Project Report

on

Compiler for <<String Operations Using THT Language>>

Developed by

Triparna Baxi – IT002 – 20ITUON012

Tejendra Dhanani – IT003 -20ITUOF013

Harsh Gondaliya – IT004-20ECUBG049

Guided By:

Prof. Nikita P. Desai

Dept. of Information Technology

Department of Information Technology
Faculty of Technology, Dharmsinh Desai University
College Road, Nadiad-387001
2022-2023

DHARMSINH DESAI UNIVERSITY

NADIAD-387001, GUJARAT

CERTIFICATE

This is to certify that the project entitled "**Compiler for String Operations using THT Language** " is a bonafied report of the work carried out by

1) Triparna Baxi, Student ID No: 20ITUON012
2) Tejendra Dhanani , Student ID No: 20ITUOF013
3) Harsh Gondaliya , Student ID No: 20ECUBG049

of Department of Information Technology, semester VI, under the guidance and supervision for the award of the degree of Bachelor of Technology at Dharmsinh Desai University, Nadiad (Gujarat). They were involved in Project in subject of "**Language Translator**" during academic year 2020-2021.

Prof. N.P. Desai
(Lab Incharge)
Department of Information Technology,
Faculty of Technology,
Dharmsinh Desai University, Nadiad
Date:

Prof. (Dr.)V K Dabhi,
Head , Department of Information Technology,
Faculty of Technology,
Dharmsinh Desai University, Nadiad
Date:

Index

1.0 INTRODUCTION

1.0.1 Project Details

Language Name: String Operations using THT language

Language description:

Write an appropriate language description for a layman language which can do string operations using THT sentences ,written in roman script .

Example of valid program in this language is –

```
i := 11;
print "Value of i: ";
print i;
print "\n";
```

1.0.2 Project Planning

List of Students with their Roles/Responsibilities:

IT002 TRIPARNA BAXI : DFA Design,Algorithm Design.

IT003 TEJENDRA DHANANI : implementation, Scanner phase Implementation ,YACC implementation.

IT004 HARSH GONDALIYA : Regular Expression, Grammar rules, Final Report.

2.0 LEXICAL PHASE DESIGN

2.0.1 Regular Expression:

Keywords :

RE	**Token**
print	print
not	not
and	and
or	or
exit	exit

Operations :

RE	**Token**	**Attribute**
+	op	pop
*	op	mop
-	op	sop
/	op	Dop
<:	op	Gop
>:	op	Lop
>=	op	Geop
=<	op	Leop
!=	op	Nop
^	op	Pwop

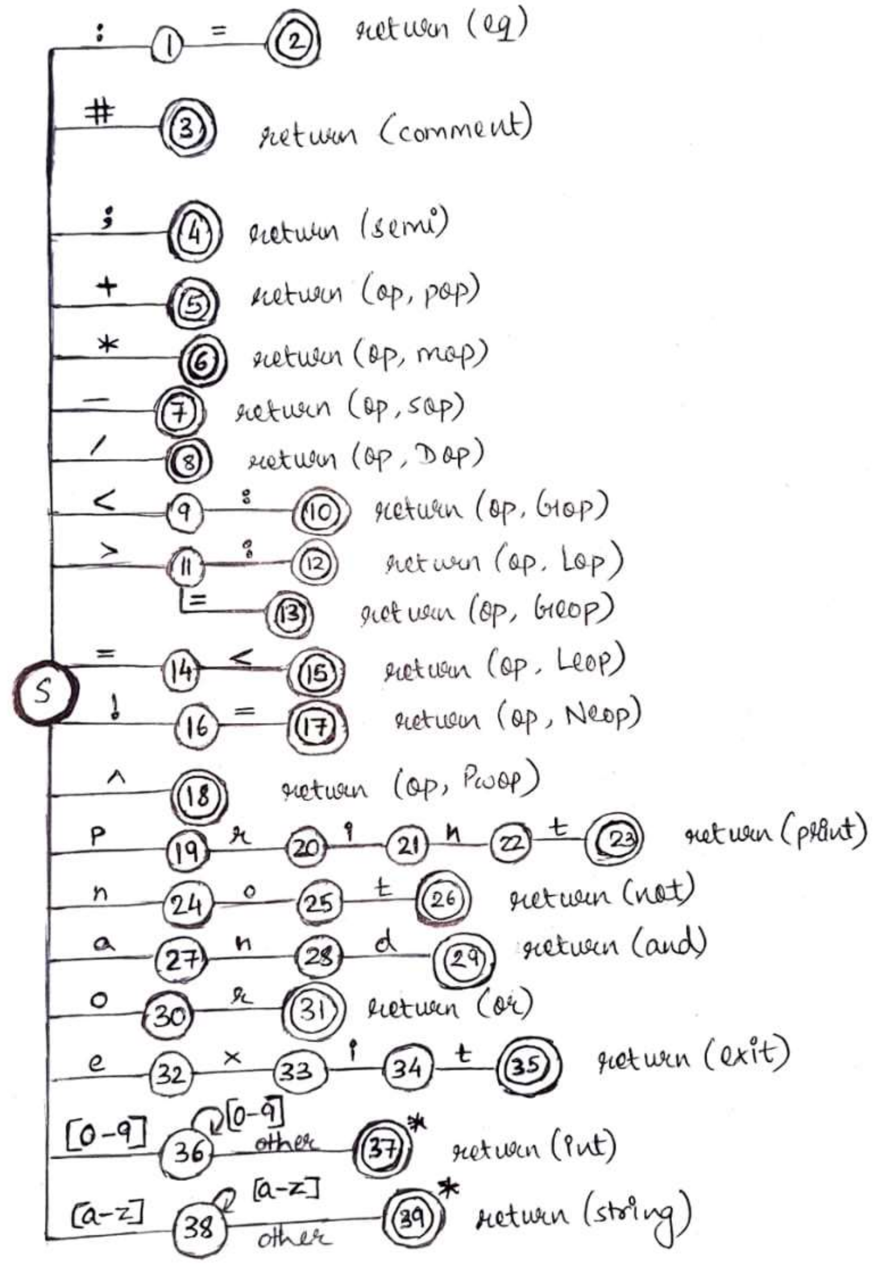
Values type : int and float

RE	**Token**
[0-9]+	int
[(a-z)+(A-Z)]+	string

Delimiters : {. , ? \t}

RE	**Token**
:=	eq
#	comment
;	semi

2.0.2 Deterministic Finite Automata design for lexer



S
: 1 = 2 return (eq)
# 3 return (comment)
; 4 return (semi)
+ 5 return (op, pop)
* 6 return (op, mop)
— 7 return (op, sop)
/ 8 return (op, Dop)
< 9 : 10 return (op, Gop)
> 11 : 12 return (op, Lop)
= 13 return (op, Geop)
= 14 < 15 return (op, Leop)
! 16 = 17 return (op, Neop)
^ 18 return (op, Pwop)
p 19 r 20 i 21 n 22 t 23 return (print)
n 24 o 25 t 26 return (not)
a 27 n 28 d 29 return (and)
o 30 r 31 return (or)
e 32 x 33 i 34 t 35 return (exit)
[0-9] 36 [0-9] other 37* return (int)
[a-z] 38 [a-z] other 39* return (string)

2.0.3 Algorithm of lexer

```
lexer{
   int c=0;
   bool f=flase;
   int len=string.length();
   while not eof do
   {
      state="S";
      while not eof do(c<len)
      {
         if(f)
         {
            f=false;
         }
         char ch=nextchar();
         switch(state){
            case state of "S":
               case state of
               ':':
                  state="1";
                  ch=nextchar();
                  f=true;
                  break;

               '#':
                  state="3";
                  ch=nextchar();
                  f=true;
                  break;

               ';':
                  state="4";
                  ch=nextchar();
                  f=true;
                  break;

               '+':
```

```
        state="5";
        ch=nextchar();
        break;

    '*':
        state="6";
        ch=nextchar();
        break;

    '-':
        state="7";
        ch=nextchar();
        break;

    '/':
        state="8";
        ch=nextchar();
        break;

    '<':
        state="9";
        ch=nextchar();
        break;

    '>':
        state="11";
        ch=nextchar();
        break;

    '=':
        state="14";
        ch=nextchar();
        break;

    '!':
        state="16";
        ch=nextchar();
        break;
```

```
'^':
   state="18";
   ch=nextchar();
   break;

'p':
   state="19";
   ch=nextchar();
   break;

'n':
   state="24";
   ch=nextchar();
   break;

'a':
   state="27";
   ch=nextchar();
   break;

'o':
   state="30";
   ch=nextchar();
   break;

'e':
   state="32";
   ch=nextchar();
   break;

[0-9]:
   state="36";
   ch=nextchar();
   break;

[a-z]:
   state="38";
   ch=nextchar();
   break;
```

```
    }
    default:
        f=true;
    end case

case state of "1":
    case state of '=':
        state="2";
        ch=nextchar();
        break;

    default:
        f=true;

case state of "9":
    case state of ':':
        state="10";
        ch=nextchar();
        break;

    default:
        f=true;

case state of "11":
    case state of ':':
        state="12";
        ch=nextchar();
        break;
    case state of '=':
        state="13";
        ch=nextchar();
        break;
    default:
        f=true;

case state of "14":
    case state of '<':
        state="15";
        ch=nextchar();
        break;
    default:
        f=true;

case state of "16":
    case state of '=':
        state="17";
        ch=nextchar();
        break;

    default:
        f=true;

case state of "19":
    case state of 'r':
        state="20";
        ch=nextchar();
case state of "20":
    case state of 'i':
        state="21";
        ch=nextchar();
case state of "21":
    case state of 'n':
        state="22";
        ch=nextchar();
case state of "22":
    case state of 't':
        state="23";
        ch=nextchar();
        f=true;

case state of "24":
    case state of 'o':
        state=25;
        ch=nextchar();
case state of "25":
    case state of 't':
        state=26;
        ch=nextchar();
        f=true;

case state of "27":
    case state of 'n':
        state=28;
        ch=nextchar();
case state of "28":
    case state of 'd':
        state=29;
        ch=nextchar();
        f=true;

case state of "30":
    case state of 'r':
        state=31;
        ch=nextchar();
        f=true;

case state of "32":
    case state of 'x':
        state="33";
        ch=nextchar();
case state of "33":
    case state of 'i':
        state="34";
        ch=nextchar();
case state of "34":
    case state of 't':
        state="35":
        ch=nextchar();
        f=true;

case state of "36":
    case state of [0-9]:
        ch=nextchar();
    default:
        state="37";
        f=true;
```

```
            case state of "38":
                case state of [a-z]:
                    ch=nextchar();
                default:
                    satte="39";
                    f=true;
        }


        case state of
            "2":
            print("eq");

            "3":
            print("comment");

            "4":
            print("semi");

            "5" | "6" | "7" | "8" | "10" | "12" | "13" | "15" | "17" | "18":
            print("operator");

            "23" | "26" | "29" | "31" | "35" | "37" | "39":
            print("keyword");

            "37":
            print("int");

            "39":
            print("string");

            default:
                print("invalid input");
                ch := nextchar();
            end case;
    }
}
```

2.0.4 Implementation of lexer

Flex Program:

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

Keyword  "print"|"not"|"and"|"or"|"exit”
Op         "+"|"-"|"/"|"*"|
           “<:”|”>:”|”>=”|”=<”|”!=”|”^”
Digit      [0-9]
String     [(A-Z)+(a-z)]
Int         {Digit}+
St          {String}+
eq              ":="
comment    “#”
semi      ":"

%%

{Keyword}  {printf("Keyword - %s\n",yytext);}
{Op}         {printf("Operator  - %s\n",yytext);}
{Int}        {printf("Integer - %s\n",yytext);}
{String}     {printf("String Number - %s\n",yytext);}
{eq}         {printf("que tag - %s\n",yytext);}
{comment}  {printf("eos - %s\n",yytext);}
{semi}       {printf("sep - %s\n",yytext);}

%%
int yywrap(){return 1;}
int main()
{
   yylex();
   return 0;
}
```

2.0.5 Execution environment setup

Step by Step Guide to Install FLEX and Run FLEX Program using Command Prompt(cmd)

Step 1

/*For downloading CODEBLOCKS */

- Open your Browser and type in "codeblocks"
- Goto to Code Blocks and go to downloads section
- Click on "Download the binary release"
- Download codeblocks-20.03mingw-setup.exe
- Install the software keep clicking on next

/*For downloading FLEX GnuWin32 */

- Open your Browser and type in "download flex gnuwin32"
- Goto to "Download GnuWin from SourceForge.net"
- Downloading will start automatically
- Install the software keep clicking on next

/*SAVE IT INSIDE C FOLDER*/

Step 2 /*PATH SETUP FOR CODEBLOCKS*/

- After successful installation

Goto program files->CodeBlocks-->MinGW-->Bin

- Copy the address of bin :-

it should somewhat look like this

C:\Program Files (x86)\CodeBlocks\MinGW\bin

- Open Control Panel-->Goto System-->Advance System Settings-->Environment Variables
- Environment Variables--> Click on Path which is inside System variables - Click on edit
- Click on New and paste the copied path to it:-
- C:\Program Files (x86)\CodeBlocks\MinGW\bin

- Press Ok!

Step 3 /*PATH SETUP FOR GnuWin32*/

- After successful installation Goto C folder

- Goto GnuWin32-->Bin

- Copy the address of bin it should somewhat look like this

C:\GnuWin32\bin

- Open Control Panel-->Goto System-->Advance System Settings-->Environment Variables

- Environment Variables--> Click on Path which is inside System variables - Click on edit
- Click on New and paste the copied path to it:-

- C:\GnuWin32\bin

- Press Ok!

/*WARNING!!! PLEASE MAKE SURE THAT PATH OF CODEBLOCKS IS BEFORE GNUWIN32---THE ORDER MATTERS*/

Step 4

- Create a folder on Desktop flex_programs or whichever name you like - Open notepad type in a flex program
- Save it inside the folder like filename.l

-Note :- also include “”” void yywrap(){} “””””” in the .l file

/*Make sure while saving save it as all files rather than as a text document*/
Step 5 /*To RUN FLEX PROGRAM*/

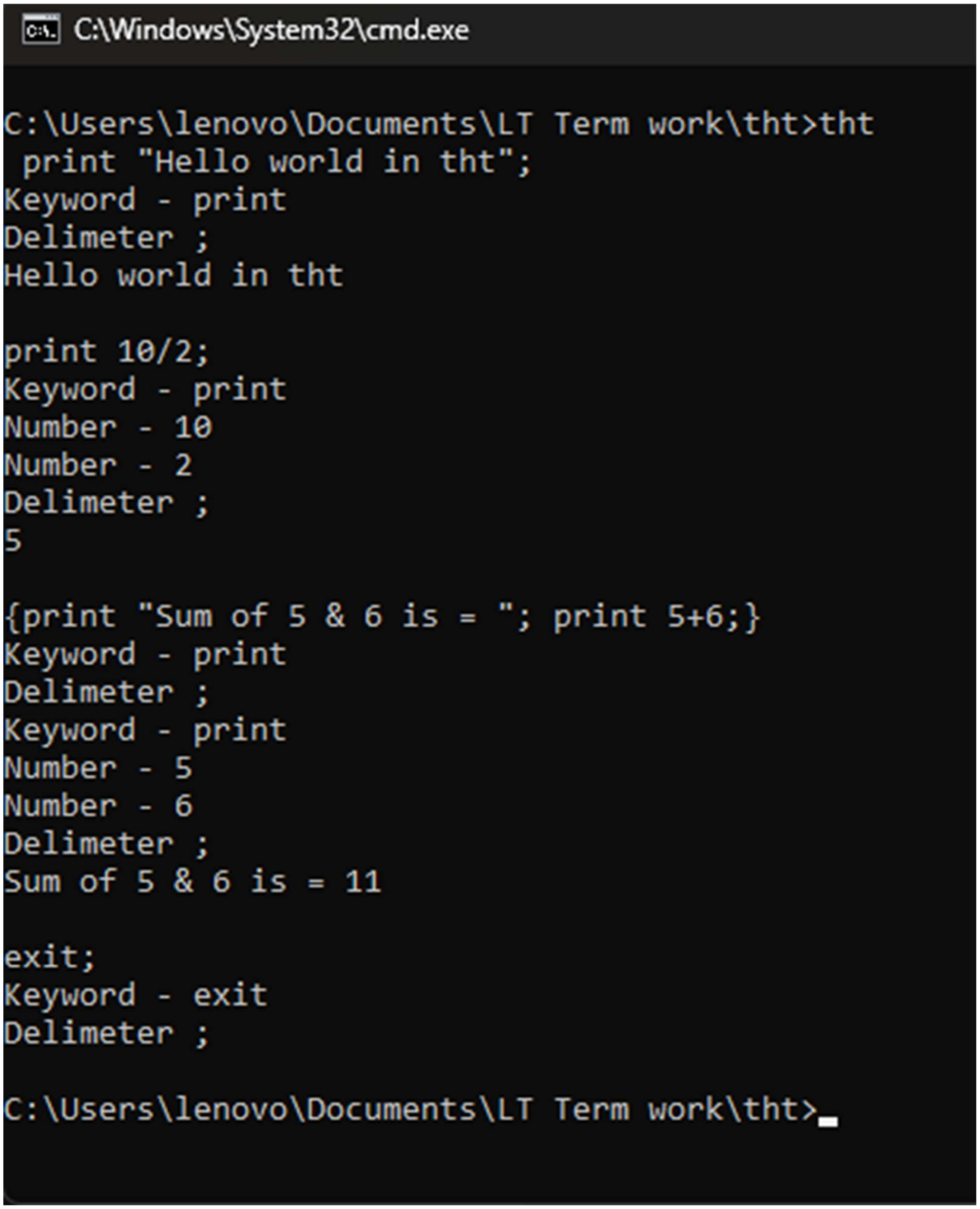
- Goto to Command Prompt(cmd)

- Goto the directory where you have saved the program - Type in command :- **flex filename.l**
- Type in command :- **gcc lex.yy.c**

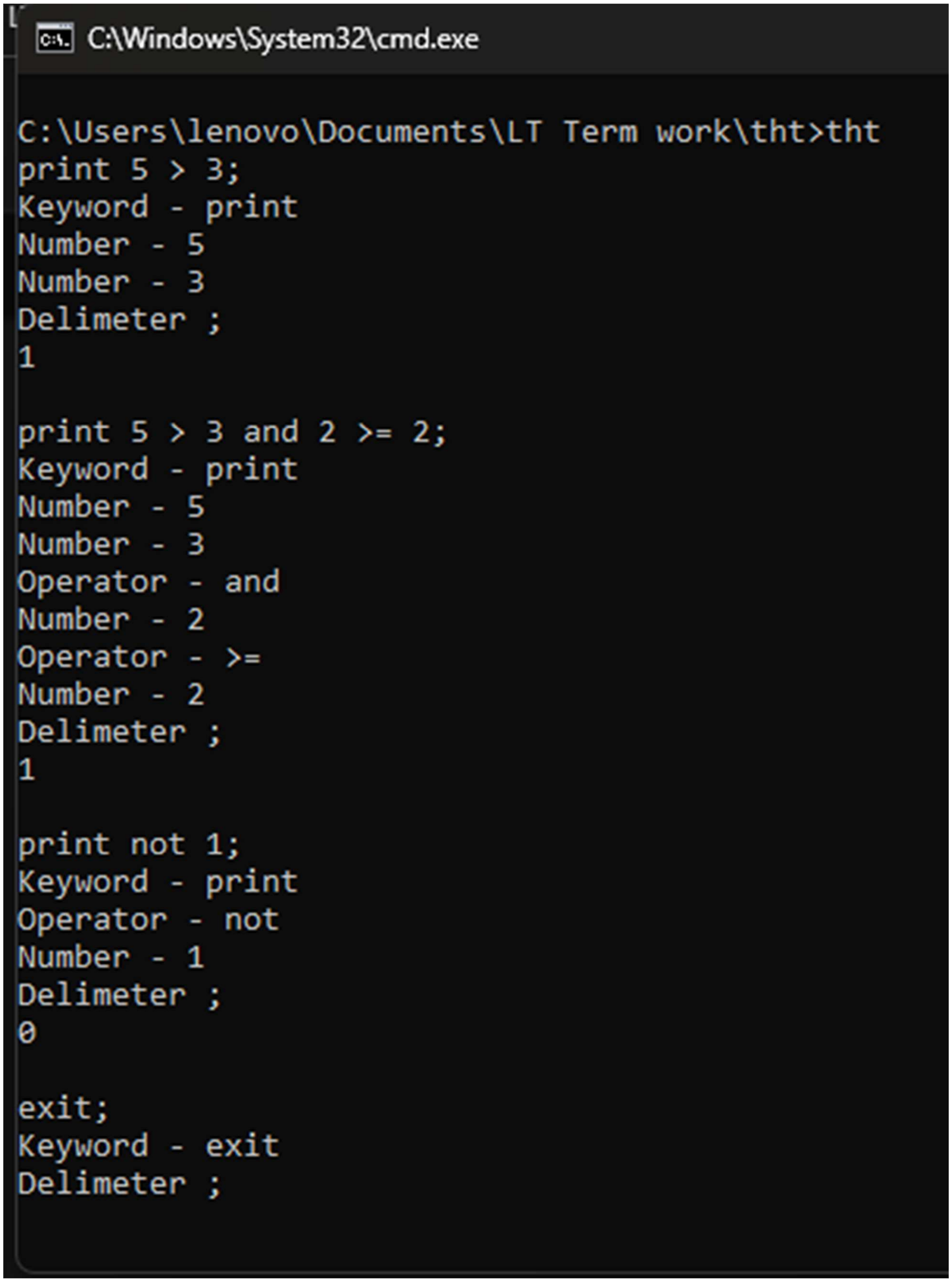
- Execute/Run for windows command promt :- **a.exe**

Step 6
- Finished

2.0.6 Output screenshots of lexer.


C:\Windows\System32\cmd.exe

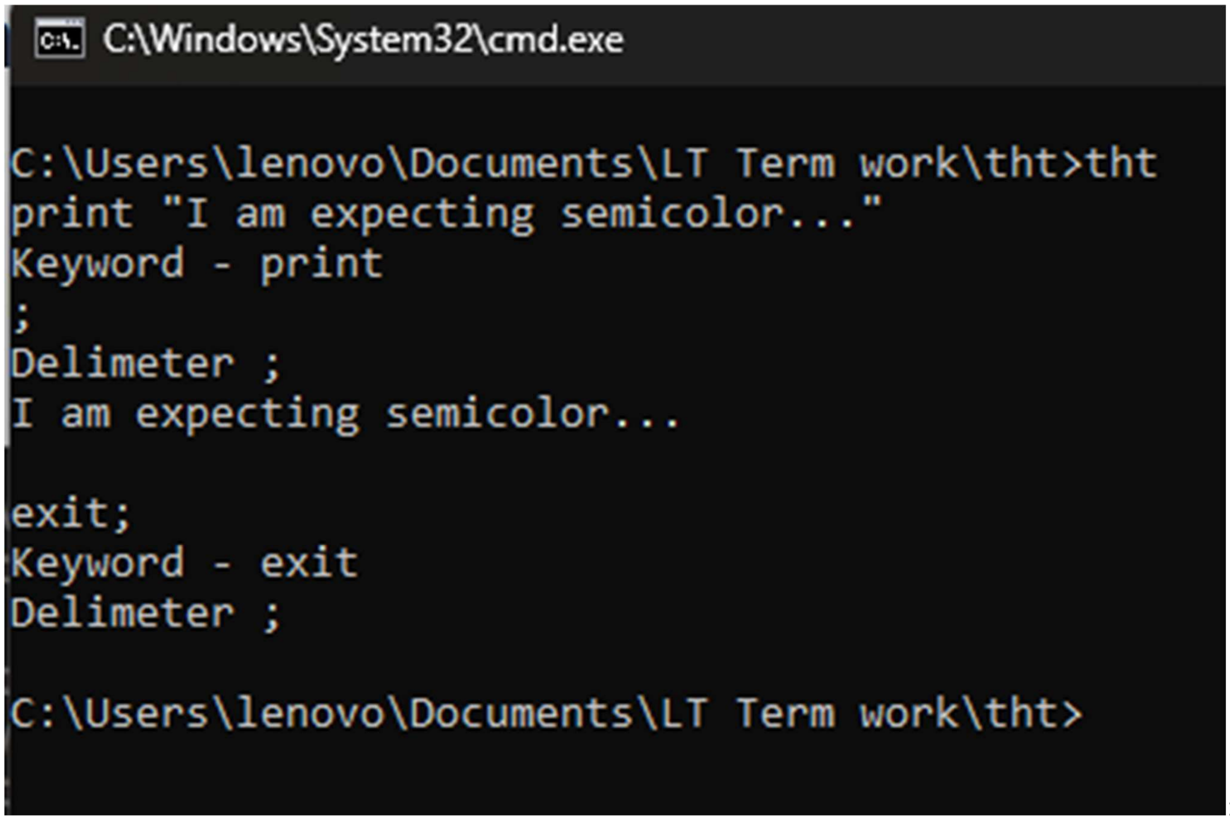
```
C:\Users\lenovo\Documents\LT Term work\tht>tht
 print "Hello world in tht";
Keyword - print
Delimeter ;
Hello world in tht

print 10/2;
Keyword - print
Number - 10
Number - 2
Delimeter ;
5

{print "Sum of 5 & 6 is = "; print 5+6;}
Keyword - print
Delimeter ;
Keyword - print
Number - 5
Number - 6
Delimeter ;
Sum of 5 & 6 is = 11

exit;
Keyword - exit
Delimeter ;

C:\Users\lenovo\Documents\LT Term work\tht>
```



C:\Windows\System32\cmd.exe

```
C:\Users\lenovo\Documents\LT Term work\tht>tht
print 5 > 3;
Keyword - print
Number - 5
Number - 3
Delimeter ;
1

print 5 > 3 and 2 >= 2;
Keyword - print
Number - 5
Number - 3
Operator - and
Number - 2
Operator - >=
Number - 2
Delimeter ;
1

print not 1;
Keyword - print
Operator - not
Number - 1
Delimeter ;
0

exit;
Keyword - exit
Delimeter ;
```

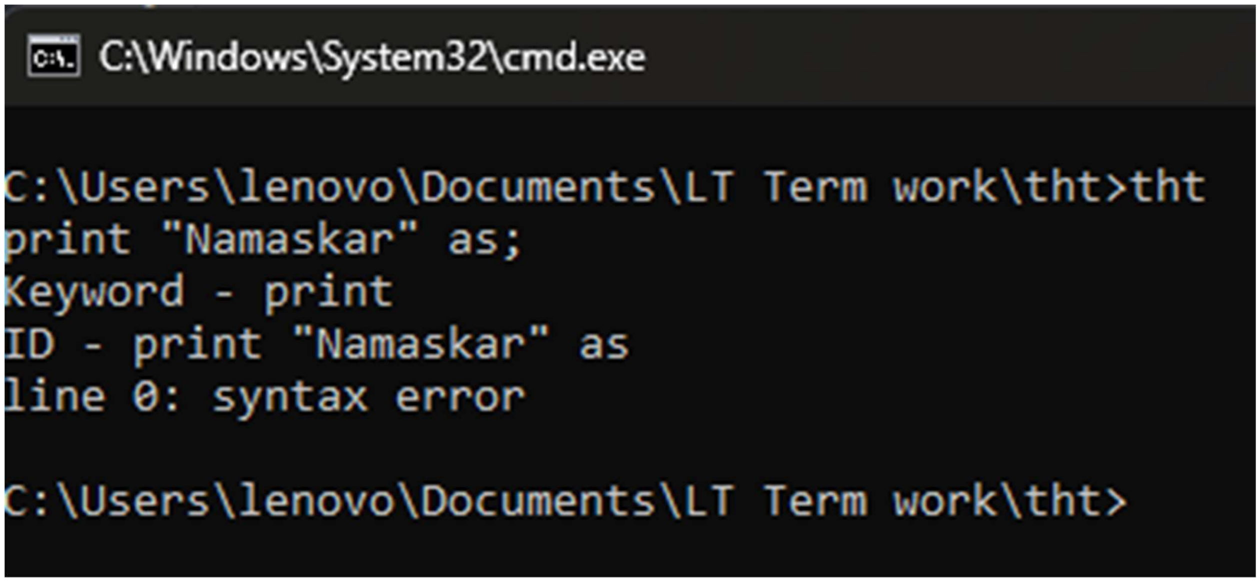
Invalid :



C:\Windows\System32\cmd.exe

```
C:\Users\lenovo\Documents\LT Term work\tht>tht
print "I am expecting semicolor..."
Keyword - print
;
Delimeter ;
I am expecting semicolor...

exit;
Keyword - exit
Delimeter ;

C:\Users\lenovo\Documents\LT Term work\tht>
```

:



C:\Windows\System32\cmd.exe

```
C:\Users\lenovo\Documents\LT Term work\tht>tht
print "Namaskar" as;
Keyword - print
ID - print "Namaskar" as
line 0: syntax error

C:\Users\lenovo\Documents\LT Term work\tht>
```

3.0 SYNTAX ANALYZER DESIGN

3.0.1 Grammar rules

S -> A | B | E

A -> KEYW C SEMI | KEYW SEMI

C -> NUM | NUM OP C | ST

B -> ST EQ SEMI

E -> COMM ST

OP-> + | - | / | * | <: | >: | >= | =< | != | ^

KEYW-> print | not | and | or | exit

NUM -> int

ST -> String

EQ -> **:=**

COMMENT -> #

SEMI -> ;

3.0.2 Yacc based imlementation of syntax analyzer

- **tht.l (Lex file)**

```
%{
   #include <stdlib.h>
   #include <string.h>
   #include "tht.h"
   #include "y.tab.h"

   void yyerror(char *s);
   int lineno = 0;
%}

ID      [a-zA-Z][a-z0-9A-Z_]*
EXPO    [Ee][-+]?[0-9]+

%%
"#".*   /* Single-line comment */

[0-9]+|[0-9]+"."[0-9]*{EXPO}?|"."?[0-9]+{EXPO}? {
   printf("Number - %s\n",yytext);
   yylval.dValue = atof(yytext);
   return NUMBER;
}

\"[^"\n]*["\n] {
   yylval.sValue = strdup(yytext+1);
   if (yylval.sValue[yyleng-2] != '"')
      yyerror("Improperly Terminated String");
   else {
      yylval.sValue[yyleng-2] = 0;
      return STRING;
   }
}

";" {

   printf("Delimeter ;\n");
   return *yytext;
}

[-()<>=+*/%^,;:{}] {
   return *yytext;
}

">="    {printf("Operator - %s\n",yytext); return GE; }
"<="    {printf("Operator - %s\n",yytext); return LE; }
"!="    {printf("Operator - %s\n",yytext); return NE; }
```

```
"and"   {printf("Operator - %s\n",yytext); return AND; }
"or"    {printf("Operator - %s\n",yytext); return OR; }
"not"   {printf("Operator - %s\n",yytext); return NOT; }

"print" {printf("Keyword - %s\n",yytext); return PRINT;}
"exit"  {printf("Keyword - %s\n",yytext); return EXIT;}

{ID} {
   printf("ID - %s\n");
   yylval.vName = strdup(yytext);
   return EXIT;
}

[ \t]

\n { ++lineno; }

. {
   printf("Unknown character!\n");
   // yyerror("Unknown character\n");
   // exit(1);
}
%%

int yywrap(void) {
   return 1;
}

void yyerror(char *s) {
   fprintf(stdout, "line %d: %s\n", lineno, s);
}
```

- **tht.y (yacc code)**

```
%{
   #include <stdio.h>
   #include <stdlib.h>
   #include <stdarg.h>
   #include <string.h>
   #include <time.h>
   #include <math.h>
   #include "tht.h"

   #define YYDEBUG 0

   nodeType cond(double dValue);          / Constant double type node */
   nodeType cons(char *sValue);           / Constant string type node */
   nodeType opr(int oper, int nops, ...); / Operator type node */
   void freeNode(nodeType p);             / Free the node */
   double ex(nodeType p);                 / Execute graph */
   int yylex(void);

   void yyerror(char *);
   double sym[SYMSIZE];        /* Symbol table */
   char vars[SYMSIZE][IDLEN];  /* Variable table: for mapping
variables to symbol table */
   unsigned int seed;
%}

%union {
   double dValue;
   char *sValue;
   char *vName;
   nodeType *nPtr;
}

// double value token for number
%token <dValue> NUMBER
// string value token for strings
%token <sValue> STRING
```

```
// rest tokens
%token PRINT EXIT


// left associates with other tokens
%left AND OR
%left GE LE '=' NE '>' '<'
%left '+' '-'
%left '*' '/' '%'
%left NOT
%left '^'

// no associations
%nonassoc UMINUS

// non terminal type. nPtr is custome type to store both doubles and
string
%type <nPtr> statement expression statement_list

%%
program : function { exit(0); }
        ;

function :
        | function statement { ex($2); }
        ;

statement : ';' { $$ = opr(';', 2, NULL, NULL); }
         | expression ';' { $$ = $1; }
         | EXIT ';' { exit(0); }
         | PRINT expression ';' { $$ = opr(PRINT, 1, $2); }
         | PRINT STRING ';' { $$ = opr(PRINT, 1, cons($2)); }
         | '{' statement_list '}' { $$ = $2; }
         ;

statement_list : statement { $$ = $1; }
             | statement_list statement { $$ = opr(';', 2, $1, $2); }
             ;
```

```
expression : NUMBER { $$ = cond($1); }
        | '-' expression %prec UMINUS { $$ = opr(UMINUS, 1, $2); }
        | expression '^' expression { $$ = opr('^', 2, $1, $3); }
        | expression '+' expression { $$ = opr('+', 2, $1, $3); }
        | expression '-' expression { $$ = opr('-', 2, $1, $3); }
        | expression '*' expression { $$ = opr('*', 2, $1, $3); }
        | expression '/' expression { $$ = opr('/', 2, $1, $3); }
        | expression '%' expression { $$ = opr('%', 2, $1, $3); }
        | expression '<' expression { $$ = opr('<', 2, $1, $3); }
        | expression '>' expression { $$ = opr('>', 2, $1, $3); }
        | expression GE expression { $$ = opr(GE, 2, $1, $3); }
        | expression LE expression { $$ = opr(LE, 2, $1, $3); }
        | expression '=' expression { $$ = opr('=', 2, $1, $3); }
        | expression NE expression { $$ = opr(NE, 2, $1, $3); }
        | expression AND expression { $$ = opr(AND, 2, $1, $3); }
        | expression OR expression { $$ = opr(OR, 2, $1, $3); }
        | NOT expression { $$ = opr(NOT, 1, $2); }
        | '(' expression ')' { $$ = $2; }
        ;
%%



nodeType *cond(double dValue) {
   nodeType *p;

   /* allocate node */
   if ((p = malloc(sizeof(nodeType))) == NULL)
      yyerror("out of memory");

   /* copy information */
   p->type = typeCon;
   p->con.type = typeNum;
   p->con.dValue = dValue;

   return p;
}

nodeType *cons(char *sValue) {
   nodeType *p;

   /* allocate node */
   if ((p = malloc(sizeof(nodeType))) == NULL)
      yyerror("out of memory");

   /* copy information */
   p->type = typeCon;
   p->con.type = typeStr;
   p->con.sValue = strdup(sValue);

   return p;
}

nodeType *opr(int oper, int nops, ...) {
   va_list ap;
   nodeType *p;

   /* allocate node */
   if ((p = malloc(sizeof(nodeType))) == NULL)
      yyerror("out of memory");
   if ((p->opr.op = malloc(nops * sizeof(nodeType *))) == NULL)
      yyerror("out of memory");

   /* copy information */
   p->type = typeOpr;
   p->opr.oper = oper;
   p->opr.nops = nops;

   va_start(ap, nops);
   for (int i = 0; i < nops; i++)
      p->opr.op[i] = va_arg(ap, nodeType *);
   va_end(ap);

   return p;
}

double ex(nodeType *p) {
   if (!p) return 0;
   switch (p->type) {
      case typeCon: return p->con.dValue;
      case typeId: return sym[p->id.i];
      case typeOpr:
          switch (p->opr.oper) {
              case PRINT:
                  if (p->opr.op[0]->type == typeCon && p->opr.op[0]-
>con.type == typeStr) {
                        char *sValue = p->opr.op[0]->con.sValue;
                        int i, slen = strlen(sValue);
                        for (i = 0; i < slen-1; i++) {
                            if (sValue[i] == '\\' && sValue[i+1] == 'n') {
                                printf("\n");
                                i++;
                            }
                            else if (sValue[i] == '\\' && sValue[i+1] == 't') {
                                printf("\t");
                                i++;
                            }
                            else printf("%c", sValue[i]);
                        }
                        if (i == slen-1) printf("%c", sValue[i]);
                        return 0;
                  }
                  else {
                        double dValue = ex(p->opr.op[0]);
                        if (dValue == floor(dValue)) printf("%d", (int)dValue);
                        else if (dValue - floor(dValue) < 1e-6) printf("%e",
dValue);
                        else printf("%lf", dValue);
                        return 0;
                  }
              case ';':
                  ex(p->opr.op[0]);
                  return ex(p->opr.op[1]);
              case UMINUS: return -ex(p->opr.op[0]);
              case '^': return pow(ex(p->opr.op[0]), ex(p->opr.op[1]));
              case '+': return ex(p->opr.op[0]) + ex(p->opr.op[1]);
              case '-': return ex(p->opr.op[0]) - ex(p->opr.op[1]);
              case '*': return ex(p->opr.op[0]) * ex(p->opr.op[1]);
              case '/': return ex(p->opr.op[0]) / ex(p->opr.op[1]);
              case '%': return (int)ex(p->opr.op[0]) % (int)ex(p->opr.op[1]);
              case '>': return ex(p->opr.op[0]) > ex(p->opr.op[1]);
              case '<': return ex(p->opr.op[0]) < ex(p->opr.op[1]);
              case GE: return ex(p->opr.op[0]) >= ex(p->opr.op[1]);
              case LE: return ex(p->opr.op[0]) <= ex(p->opr.op[1]);
              case '=': return ex(p->opr.op[0]) == ex(p->opr.op[1]);
              case NE: return ex(p->opr.op[0]) != ex(p->opr.op[1]);
              case AND: return (int)ex(p->opr.op[0]) && (int)ex(p-
>opr.op[1]);
              case OR: return (int)ex(p->opr.op[0]) || (int)ex(p->opr.op[1]);
              case NOT: return !(int)ex(p->opr.op[0]);
          }
   }
   return 0;
}

int main(int argc, char **argv) {
   #if YYDEBUG
      yydebug = 1;
   #endif

   seed = time(NULL);

   /* Initialize variable table */
   for (int i = 0; i < SYMSIZE; i++) strcpy(vars[i], "-1");

   if (argc < 2)
      yyparse();
   else {
      freopen(argv[1], "r", stdin);
      yyparse();
   }

   return 0;
}
```

3.0.3 Execution environment setup

Download flex and bison from the given links.

http://gnuwin32.sourceforge.net/packages/flex.htm
http://gnuwin32.sourceforge.net/packages/bison.htm

when installing on windows you store this in c:/gnuwin32 folder and not in c:/program files(X86)/gnuwin32

Download IDE

https://sourceforge.net/projects/orwelldevcpp/ set environment variable for flex and bison.

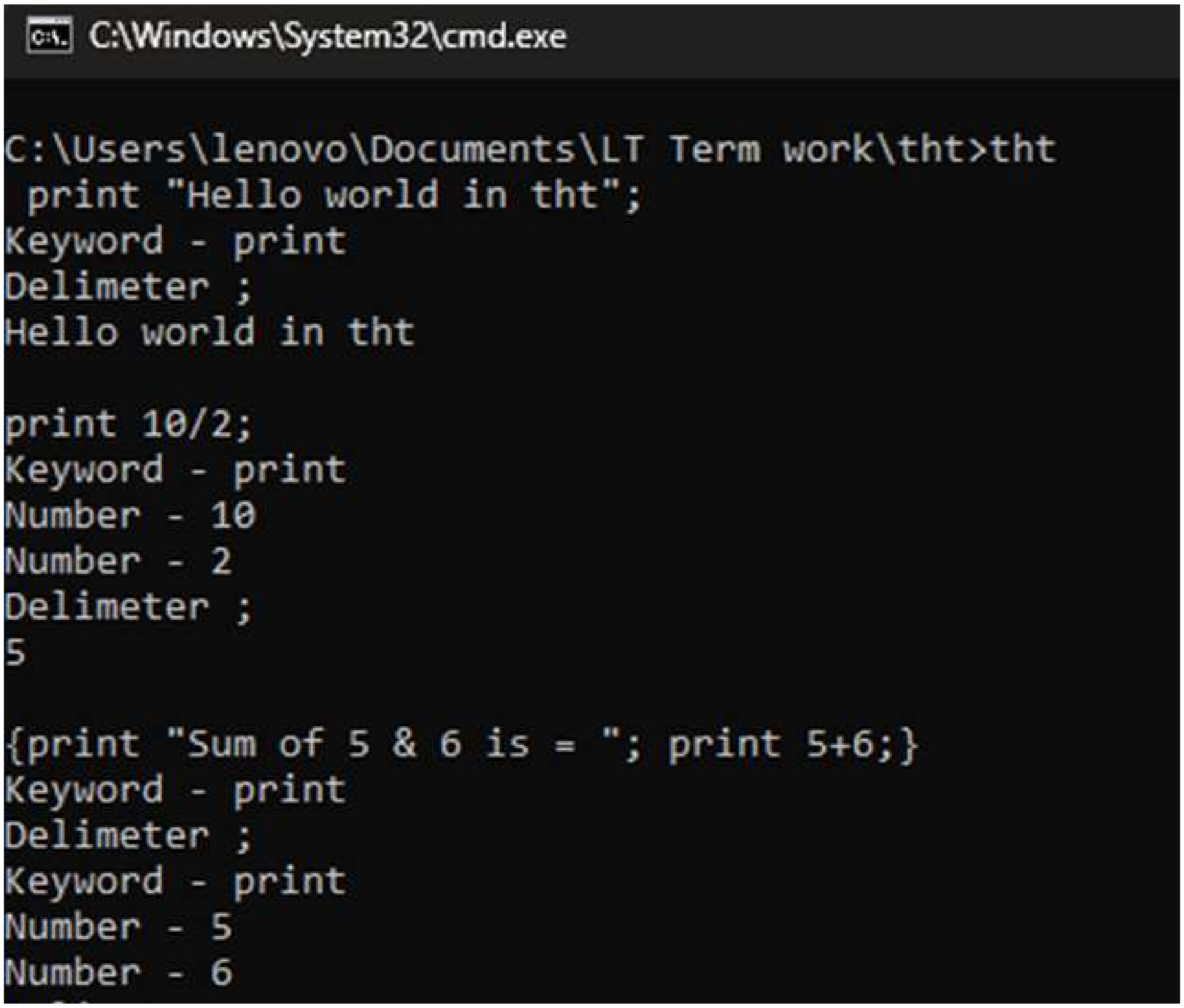
To run the program:

Open a prompt, cd to the directory where your ".l" and ".y" are, and compile them with:

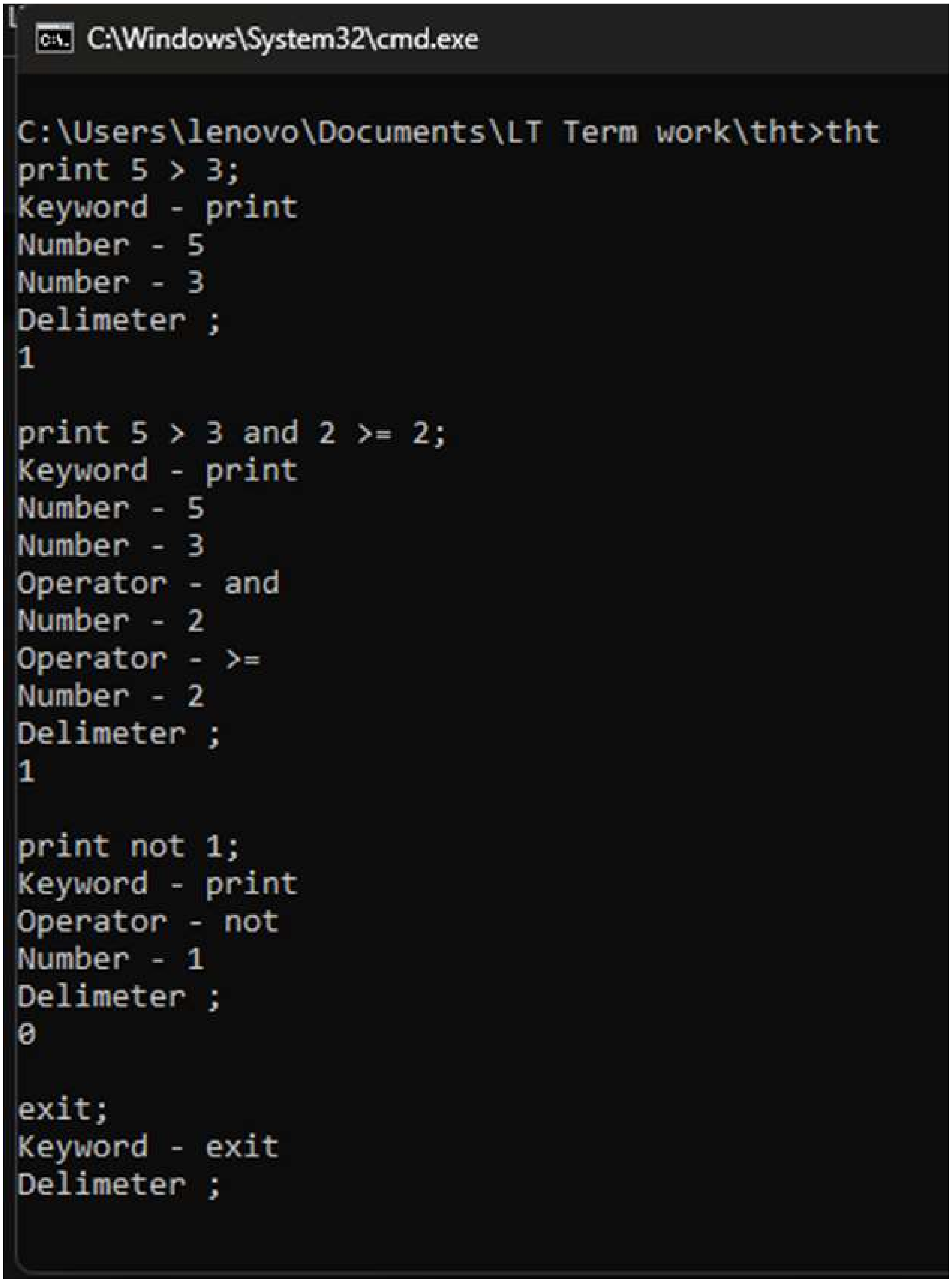
```
flex  yacc.l
bison -dy yacc.y
gcc lex.yy.c y.tab.c -o yacc.exe
```

3.0.4 Output screenshots of yacc based implementation

- **Valid Input with all the possible combinations:**



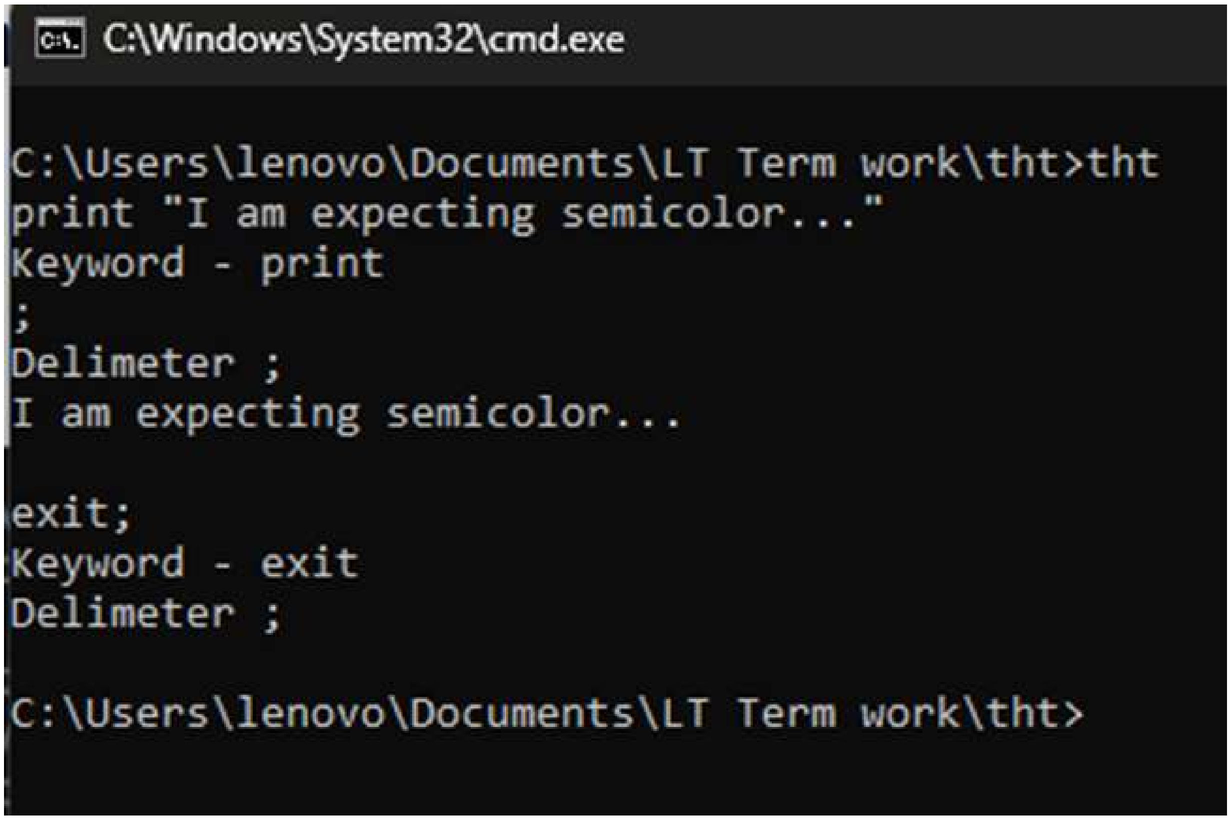
```
C:\Windows\System32\cmd.exe

C:\Users\lenovo\Documents\LT Term work\tht>tht
 print "Hello world in tht";
Keyword - print
Delimeter ;
Hello world in tht

print 10/2;
Keyword - print
Number - 10
Number - 2
Delimeter ;
5

{print "Sum of 5 & 6 is = "; print 5+6;}
Keyword - print
Delimeter ;
Keyword - print
Number - 5
Number - 6
```


C:\Windows\System32\cmd.exe

```
C:\Users\lenovo\Documents\LT Term work\tht>tht
print 5 > 3;
Keyword - print
Number - 5
Number - 3
Delimeter ;
1

print 5 > 3 and 2 >= 2;
Keyword - print
Number - 5
Number - 3
Operator - and
Number - 2
Operator - >=
Number - 2
Delimeter ;
1

print not 1;
Keyword - print
Operator - not
Number - 1
Delimeter ;
0

exit;
Keyword - exit
Delimeter ;
```

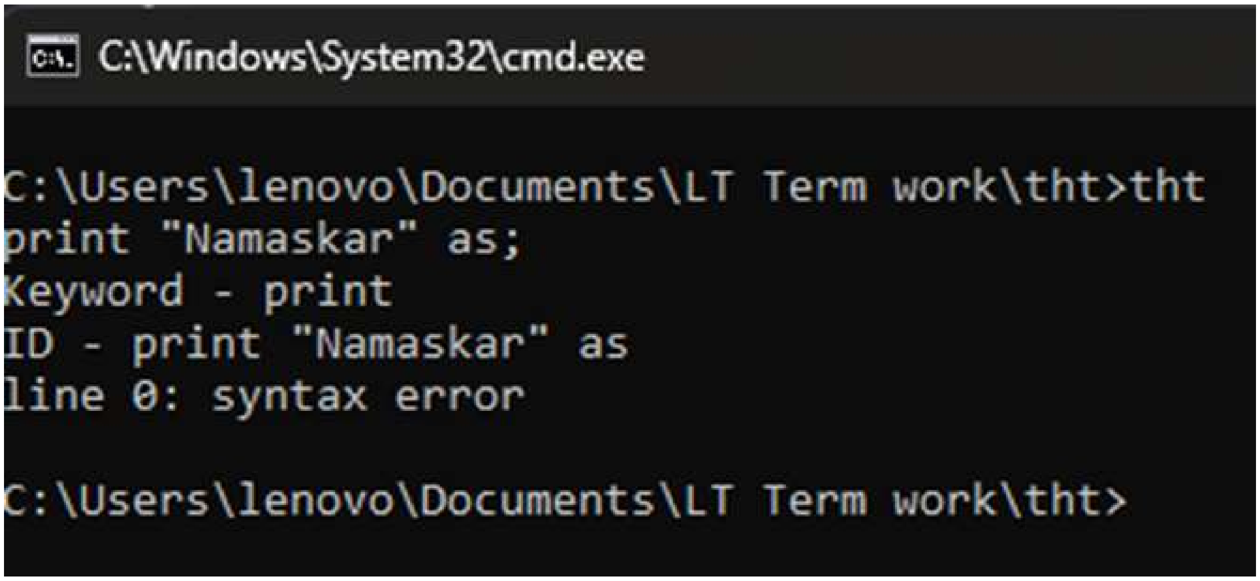
- **Invalid Syntax:**

1. **Program is not complete yet (expecting semicolon at last)**



```
C:\Windows\System32\cmd.exe

C:\Users\lenovo\Documents\LT Term work\tht>tht
print "I am expecting semicolor..."
Keyword - print
;
Delimeter ;
I am expecting semicolor...

exit;
Keyword - exit
Delimeter ;

C:\Users\lenovo\Documents\LT Term work\tht>
```

2. **Enter Wrong Syntax sentence**



```
C:\Windows\System32\cmd.exe

C:\Users\lenovo\Documents\LT Term work\tht>tht
print "Namaskar" as;
Keyword - print
ID - print "Namaskar" as
line 0: syntax error

C:\Users\lenovo\Documents\LT Term work\tht>
```

4.0 CONCLUSION

This project has been implemented from what we have learned in our college curriculum and many rich resources from the web. After doing this project we conclude that we have got more knowledge about how different compilers are working in practical world and also how various types of errors are handled.

Project Link!!