A picture containing text, clipart, sign

Description automatically generated

***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**  **Abhishek R**  **Vybhav Bhadri S** | **PES1201801481**  **PES1201801682**  **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**

|  |  |
| --- | --- |
| **Chapter No.** | **Title** |
|  | **INTRODUCTION (Mini-Compiler is built for which language. Provide sample input and output of your project)** |
|  | **ARCHITECTURE OF LANGUAGE:**   * **What all have you handled in terms of syntax and semantics for the chosen language?** |
|  | **LITERATURE SURVEY (if any paper referred or link used)** |
|  | **CONTEXT FREE GRAMMAR (which you used to implement your project)** |
|  | **DESIGN STRATEGY (used to implement the following)**   * **SYMBOL TABLE CREATION** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** |
|  | **IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following):**   * **SYMBOL TABLE CREATION** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** * **Provide instructions on how to build and run your program.** |
|  | **RESULTS AND possible shortcomings of your Mini-Compiler** |
|  | **SNAPSHOTS (of different outputs)** |
|  | **CONCLUSIONS** |
|  | **FURTHER ENHANCEMENTS** |
| **REFERENCES/BIBLIOGRAPHY** | |

# 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.

# 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.

# 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>

# 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 LBRACE 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 delete\_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

# DESIGN STRATEGY

## 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.

## 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.

## 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.

## 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.

# IMPLEMENTATION DETAILS

## 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.

## 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.

## 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.

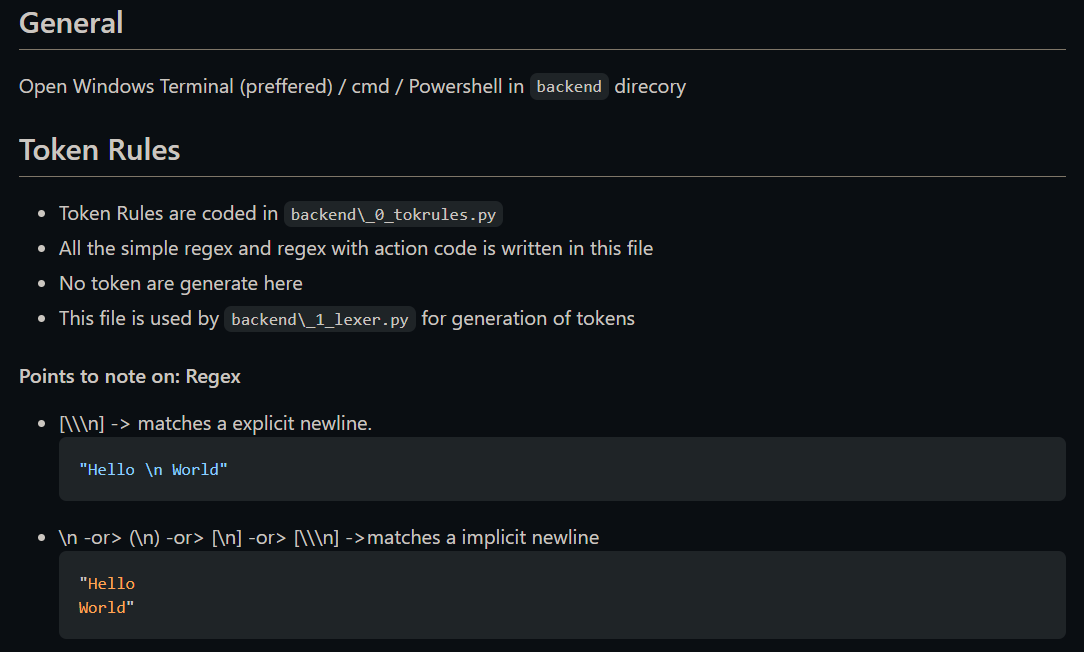
## Error Handling

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

## Tools and Instruction to run the code

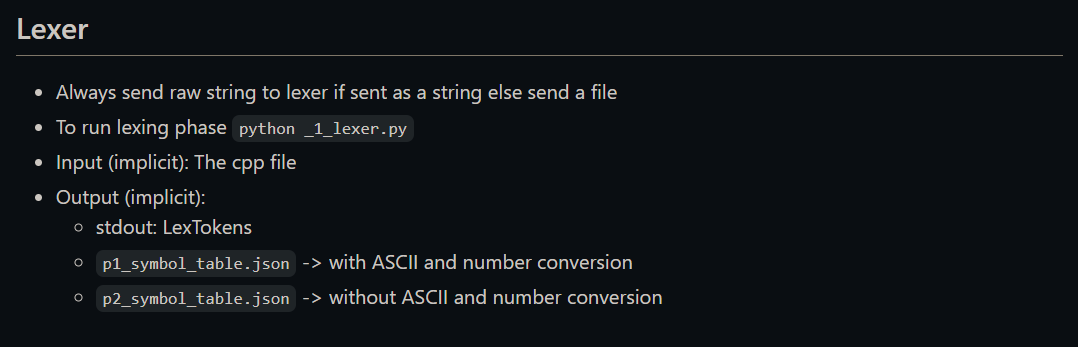
* Visual Studio Code
* Python 3 and above
* Python module requirements:
* ply
* termcolor
* pandas
* tabulate

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



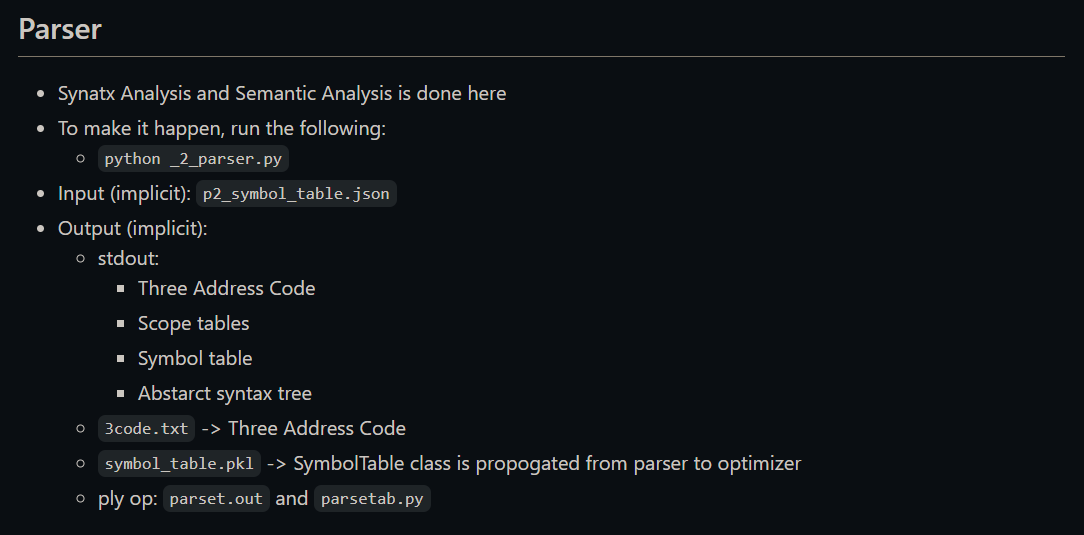
* Run lexer to get tokens with simple details in symbol table with appropriate C++ input file.

**$ python3 \_1\_lexer.py**

****

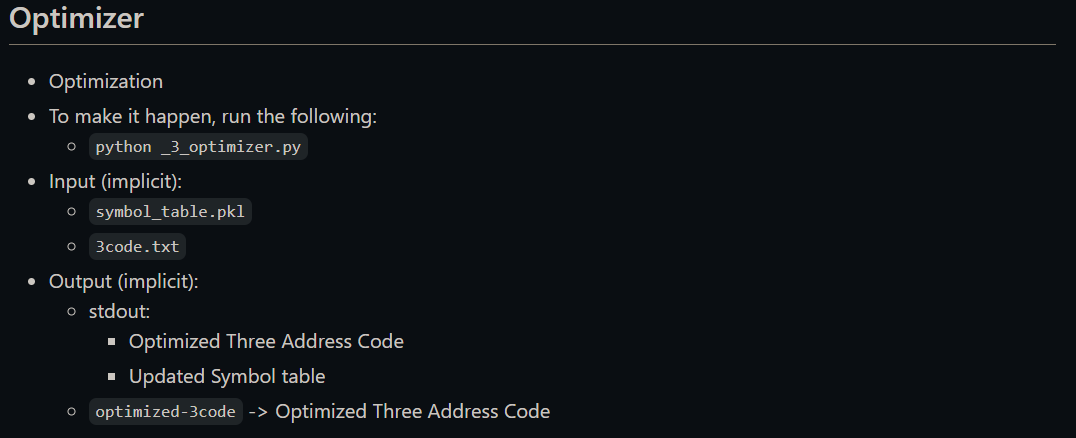
* Run parser to get TAC, AST, Scope table and updated Symbol table. The output will be written to 3code.txt file which will be given as input to optimizer.

**$ python3 \_2\_parser.py**

****

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

**$ python3 \_3\_optimizer.py 3code.txt**

****

# 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.

# SNAPSHOTS

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

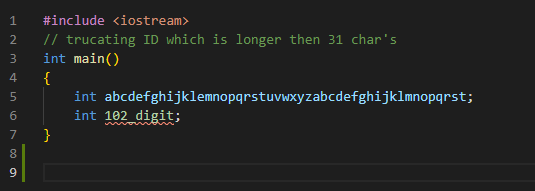


Figure : Source Code with ID longer than 31char’s.

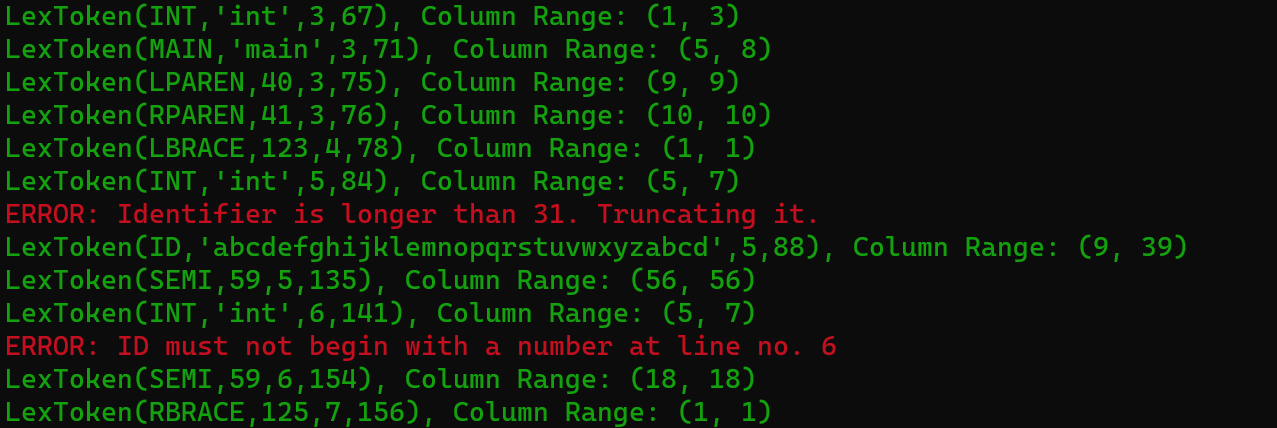


Figure : Identifying the ID longer than 31 and truncating it to 31 char’s.

## INT, FLOAT & ASCII Conversions

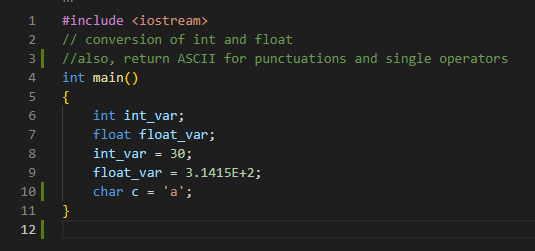


Figure : Source Code with different Datatypes.

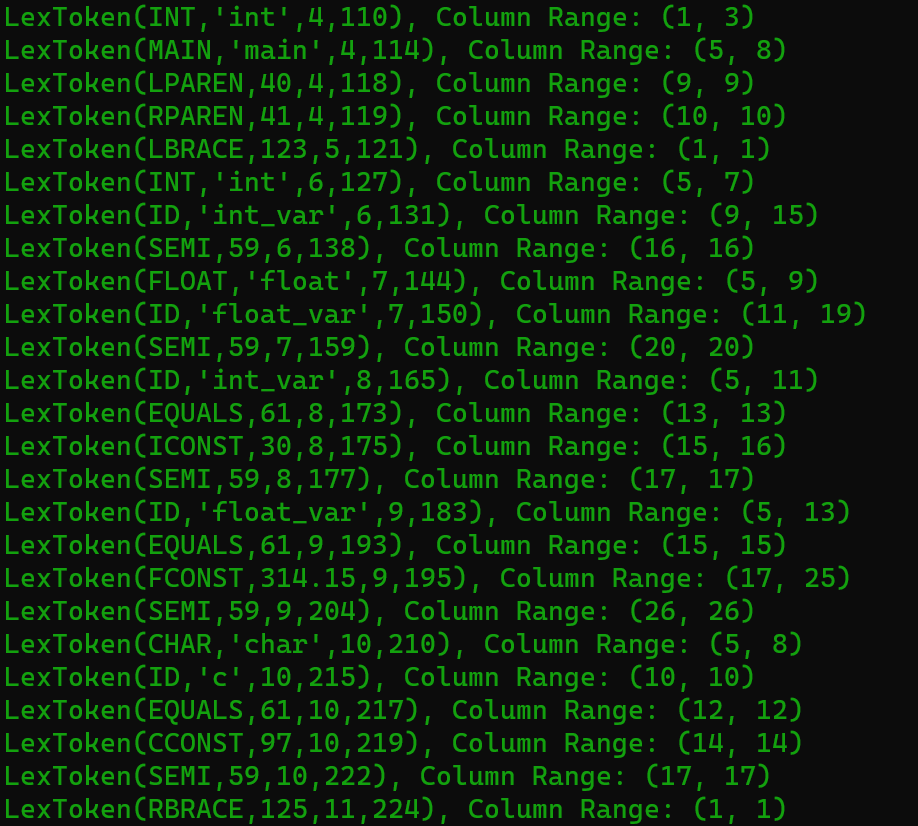


Figure : Type conversion reflected in LexToken.

## Unterminated Comment

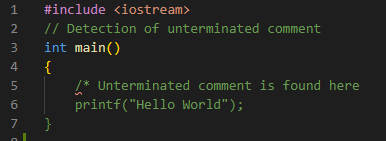


Figure : Source Code with Unterminated Comment.

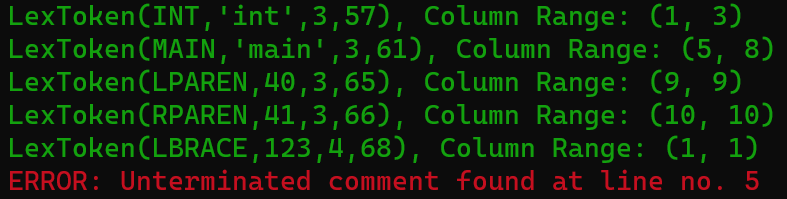


Figure : Detection of Unterminated Comment and exiting the lexing process.

## For Loop Demonstration

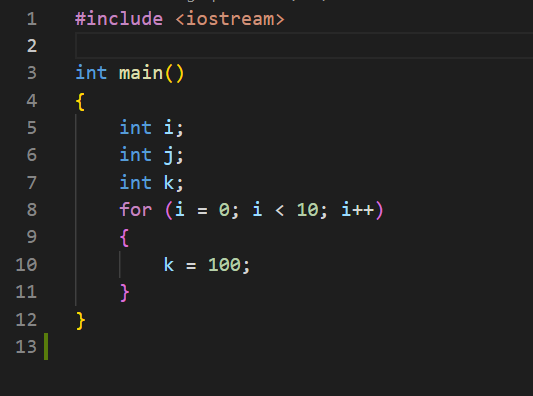


Figure Sample C++ code for 'for'.

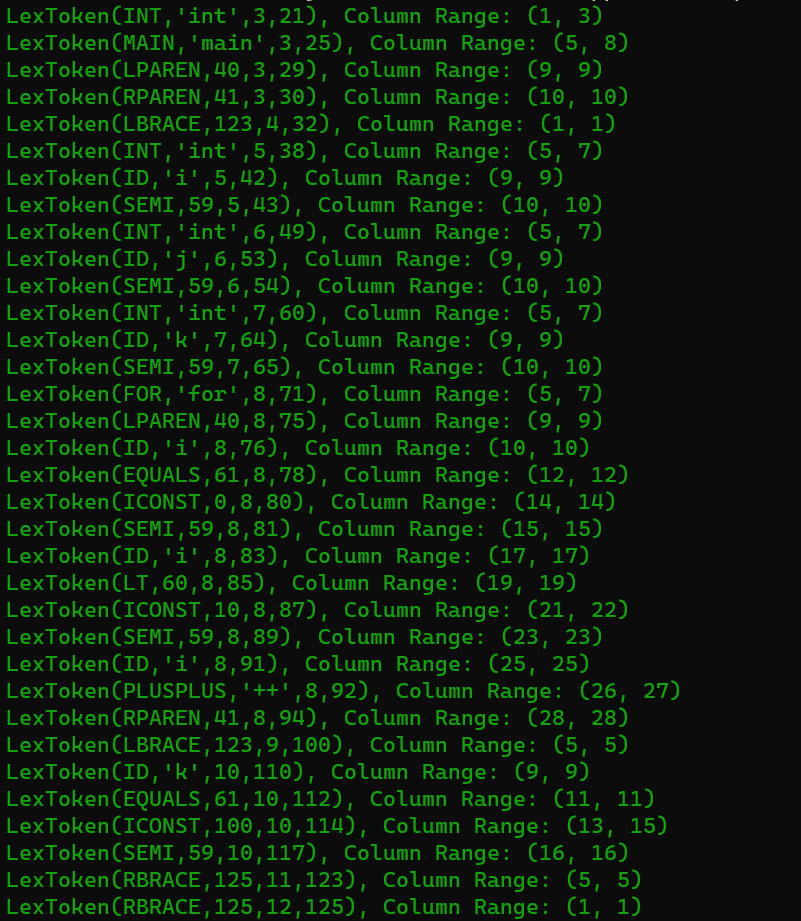


Figure Tokens generated from source code.

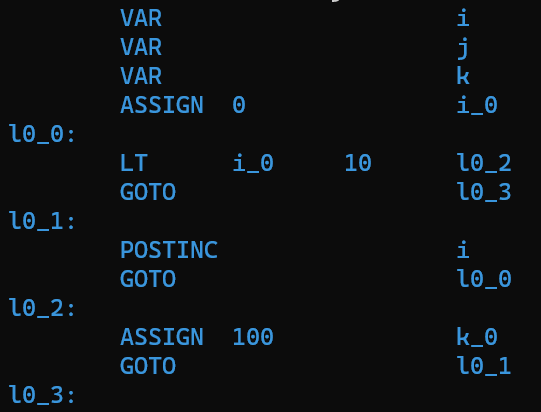


Figure Three Address Code.

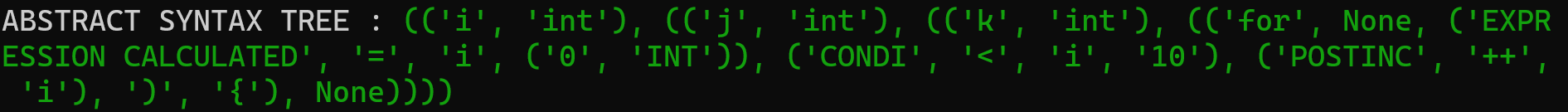


Figure : Abstract Syntax Tree.



Figure Optimized Three Address Code.

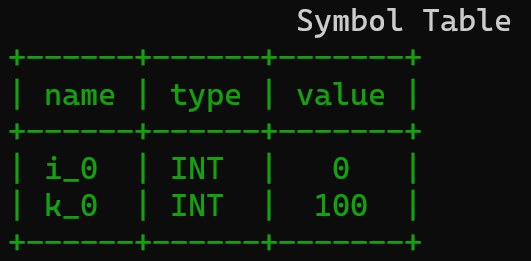


Figure Symbol Table After Optimization.

## Switch Demonstration

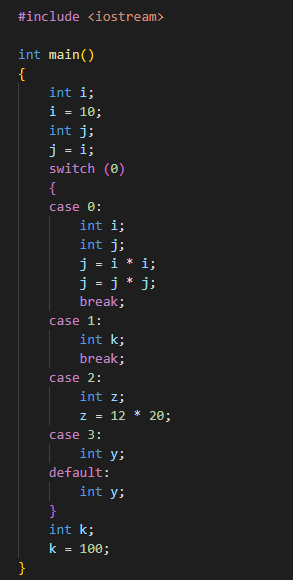


Figure Source code for ‘swicth’.

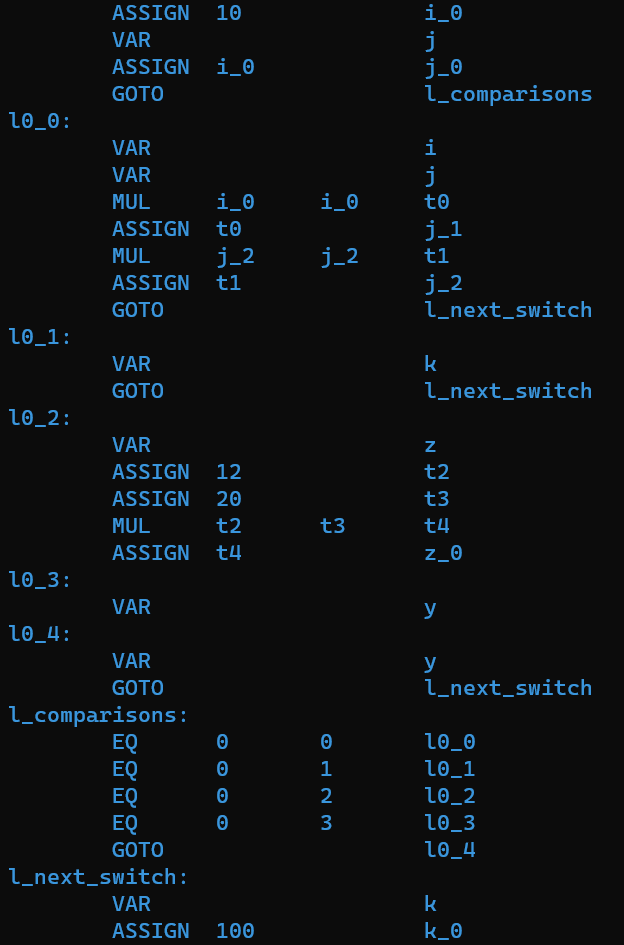


Figure : Three Address Code.

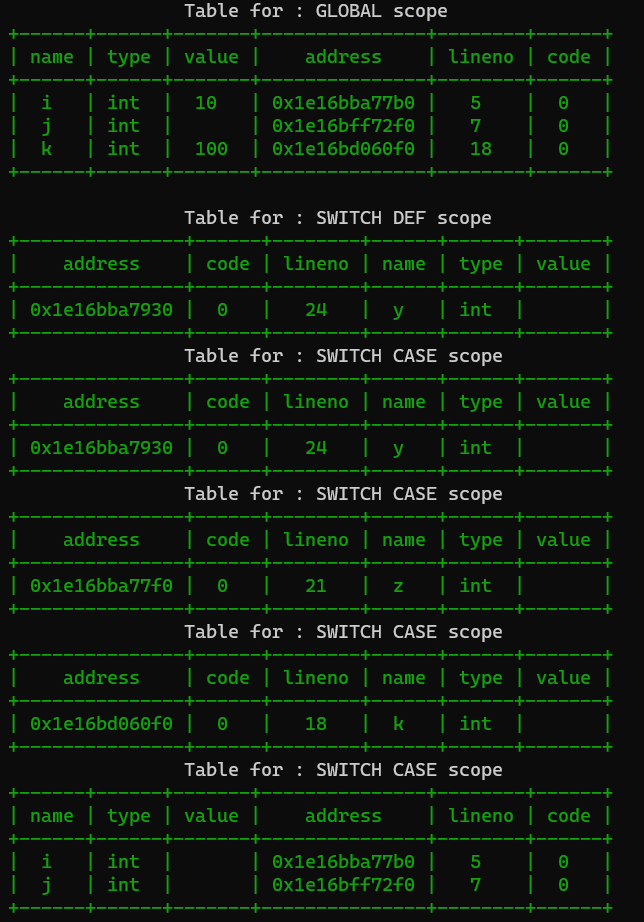


Figure : Scope Table .

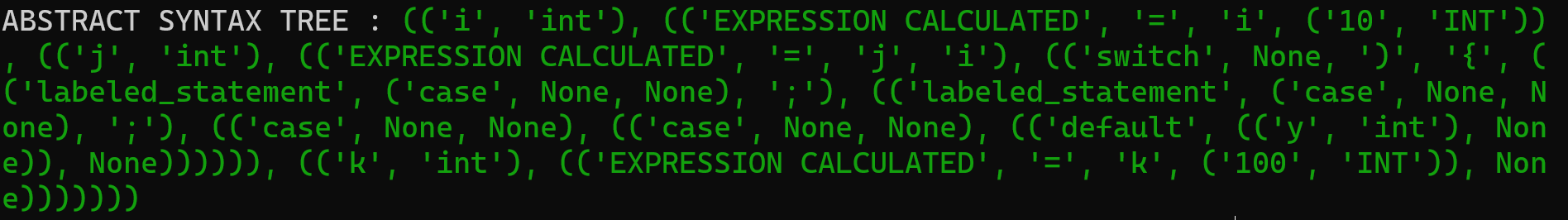


Figure : Abstract Syntax Tree.

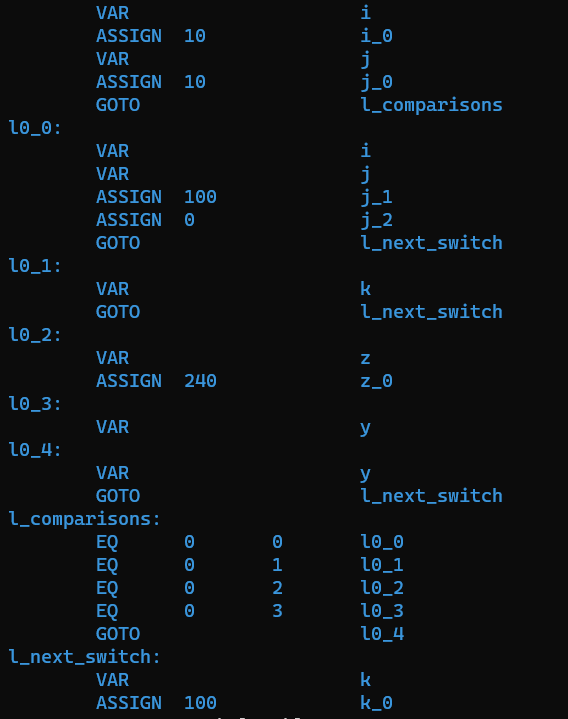


Figure : Optimized Three Address Code.

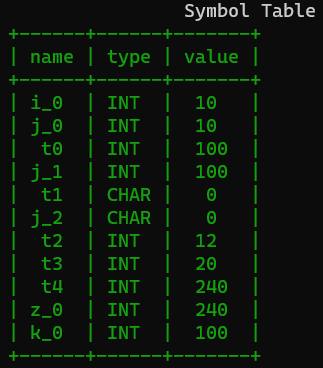


Figure : Symbol Table After Optimization.

## Nested For Demonstration

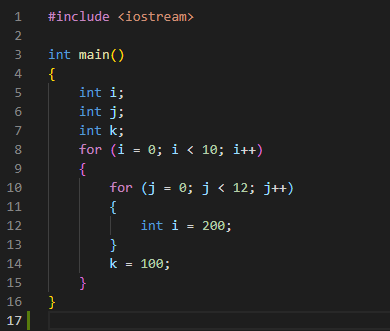


Figure : Source code for ‘Nested for’.

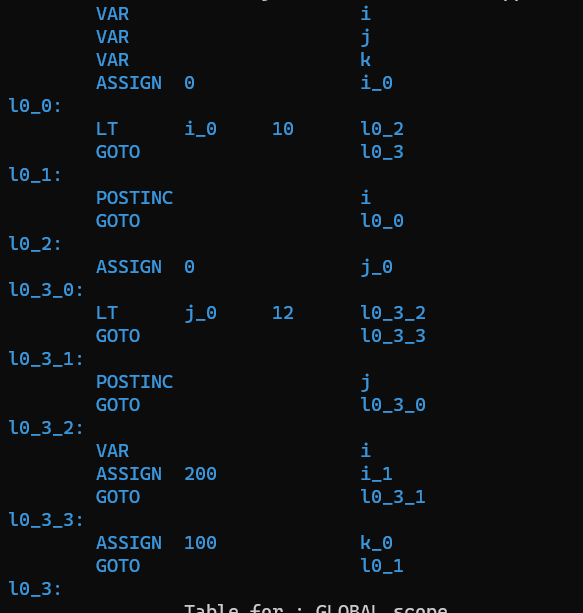


Figure : Three Address Code.

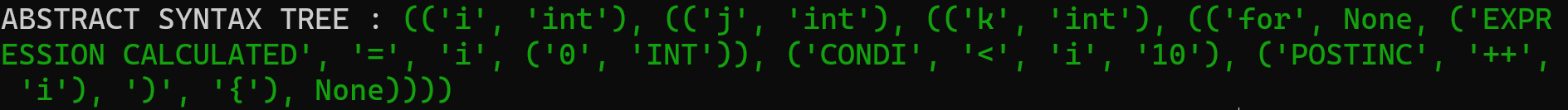


Figure : Abstract Syntax Tree.

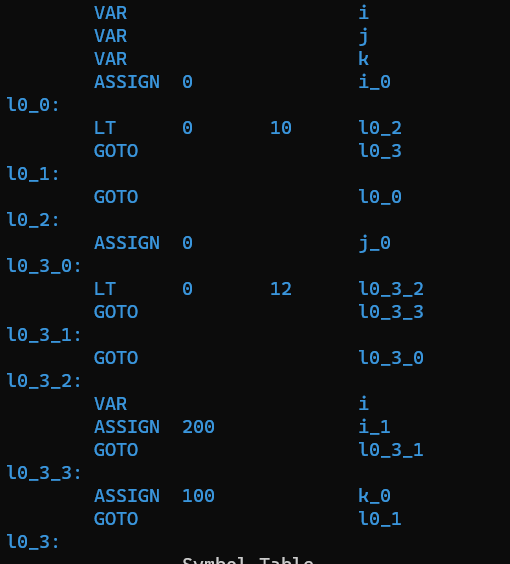


Figure : Optimized Three Address Code.

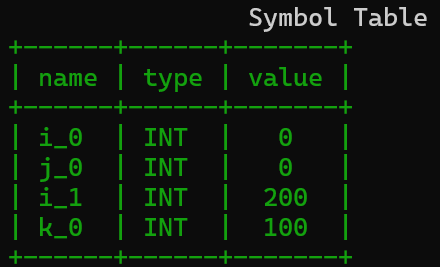


Figure : Symbol Table After Optimization.

## Variable Propagation & Dead Code Elimination

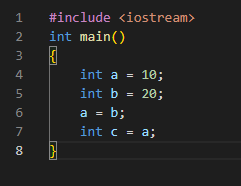


Figure : Source Code for “Optimization”.

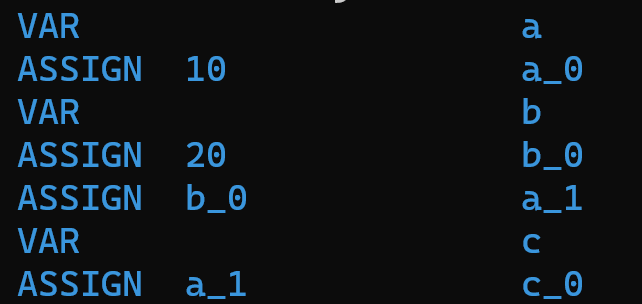


Figure : Three Address Code.

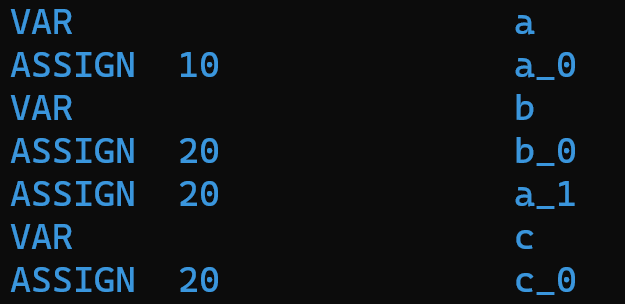


Figure : Optimized Three Address Code.

# 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.

# 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