Report on

"Mini-compiler in C for code in C++ language"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Niharika Pentapati	**PES1201700215**
Shivangi Gupta	**PES1201700274**
Greeshma Karanth	**PES1201700407**

Under the guidance of

Prof. C O Prakasha
Compiler Design
PES University, Bengaluru

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

The mini-compiler has been built in C for code in C++ language.

Given Input:

```
1 #include<iostream>
2 using namespace std;
3 int main()
4 {
5         float x,y;
6         x=5;
7         int z;
8         int a,b,c;
9         a = 10;
10        b = 45-a/2;
11        c = b*(200/a+2);
12
13        //--z;
14        if(x>=10)
15        {
16                y = a+b;
17                z = 10;
18        }
19        else
20        {
21                a=b;
22                cout<<"Successful";
23        }
24
25        for(y=10;y<=5;y++)
26        {
27                w=a*y;
28                //--b;
29        }
30        int t;
31        while(x<10)
32        { //x++;
33                for(y=10; y>=10; y++)
34                {
35                        x=x+2;}
36        }
37        //cout<<x+y;
38        x=10;
39 }
```

Final Output:

```
1 .text
2 MOV R0,#45
3 MOV R1,=t0
4 MOV R2,[R1]
5 MOV R3,=t1
6 MOV R4,[R3]
7 SUBS R4,R0,R2
8 STR R4, [R3]
9 MOV R5,=t1
10 MOV R6,[R5]
11 MOV R7,=t3
12 MOV R8,[R7]
13 MOV R9,=t4
14 MOV R10,[R9]
15 MUL R10,R6,R8
16 STR R10, [R9]
17 MOV R11,=t5
18 MOV R12,[R11]
19 MOV R0,=
20 MOV R1,[R0]
21 CMP R12,R1
22 MOV R2,#10
23 MOV R3,=t1
24 MOV R4,[R3]
25 MOV R5,=t6
26 MOV R6,[R5]
27 ADD R6,R2,R4
28 STR R6, [R5]
29 goto L1
30 MOV R7,=a
31 MOV R8,[R7]
32 MOV R9,#t6
33 STR R9, [R7]
34 MOV R10,=y
35 MOV R11,[R10]
36 MOV R12,#t6
37 STR R12, [R10]
38 MOV R0,=t6
39 MOV R1,[R0]
40 MOV R2,#5
41 MOV R3,=t7
42 MOV R4,[R3]
43 STR R4, [R3]
```

ARCHITECTURE

The main constructs that have been handled in this project include:

- if-else
- while
- for

For every construct that has been focused on, the syntax is the same as C++. The implementation of the semantics is ensured to be about as close to the C++ 14 standard as possible. Many error checking mechanisms have been added as well, for example, missing semicolons, missing bracket/parenthesis, etc.

Variable declaration of any inbuilt datatype, variable assignment/initialization, evaluation of expressions, nested loops/conditions have also been taken care of in an intricate manner.

Lex and Yacc tools have been used to accurately implement these functionalities.

CONTEXT FREE GRAMMAR

```
start : begin bopen bclose bropen stmt brclose;

bopen : b_open;

bclose : b_close;

bropen : br_open;

stmt : stmt statement | stmt vardec | ;

vardec : dt ID dec ;

dec: without_init | with_init;

without_init : eos | var without_init;

var : com ID;
```

```
with_init : eq num more;

more: eos | com ID with_init;

statement : statement ctrl_stmt | statement cond | statement
print_stmt | ;

ctrl_stmt : if | iter_stmt;

if  : matched  | unmatched;

matched : tif b_open cond b_close matched telse matched |
n_statement;

n_statement: br_open statement br_close;

unmatched : tif b_open cond b_close br_open statement br_close | tif
b_open cond b_close matched telse unmatched;

iter_stmt : for | while;

while : twhile bopen cond bclose bropen statement br_close;

for : tfor bopen prfx with_init cond loop_opn bclose bropen statement
br_close;

prfx: dt ID | ID;

cond : exp | loop_opn cond | eos ;

exp : val operator cond | val | val eos | ID eq assignexp  ;

assignexp: num op num ;

val: ID | num;

operator : relop | op | eq ;

op: addop | mulop | divop | subop;

loop_opn : incop ID | ID incop | ID combop val;

print_stmt : tcout coutop sent;

sent : cond | s eos;
```

```
eos : semi;

brclose : br_close;

ID : [A-Za-z] ;

num : [0-9] ;

b_open : { ;

b_close : } ;

br_open : ( ;

br_close : ) ;

dt : int | char | float | long | double ;

semi : ';' ;

eq : = ;

com : , ;

tif : if ;

telse : else ;

relop : < | > | <= | >= ;

addop : + ;

mulop : * ;

divop : / ;

subop : - ;

incop : ++ ;

tfor : for ;

twhile : while ;

tcout : cout ;
```

```
coutop : << ;

combop : += | -= | /= | *= ;
```

DESIGN STRATEGIES AND IMPLEMENTATION DETAILS

Symbol Table Creation

An array of structure has been used to implement the symbol table, with the help of an installID() function which is called every time a new token that's an identifier has been recognized. It is created on the Heap and is called to a print function just before the compilation ends. The print function outputs a formatted symbol table to STDOUT.

Each entry of the symbol table has the following structure:

1. char Name[20]
2. char value[50]
3. int line_number
4. int scope (0 for Main and >0 for extraneous scopes)
5. char datatype[20]
6. int id_value

All of these are displayed in a tabular format at the end of the program to represent the symbol table that has been generated during the first phase.

The entire table is stored in a dynamic table-esque data structure.

Error Handling

This compiler delicately handles some basic syntax errors such as missing punctuations and parenthesis. *Panic-mode recovery* has been implemented so that it continues parsing despite finding an error. It reports the type of error and the line number that the error has been encountered in.

Another error that has been handled is usage of a variable that has not been declared. Any such variable that is being used in the code without declaration, does not have an entry in the official symbol table. Hence, any attempt at trying to access a variable, scans the symbol table and throws an error to STDOUT if it has not been found.

All these errors are efficiently handled in this compiler with almost zero inaccuracies.

Abstract Syntax Tree

The Abstract Syntax Tree is generated entirely using Yacc. Parent nodes in an AST represent either an operator, such as =, +, or a keyword like if, while etc. The children of the tree represent statements that correspond to the parent

The tree generated is stored as a series of node structures.

The tree Node has:

1. token
2. name
3. value
4. dataType
5. lineNo
6. scope

Every production is evaluated to form the tree. So, as soon as a non-terminal is encountered, the expression is processed through the Yacc grammar to generate nodes in the AST. It contains all the meta-data as enumerated above. Dollar variables ($) are used in Yacc to store the various tokens and expressions that are parsed.

There is also a stack called TreeStack that tracks all the new nodes that have been created. Whenever a node is created, the top two nodes of the stack are popped and assigned as the left and right child of the new node, depending on the input parameters passed. Then, the new node is pushed onto the stack.

In this way, the tree is built in a bottom-up fashion, starting from the leaf nodes to the root node. This is because the innermost operations, i.e. those that do not rely on the result of other operations, are performed first and then the outer operations, which depend on the results of these inner operations.

To print (export) the syntax tree as a text file, all of the above information is printed to a single line for every node. Indentation is used to mimic a “tree” visualization. A pre-order traversal of the tree is also shown as output for a more accurate representation.

Intermediate Code Generation

This phase is implemented with the help of several functions and data structures that are used to generate and store the Intermediate Code. An array of structures is used to store the Quadruples generated by the compiler. A three address code format is also generated and stored into a .txt file once the quadruples have been generated.

AddQuadruple(), which takes four arguments, namely, operator, arg1, arg2 and result, is the function that is mainly responsible for adding entries to the quadruple table. Several functions are then used to accomplish the subsequent

processing. Temporary variables are generated and added to the symbol table as and when required.

A stack is used to implement the code generation for if-else conditions, while, and for loops. Every time such a statement is encountered, it enters the construct and for every new statement within it, it pushes onto the stack and a counter is incremented and once it's taken care of by AddQuadruple() it's popped out. Hence, the compiler knows the exact lines where GOTO labels will be pointed at, once the intermediate code has been generated.

A print function is then used to neatly arrange the Intermediate Code in quadruple format and print it onto STDOUT.

Optimization of the same code is also taken care of in this mini-compiler. Data-flow analysis is followed among the optimization techniques and the types of optimizations performed are, constant folding, and copy propagation. In the former, constant expressions are recognised and evaluated at compile time whereas, in the latter, the values of known constants are substituted at compile time. This is done by simply using a switch statement and identifying which lines of the previously generated quadruple code can the mentioned techniques be applied on.

This helps get rid of several temporary variables and reduce the size of the symbol table by a significant amount. After performing these optimizations, the optimized intermediate code is printed once again onto the STDOUT.

Assembly Code Generation

The assembly code is generated by taking the three address code as input from the previous phase and producing the assembly file as output. Each line from the input file is read and a ".split" operation is performed to get the number of arguments. Based on this information, assembly instructions are generated using 13 registers.

If the number of arguments equals two, the three address code can either be a GOTO or a label. For GOTO instructions, the assembly code outputs a branch

statement. In the case of labels, no changes are made and the assembly instructions are also the same.

If there are three arguments present, the address code is definitely an assignment operation. To load the constant and variable, a series of MOV operations are performed along with an STR operation.

If the number of arguments equals four, we know that it's a conditional statement. For such cases, the condition is first checked and loads the required variables or constants. Upon generating the MOV instructions, the CMP instruction with correct registers is also printed. Once this is done, the operation is printed along with GOTO labels.

Finally, if five arguments are present, the statement can be concluded as an operation and assignment statement. Based on the arguments, the variable or constant is loaded with register values and the operation argument decides which label among ADD, SUB, MUL, SDIV to be printed.

RESULTS AND SHORTCOMINGS

The compiler that has been built for this mini project is able to process C++ files with header files, print statements, arithmetic operations, basic looping and conditional constructs. It compiles and generates code for the sample input files accurately. It can detect a plethora of errors and satisfactorily compile and produce optimal assembly code.

The compiler built is a mini-compiler that does not work for all C++ code. The constructs that have been taken care of are few and the errors that have been handled do not encompass all the errors possible.

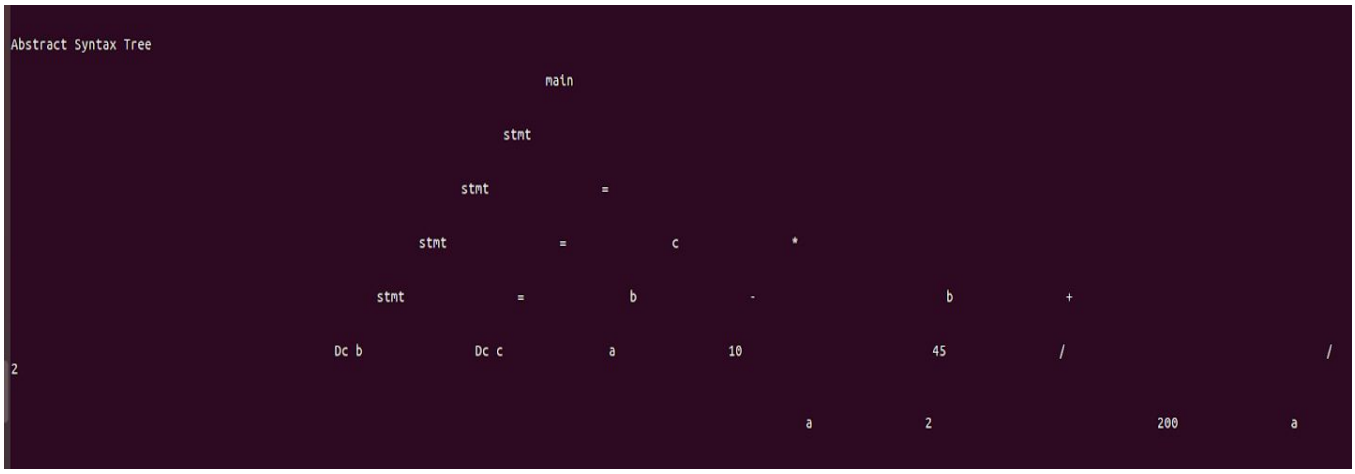
SNAPSHOTS

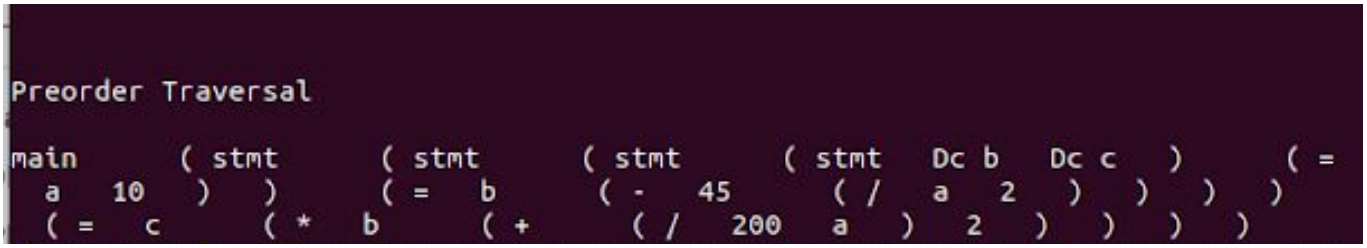
```
----------------
TOKEN GENERATOR
----------------

LINE    LEXEME                          TOKEN                   SCOPE
1       (                               BRACKET OPEN
1       )                               BRACKET CLOSE
2       {                               BRACE OPEN
3       cout                            INBUILT FUNCTION
3       <<                              OPERATOR
3       "Hello"                         STRING CONSTANT
3       ;                               TERMINATOR
4       float                           DATATYPE
4       x                               IDENTIFIER              1
4       y                               IDENTIFIER              1
4       ;                               TERMINATOR
5       x                               IDENTIFIER              1
5       =                               ASSIGNMENT
5       5                               INTEGER
```

```
---------------
SYMBOL TABLE
---------------

SNo.    Name            Identifier      Line number     Scope   Datatype        Value
1       IDENTIFIER      z               6               1       int             1714636915
2       IDENTIFIER      z               7               1       int             8
3       IDENTIFIER      b               10              1       int             2
4       IDENTIFIER      c               11              1       int             2
5       IDENTIFIER      b               11              1       int             2
6       IDENTIFIER      a               11              1       int             2
7       IDENTIFIER      z               16              2       int             13
8       IDENTIFIER      t               22              1       int             304089172
9       IDENTIFIER      y               23              1       int             521595368
10      IDENTIFIER      t               25              2       int             9
11      IDENTIFIER      x               28              1       float           10
12      IDENTIFIER      x               29              2       float           10
13      IDENTIFIER      y               30              2       int             5
14      IDENTIFIER      y               33              3       int             1
15      IDENTIFIER      x               34              3       float           15
```





Pos	Operator	Arg1	Arg2	Result
0	=	5		x
1	=	8		z
2	=	10		a
3	/	a	2	t0
4	-	45	t0	t1
5	=	t1		b
6	/	200	a	t2
7	+	t2	2	t3
8	*	b	t3	t4
9	=	t4		c
10	-	a	1	t5
11	=	t5		a

Optimized Code

0	=	5		x
1	=	8		z
2	=	10		a
3	/	10	2	5
4	-	45	5	40
5	=	40		b
6	/	200	10	20
7	+	20	2	22
8	*	40	22	880
9	=	880		c
10	-	10	1	9
11	=	9		a

```
Generating Assembly ...
x: .WORD 5
a: .WORD 10
MOV R0,#45
MOV R1,=t0
MOV R2,[R1]
MOV R3,=t1
MOV R4,[R3]
SUBS R4,R0,R2
STR R4, [R3]
b: .WORD t1
MOV R5,=t1
MOV R6,[R5]
MOV R7,=t3
MOV R8,[R7]
MOV R9,=t4
MOV R10,[R9]
MUL R10,R6,R8
STR R10, [R9]
c: .WORD t4
t5: .WORD None
MOV R11,=t5
MOV R12,[R11]
MOV R0,=
MOV R1,[R0]
CMP R12,R1
MOV R2,#10
MOV R3,=t1
MOV R4,[R3]
MOV R5,=t6
MOV R6,[R5]
ADD R6,R2,R4
STR R6, [R5]
y: .WORD t6
z: .WORD t6
goto L1
MOV R7,=a
MOV R8,[R7]
MOV R9,#t6
STR R9, [R7]
MOV R10,=y
MOV R11,[R10]
MOV R12,#t6
STR R12, [R10]
MOV R0,=t6
MOV R1,[R0]
MOV R2,#5
```

CONCLUSIONS

Since this mini-project primarily used Lex and Yacc to build a compiler for the C++ language, and this compiler was able to produce satisfactory results, it can be concluded that Lex and Yacc are powerful tools that can be used to build functional compilers for different languages. Various phases of a standard compiler can be built and implemented using these tools and by following all regulations, a standard compiler can be built for almost any language.

FURTHER ENHANCEMENTS

The mini-compiler can be developed into a full fledged compiler by extending its capabilities to handle other constructs such as arrays, functions, Object Oriented Programming concepts and STL. Since errors of a wide variety are possible in any code, the compiler can also be improved to handle as many errors as possible. In this manner, it can be ensured that the mini compiler resembles the official C++ compiler to the maximum.