**9..**Write a program using LEX Tool, to implement a lexical analyzer for parts of speech for given English language without a Symbol table.

**INPUT**

Dread it. Run from it.

Destiny arrives all the same.

**code:**

%{

%}

%%

"run"|"walk"|"eat"|"sleep" { printf("Verb : %s\n", yytext); }

"I"|"you"|"he"|"she"|"it"|"we"|"they" { printf("Pronoun : %s\n", yytext); }

"quickly"|"slowly"|"happily"|"sadly" { printf("Adverb : %s\n", yytext); }

"dog"|"cat"|"car"|"tree" { printf("Noun : %s\n", yytext); }

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

. { printf("Unknown : %s\n", yytext); }

%%

int yywrap() {

return 1;

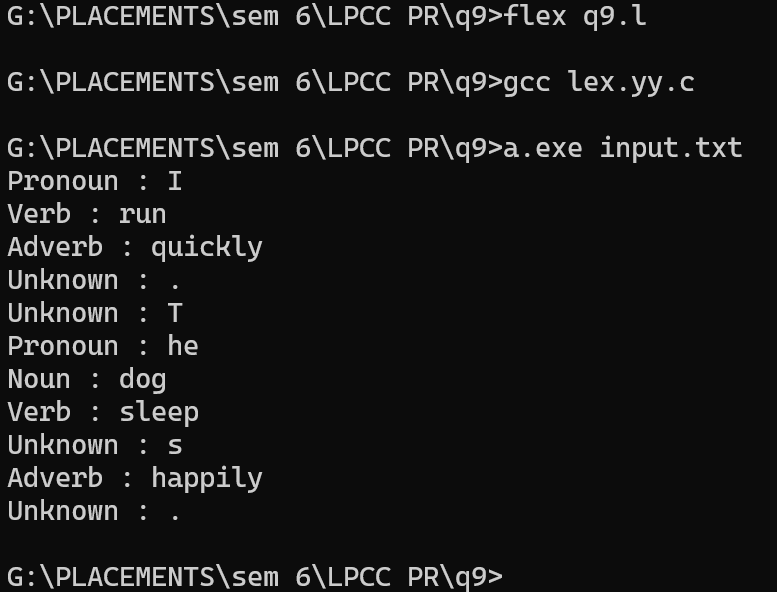
}

int main(int argc, char \*argv[]) {

++argv, --argc; /\* skip over program name \*/

if (argc > 0)

yyin = fopen(argv[0], "r");

else

yyin = stdin;

yylex();

}

**STEPS TO RUN:**

**flex your\_file\_name.l**

**gcc lex.yy.c -o lexer -lfl**

**./lexer input.txt**

**10.**Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table.

**INPUT**

{

int m=10,n=2,o;

o = m – n;

}

%{

#include <stdio.h>

%}

%option noyywrap

%%

"{" { printf("LEFT\_BRACE\n"); }

"}" { printf("RIGHT\_BRACE\n"); }

"int" { printf("INT\_KEYWORD\n"); }

";" { printf("SEMICOLON\n"); }

[ \t] ; /\* Skip whitespace \*/

\n ; /\* Skip newline \*/

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("IDENTIFIER\n"); }

[0-9]+ { printf("NUMBER\n"); }

"+" { printf("PLUS\n"); }

"-" { printf("MINUS\n"); }

"=" { printf("EQUALS\n"); }

"," { printf("COMMA\n"); }

. ; /\* Ignore other characters \*/

%%

int main() {

yylex();

return 0;

}

**11.**Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table.

**INPUT**

{

int a=3;

int b=4;

float c;

c = (a\*a + b\*b) \*2

}

%{

#include <stdio.h>

%}

%option noyywrap

%%

"{" { printf("LEFT\_BRACE\n"); }

"}" { printf("RIGHT\_BRACE\n"); }

"int" { printf("INT\_KEYWORD\n"); }

"float" { printf("FLOAT\_KEYWORD\n"); }

";" { printf("SEMICOLON\n"); }

"=" { printf("ASSIGNMENT\n"); }

"(" { printf("LEFT\_PAREN\n"); }

")" { printf("RIGHT\_PAREN\n"); }

"\*" { printf("MULTIPLICATION\n"); }

"+" { printf("ADDITION\n"); }

[ \t] ; /\* Skip whitespace \*/

\n ; /\* Skip newline \*/

[a-zA-Z\_][a-zA-Z0-9\_]\* { printf("IDENTIFIER\n"); }

[0-9]+ { printf("NUMBER\n"); }

"," { printf("COMMA\n"); }

. ; /\* Ignore other characters \*/

%%

int main() {

yylex();

return 0;

}

**12.** Write a program using LEX Tool, to implement a lexical analyzer for given C programming language without Symbol table.

**INPUT**

{

int total =100;

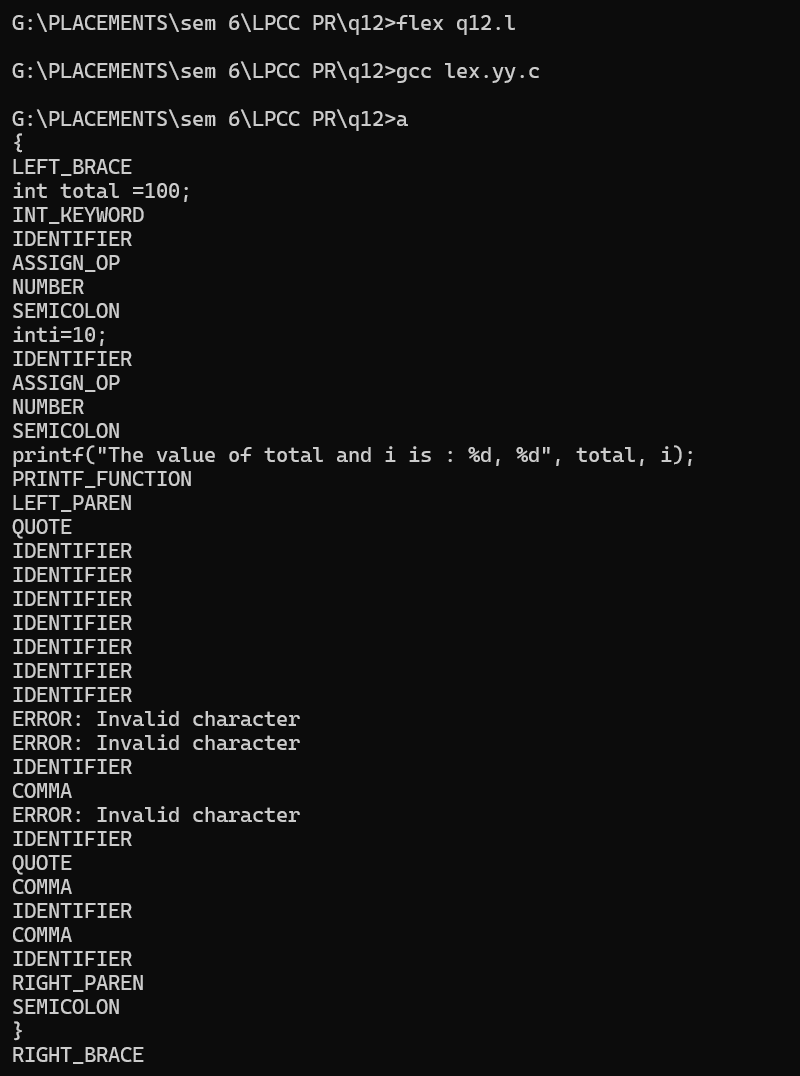
inti=10;

printf("The value of total and i is : %d, %d", total, i);

}

**COde:**

%{

#include <stdio.h>

%}

DIGIT [0-9]

LETTER [a-zA-Z]

%%

"int" { printf("INT\_KEYWORD\n"); }

"printf" { printf("PRINTF\_FUNCTION\n"); }

"=" { printf("ASSIGN\_OP\n"); }

";" { printf("SEMICOLON\n"); }

"," { printf("COMMA\n"); }

"\"" { printf("QUOTE\n"); }

"(" { printf("LEFT\_PAREN\n"); }

")" { printf("RIGHT\_PAREN\n"); }

"{" { printf("LEFT\_BRACE\n"); }

"}" { printf("RIGHT\_BRACE\n"); }

{LETTER}({LETTER}|{DIGIT})\* { printf("IDENTIFIER\n"); }

{DIGIT}+ { printf("NUMBER\n"); }

[ \t\n] ; // skip whitespace and newlines

. { printf("ERROR: Invalid character\n"); }

%%

int main() {

yylex();

return 0;

}

**13.**Write a program to evaluate a given arithmetic expression using YACC specification.

**INPUT**

0.33\*12-4-4+(3\*2)

**final code:**

**LEX:**

%{

#include "q13.tab.h" // Include Bison-generated header for YYSTYPE and token types

%}

%%

[0-9]+(\.[0-9]+)? {

yylval.val = atof(yytext); // Assign the atof result to the union member 'val'

return NUMBER;

}

[ \t]+ ; // Ignore spaces and tabs

\n { return 0; } // Return 0 on a newline to signal the end of input

. { return yytext[0]; } // Return any other character as itself

%%

int yywrap() {

return 1;

}

**YACC:**

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

int yylex(void);

%}

%union {

double val; // Define a union member for double values

}

%token <val> NUMBER // Specify that NUMBER carries a double

%type <val> E // The non-terminal E also returns a double

%left '+' '-'

%left '\*' '/'

%left '(' ')'

%%

ArithmeticExpression:

E { printf("\nResult = %.2f\n", $1); return 0; }

;

E:

E '+' E { $$ = $1 + $3; }

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

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

| E '/' E { if ($3 == 0.0) { yyerror("Division by zero"); $$ = 0.0; } else { $$ = $1 / $3; } }

| '(' E ')' { $$ = $2; }

| NUMBER { $$ = $1; }

;

%%

void yyerror(const char \*s) {

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

}

int main() {

printf("\nEnter an Arithmetic Expression (e.g., 2.5 + 3.2 \* 5):\n");

if (!yyparse())

printf("\nThe arithmetic expression is valid.\n");

else

printf("\nInvalid expression.\n");

return 0;

}

**commands:**

flex calc.l

bison -d calc.y

gcc lex.yy.c calc.tab.c

./a.exe

**14.**Write a program to evaluate a given variable name using YACC specification.

**SAMPLE INPUT**

1. pune
2. PUNE
3. Pune1
4. pUNE\_2

**LEX:**

%{

#include "varname.tab.h"

#include <string.h> // Include for strdup function

%}

%%

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

yylval.str = strdup(yytext); // Copy string to pass to Bison

return IDENTIFIER;

}

[ \t]+ ; // Ignore spaces and tabs

\n { return EOL; } // End of line, to process input

. { return yytext[0]; } // Catch-all for any other character

%%

int yywrap() {

return 1;

}

**YACC:**

%{

#include <stdio.h>

#include <stdlib.h>

void yyerror(const char \*s);

int yylex(void);

%}

%union {

char\* str; // For storing strings

}

%token <str> IDENTIFIER

%token EOL

%%

input:

| input line

;

line: IDENTIFIER EOL { printf("%s is a valid variable name.\n", $1); free($1); }

| EOL

| error EOL { yyerror("Invalid variable name."); }

;

%%

void yyerror(const char \*s) {

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

}

int main() {

printf("Enter variable names, one per line:\n");

yyparse();

return 0;

}

**commands:**

flex varname.l

bison -d varname.y

gcc lex.yy.c varname.tab.c

./a.exe

**15.**Write a program to evaluate a given built-in function using YACC specification.

**INPUT**

1.u= sqrt(36)

2. v = strlen(“pune”)

**LEX:**

%{

#include "functions.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]; }

%%

**YACC:**

%{

#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();

}

**commands:**

flex functions.l

bison -d functions.y

gcc lex.yy.c functions.tab.c

./a.exe

**16.**Write a program to evaluate a given built-in function using YACC specification.

**INPUT**

u= sin(12)+cos(12)

LEX:

filename: lexer.l

Code:

%{

#include "lexer.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

%%

YACC:

file name: lexer.y

Code:

%{

#include <stdio.h>

#include <stdlib.h>

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

#include "lexer.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);

}

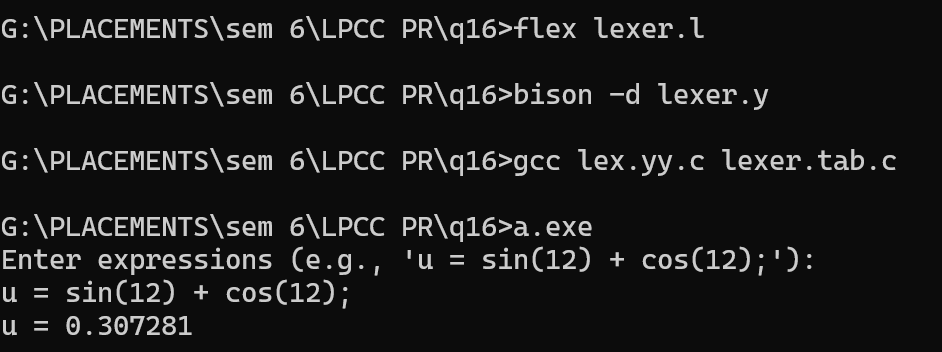
int main(void) {

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

yyparse();

return 0;

}



**17.**Write a program to evaluate a given built-in function using YACC specification.

**INPUT**

p= pow(3,2) / log (24);

lex:

filename: q17.l

code:

%{

#include "q17.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 ''; }

"/" { return '/'; }

. { yyerror("Invalid character"); }

%%

yacc:

filename: q17.y

code:

%{

#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);

}

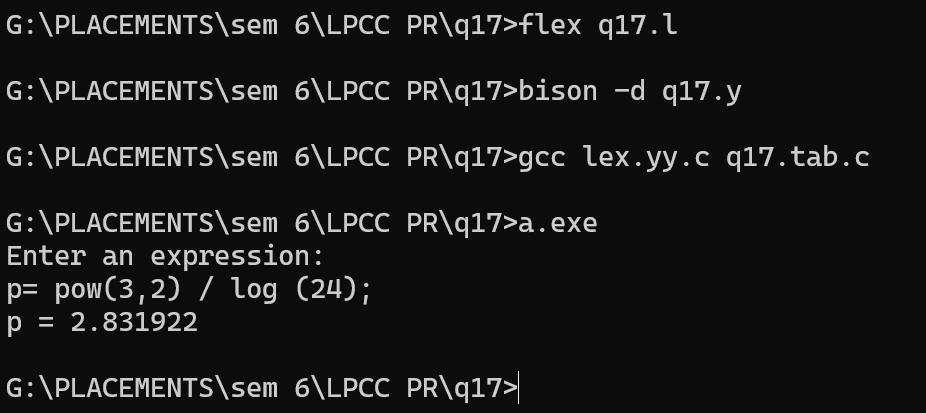
int main(void) {

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

yyparse();

return 0;

}



**1.** Write a program to generate a Symbol table of a two-pass Assembler for the given Assembly language source code.

| **INPUT/CODE**  START 180  READ M  READ N  LOOP MOVER AREG, M  MOVER BREG, N  COMP BREG, =’200’  BC GT, LOOP  BACK SUB AREG, M  COMP AREG, =’500’  BC LT, BACK  STOP  M DS 1  N DS 1  END |
| --- |

**2.** Write a program to generate a Literal table of a two-pass Assembler for the given Assembly language source code.

| **INPUT/CODE**  START 100  READ A  READ B  MOVER AREG, =’50'  MOVER BREG, =’60’  ADD AREG, BREG  LOOP MOVER CREG, A  ADD CREG, ='10'  COMP CREG, B  BC LT, LOOP  NEXT SUB AREG, ='10'  COMP AREG, B  BC GT, NEXT  STOP  A DS 1  B DS 1  END |
| --- |

**3**.Write a program to generate a Pool table of a two-pass Assembler for the given Assembly language source code.

| **INPUT/CODE**  START 100  READ A  MOVER AREG, ='1'  MOVEM AREG, B  MOVER BREG, ='6'  ADD AREG, BREG  COMP AREG, A  BC GT, LAST  LTORG  NEXT SUB AREG, ='1'  MOVER CREG, B  ADD CREG, ='8'  MOVEM CREG, B  PRINT B  LAST STOP  A DS 1  B DS 1  END |
| --- |

**class Assembler:**

**def \_\_init\_\_(self):**

**self.symbol\_table={}**

**self.literal\_table=[]**

**self.pool\_table=[]**

**self.lc=0**

**self.pool\_start=0**

**def first\_pass(self,source\_code):**

**for line in source\_code:**

**line=line.strip()**

**if line.startswith("START"):**

**self.lc=int(line.split()[1])**

**elif line.startswith("END"):**

**self.process\_ltorg()**

**break**

**elif line:**

**if "=" in line:**

**self.process\_literals(line)**

**elif line.startswith("LTORG"):**

**self.process\_ltorg()**

**else:**

**self.process\_label(line)**

**self.process\_instruction(line)**

**def process\_ltorg(self):**

**for i in range(self.pool\_start,len(self.literal\_table)):**

**self.literal\_table[i][1]=str(self.lc)**

**self.lc+=1**

**self.pool\_table.append(self.pool\_start)**

**self.pool\_start=len(self.literal\_table)**

**def process\_literals(self, line):**

**literal=line.split("='")[1].split("'")[0]**

**self.literal\_table.append([literal," "])**

**def process\_label(self, line):**

**parts=line.split()**

**if len(parts)>2 and parts[1]=="DS":**

**self.symbol\_table[parts[0]]=self.lc**

**elif len(parts)>1 and parts[0] not in ["MOVER","MOVEM","ADD","SUB","COMP","BC","READ","PRINT","STOP"]:**

**self.symbol\_table[parts[0]]=self.lc**

**def process\_instruction(self, line):**

**if not line.startswith("DS") and not line.startswith("LTORG"):**

**self.lc+=1**

**def generate\_tables(self):**

**print("SYMBOL TABLE :")**

**print("Label\tAddress")**

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

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

**print("Literal\tAddress")**

**for literal, address in self.literal\_table:**

**print(f"{literal}\t{address}")**

**print("\nPool Table:")**

**for address in self.pool\_table:**

**print(f"#{address}")**

**def assemble(self, source\_code):**

**self.first\_pass(source\_code)**

**self.generate\_tables()**

**def main():**

**source\_code=[**

**"START 100",**

**"READ A",**

**"MOVER AREG, ='1'",**

**"MOVEM AREG, B",**

**"MOVER BREG, ='6'",**

**"ADD AREG, BREG",**

**"COMP AREG, A",**

**"BC GT, LAST",**

**"LTORG",**

**"NEXT SUB AREG, ='1'",**

**"MOVER CREG, B",**

**"ADD CREG, ='8'",**

**"MOVEM CREG, B",**

**"PRINT B",**

**"LAST STOP",**

**"A DS 1",**

**"B DS 1",**

**"END"**

**]**

**assembler=Assembler()**

**assembler.assemble(source\_code)**

**if \_\_name\_\_=="\_\_main\_\_":**

**main()**

**4.**Write a program to generate Intermediate code of a two-pass Assembler for the given Assembly language source code.

| **INPUT/CODE**  START 100  READ A  READ B  MOVER AREG, A  SUB AREG, B  STOP  A DS 1  B DS 1  END |
| --- |

class IntermediateCodeGenerator:

def \_\_init\_\_(self):

self.source\_code = [

"START 100",

"READ A",

"READ B",

"MOVER AREG, A",

"SUB AREG, B",

"STOP",

"A DS 1",

"B DS 1",

"END"

]

self.intermediate\_code = []

def generate\_intermediate\_code(self):

for line in self.source\_code:

tokens = line.split()

opcode = tokens[0]

operand = ""

if len(tokens) > 1:

operand = tokens[1]

if opcode.upper() == "START":

self.intermediate\_code.append("AD " + opcode + ", " + operand)

elif opcode.upper() == "READ":

self.intermediate\_code.append("IS 1, " + operand)

elif opcode.upper() == "MOVER":

self.intermediate\_code.append("IS 4, " + operand + " AREG")

elif opcode.upper() == "SUB":

self.intermediate\_code.append("IS 2, " + operand + " BREG")

elif opcode.upper() == "STOP":

self.intermediate\_code.append("IS 0")

elif opcode.upper() == "DS":

self.intermediate\_code.append("DL 1, " + operand)

elif opcode.upper() == "END":

self.intermediate\_code.append("AD " + opcode)

def print\_intermediate\_code(self):

print("Intermediate Code:")

for code in self.intermediate\_code:

print(code)

# Create an instance of IntermediateCodeGenerator

generator = IntermediateCodeGenerator()

# Generate intermediate code

generator.generate\_intermediate\_code()

# Print intermediate code

generator.print\_intermediate\_code()

**5.**Write a program to generate Intermediate code of a two-pass Macro processor.

| **INPUT/CODE**  LOAD A  MACRO ABC  LOAD p  SUB q  MEND  STORE B  MULT D  MACRO ADD1 ARG  LOAD X  STORE ARG  MEND  …continued… | …continued…  LOAD B  MACRO ADD5 A1, A2, A3  STORE A2  ADD1 5  ADD1 10  LOAD A1  LOAD A3  MEND  ADD1 t  ABC  ADD5 D1, D2, D3  END |
| --- | --- |

**6.**Write a program to generateIntermediate code of a two-pass Macro processor.

| **INPUT/CODE**  LOAD J  STORE M  MACRO EST  LOAD e  ADD d  MEND  LOAD S  MACRO SUB4 ABC  LOAD U  STORE ABC  MEND | …continued….  LOAD P  ADD V  MACRO ADD7 P4, P5, P6  LOAD P5  SUB4 XYZ  SUB 8  SUB 2  STORE P4  STORE P6  MEND  EST  ADD7 C4, C5, C6  SUB4 z  END |
| --- | --- |

**7.**Write a program to generate MDT MNT(Macro Definition Table) of a two-pass Macro processor.

| **INPUT/CODE**  LOAD A  STORE B  MACRO ABC  LOAD p  SUB q  MEND  MACRO ADD1 ARG  LOAD X  STORE ARG  MEND  ….Continued…. | …continued….  MACRO ADD5 A1, A2, A3  STORE A2  ADD1 5  ADD1 10  LOAD A1  LOAD A3  MEND  ABC  ADD5 D1, D2, D3  END |
| --- | --- |

**8.**Write a program to generate MDT MNT(Macro Name Table) of a two-pass Macro processor.

| **INPUT/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 |
| --- | --- |

**def process\_macro\_definition(lines):**

**MNT = [] # Macro Name Table**

**MDT = [] # Macro Definition Table**

**ALA = {} # Argument List Array**

**macro\_name = None**

**macro\_args = []**

**macro\_started = False**

**macro\_start\_index = None**

**for line\_index, line in enumerate(lines):**

**tokens = line.strip().split()**

**if tokens and tokens[0] == 'MACRO':**

**macro\_name = tokens[1]**

**MNT.append([macro\_name, 0, len(MDT) + 1, None]) # Add starting index as None initially**

**if len(tokens) > 2:**

**macro\_args = [arg.rstrip(',') for arg in tokens[2:]] # Remove trailing commas**

**for i, arg in enumerate(macro\_args, 1):**

**ALA[arg] = f"#{i}"**

**MNT[-1][1] = len(macro\_args)**

**macro\_started = True**

**macro\_start\_index = line\_index**

**continue**

**elif tokens and tokens[0] == 'MEND':**

**MDT.append(line.strip())**

**MNT[-1][3] = len(MDT) - 1 # Update the starting index in MNT**

**macro\_started = False**

**continue**

**if macro\_started:**

**formatted\_line = []**

**for token in tokens:**

**if token in ALA:**

**token = ALA[token] # Replace macro arguments with positional parameters**

**formatted\_line.append(token)**

**MDT.append(' '.join(formatted\_line))**

**else:**

**print(line.strip()) # Print intermediate code**

**return MNT, MDT, ALA**

**def expand\_nested\_macros(MDT, MNT):**

**for i, line in enumerate(MDT):**

**tokens = line.strip().split()**

**if tokens[0] in [entry[0] for entry in MNT]: # Check if it's a macro call**

**macro\_index = [entry[0] for entry in MNT].index(tokens[0])**

**macro\_start\_index = MNT[macro\_index][2]**

**macro\_end\_index = MNT[macro\_index][3]**

**if macro\_end\_index is None:**

**print("Error: MEND not found for nested macro")**

**continue**

**macro\_definition = MDT[macro\_start\_index-1:macro\_end\_index]**

**MDT = MDT[:i] + macro\_definition + MDT[i+1:]**

**return MDT**

**def replace\_parameters(MDT, ALA):**

**for i, line in enumerate(MDT):**

**tokens = line.strip().split()**

**for j, token in enumerate(tokens):**

**if token in ALA:**

**tokens[j] = ALA[token]**

**MDT[i] = ' '.join(tokens)**

**return MDT**

**def print\_mnt(MNT):**

**print("\nMNT:")**

**print("(Name of macro, No. of parameters, Start index in MDT)")**

**for entry in MNT:**

**print(entry[:3])**

**def print\_mdt(MDT):**

**print("\nMDT:")**

**for i, line in enumerate(MDT, start=1):**

**print(f"{i}) {line}")**

**def print\_ala(ALA):**

**print("\nALA:")**

**for key, value in ALA.items():**

**print(f"{key}: {value}")**

**def main():**

**with open("input.txt", "r") as file:**

**lines = file.readlines()**

**MNT, MDT, ALA = process\_macro\_definition(lines)**

**MDT = expand\_nested\_macros(MDT, MNT)**

**MDT = replace\_parameters(MDT, ALA)**

**print("\n--------------------------------------------------------------------")**

**print\_mnt(MNT)**

**print("\n--------------------------------------------------------------------")**

**print\_mdt(MDT)**

**print("\n--------------------------------------------------------------------")**

**# print\_ala(ALA)**

**if \_\_name\_\_ == "\_\_main\_\_":**

**main()**

**18.**Write a program to generate three address codes for the given simple expression.

**INPUT**

w = u\*u - u\*v+ v\*v

class generate\_TAC:

temp\_count = 1

@staticmethod

def precedence(op):

if op in ['+', '-']:

return 1

elif op in ['\*', '/']:

return 2

return -1

@staticmethod

def apply\_op(op, a, b):

result = "t" + str(generate\_TAC.temp\_count)

generate\_TAC.temp\_count += 1

print(f"{result} = {a} {op} {b}")

return result

@staticmethod

def infix\_to\_tac(exp):

operators = []

values = []

i = 0

while i < len(exp):

c = exp[i]

if c == ' ':

i += 1

continue

if c.isalnum():

sbuf = ''

while i < len(exp) and (exp[i].isalnum() or exp[i] == '\_'):

sbuf += exp[i]

i += 1

values.append(sbuf)

i -= 1

elif c == '(':

operators.append(c)

elif c == ')':

while operators and operators[-1] != '(':

val2 = values.pop()

val1 = values.pop()

op = operators.pop()

values.append(generate\_TAC.apply\_op(op, val1, val2))

operators.pop()

else:

while operators and generate\_TAC.precedence(operators[-1]) >= generate\_TAC.precedence(c):

val2 = values.pop()

val1 = values.pop()

op = operators.pop()

values.append(generate\_TAC.apply\_op(op, val1, val2))

operators.append(c)

i += 1

while operators:

val2 = values.pop()

val1 = values.pop()

op = operators.pop()

values.append(generate\_TAC.apply\_op(op, val1, val2))

print("Result =", values.pop())

@staticmethod

def main():

exp = input("Enter an infix expression: ")

print("Infix Expression:", exp)

print("Generated Three Address Code:")

generate\_TAC.infix\_to\_tac(exp)

if \_\_name\_\_ == "\_\_main\_\_":

generate\_TAC.main()