Report on

C++ Mini-Compiler

Submitted in partial fulfillment of the requirements for **Sem VI**

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Kashish Oberoi	**PES1201700113**
Sagar Ratan Garg	**PES1201700913**
Vivek Aditya	**PES1201701125**

Under the guidance of
Preet Kanwal
Assistant Professor
PES University

January – May 2020

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

TABLE OF CONTENTS

INTRODUCTION

This project being a Mini Compiler for the C++ programming language, focuses on generating an intermediate code for the language for specific constructs.
It works for constructs such as conditional statements, loops (for and while).
The main functionality of the project is to generate an optimized intermediate code for the given C++ source code and also assembly code using this optimized intermediate code generated.
This is done using the following steps:

i) Generate symbol table after performing expression evaluation
ii) Generate Abstract Syntax Tree for the code
iii) Generate 3 address code followed by corresponding quadruples
iv) Perform Code Optimization
v) Generate Assembly code

The main tools used in the project include LEX which identifies pre-defined patterns and generates tokens for the patterns matched and YACC which parses the input for semantic meaning and generates an abstract syntax tree and
intermediate code for the source code.
PYTHON is used to optimize the intermediate code generated by the parser and generating Assembly code from intermediate code.

LITERATURE SURVEY AND OTHER REFERENCES

- 2014 IJIRT | Volume 1 Issue 5 | ISSN : 2349-6002 IJIRT 100158 INTERNATONAL JOURNAL OF INNOVATIVE RESEARCH IN TECHNOLOGY 151 “Research paper on Compiler Design” - Jatin Chhabra, Hiteshi Chopra, Abhimanyu Vats

- Paper on Symbol Table Implementation in Compiler Design- Dr.Jad Matta

- IJCSMC, Vol. 2, Issue. 10, October 2013, pg.115 – 125 SURVEY ARTICLE “Analysis of Parsing Techniques & Survey on Compiler Applications” - Ch. Raju , Thirupathi Marupaka, Arvind Tudigani

- “A Study on the Impact of Compiler Optimizations on High-Level Synthesis” - Jason Cong, Bin Liu, Raghu Prabhakar, and Peng Zhang University of California, Los Angeles.

ARCHITECTURE OF LANGUAGE

C++ constructs implemented:

1. Simple If
2. If-else
3. While loop
4. For-loop

- Arithmetic expressions with +, -, *, /, ++, -- are handled
- Boolean expressions with >,<,>=,<=,== are handled
- Error handling reports undeclared variables
- Error handling also reports syntax errors with line numbers
- Error handling also reports if the same variable is declared twice in the same scope.

DESIGN STAGES AND IMPLEMENTATION

Phase 1: (a)Lexical Analysis

- LEX tool was used to create a scanner for C++ language
- The scanner transforms the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler.
- The scanner also scans for the comments (single-line and multi-line comments) and writes the source file without comments onto an output file which is used in the further stages.
- All tokens included are of the form T_<token-name>.Eg: T_pl for ‘+’,T_min for ‘-’, T_lt for ‘<’ etc.
- A global variable ‘yylavl’ is used to record the value of each lexeme scanned. ‘yytext’ is the lex variable that stores the matched string.
- Skipping over white spaces and recognizing all keywords, operators, variables and constants is handled in this phase.
- Scanning error is reported when the input string does not match any rule in the lex file.
- The rules are regular expressions which have corresponding actions that execute on a match with the source input.

The following is the lex file used -

```
%{
    #include<string.h>
    #include<stdio.h>
    int line = 0;
    #define YYSTYPE char *
%}

alpha [A-Za-z_]
digit [0-9]
%option yylineno
%%
[ \t\n] {yylval = strdup(yytext);}
":" {yylval = strdup(yytext);return T_colon;}
"?" {yylval = strdup(yytext);return T_ques;}
"while" {yylval = strdup(yytext);return WHILE;}
"for" {yylval = strdup(yytext);return FOR;}
"if" {yylval = strdup(yytext);return IF;}
"else" {yylval = strdup(yytext);return ELSE;}
"cout" {yylval = strdup(yytext);return COUT;}
"endl" {yylval = strdup(yytext);return ENDL;}
"break" {yylval = strdup(yytext);return BREAK;}
"continue" {yylval = strdup(yytext);return CONTINUE;}
"int" {yylval = strdup(yytext);return INT;}
"float" {yylval = strdup(yytext);return FLOAT;}
"char" {yylval = strdup(yytext);return CHAR;}
"void" {yylval = strdup(yytext);return VOID;}
"#include" {yylval = strdup(yytext);return INCLUDE;}
"main()" {yylval = strdup(yytext);return MAINTOK;}
{digit}+    {yylval = strdup(yytext);return NUM;}
{digit}+.{digit}+ {yylval = strdup(yytext);return FLOAT;}
{alpha}({alpha}|{digit})* {yylval = strdup(yytext);return ID;}
{alpha}({alpha}|{digit})*"\.h"?  {yylval = strdup(yytext);return H;}
\".*\" {yylval = strdup(yytext);return STRING;}
"<" {yylval = strdup(yytext);return T_lt;}
">"     {yylval = strdup(yytext);return T_gt;}
"="     {yylval = strdup(yytext);return T_eq;}
"<="    {yylval = strdup(yytext);return T_lteq;}
">="    {yylval = strdup(yytext);return T_gteq;}
"=="    {yylval = strdup(yytext);return T_eqeq;}
"!="    {yylval = strdup(yytext);return T_neq;}
"+"     {yylval = strdup(yytext);return T_pl;}
"-"     {yylval = strdup(yytext);return T_min;}
"*"     {yylval = strdup(yytext);return T_mul;}
"/"     {yylval = strdup(yytext);return T_div;}
"++"    {yylval = strdup(yytext);return T_incr;}
"--"    {yylval = strdup(yytext);return T_decr;}
"!"     {yylval = strdup(yytext);return T_neq;}
"||"    {yylval = strdup(yytext);return T_or;}
"&&"    {yylval = strdup(yytext);return T_and;}

.    return yytext[0];

%%
```

Phase 1: (b)Syntax Analysis

- Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language grammar.
- The design implementation supports
  1. Variable declarations and initializations
  2. Variables of type int,float and char
  3. Arithmetic and boolean expressions
  4. Postfix and prefix expressions
  5. Constructs - **if-else,while loop and for loop**
- Yacc tool is used for parsing. It reports shift-reduce and reduce-reduce conflicts on parsing an ambiguous grammar.

The following is the CFG used -

```
S
      : START
      ;

START
      : INCLUDE T_lt H T_gt MAIN
      | INCLUDE "\"" H "\"" MAIN
      ;

MAIN
      : VOID MAINTOK BODY
      | INT MAINTOK BODY
      ;

BODY
      : '{' C '}'
      ;

C
      : C statement ';'
      | C LOOPS
      | statement ';'
      | LOOPS
      ;

LOOPS
      : WHILE '(' COND ')' LOOPBODY
      | FOR '(' ASSIGN_EXPR ';' COND ';' statement ')' LOOPBODY
      | IF '(' COND ')' LOOPBODY
      | IF '(' COND ')' LOOPBODY ELSE LOOPBODY
      ;
```

```
LOOPBODY
	: '{' LOOPC '}'
	| ';'
	| statement ';'
	;

LOOPC
	: LOOPC statement ';'
	| LOOPC LOOPS
	| statement ';'
	| LOOPS
	;

statement
	: ASSIGN_EXPR
	| EXP
	| TERNARY_EXPR
	| PRINT
	;

COND
	: LIT RELOP LIT
	| LIT
	| LIT RELOP LIT bin_boolop LIT RELOP LIT
	| un_boolop '(' LIT RELOP LIT ')'
	| un_boolop LIT RELOP LIT
	| LIT bin_boolop LIT
	| un_boolop '(' LIT ')'
	| un_boolop LIT
	;

ASSIGN_EXPR
	: ID T_eq EXP
	| TYPE ID T_eq EXP
	;

EXP
	: ADDSUB
	| EXP T_lt ADDSUB
	| EXP T_gt ADDSUB
	;

ADDSUB
	: TERM
	| EXP T_pl TERM
	| EXP T_min TERM
	;

TERM
	: FACTOR
	| TERM T_mul FACTOR
	| TERM T_div FACTOR
	;
```

```
FACTOR
      : LIT
      | '(' EXP ')'
      ;

PRINT
      : COUT T_lt T_lt STRING
      | COUT T_lt T_lt STRING T_lt T_lt ENDL
      ;
LIT
      : ID
      | NUM
      ;
TYPE
      : INT
      | CHAR
      | FLOAT
      ;
RELOP
      : T_lt
      | T_gt
      | T_lteq
      | T_gteq
      | T_neq
      | T_eqeq
      ;

bin_boolop
      : T_and
      | T_or
      ;

un_arop
      : T_incr
      | T_decr
      ;

un_boolop
      : T_not
      ;
```

Phase 2: Symbol table with expression evaluation

- A structure is maintained to keep track of the variables, constants, operators and the keywords in the input. The parameters of the structure are the name of the token, the line number of occurrence, the category of the token (constant, variable, keyword,operator),the value that it holds the datatype.

```
typedef struct symbol_table
  {
    int line;
    char name[31];
    char type;
    char *value;
    char *datatype;
  }ST;
```

- As each line is parsed, the actions associated with the grammar rules is executed. Symbol tables functions such as lookup, search_id, update and get_val are called appropriately with each production rule.
- $1 is used to refer to the first token in the given production and $$ is used to refer to the resultant of the given production.
- Expressions are evaluated and the values of the used variables are updated accordingly.
- At the end of the parsing, the updated symbol table is displayed.

For the following input, the corresponding symbol table generated is shown:

```
#include<stdio.h>
void main()
{
        int a = 4 * 5 / 2;
        int b = a * 7;

        int c = a / b + 8 / 4;
        int d = a + b * c;
        b = 100 * 100 - d + c;

}

```

```
INPUT ACCEPTED.
Parsing Complete
Number of entries in the symbol table = 19

---------------------------------Symbol Table---------------------------------

S.No    Token       Line Number   Category     DataType    Value
1       int           8           keyword       NULL       (null)
2       4             7           constant      NULL       (null)
3       5             4           constant      NULL       (null)
4       *             9           operator      NULL       (null)
5       2             4           constant      NULL       (null)
6       /             7           operator      NULL       (null)
7       a             4           identifier    int          10
8       =             9           operator      NULL       (null)
9       7             5           constant      NULL       (null)
10      b             9           identifier    int          9852
11      8             7           constant      NULL       (null)
12      +             9           operator      NULL       (null)
13      c             7           identifier    int          2
14      d             8           identifier    int          150
15      100           9           constant      NULL       (null)
16      -             9           operator      NULL       (null)
17      void          2           keyword       NULL       (null)
18      main()        2           keyword       NULL       (null)
19      #include      1           keyword       NULL       (null)
```

Phase 3: Abstract Syntax Tree

A tree structure representing the syntactical flow of the code is generated in this phase. For expressions associativity is indicated using the %left and %right fields. Precedence of operations - last rule gets higher precedence and hence it is:

```
%left T_lt T_gt
%left T_pl T_min
%left T_mul T_div
```

To build the tree, a structure is maintained which has pointers to its children and a container for its data value.

```
typedef struct Abstract_syntax_tree
  {
      char *name;
      struct Abstract_syntax_tree *left;
      struct Abstract_syntax_tree *right;
  }node;
```

When every new token is encountered during parsing, the buildTree function takes in the value of the token, creates a node of the tree and attaches it to its parent(head of the reduced production). When the head production of the construct is reached the printTree function displays the tree for it. A node named SEQ is used to connect consecutive statements in the construct that are not related.

Sample Input 1:

```
if ( a < b )
      {
            a = 10;
            b = 2 * 3;
            a = 0;
      }
```

Sample Output 1:

```
( IF ( <  a  b )( SEQ ( SEQ ( =  a  10 )( =  b ( *  2  3 )))( =  a  0 )))
```

Sample Input 2:

```
#include<stdio.h>
void main()
{

        int a = 4 * 5 / 2;
        int b = a * 7;

        while( a>b ){
                a = a+1;
        }

        int x = 20*a;

        if( b <= x ){
                a = 10;
        }
        c = 10;
        a = 100;
        int i = 1;
                if( a > 0)
                {
                        i = 2;
                }

        int y = a+b;


}
```

Sample Output 2:

```
vivek@vivek-virtual-machine: ~/Desktop/cd/AST
vivek@vivek-virtual-machine:~/Desktop/cd/AST$ ./a.out
( a  = (( 4  *  5 ) /  2 ))
-----------------------------------------------------------
( b  = ( a  *  7 ))
-----------------------------------------------------------
(( a  >  b ) WHILE ( a  = ( a  +  1 )))
-----------------------------------------------------------
( x  = ( 20  *  a ))
-----------------------------------------------------------
(( b  <=  x ) IF ( a  =  10 ))
-----------------------------------------------------------
( c  =  10 )
-----------------------------------------------------------
( a  =  100 )
-----------------------------------------------------------
( i  =  1 )
-----------------------------------------------------------
(( a  >  0 ) IF ( i  =  2 ))
-----------------------------------------------------------
( y  = ( a  +  b ))
-----------------------------------------------------------
Input accepted.
Parsing Complete
```

Phase 4: Intermediate Code Generation (ICG)

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation. Intermediate code tends to be machine independent code.

Three-Address Code –
A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z, here x, y, z will have an address (memory location).

Example – The three address code for the expression a + b * c + d :

```
T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d
```

T1, T2, T3 are temporary variables.

The data structure used to represent Three address Code is the Quadruples. It is shown with 4 columns- operator, operand1, operand2, and result.

Sample Input:

```
#include<stdio.h>
void main()
{
		int i;

		int b = a*b;

		while( a > b ){
				a = a+1;
		}
		if( b < = c ){

				a = 10;
		}
		else{

				a = 20;
		}

		a = 100;

	for(i=0;i<10;i = i+1){

			a = a+1;
	}

}
```

Sample Output:

1. Three Address Code

```
T0 = a * b
b = T0
L0:
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
L1:
T4 = b < = c
T5 = not T4
if T5 goto L3
a = 10
goto L4
L3:
a = 20
L4:
a = 100
i = 0
L5:
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
L8:
T8 = i + 1
i = T8
goto L5
L7:
T9 = a + 1
a = T9
goto L8
L6:
Input accepted.
Parsing Complete
```

2. Quadruples

--------------------Quadruples-----------------------

Operator	Arg1	Arg2	Result
*	a	b	T0
=	T0	(null)	b
Label	(null)	(null)	L0
>	a	b	T1
not	T1	(null)	T2
if	T2	(null)	L1
+	a	1	T3
=	T3	(null)	a
goto	(null)	(null)	L0
Label	(null)	(null)	L1
<=	b	c	T4
not	T4	(null)	T5
if	T5	(null)	L3
=	10	(null)	a
goto	(null)	(null)	L4
Label	(null)	(null)	L3
=	20	(null)	a
Label	(null)	(null)	L4
=	100	(null)	a
=	0	(null)	i
Label	(null)	(null)	L5
<	i	10	T6
not	T6	(null)	T7
if	T7	(null)	L6
goto	(null)	(null)	L7
Label	(null)	(null)	L8
+	i	1	T8
=	T8	(null)	i
goto	(null)	(null)	L5
Label	(null)	(null)	L7
+	a	1	T9
=	T9	(null)	a
goto	(null)	(null)	L8
Label	(null)	(null)	L6

Phase 5: Code Optimization

The code optimizer maintains a key-value mapping that resembles the symbol table structure to keep track of variables and their values (possibly after expression evaluation). This structure is used to perform constant propagation and constant folding in sequential blocks followed by dead code elimination.

Sample Input(Quadruples)

```
= 3 NULL a
+ a 5 b
+ a b c
* c e d
= 8 NULL a
* a 2 f
if x NULL L0
```

Sample Input(3 Address Code)

```
a = 3
b = a + 5
c = a + b
d = c * e
a = 8
f = a * 2
if ( x ) L0:
```

Sample Output:

```
Python 3.5.2 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:18:55) [MSC v.1900 64 bit (AM
D64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
================= RESTART: C:/Users/sandy/Desktop/codeopt.py =================
Quadruple form after Constant Folding
-------------------------------------
= 3 NULL a
= 8 NULL b
= 11 NULL c
* 11 e d
= 8 NULL a
= 16 NULL f
if x NULL L0


Constant folded expression -
--------------------
a = 3
b = 8
c = 11
d = 11 * e
a = 8
f = 16
if x goto L0


After dead code elimination -
------------------------------
d = 11 * e
if x goto L0
>>>
```

Phase 6 – Assembly Code Generation

This phase is used to produce target codes for three-address statements produced in the Intermediate-code generation phase.

Operations:
1. Load (from memory) (LDR Dest(Reg), Src(memloc))
2. Store (to memory) (STR Dest(memloc), Src(Reg))
3. Move (between registers) (MOV R1, R2)
4. Computations (op, dest, src1, src2)
1. ADD
2. SUB
3. MUL
4. DIV
5. Unconditional jumps (BR L)
6. Conditional jumps (Bcond R, L)
cond : LZ, GZ, EZ, LEZ, GEZ, NE

Example:
```
i=0
T0=a * b
b=T0
```

Output:

```
MOV R0,#0
LDR R1,a
LDR R2,b
MUL R2,R1,R2
```

Input sample file:

```
i = 0
T0 = a * b
b = T0
L0:
T1 = a > b
T2 = not T1
if T2 goto L1
T3 = a + 1
a = T3
goto L0
L1:
T4 = b <= c
T5 = not T4
if T5 goto L3
a = 10
goto L4
L3:
a = 20
L4:
a = 100
i = 0
L5:
T6 = i < 10
T7 = not T6
if T7 goto L6
goto L7
L8:
T8 = i + 1
i = T8
goto L5
L7:
T9= a + 1
a = T9
goto L8
L6:
```

```
C:\Users\Hp\Desktop>python a.py
MOV R0,#0
LDR R1,a
LDR R2,b
MUL R2,R1,R2
L0:
SUB R3,R1,R2,
BLZ R3,L1
ADD R1,R1,#1
BR  L0
L1:
LDR R3,c
SUB R4,R2,R3,
BGEZ R4,L3
MOV  R1 # 10
BR  L4
L3:
MOV  R1 # 20
L4:
MOV  R1 # 100
MOV  R0 # 0
L5:
SUB R4,R0,#10,
BGZ R4,L6
BR  L7
L8:
ADD R0,R0,#1
BR  L5
L7:
ADD R1,R1,#1
BR  L8
L6:
```

RESULTS AND POSSIBLE SHORTCOMINGS:

Thus, we have seen the design strategies and implementation of the different stages involved in building a mini compiler and successfully built a working compiler that generates an intermediate code, given a C++ code as input.

There are a few shortcomings with respect to our implementation. The symbol table structure is same across all types of tokens (constants, identifiers and operators). This leads to some fields being empty for some of the tokens. This can be optimized by using a better representation.

The Code optimizer does not work well when propagating constants across branches (At if statements and loops). It works well only in sequential programs. This needs to be rectified.

FUTURE ENHANCEMENTS:

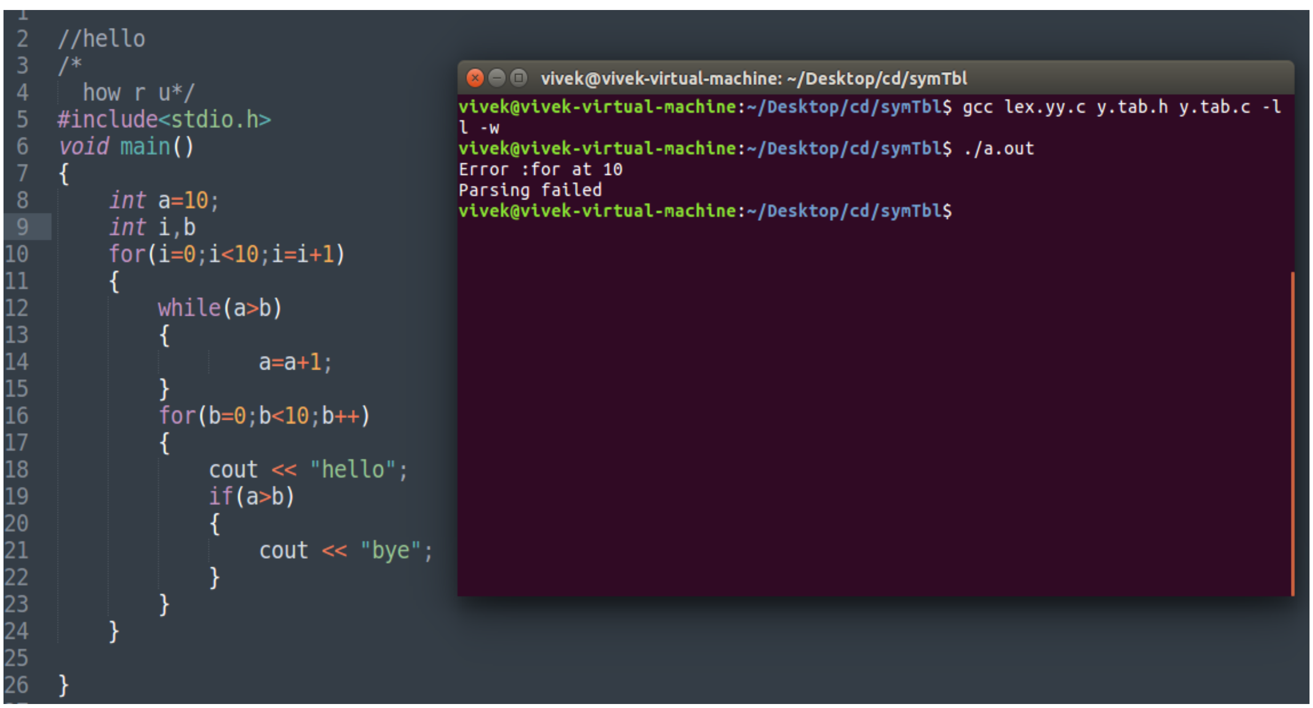
As mentioned above, we can use separate structures for the different types of tokens and then declare a union of these structures. This way, memory will be properly utilized.

For constant propagation at branches, we need to implement SSA form of the code. This will work well in all cases and yield the right output.

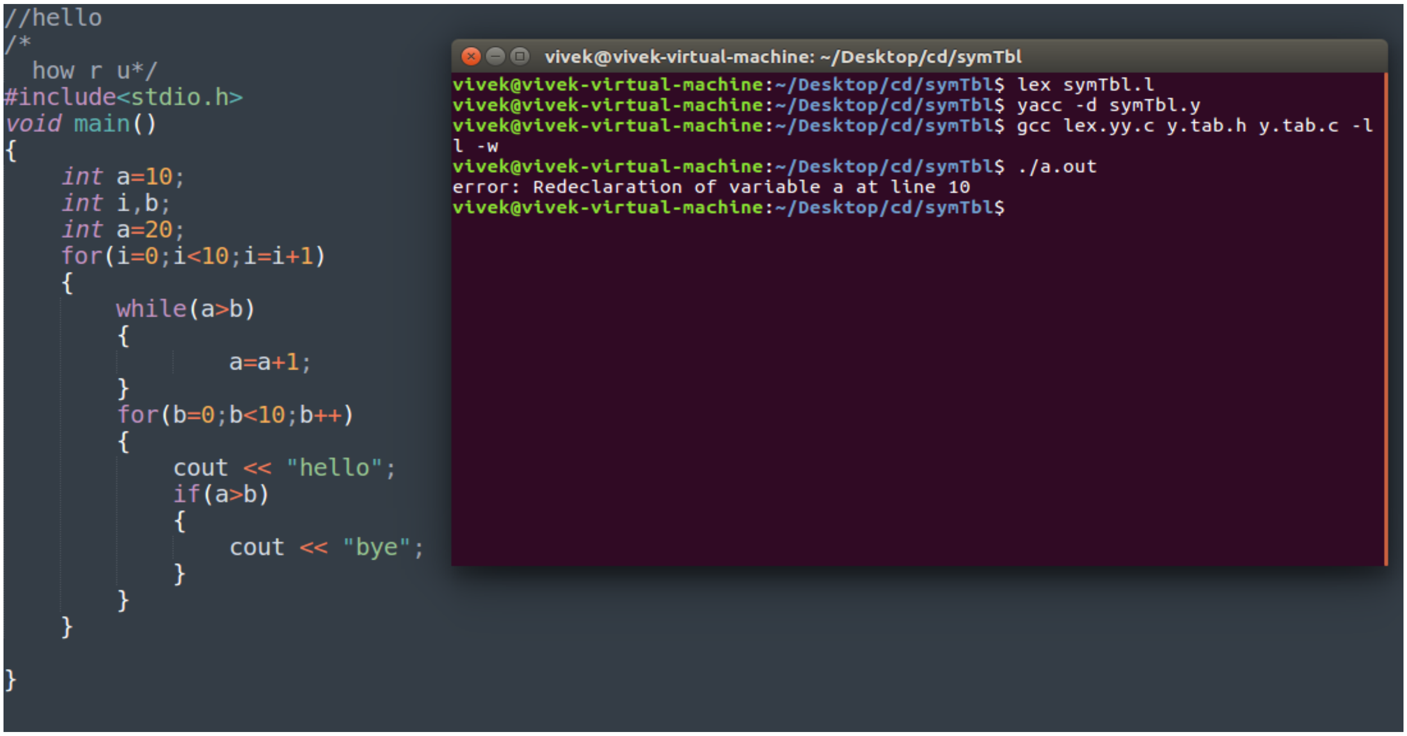
Snapshots:

This shows the detection of an undeclared variable

This shows the detection of invalid syntax at line 10



This shows detection of redeclaration of a variable



References and Bibliography

https://www.lysator.liu.se/c/ANSI-C-grammar-y.html
http://cse.iitkgp.ac.in/~bivasm/notes/LexAndYaccTutorial.pdf
http://dinosaur.compilertools.net/
https://www.javatpoint.com/code-generation
https://web.cs.ucdavis.edu/~pandey/Teaching/ECS142/Lects/final.codegen.pdf