append(parts[0])

         answer+=(IS(parts[1]))

         answer+=(RG(parts[2]))

         if(parts[3].startswith("='")):

           if( parts[3] in LT):

              answer+=("(L,")

              answer+=(LT.index(parts[3]))

              answer+=(")")

           else:

              LT.append(parts[3])

              PT+=1

              answer+=("(L,")

              answer+=(str(PT))

              answer+=(")")

         else:

           if parts[3] in ST:

              answer+=("(S,")

              answer+=(str(ST.index(parts[3])))

           else:

              answer+=("(S,")

              answer+=(parts[3])

              ST.append(parts[3])

         answer+=(")")

    print(answer)

assembly\_code = [

    "START 200",

    "READ X",

    "READ Y",

    "MOVER AREG, ='5'",

    "MOVER BREG, ='6'",

    "ADD AREG, BREG",

    "LOOP MOVER CREG, X",

    "ADD CREG, ='1'",

    "COMP CREG, Y",

    "BC LT, LOOP",

    "LTORG",

    "NEXT SUB AREG, ='1'",

    "COMP AREG, Y",

    "BC GT, NEXT",

    "STOP",

    "X DS 1",

    "Y DS 1",

    "END"

]

# Generate literal table

IC(assembly\_code)

**Mnt and mdt**

file = open("INPUT CODE 6.txt")

lines= file.readlines()

tlines = []

for line in lines:

  tokens = line.split()

  tlines.append(tokens)

MNTP= 0

MDTP = 0

#Macro name, MDT pointer

MNT = []

#Definition

MDT = []

ALAA = []

#[[arg1, #1,]]

ALA = []

def formatString(word):

    return word.split(',')[0]

def prepALA(parameters):

  d = {}

  for i in range(len(parameters)):

    d[parameters[i]] = "#"+str(i+1)

  #print(d)

  return d

def formatLine(ALA, line):

  parameters = list(ALA.keys())

  for i in range(len(line)):

    if(line[i] in parameters):

      line[i] = ALA[line[i]]

  return (line)

i = 0

n = len(tlines)

for line in tlines:

    for i in range(len(line)):

        line[i] = formatString(line[i])

i=0

ALAPointer = 0

while(i < n):

  line = tlines[i]

  if(line[0] == 'MACRO'):

    MNT.append([line[1], MDTP])

    MDT.append(line)

    MDTP+=1

    ALA = prepALA(line[2:])

    ALAA.append(ALA)

    ALAPointer +=1

    j = i+1

    line = tlines[j]

    while(j < n and line[0] != 'MEND'):

      j+=1

      line = tlines[j]

      l = formatLine(ALA, line)

      MDT.append(l)

      MDTP+=1

    i = j

  elif(line[0] == 'MEND'):

    i+=1

    MDTP+=1

    MNTP+=1

  else:

    i+=1

print("MNT")

for i in MNT:

    print(i)

print("\nMDT")

for i in MDT:

    print(i)

**three address code**

import java.util.\*;

public class ThreeAddressCode {

    private List<String[]> code;

    private int tempCount;

    public ThreeAddressCode() {

        code = new ArrayList<>();

        tempCount = 0;

    }

    public String newTemp() {

        String tempName = "t" + tempCount;

        tempCount++;

        return tempName;

    }

    public void genCode(String op, String arg1, String arg2, String result) {

        String[] instruction = {op, arg1, arg2, result};

        code.add(instruction);

    }

    public String infixToPostfix(String expression) {

        Map<Character, Integer> precedence = new HashMap<>();

        precedence.put('+', 1);

        precedence.put('-', 1);

        precedence.put('\*', 2);

        precedence.put('/', 2);

        precedence.put('^', 3);

        Stack<Character> stack = new Stack<>();

        StringBuilder postfix = new StringBuilder();

        String[] tokens = expression.split("\\s+");

        for (String token : tokens) {

            char ch = token.charAt(0);

            if (Character.isDigit(ch) || Character.isLetter(ch)) {

                postfix.append(token).append(" ");

            } else if (ch == '(') {

                stack.push(ch);

            } else if (ch == ')') {

                while (!stack.isEmpty() && stack.peek() != '(') {

                    postfix.append(stack.pop()).append(" ");

                }

                stack.pop(); // Discard '('

            } else {

                while (!stack.isEmpty() && precedence.getOrDefault(stack.peek(), 0) >= precedence.getOrDefault(ch, 0)) {

                    postfix.append(stack.pop()).append(" ");

                }

                stack.push(ch);

            }

        }

        while (!stack.isEmpty()) {

            postfix.append(stack.pop()).append(" ");

        }

        return postfix.toString().trim();

    }

    public String generateExpressionCode(String expression) {

        String postfixExpression = infixToPostfix(expression);

        String[] tokens = postfixExpression.split("\\s+");

        Stack<String> stack = new Stack<>();

        for (String token : tokens) {

            char ch = token.charAt(0);

            if (Character.isDigit(ch) || Character.isLetter(ch)) {

                stack.push(token);

            } else {

                String op = token;

                String arg2 = stack.pop();

                String arg1 = stack.pop();

                String result = newTemp();

                genCode(op, arg1, arg2, result);

                stack.push(result);

            }

        }

        return stack.pop();

    }

    public String displayReadableCode() {

        StringBuilder readableCode = new StringBuilder();

        for (String[] instruction : code) {

            String readableInstruction = instruction[3] + " = " + instruction[1] + " " + instruction[0] + " " + instruction[2];

            readableCode.append(readableInstruction).append("\n");

        }

        return readableCode.toString();

    }

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        System.out.print("Enter the expression: ");

        String expression = scanner.nextLine();

        ThreeAddressCode tac = new ThreeAddressCode();

        String result = tac.generateExpressionCode(expression);

        System.out.println("\nReadable code:");

        System.out.println(tac.displayReadableCode());

    }

}

**Lex (part of speech)**

%{ /\*First lex program \*/

#include<stdio.h>

%}

%%

[\t]+ ;

Dread|Run|arrives {printf("%s, is a verb", yytext);}

it {printf("%s,is a pronoun",yytext);}

Destiny {printf("%s,is a noun",yytext);}

same {printf("%s,is a adjective",yytext);}

all {printf("%s,is a adverb",yytext);}

from {printf("%s,is a preposition",yytext);}

the {printf("%s,is a preposition",yytext);}

[a-zA-Z]+   {printf("%s, is a invalid token",yytext);}

%%

int yywrap(void)

{

return 1;

}

int main(void)

{

yylex();

yywrap();

return 0;

}

**Uppercase/lowercase**

**Small.y**

Converter.y file:

%{

#include <stdio.h>

int yylex();

void yyerror(const char \*);

%}

%union {

    char c;

}

%token <c> LETTER

%token NEWLINE

%%

input: /\* empty \*/

    | input line

line: letters NEWLINE    { printf("\n"); }

    ;

letters: LETTER         { printf("%c", $1); }

       | letters LETTER { printf("%c", $2); }

       ;

%%

int main() {

    yyparse();

    return 0;

}

void yyerror(const char \*s) {

    printf("%s\n", s);

}

**Lexer\_small.l**

%{

#include "y.tab.h"

%}

%%

[a-z]   { yylval.c = yytext[0] - 32; return LETTER; }

[A-Z]   { yylval.c = yytext[0] + 32; return LETTER; }

\n      { return NEWLINE; }

.       { return yytext[0]; }

%%

int yywrap() {

    return 1;

}

**Yacc**

%{

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

// Function prototypes

int yylex(void);

void yyerror(const char \*);

// Define YYSTYPE as a double

#define YYSTYPE double

%}

%token NUM

%left '+' '-'

%left '\*' '/'

%%

expression:

expr

{

    printf("Result: %.2f\n", $1);

    return 0;

}

    ;

expr: expr '+' expr     { $$ = $1 + $3; }

    | expr '-' expr     { $$ = $1 - $3; }

    | expr '\*' expr     { $$ = $1 \* $3; }

    | expr '/' expr     { $$ = $1 / $3; }

    | '(' expr ')'      { $$ = $2; }

    | NUM               { $$ = $1; }

    ;

%%

int yylex() {

    int c;

    do {

        c = getchar();

    } while (c == ' ' || c == '\t');

    if (isdigit(c) || c == '.') {

        ungetc(c, stdin);

        scanf("%lf", &yylval);

        return NUM;

    }

    return c;

}

void yyerror(const char \*s) {

    fprintf(stderr, "%s\n", s);

}

int main() {

printf("\nEnter Any Arithmetic Expression :");

    yyparse();

    return 0;

}

**Yacc(pow,log)**

**.l file**

**%{**

**#include "y.tab.h"**

**#include <math.h>**

**#include <string.h>**

**extern void yyerror(const char \*);**

**%}**

**%option noyywrap**

**%%**

**[ \t\n]+ { /\* Ignore whitespace \*/ }**

**[a-zA-Z\_][a-zA-Z0-9\_]\* {**

**if (strcmp(yytext, "pow") == 0) return POW;**

**if (strcmp(yytext, "log") == 0) return LOG;**

**yylval.str = strdup(yytext);**

**return IDENTIFIER;**

**}**

**[0-9]+(\.[0-9]+)? { yylval.num = atof(yytext); return NUMBER; }**

**"=" { return '='; }**

**";" { return ';'; }**

**"(" { return '('; }**

**")" { return ')'; }**

**"," { return ','; }**

**"+" { return '+'; }**

**"-" { return '-'; }**

**"/" { return '/'; }**

**. { yyerror("Invalid character"); }**

**%%**

**.y file**

**%{**

**#include <stdio.h>**

**#include <math.h>**

**#include <stdlib.h>**

**#include <string.h>**

**void yyerror(const char \*s);**

**extern int yylex();**

**%}**

**%union {**

**double num; // For numerical values**

**char\* str; // For string values (e.g., identifiers)**

**}**

**%token <str> IDENTIFIER**

**%token <num> NUMBER**

**%token POW LOG**

**%type <num> expr // Expressions return a numeric value**

**%type <num> statement // Statements return a numeric value (for expression statements)**

**%left '+' '-'**

**%left '\*' '/'**

**%right NEG**

**%%**

**program:**

**| program statement**

**;**

**statement:**

**expr ';' { printf("Result = %lf\n", $1); }**

**| IDENTIFIER '=' expr ';' {**

**printf("%s = %lf\n", $1, $3);**

**free($1);**

**}**

**;**

**expr:**

**NUMBER { $$ = $1; }**

**| IDENTIFIER { printf("Variable [%s] used, but not defined in this scope.\n", $1); free($1); $$ = 0; }**

**| expr '+' expr { $$ = $1 + $3; }**

**| expr '-' expr { $$ = $1 - $3; }**

**| expr '\*' expr { $$ = $1 \* $3; }**

**| expr '/' expr { $$ = $1 / $3; }**

**| '-' expr %prec NEG { $$ = -$2; }**

**| '(' expr ')' { $$ = $2; }**

**| POW '(' expr ',' expr ')' { $$ = pow($3, $5); }**

**| LOG '(' expr ')' { $$ = log($3); }**

**;**

**%%**

**void yyerror(const char \*s) {**

**fprintf(stderr, "Error: %s\n", s);**

**}**

**int main(void) {**

**printf("Enter an expression:\n");**

**yyparse();**

**return 0;**

**}**

**###################################################32**

**yacc(sin,cos)**

**.l file**

**%{**

**#include "y.tab.h" // make sure this matches your Bison-generated header**

**#include <math.h>**

**#include <string.h>**

**%}**

**%option noyywrap**

**%option yylineno**

**%%**

**[uU][a-zA-Z\_]\* { yylval.var = strdup(yytext); return VARIABLE; }**

**[0-9]+(\.[0-9]+)? { yylval.num = atof(yytext); return NUMBER; }**

**"sin" { return SIN; }**

**"cos" { return COS; }**

**"+" { return PLUS; }**

**"-" { return MINUS; }**

**"\*" { return TIMES; }**

**"/" { return DIVIDE; }**

**"(" { return LPAREN; }**

**")" { return RPAREN; }**

**"=" { return EQUALS; }**

**";" { return SEMICOLON; }**

**[ \t]+ ; // Ignore whitespace**

**. { return yytext[0]; } // Handle other characters**

**%%**

**.y file**

**%{**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <math.h> // Include for math functions**

**#include "y.tab.h" // Ensure this is the header generated by Flex**

**void yyerror(const char \*s);**

**int yylex(void);**

**extern int yylineno; // External declaration if using %option yylineno in Flex**

**double vars[256]; // Simple variable storage based on ASCII index**

**%}**

**%union {**

**double num; // For numerical values**

**char\* var; // For variable names**

**}**

**%token <var> VARIABLE**

**%token <num> NUMBER**

**%token SIN COS**

**%token PLUS MINUS TIMES DIVIDE**

**%token LPAREN RPAREN**

**%token EQUALS SEMICOLON**

**%type <num> expression term factor**

**%type <var> assignment**

**%%**

**input:**

**| input line**

**;**

**line:**

**assignment SEMICOLON { printf("%s = %f\n", $1, vars[$1[0]]); }**

**| error SEMICOLON { yyerror("syntax error"); }**

**;**

**assignment:**

**VARIABLE EQUALS expression { vars[$1[0]] = $3; $$ = $1; }**

**;**

**expression:**

**expression PLUS term { $$ = $1 + $3; }**

**| expression MINUS term { $$ = $1 - $3; }**

**| term { $$ = $1; }**

**;**

**term:**

**term TIMES factor { $$ = $1 \* $3; }**

**| term DIVIDE factor { $$ = $1 / $3; }**

**| factor { $$ = $1; }**

**;**

**factor:**

**NUMBER { $$ = $1; }**

**| VARIABLE { $$ = vars[$1[0]]; }**

**| LPAREN expression RPAREN { $$ = $2; }**

**| SIN LPAREN expression RPAREN { $$ = sin($3); }**

**| COS LPAREN expression RPAREN { $$ = cos($3); }**

**;**

**%%**

**void yyerror(const char \*s) {**

**fprintf(stderr, "Error near line %d: %s\n", yylineno, s);**

**}**

**int main(void) {**

**printf("Enter expressions (e.g., 'u = sin(12) + cos(12);'):\n");**

**yyparse();**

**return 0;**

**}**

**####################################################################################**

**YACC (Sqrroot, strlength)**

**.l file**

**%{**

**#include "y.tab.h"**

**#include <stdlib.h>**

**#include <string.h>**

**%}**

**%option noyywrap**

**%%**

**[0-9]+ { yylval.ival = atoi(yytext); return NUMBER; }**

**\"[^\"]\*\" { yylval.sval = strdup(yytext); return STRING; }**

**"sqrt" { return SQRT; }**

**"strlen" { return STRLEN; }**

**"=" { return '='; }**

**[ \t\n]+ { /\* ignore whitespace \*/ }**

**. { return yytext[0]; }**

**%%**

**.y file**

**%{**

**#include <stdio.h>**

**#include <math.h>**

**#include <string.h>**

**extern int yylex();**

**void yyerror(const char \*s) { fprintf(stderr, "%s\n", s); }**

**%}**

**%union {**

**int ival;**

**char \*sval;**

**}**

**%token <sval> STRING**

**%token <ival> NUMBER SQRT STRLEN**

**%type <ival> expr function**

**%type <sval> var**

**%%**

**program:**

**program statement**

**|**

**;**

**statement:**

**var '=' expr { printf("%s = %d\n", $1, $3); }**

**;**

**var:**

**'u' { $$ = strdup("u"); }**

**| 'v' { $$ = strdup("v"); }**

**;**

**expr:**

**function**

**;**

**function:**

**SQRT '(' NUMBER ')' { $$ = (int) sqrt($3); }**

**| STRLEN '(' STRING ')' { $$ = strlen($3) - 2; /\* Subtract 2 for the quotes \*/ }**

**;**

**%%**

**int main(void) {**

**printf("Enter expressions like 'u = sqrt(36)' or 'v = strlen(\"pune\")':\n");**

**return yyparse();**

**}**

**#############################################################**

**MDt and MNT**

**# Assembly code with macro definitions**

**assembly\_code = [**

**"LOAD A",**

**"STORE B",**

**"MACRO ABC",**

**"LOAD p",**

**"SUB q",**

**"MEND",**

**"MACRO ADD1 ARG",**

**"LOAD X",**

**"STORE ARG",**

**"MEND",**

**"MACRO ADD5 A1, A2, A3",**

**"STORE A2",**

**"ADD1 5",**

**"ADD1 10",**

**"LOAD A1",**

**"LOAD A3",**

**"MEND",**

**"ABC",**

**"ADD5 D1, D2, D3",**

**"END",**

**]**

**# Step 1: Define Macro Definition Table (MDT) and Macro Name Table (MNT)**

**macro\_definition\_table = []**

**macro\_name\_table = {}**

**current\_macro\_name = None**

**current\_macro\_start = None**

**# Parse the assembly code to find macro definitions**

**for line in assembly\_code:**

**parts = line.split()**

**if len(parts) == 0:**

**continue**

**if parts[0] == "MACRO":**

**# Start of a new macro definition**

**current\_macro\_name = parts[1]**

**current\_macro\_start = len(macro\_definition\_table) # Starting index in MDT**

**elif parts[0] == "MEND":**

**# End of the current macro definition**

**if current\_macro\_name:**

**# Update the MNT with macro name and start index in MDT**

**macro\_name\_table[current\_macro\_name] = current\_macro\_start**

**current\_macro\_name = None**

**current\_macro\_start = None**

**else:**

**if current\_macro\_name:**

**# Add macro body lines to the MDT**

**macro\_definition\_table.append(line)**

**# Display the Macro Name Table (MNT)**

**print("Macro Name Table (MNT):")**

**print("Macro Name\tMDT Index")**

**for macro\_name, mdt\_index in macro\_name\_table.items():**

**print(f"{macro\_name}\t\t{mdt\_index}")**

**# Display the Macro Definition Table (MDT)**

**print("\nMacro Definition Table (MDT):")**

**print("MDT Index\tInstruction")**

**for index, instruction in enumerate(macro\_definition\_table):**

**print(f"{index}\t\t{instruction}")**

**---------------------------------------------------**

**Q7**

**# Assembly source code with macro definitions**

**assembly\_code = [**

**"LOAD J",**

**"STORE M",**

**"MACRO EST1",**

**"LOAD e",**

**"ADD d",**

**"MEND",**

**"MACRO EST ABC",**

**"EST1",**

**"STORE ABC",**

**"MEND",**

**"MACRO ADD7 P4, P5, P6",**

**"LOAD P5",**

**"EST 8",**

**"SUB4 2",**

**"STORE P4",**

**"STORE P6",**

**"MEND",**

**"EST",**

**"ADD7 C4, C5, C6",**

**"END",**

**]**

**# Step 1: Initialize the MDT and MNT**

**macro\_definition\_table = [] # Stores macro content**

**macro\_name\_table = {} # Stores macro names with MDT index reference**

**current\_macro\_name = None # Current macro name being processed**

**current\_macro\_start = None # Starting index of the current macro in MDT**

**# Step 2: Parse the assembly code to find macro definitions**

**for line in assembly\_code:**

**parts = line.split()**

**if len(parts) == 0:**

**continue**

**if parts[0] == "MACRO":**

**# Start of a new macro definition**

**current\_macro\_name = parts[1] # Macro name**

**current\_macro\_start = len(macro\_definition\_table) # MDT index**

**elif parts[0] == "MEND":**

**# End of the macro definition**

**if current\_macro\_name is not None:**

**# Add the macro to the MNT**

**macro\_name\_table[current\_macro\_name] = current\_macro\_start**

**current\_macro\_name = None**

**current\_macro\_start = None**

**else:**

**if current\_macro\_name:**

**# Add the macro content to the MDT**

**macro\_definition\_table.append(line)**

**# Step 3: Display the Macro Name Table (MNT)**

**print("Macro Name Table (MNT):")**

**print("Macro Name\tMDT Index")**

**for macro\_name, mdt\_index in macro\_name\_table.items():**

**print(f"{macro\_name}\t\t{mdt\_index}")**

**# Step 4: Display the Macro Definition Table (MDT)**

**print("\nMacro Definition Table (MDT):")**

**print("MDT Index\tInstruction")**

**for index, instruction in enumerate(macro\_definition\_table):**

**print(f"{index}\t\t{instruction}")**

**##################################################**

**Pune YACC**

**%{**

**#include &lt;stdio.h&gt;**

**#include &lt;string.h&gt;**

**#include &lt;ctype.h&gt;**

**%}**

**%token VARIABLE\_NAME**

**%%**

**input: VARIABLE\_NAME { printf(&quot;%s is a valid variable name.\n&quot;, $1); }**

**;**

**VARIABLE\_NAME: LETTER (LETTER | DIGIT | &#39;\_&#39;)\***

**{ yylval = strdup(yytext); }**

**;**

**LETTER: [a-zA-Z]**

**;**

**DIGIT: [0-9]**

**;**

**%%**

**int main() {**

**yyparse();**

**return 0;**

**}**