CSPC62 – Compiler Design Assignment

Semester: Sixth

Programming Language

Team Members:

106120011 – Amarjit M

106120031 – Dharanish Rahul S

106120069 – Mithilesh K

106120149 – Yuvaraj A

CSPC62 – Compiler Design Assignment - 01

Dates: 16/02/2023 & 02/03/2023

Lexical Analyzer

Team Members:

106120011 – Amarjit M – Language Tokens

106120031 – Dharanish Rahul S – Regular expression

106120069 – Mithilesh K – NFA/DFA

106120149 – Yuvaraj A – LEX code

A) Components of Language:

We have developed a Language using Morse code for keywords and datatypes.

B) Regular expression for tokens:

Refer the Attachment.

C & D) NFA & DFA for the patterns:

Refer the attachment.

E & F) Lex Code for implementing patterns and handling errors:

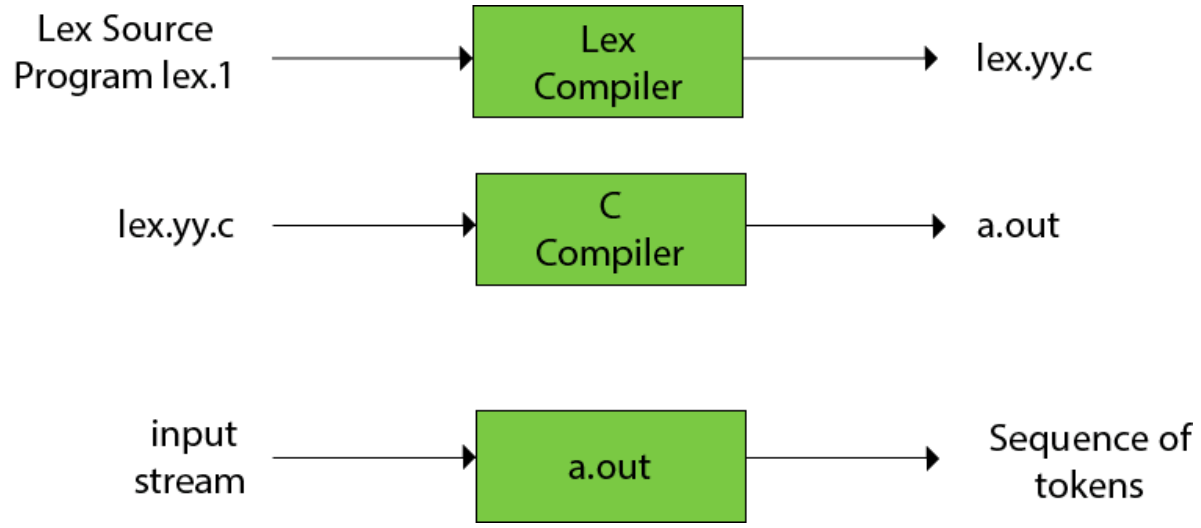
#LEX CODE

```
%{
    #include<string.h>
    #include<stdio.h>
    #define YYSTYPE char *
    extern int id_n=0;
    extern int key_n=0;
    extern int op= 0;
    extern int num_n=0;
%}

letter [a-zA-Z]
digit [0-9]

%%

[ \t]                ;
[ \n\n]   { yylineno = yylineno + 1;}

"..--."            {key_n++;printf("%s - REAL --- KEYWORD\n\n",
yytext);yylval="real";return REAL;}
"..---"            {key_n++;printf("%s - CHAR--- KEYWORD\n\n",
yytext);yylval="char";return CHAR;}
".-..-"         {key_n++;printf("%s - STRING--- KEYWORD\n\n",
yytext);yylval="string";return STRING;}
"---.."                {key_n++;printf("%s - BOOL--- KEYWORD\n\n",
yytext);yylval="bool"; return BOOL;}

"true"                                      {key_n++;printf("%s - TRUE_---
KEYWORD\n\n", yytext);yylval="true"; return TRUE_;}
"false"                                      {key_n++;printf("%s - FALSE_---
KEYWORD\n\n", yytext);yylval="false"; return FALSE_;}


".-.-."         {key_n++;printf("%s - BREAK---
KEYWORD\n\n",yytext);yylval="break";return BREAK;}
".----" {key_n++;printf("%s - VOID--- KEYWORD\n\n", yytext);yylval="void";return
VOID;}
"..-.."        {key_n++;printf("%s - WHILE--- KEYWORD\n\n",
yytext);yylval="while";return WHILE;}
"....-"        {key_n++;printf("%s - IF--- KEYWORD\n\n", yytext);yylval="if";return
IF;}
"...--"   {key_n++;printf("%s - FOR--- KEYWORD\n\n", yytext);yylval="for";return
FOR;}
"...-."                {key_n++;printf("%s - ELSE--- KEYWORD\n\n",
yytext);yylval="else";return ELSE;}
```

```
".--.."    {key_n++;printf("%s - DO--- KEYWORD\n\n", yytext);yylval="do";return
DO;}

"AS"     {key_n++;printf("%s - ASGN--- KEYWORD\n\n", yytext);yylval="assign";
return ASGN;}
"MAIN" {key_n++;printf("%s - MAIN--- KEYWORD\n\n", yytext);return MAIN;}
"DECLARE" {key_n++; printf("%s - DECLARE--- KEYWORD\n\n", yytext);yylval="declare";
return DECLARE;}
"SET" {key_n++;printf("%s - SET--- KEYWORD\n\n", yytext);yylval="set"; return SET;}


"-..."   {op++;printf("%s - LE--- OPERATOR\n\n", yytext);yylval="<=";return LE;}
".---"    {op++;printf("%s - GE--- OPERATOR\n\n",yytext);yylval=">=";return GE;}
"...."    {op++;printf("%s - EQ--- OPERATOR\n\n", yytext);yylval="==";return EQ;}
"--.."    {op++;printf("%s - NE--- OPERATOR\n\n", yytext);yylval="!=";return NE;}
"-..-"        {op++;printf("%s - GT--- OPERATOR\n\n", yytext);yylval=">";return
GT;}
"-.-."        {op++;printf("%s - LT--- OPERATOR\n\n", yytext);yylval="<";return
LT;}
"---."                  {op++;printf("%s - LOR--- OPERATOR\n\n",
yytext);yylval="||";return LOR;}
"----"          {op++;printf("%s - LAND--- OPERATOR\n\n",
yytext);yylval="&&";return LAND;}
"-.--"            {op++;printf("%s - LNOT--- OPERATOR\n\n", yytext);yylval="NOT";
return LNOT;}
"---."         {op++;printf("%s - BOR--- OPERATOR\n\n", yytext);yylval="|";return
BOR;}
"--.-"        {op++;printf("%s - BAND--- OPERATOR\n\n", yytext);yylval="&";return
BAND;}


"---"                         {op++;printf("%s - ADD--- OPERATOR\n\n",
yytext);yylval = '+'; return ADD;}
"--."                         {op++; printf("%s - SUB--- OPERATOR\n\n",
yytext);yylval = "-"; return SUB;}
".-."                         {op++; printf("%s - BXOR--- OPERATOR\n\n",
yytext);yylval = "xor"; return BXOR;}
"-."                          {op++;printf("%s - MULTI--- OPERATOR\n\n",
yytext);yylval = "*"; return MULTI;}
".--"                         {op++;printf("%s - DIV--- OPERATOR\n\n",
yytext);yylval = "/"; return DIV;}
"-.."                          {op++;printf("%s - MODULUS--- OPERATOR\n\n",
yytext);yylval = "%%"; return MODULUS;}
"..."          { op++; printf("%s - POWER--- OPERATOR\n\n", yytext);yylval = "^";
return POWER;}

{digit}+                        {
                                    int i,var=0,j=0;
                                    for(i=yyleng-1;i>=0;i--)
                                   {
                                       va=var+(*(yytext+j)48)*pow(10,i);
                                       j++;
                                   }
                                    yylval = var;
                                    printf("%s - NUM--- LITERAL\n\n", yytext);
                                    num_n++;
                                    return NUM;
                                }
```

```
{letter}({letter}|{digit})*       {
                                    yylval=strdup(yytext);
                                    id_n++;
                                     printf("%s - ID---IDENTIFIER\n\n", yytext);
                                    return ID;
                                  }


[\']{letter}[\']                {printf("%s - CHAR_CONSTANT---LITERAL\n\n",
yytext);return CHAR_CONSTANT;}

[\"]{letter}*[\"]                {printf("%s - CONSTANT---LITERAL\n\n",
yytext);return CONSTANT;}

"//"(.)*[\n\n]                        {printf("single-line comment\n\n");}
"/*"(.|\n\n)*"*/"                       {printf("multiline comment\n\n");}

\/\/.* ;
\/\*(.*\n\n)*.*\*\/ ;
.       {printf(%s – Invalid Token”, yytext); return yytext[0];}
%%


int main(){
    yyin = fopen("testa.txt", "r");
    yylex();
    return 0;
}
```

G) Compiling Lex Code:



1) Firstly, lexical analyser creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
2) Finally, C compiler runs the lex.yy.c program and produces an object program a.out.
3) a.out is lexical analyser that transforms an input stream into a sequence of tokens.

H) Program in our language:

Sample program in our language with error/without error:

```
    MAIN
$
  DECLARE ..--.  a, b, c;
  SET a AS 10;
  SET b AS 5;
  SET c AS 0;

  DECLARE ..--. i;
  SET i AS 0;

  ..-..(i -.-. 5)
  $
    SET c AS a --- b --- c;
    SET a AS a --. 1;
    SET b AS b --- 1;
  $

  ....-(c .--- 10)
  $
    SET c AS 0;
  $
  ...-.
  $
    SET c AS 1;
  $

DECLARE  ..--. errorId;
SET c AS 90s;

$
```

O/P:

```
    yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ lex program.l
program.l:51: warning, rule cannot be matched
program.l:91: warning, rule cannot be matched
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ yacc -d parser.y
parser.y: warning: 38 shift/reduce conflicts [-Wconflicts-sr]
parser.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ gcc y.tab.c -ll -lm -w
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ ./a.out


Lexeme    |    Token    |  General Name  |

MAIN - MAIN--- KEYWORD

DECLARE - DECLARE--- KEYWORD

..--. - REAL --- KEYWORD

a - ID---IDENTIFIER
```

```
b - ID---IDENTIFIER

c - ID---IDENTIFIER

SET - SET--- KEYWORD

a - ID---IDENTIFIER

AS - ASGN--- KEYWORD

10 - NUM--- LITERAL

SET - SET--- KEYWORD

b - ID---IDENTIFIER

AS - ASGN--- KEYWORD

5 - NUM--- LITERAL

SET - SET--- KEYWORD

c - ID---IDENTIFIER

AS - ASGN--- KEYWORD

0 - NUM--- LITERAL

DECLARE - DECLARE--- KEYWORD

..--. - REAL --- KEYWORD

i - ID---IDENTIFIER

SET - SET--- KEYWORD

i - ID---IDENTIFIER

AS - ASGN--- KEYWORD

0 - NUM--- LITERAL

..-.. - WHILE--- KEYWORD

i - ID---IDENTIFIER

-.-. - LT--- OPERATOR

5 - NUM--- LITERAL

SET - SET--- KEYWORD

c - ID---IDENTIFIER

AS - ASGN--- KEYWORD

a - ID---IDENTIFIER
```

```
--- - ADD--- OPERATOR

b - ID---IDENTIFIER

--- - ADD--- OPERATOR

c - ID---IDENTIFIER

SET - SET--- KEYWORD

a - ID---IDENTIFIER

AS - ASGN--- KEYWORD

a - ID---IDENTIFIER

--. - SUB--- OPERATOR

1 - NUM--- LITERAL

SET - SET--- KEYWORD

b - ID---IDENTIFIER

AS - ASGN--- KEYWORD

b - ID---IDENTIFIER

--- - ADD--- OPERATOR

1 - NUM--- LITERAL

....- - IF--- KEYWORD

c - ID---IDENTIFIER

.--- - GE--- OPERATOR

10 - NUM--- LITERAL

SET - SET--- KEYWORD

c - ID---IDENTIFIER

AS - ASGN--- KEYWORD

0 - NUM--- LITERAL

...-. - ELSE--- KEYWORD

SET - SET--- KEYWORD

c - ID---IDENTIFIER

AS - ASGN--- KEYWORD

1 - NUM--- LITERAL

DECLARE – DECLARE--- KEYWORD
```

```
..--. - REAL --- KEYWORD

errorId – Invalid

SET - SET--- KEYWORD

c - ID---IDENTIFIER

AS - ASGN--- KEYWORD

90s – Invalid Token


Number of identifiers  = 20
Number of keywords  = 26
Number of operators  = 6
Number of numbers  = 10
total number of tokens  = 62
Number of lines = 28
```

##Explanation:

The Lex code contains regular definitions, translational rules, and auxiliary functions. The first part of the Lex code includes the regular definitions for digits, letters. The second part of the code includes translation rules. Translation rules are that which implement these regular definitions to define tokens present in the program, tokens for TRUE, WHILE, IF, FOR, VOID, BREAK, DO, ASGN etc. are defined in this part of the program. The third part of the program includes auxiliary function. This part also includes error handling through the yyerror() function. This function helps detect error. It also returns the line number and kind of error detected.

CSPC62 – Compiler Design Assignment - 02

Dates: 30/03/2023

Syntax Analyzer (Parser)

Team Members:

106120011 – Amarjit M – Language Structure

106120031 – Dharanish Rahul S – Production and parsing

106120069 – Mithilesh K – Execution(lex and yacc)

106120149 – Yuvaraj A – Execution (lex and yacc)

Parse code:

#LEX CODE:

```
%{
   #include<string.h>
   #include<stdio.h>
   #define YYSTYPE char *
%}

letter [a-zA-Z]
digit [0-9]

%%

[ \t]                   ;
[ \n]   { yylineno = yylineno + 1;}

"..--."         {yylval="real";return REAL;}
"..---"         {yylval="char";return CHAR;}
".-..-"     {yylval="string";return STRING;}
"---.."                 {yylval="bool"; return BOOL;}

"true"                                  {yylval="true"; return TRUE_;}
"false"                                 {yylval="false"; return FALSE_;}


".-.-."     {yylval="break";return BREAK;}
".----" {yylval="void";return VOID;}
"..-.."     {yylval="while";return WHILE;}
"....-"     {yylval="if";return IF;}
"...--"   {yylval="for";return FOR;}
"...-."                 {yylval="else";return ELSE;}
".--.."    {yylval="do";return DO;}

"AS"     {yylval="assign"; return ASGN;}
"MAIN" {return MAIN;}
"DECLARE" {yylval="declare"; return DECLARE;}
"SET" {yylval="set"; return SET;}


"-..."   {yylval="<=";return LE;}
".---"    {yylval=">=";return GE;}
"...."    {yylval="==";return EQ;}
"--.."    {yylval="!=";return NE;}
"-..-"      {yylval=">";return GT;}
"-.-."      {yylval="<";return LT;}
"---."                  {yylval="||";return LOR;}
```

```
"----"    {yylval="&&";return LAND;}
"-.--"      {yylval="NOT"; return LNOT;}
"---."    {yylval="|";return BOR;}
"--.-"    {yylval="&";return BAND;}


"---"                   {yylval = '+'; return ADD;}
"--."                   {yylval = "-"; return SUB;}
".-."                   {yylval = "xor"; return BXOR;}
"-.-"                   {yylval = "*"; return MULTI;}
".--"                   {yylval = "/"; return DIV;}
"-.."                   {yylval = "%%"; return MODULUS;}
"..."       { yylval = "^"; return POWER;}

{digit}+                {
                        int i,var=0,j=0;
                                                for(i=yyleng-1;i>=0;i--)
                                                {
                                                   var = var +(*(yytext+j)-48)*pow(10,i);
                                                   j++;
                                                }
                                                yylval = var;
                        return NUM;
                      }

{letter}({letter}|{digit})*     {
                        yylval=strdup(yytext);
                        return ID;
                      }


[\']{letter}[\']        {return CHAR_CONSTANT;}
[\"]{letter}*[\"]       {return CONSTANT;}
"//"(.)*[\n]                    {printf("single-line comment\n");}
"/*"(.|\n)*"*/"                 {printf("multiline comment\n");}

\/\/.* ;
\/\*(.*\n)*.*\*\/ ;
.     return yytext[0];
%%
```

Yacc file:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#define YYSTYPE char *
extern char* yytext;

extern FILE *inFile;
FILE * outFile;

%}

%token REAL CHAR STRING BOOL VOID
%token WHILE FOR DO
%token IF ELSE BREAK
%token NUM ID
%token MAIN DECLARE SET
%token ADD SUB MULTI DIV POWER TRUE_ FALSE_ MODULUS CHAR_CONSTANT CONSTANT


%right ASGN
%left LOR
%left LAND
%left BOR
%left BXOR
%left BAND
%left LNOT
%left EQ NE
%left LE GE LT GT
%left ADD SUB
%left MULTI DIV MODULUS
%right POWER
%right UMINUS

%%

start_program          : declare_statement MAIN statement_start {printf("start_program\n");}
                       ;

statement_start        : '$' statement_body '$'    {printf("statement_start\n");}
          |
                       ;

statement_body                 : statement statement_body    {printf("statement_body\n");}
                       |
                       ;

statement              : declare_statement     {printf("statement\n");}
                       | assign_statement
                       | if_statement
                       | while_statement
                       | for_statement
                       | do_statement
                       | ';'
```

```
                                        ;

        for_statement           :  FOR '(' assign_statement_2 ';'  expression ';'  assign_statement_2 ')' whilebody
{printf("for_statement\n");}
                                        ;

        do_statement       : DO  do_while_body  WHILE '(' expression  ')' ';'       {printf("do_while_statement\n");}
                ;

        if_statement            : IF '(' expression ')'  statement_start else_statement
{printf("ifelse_statement\n");}
                                        ;
        else_statement          : ELSE  statement_start
                                        |
                                        ;

        while_statement                 : WHILE '(' expression ')'  whilebody         {printf("while_statement\n");}
                                        ;

        do_while_body   : statement_start     {printf("doWhile_body\n");}
                |statement
                ;

        whilebody               : statement_start         {printf("whilebody\n");}
                                        | statement

                                        ;

        declare_statement       : DECLARE TYPE   ID  ids       {printf("declare_statement\n"); countVars();}
                                        |
                                        ;

        ids                     : ';'
                                        | ','  ID {countVars();} ids
                                        ;

        assign_statement                : SET ID ASGN    expression        ';'       {printf("assignment_statement\n");}

                ;

        assign_statement_2      : SET ID  ASGN    expression        {printf("assignment_statement_2\n");}
                ;

        expression                      : expression ADD          expression       {printf("arithmetic_ADD\n");}
                                        | expression SUB   expression            {printf("arithmetic_SUB\n");}
                                        | expression MULTI   expression          {printf("arithmetic_MULTI\n");}
                                        | expression DIV  expression             {printf("arithmetic_DIVISION\n");}
                                        | expression MODULUS  expression
{printf("arithmetic_MODULUS\n");}
                                        | expression POWER   expression          {printf("POWER\n");}
```

```
        | expression  LT   expression          {printf("relop_LT\n");}
                        | expression  LE   expression          {printf("relop_LE\n");}
                        | expression  GT   expression          {printf("relop_GT\n");}
                        | expression  GE   expression          {printf("relop_GE\n");}
                        | expression  NE   expression          {printf("relop_NE\n");}
                        | expression  EQ   expression          {printf("relop_EQ\n");}
        | expression   LOR  expression          {printf("logical_OR\n");}
                        | expression   LAND expression          {printf("logical_AND\n");}
                        | expression   BOR  expression          {printf("logical_BITWISE_OR\n");}
                        | expression   BXOR expression          {printf("logical_BITWISE_XOR\n");}
                        | expression   BAND expression          {printf("logical_BITWISE_AND\n");}
                        | '-' expression %prec UMINUS          {printf("Unary_MINUS\n");}
                        | LNOT expression                   {printf("logical_NOT\n");}
                        | '(' expression ')'                {printf("(expr)\n");}
                        | expression '+' '(' '-'  expression ')'
                        | ID  {$$=$1;  }                    {printf("Identifier\n");}
                        | NUM {$$=$1; }                     {printf("Number\n");}
                        ;



TYPE                    : REAL  {printf("dataType_REAL\n");}
                        |CHAR   {printf("dataType_CHAR\n");}
                        |STRING    {printf("dataType_STRING\n");}
                        |BOOL       {printf("dataType_BOOL");}
                        ;


%%


#include"lex.yy.c"

int varCount = 0;

int main(int argc, char *argv[])
{
        yyin = fopen(argv[1], "r");
        outFile=fopen("result.txt","w");

   printf("Parsing levels:----------------------------------------\n\n\n");
  if(!yyparse())
                printf("\nParsing successful:)--------------------------------\n");
        else
        {
                printf("\nParsing unsuccessful:(\n");
                exit(0);
        }

   printf("Number of variables declarations = %d\n", varCount);
```

```
    printf("Number of lines = %d\n", yylineno);

            fclose(yyin);
            fclose(outFile);

    return 0;
}

yyerror(char *s) {
            printf("Syntex Error in line number : %d : %s %s\n", yylineno, s, yytext );
}

void countVars(){
    varCount++;
}
```

Input/ouput: ##INPUT/OUTPUT for sample code:

```
MAIN
$
  DECLARE ..--.  a, b, c;
  SET a AS 10;
  SET b AS 5;
  SET c AS 0;

  DECLARE ..--. i;
  SET i AS 0;

  ..-..(i -.-. 5)
  $
    SET c AS a --- b --- c;
    SET a AS a --. 1;
    SET b AS b --- 1;
  $

  ....-(c .--- 10)
  $
    SET c AS 0;
  $
  ...-.
  $
    SET c AS 1;
  $

$

/* multiline comment used for better understanding of the morse code usage
MAIN
$
```

```
  DECLARE real  a, b, c;
  SET a AS 10;
  SET b AS 5;
  SET c AS 0;

  DECLARE real i;
  SET i AS 0;

  while(i < 5)
  $
    SET c AS a + b + c;
    SET a AS a - 1;
    SET b AS b + 1;
  $

  ifc >= 10)
  $
    SET c AS 0;
  $
  else
  $
    SET c AS 1;
  $

$
*/
```

Output:

```
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ ./a.out testa.txt
Parsing levels:---------------------------------------


dataType_REAL
declare_statement
statement
Number
assignment_statement
Number
assignment_statement
Number
assignment_statement
dataType_REAL
declare_statement
statement
Number
assignment_statement
Identifier
Number
relop_LT
Identifier
Identifier
arithmetic_ADD
Identifier
arithmetic_ADD
assignment_statement
Identifier
Number
arithmetic_SUB
assignment_statement
Identifier
```

```
Number
arithmetic_ADD
assignment_statement
statement_body
statement_body
statement_body
statement_start
whilebody
while_statement
Identifier
Number
relop_GE
Number
assignment_statement
statement_body
statement_start
Number
assignment_statement
statement_body
statement_start
ifelse_statement
statement_body
statement_body
statement_body
statement_body
statement_body
statement_body
statement_body
statement_body
statement_start
start_program

Parsing successful:)---------------------------------
Number of variables declarations = 4
Number of lines = 28
```

##1 INPUT/OUTPUT for error handling:

I/P:

```
MAIN
$
   DECLARE ..--.  4q, b, c; //Error: invalid identifier
   SET a AS 10;
   SET b AS 5;
   SET c AS 0;
$
----
```

o/p:

```
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ ./a.out testa.txt
Parsing levels:--------------------------------------


dataType_REAL
Syntex Error in line number : 3 : syntax error 4

Parsing unsuccessful:(
```

##2 INPUT/OUTPUT for error handling:

I/P:

```
MAIN
$
   DECLARE ..--.  a, b, c;
   SET a AS 10;
   SET b AS 5;
   SET c AS 0;
```

```
   DECLARE ..---. i;
   i AS 0; //Error: SET is missing

$
```

O/P:

```
---
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ ./a.out testa.txt
Parsing levels:--------------------------------------


dataType_REAL
declare_statement
statement
Number
assignment_statement
Number
assignment_statement
Number
assignment_statement
dataType_REAL
declare_statement
statement
statement_body
statement_body
statement_body
statement_body
statement_body
Syntex Error in line number : 9 : syntax error i

Parsing unsuccessful:(
```

Parse Tree:

Refer the attachment.

##Explanation:

The Lex code contains regular definitions, translational rules, and auxiliary functions.

This lex code also contains a link to the Yacc file. The first part of the Lex code includes the regular definitions for digits, letters. The second part of the code includes translation rules. Translation rules are that which implement these regular definitions to define tokens present in the program, tokens for TRUE, WHILE, IF, FOR, VOID, BREAK, DO, ASGN etc. are defined in this part of the program. The third part of the program includes auxiliary function. This part also includes error handling through the yyerror() function. This function helps detect error. It also returns the line number and kind of error detected. This lex file also includes a link to the Yacc file using the code #include “y.tab.h”

The Yacc code includes regular definitions and tokens that are to be used in the code and since error detection is done through the Yacc code, it has to have a variable declared “ErrorRecovered” to return the count and type of error that the code detects in the input file. The second part of the Yacc file consists of production rules/grammar that defines the language that is being parsed. These production rules define the way of constructing and executing the parse tree which then allows the program to detect error if any. The last part of the code consists of the supporting sub-routines, the error detecting part of the language using the yyerror() function. Also present is the main() function that prints the outputs from these sub-routines.

CSPC62 – Compiler Design Assignment - 03

Dates: 06/04/2023

Three Address Code generation

Team Members:

106120011 – Amarjit M – Applied the concept of backpatching

106120031 – Dharanish Rahul S – parse tree with sdt, 3-ADC

106120069 – Mithilesh K – Tested arbitrary code & symbol table

106120149 – Yuvaraj A – implemented 3-ADC & symbol table, parse tree

Code:

##LEX CODE:

```
%{
   #include<string.h>
   #include<stdio.h>
   #define YYSTYPE char *
%}

letter [a-zA-Z]
digit [0-9]

%%

[ \t]                   ;
[ \n]   { yylineno = yylineno + 1;}

"..--."         {yylval="real";return REAL;}
"..---"         {yylval="char";return CHAR;}
".-..-"     {yylval="string";return STRING;}
"---.."                 {yylval="bool"; return BOOL;}

"true"                                  {yylval="true"; return TRUE_;}
"false"                                 {yylval="false"; return FALSE_;}


".-.-."     {yylval="break";return BREAK;}
".----" {yylval="void";return VOID;}
"..-.."     {yylval="while";return WHILE;}
"....-"     {yylval="if";return IF;}
"...--"   {yylval="for";return FOR;}
"...-."                 {yylval="else";return ELSE;}
".--.."    {yylval="do";return DO;}

"AS"     {yylval="assign"; return ASGN;}
"MAIN" {return MAIN;}
"DECLARE" {yylval="declare"; return DECLARE;}
"SET" {yylval="set"; return SET;}


"-..."   {yylval="<=";return LE;}
".---"    {yylval=">=";return GE;}
"...."    {yylval="==";return EQ;}
"--.."    {yylval="!=";return NE;}
"-..-"      {yylval=">";return GT;}
"-.-."      {yylval="<";return LT;}
"---."                  {yylval="||";return LOR;}
```

```
"----"    {yylval="&&";return LAND;}
"-.--"      {yylval="NOT"; return LNOT;}
"---."    {yylval="|";return BOR;}
"--.-"    {yylval="&";return BAND;}


"---"                   {yylval = '+'; return ADD;}
"--."                   {yylval = "-"; return SUB;}
".-."                   {yylval = "xor"; return BXOR;}
"-.-"                   {yylval = "*"; return MULTI;}
".--"                   {yylval = "/"; return DIV;}
"-.."                   {yylval = "%%"; return MODULUS;}
"..."       { yylval = "^"; return POWER;}

{digit}+                {
                        int i,var=0,j=0;
                                                for(i=yyleng-1;i>=0;i--)
                                                {
                                                   var = var +(*(yytext+j)-48)*pow(10,i);
                                                   j++;
                                                }
                                                yylval = var;
                        return NUM;
                      }

{letter}({letter}|{digit})*     {
                        yylval=strdup(yytext);
                        return ID;
                      }



[\']{letter}[\']            {return CHAR_CONSTANT;}
[\"]{letter}*[\"]           {return CONSTANT;}
"//"(.)*[\n]                    {printf("single-line comment\n");}
"/*"(.|\n)*"*/"                 {printf("multiline comment\n");}

\/\/.* ;
\/\*(.*\n)*.*\*\/ ;
.     return yytext[0];
%%
```

YACC CODE:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>

#define YYSTYPE char *
extern char* yytext;

extern FILE *inFile;
FILE * outFile;

%}

%token REAL CHAR STRING BOOL VOID
%token WHILE FOR DO
%token IF ELSE BREAK
%token NUM ID
%token MAIN DECLARE SET
%token ADD SUB MULTI DIV POWER TRUE_ FALSE_ MODULUS CHAR_CONSTANT CONSTANT
%right ASGN
%left LOR
%left LAND
%left BOR
%left BXOR
%left BAND
%left LNOT
%left EQ NE
%left LE GE LT GT
%left ADD SUB
%left MULTI DIV MODULUS
%right POWER
%right UMINUS

%%
start_program           :  MAIN  statement_start
                        ;
statement_start         : '$' statement_body '$'|
                        ;

statement_body                  : statement  statement_body
                        |
                        ;
statement               : declare_statement
                        | assign_statement
                        | if_statement
                        | while_statement
                        | for_statement
```

```
                                        | do_statement
                                        | ';'
                                        ;
        for_statement           : {forStart();} FOR '(' assign_statement_2 ';' {forMiddle();} expression ';'
{forRepetition();} assign_statement_2')' whilebody
                                        ;
do_statement      : {whileStart();} DO  do_while_body  WHILE '(' expression {whileRepetition();} ')' ';' {whileEnd();}
                ;
        if_statement            : IF '(' expression ')'  {ifLabel();} statement_start else_statement
                                        ;
        else_statement          : ELSE {elseLabel();} statement_start {ifElseEndLabel();}
                                        |       {ifElseEndLabel();}
                                        ;
        while_statement                 :{ whileStart();} WHILE '(' expression ')' {whileRepetition();} whilebody
                                        ;
        do_while_body   : statement_start
                        |statement
                        ;

        whilebody               : statement_start {whileEnd();}
                                        | statement {whileEnd();}


                                        ;
        declare_statement       : DECLARE TYPE {setDatatype($2);}  ID {insertIntoSymbolTable("identifier");} ids
                                        ;
        ids                     : ';'
                                        | ','  ID {insertIntoSymbolTable("identifier");} ids
                                        ;
        assign_statement                : SET ID {pushIdNum(); invokeId();}      ASGN    {pushop("=");}   expression
        { codeGenerateAssign();} ';'

                ;
        assign_statement_2      : SET ID  {pushIdNum(); invokeId();}      ASGN    {pushop("=");}   expression
        {codeGenerateAssign();}
                ;
        expression                      : expression ADD        {pushop("+");}  expression  {generateAlgebric(); }
                                        | expression SUB {pushop("-");}  expression  {generateAlgebric();}
                                        | expression MULTI {pushop("*");}  expression  {generateAlgebric();}
                                        | expression DIV {pushop("/");}  expression  {generateAlgebric();}
                                        | expression MODULUS {pushop("%");}  expression  {generateAlgebric();}
                                        | expression POWER {pushop("^");}  expression  {generateAlgebric();}
                | expression  LT {pushop("<");}  expression  {generateLogical();}
                                        | expression  LE {pushop("<=");}  expression  {generateLogical();}
                                        | expression  GT {pushop(">");}  expression  {generateLogical();}
                                        | expression  GE {pushop(">=");}  expression  {generateLogical();}
                                        | expression  NE {pushop("!=");}  expression  {generateLogical();}
                                        | expression  EQ {pushop("==");}  expression  {generateLogical();}
                | expression  {pushop("||");}  LOR  expression  {generateLogical();}
                                        | expression  {pushop("&&");}  LAND expression  {generateLogical();}
                                        | expression  {pushop("|");}  BOR  expression  {generateLogical();}
```

```
                                | expression  {pushop("XOR");}  BXOR expression  {generateLogical();}
                                | expression  {pushop("&");}  BAND expression  {generateLogical();}
                                | '-' expression %prec UMINUS {generateUnaryMinus();}
                                | LNOT expression  {lNOT_func();}
                                | '(' expression ')'
                                | expression '+' '(' '-' {pushop('-');} expression ')' {generateAlgebric(); }
                                | ID  {$$=$1; check(); pushIdNum();  }
                                | NUM {$$=$1; pushIdNum();  mapNum($1); }
                                ;
        TYPE                    : REAL
                                |CHAR
                                |STRING
                                |BOOL
                                ;


%%
#include"lex.yy.c"
int count=0;

char stack[1000][10];
int top=0;
int i=0;
char temp[2]="t";
int label[200];
int labelNumber=0;
int labelTop=0;
int stop=0;
char type[10];
char* tempId;
struct Table
{
        char id[20];
        char type[10];
        char category[15];
        int value;
        int lineno;
}symbolTable[10000];
int tableLength=0;

int main(int argc, char *argv[])
{
        yyin = fopen(argv[1], "r");
        outFile=fopen("result.txt","w");

  if(!yyparse())
                printf("\nParsing successful:)\n");
        else
        {
                printf("\nParsing unsuccessful:(\n");
```

```
                exit(0);
        }
        printf("-----------------------------------------------------------------------------------------------------\n");
        printf("| SNo. | name | Datatype | Category | Value | Lineno |\n");
        printf("-----------------------------------------------------------------------------------------------------\n");
        for(int i=0; i<tableLength; i++){
        printf("| %d | %s | %s | %s | %d | %d |\n", i+1,
symbolTable[i].id, symbolTable[i].type, symbolTable[i].category, symbolTable[i].value, symbolTable[i].lineno);
        }
        printf("-----------------------------------------------------------------------------------------------------\n");
        fclose(yyin);
        fclose(outFile);

    return 0;
}
yyerror(char *s) {
        printf("Syntex Error in line number : %d : %s %s\n", yylineno, s, yytext );
}
//pushingid and numbers into stack for 3-address code generation
pushIdNum()
{
        strcpy(stack[++top],yytext);
}

//function for pushing operatos into stack
pushop(char* optr)
{
        strcpy(stack[++top],optr);
}
//for logical expression translation
generateLogical()
{
        sprintf(temp,"t%d",i);
        fprintf(outFile,"\t%s=%s%s%s\n",temp,stack[top-2],stack[top-1],stack[top]);
        top-=2;
        strcpy(stack[top],temp);
        i++;
}

//for arithmetic expression translation
generateAlgebric()
{
        sprintf(temp,"t%d",i); // converts temp to reqd format
        fprintf(outFile,"\t%s=%s%s%s\n",temp,stack[top-2],stack[top-1],stack[top]);
        top-=2;
        strcpy(stack[top],temp);
        i++;
}
//for assignment statement translation
codeGenerateAssign()
```

```
{
	fprintf(outFile,"\t%s=%s\n",stack[top-2],stack[top]);
	top-=3;
}
//for unary minus expression translation
generateUnaryMinus()
{
	sprintf(temp,"t%d",i);
	fprintf(outFile,"\t%s=-%s\n",temp,stack[top]);
	strcpy(stack[top],temp);
	i++;
}
//for logical NOT translation
lNOT_func()
{
	sprintf(temp,"t%d",i);
	fprintf(outFile,"\t%s= not %s\n",temp,stack[top]);
	strcpy(stack[top],temp);
	i++;
}
//for first if block code translation
ifLabel()
{
	labelNumber++;
	fprintf(outFile,"\tif( not %s)",stack[top]);
	fprintf(outFile,"\tgoto L%d\n",labelNumber);
	label[++labelTop]=labelNumber;
}
//for else block translation
elseLabel()
{
	int x;
	labelNumber++;
	x=label[labelTop--];
	fprintf(outFile,"\tgoto L%d\n",labelNumber);
	fprintf(outFile,"L%d: \n",x);
	label[++labelTop]=labelNumber;
}
// translation for ifelse block ending
ifElseEndLabel()
{
	int y;
	y=label[labelTop--];
	fprintf(outFile,"L%d: \n",y);
	top--;
}
//for loop initial label code generation
forStart()
{
	labelNumber++;
```

```
	fprintf(outFile,"L%d:\n",labelNumber);
}

forMiddle()
{
	labelNumber++;
	label[++labelTop]=labelNumber;
	fprintf(outFile,"L%d:\n",labelNumber);
}
//loop statements code generation …with goto labels
forRepetition()
{
	labelNumber++;
	fprintf(outFile,"\tif( not %s)",stack[top]);
	fprintf(outFile,"\tgoto L%d\n",labelNumber);
	label[++labelTop]=labelNumber;
}
//initial code for while labels
whileStart()
{
	labelNumber++;
	label[++labelTop]=labelNumber;
	fprintf(outFile,"L%d:\n",labelNumber);
}
//inside while loop code generation
whileRepetition()
{
	labelNumber++;
	fprintf(outFile,"\tif( not %s)",stack[top]);
	fprintf(outFile,"\tgoto L%d\n",labelNumber);
	label[++labelTop]=labelNumber;



}
whileEnd()
{
	int x,y;  y=label[labelTop--];
	x=label[labelTop--];
	fprintf(outFile,"\tgoto L%d\n",x);
	fprintf(outFile,"L%d: \n",y);
	top--;
}
/*
 for symbol table generation
check whether the used identifier is already declared; if not throw error else continue ( for non-declaration
statements)
for declaration statements check – error is redeclaration)
*/
check()
```

```
{
	char temp[20];
	strcpy(temp,yytext);
	int flag=0;
	for(i=0;i<tableLength;i++)
	{
		if(!strcmp(symbolTable[i].id,temp))
		{
			flag=1;
			break;
		}
	}
	if(!flag)
	{
		yyerror("Variable is not declard");
		exit(0);
	}
}
//setting datatype for the identifier during declaration
setDatatype(char* t)
{
	strcpy(type,t);
}
//insertion into symbol table
insertIntoSymbolTable(char* category)
{
	char temp[20];
	int i,flag;
	flag=0;
	strcpy(temp,yytext);
	for(i=0;i<tableLength;i++)
	{
		if(!strcmp(symbolTable[i].id,temp))
			{
			flag=1;
			break;
				}
	}
	if(flag)
	{
		yyerror("Redeclare of ");
		exit(0);
	}
	else
	{
		strcpy(symbolTable[tableLength].id,temp);
		strcpy(symbolTable[tableLength].type,type);
		strcpy(symbolTable[tableLength].category,category);
		symbolTable[tableLength].lineno = yylineno;
		tableLength++;
```

```
        }
}
//update the value of identifier in symbol table when updated in program
void update(char *token,int value)
{
 int flag = 0;
 for(int i = 0;i < tableLength;i++)
 {
  if(!strcmp(symbolTable[i].id,token))
  {
   flag = 1;
   symbolTable[i].value = (int*)malloc(sizeof(int));
   symbolTable[i].value=value;
   return;
  }
 }
 if(flag == 0)
 {
  printf("Error at line %d : %s is not defined\n",8,token);
  exit(0);
 }
}

//for initialization of identifiers
void invokeId(){
        tempId = strdup(yytext);
}
//for initial assignment of identifiers
void mapNum(int val){
        update(tempId, val);
}
```

Input/Output:

Input-1:

```
MAIN
$
  DECLARE ..--.  a, b, c;
  SET a AS 10;
  SET b AS 5;
  SET c AS 0;

  DECLARE ..--. i;
  SET i AS 0;

  ..-..(i -.-. 5)
  $
```

```
    SET c AS a --- b --- c;
    SET a AS a --. 1;
    SET b AS b --- 1;
  $

  ....-(c .--- 10)
  $
    SET c AS 0;
  $
  ...-.
  $
    SET c AS 1;
  $

$

/* multiline comment used for better understanding of the morse code usage
MAIN
$
  DECLARE real  a, b, c;
  SET a AS 10;
  SET b AS 5;
  SET c AS 0;

  DECLARE real i;
  SET i AS 0;

  while(i < 5)
  $
    SET c AS a + b + c;
    SET a AS a - 1;
    SET b AS b + 1;
  $

  ifc >= 10)
  $
    SET c AS 0;
  $
  else
  $
    SET c AS 1;
  $

$
*/
```

OUTPUT-1:

```
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ ./a.out testa.txt
multiline comment

Parsing successful:)
-------------------------------------------------------------------------------------------------------------
|    SNo.    |       name       |       Datatype    |     Category      |       Value       |      Lineno      |
-------------------------------------------------------------------------------------------------------------
|     1      |        a         |      real         |       identifier          |      1       |        3         |
|     2      |        b         |      real         |       identifier          |      10      |        3         |
|     3      |        c         |      real         |       identifier          |      1       |        3         |
|     4      |        i         |      real         |       identifier          |      5       |        8         |
-------------------------------------------------------------------------------------------------------------
```

3-address code:

```
      a=10
        b=5
        c=0
        i=0
L1:
        t3=i<5
        if( not t3)        goto L2
        t1=a+b
        t2=t1+c
        c=t2
        t0=a-1
        a=t0
        t1=b+1
        b=t1
        goto L1
L2:
        t2=c>=10
        if( not t2)        goto L3
        c=0
        goto L4
L3:
        c=1
L4:
```

INPUT_2:

DECLARE ..--. a,b,c,d;

MAIN
$

..-..(a -.-. c ---- b -.-. c ---. c -..- d)

```
  $
    SET a AS b .-- c -.- d -.- (-c);
  $

  DECLARE ..--. i, total;
  SET total AS 10;

  ...--( SET i AS 20; i -..- 10; SET i AS i --. 1 )
  $
    SET total AS total -.- i;
  $

  DECLARE ..--. result;
  ....-(a -.-. total)
  $
    SET result AS 1;
  $
  ...-.
  $
    SET result AS 0;
  $

$
/* multiline comment used for better understanding of the morse code usage
DECLARE real a,b,c,d;

MAIN
$

  while(a < c && b < c || c > d)
  $
     a=b/c*d*(-c);
  $

  DECLARE real i, total;
  SET total AS 10;
  for(SET i AS 20; i > 10; SET i AS i - 1 )
  $
    SET total AS total * i;
  $

  DECLARE real result;
  if(a > total)
  $
    SET result AS 1;
  $
  else
  $
    SET result AS 0;
  $
```

```
      $
      */
```

OUTPUT-2:

```
      yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ ./a.out testb.txt
multiline comment

Parsing successful:)



--------------------------------------------------------------------------------
-------------------------
|    SNo.    |       name       |       Datatype    |     Category      |       Value
    |      Lineno      |
--------------------------------------------------------------------------------
-------------------------
|     1     |        a        |      real       |       identifier         |      0
     |       1          |
|     2     |        b        |      real       |       identifier         |      0
     |       1          |
|     3     |        c        |      real       |       identifier         |      0
     |       1          |
|     4     |        d        |      real       |       identifier         |      0
     |       1          |
|     5     |        i        |      real       |       identifier         |      1
     |       11         |
|     6     |        total        |      real       |       identifier         |
10         |       11         |
|     7     |        result        |      real       |       identifier         |
0         |       19         |
--------------------------------------------------------------------------------
-------------------------
3-address code:
      L1:
        t2=a<c
        t2=b<c
        t3=c>d
        t4=t2||t3
        t5=t2&&t4
        if( not t5)        goto L2
        t2=b/c
        t3=t2*d
        t2=-c
        t3=t3*t2
        a=t3
        goto L1
L2:
        total=10
L3:
        i=20
L4:
```

```
        t4=i>10
        if( not t4)        goto L5
        t4=i-1
        i=t4
        t4=total*i
        total=t4
        goto L4
L5:
        t5=a<total
        if( not t5)        goto L6
        result=1
        goto L7
L6:
        result=0

      L7:
```

##Explanation:

The Lex code contains regular definitions, translational rules, and auxiliary functions.

This lex code also contains a link to the Yacc file using the code #include “lex.yy.c”. The first part of the Lex code includes the regular definitions for digits, letters. The second part of the code includes translation rules. Translation rules are that which implement these regular definitions to define tokens present in the program, tokens for TRUE, WHILE, IF, FOR, VOID, BREAK, DO, ASGN etc. are defined in this part of the program. The third part of the program includes auxiliary function.

The Yacc code includes regular definitions and tokens that are to be used in the code and since error detection is done through the Yacc code, it has to have a variable declared “Error Recovered” to return the count and type of error that the code detects in the input file. The second part of the Yacc file consists of production rules/grammar that defines the language that is being parsed. These production rules define the way of constructing and executing the parse tree which then allows the program to detect error if any. The last part of the code consists of the supporting sub-routines, the error detecting part of the language using the yyerror() function. Also present is the main() function that prints the outputs from these sub-routines. We have also included the code for three address code generation and the symbol table productions. For each input we are given we will get the exact input tokens from the symbol table and the three address code for the given sample code will accurately generated.

Three address code generation:

We have used stack based storage and retrieval of identifiers, numbers (operands and operators) and their value and with the use of symbol table.

Here, specific functions are used for various program constructs to generate three address code and more specific functions during parsing of constructs like ifelse, while, for, do … with semantic actions done in between.

CSPC62 – Compiler Design Assignment - 04

Dates: 27/04/2023

Code optimization and DAG

Team Members:

106120011 – Amarjit M – Implemented Directed acyclic graph for the blocks.

106120031 – Dharanish Rahul S – implemented some types of code optimization techniques

106120069 – Mithilesh K – implemented common subexpression and flow of blocks

106120149 – Yuvaraj A – implemented the 3ADC to code optimization using common subexpression elimination

#Code:

```
#Common subexpression elimination
import re

inFile = open("result_IC.txt","r")

outFile = open("result_of_cse.txt", "w")

content = inFile.readlines()
subexpr_table=dict()



#Maintain 3 order tuple(op,op1,op2) in table. Store the temporary in temporaries list.
#if any of op1 or op2 is defined again later, remove the tuple from table.

for i in range(len(content)):
        if '=' in content[i] and not '==' in content[i]: #For cases like L1: t1=10
                Assignexpr = content[i].strip().split('=')
                variable=Assignexpr[0]  #variable holds the LHS value of assignment
                if ':' in Assignexpr[0]:
                        subexpr_table=dict()
                        lhs=Assignexpr[0].replace(" ","").split(":")
                        variable=lhs[1]

                var_list=re.split('\+|-|\*|/|%|>|<|>=|<=', Assignexpr[1])
                if len(var_list)==1:
                        found=0
                        temp=""
                        for key,value in subexpr_table.items():
                                for j in value:

                                        #one of the operands got redefined, so remove that expression
                                        if variable==j:
                                                found=1
                                                temp=key
```

```
                                        break
                if(found==1):
                        subexpr_table.pop(temp)
                outFile.write(content[i].strip() + '\n')

        if len(var_list)==2:
                tup=[]
                op1 = var_list[0]
                op2 = var_list[1]
                op=""
                flag=0
                line=content[i]
                if '+' in line:
                        op='+'
                if '-' in line:
                        op='-'
                if '*' in line:
                        op='*'
                if '/' in line:
                        op='/'
                if '<' in line:
                        op='<'
                if '>' in line:
                        op='>'
                if '<=' in line:
                        op='<='
                if '>=' in line:
                        op='>='
                if '%' in line:
                        op='%'
                tup=[op,op1,op2]

                for key,value in subexpr_table.items():
                        if tup==value: #common subexpression found
                                #Assignment[1] is RHS of assignment, so replace it with intermediate value
holding the value of CS
```

```
                    Assignexpr[1]=key
                    flag=1
            if(flag==0):
                #unique RHS,insert into subexpr table
                subexpr_table[variable]=tup

            outFile.write(str(Assignexpr[0])+'='+str(Assignexpr[1]) + '\n')

elif '==' in content[i]:
    spl=content[i].strip().split('==')
    if '=' in spl[0]:
        tup=[]
        #Assignexpr[0] is lhs, assignexpr[1] is first operand
        Assignexpr=spl[0].split('=')
        variable=Assignexpr[0]
        if ':' in Assignexpr[0]:
            lhs=Assignexpr[0].replace(" ","").split(":")
            variable=lhs[1]

        op1=Assignexpr[1]
        op2=spl[1]
        op="=="
        flag=0
        tup=[op,op1,op2]
        rhs=str(Assignexpr[1])+'=='+str(spl[1])

        for key,value in subexpr_table.items():
            if tup==value:  #Common subexpression found
                rhs=key
                flag=1
        if(flag==0):
            subexpr_table[variable]=tup #unique expression, so insert into subexpr table
        outFile.write(str(Assignexpr[0])+'='+rhs +'\n')
else:
    outFile.write(content[i].strip() + '\n')
```

```
print("Optimized with Common Subexpression Elimination...\nCheck the result in result_of_cse.txt file :)\n")

#constant folding
import re
import operator

def get_operator_fn(op):
   return {
      '+' : operator.add,
      '-' : operator.sub,
      '*' : operator.mul,
      '/' : operator.truediv,
      '%' : operator.mod,
      '^' : operator.xor,
      }[op]

def eval_binary_expr(op1, oper, op2):
   op1,op2 = int(op1), int(op2)
   return get_operator_fn(oper)(op1, op2)


inFile = open("result_of_cse.txt","r")
content = inFile.readlines()

outFile = open("result_of_folding.txt", "w")


#dictionary with key as variable and value as its constant
constant_table=dict()

for i in range(len(content)):

        if '=' in content[i] and not '==' in content[i]: #for case like L1: t1=10
                Assignexpr = content[i].strip().split('=')
                variable=Assignexpr[0]
```

```
if ':' in Assignexpr[0]:
        lhs=Assignexpr[0].replace(" ","").split(":")
        variable=lhs[1]
        constant_table={}

var_list=re.split('\+|-|\*|/|%', Assignexpr[1])
if len(var_list)==1: #pure assignment
        if var_list[0].isdigit():
                constant_table[variable]=var_list[0] #Case 2
        else:
                if var_list[0] in constant_table.keys():
                        Assignexpr[1]=constant_table[var_list[0]] #Case 1
        outFile.write(str(Assignexpr[0]) + '=' + str(Assignexpr[1]) + '\n')



#RHS contains multiple operands - 4 types
# Type 1 - op1 is digit op2 is digit
# Type 2 - op1 is digit op2 is variable
# Type 3 - op1 is variable op2 is digit
# Type 4 - op1 is variable op2 is variable

if len(var_list)==2: #Case 3
        constant_value="NOCHANGE"
        op1 = var_list[0]
        op2 = var_list[1]
        if '+' in content[i]:
                op='+'
        if '-' in content[i]:
                op='-'
        if '*' in content[i]:
                op='*'
        if '/' in content[i]:
                op='/'

        if op1.isdigit() and op2.isdigit():
                constant_value=eval_binary_expr(op1, op, op2)
```

```
                    constant_table[Assignexpr[0]]=constant_value

                if op1.isdigit() and op2.isdigit()!=1:
                    if op2 in constant_table.keys():
                        constant_value=eval_binary_expr(op1, op, constant_table[op2])
                        constant_table[Assignexpr[0]]=constant_value

                if op1.isdigit()!=1 and op2.isdigit():
                    if op1 in constant_table.keys():
                        constant_value=eval_binary_expr(constant_table[op1], op, op2)
                        constant_table[Assignexpr[0]]=constant_value

                if op1.isdigit()!=1 and op2.isdigit()!=1:
                    if op1 in constant_table.keys():
                        if op2 in constant_table.keys():
                            constant_value=eval_binary_expr(constant_table[op1], op,
constant_table[op2])
                            constant_table[Assignexpr[0]]=constant_value
                        else: #only op1 in constant table
                            Assignexpr[1]=str(constant_table[op1])+str(op)+str(op2)
                    elif op2 in constant_table.keys():
                        Assignexpr[1]=str(op1)+str(op)+str(constant_table[op2])

                if constant_value!="NOCHANGE":
                    Assignexpr[1]=constant_value
                outFile.write(str(Assignexpr[0]) + '=' + str(Assignexpr[1]) + '\n')

        elif ':' in content[i]:
            constant_table={}
            outFile.write(content[i])

        else:
            outFile.write(content[i].strip() + '\n')

print("Optimized with Constant Folding and Propagation...\nCheck the result in result_of_folding.txt file :)\n")
```

Sample Input/ output file:

Input.txt

```
MAIN
$
  DECLARE ..--.  a, b, c, d, e;
  SET a AS 10 --- 10;
  SET b AS a --- 10;
  SET c AS 20 -.- 3;
  SET d AS 25 --. 10;
  SET e AS c --- d;

  DECLARE ..--. i, j, k;
  SET i AS 0;

  ..-..(i -.-. 5)
  $
    SET c AS a --- b;
    SET d AS a --- b;
  $

  SET i AS c -.- d;
  SET j AS c -.- d;
  SET k AS c -.- d;

  DECLARE ..--. result;
  SET result AS c --- e;

$

/*
MAIN
$
  DECLARE real  a, b, c, d;
  SET a AS 10;
  SET b AS 5;
```

```
  SET c AS 0;
  SET d AS 0;

  DECLARE real i, j, k;
  SET i AS 0;

  while(i < 5)
  $
    SET c AS a + b;
    SET d AS a + b;
  $

  SET i AS c -.- d;
  SET j AS c -.- d;
  SET k AS c -.- d;

$
*/
```

```
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ lex program.l
program.l:47: warning, rule cannot be matched
program.l:83: warning, rule cannot be matched

yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ yacc -d program.y
program.y: warning: 39 shift/reduce conflicts [-Wconflicts-sr]
program.y: warning: 127 reduce/reduce conflicts [-Wconflicts-rr]
program.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples

yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ gcc y.tab.c -ll -lm -w
yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ python3 commonSubElimination.py
Optimized with Common Subexpression Elimination...
Check the result in result_of_cse.txt file :)

yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$ python3 constantFold.py
Optimized with Constant Folding and Propagation...
Check the result in result_of_folding.txt file :)
```

yuvaraj@yuvaraj-Inspiron-3501:~/Desktop/Compiler_design_lab$

Output of Program.y (from asgn3):

3-Address code:

```
        t0=10+10
        a=t0
        t0=a+10
        b=t0
        t1=20*3
        c=t1
        t2=25-10
        d=t2
        t3=c+d
        e=t3
        i=0
L1:
        t5=i<5
        if( not t5)         goto L2
        t1=a+b
        c=t1
        t1=a+b
        d=t1
        goto L1
L2:
        t3=c*d
        i=t3
        t3=c*d
        j=t3
        t3=c*d
        k=t3
        t4=c+e
        result=t4
```

Output of Common subexpression (commonSubElimination.py) :

```
t0=10+10
a=t0
t0=a+10
b=t0
t1=20*3
c=t1
t2=25-10
d=t2
t3=c+d
e=t3
i=0
L1:
t5=i<5
if( not t5)          goto L2
t1=a+b
c=t1
t1=t1
d=t1
goto L1
L2:
t3=c*d
i=t3
t3=t3
j=t3
t3=t3
k=t3
t4=c+e
result=t4
```

Output of Constant Folding (constantFolding.py) :

```
t0=20
a=20
```

```
t0=a+10
b=20
t1=60
c=60
t2=15
d=15
t3=c+d
e=t3
i=0
L1:
t5=i<5
if( not t5)          goto L2
t1=a+b
c=t1
t1=t1
d=t1
goto L1
L2:
t3=c*d
i=t3
t3=t3
j=t3
t3=t3
k=t3
t4=c+e
result=t4
```