Report on

"C Mini Compiler for 'for' and 'if-else' constructs"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Swathi	**PES1201701826**
Aishwarya M M	**PES1201802368**
Adarsh	**PES1201701399**
Yadhunand Segar	**PES1201701404**

Under the guidance of

Prakash C O
Assistant Professor
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

CHAPTER 1. INTRODUCTION

This mini project is for building the compiler for the C programming language. We have implemented the front end of the compiler for C language using Lex and Yacc for the following constructs :

1. For Loop
2. If – Else Statement

Using GNU C Compiler i.e gcc's latest version as our reference, we developed a mini-compiler that can handle basic C operations along with the ones chosen. Given source program in C can be translated to a symbol table, abstract syntax tree, intermediate code, optimized intermediate code and target code.

Sample input:

```
#include<stdio.h>
#include<stdlib.h>
float g;
int main(){
      int b = 10;
      float c = 5.2;
      char d;
      int 2b=5;
      /*
            Random
      */
      for(int i = 0; i < 100; i++)
      {
            int h;
            {
                  int y;
                  b++;
            }
            //increment b
      }
      if (b>5)
            printf("hello");
      int e = (b > a) ?  10: b
}
```

CHAPTER 2. ARCHITECTURE OF LANGUAGE

- Used Flex/lex to create the scanner for our language.
- Used Bison/yacc to implement grammar rules to the token generated in the scanner phase.
- All token names are in capitals and everything else is in caps.
- The following are the operators and special characters implemented in our programming language:
    - Binary operators: + - * / %
    - Unary operators: ! ~ + - ++ -- (postfix

    and prefix)
- Ignore comments and white-spaces
    - Single line comments starting with //
    - Multi-line comments enclosed within /* .... */
- Types: int, float, char, void
- Constructs ‘for’ loop and ‘if-else’ condition statement.
- Includes functions definitions and function calls.
- No conflicts and errors in our code/grammar.
- Maximum identifier length = 15
- Warnings and Error recovery:
    - Type Checking :
        - implicit conversion from float to char
        - implicit conversion from float to int
        - implicit conversion from char to float
    - Missing type specifier in the function definition. Defaults type to INT.
    - Division by zero. Assign INT_MAX and continue.
- Errors:
    - Redefinition of identifiers within the same scope
    - Use of undeclared identifiers
    - Invalid operands to ‘%’. Implicit conversion to INT
    - Truncate lengthy identifiers to length = 15
    - Syntax errors based on the specified grammar.
- All the errors and warnings are displayed along with line number
- If the same variable name is used within a nested scope, the *most closely nested loop* rule is used instead of giving an error (undeclared variable). It uses the previously defined value in the higher scope.
- Error handling related to scope and declaration.
- Since C doesn’t have a bool type, 0 and 1 are used to indicate false and true respectively. The test expression used in the ternary statement evaluates to 0 or 1 accordingly.
- Code Optimizations techniques used :
    - Loop unrolling
    - Constant folding

CHAPTER 3. LITERATURE SURVEY

https://www.lysator.liu.se/c/ANSI-C-grammar-y.html - Helped us write the grammar for our compiler

https://softwareengineering.stackexchange.com/questions/165543/how-to-write-a-very-basic-compiler - A reference link for writing the code and taking ideas.

https://www.sigbus.info/how-i-wrote-a-self-hosting-c-compiler-in-40-days.html - A detailed guide link for writing the code and taking ideas.

```
symbol:
        ext_dec
      | symbol ext_dec
      ;

ext_dec
      : INT MAIN '(' ')' compound_statement
      | VOID MAIN '(' ')' compound_statement
      | declaration
      | headers
      ;

headers
      : HASH INCLUDE HEADER_LITERAL
      | HASH INCLUDE '<' libraries '>'
      ;

libraries
      : STDIO
      | STDLIB
      | MATH
      | STRING
      | TIME
      ;

compound_statement
      : '{' '}'
      | '{' block_item_list '}'
      ;

block_item_list
      : block_item
      | block_item_list block_item
      ;

block_item
      : declaration
      | statement {
                        struct node *ftp;
                        ftp = first;
                        while(ftp!=NULL){
                              if(ftp->scope == scope && ftp->valid == 1){
                                    ftp = ftp->link;
                              }
                              ftp=ftp->link;
                        }
                        scope--;
                  }
      ;

declaration
      : type_specifier init_declarator_list ';'
      ;

statement
      : compound_statement
      | expression_statement
      | iteration_statement
      ;

type_specifier
      : VOID
      | CHAR
      | INT
      | FLOAT
      ;

init_declarator_list
      : init_declarator
      | init_declarator_list ',' init_declarator
      ;

init_declarator
      : IDENTIFIER '=' assignment_expression
      | IDENTIFIER
      ;

assignment_expression
      : conditional_expression
      | unary_expression assignment_operator assignment_expression
      ;

conditional_expression
      : equality_expression
      | equality_expression '?' expression ':' conditional_expression
      ;

assignment_operator
      : '='
      | ADD_ASSIGN
      | SUB_ASSIGN
      ;

expression_statement
      : ';'
      | expression ';'
      ;

expression
      : assignment_expression
      | expression ',' assignment_expression
      ;

iteration_statement
      : FOR '(' expression_statement expression_statement ')' statement
      | FOR '(' expression_statement expression_statement expression ')' statement
      | FOR '(' declaration expression_statement ')' statement
      | FOR '(' declaration expression_statement expression ')' statement
      ;

primary_expression
      : IDENTIFIER
      | INTEGER_LITERAL
      | STRING_LITERAL
      | FLOAT_LITERAL
      | CHARACTER_LITERAL
      | '(' expression ')'
      ;

postfix_expression
```

```
        : primary_expression
        | postfix_expression '(' ')'
        | postfix_expression '.' IDENTIFIER
        | postfix_expression INC_OP
        | postfix_expression DEC_OP
        ;

unary_expression
        : postfix_expression
        | unary_operator unary_expression
        ;

unary_operator
        : '+'
        | '-'
        | '&'
        | '|'
        | '!'
        ;

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

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

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

equality_expression
        : relational_expression
        | equality_expression EQ_OP relational_expression
        | equality_expression NE_OP relational_expression
        ;
```

CHAPTER 5. DESIGN STRATERGY

Design

Language: C
Tools: Lex and Yacc
Data Types: int, float, char, void
Constructs: for and if-else

- **SYMBOL TABLE CREATION:** The symbol table contains information about all the variables declared. Scoping is also implemented with the use of arrays. It contains: the variable name, the variable type, the size of variable, and the scope variables. Each time a variable is declared, first it is checked whether that variable already exists in the symbol table in that scope, and then it is entered into the symbol table if it doesn't already exist. Otherwise, it gives a re-declaration error. Every time a variable is assigned, it checks the symbol table to see if it is a variable that has previously been declared. Otherwise, it throws an undeclared variable error.

```
Symbol(identifier, param, function), Name, Type,
Scope, Line , Value)
```

- **ABSTRACT SYNTAX TREE:** A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree. The parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase. Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

- **INTERMEDIATE CODE GENERATION:** The intermediate code is generated on the fly, as we parse the code and check its grammar, the intermediate code is generated. The representation of ICG is done by using quadruple table. The quadruple table consists of operator, first parameter, second parameter and the result.

- **CODE OPTIMIZATION:** To increase efficiency the code optimization is done on the generated ICG. We have implemented loop unrolling and constant folding.

- **ERROR HANDLING:** In case of syntax error, the compilation is halted, and an error message along with the line number where error occurred is displayed. Semantic errors such as division by zero, multiple declaration of the same variable, invalid assignment, scope errors are also explicitly pointed out. All of which are specified as production rules within the grammar. Type checking is done and if the types do not match in the expressions, a warning is displayed and implicit conversion is done to recover from errors.

- **TARGET CODE GENERATION:** Each instruction takes its operands from the top of the stack, removes those operands from the stack, computes the required operation on them, and pushes the result on to the stack.

CHAPTER 6. IMPLEMENTATION DETAILS

- The tools we have used for implementing the code are lex and yacc.
- The lex file has all the tokens specified with the help of regular expressions and the yacc file has grammar rules with corresponding actions.
- As the code is being parsed, the tokens are generated and comments and extra

spaces are ignored. For every new variable encountered, it is entered into the symbol table along with its attributes.

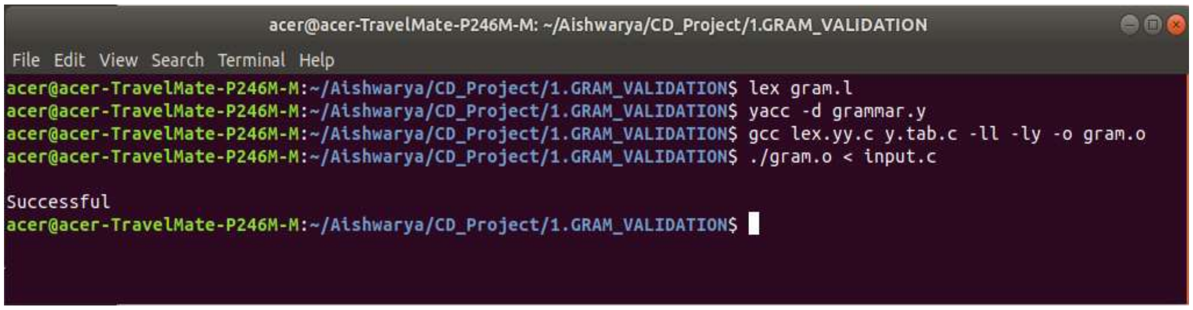
- A union is used which consists of int, float and char and void type to use it for assigning data type of each variable.
- Semantics Analysis uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct (type checking) and update it accordingly.
- The scope check is done by having a variable which increments on every level of nesting. In this manner, the scope is checked for each variable and error messages are displayed if anything is used out of scope.
- Integer, float and char literal constants.
- We have written appropriate rules to check for semantic validity (type checking, declare before use, etc.)
- Type checking is done by checking type of each variable in the expression and implicit conversions are done to avoid errors along with display of warnings.
- Variables must be declared and can only be used in ways that are acceptable for the declared type.
- Once parsing is successful, we generate an abstract tree and it is shown in pre-order manner.
- The intermediate code generation also happens on the fly.
- After generating intermediate code, optimization is done by doing by
    - Constant folding: This kind of optimization technique evaluates the right hand side of expressions and stores the evaluated value, instead of storing the expression itself. It is implemented in the expression grammar. If the RHS of an expression contains only numbers, then they are converted to numbers, evaluated, and copied to $$.addr.
    - Loop unrolling: Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program by removing or reducing iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions. Here, in the for loops, the condition variable and initializations of the loop variables are used to calculate the number of iterations, and the code inside the loop is unrolled.
- Code optimization uses information present in symbol table for machine dependent optimization.
- Target code generation uses a simple compiler, where each operation takes operands from the same place and puts result in the same place.

CHAPTER 7. RESULT

Initially we parse the input. After parsing, if there are errors then the line numbers of those errors are displayed along with a 'parsing failed' on the terminal. Otherwise, a 'parsing successful' message is displayed on the console. The symbol table with stored and updated values is always displayed, irrespective of errors. Also, the three address codes along with the temporary variables are also displayed along with the flow of the conditional and iterative statements. Then we will get a optimized code and a target code.

CHAPTER 8. SNAPSHOTS

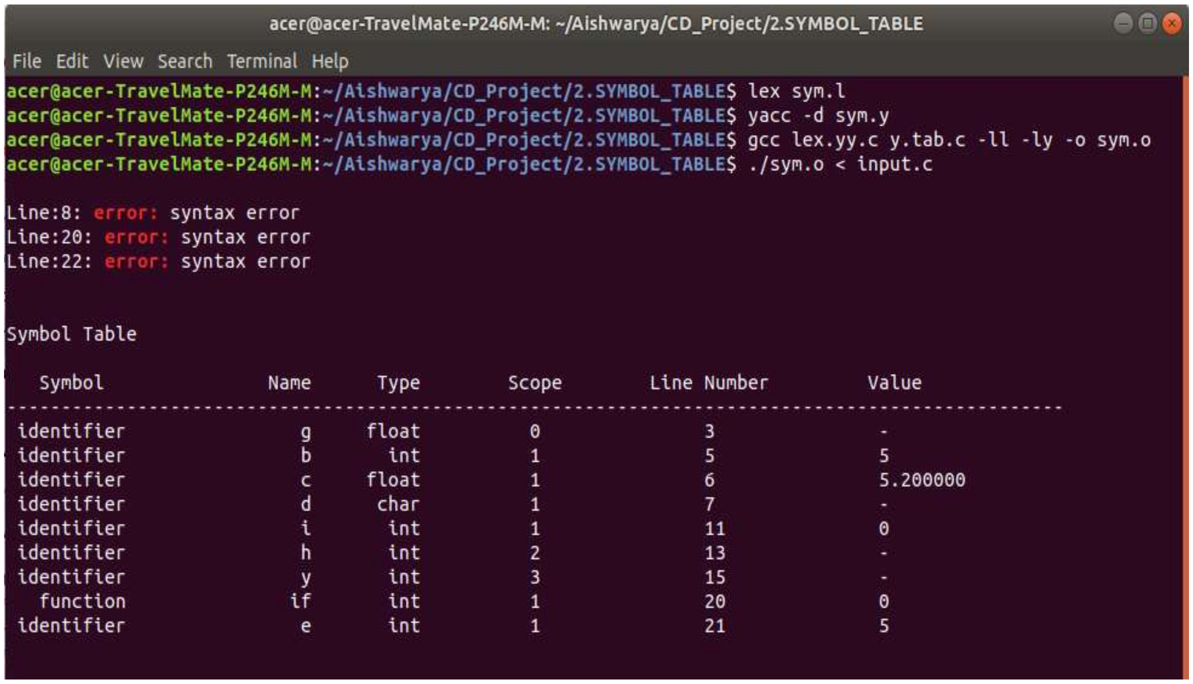
Parsing of the input:



acer@acer-TravelMate-P246M-M: ~/Aishwarya/CD_Project/1.GRAM_VALIDATION
File Edit View Search Terminal Help

```
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/1.GRAM_VALIDATION$ lex gram.l
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/1.GRAM_VALIDATION$ yacc -d grammar.y
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/1.GRAM_VALIDATION$ gcc lex.yy.c y.tab.c -ll -ly -o gram.o
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/1.GRAM_VALIDATION$ ./gram.o < input.c

Successful
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/1.GRAM_VALIDATION$
```

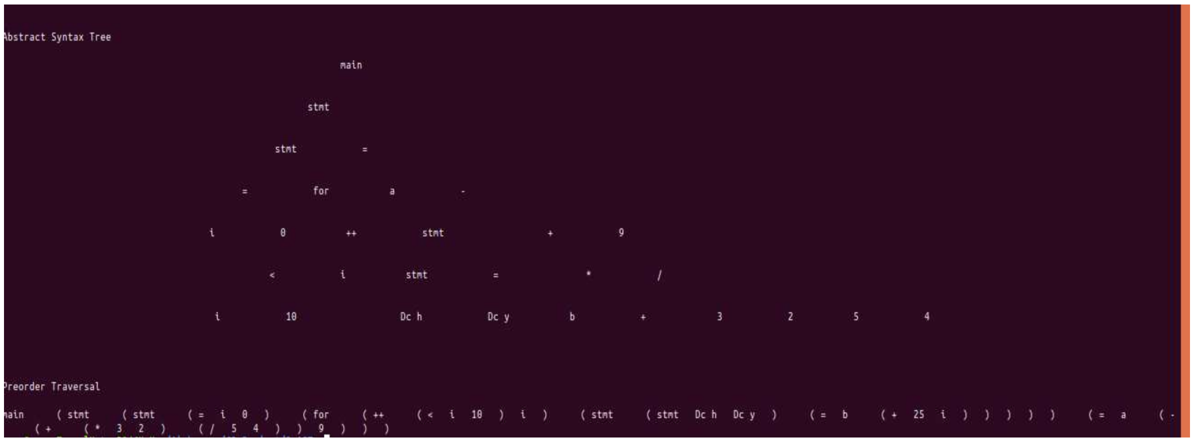
Symbol table with error handling:



acer@acer-TravelMate-P246M-M: ~/Aishwarya/CD_Project/2.SYMBOL_TABLE
File Edit View Search Terminal Help

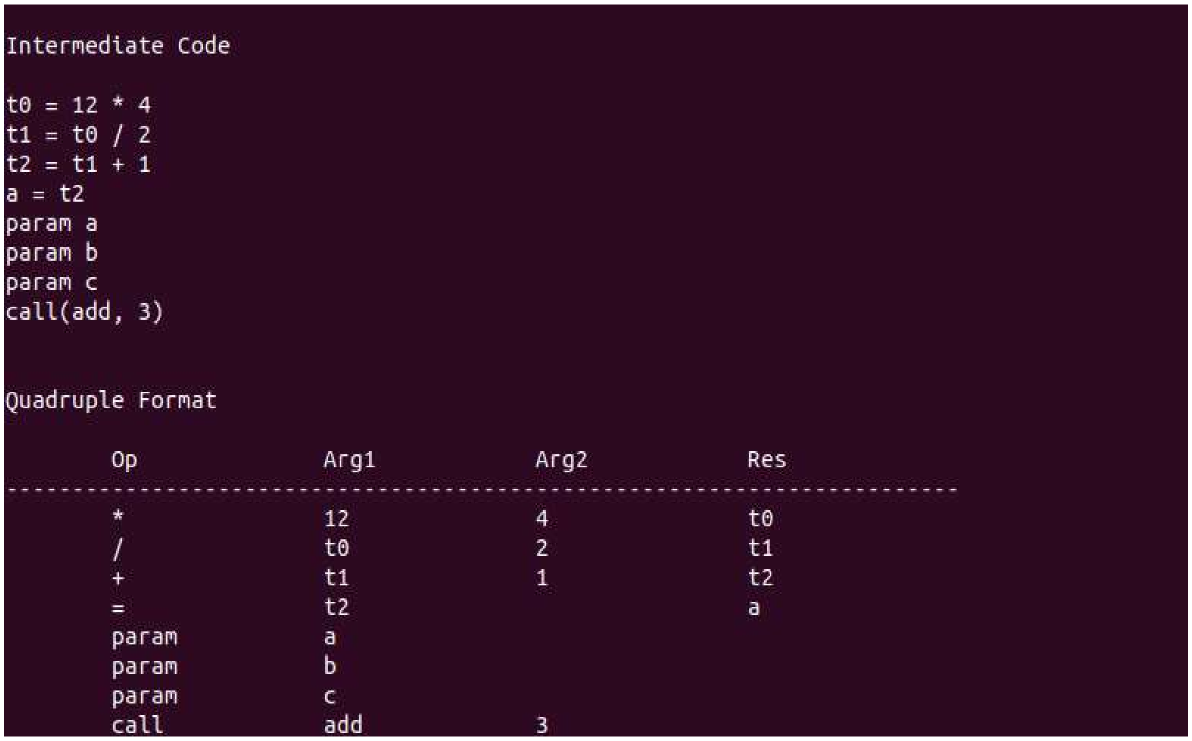
```
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/2.SYMBOL_TABLE$ lex sym.l
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/2.SYMBOL_TABLE$ yacc -d sym.y
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/2.SYMBOL_TABLE$ gcc lex.yy.c y.tab.c -ll -ly -o sym.o
acer@acer-TravelMate-P246M-M:~/Aishwarya/CD_Project/2.SYMBOL_TABLE$ ./sym.o < input.c

Line:8: error: syntax error
Line:20: error: syntax error
Line:22: error: syntax error


Symbol Table

   Symbol              Name      Type        Scope        Line Number         Value
--------------------------------------------------------------------------------------------------
 identifier               g     float          0              3              -
 identifier               b       int          1              5              5
 identifier               c     float          1              6              5.200000
 identifier               d      char          1              7              -
 identifier               i       int          1             11              0
 identifier               h       int          2             13              -
 identifier               y       int          3             15              -
   function              if       int          1             20              0
 identifier               e       int          1             21              5
```

Abstract syntax tree:



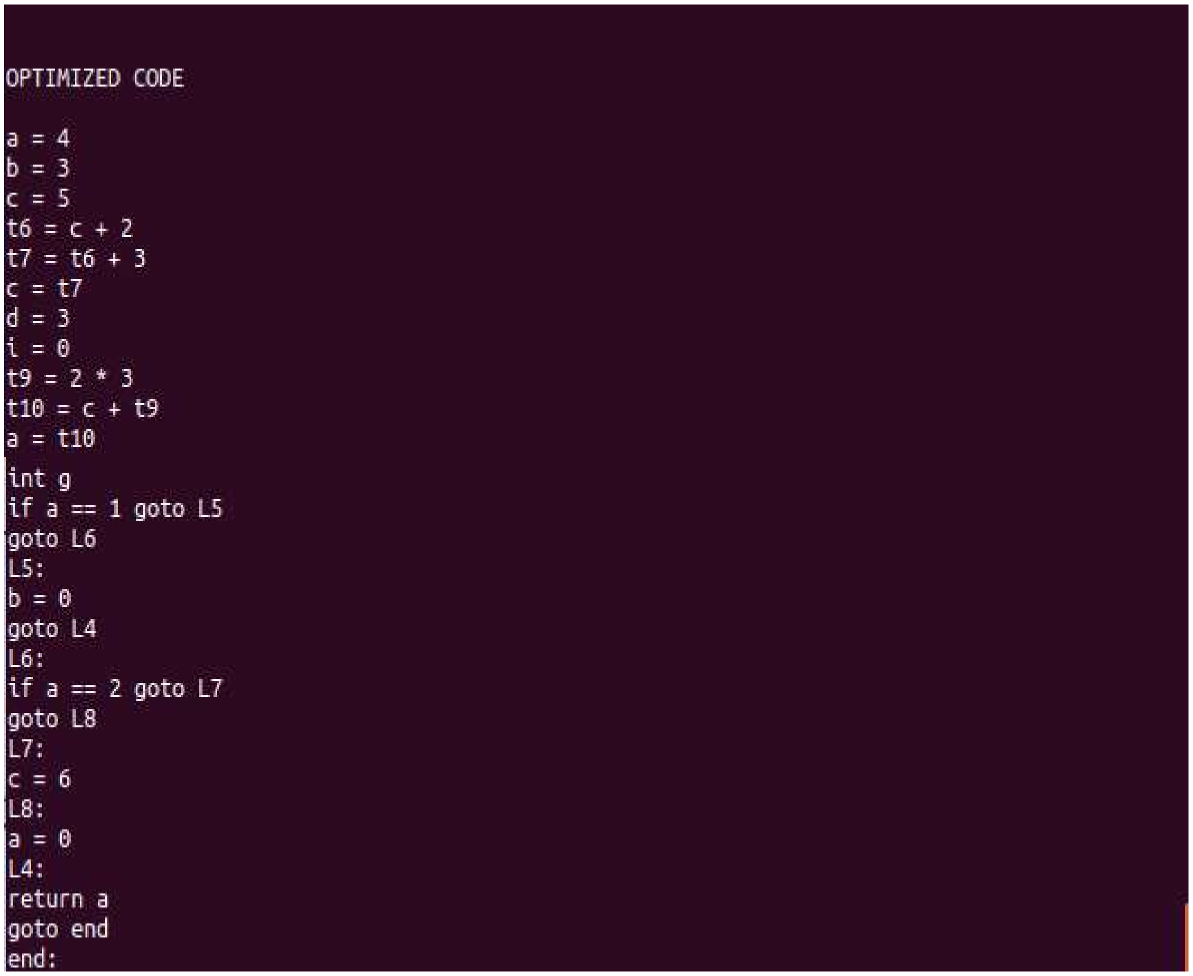
```
bstract Syntax Tree
                                                             main

                                                       stmt

                                                 stmt            =

                                           =           for           a            -

                                      i           0           ++            stmt                   +            9

                                                 <            i          stmt            =                *            /

                                       i            10                  Dc h           Dc y          b            +            3            2          5            4


reorder Traversal

ain     ( stmt     ( stmt     ( =  i  0  )     ( for     ( ++     ( <  i  10  )  i  )     ( stmt     ( stmt  Dc h  Dc y  )     ( =  b     ( +  25  i  )  )  )  )  )     ( =  a     ( -
     ( +     ( *  3  2  )     ( /  5  4  )  )  9  )  )  )
```

Intermediate code generation:



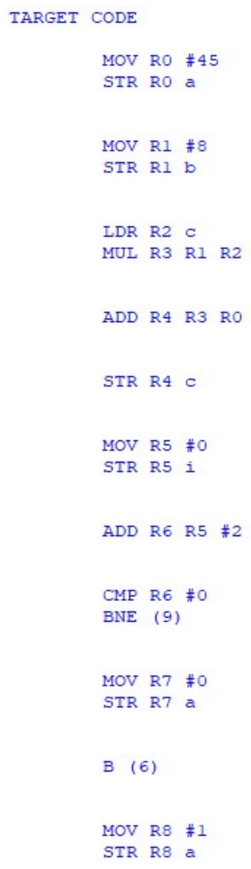
```
Intermediate Code

t0 = 12 * 4
t1 = t0 / 2
t2 = t1 + 1
a = t2
param a
param b
param c
call(add, 3)


Quadruple Format

        Op              Arg1            Arg2            Res
------------------------------------------------------------------------
        *               12              4               t0
        /               t0              2               t1
        +               t1              1               t2
        =               t2                              a
        param           a
        param           b
        param           c
        call            add             3
```

Optimized code generation:



```
OPTIMIZED CODE

a = 4
b = 3
c = 5
t6 = c + 2
t7 = t6 + 3
c = t7
d = 3
i = 0
t9 = 2 * 3
t10 = c + t9
a = t10
int g
if a == 1 goto L5
goto L6
L5:
b = 0
goto L4
L6:
if a == 2 goto L7
goto L8
L7:
c = 6
L8:
a = 0
L4:
return a
goto end
end:
```

Target code generation:



```
TARGET CODE

        MOV R0 #45
        STR R0 a


        MOV R1 #8
        STR R1 b


        LDR R2 c
        MUL R3 R1 R2


        ADD R4 R3 R0


        STR R4 c


        MOV R5 #0
        STR R5 i


        ADD R6 R5 #2


        CMP R6 #0
        BNE (9)


        MOV R7 #0
        STR R7 a


        B (6)


        MOV R8 #1
        STR R8 a
```

CHAPTER 9. CONCLUSIONS

A complier for C was thus created using lex and yacc. In addition to the for and if-else statement, error handling was implemented. This project is done by keeping the various stages of Compiler Design, i.e., lexical analysis, syntax analysis, semantic analysis, intermediate code generation and code optimization in mind.

As a part of each stage, an auxiliary part of the compiler was built(symbol table, abstract syntax tree, intermediate code generation, code optimization and target code generation).

CHAPTER 10. FUTURE ENHANCEMENTS

Include:

- Other looping constructs like while, do-while.
- Conditional jumps like continue and break.
- Conditional statements like switch case.
- More data types.