Report on

"Mini-compiler for do-while and functions"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Sumanth V Rao	**01FB16ECS402**
Sumedh Pb	**01FB16ECS403**
Suraj Aralihalli	**01FB16ECS405**

Under the guidance of

Madhura V
Assistant Professor,
Dept. of CSE,
PES University, Bangalore.

January – May 2019

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

1. INTRODUCTION

C++ was the language chosen as the basis of this mini compiler. Simple constructs from the language were implemented. The frontend of the compiler includes :
1. Symbol table generation
2. Abstract Syntax tree construction **(AST)**
3. Intermediate Code generation and Code **(ICG)**
4. Optimization was implemented using flex and bison

The language the compiler is designed for is C. The properties we have implemented includes the basic structure of C which includes initializing variables, creating main and other functions, preprocessor directives limited to including **<stdio.h>** and so on.

Sample Input :

```
#include<stdio.h>
//single line comment

int copy = 80*10;
int glob1;
int fun();

int main()
{
    func(f);
    local = 1;
    if(copy > 10 )
    {
        int lhs = copy;
    }
    do
    {
        int lhs = rhs;
    }while(rhs > 2);

    if(local==1)
    {
        if(copy!=0)
        {
            int new=19;

        }
    }

    int copy2 = copy;
```

```
    // All undeclared variables initialized to 0.
    int glob2;
}
/*This function sets the
value of temp to 120*/
int fun()
{
    int temp=120;
}

$
```

Sample Output :

```
Inputs :\n
#include<stdio.h>
//single line comment

int copy = 80*10;
int glob1;
int fun();

int main()
{
     func(f);
     local = 1;
     if(copy > 10 )
     {
          int lhs = copy;
     }
     do
     {
          int lhs = rhs;
     }while(rhs > 2);
     int copy2 = copy;
     // All undeclared variables initialized to 0.
     int glob2;
}
/*This function sets the
value of temp to 120*/
int fun()
{
     int temp=120;
}
```

```
$
--------------------
After comments are removed:

#include<stdio.h>


int copy = 80*10;
int glob1;
int fun();

int main()
{
     func(f);
     local = 1;
     if(copy > 10 )
     {
          int lhs = copy;
     }
     do
     {
          int lhs = rhs;
     }while(rhs > 2);
     int copy2 = copy;

     int glob2;
}

int fun()
{
     int temp=120;
}

$
--------------------

 Symbol table generated ->


Type | Identifier | Attribute | Parameters | scope | scope no
int  copy variable           800            global     A0
int  glob1     variable           0         global     A0
int  fun  function                           A0
```

```
int  f    variable           0         main       B1
int  lhs  variable           10        main       B2
int  copy2     variable           0         main       B1
int  glob2     variable           0         main       B1
int  temp variable           120            fun   C1
```

```
ICG Successfully Created!!!

#include<stdio.h>
t1  = 80 * 10
copy = t1
glob1
int fun ()
L4: param f
call(func)
local = 1
if copy > 10 goto L1:
L1: lhs = copy

L2: lhs = rhs
if rhs > 2 goto L2:

copy2 = copy
glob2

L3: temp = 120
```

____

```
Optimized ICG Successfully Created!!!

#include<stdio.h>
copy = 800
glob1
int fun ()
L4: param f
call(func)
local = 1
```

```
if 800 > 10 goto L1:
L1: lhs = 800

L2: lhs = rhs
if rhs > 2 goto L2:

copy2 = 800
glob2

L3: temp = 120
```

Inorder traversal

```
( include (( int  global ( copy  = ( 80  *  10 ))) global_dec
(( int  ,  glob1 ) global_dec (( int  func  fun ) global (((
func  call  f )  (( local  =  1 )  ((( copy  >  10 ) IF ( int
( lhs  =  copy )))  ((( int   ( lhs  =  rhs )) do-while ( rhs
>  2 ))  (( int   ( copy2  =  copy ))  ( int   glob2 ))))))
main (( int  func  fun ) func ( int   ( temp  =  120 ))))))))
```

```
(include)
 |__(global_dec)
     |__(global)
     |   |__(int)
     |   |__(=)
     |       |__(copy)
     |       |__(*)
     |           |__(80)
     |           |__(10)
     |__(global_dec)
         |__(,)
         |   |__(int)
         |   |__(glob1)
         |__(global)
             |__(func)
             |   |__(int)
             |   |__(fun)
             |__(main)
                 |__()
                 |   |__(call)
                 |   |   |__(func)
                 |   |   |__(f)
                 |   |__()
                 |       |__(=)
                 |       |   |__(local)
```

```
|       |   |__(1)
|       |__()
|           |__(IF)
|           |   |__(>)
|           |   |   |__(copy)
|           |   |   |__(10)
|           |   |__()
|           |       |__(int)
|           |       |__(=)
|           |           |__(lhs)
|           |           |__(copy)
|           |__()
|               |__(do-while)
|               |   |__()
|               |   |   |__(int)
|               |   |   |__(=)
|               |   |       |__(lhs)
|               |   |       |__(rhs)
|               |   |__(>)
|               |       |__(rhs)
|               |       |__(2)
|               |__()
|                   |__()
|                   |   |__(int)
|                   |   |__(=)
|                   |       |__(copy2)
|                   |       |__(copy)
|                   |__()
|                       |__(int)
|                       |__(glob2)
|__(func)
    |__(func)
    |   |__(int)
    |   |__(fun)
    |__()
        |__(int)
        |__(=)
            |__(temp)
            |__(120)
```

2. ARCHITECTURE OF THE LANGUAGE

Compiler for the following constructs:

1. do while loop
2. if else statements
3. int, void, char - data types
4. basic function declaration and definition
5. arithmetic and logical operators

3. REFERENCES

Lex Yacc and its internal working :
https://www.tldp.org/HOWTO/Lex-YACC-HOWTO.html#toc1
IBM Docs on working of lex and yacc :
https://www.ibm.com/developerworks/library/l-lexyac/index.html
Expression evaluation using Abstract Syntax Tree :
https://mariusbancila.ro/blog/2009/02/03/evaluating-expressions-part-1

4. CONTEXT-FREE GRAMMAR

assignment_expression: conditional_expression| unary_expression assignment_operator assignment_expression;

statement: compound_statement| expression_statement| selection_statement| iteration_statement| jump_statement;

compound_statement: '{' '}'| '{' statement_list '}'| '{' declaration_list '}'| '{' declaration_list statement_list '}';

statement_list: statement| statement_list statement;expression_statement: ';'| expression ';';
function_definition: declaration_specifiers declarator declaration_list compound_statement| declaration_specifiers declarator compound_statement| declarator declaration_list compound_statement| declarator compound_statement;

primary_expression: IDENTIFIER | CONSTANT | STRING_LITERAL| '(' expression ')';
unary_expression: postfix_expression| INC_OP unary_expression| DEC_OP unary_expression|
unary_operator unary_expression| SIZEOF unary_expression| SIZEOF '(' type_name ')';
unary_operator: '&'| '*'| '+'| '-'| '~'| '!';

multiplicative_expression: cast_expression| multiplicative_expression '*' cast_expression|multiplicative_expression '/' cast_expression| multiplicative_expression '%' cast_expression;

additive_expression : multiplicative_expression | additive_expression '+'multiplicative_expression| additive_expression '-' multiplicative_expression;

relational_expression: shift_expression| relational_expression '<' shift_expression |relational_expression '>' shift_expression| relational_expression LE_OP shift_expression|relational_expression GE_OP shift_expression;

equality_expression : relational_expression| equality_expression EQ_OP relational_expression|
equality_expression NE_OP relational_expression;
assignment_expression: conditional_expression| unary_expression assignment_operator assignment_expression;

assignment_operator: '=';
selection_statement: IF '(' expression ')' statement| IF '(' expression ')' statement ELSE statement;

iteration_statement : WHILE '(' expression ')' statement| DO statement WHILE '(' expression ')'';'

---

5. DESIGN STRATEGY

● **Symbol table creation** - The symbol table contents are was implemented in a linear array structure :

1. Type - *int/float*
2. Identifier - *name of the identifier*
3. Attribute - *whether it is a variable / function.*
4. parameters - *value it holds.*
5. scope - *global / local*
6. scope count - *Incrementing numbers and alphabet indicating the current scope of the identifier.*

The structure of the symbol table looks like this ->

```
struct symtab
{
char identifier[20];
char type[20];
char attribute[20];
int val;
char pars[100];
```

```
char scope[20];
int spec;
};
```

● **Abstract Syntax Tree** - This tree is constructed as the input is parsed. We print the inorder traversal as well as a hierarchical structure as the output. The structure looks like ->

```
typedef struct node
{
	struct node* left;
	struct node *right;
	char* token;
} node;
```

● **Intermediate Code Generation** - Intermediate code was generated that makes use of temporary variables and labels. Also all if-else statements were optimized to if-False statements to reduce the number of **goto** statements (an additional optimization provided).

● **Code Optimization** - Constant folding and Constant propagation were implemented as part of machine independent code optimization.

a) Constant Folding

When an arithmetic expression is encountered, we check to see if all the operands contain digits and are not identifiers. If all the operands are numbers we evaluate the expression.

b) Constant Propagation

When an identifier is encountered, we check the symbol table to see if an entry exists. If the entry exists we perform constant propagation.

● **Error handling** - Type error and semicolon missing error have been handled.

---

6. IMPLEMENTATION DETAILS

Lex and Yacc were used to implement the following:

● **Symbol table creation** - Implemented in **symtab.y**
The symbol table is a linear array of the following structure

● **Abstract Syntax Tree** - Implemented in **ast.y**
To implement this in lex yacc, we first redefine the YYSTYPE in the yacc file that defaults to int.

● **Intermediate Code Generation** - Implemented in **icg.y**
The given code was converted into 3 address code

● **Code Optimization** - Implemented in opt.y

a) Constant Folding

Using the function all digits we perform this check. If the operands are all numbers. We evaluate the expression immediately and store it in the symbol table.

b) Constant Propagation

When an identifier is found in symbol table the synthesised attribute of the non-terminal, code stores the value of the identifier, else it stores the name of the identifier itself.

● **Error Handling** - Implemented in symtab.y using conditional statements for type error and semicolon missing is checked using the grammar

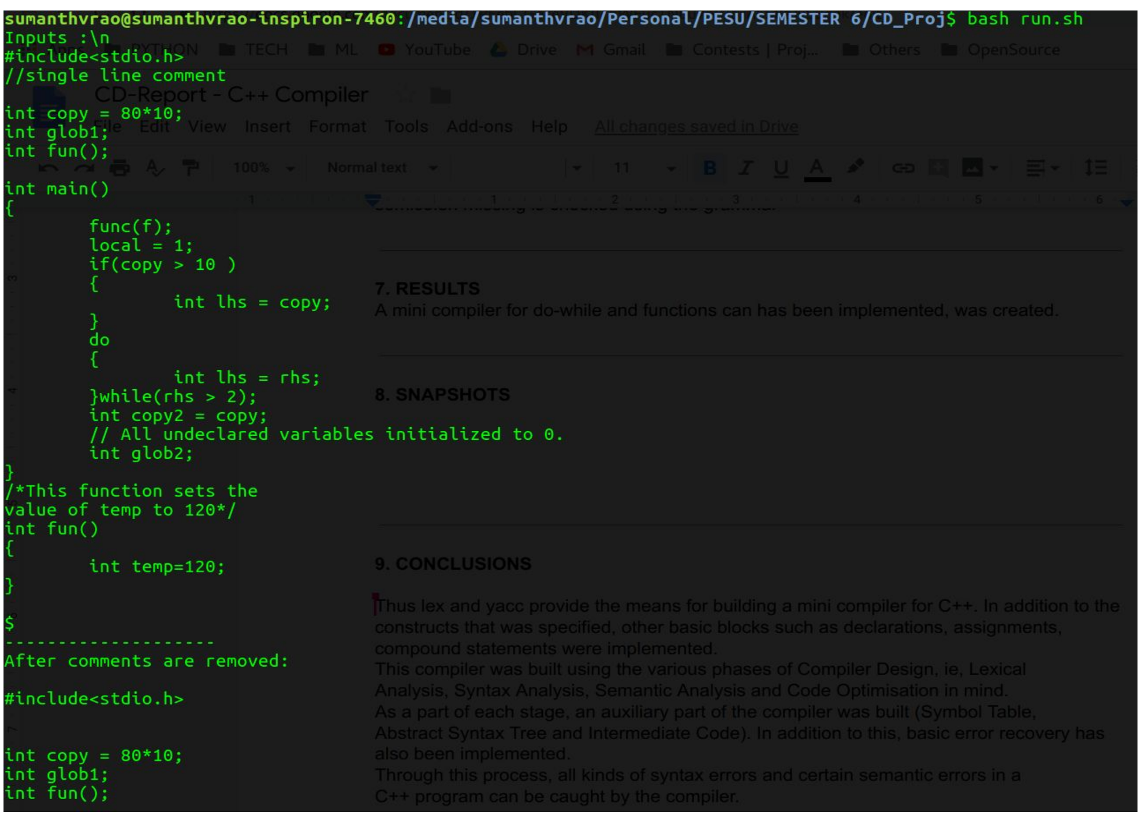
---

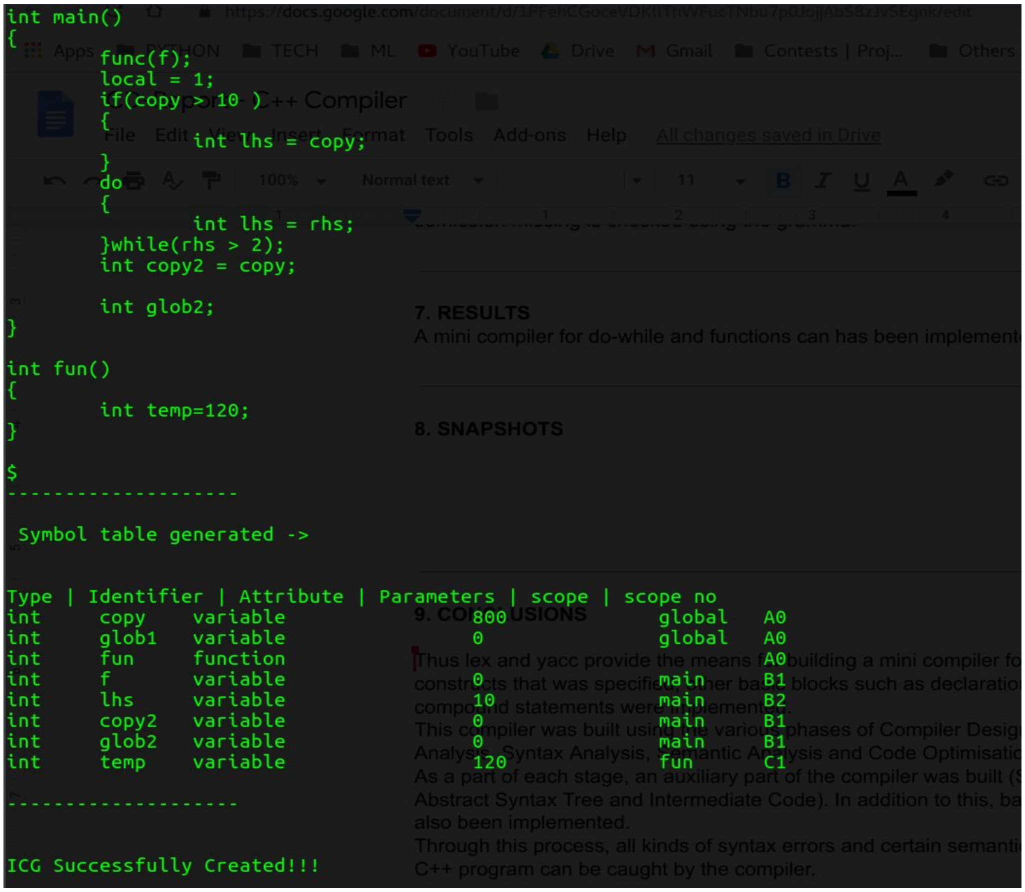
7. RESULTS

A mini compiler for do-while and functions can has been implemented, was created.

---

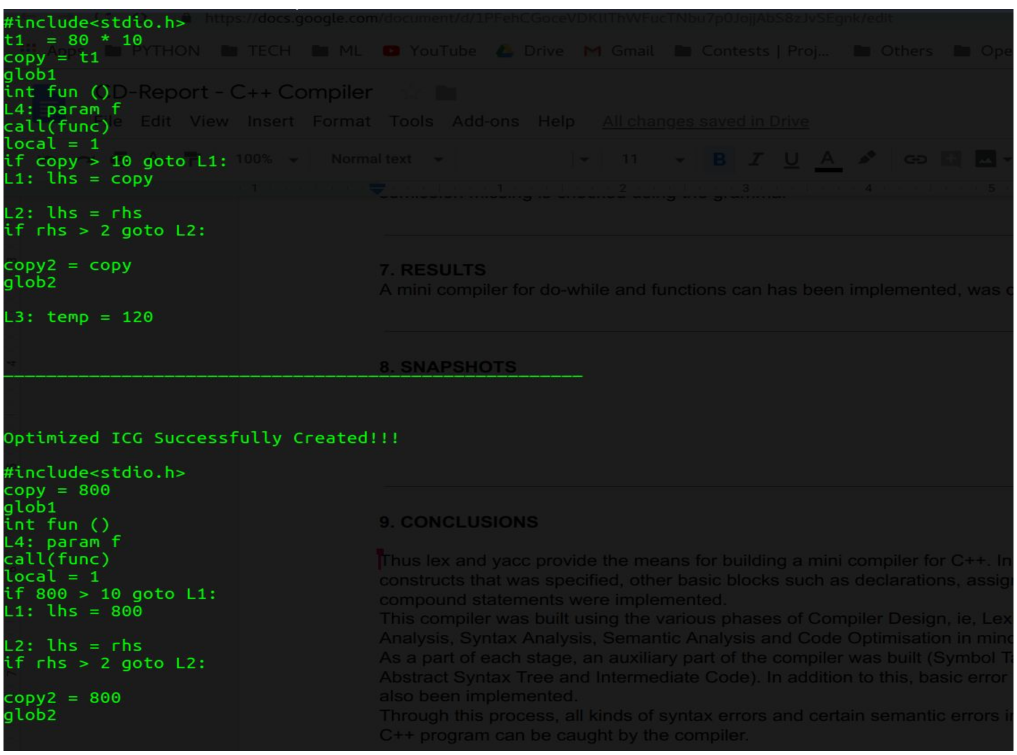
8. SNAPSHOTS

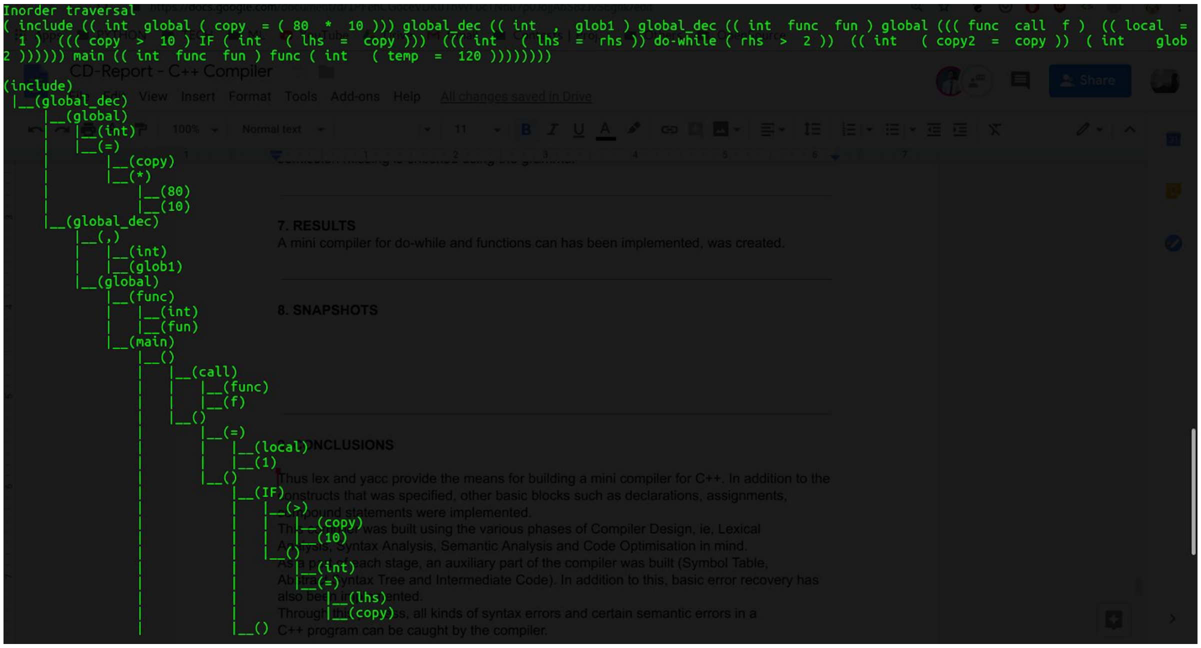



int main()
{
func(f);
local = 1;
if(copy > 10)
{
int lhs = copy;
}
do
{
int lhs = rhs;
}while(rhs > 2);
int copy2 = copy;
int glob2;
}
int fun()
{
int temp=120;
}
$

Symbol table generated ->
Type | Identifier | Attribute | Parameters | scope | scope no
int copy variable 800 global A0
int glob1 variable 0 global A0
int fun function A0
int f variable 0 main B1
int lhs variable 10 main B2
int copy2 variable 0 main B1
int glob2 variable 0 main B1
int temp variable 120 fun C1

ICG Successfully Created!!!


#include<stdio.h>
t1 = 80 * 10
copy = t1
glob1
int fun ()
L4: param f
call(func)
local = 1
if copy > 10 goto L1:
L1: lhs = copy
L2: lhs = rhs
if rhs > 2 goto L2:
copy2 = copy
glob2
L3: temp = 120
Optimized ICG Successfully Created!!!
#include<stdio.h>
copy = 800
glob1
int fun ()
L4: param f
call(func)
local = 1
if 800 > 10 goto L1:
L1: lhs = 800
L2: lhs = rhs
if rhs > 2 goto L2:
copy2 = 800
glob2



9. CONCLUSIONS

Thus lex and yacc provide the means for building a mini compiler for C++. In addition to the constructs that was specified, other basic blocks such as declarations, assignments, compound statements were implemented.
This compiler was built using the various phases of Compiler Design, ie, Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Optimisation in mind.
As a part of each stage, an auxiliary part of the compiler was built (Symbol Table, Abstract Syntax Tree and Intermediate Code). In addition to this, basic error recovery has also been implemented.
Through this process, all kinds of syntax errors and certain semantic errors in a C++ program can be caught by the compiler.

10. FURTHER ENHANCEMENTS

- Currently, we use a marker for the end symbol - $ was used. We could replace this in the future.
- More error recovery methods can be incorporated.