Report on

“C++ Mini Compiler for while and for loop constructs”

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Jai Agarwal	**01FB16ECS144**
H. Adarsha Nayak	**01FB16ECS127**
Hrishikesh S.	**01FB16ECS139**

Under the guidance of

Preet Kanwal
Assistant Professor
PES University, Bengaluru

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

INTRODUCTION

We have build a self-compiling mini-compiler for C++.

C++ is a general purpose programming language and widely used now a days for competitive programming. It has imperative, object-oriented and generic programming features.

The two main constructs that we have focused on while building this compiler are ‘while’ and ‘for’ loops. The compiler also identifies arithmetic, boolean and logical operations.

The expected outcome of this project is generate a Symbol Table, an Abstract Syntax Tree and Intermediate Three-Address code along with optimization.

INPUT

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

    int b = 11;
    while(b > 10)
    {
        int y;
        y = y + 1;
    }

    int c = 10;
}
```

OUTPUT

```

main:
int b = 11
 L0:

t0 = b>10
if t0 goto L1
goto G0

L1:
int y

t1 = y + 1
y = t1

goto L0
G0:

 int c = 10

```

ARCHITECTURE.

The mini compiler handles most cases in the C++09 .
Features Of The Lexer.

1. Identifies and removes comments
2. Identifies various operators in the language
3. Checks for validity of the identifiers
4. Identifying numeric constants and std::string type
5. Ignores white-spaces
6. Identifies scope of variables

Syntax is handled by YACC where grammar rules are specified for the entire language.

Semantics are handled using semantic rules for type checking while performing operations, to ensure operations are valid.

LITERATURE SURVEY.

1. Lex & Yacc, O'Reilly, John R. Levine, Tony Mason, Doug Brown

CONTEXT FREE GRAMMAR.

```
P
            : program
            ;

program
            : external_declaration
            | program external_declaration
            ;

external_declaration
            : header_stmt
            | global_stmt
            | main_fun
            ;

header_stmt
            : '#' T_INCLUDE '<' T_IDH '>'
            | '#' T_INCLUDE  T_IDH
            ;

global_stmt
            : declaration_statement
            | fun_declr
            | user_defined_ds
            | statement
            ;

main_fun
            : T_TYPE T_MAIN '(' ')' compound_stmt
          ;


declaration_statement
            : T_TYPE list_identifier ';'
            ;

list_identifier
            : list_identifier ',' variable
```

```
            | list_identifier ',' init
            | init
            | variable
            ;

variable
            : T_IDENTIFIER
            | T_IDENTIFIER array
            ;

array
            : array '[' T_INTEGER_VAL ']'
            | '[' T_INTEGER_VAL ']'
            ;

init
            : var_init
      | array_init
            ;

var_init
            : T_IDENTIFIER '=' expression
            ;

array_init
            : T_IDENTIFIER array '=' '{' values '}'
            ;

values
            : values ',' constant
          | constant
            ;

constant
            : T_INTEGER_VAL
            | T_STRING_VAL
            | T_CHAR_VAL
            | T_FLOAT_VAL
            ;

declarator
            : '(' ')'
            | '(' params ')'
            ;

fun_declr
            : T_TYPE T_IDENTIFIER declarator compound_stmt
            ;

params_list
            : T_TYPE T_IDENTIFIER
            | params_list ',' T_TYPE T_IDENTIFIER
```

```
            ;

params
            : params_list
            ;

compound_stmt
            : start_paren end_paren
            | start_paren block_item_list end_paren
            ;

start_paren
            : '{'
            ;

end_paren
            : '}'
            ;
block_item_list
            : block_item
            | block_item_list block_item
            ;

block_item
            : declaration_statement
            | statement
            ;

user_defined_ds
            : class
            | structure
            ;

class
            : T_CLASS T_IDENTIFIER class_body_stmt ';'
            ;

structure
            : T_STRUCT T_IDENTIFIER struct_body_stmt ';'
            ;

class_body_stmt
            : start_paren end_paren
            | start_paren class_mems end_paren
            ;

struct_body_stmt
            : start_paren end_paren
            | start_paren struct_mems end_paren
            ;

struct_mems
```

```
            : struct_mem
            | struct_mems struct_mem
            ;

class_mems
            : class_mem
            | class_mems class_mem
            ;

struct_mem
            : declaration_statement
            | fun_declr
            ;

class_mem
            :  T_TYPE class_var_declaration ';'
            |  fun_declr
            |  access_specifier ':'
            ;

class_var_declaration
            : T_IDENTIFIER
            | class_var_declaration ',' T_IDENTIFIER
            ;

access_specifier
            : T_PRIVATE
        | T_PUBLIC
            | T_PROTECTED
            ;

statement
            : expression_stmt
            | compound_stmt
            | iterative_statement
            | selection_statement
            | input_output_statements
        | return_stmt
            ;

expression_stmt
            : ';'
            | expression ';'
            ;
selection_statement
            : T_IF '(' expression ')' statement
            | T_IF '(' expression ')' statement T_ELSE else_stmt statement
            ;

iterative_statement
            : for_loop
            | while_loop
```

```
            ;

for_loop
            : T_FOR '(' for_assgn_stmt ';' {} expression ';' unary_exprn ')'
statement
            ;

for_assgn_stmt
            : T_TYPE for_decl_stmt
            | assignment_expression
            ;

for_decl_stmt
            : T_IDENTIFIER '=' expression
            | for_decl_stmt ',' T_IDENTIFIER '=' expression
            ;

while_loop
            : T_WHILE '(' expression ')' statement
            ;

return_stmt
            : T_RETURN ';'
            | T_RETURN expression ';'
            ;

expression
            : assignment_expression
            | simple_expression
            ;

assignment_expression
            : T_IDENTIFIER '=' expression

            | unary_exprn
            ;

unary_exprn
            : T_ADDADD T_IDENTIFIER
            | T_MINMIN T_IDENTIFIER
            | postfix_expression
      | T_IDENTIFIER uop_shorthd expression
            ;

postfix_expression
            : T_IDENTIFIER T_ADDADD
            | T_IDENTIFIER T_MINMIN
            ;

uop_shorthd
            : T_ADDEQ
            | T_MINEQ
```

```
            | T_MULEQ
            | T_DIVEQ
            ;

simple_expression
            : additive_expression
            | additive_expression relop additive_expression
            | additive_expression logop additive_expression
            | additive_expression bitop additive_expression
            ;

bitop
            : T_OR
            | T_AND
            | T_XOR
            | T_LRSHIFT
            | T_LLSHIFT
            ;

relop
            : '<'
            | '>'
            | T_LTEQ
            | T_GTEQ
            | T_NEQEQ
            | T_EQEQ
            ;

logop
            : T_OROR
            | T_ANDAND
            ;

additive_expression
            : term
            | additive_expression '+' term
            | additive_expression '-' term
            | '+' additive_expression %prec '*'
            | '-' additive_expression %prec '*'
            ;

term
            : factor
            | term '*' factor
            | term '/' factor
            ;

factor
            : '(' expression ')'
            | T_IDENTIFIER
            | call
            | T_INTEGER_VAL
```

```
            | T_FLOAT_VAL
            | T_STRING_VAL
            | T_CHAR_VAL
            ;

call
            : T_IDENTIFIER '(' ')'
            | T_IDENTIFIER '(' args ')'
            ;

args   : expression
            | expression ',' args
            ;

input_output_statements
            : T_COUT T_LLSHIFT expression ';'
            | T_CIN T_LRSHIFT T_IDENTIFIER ';'
            ;
```

DESIGN STRATEGIES.

SYMBOL TABLE.

The Symbol Table is used for storing variables declared and their attributes, along with details about function calls. The Symbol table stores the size of a variable, it's scope and also the line numbers where the particular variable is used.

We used hash tables to implement the symbol table. The variable names were hashed and allocated a fixed amount of space in the table. Every new scope has it's own symbol table, which would be destroyed once the scope would end. All symbol tables are connected using an n-ary tree. So basically, we have implemented an n-ary tree of hash tables. Starting with the root node which has the global symbol table. Upon encountering a new scope, a new hash table is created as a child node of the root node. If within the same scope another new scope is encountered a new child node of the current scope node is created, and so on. Sibling nodes (i.e child nodes on same level) have different hash tables and hence cannot access each other's data. But a child node can access data from it's parent hash table.

ABSTRACT SYNTAX TREE.

The AST (Abstract Syntax Tree) is an attributed implementation of a generic tree, which can accomodate any number of leafs and nodes and is not limited in any way. Each node uses the same structure type and this allowed easy scaling of the project.

The leafs/nodes/sub-tree is stored in a double-ended queue, also called a dequeue. The dequeue is available from STL of C++.

Flags for detecting if-else, for, while and declaration statements which will be used to decide how we will push and pop the nodes/leafs in the dequeue. For each construct we have a different function, as the method of popping and push nodes/leafs is different is various constructs. The actions are defined as semantic rules in the yacc file. The tree is printed in a branch-wise level-order fashion.

INTERMEDIATE CODE GENERATION.

To convert our given C language into intermediate code, we have used the SDT scheme and made use of marker non-terminals in place of action records.
Label generation:

We have made use of stack whose elements are of type records.
Each entry to the stack is a hash table for which we have used maps.
Each of marker non-terminal will push a map record to the stack which contains
Information of the label's used by the statement.

temp_func()

It returns the next unused temporary variable

label_func() and global_func()

Returns the next unused label

CODE OPTIMIZATION.

We have implemented Dead Code elimination for the ICG generated. Dead codes are pieces of code that contain temporaries that are not used further or anywhere else in the generated ICG. We keep track of all the useful temporaries and hence when we encounter a line which used a non-useful temporary, we can eliminate that line of code from the ICG.

ERROR HANDLING.

We are handling syntax errors, which are generated during parsing. We are also handling re-declaration of variables in the same scope, and are showing appropriate error messages. We stop parsing the input on encountering these errors and display the line number of the errors, intending the user to resolve the issue. Also handling type mismatch errors.

IMPLEMENTATION DETAILS.

SYMBOL TABLE.

Data Structure Implementation

Structure to point to reference of each variable occurrence in program

```
typedef struct RefList{
    int lineno;
    struct RefList *next;
}RefList;
```

A single entry into the hash table :

```
class entry{
      public:
            char st_name[MAXTOKENLEN];
            unsigned int st_size;//Size of the token name
            unsigned int scope_id;//Scope resolution of the variable
            RefList *lines;//Refer to multiple definitions of same token

            // to store value and sometimes more information
            int st_ival; double st_fval; char *st_sval;

            unsigned int type; //declaration type, for variables
            unsigned int ret_type; //For return types of functions
            unsigned int size; //Storage required
};
```

A Node of the n-ary tree of symbol tables

```
struct Node{
      std::unordered_map<std::string, entry> symbolTable;
      std::vector< Node*> child;
      Node* parent;
};
```

Functions Implemented.

```
void insert(char *name, int len, int type, int lineno);
// Add a record into the hash table

void create_new_scope();
//Creates a child node on entering new_scope

void exit_scope();
//Exit the scope, and hence move back to parent scope/parent symbol table

void LevelOrderTraversal(Node * root);
//Traverse the n-ary tree while displaying the Symbol Table

void print_symbolTable(std::unordered_map<std::string, entry> symbolTable);
//Print the symbol table
```

ABSTRACT SYNTAX TREE.

Data Structure Implementation

Structure for each node/leaf in the AST.

```
struct ast_node{
      int node_type;     // Unique number of the node
      std::string symbol;      // Symbol of the node
      std::vector<ast_node *> children; // Children of the node
};
```

Syntax Tree.

```
struct ast_body{
      const char *head_symbol = "<body>";  // root of tree
      std::vector<ast_node*> children;     // children of root
};
```

Double-Ended queue

```
// stack to maintain leafs and nodes for AST
std::deque<ast_node*> S_ast;
```

Functions Implemented.

```
// Prints the branch in pre-order like fashion
void LevelOrderTraversalEACHROOT(ast_node *root);

// create a branches on node
ast_node *create_branch(ast_node *i_root, ast_node *branch);

// central node creation of exp evaluation
ast_node *central_node_creation(std::string op);

// alternate central node creation for exp evaluation
ast_node *central_node_creation_exp(std::string op);

// declaration and expression
ast_node *central_node_creation_declaration(std::string op);

// branch-wise level order traversal of tree
void LevelOrderTraversalAST();

// print all dequeue elements
void print_stack_elements();

// declare and assigned branch
void declare_and_assign_branch();

// declare and assign node
void declare_assign_node_creation();
```

```
// popping pattern in loops
void general_declaration_in_loop();

// if branch execution
void IF_Alternate(int if_cond_flag);

// for branch execution
void for_creation(int for_flag);

// for unary expression (postfix & prefix)
void unary_expression_branch();

// while branch execution
void while_creation(int while_flag);

// else branch execution;
void else_creation();
```

INTERMEDIATE CODE GENERATION.

Data Structure Implementation

Following is the type of non-terminals used in our grammar.

```
Struct{
      char* next;
            char code[1500];
            char addr[50];
            char* true_label;
            char* false_label;
}node;
```

Functions Implemented.

If marker non-terminal action record:

```
map <string,string> m;
char *begin = label_func();
m["true"] = string(begin);
free(begin);
m["false"] = string(global_func());
m["next"] = m["false"];
v.push_back(m);
```

Here we can see that for the hash table key is the attribute name and its value is the label the attribute is going to hold. In this case, we push this record before going to the condition non-terminal.

Our stack is implemented using vector which can be seen as a generic list. The element type of the vector used is map.

CODE OPTIMIZATION.

Data Structure Implementation

Used a vector of strings to store the input lines of ICG code one by one.

```
std::vector<std::string> list_of_lines;
```

List of strings after eliminating dead code is stored in :

```
std::vector<std::string> without_deadcode;
```

Used the following two regex to detect temporary variable or identifier

```
std::regex istemp("^t[0-9]*$");
std::regex isid("^[A-Za-z][A-Za-z0-9_]*$");
```

List to store the valid operators that can be used in the ICG.

```
char arr[][3] = {"+", "-", "*", "/", "*", "&", "|", "^", ">>", "<<", "==",
">=", "<=", "!=", ">", "<"};
std::vector<std::string> binary_operators(arr,arr+16);
```

Functions Implemented.

```
void printicg( std::vector<std::string>list_of_lines, std::string
message="");
// Used to print the ICG after optimisation

std::vector<std::string>remove_dead_code(std::vector<std::string>
list_of_lines);
//Used to remove dead_code from input ICG.
```

ERROR HANDLING.

Function to handle syntax errors, gets called automatically by yacc on encountering syntax errors

```
void yyerror(const char *str)
{
      printf("line no :%d  %s near %s\n", yylineno, str, yytext );
}
```

For re-declaration check we used a flag variable declare=0, which would be set and unset accordingly in the same scope. If it was previously set and is set again in the same scope, we can catch the error and display appropriate error message.

RESULTS.

We were able to successfully generate the different representations of input code written in C++ along with performing simple machine independent optimization on intermediate code which was generated.

Shortcomings.

1. Error handling could have been done better by adding more rules.
2. More optimizations could have been implemented.
3. C++14 and namespaces not taken into consideration.

SNAPSHOTS.

INPUT.

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

    int b = 11;
    while(b > 10)
    {
        int y;
        y = y + 1;
    }

    int c = 10;
}
```

SYMBOL TABLE.

```
Identifier Name c
Scope Id    1
Type        1
Value       10
Line Numbers
12
Size        0
--------------------
Identifier Name b
Scope Id    1
Type        1
Value       11
Line Numbers
5
Size        0
--------------------

Identifier Name y
Scope Id    2
Type        1
Value       6419713
Line Numbers
8
Size        0
--------------------
```

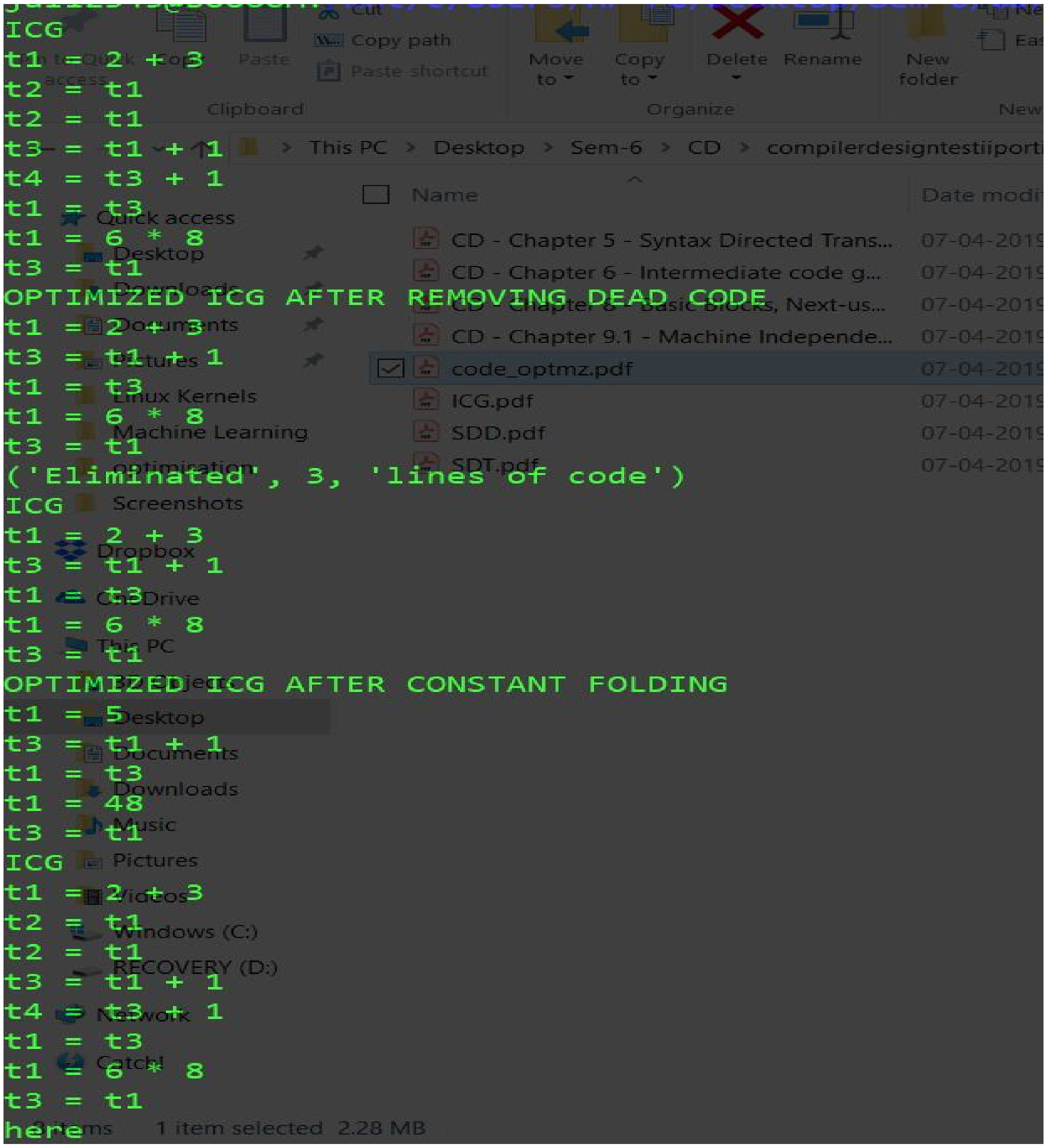
ABSTRACT SYNTAX TREE.

```
-----------------
SYNTAX TREE (BRANCH-WISE - LevelOrder)
<body>
||
||
||
||
DECLR_STAT
int
b
11

||
||
||
||
WHILE_STRUCT
while
COND STATEMENT
>       DECLR_STAT       =
10
b
int
y
y
+
y
1

||
||
||
||
DECLR_STAT
int
c
10
```

INTERMEDIATE CODE GENERATION.

```
main:
int b = 11
 L0:

t0 = b>10
if t0 goto L1
goto G0

L1:
int y

t1 = y + 1
y = t1

goto L0
G0:

 int c = 10
```

CODE OPTIMIZATION.



```
ICG
t1 = 2 + 3
t2 = t1
t2 = t1
t3 = t1 + 1
t4 = t3 + 1
t1 = t3
t1 = 6 * 8
t3 = t1
OPTIMIZED ICG AFTER REMOVING DEAD CODE
t1 = 2 + 3
t3 = t1 + 1
t1 = t3
t1 = 6 * 8
t3 = t1
('Eliminated', 3, 'lines of code')
ICG
t1 = 2 + 3
t3 = t1 + 1
t1 = t3
t1 = 6 * 8
t3 = t1
OPTIMIZED ICG AFTER CONSTANT FOLDING
t1 = 5
t3 = t1 + 1
t1 = t3
t1 = 48
t3 = t1
ICG
t1 = 2 + 3
t2 = t1
t2 = t1
t3 = t1 + 1
t4 = t3 + 1
t1 = t3
t1 = 6 * 8
t3 = t1
here
```

CONCLUSIONS.

By doing this project, we have gained a better insight into the phases of compiler. YACC provided us with a better knowledge about bottom-up parser and while performing the different phases of the compiler, our code efficiency in writing code and dealing with complex data structures has significantly improved.

REFERENCES/BIBLIOGRAPHY.

Lex & Yacc, O'Reilly, John R Levine, Tony Mason, Doug Brown

Our code can be found at https://github.com/redlegblackarm/CppCompiler .