

***Report on***

**“**​**C Compiler in java for ‘While’ , ’ if else’ , ‘Ternary’ and ‘for’ constructs”**

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

***Compiler Design Laboratory***

**Bachelor of Technology**

**in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **Anush V Kini** | **PES1201701646** |
| **Pruthvish E** | **PES1201701629** |
| **Pradyumna YM** | **PES1201700986** |

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

[**CHAPTER 1. INTRODUCTION**](#_wcp1qmb0rm66) **4**

[**CHAPTER 2. ARCHITECTURE**](#_ughvfpgzo4k5) **5**

[**CHAPTER 3. LITERATURE SURVEY**](#_cx0ki08fg8if) **7**

[**CHAPTER 4. CONTEXT-FREE GRAMMAR**](#_2iadjw7mcyol) **7**

[**CHAPTER 5. DESIGN AND IMPLEMENTATION**](#_o7hmafyt372m) **12**

[**CHAPTER 6. SNAPSHOTS**](#_7hn8inauvmm2) **14**

[**CHAPTER 7. RESULTS AND CONCLUSION**](#_xh4xhiwnbfhk) **20**

[**CHAPTER 8. SHORTCOMINGS**](#_a0jt8nh47awb) **21**

[**CHAPTER 9. FUTURE ENHANCEMENTS**](#_49o9jbdt8w3r) **21**

[**REFERENCES**](#_yu23rolqlyoc) **21**

**TABLE OF CONTENTS**

|  |  |  |  |
| --- | --- | --- | --- |
| **Chapter** | **Title** |  | **Page No.** |
| **No.** |  |  |  |
| **1.** | **INTRODUCTION (Mini-Compiler is built for which language.** | | **04** |
|  | **Provide sample input and output of your project)** | |  |
| **2.** | **ARCHITECTURE OF LANGUAGE:** | | **05** |
|  | **● What all have you handled in terms of syntax and** | |  |
|  |  | **semantics for the chosen language?** |  |
| **3.** | **LITERATURE SURVEY (if any paper referred or link used)** | | **07** |
| **4.** | **CONTEXT-FREE GRAMMAR (which you used to implement** | | **07** |
|  | **your project)** | |  |
| **5.1** | **DESIGN STRATEGY (used to implement the following)** | | **12** |
|  | **●** | **SYMBOL TABLE CREATION** |  |
|  | **●** | **ABSTRACT SYNTAX TREE** |  |
|  | **●** | **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).** |  |
| **5.2** | **IMPLEMENTATION DETAILS (TOOL AND DATA** | | **13** |
|  | **STRUCTURES USED in order to implement the following):** | |  |
|  | **●** | **SYMBOL TABLE CREATION** |  |
|  | **● ABSTRACT SYNTAX TREE (**​**internal representation)** | |  |
|  | **●** | **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** | |  |
|  |  | **computer** |  |
| **6.** | **SNAPSHOTS (of different outputs)** | | **14** |
| **7.** | **RESULTS AND CONCLUSION** | | **20** |
| **8.** | **SHORTCOMINGS**​ **of our Mini-Compiler** | | **21** |
| **9.** | **FURTHER ENHANCEMENTS** | | **21** |
| **REFERENCES/BIBLIOGRAPHY** | | | **21** |

# CHAPTER 1. INTRODUCTION

In this document, we report on our attempt to create a mini compiler of the C programming language. We have implemented the front end of the compiler for C language using Lex and Yacc. This compiler can handle the following constructs.

* While loop
* If else statements
* Ternary Operator
* For loop

Using GNU C Compiler that is, gcc’s latest version as our reference, we developed a mini-compiler that can handle C operations of basic and intermediate complexity along with the constructs mentioned above. A source program in C is translated to a symbol table, abstract syntax tree, intermediate code, and optimized intermediate code.

# 

# CHAPTER 2. ARCHITECTURE

1. Used Flex/lex to create the scanner for our language.
2. Used Bison/yacc to implement grammar rules to the token generated in the scanner phase.
3. All token names are in capitals and everything else is in caps.
4. Operators and special characters implemented in our programming language:
   1. Binary operators: + - \* / %
   2. Unary operators: ! ~ + - ++ -- (postfix and prefix)
   3. Ternary operator: conditional operator variable = (expression) ? literal1 : literal2
5. Ignore comments and white-spaces

Single line comments starting with //

Multi-line comments enclosed within /\* …... \*/

1. Types: int, float, char, void
2. Constructs ‘ while’ and ‘for’ loops, if else and ternary operators.
3. Includes functions definitions and function calls.
4. No conflicts and errors in our code/grammar.
5. Warnings and Error recovery
6. 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.

1. Division by zero. Assign value of INT\_MAX and continue.
2. Errors:
3. Redefinition of identifiers within the same scope
4. Use of undeclared identifiers
5. Invalid operands to ‘%’. Implicit conversion to INT
6. Syntax errors based on the specified grammar.
7. Error handling related to scope and declaration.
8. All the errors and warnings are displayed along with line number
9. 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.
10. 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.
11. Code Optimizations techniques implemented in java:
12. Common Subexpression elimination
13. Dead Code Elimination
14. Intermediate code generation - In quadruple format

# 

# CHAPTER 3. LITERATURE SURVEY

3.1

Course material shared for Compiler Design Course (especially ICG and Code optimisation)

3.2

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

# CHAPTER 4. CONTEXT-FREE GRAMMAR

S : program

program

: HASH INCLUDE '<' libraries '>' program

* HASH INCLUDE HEADER\_LITERAL program
* translation\_unit

;

translation\_unit

: ext\_dec

* translation\_unit ext\_dec

;

ext\_dec

: declaration

* function\_definition

;

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

| function\_call ';'

| RETURN expression\_statement | printstat ';'

;

printstat

: PRINT '(' STRING\_LITERAL ')'

* PRINT '(' STRING\_LITERAL ',' expression ')'

;

declaration

* type\_specifier init\_declarator\_list ';'

;

statement

* compound\_statement

| expression\_statement | iteration\_statement

| while\_statement

| if\_else\_statement

;

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

;

while\_statement

: [WHILE](https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#WHILE) '(' [expression](https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#expression) ')' [statement](https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#statement) ;

if\_else\_statement

: [IF](https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#IF) '(' [expression](https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#expression) ')' [statement](https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#statement)

| [IF](https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#IF) '(' [expression](https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#expression) ')' [statement](https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#statement) [ELSE](https://www.lysator.liu.se/c/ANSI-C-grammar-l.html#ELSE) [statement](https://www.lysator.liu.se/c/ANSI-C-grammar-y.html#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

;

assignment\_operator

: '='

| ADD\_ASSIGN

* okSUB\_ASSIGN
* MUL\_ASSIGN
* DIV\_ASSIGN
* MOD\_ASSIGN

;

conditional\_expression

: equality\_expression

* equality\_expression '?' expression ':' conditional\_expression

;

expression\_statement

: ';'

* expression ';'

;

expression

: assignment\_expression

* expression ',' assignment\_expression

;

primary\_expression

: IDENTIFIER

* INTEGER\_LITERAL
* FLOAT\_LITERAL
* CHARACTER\_LITERAL
* '(' expression ')'

;

postfix\_expression

: primary\_expression

* postfix\_expression INC\_OP
* postfix\_expression DEC\_OP

;

unary\_expression

: postfix\_expression

* unary\_operator unary\_expression

;

unary\_operator

* '+' | '-' | '!' | '~'

| "INC\_OP" | "DEC\_OP"

;

equality\_expression

: relational\_expression

* equality\_expression EQ\_OP relational\_expression
* equality\_expression NE\_OP relational\_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

;

additive\_expression

: multiplicative\_expression

* additive\_expression '+' multiplicative\_expression
* additive\_expression '-' multiplicative\_expression

;

multiplicative\_expression

: unary\_expression

* multiplicative\_expression '\*' unary\_expression
* multiplicative\_expression '/' unary\_expression
* multiplicative\_expression '%' unary\_expression

;

function\_definition

* type\_specifier declarator compound\_statement | declarator compound\_statement

;

function\_call

* declarator '(' identifier\_list ')' | declarator '(' ')'

;

declarator

: IDENTIFIER

* declarator '(' parameter\_list ')'
* declarator '(' identifier\_list ')'
* declarator '(' ')'

;

parameter\_list

: parameter\_declaration

* parameter\_list ',' parameter\_declaration

;

parameter\_declaration

* type\_specifier IDENTIFIER | type\_specifier

;

identifier\_list

: IDENTIFIER

* identifier\_list ',' IDENTIFIER;

# CHAPTER 5. DESIGN AND IMPLEMENTATION

**5.1 Design**

Language: C compiler in java

Tools : Lex,Yacc and java.

Data Types : int, float, char, void

Constructs : if else , while ,for and ternary

* Symbol Table : Symbol table is a data structure that tracks the current bindings of identifiers for performing semantic checks and generating code efficiently. We have implemented the symbol table as a linked list of structures. The members of the structures include variable name, line of declaration, data type, value, scope. Every new variable encountered in the program is entered into the symbol table.

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

* Abstract Syntax Tree : We have implemented a binary tree to represent the abstract syntax tree internally , we have executed this for if else , while , ternary expressions and the for construct, and output the tree in preorder manner and in tree format.
* 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 a quadruple table. The quadruple table consists of an 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 common subexpression elimination and dead code elimination in java.
* Error Handling : In case of syntax error, the compilation is halted, and an error message along with the line number where error occured 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.

**5.2 Implementation :**

* The tools we have used for implementing the code are lex, yacc and java .
* 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 the 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 and in graphical tree format.
* The intermediate code generation also happens on the fly.
* After generating intermediate code, optimization is done by doing dead code elimination and common subexpression elimination.
* Code optimisation uses intermediate three address code for optimization.

Commands to execute the code:

lex ast.l

yacc -d ast.y

gcc lex.yy.c y.tab.c -ll -ly -o AST

lex icg.l

yacc -d icg.y

gcc lex.yy.c y.tab.c -ll -ly -o ICG

./AST <test.c

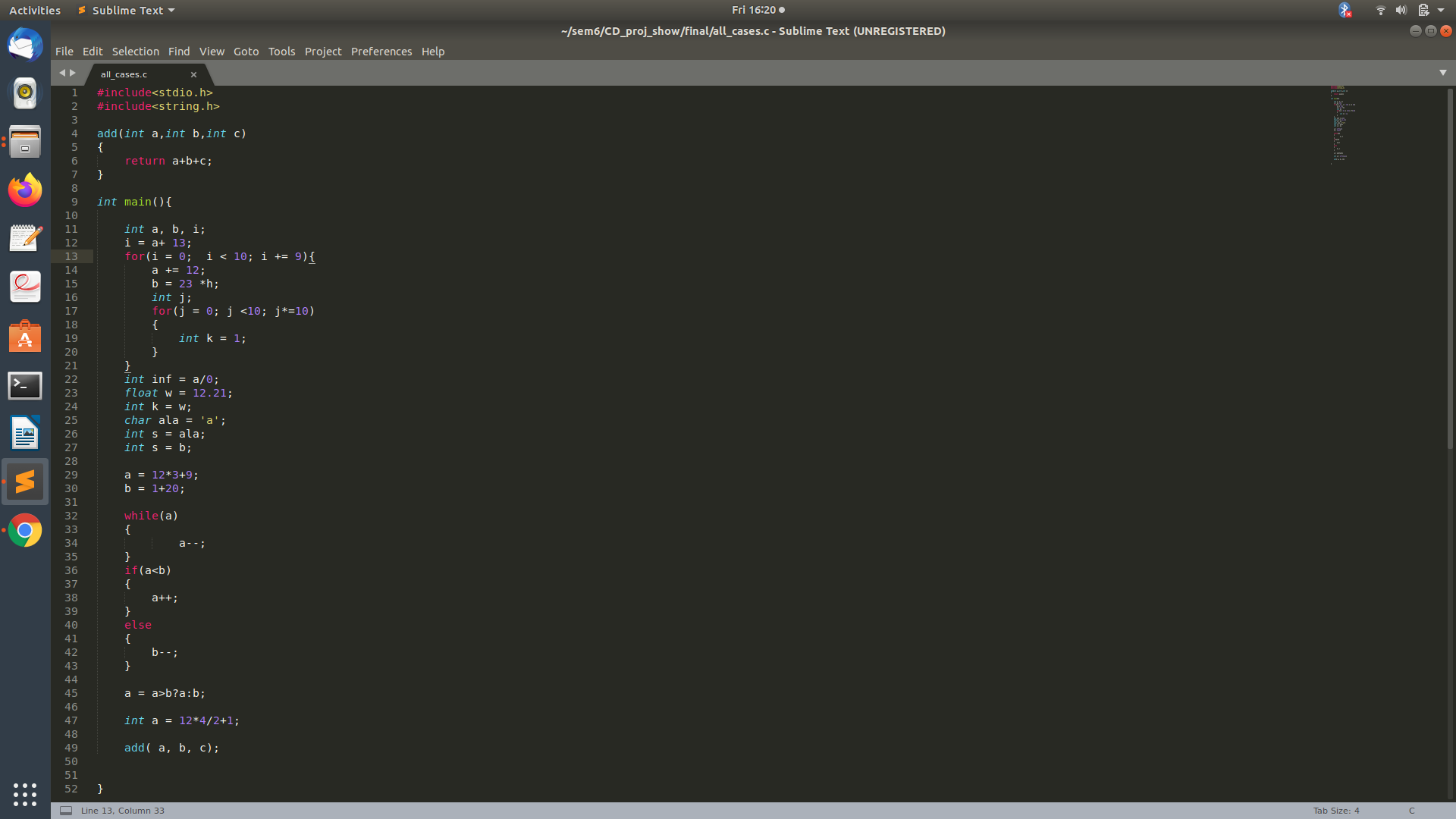
./ICG <test.c

java optimize1.java

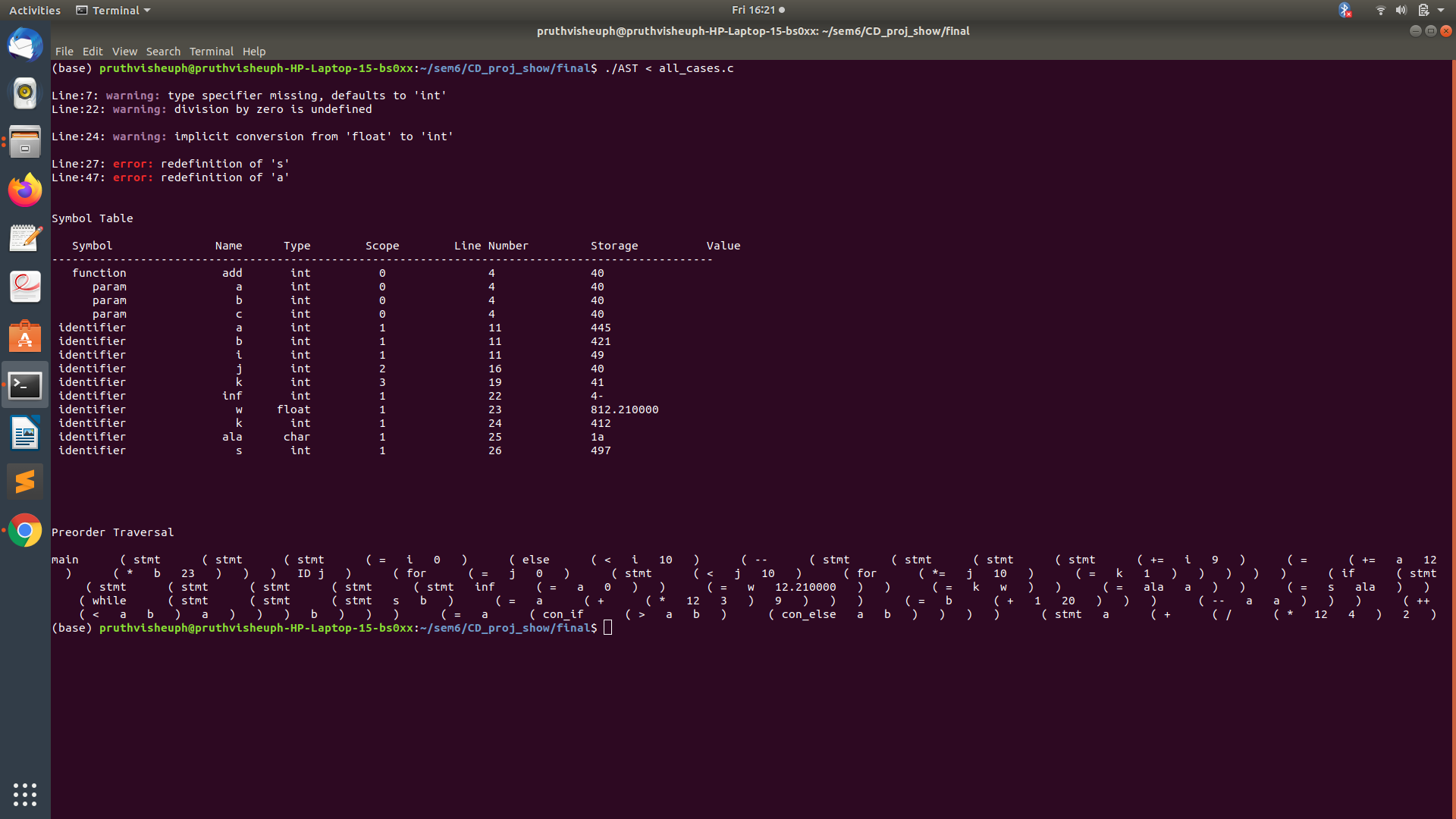
java optimize2.java

# CHAPTER 6. SNAPSHOTS

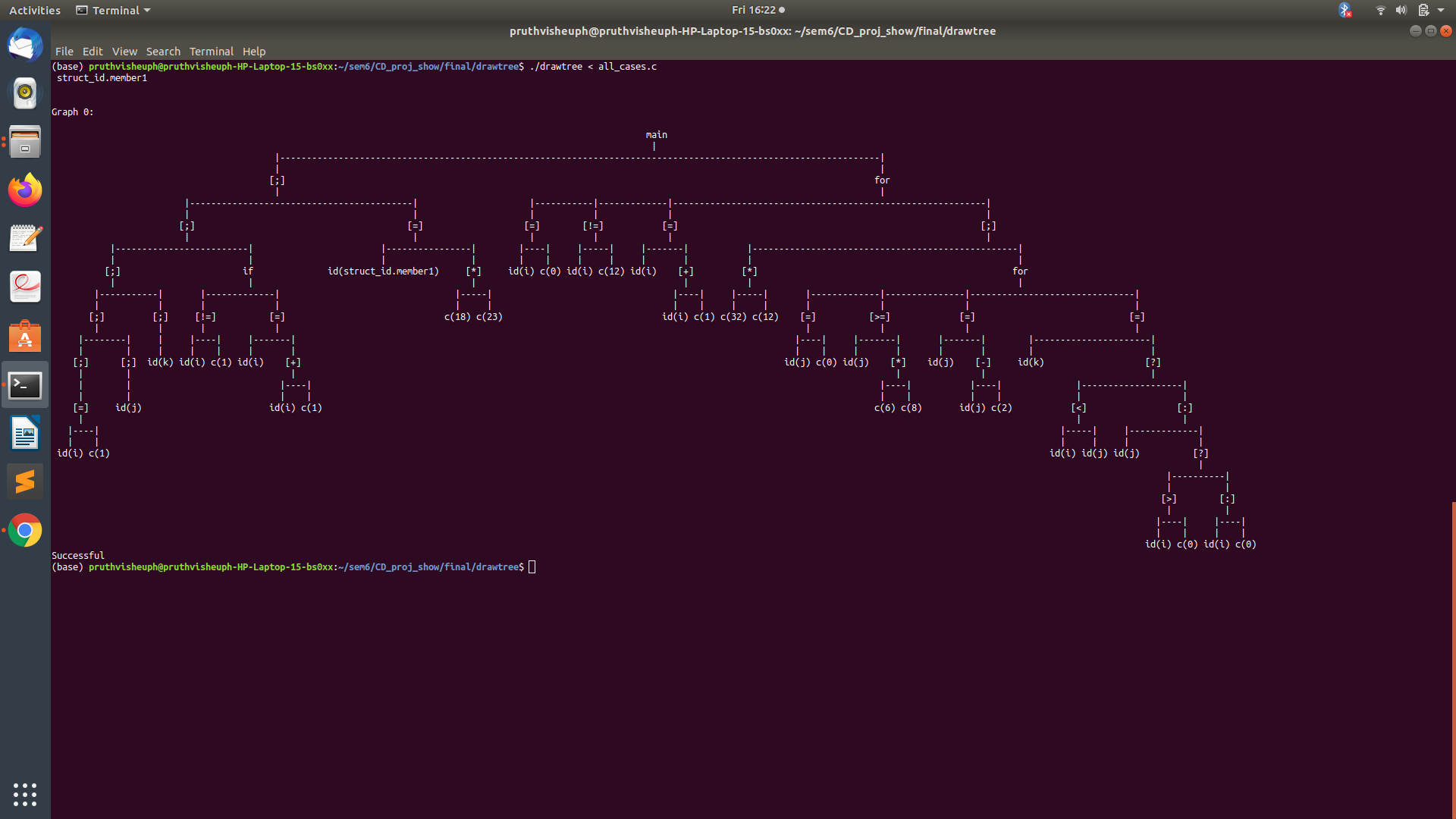
1. TestFile.c



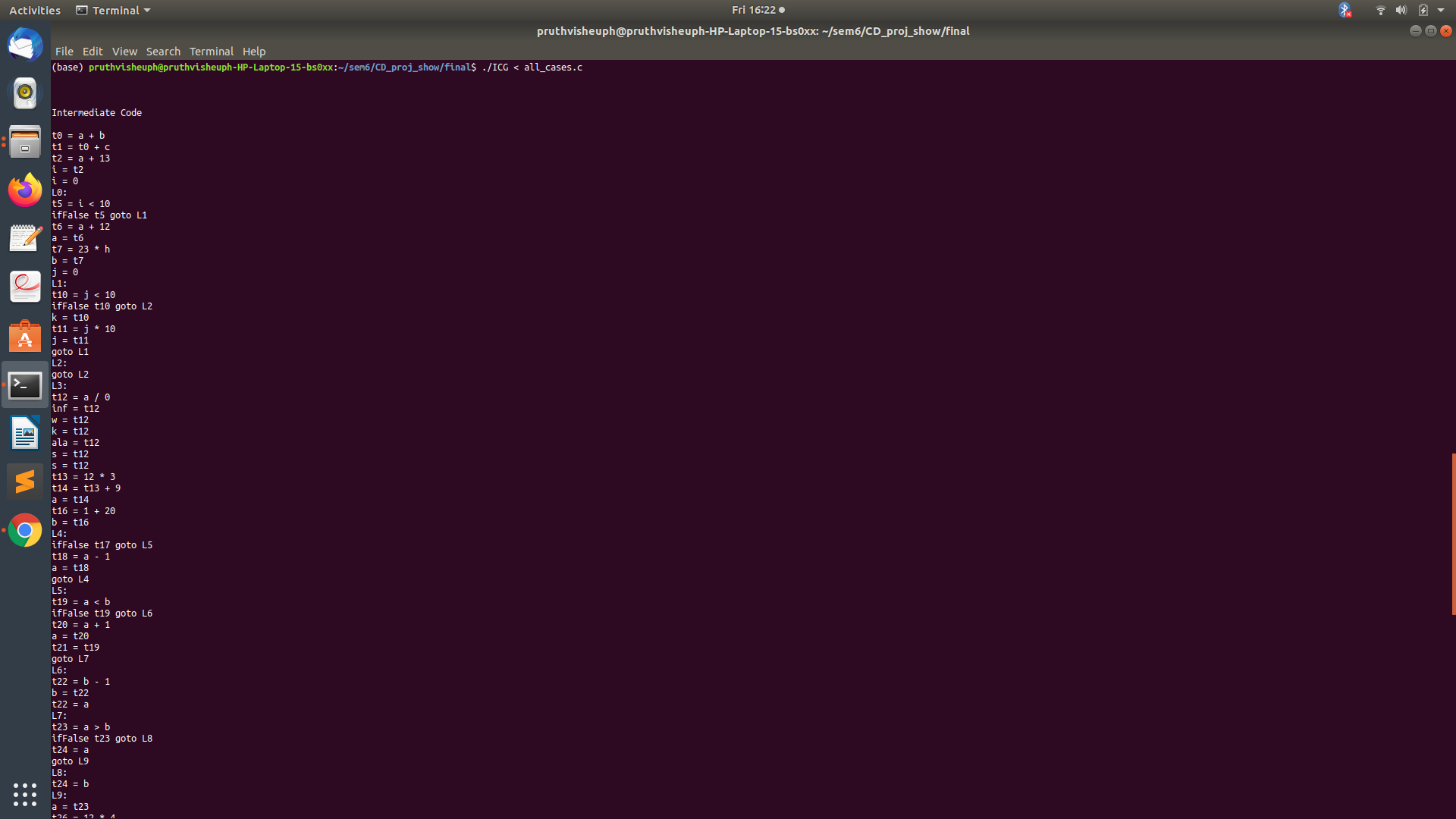
1. Symbol table and error handling

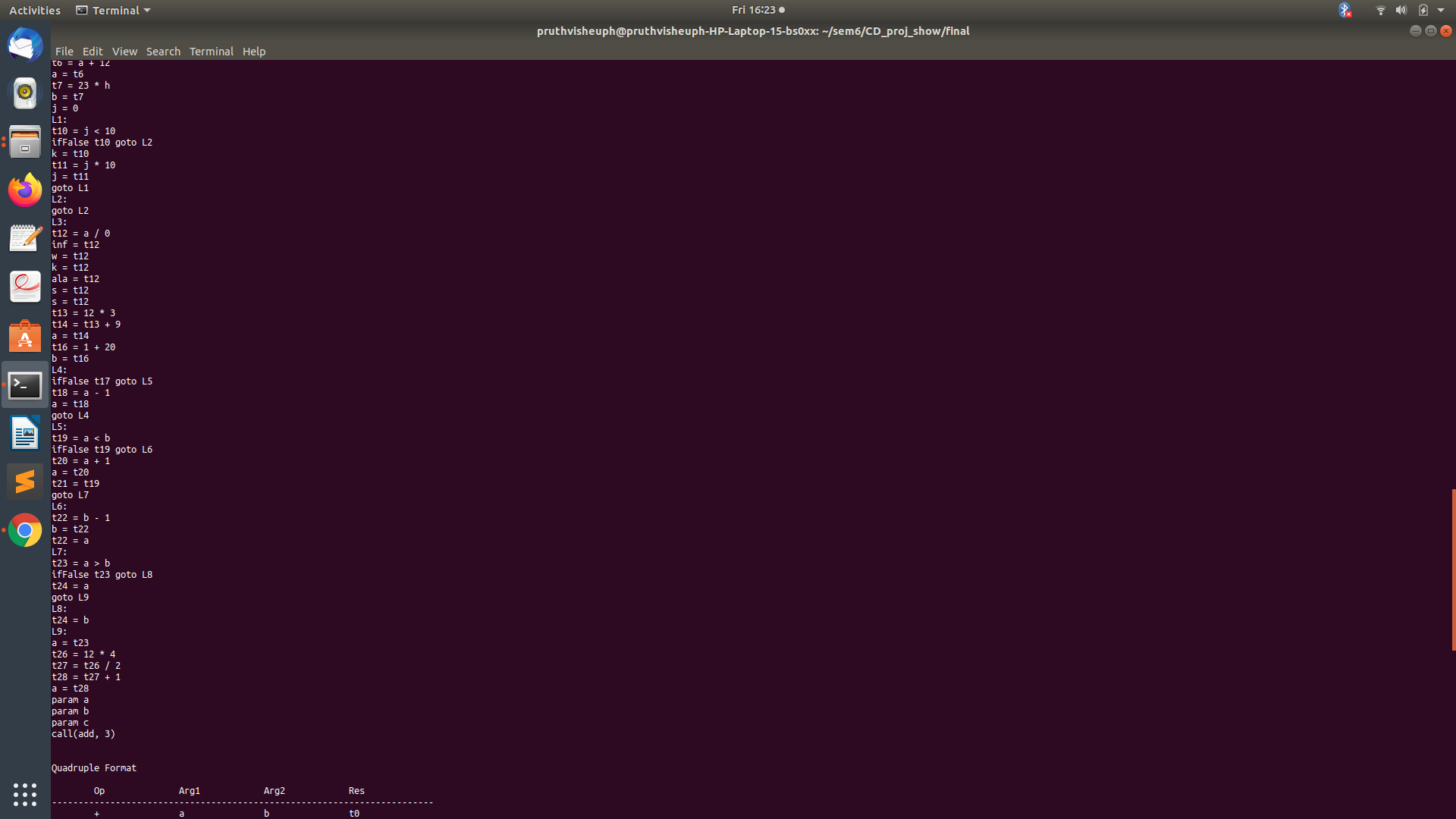


1. AST

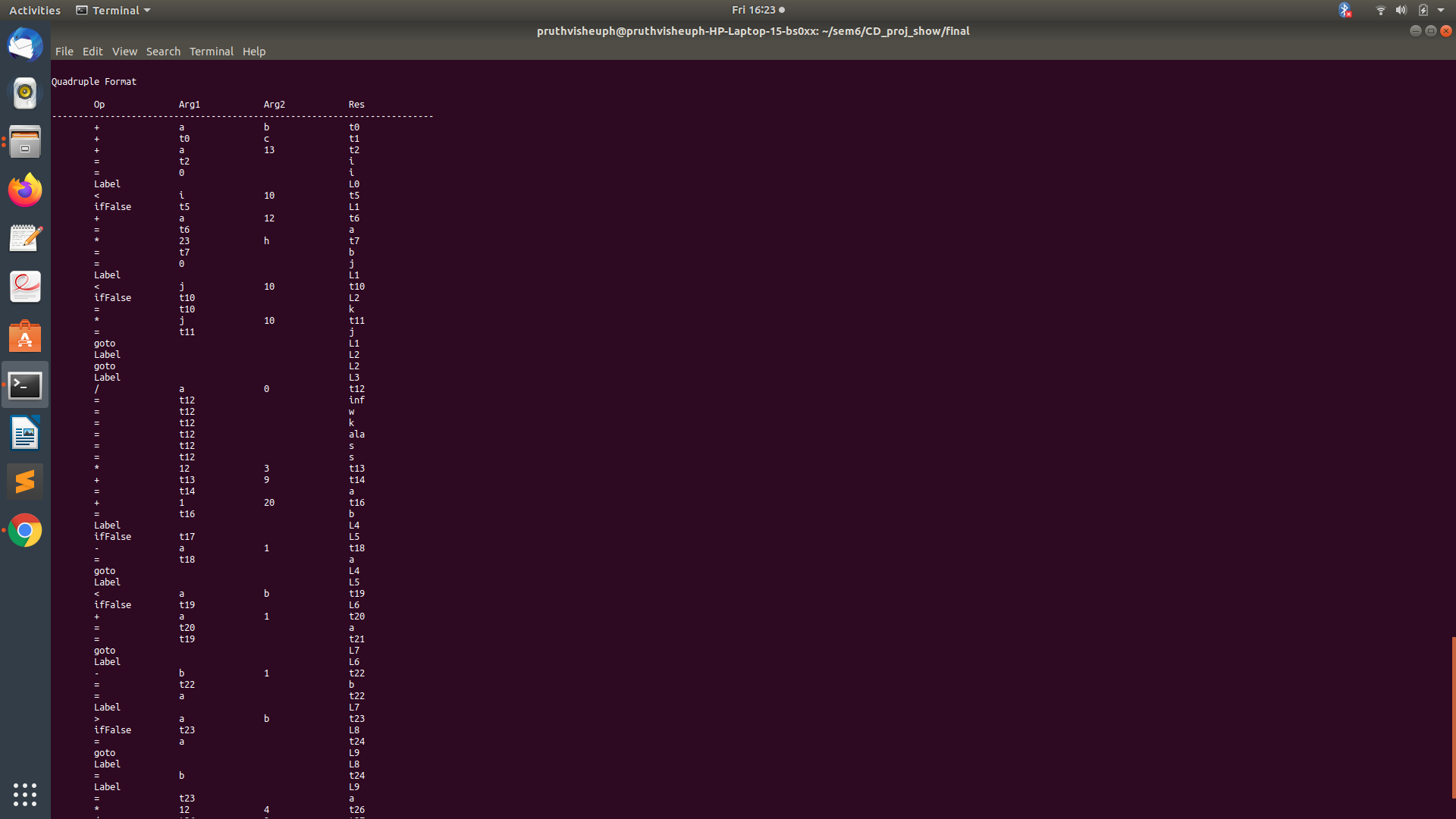


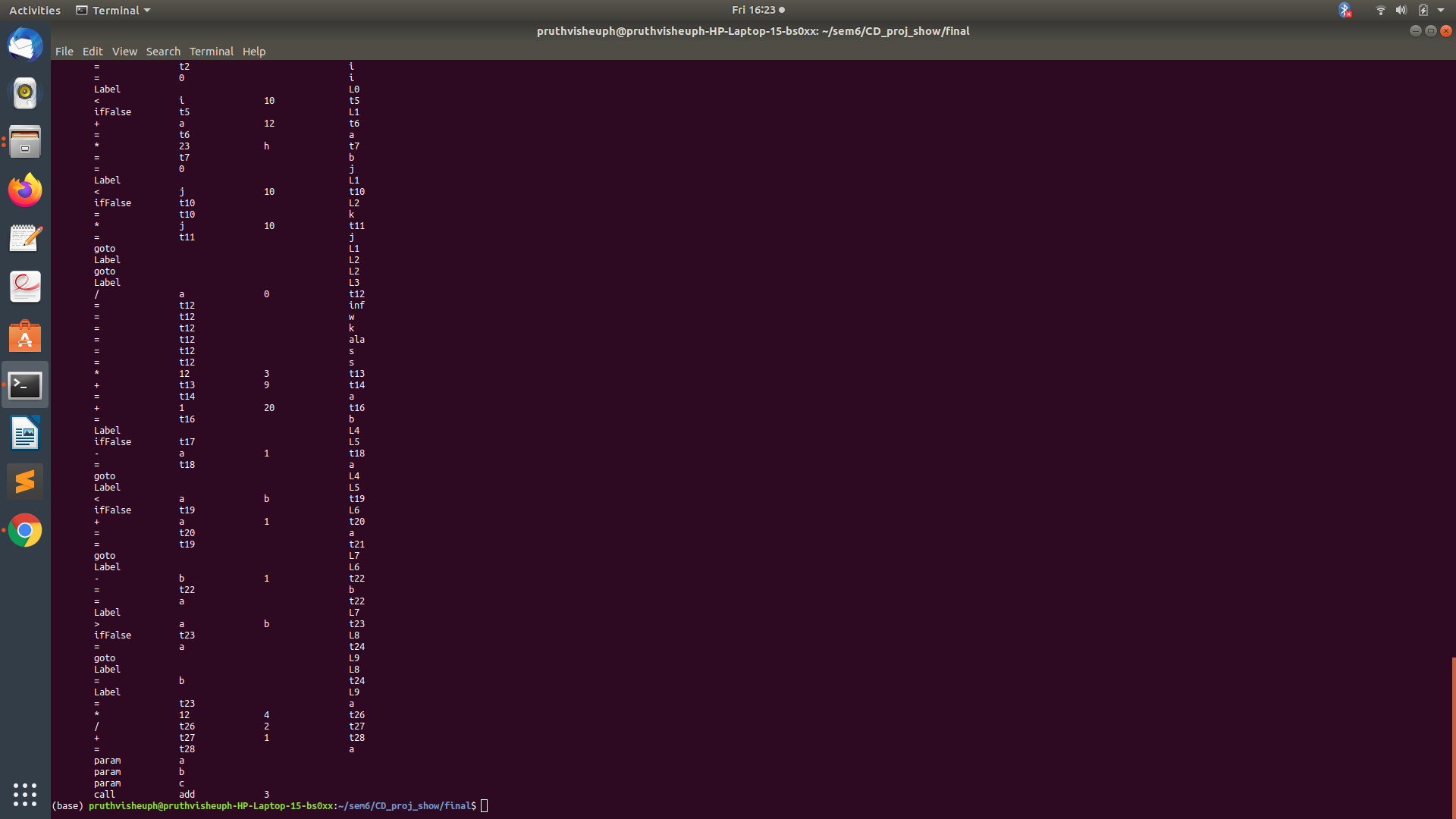
4.ICG Three-address-code



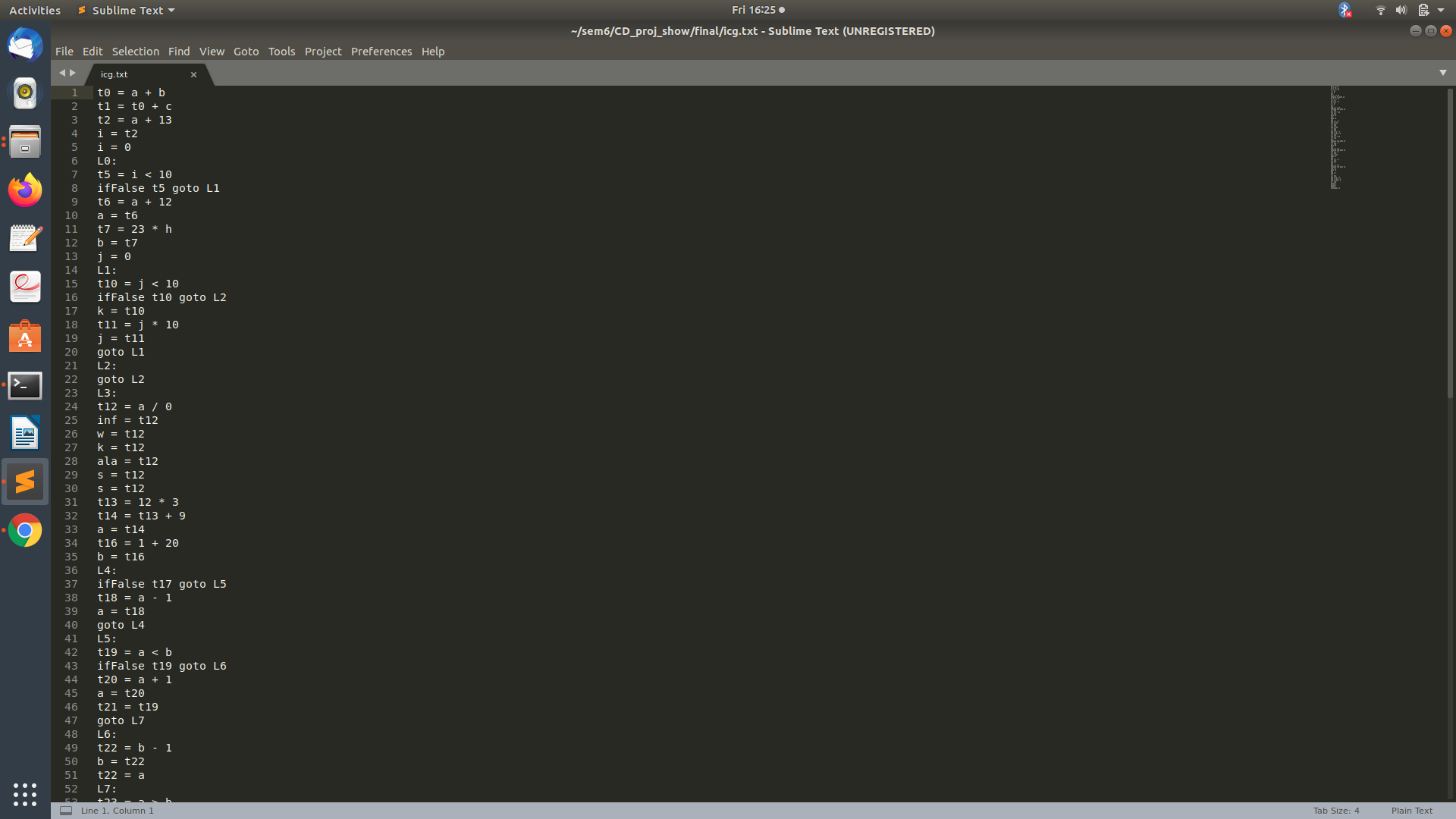


5.ICG quadruple format

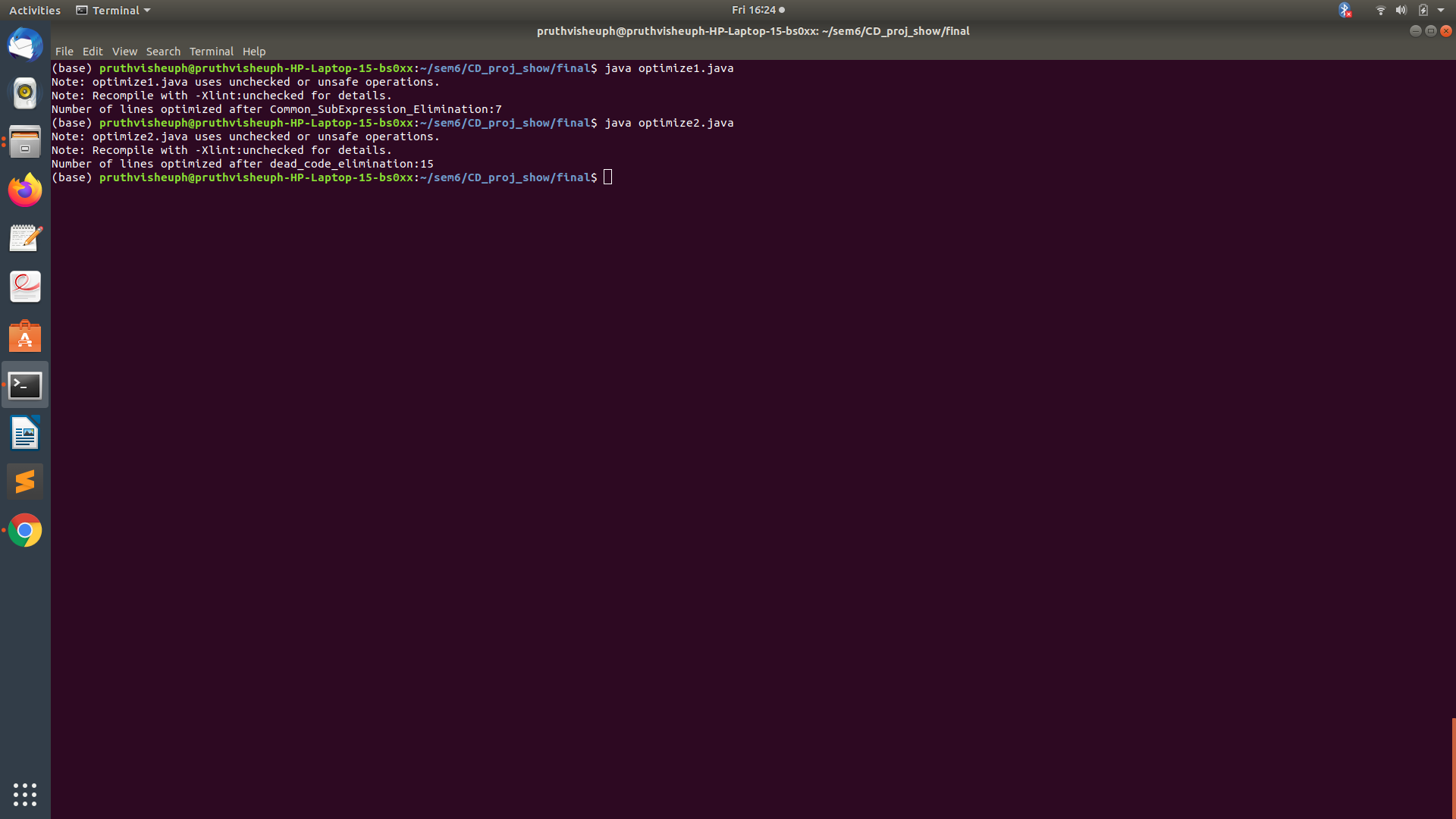




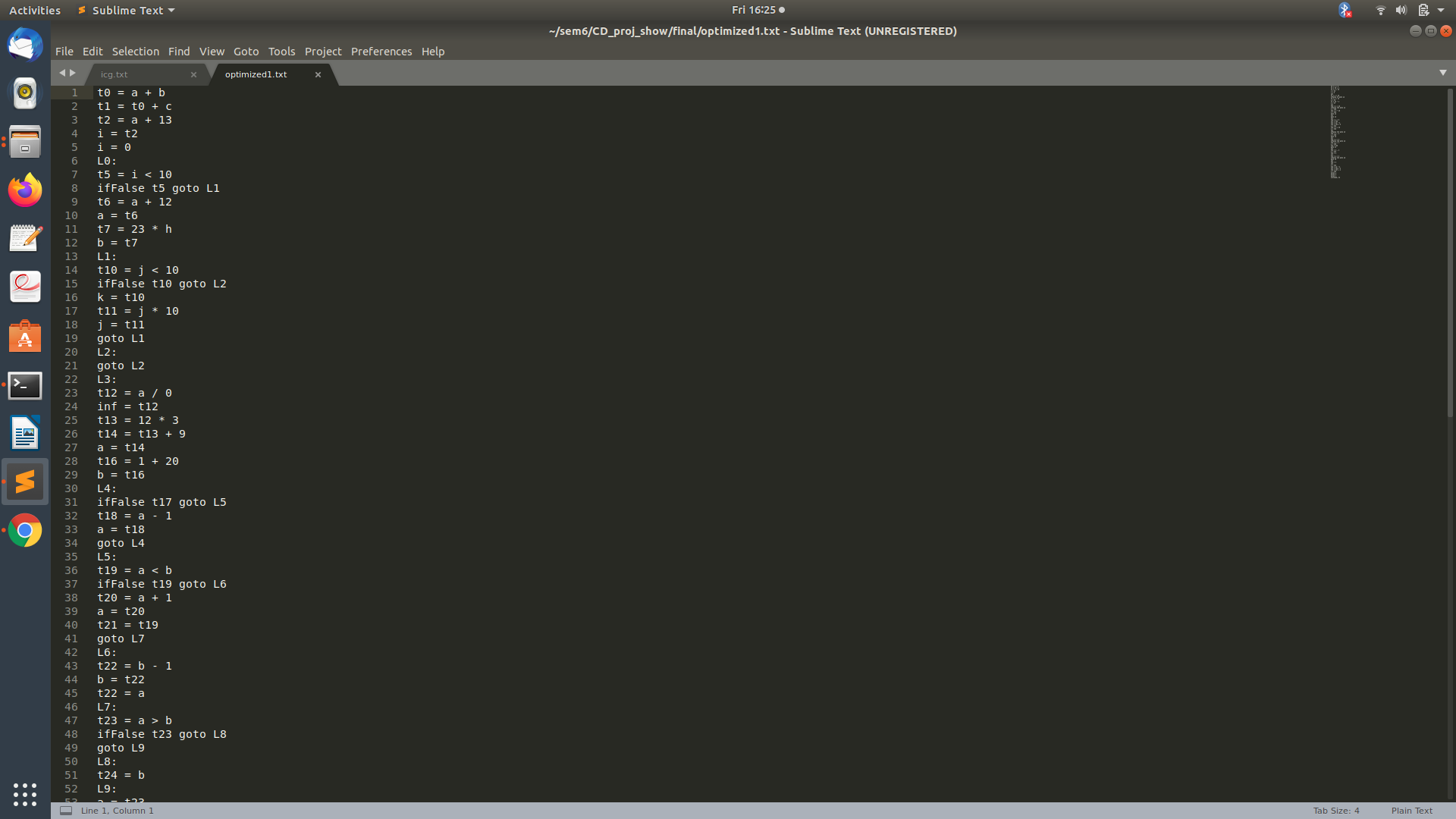
6. ICG Before optimization



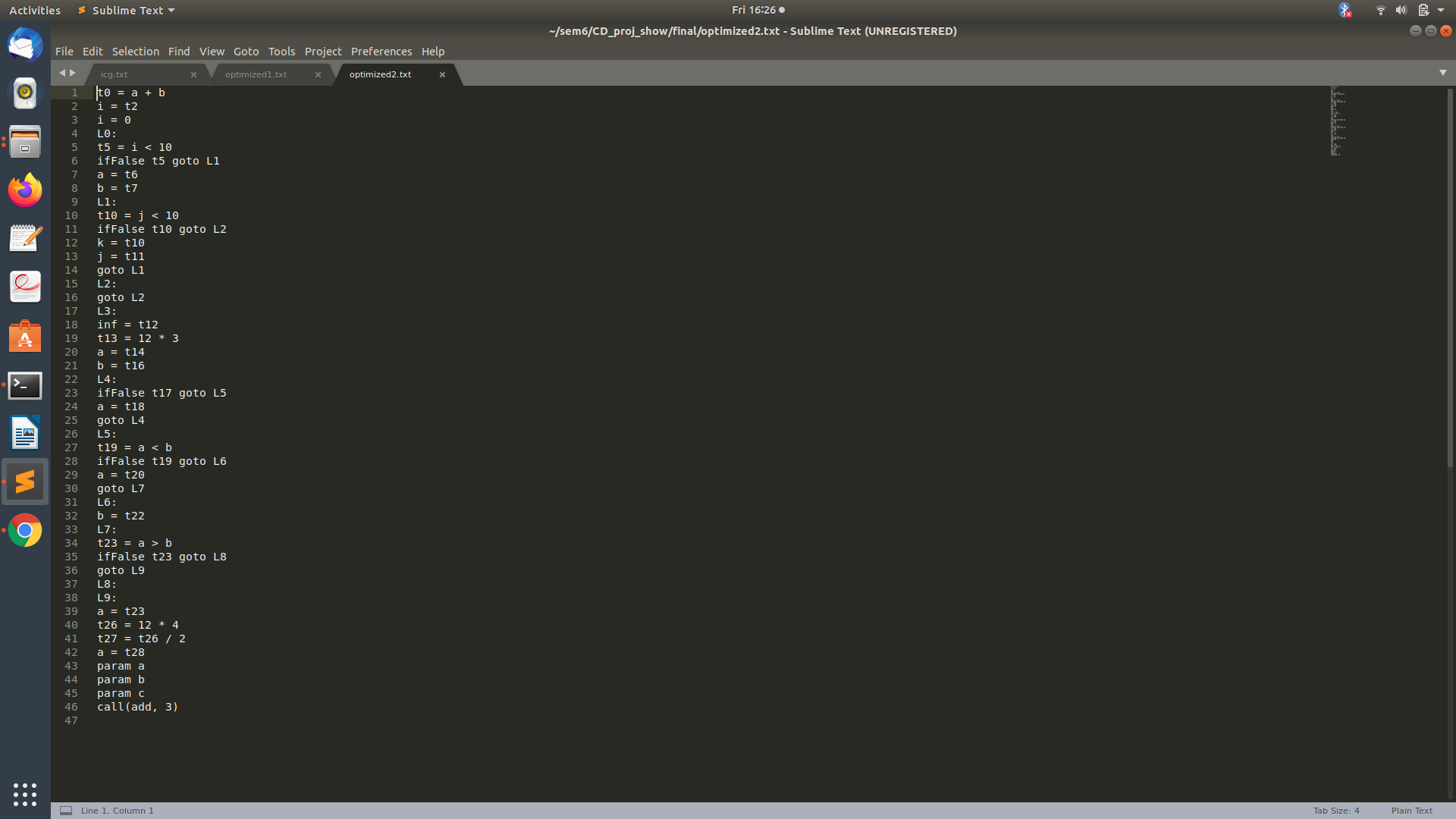
7.optimization procedure



8. After Common\_subexpression\_elimination



9. After Dead Code elimination



# 

# 

# 

# 

# CHAPTER 7. RESULTS AND CONCLUSION

The lex and yacc codes are compiled and executed by the following terminal commands to parse the given input file.

lex ast.l

yacc -d ast.y

gcc lex.yy.c y.tab.c -ll -ly -o AST

lex icg.l

yacc -d icg.y

gcc lex.yy.c y.tab.c -ll -ly -o ICG

./AST <test.c

./ICG <test.c

java optimize1.java

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 complete’ message is displayed on the console. The symbol table with stored & 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.

# 

# 

# CHAPTER 8. SHORTCOMINGS

Traversing the symbol table is time consuming as we have implemented a linked list, no random access possible.

# CHAPTER 9. FUTURE ENHANCEMENTS

1. Other looping constructs like do-while.

Conditional jumps like goto, continue and break.

Conditional statements like switch case.

1. More data types.

# REFERENCES

1. Compilers – Principles, Techniques, and Tools By Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
2. [https://www.tutorialspoint.com/compiler\_design/compiler\_design\_intermedia](https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_c) [te\_c ode\_generations.htm](https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_c)
3. https://www.youtube.com/watch?v=bc3ysHc5RH0