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:

Tejas Srinivasan	**PES1201800110**
Arjun Chengappa	**PES1201800119**
Lavitra Kshitij Madan	**PES1201800137**
Joseph Dominic Cherukara	**PES1201800328**

Under the guidance of

Preet Kanwal
Associate Professor
PES University, Bengaluru

January – May 2021

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

CHAPTER 1

INTRODUCTION

We have designed a mini C++ compiler christened '++g' which is in accordance with our objective of generation of an optimized representation of intermediate code. The optimised code can be converted to machine code and can be converted back from this representation to the intermediate form using a DAG representation.

The phases handled by the compiler are -

- Token recognition and generation from the lex file
- Syntax validation in the yacc file
- Semantic analysis and updation of the symbol table
- Scoping using gpt and stack
- Intermediate code generation
- Code Optimizations (4 implemented as will be discussed in detail in the respective section).

To ensure that our compiler works on many edge cases we tried to cover different tests in our input file and ensured that all phases were functional.

CHAPTER 2

ARCHITECTURE OF LANGUAGE

2.1 The constructs we were assigned were ‘while’ and ‘switch’ and the following list of operators have been taken care of -

- Arithmetic operators
- Logical operators
- Assignment operators
- Unary operators
- Relational operators

Error handling and recovery are taken care of by yyerror in the yacc file.
Syntax errors and semantic errors that are required as per the guidelines document are all checked for.

CHAPTER 3

LITERATURE SURVEY

3.1 Documentation on lexer - https://westes.github.io/flex/manual/

Documentation on parser -
https://www.ibm.com/docs/en/zos/2.2.0?topic=tools-generating-parser-using-yacc

Mid-rule actions:
https://www.gnu.org/software/bison/manual/html_node/Typed-Mid_002dRule-Actions.html#Typed-Mid_002dRule-Actions
https://www.gnu.org/software/bison/manual/html_node/Mid_002dRule-Action-Translation.html#Mid_002dRule-Action-Translation
yacc flags:
https://pubs.opengroup.org/onlinepubs/009604599/utilities/yacc.html

CHAPTER 4

CONTEXT FREE GRAMMAR

```
START
    : PROG
    ;

PROG
    : T_INCLUDE '<' T_HEADER '>' PROG
    | T_DEFINE T_ID T_NUM PROG
    | T_DEFINE T_ID T_STRINGLITERAL PROG
    | T_CLASS T_ID '{' CLASS_STMT '}' PROG
    | MAIN PROG
    | FUNC_DECLR PROG
    | DECLR ';' PROG
    | ASSGN ';' PROG
    |
    ;

FUNC_DECLR
    : TYPE T_ID '(' EMPTY_LISTVAR ')' FUNC_DECLR2
    ;

EMPTY_LISTVAR
    : LISTVAR
    |
    ;

FUNC_DECLR2
```

```
    : '{' STMT '}'
    | ';'
    ;

MAIN
    : TYPE T_MAIN '(' EMPTY_LISTVAR ')' '{' STMT '}'
    | T_MAIN '(' EMPTY_LISTVAR ')' '{' STMT '}'
    ;

CLASS_STMT
    : CLASS CLASS_STMT
    | CLASS
    ;

CLASS
    : CLASS_LABEL ':'
    | DECLR2
    | FUNC_DECLR
    ;

DECLR2
    : TYPE LISTVAR2
    ;

LISTVAR2
    : LISTVAR2 ',' T_ID ARRAY
    | T_ID ARRAY
    ;

CLASS_LABEL
    : T_PUBLIC
    | T_PRIVATE
```

```
    | T_PROTECTED
    ;

STMT
    : STMT_NO_BLOCK STMT
    | BLOCK STMT
    |
    ;

STMT_NO_BLOCK
    : DECLR ';'
    | ASSGN ';'
    | WHILE
    | SWITCH
    | T_COUT T_COUT_OP  COUT ';'
    | T_CIN T_CIN_OP T_ID CIN ';'
    | T_RETURN ';'
    ;

BLOCK
    : '{' STMT '}'
    ;

SWITCH
    : T_SWITCH '(' SWITCH2
    ;

SWITCH2
    : EXPR ')' '{' SWT_BLOCK '}'
    | ASSGN ')' '{' SWT_BLOCK '}'
    ;
```

```
SWT_BLOCK
    : STMT
    | CASE SWT_BLOCK
    | T_DEFAULT ':' STMT
    ;

CASE
    : T_CASE T_ID ':' STMT BREAK
    | T_CASE T_NUM ':' STMT BREAK
    ;

BREAK
    : T_BREAK ';'
    |
    ;

WHILE
    : T_WHILE '(' WHILE_PARAN ')' WHILE2
    ;

WHILE2
    : '{' WHL_BLOCK '}'
    | ';'
    | STMT_NO_BLOCK
    ;

WHL_BLOCK
    : STMT
    | WHL_BLOCK BREAK
    ;

WHILE_PARAN
```

```
    : EXPR
    | ASSGN
    ;

COUT
    : COUT T_COUT_OP T_STRINGLITERAL
    | COUT T_COUT_OP T_ID
    | T_COUT_OP T_STRINGLITERAL
    | T_COUT_OP T_ID
    ;

CIN
    : CIN T_CIN_OP T_ID
    | T_CIN_OP T_ID
    ;

DECLR
    : TYPE LISTVAR
    ;

TYPE
    : T_VOID
    | T_INT
    | T_FLOAT
    | T_CHAR
    | T_DOUBLE
    | T_SHORT
    ;

LISTVAR
    : LISTVAR ',' VAR
    | VAR
```

```
    ;

VAR
    : T_ID ARRAY
    | T_ID '=' EXPR
    ;

ARRAY
    : ARRAY '[' T_NUM ']'
    | '[' T_NUM ']'
    |
    ;

ASSGN
    : T_ID '=' EXPR
    ;

EXPR
    : EXPR REL_OP T
    | T
    ;

T
    : T '+' F
    | T '-' F
    | F
    ;

F
    : F '*' G
    | F '/' G
    | F '%' G
```

```
    | G
    ;

G
    : '(' EXPR ')'
    | T_ID POSTFIX_OP
    | T_NUM
    | T_INCREMENT T_ID
    | T_DECREMENT T_ID
    ;

REL_OP
    : T_LESSTHANEQUAL
    | T_GREATERTHANEQUAL
    | '<'
    | '>'
    | T_EQUALTO
    | T_NOTEQUALTO
    ;

POSTFIX_OP
    : T_INCREMENT
    | T_DECREMENT
    |
    ;
```

CHAPTER 5

DESIGN STRATEGY

5.1 SYMBOL TABLE CREATION

Every time a new variable is declared, it is added to the symbol table.
The Symbol table is implemented as a linked list of structures with the below values.
The structure of each entry in the symbol table is as follows -

- Name : name of the identifier
- type : integer that stores the type of the symbol table entry (function return type and datatype)
- len : stores the length of the identifier name
- lineno : stores the line no. where the variable was declared
- scope : stores the scope of the identifier

5.2 INTERMEDIATE CODE GENERATION

We followed the steps provided to us in the guidelines and finally obtained a quadruple format of three address code representation of the file. This also involves storing the temporaries generated in the symbol table.
The structure of the quadruple format is as follows -

- OP - This field stores the operator
- ARG1 - This field stores the first argument
- ARG2 - This field stores the second argument
- RESULT - This field stores the result

5.3 CODE OPTIMIZATION

As per the requirements, 4 optimizations were implemented -

- Constant folding - It is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime.
- Constant propagation - It is the process of substituting the values of known constants in expressions at compile time.
- Common subexpression elimination - It is the process of searching for instances of identical expressions and analyzing whether it is worthwhile replacing them with a single variable holding the computed value.
- Dead code elimination - It is the process of removing code that does not affect the program results. We have implemented dead code elimination of redundant conditional jumps like iffalse.

5.4 ERROR HANDLING

Some of the error handling is done by the ‘yyerror’ function in the yacc file. Apart from this, there were also some semantic errors handled by some functions like:

- Function already defined
- Identifier undeclared before use
- Identifier redefined

CHAPTER 6

IMPLEMENTATION DETAILS

6.1 SYMBOL TABLE CREATION

Scoping is handled using a General Purpose tree. The general method of incrementing and decrementing the scope every time a scope started or ended would make the symbol table structure more complex. This is normally done using a separate symbol table for each scope level.

Instead of this strategy, we've used a separate scoping strategy thus making the symbol table structure much simpler. Every time a new scope is created, a scope variable is incremented and pushed onto a stack and inserted into the GPT. When the scope ends, the scope is popped off the stack.

The general purpose tree is then used to verify if a variable has already been defined in a given scope or if it is being used without definition in the current scope.

6.2 INTERMEDIATE CODE GENERATION

The intermediate code (TAC and QUADRUPLE format) is printed on the fly using fprintf to print to a file and sprintf to format the file. The functions are defined in a separate code file called icg_gen_server.c.

6.3 CODE OPTIMIZATION

The code optimizations are performed using different functions in a Python file as it allows easy manipulation of numbers and strings using flexible data structures like dictionaries and different formats like csvs. The code from the icg in the quadruple format is divided into

different “basic blocks” and the entry points to identify the different basic blocks are obtained from the icg. These blocks are further divided into a list of dictionaries of the same quadruple format.

6.4 ERROR HANDLING

Everytime a variable is used or created, the symbol table is checked for clashes in identifier name and use. The stack is referred to find the current and parent scopes. These scopes are checked in the symbol table to find clashes and appropriate error messages are printed.

At the end of parsing, the symbol table is checked to see if a main function has been defined. If it has not been defined, an appropriate error message is displayed.

Steps to be followed to execute our code -

The make file is as follows-

```
./a.out : y.tab.c lex.yy.c
	gcc -g y.tab.c lex.yy.c -ll -o ++g
	rm y.tab.c y.tab.h lex.yy.c
y.tab.c y.tab.h : yacc_final.y
	yacc -d -v yacc_final.y
lex.yy.c : lex.l sym_tab.c sym_tab.h y.tab.h
	lex lex.l
```

This make file has to be run using

```
make -f makefile.mk
```

and then the generated file should be run using

```
./++g < [input_file]
```

where [input file] is the file with C++ code that is to be compiled.

Helper functions used

1. Yacc file

```
void create_scope()
{
    insert_to_gpt(scope);
    fprintf(sym_tab_debug, "inserted scope: %d, scope_1: %d\n\n",
scope, scope_1);
    //display_sym_tab();
    ++scope;
    ++scope_1;
    fprintf(gpt_debug, "displaying gpt\n");
    disp_gpt();
    fprintf(gpt_debug, "displaying stack\n");
    disp_stack();
}

void remove_scope()
{
    --scope_1;
    stack_pop();
    fprintf(gpt_debug, "displaying gpt\n");
    disp_gpt();
    fprintf(gpt_debug, "displaying stack\n");
    disp_stack();
}
```

2. check_sym_tab.c

```
symbol* check_sym_tab(char* name)
{
    //printf("NAME: %s\n",name);
    s_node* s_n = s->top;                          //stack node
    while(s_n != NULL)
    {
        symbol* n = t->head;                       //symbol table node
        while(n != NULL)
```

```
        {
            //printf("n->scope: %d s_n->val->scope: %d\n", n->scope,
s_n->val->scope);
            if(n->scope == s_n->val->scope)
            {
                //printf("n->name: %s, name: %s\n",n->name, name);
                if(strcmp(n->name, name) == 0)
                    return n;
            }
            n = n->next;
        }
        s_n = s_n->next;
    }
    return NULL;
}
```

3. gpt_server.c

```
void insert_to_gpt(int scope)
{
    node* n = init_node(scope);
    if(s->top != NULL)                  //global scope has already been
created
    {
        node* parent = s->top->val;
        n->sibling = parent->child;     //storing the parent's children
as siblings to the new node
        parent->child = n;              //updating the new node as a
child to the parent
    }
    else                                //creating global scope
        g->root = n;
    stack_push(n);
}

void disp_gpt()
{
    if(g->root == NULL)
        fprintf(gpt_debug, "GPT Empty\n");
```

```
    else
        rec_disp_gpt(g->root);
    fprintf(gpt_debug, "\n\n");
}

void rec_disp_gpt(node* root)
{
    if(root == NULL)
        return;
    fprintf(gpt_debug, "%d has children: ",root->scope);
    node* c = root->child;
    while(c != NULL)
    {
        fprintf(gpt_debug, "%d\t", c->scope);
        c = c->sibling;
    }
    fprintf(gpt_debug, "\n");
    rec_disp_gpt(root->sibling);
    rec_disp_gpt(root->child);
}
```

4. sym_tab.c

```
void insert_symbol(char* name, int len, int type, int lineno, int
scope, int scope_1)
{
    symbol* s = init_symbol(name, len, type, lineno, scope, scope_1);
    if(t->head == NULL)
    {
        t->head = s;
        return;
    }
    symbol* curr = t->head;
    while(curr->next!=NULL)
        curr = curr->next;
    curr->next = s;
}
```

CHAPTER 7

RESULTS

The ++C (Mini C++ compiler) that we built generates intermediate code for the constructs we have at hand. The grammar is such that it covers a wide variety of features of the language, can semantically analyse the code and report errors in the code which we can use to accurately pinpoint if something at all has gone wrong. We also performed a series of optimizations which aided us in making an efficient compiler.

Some possible shortcomings of the Mini Compiler could be -

- The optimizations are not very strong.
- Classes and arrays are handled only by the lexer and in the syntax phase but not in the other phases.
- Functions sometimes cause some issues during optimizations.

CHAPTER 8

SNAPSHOTS

Screenshots of Inputs and Output files

Input file - test_1.c

```
void func()
{
    int a = 5;
    float b = 6.0;
    double c = 10.9;
    while(d < 5)
    {
        e = f + 10;
        while(z = 10)
        {
            y = 10;
            break;
        }
        while(x = 100);
        int z = 1000;
        int x = a / b;
    }
    switch(a)
    {
        case 1:
            b=10.0;
            break;
        case 5:
            d = 100;
            while(x = 100);
            int m;
            m = s;
```

```
        default:
            c = 2.0;
    }
}

int func()
{
    int g = 9;
    int h = 10;
    while(i < 5)
        j = 10;
}

int func2()
{
    int hmmm = 0;
}
```

Output of compiler on input of test_1.c

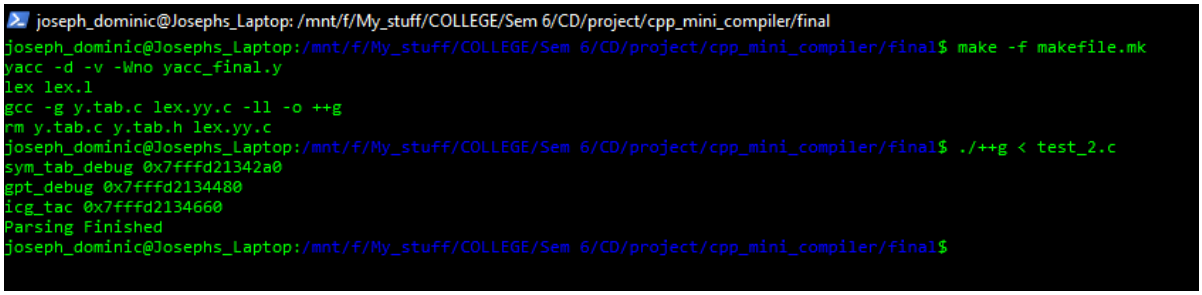
```
joseph_dominic@Josephs_Laptop: /mnt/f/My_stuff/COLLEGE/Sem 6/CD/project/cpp_mini_compiler/final
joseph_dominic@Josephs_Laptop:/mnt/f/My_stuff/COLLEGE/Sem 6/CD/project/cpp_mini_compiler/final$ make -f makefile.mk
yacc -d -v -Wno yacc_final.y
lex lex.l
gcc -g y.tab.c lex.yy.c -ll -o ++g
rm y.tab.c y.tab.h lex.yy.c
joseph_dominic@Josephs_Laptop:/mnt/f/My_stuff/COLLEGE/Sem 6/CD/project/cpp_mini_compiler/final$ ./++g < test_1.c
sym_tab_debug 0x7fffecce52a0
gpt_debug 0x7fffecce5480
icg_tac 0x7fffecce5660
On line no.: 6, Identifier 'd' not defined before use

On line no.: 8, Identifier 'f' not defined before use

On line no.: 8, Identifier 'e' not defined before use

On line no.: 9, Identifier 'z' not defined before use

On line no.: 11, Identifier 'y' not defined before use

On line no.: 14, Identifier 'x' not defined before use

On line no.: 25, Identifier 'd' not defined before use

On line no.: 26, Identifier 'x' not defined before use

On line no.: 28, Identifier 's' not defined before use

On line no.: 34, Function func already defined on 1

On line no.: 38, Identifier 'i' not defined before use

On line no.: 39, Identifier 'j' not defined before use

On line no.: 45, Function main not found
```

Input file test_2.c

test_2.c

```
1  void main()
2  {
3      int a = 5;
4      float b = 6.0;
5      double c = 10.9;
6      int d = 0;
7      while(d < 5)
8      {
9          //e = f + 10;
10         while(d = 10)
11         {
12             d = 10;
13             break;
14         }
15         while(d = 100);
16         int z = 1000;
17         int x = a / b;
18         a = a + b;
19     }
20     switch(a)
21     {
22         case 1:
23             b=10.0;
24             break;
25         case 5:
26             d = 100;
27             while(d = 100);
28             int m;
29         default:
30             c = 2.0;
31     }
32 }
33
34 int func()
35 {
36     int g = 9;
37     int h = 10;
38     //while(i < 5)
39     //  j = 10;
40 }
41
42 int func2()
43 {
44     int hmmm = 0;
45 }
```

Output of compiler on input of test_1.c



Output file in Quadruple format -

OP,ARG1,ARG2,RES
=,5,,a
*,a,5,_0
=,_0,,b
=,0,,i
label,,,L0
<,i,10,_1
if,_1,,L1
goto,,,L2
label,,,L3
+,i,1,_2
=,_2,,i
goto,,,L0
label,,,L1
<,a,10,_3
if,_3,,L4
goto,,,L5
label,,,L4
=,4,,b
goto,,,L6
label,,,L5
=,5,,b
label,,,L6
goto,,,L3
label,,,L2

Output file after 3 optimizations (Constant folding, Constant propagation and Common Subexpression Elimination)

```
a = 5
_0 = 25
b = 25
i = 0
L0:
_1 = i < 10
if _1 goto L1
goto L2
L3:
_2 = i + 1
i = _2
goto L0
L1:
_3 = a < 10
if _3 goto L4
goto L5
L4:
b = 4
goto L6
L5:
b = 5
L6:
goto L3
L2:
-----------------------------------
```

Output file after the dead code elimination

```
set()
set()
a = 5
_0 = 25
b = 25
i = 0
L0:
_1 = i < 10
if _1 goto L1
goto L2
L3:
_2 = i + 1
i = _2
goto L0
L1:
_3 = a < 10
if _3 goto L4
goto L5
L4:
b = 4
goto L6
L5:
b = 5
L6:
goto L3
L2:
```

CHAPTER 9

CONCLUSIONS

We have obtained a much deeper understanding of both the grammar and structure of the language assigned to us (i.e C++) and variations of the constructs we were assigned (while and switch). We also clearly understood the distinction between the different phases of the compiler and the transit of the code (and its different representations) between each of these phases. Finally we can conclude that we have gotten enough understanding of the process of building the different phases of the compiler to build a basic working compiler for any language given that we know its grammar rules.

CHAPTER 10

FURTHER ENHANCEMENTS

1. We can include enhancements in our grammar to make it more similar to an actual C++ compiler by covering more features in the language.
2. We can include more optimisations like moving loop invariant code outside the loop, temporary packing and other such optimisations.
3. We can include the functionality of taking input from multiple files especially since classes and objects can be present in separate files in a C++ implementation.
4. Furthermore a more extensive documentation of all the files would be really helpful in making our code more readable and easier to execute.

Some of the enhancements mentioned are out of scope of the project and hence weren't implemented. The rest were avoided to keep the implementation of the code minimal with the time we had in hand.

REFERENCES/BIBLIOGRAPHY

1. C++ ISO 2013 standard - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf
2. Lex and Yacc tutorial by Tom Niemann - https://lafibre.info/images/doc/201705_lex_yacc_tutorial.pdf
3. Yacc error message handling - https://www.epaperpress.com/lexandyacc/err.html
4. Error recovery in bison - http://dinosaur.compilertools.net/bison/bison_9.html
5. Implementing translation schemes in Yacc - https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Translation.html/node5.html
6. Typed mid rule actions - https://www.gnu.org/software/bison/manual/html_node/Typed-Mid_002dRule-Actions.html#Typed-Mid_002dRule-Actions
7. Reduce/reduce conflicts - https://www.gnu.org/software/bison/manual/html_node/Reduce_002fReduce.html
8. View statistics of grammar - http://smlweb.cpsc.ucalgary.ca/start.html
9. Code optimizations in compiler - https://www.seas.harvard.edu/courses/cs153/2019fa/lectures/Lec19-Optimization.pdf
10. Optimizations in C++ compilers - https://queue.acm.org/detail.cfm?id=3372264