***Report on***

# **"Mini Compiler for C++"**

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

## ***Compiler Design Laboratory***

## **Bachelor of Technology**

## **in**

## **Computer Science & Engineering**

***Submitted by:***


| | |
|---|---|
| **Kushal K** | **PES1201801481** |
| **Abhishek R** | **PES1201801682** |
| **Vybhav Bhadri S** | **PES1201801764** |

*Under the guidance of*

**Mahesh H. B.**

Assistant 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

# 1 INTRODUCTION

The objective of this project is to build a Mini C++ compiler using lexx and yacc or ply module for python. This project includes the 3 main files for lexing, parsing and optimization which is implemented in python using ply module. The output of the lexer is tokens and symbol table, tokens are input to the parser and output of parser is three-address code, AST with scope table and updated symbol table. The three-address code is passed to optimizer code that generates the and outputs the optimized three address code. The C++ constructs for which mini- compiler is built for is ‘switch’ and ‘for’. The compiler also identifies arithmetic, Boolean and logical operations and takes care of reserved words. It also includes some the basic functionality required to build an effective compiler.

# 2 ARCHITECTURE

The mini-compiler supports the syntax and semantics of C++ for all the constructs that are supported. It supports all valid C++ programs with arithmetic expressions, declaration statements (including arrays), assignment statements, if-else construct, while construct and switch construct. We have handled arithmetic expressions which involve arithmetic operators, logical operators, relational operators and assignment operators.

We have handled all the possible errors and edge test cases for every construct supported by our compiler. The error recovery method used by our compiler is panic mode recovery.

# 3 LITERATURE SURVEY AND OTHER REFERENCES


1. https://ply.readthedocs.io/en/latest/
2. https://www.dabeaz.com/ply/ply.html#ply_nn4
3. https://www.youtube.com/watch?v=Hh49BXmHxX8
4. https://www.youtube.com/watch?v=orI232lQv6U
5. https://www.youtube.com/watch?v=Zbk0lic04SI

# 4 CONTEXT FREE GRAMMAR

```
Rule 0     S' -> start
Rule 1     start -> INT MAIN LPAREN RPAREN LBRACE statement_list RBRACE
Rule 2     for -
> FOR LPAREN check_for new_scope statement gen_new_label cond SEMI unary RPAREN LBR
ACE gen_new_label statement_list cond_label RBRACE uncheck_for delete_scope
Rule 3     check_for -> empty
Rule 4     uncheck_for -> empty
Rule 5     cond_label -> empty
Rule 6     new_scope -> empty
Rule 7     delete_scope -> empty
Rule 8     new_tab -> empty
Rule 9     gen_new_label -> empty
Rule 10    switch -
> SWITCH LPAREN new_scope switch_expr RPAREN LBRACE labeled_statement_list RBRACE d
elete_scope
Rule 11    switch_expr -> ID
Rule 12    switch_expr -> ICONST
Rule 13    labeled_statement_list -> labeled_statement labeled_statement_list
Rule 14    labeled_statement_list -> empty
Rule 15    labeled_statement -
> CASE gen_new_label const_expr COLON new_scope statement_list delete_scope
Rule 16    labeled_statement -
> DEFAULT COLON gen_new_label new_scope statement_list delete_scope
Rule 17    labeled_statement -> labeled_statement BREAK SEMI
Rule 18    const_expr -> ICONST
Rule 19    const_expr -> CCONST
Rule 20    statement_list -> statement statement_list
Rule 21    statement_list -> empty
Rule 22    statement -> unary
Rule 23    statement -> assign
Rule 24    statement -> declaration
Rule 25    statement -> for
Rule 26    statement -> switch
Rule 27    empty -> <empty>
Rule 28    assign -> ID EQUALS expr SEMI
Rule 29    assign -> ID EQUALS CCONST SEMI
Rule 30    cond -> ID LT ICONST
Rule 31    cond -> ID LE ICONST
Rule 32    cond -> ID GE ICONST
Rule 33    cond -> ID GT ICONST
Rule 34    cond -> ID LT ID
Rule 35    cond -> ID LE ID
Rule 36    cond -> ID GE ID
Rule 37    cond -> ID GT ID
```

```
Rule 38    cond -> ID NE ICONST
Rule 39    cond -> ID NE ID
Rule 40    cond -> ID NE FCONST
Rule 41    cond -> ICONST NE ID
Rule 42    cond -> ICONST NE ICONST
Rule 43    cond -> ID LE FCONST
Rule 44    cond -> ID GE FCONST
Rule 45    cond -> ID GT FCONST
Rule 46    cond -> ID LT FCONST
Rule 47    cond -> ICONST LE ICONST
Rule 48    cond -> ICONST GE ICONST
Rule 49    cond -> ICONST GT ICONST
Rule 50    cond -> ICONST LT ICONST
Rule 51    cond -> FCONST LE FCONST
Rule 52    cond -> FCONST GE FCONST
Rule 53    cond -> FCONST GT FCONST
Rule 54    cond -> FCONST LT FCONST
Rule 55    cond -> ID EQ ID
Rule 56    cond -> ID EQ ICONST
Rule 57    cond -> ID EQ FCONST
Rule 58    cond -> ICONST EQ ID
Rule 59    cond -> FCONST EQ ID
Rule 60    cond -> ICONST EQ ICONST
Rule 61    cond -> FCONST EQ FCONST
Rule 62    cond -> ID
Rule 63    unary -> PLUSPLUS ID
Rule 64    unary -> MINUSMINUS ID
Rule 65    unary -> ID PLUSPLUS
Rule 66    unary -> ID MINUSMINUS
Rule 67    declaration -> types vee SEMI
Rule 68    declaration -> types arr SEMI
Rule 69    types -> INT
Rule 70    types -> FLOAT
Rule 71    types -> DOUBLE
Rule 72    types -> CHAR
Rule 73    types -> LONG
Rule 74    types -> REGISTER
Rule 75    vee -> vee COMMA vee
Rule 76    vee -> ID
Rule 77    vee -> init
Rule 78    init -> ID EQUALS expr
Rule 79    init -> ID EQUALS CCONST
Rule 80    arr -> ID open_bracket
Rule 81    open_bracket -> LBRACKET ICONST RBRACKET
Rule 82    open_bracket -> LBRACKET ICONST RBRACKET open_bracket
Rule 83    expr -> expr PLUS term
```

```
Rule 84    expr -> expr MINUS term
Rule 85    expr -> term
Rule 86    term -> term TIMES factor
Rule 87    term -> term DIVIDE factor
Rule 88    term -> factor
Rule 89    factor -> ID
Rule 90    factor -> ICONST
Rule 91    factor -> FCONST
```

# 5 DESIGN STRATEGY

## 5.1 Symbol Table

We create a new symbol table every time we encounter a new scope or enter into a new scope and delete the table on exiting the scope. For every variable in the scope, we keep track of its name, type, value, address and line number. Name here refers to the name of the variable declared, type refers to the type of the declared variable, address refers to the address of the corresponding variable and line number is used to indicate the line number where the variable is declared.

## 5.2 Intermediate Code Generator

For generating intermediate code, we use another table that keeps track of all the variables declared irrespective of their scope. We keep track of the version number of that corresponding variable which helps in code optimizations. We also use a stack to keep track of version numbers of the variable in each scope. We make use of the tokens generated by the lexer and the symbol table to generate the intermediate code. We make use of the version number to show handle assignment with respect to scope. Intermediate code is generated in quadruple format.

## 5.3 code optimization

Code optimization was done on the output file containing the intermediate three address code. This was read into a Python list and optimization actions were performed. Constant Folding and Propagation, Copy Propagation, Common Subexpression Elimination, and partial Dead Code

Elimination were the optimizations performed. A symbol table was used to assist the optimization process and multiple utility functions were used to ease the process of writing code.

## 5.4 Error handling

We have delegated all the syntactical errors to p_error( ) of ply but we do handle some of the semantic errors through our functions. p_error( ) recovers from the error by panic mode recovery. For semantic errors like using an undeclared variable or redeclaring a variable we display the appropriate message.

# 6 IMPLEMENTATION DETAILS

## 6.1 Symbol Table Creation

We create a new symbol table every time we encounter a new scope or enter into a new scope and delete the table on exiting the scope. For every variable in the scope, we keep track of its name, type, value, address and line number. Name here refers to the name of the variable declared, type refers to the type of the declared variable, address refers to the address of the corresponding variable and line number is used to indicate the line number where the variable is declared.

## 6.2 Intermediate Code Generation

For generating intermediate code, we use another table that keeps track of all the variables declared irrespective of their scope. We keep track of the version number of that corresponding variable which helps in code optimizations. We also use a stack to keep track of version numbers of the variable in each scope. We make use of the tokens generated by the lexer and the symbol table to generate the intermediate code. We make use of the version number to show handle assignment with respect to scope. Intermediate code is generated in quadruple format.

## 6.3 Code Optimization

Code optimization was done on the output file containing the intermediate three address code. This was read into a Python list and optimization actions were performed. Constant Folding and

Propagation, Copy Propagation, Common Subexpression Elimination, and partial Dead Code Elimination were the optimizations performed. A symbol table was used to assist the optimization process and multiple utility functions were used to ease the process of writing code.

## 6.4 Error Handling

For semantic errors like using an undeclared variable or redeclaring a variable we display the appropriate message.

## 6.5 Tools and Instruction to run the code

- Visual Studio Code
- Python 3 and above
- Python module requirements:
  - ply
  - termcolor
  - pandas
  - tabulate

Source for these screenshots (Our GitHub Repo readme.md):
[https://github.com/abhira0/cpp_mini_compiler/blob/main/README.md](https://github.com/abhira0/cpp_mini_compiler/blob/main/README.md)

## General

Open Windows Terminal (preffered) / cmd / Powershell in `backend` direcory

## Token Rules

- Token Rules are coded in `backend\_0_tokrules.py`
- All the simple regex and regex with action code is written in this file
- No token are generate here
- This file is used by `backend\_1_lexer.py` for generation of tokens

**Points to note on: Regex**

- [\\\n] -> matches a explicit newline.

```
"Hello \n World"
```

- \n -or> (\n) -or> [\n] -or> [\\\n] ->matches a implicit newline

```
"Hello
World"
```

➢ Run lexer to get tokens with simple details in symbol table with appropriate C++ input file.

**$ python3 _1_lexer.py**

## Lexer

- Always send raw string to lexer if sent as a string else send a file
- To run lexing phase `python _1_lexer.py`
- Input (implicit): The cpp file
- Output (implicit):
  - stdout: LexTokens
  - `p1_symbol_table.json` -> with ASCII and number conversion
  - `p2_symbol_table.json` -> without ASCII and number conversion

➢ Run parser to get TAC, AST, Scope table and updated Symbol table. The output will be written to 3code.txt file which will be given as input to optimizer.

**$ python3 _2_parser.py**

## Parser

- Synatx Analysis and Semantic Analysis is done here
- To make it happen, run the following:
  - `python _2_parser.py`
- Input (implicit): `p2_symbol_table.json`
- Output (implicit):
  - stdout:
    - Three Address Code
    - Scope tables
    - Symbol table
    - Abstarct syntax tree
  - `3code.txt` -> Three Address Code
  - `symbol_table.pkl` -> SymbolTable class is propogated from parser to optimizer
  - ply op: `parset.out` and `parsetab.py`

➢ Run optimizer to get the optimized TAC, the optimized TAC will be written to optimized-3code.txt

**$ python3 _3_optimizer.py 3code.txt**

## Optimizer

- Optimization
- To make it happen, run the following:
  - `python _3_optimizer.py`
- Input (implicit):
  - `symbol_table.pkl`
  - `3code.txt`
- Output (implicit):
  - stdout:
    - Optimized Three Address Code
    - Updated Symbol table
  - `optimized-3code` -> Optimized Three Address Code

# 7 RESULTS and SHORTCOMINGS

This project has helped us understand the work that is done into building a compiler. Each phase brings its own difficulties, and each phase requires a different strategy to overcome it. The implementation was worked upon in stages and a successful mini compiler for the “switch” and “for” constructs in C++ was built. Given C++ code as input, intermediate code is generated as required.

But a few shortcomings noticed in the implementation were, the compiler we built is a mini-compiler and doesn’t entirely mimic or compile all C++ code. We haven’t implemented Object Oriented Programming or STLs that make up the majority of C++. And the functions we have generated have been optimized specifically for the current language and grammar that has been elaborated on in this document. The generated code may be a bit buffed up compared to a highly optimized version of the same generated by an Official C++ Compiler.

# 8 SNAPSHOTS

## 8.1 Truncating ID Which Is Longer Than 31 Char’s

```
#include <iostream>
// trucating ID which is longer then 31 char's
int main()
{
    int abcdefghijklemnopqrstuvwxyzabcdefghijklmnopqrst;
    int 102_digit;
}


```

*Figure 1: Source Code with ID longer than 31char’s.*

```
LexToken(INT,'int',3,67), Column Range: (1, 3)
LexToken(MAIN,'main',3,71), Column Range: (5, 8)
LexToken(LPAREN,40,3,75), Column Range: (9, 9)
LexToken(RPAREN,41,3,76), Column Range: (10, 10)
LexToken(LBRACE,123,4,78), Column Range: (1, 1)
LexToken(INT,'int',5,84), Column Range: (5, 7)
ERROR: Identifier is longer than 31. Truncating it.
LexToken(ID,'abcdefghijklemnopqrstuvwxyzabcd',5,88), Column Range: (9, 39)
LexToken(SEMI,59,5,135), Column Range: (56, 56)
LexToken(INT,'int',6,141), Column Range: (5, 7)
ERROR: ID must not begin with a number at line no. 6
LexToken(SEMI,59,6,154), Column Range: (18, 18)
LexToken(RBRACE,125,7,156), Column Range: (1, 1)
```

*Figure 2: Identifying the ID longer than 31 and truncating it to 31 char’s.*

## 8.2 INT, FLOAT & ASCII Conversions

```
#include <iostream>
// conversion of int and float
//also, return ASCII for punctuations and single operators
int main()
{
    int int_var;
    float float_var;
    int_var = 30;
    float_var = 3.1415E+2;
    char c = 'a';
}

```

*Figure 3: Source Code with different Datatypes.*

```
LexToken(INT,'int',4,110), Column Range: (1, 3)
LexToken(MAIN,'main',4,114), Column Range: (5, 8)
LexToken(LPAREN,40,4,118), Column Range: (9, 9)
LexToken(RPAREN,41,4,119), Column Range: (10, 10)
LexToken(LBRACE,123,5,121), Column Range: (1, 1)
LexToken(INT,'int',6,127), Column Range: (5, 7)
LexToken(ID,'int_var',6,131), Column Range: (9, 15)
LexToken(SEMI,59,6,138), Column Range: (16, 16)
LexToken(FLOAT,'float',7,144), Column Range: (5, 9)
LexToken(ID,'float_var',7,150), Column Range: (11, 19)
LexToken(SEMI,59,7,159), Column Range: (20, 20)
LexToken(ID,'int_var',8,165), Column Range: (5, 11)
LexToken(EQUALS,61,8,173), Column Range: (13, 13)
LexToken(ICONST,30,8,175), Column Range: (15, 16)
LexToken(SEMI,59,8,177), Column Range: (17, 17)
LexToken(ID,'float_var',9,183), Column Range: (5, 13)
LexToken(EQUALS,61,9,193), Column Range: (15, 15)
LexToken(FCONST,314.15,9,195), Column Range: (17, 25)
LexToken(SEMI,59,9,204), Column Range: (26, 26)
LexToken(CHAR,'char',10,210), Column Range: (5, 8)
LexToken(ID,'c',10,215), Column Range: (10, 10)
LexToken(EQUALS,61,10,217), Column Range: (12, 12)
LexToken(CCONST,97,10,219), Column Range: (14, 14)
LexToken(SEMI,59,10,222), Column Range: (17, 17)
LexToken(RBRACE,125,11,224), Column Range: (1, 1)
```

*Figure 4: Type conversion reflected in LexToken.*

## 8.3 Unterminated Comment

```
#include <iostream>
// Detection of unterminated comment
int main()
{
    /* Unterminated comment is found here
    printf("Hello World");
}
```

*Figure 5: Source Code with Unterminated Comment.*

```
LexToken(INT,'int',3,57), Column Range: (1, 3)
LexToken(MAIN,'main',3,61), Column Range: (5, 8)
LexToken(LPAREN,40,3,65), Column Range: (9, 9)
LexToken(RPAREN,41,3,66), Column Range: (10, 10)
LexToken(LBRACE,123,4,68), Column Range: (1, 1)
ERROR: Unterminated comment found at line no. 5
```

*Figure 6: Detection of Unterminated Comment and exiting the lexing process.*

## 8.4 For Loop Demonstration

```
#include <iostream>

int main()
{
    int i;
    int j;
    int k;
    for (i = 0; i < 10; i++)
    {
        k = 100;
    }
}
```

*Figure 7 Sample C++ code for 'for'.*

```
LexToken(INT,'int',3,21), Column Range: (1, 3)
LexToken(MAIN,'main',3,25), Column Range: (5, 8)
LexToken(LPAREN,40,3,29), Column Range: (9, 9)
LexToken(RPAREN,41,3,30), Column Range: (10, 10)
LexToken(LBRACE,123,4,32), Column Range: (1, 1)
LexToken(INT,'int',5,38), Column Range: (5, 7)
LexToken(ID,'i',5,42), Column Range: (9, 9)
LexToken(SEMI,59,5,43), Column Range: (10, 10)
LexToken(INT,'int',6,49), Column Range: (5, 7)
LexToken(ID,'j',6,53), Column Range: (9, 9)
LexToken(SEMI,59,6,54), Column Range: (10, 10)
LexToken(INT,'int',7,60), Column Range: (5, 7)
LexToken(ID,'k',7,64), Column Range: (9, 9)
LexToken(SEMI,59,7,65), Column Range: (10, 10)
LexToken(FOR,'for',8,71), Column Range: (5, 7)
LexToken(LPAREN,40,8,75), Column Range: (9, 9)
LexToken(ID,'i',8,76), Column Range: (10, 10)
LexToken(EQUALS,61,8,78), Column Range: (12, 12)
LexToken(ICONST,0,8,80), Column Range: (14, 14)
LexToken(SEMI,59,8,81), Column Range: (15, 15)
LexToken(ID,'i',8,83), Column Range: (17, 17)
LexToken(LT,60,8,85), Column Range: (19, 19)
LexToken(ICONST,10,8,87), Column Range: (21, 22)
LexToken(SEMI,59,8,89), Column Range: (23, 23)
LexToken(ID,'i',8,91), Column Range: (25, 25)
LexToken(PLUSPLUS,'++',8,92), Column Range: (26, 27)
LexToken(RPAREN,41,8,94), Column Range: (28, 28)
LexToken(LBRACE,123,9,100), Column Range: (5, 5)
LexToken(ID,'k',10,110), Column Range: (9, 9)
LexToken(EQUALS,61,10,112), Column Range: (11, 11)
LexToken(ICONST,100,10,114), Column Range: (13, 15)
LexToken(SEMI,59,10,117), Column Range: (16, 16)
LexToken(RBRACE,125,11,123), Column Range: (5, 5)
LexToken(RBRACE,125,12,125), Column Range: (1, 1)
```

*Figure 8 Tokens generated from source code.*

```
        VAR                         i
        VAR                         j
        VAR                         k
        ASSIGN  0                   i_0
l0_0:
        LT      i_0     10          l0_2
        GOTO                        l0_3
l0_1:
        POSTINC                     i
        GOTO                        l0_0
l0_2:
        ASSIGN  100                 k_0
        GOTO                        l0_1
l0_3:
```

*Figure 9 Three Address Code.*

```
ABSTRACT SYNTAX TREE : (('i', 'int'), (('j', 'int'), (('k', 'int'), (('for', None, ('EXPRESSION CALCULATED', '=', 'i', ('0', 'INT')), ('CONDI', '<', 'i', '10'), ('POSTINC', '++', 'i'), ')', '{'), None))))
```

*Figure 10: Abstract Syntax Tree.*

```
        VAR                         i
        VAR                         j
        VAR                         k
        ASSIGN  0                   i_0
l0_0:
        LT      0       10          l0_2
        GOTO                        l0_3
l0_1:
        GOTO                        l0_0
l0_2:
        ASSIGN  100                 k_0
        GOTO                        l0_1
l0_3:
```

*Figure 11 Optimized Three Address Code.*

Symbol Table

| name | type | value |
|---|---|---|
| i_0 | INT | 0 |
| k_0 | INT | 100 |

*Figure 12 Symbol Table After Optimization.*

## 8.5 Switch Demonstration

```
#include <iostream>

int main()
{
    int i;
    i = 10;
    int j;
    j = i;
    switch (0)
    {
    case 0:
        int i;
        int j;
        j = i * i;
        j = j * j;
        break;
    case 1:
        int k;
        break;
    case 2:
        int z;
        z = 12 * 20;
    case 3:
        int y;
    default:
        int y;
    }
    int k;
    k = 100;
}
```

*Figure 13 Source code for 'swicth'.*

```
        ASSIGN  10              i_0
        VAR                     j
        ASSIGN  i_0             j_0
        GOTO                    l_comparisons
l0_0:
        VAR                     i
        VAR                     j
        MUL     i_0     i_0     t0
        ASSIGN  t0              j_1
        MUL     j_2     j_2     t1
        ASSIGN  t1              j_2
        GOTO                    l_next_switch
l0_1:
        VAR                     k
        GOTO                    l_next_switch
l0_2:
        VAR                     z
        ASSIGN  12              t2
        ASSIGN  20              t3
        MUL     t2      t3      t4
        ASSIGN  t4              z_0
l0_3:
        VAR                     y
l0_4:
        VAR                     y
        GOTO                    l_next_switch
l_comparisons:
        EQ      0       0       l0_0
        EQ      0       1       l0_1
        EQ      0       2       l0_2
        EQ      0       3       l0_3
        GOTO                    l0_4
l_next_switch:
        VAR                     k
        ASSIGN  100             k_0
```

*Figure 14: Three Address Code.*

Table for : GLOBAL scope

| name | type | value | address | lineno | code |
|---|---|---|---|---|---|
| i | int | 10 | 0x1e16bba77b0 | 5 | 0 |
| j | int | | 0x1e16bff72f0 | 7 | 0 |
| k | int | 100 | 0x1e16bd060f0 | 18 | 0 |

Table for : SWITCH DEF scope

| address | code | lineno | name | type | value |
|---|---|---|---|---|---|
| 0x1e16bba7930 | 0 | 24 | y | int | |

Table for : SWITCH CASE scope

| address | code | lineno | name | type | value |
|---|---|---|---|---|---|
| 0x1e16bba7930 | 0 | 24 | y | int | |

Table for : SWITCH CASE scope

| address | code | lineno | name | type | value |
|---|---|---|---|---|---|
| 0x1e16bba77f0 | 0 | 21 | z | int | |

Table for : SWITCH CASE scope

| address | code | lineno | name | type | value |
|---|---|---|---|---|---|
| 0x1e16bd060f0 | 0 | 18 | k | int | |

Table for : SWITCH CASE scope

| name | type | value | address | lineno | code |
|---|---|---|---|---|---|
| i | int | | 0x1e16bba77b0 | 5 | 0 |
| j | int | | 0x1e16bff72f0 | 7 | 0 |

*Figure 15: Scope Table .*

```
ABSTRACT SYNTAX TREE : (('i', 'int'), (('EXPRESSION CALCULATED', '=', 'i', ('10', 'INT'))
, (('j', 'int'), (('EXPRESSION CALCULATED', '=', 'j', 'i'), (('switch', None, ')', '{', (
('labeled_statement', ('case', None, None), ';'), (('labeled_statement', ('case', None, N
one), ';'), (('case', None, None), (('case', None, None), (('default', (('y', 'int'), Non
e)), None)))))), (('k', 'int'), (('EXPRESSION CALCULATED', '=', 'k', ('100', 'INT')), Non
e)))))))
```

*Figure 16: Abstract Syntax Tree.*

```
        VAR                         i
        ASSIGN  10                  i_0
        VAR                         j
        ASSIGN  10                  j_0
        GOTO                        l_comparisons
l0_0:
        VAR                         i
        VAR                         j
        ASSIGN  100                 j_1
        ASSIGN  0                   j_2
        GOTO                        l_next_switch
l0_1:
        VAR                         k
        GOTO                        l_next_switch
l0_2:
        VAR                         z
        ASSIGN  240                 z_0
l0_3:
        VAR                         y
l0_4:
        VAR                         y
        GOTO                        l_next_switch
l_comparisons:
        EQ      0       0           l0_0
        EQ      0       1           l0_1
        EQ      0       2           l0_2
        EQ      0       3           l0_3
        GOTO                        l0_4
l_next_switch:
        VAR                         k
        ASSIGN  100                 k_0
```

*Figure 17: Optimized Three Address Code.*

Symbol Table

| name | type | value |
|---|---|---|
| i_0 | INT | 10 |
| j_0 | INT | 10 |
| t0 | INT | 100 |
| j_1 | INT | 100 |
| t1 | CHAR | 0 |
| j_2 | CHAR | 0 |
| t2 | INT | 12 |
| t3 | INT | 20 |
| t4 | INT | 240 |
| z_0 | INT | 240 |
| k_0 | INT | 100 |

*Figure 18: Symbol Table After Optimization.*

## 8.6 Nested For Demonstration

```
#include <iostream>

int main()
{
    int i;
    int j;
    int k;
    for (i = 0; i < 10; i++)
    {
        for (j = 0; j < 12; j++)
        {
            int i = 200;
        }
        k = 100;
    }
}

```

*Figure 19: Source code for 'Nested for'.*

```
        VAR                         i
        VAR                         j
        VAR                         k
        ASSIGN  0                   i_0
l0_0:
        LT      i_0     10          l0_2
        GOTO                        l0_3
l0_1:
        POSTINC                     i
        GOTO                        l0_0
l0_2:
        ASSIGN  0                   j_0
l0_3_0:
        LT      j_0     12          l0_3_2
        GOTO                        l0_3_3
l0_3_1:
        POSTINC                     j
        GOTO                        l0_3_0
l0_3_2:
        VAR                         i
        ASSIGN  200                 i_1
        GOTO                        l0_3_1
l0_3_3:
        ASSIGN  100                 k_0
        GOTO                        l0_1
l0_3:
                Table for : GLOBAL scope
```

*Figure 20: Three Address Code.*

```
ABSTRACT SYNTAX TREE : (('i', 'int'), (('j', 'int'), (('k', 'int'), (('for', None, ('EXPR
ESSION CALCULATED', '=', 'i', ('0', 'INT')), ('CONDI', '<', 'i', '10'), ('POSTINC', '++',
 'i'), ')', '{'), None))))
```

*Figure 21: Abstract Syntax Tree.*

```
        VAR                     i
        VAR                     j
        VAR                     k
        ASSIGN  0               i_0
l0_0:
        LT      0       10      l0_2
        GOTO                    l0_3
l0_1:
        GOTO                    l0_0
l0_2:
        ASSIGN  0               j_0
l0_3_0:
        LT      0       12      l0_3_2
        GOTO                    l0_3_3
l0_3_1:
        GOTO                    l0_3_0
l0_3_2:
        VAR                     i
        ASSIGN  200             i_1
        GOTO                    l0_3_1
l0_3_3:
        ASSIGN  100             k_0
        GOTO                    l0_1
l0_3:
```

*Figure 22: Optimized Three Address Code.*

Symbol Table

| name | type | value |
|---|---|---|
| i_0 | INT | 0 |
| j_0 | INT | 0 |
| i_1 | INT | 200 |
| k_0 | INT | 100 |

*Figure 23: Symbol Table After Optimization.*

## 8.7 Variable Propagation & Dead Code Elimination

```
#include <iostream>
int main()
{
    int a = 10;
    int b = 20;
    a = b;
    int c = a;
}
```

*Figure 24: Source Code for "Optimization".*

```
VAR                     a
ASSIGN  10              a_0
VAR                     b
ASSIGN  20              b_0
ASSIGN  b_0             a_1
VAR                     c
ASSIGN  a_1             c_0
```

*Figure 25: Three Address Code.*

```
VAR                      a
ASSIGN  10               a_0
VAR                      b
ASSIGN  20               b_0
ASSIGN  20               a_1
VAR                      c
ASSIGN  20               c_0
```

*Figure 26: Optimized Three Address Code.*

# 9 CONCLUSION

Compilers being a part of every coders life it is mastered and well known only by few. It was fun learning this subject and implementing it hands on. This project had a huge impact on the way we think when we write code, and it has made us better coders by reducing the bugs that we imply, and unwanted lines introduced during a software development. We have gained a better insight into the different phases of a compiler, in general, and understood a little more about C++ compiler in specific.

# 10 FURTHER ENHANCEMENT

The compiler implemented by us does not entirely support all constructs and optimizations provided the g++ compiler for C++. Some of them include:

1. Classes and objects
2. Array initialization
3. Handling pointer types
4. Function definition
5. Loop unrolling
6. Move loop invariant code outside the loop