

***Report on***

**“Mini Compiler for While Constructs in Python”**

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

***Compiler Design Laboratory***

**Bachelor of Technology**

**in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **Manjunath R Hegde**  **Tejus D**  **Mukthi Prada.G.M** | **PES2201800110**  **PES2201800148**  **PES2201800310** |

*Under the guidance of*

|  |
| --- |
| **Prof. Swati S. Gambhire**  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)

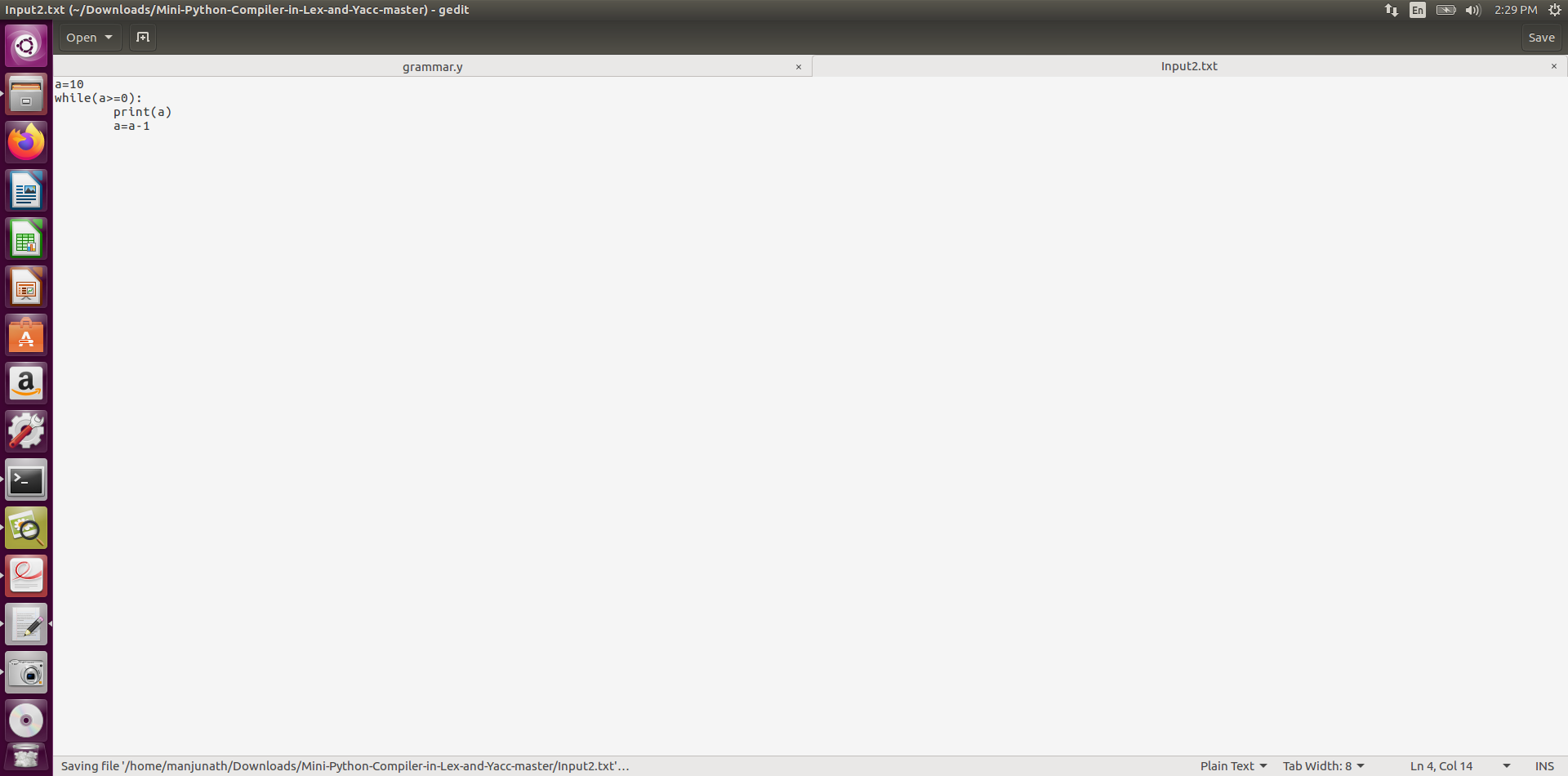
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Title** | **Page No.** |
|  | **INTRODUCTION** | **03** |
|  | **ARCHITECTURE OF LANGUAGE:** | **08** |
|  | **LITERATURE SURVEY** | **08** |
|  | **CONTEXT FREE GRAMMAR** | **09** |
|  | **DESIGN STRATEGY**   * **SYMBOL TABLE CREATION** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** | **10** |
|  | **IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following):**   * **SYMBOL TABLE CREATION** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** * **Instructions for Execution** | **12** |
|  | **RESULTS AND POSSIBLE SHORTCOMINGS** | **15** |
|  | **SNAPSHOTS** | **16** |
|  | **CONCLUSIONS** | **30** |
|  | **FURTHER ENHANCEMENTS** | **30** |

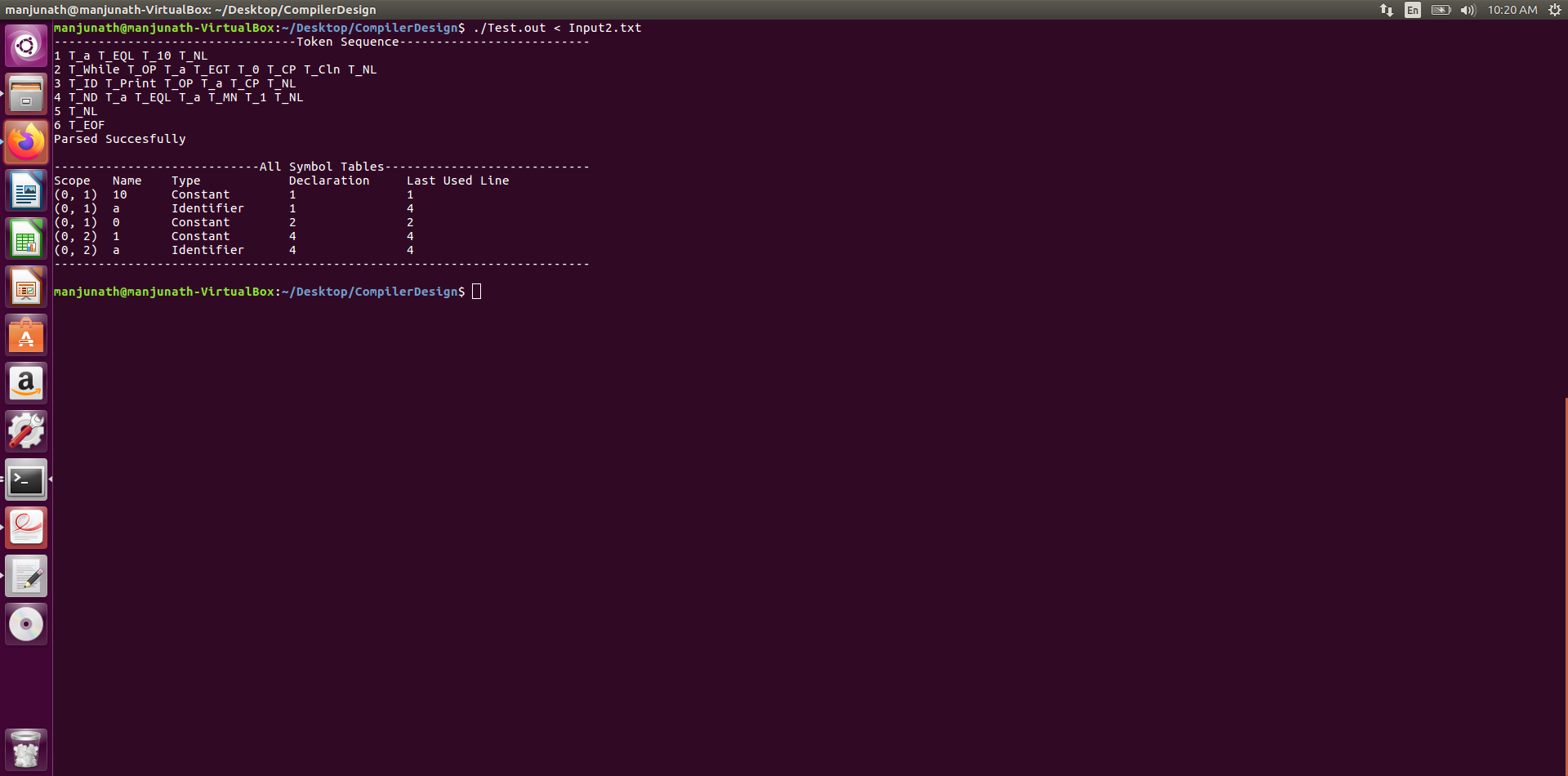
**INTRODUCTION**

This Mini Compiler is Built for Python and handles the If-Elif-Else and the While Constructs.

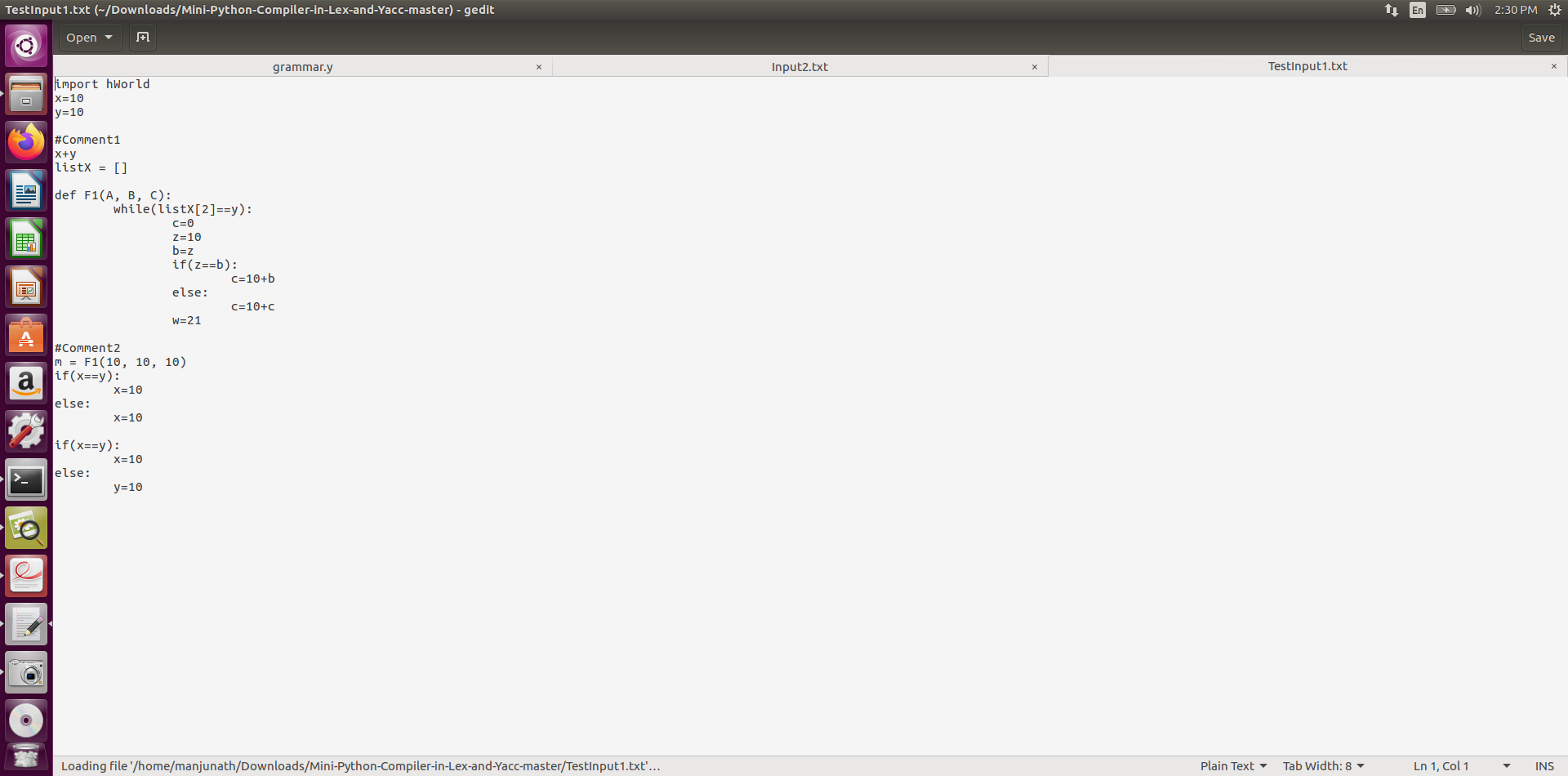


**Sample Input 1:**

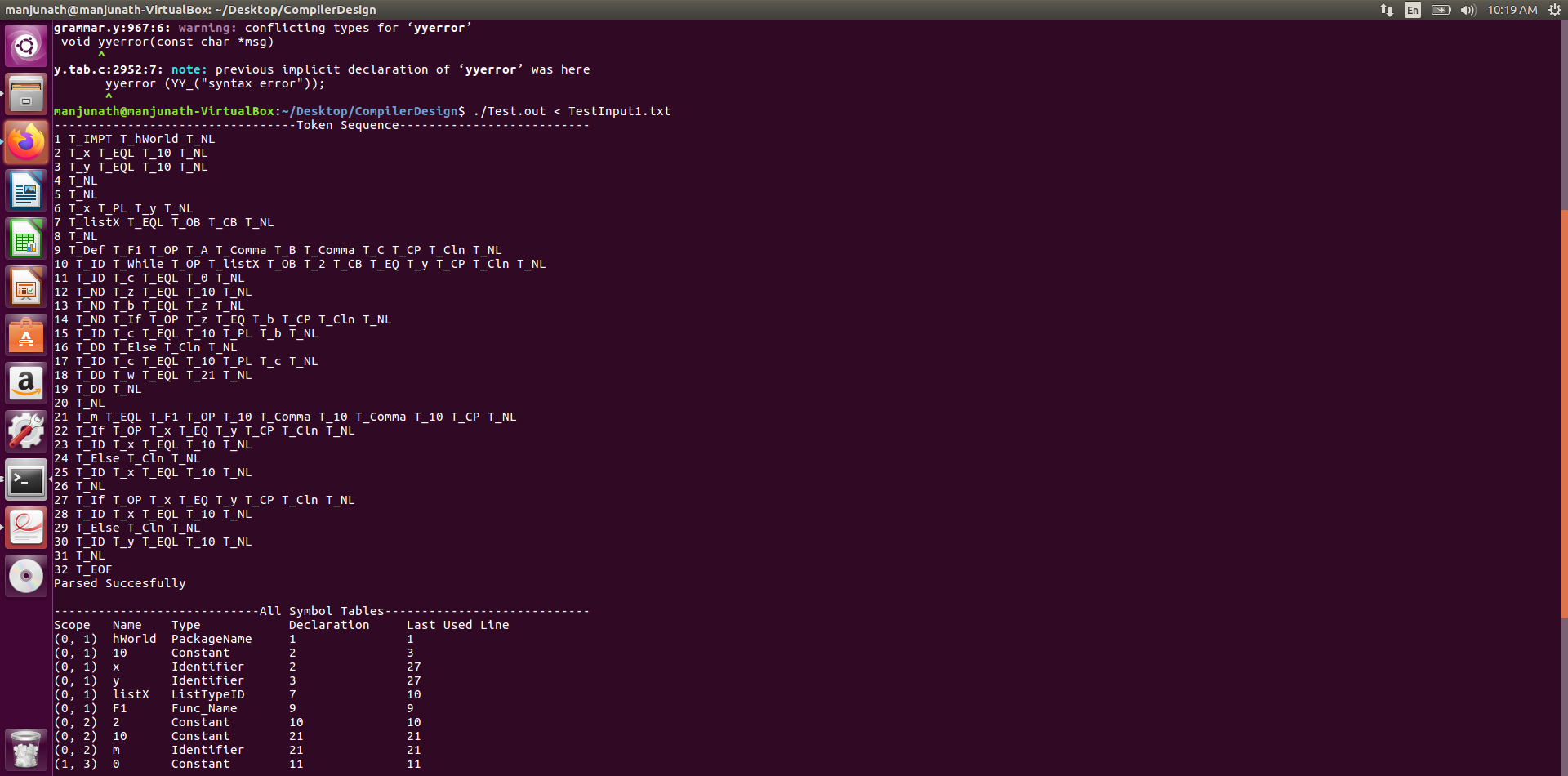
**Sample Output:**

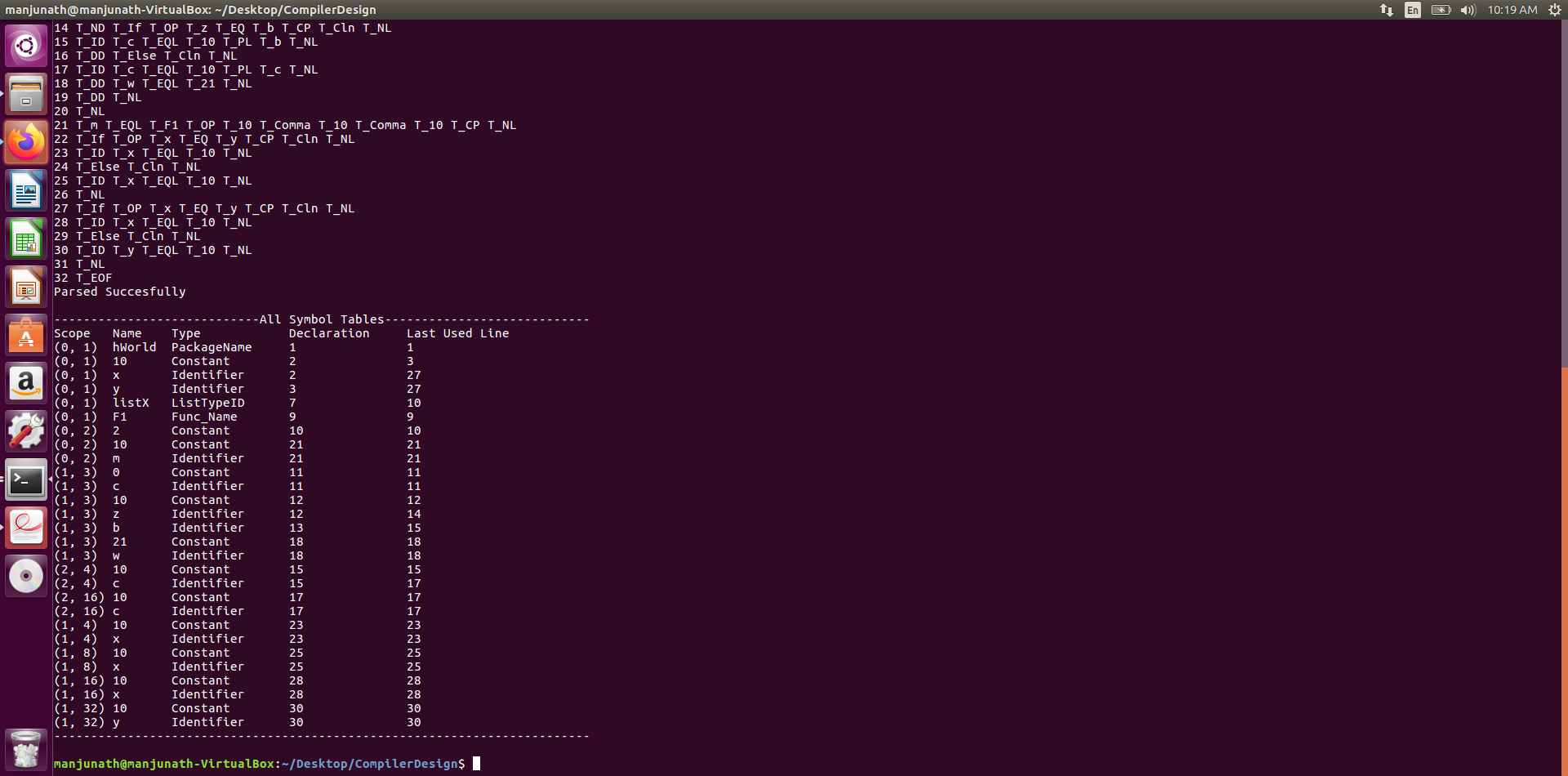


**Sample Input 2:**

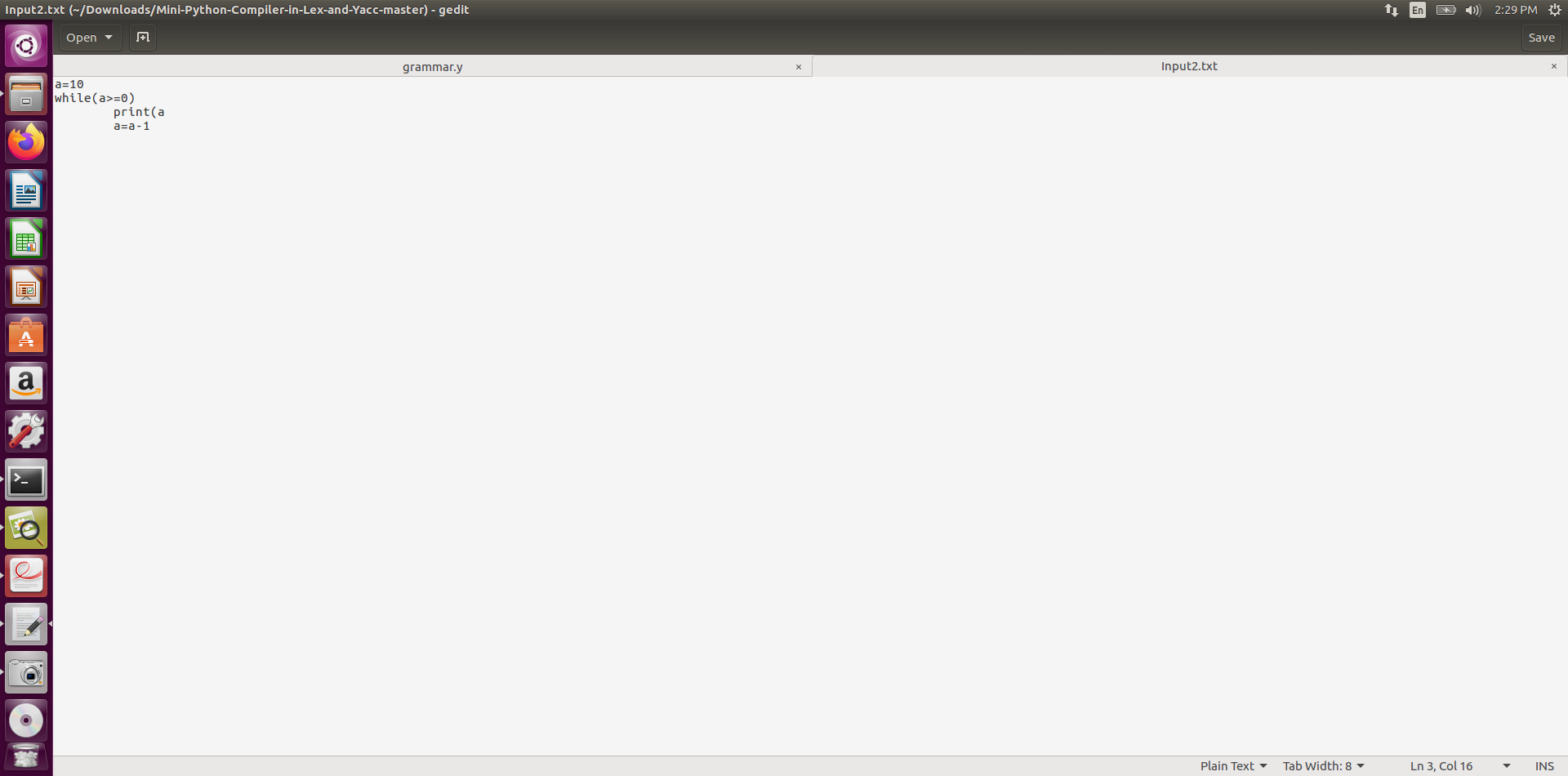


**Sample Output:**

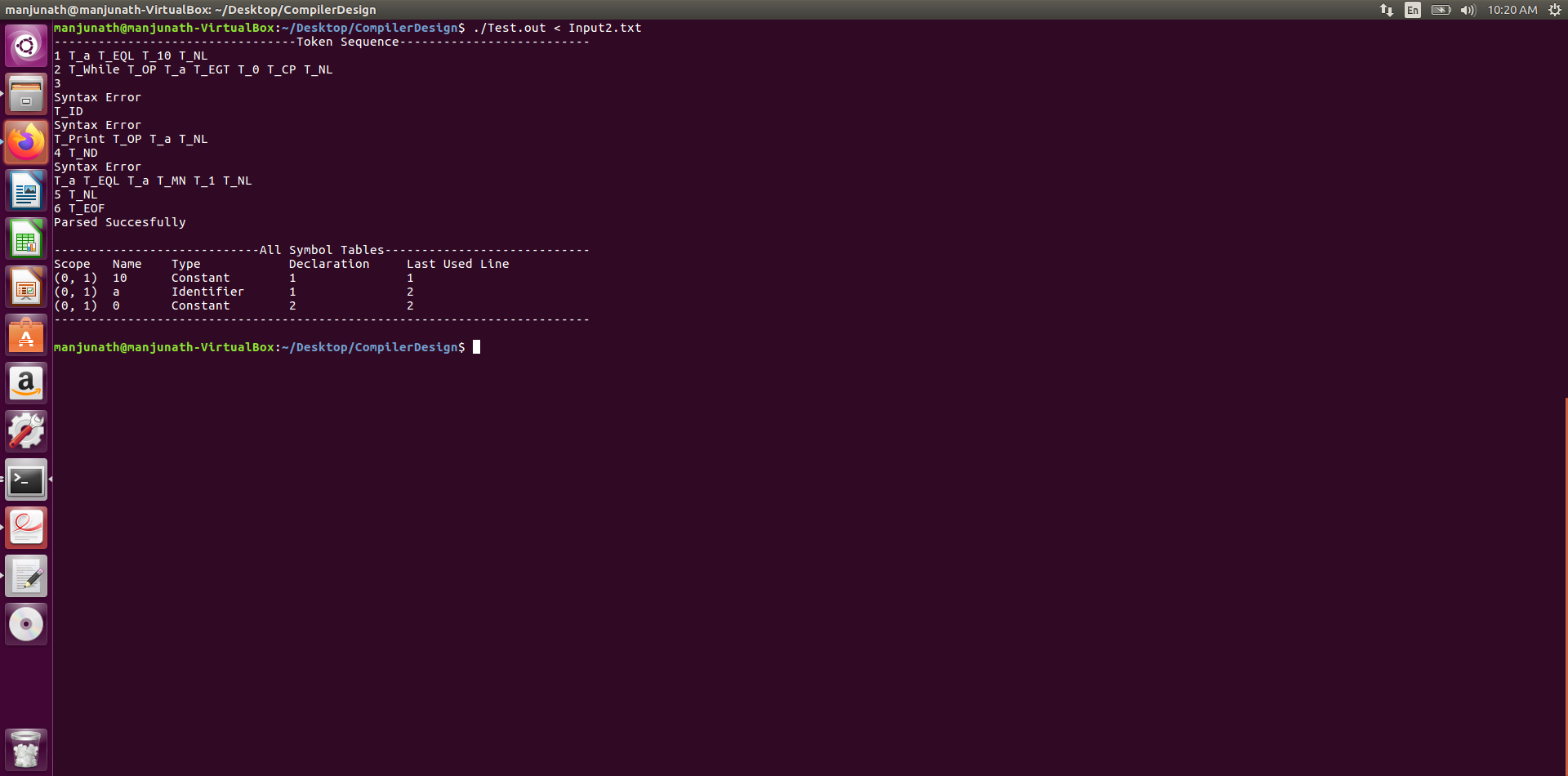




**Sample Input 3 :**

****

**Sample output :**

****

**ARCHITECTURE OF THE LANGUAGE**

The term compilation denotes the conversion of an algorithm expressed in a human-oriented source language to an equivalent algorithm expressed in a hardware-oriented target language. As we all know Python has a very flexible syntax and hence we have tried as much as possible to incorporate python into the grammar.

In this process of incorporating python into the grammar there are few things we could not take care of like: semicolons, so a semicolon will result in an error while parsing. All lines of code terminate upon seeing a newline character.

These are function we have implemented:

● If-Elif-Else and While constructs

● Print Statements

● pass, break and void returns

● Function definitions and Calls

● Lists

● All arithmetic operators and all boolean operators except standalone ‘!’ (“!=” is taken care of)

● Single Line Comments (#)

**Semantically we have checked the following :**

● Whether any variable used on the RHS is defined and in the current scope or any Enclosing Scope of the current scope.

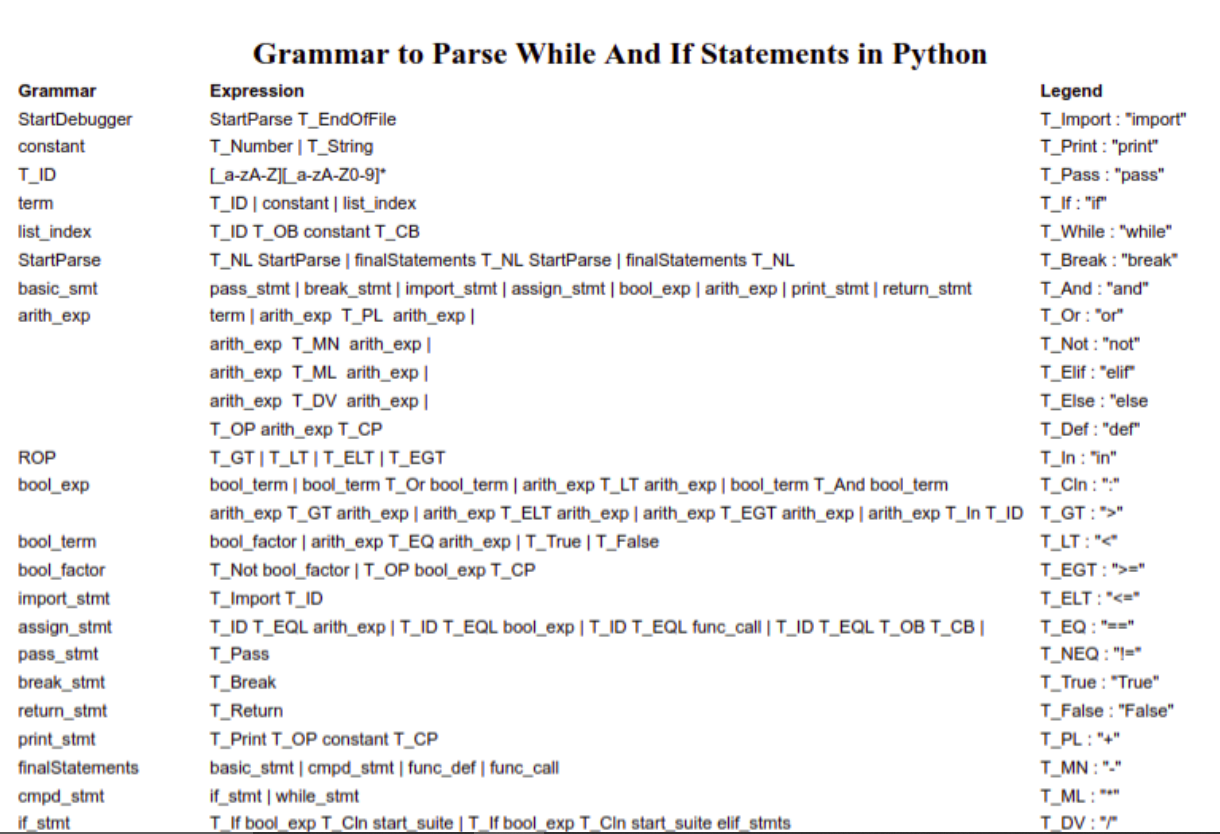
● Whether a variable being indexed is List

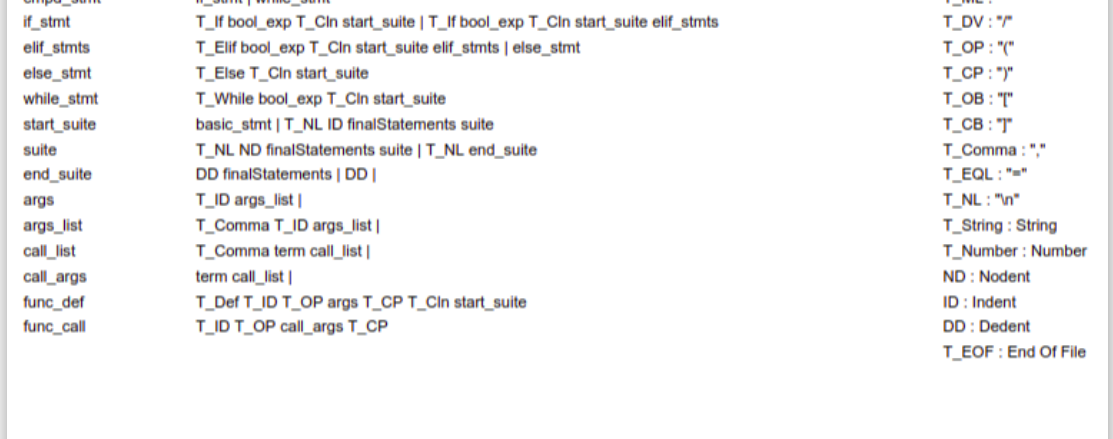
● Whether all expressions in the If and While Clauses are Boolean expressions

**LITERATURE SURVEY**

1. Lex and Yacc Doc by Tom Niemann
2. Official Bison Documentation : <https://www.gnu.org/software/bison/manual/>
3. Stackoverflow : <https://stackoverflow.com/>

**THE CONTEXT FREE GRAMMAR**





**DESIGN STRATEGY**

The Design Strategy we've got carried out is that firstly, each hyperlink is going again to the Symbol Table. This way that the desired nodes of the Abstract Syntax tree and the desired Quadruples within the Intermediate code have a hyperlink to the Symbol table.

All the compiler generated Temporaries also are saved within the image table.

The Symbol Table stores ‘Records’ having four columns as you could see withinside the pattern output, ie,

❖ Scope : Scope of every variable contained in record

❖ Name : Value/Name of every variable contained in record

❖ Type : Type of every variable contained in record

➢ PackageName

➢ Func\_Name

➢ Identifier

➢ Constant

➢ ListTypeID

➢ ICGTempVar

➢ ICGTempLabel

❖ Declaration : Line of Declaration of every variable contained in record

❖ Last Use Line

The scope is a feature of Indentation intensity and to make it particular we've a tuple of the scope of the parent and the current scope calculated the use of the Indentation intensity.

For the Abstract Syntax Tree, We have 2 Types of Nodes, Leaf nodes and Internal nodes. The nodes could have a variable variety of children (0-3) relying upon the assembly it represents. Take the instance of the If-Else Statement,

If

Condition CodeBlock Else

In order to show the AST, we use the AST to save it as a Level Matrix. We have printed each AST level, as we can see in the sample output. Each internal node also has a number attached to it, which represents the number of children it has at the next level. Leaf nodes in the AST showing identifiers, constants, lists and packages are shown in the table of symbols.

The intermediate code is created by reciprocating the AST. Each line is stored four-fold in order to easily optimise code in the following steps (Operation, Arg1, Arg2, Result).

We eliminate dead code in the entire programme, especially unused variables. If, for instance, we have the following code lines:

a = 10

b = 10

c = a + b

And the 3 variables are not applied to any other RHS, then during optimisation these 3 lines of code are removed. The code removes all dead variables.

We're doing this in the Quads.

We have implemented a stack and 3 tokens to manage the indentation-driven code structure and scoping. If the value does not change when the next line is scanned, the stack top always points to the current indentation value, imply that we are within the same range, and therefore we give the token "ND," i.e., "No-dent." Increasing the value means that we enter a sub-scope with the token 'ID,' that is, 'Indent,' and if this decreases, it means that we will come back to one of the containing scopes and return the token ' DD,' that is 'Dedent.'

Before parsing, all comments are removed from the code

**IMPLEMENTATION DETAILS**

The Symbol table uses two Structures,

typedef struct record

{

char \*type;

char \*name;

int decLineNo;

int lastUseLine;

} record;

typedef struct STable

{

int no;

int noOfElems;

int scope;

record \*Elements;

int Parent;

} STable;

Each record in the symbol table is a 'record' structure. A maximum of "MAXRECST" records may be contained in each symbol table. MAXRECST is a macro.

The structure of the 'STable' is a table of symbols. For each scope a new symbol table is created. MAXST is a macro, and therefore it can contain a maximum of "MAXST" tables and scopes.

The Abstract Syntax Tree uses one structure,

typedef struct ASTNode

{

int nodeNo;

/\*Operator\*/

char \*NType;

int noOps;

struct ASTNode\*\* NextLevel;

/\*Identifier or Const\*/

record \*id;

} node;

This structure of ASTNode is designed for both the internal and the leaf nodes

Nodes "Operator." Depending on the node type, the values are set. There may be 0-3 children at each node. We print the AST in a "MAXLEVELS" x "MAXCHILDREN" matrix, and print the Level-wise matrix. The matrix is an AST-pointer matrix. The node element "noOps" indicates the number of kids in that node.

The Three-Address Code is represented and stored as Quads that are given by the Structure,

typedef struct Quad

{

char \*R;

char \*A1;

char \*A2;

char \*Op;

int I;

} Quad;

During code optimisation, the last element, the 'I' integer is used. All three address codes in the Quads array are stored as Quads. Maximum quadruples may be available "MAX QUADS."

The scope check is done by reciprocating the scope and finding the latest variable definition. We print the error if no definition is found.

Finally, we provided a makefile to compile the code and execute the code.

If you wish to only s

ee the AST,

lex grammar.l

yacc -dv grammarAST.y

gcc lex.yy.c y.tab.c -g -ll -o TestAST.out

./TestAST.out < InputFile.txt

If you wish to only see the Intermediate code and Optimization

lex grammar.l

yacc -dv grammarICG.y

gcc lex.yy.c y.tab.c -g -ll -o TestICG.out

./TestICG.out < InputFile.txt

The makefile provided compiles the “grammar.y” file and prints everything.

./Test.out < Input.txt

**ERROR HANDLING:**

1. We used panic mode recovery strategy for syntax error handling.
2. it will continue parsing if there is any syntax error.
3. Whenever an error occurs in the parser, it prints the error column no and line no. The following errors are also shown:

<var> Identify Not stated in scope

<var> Identify Not indexed

**CODE OPTIMIZATION:**

1. **Dead Code optimization:** We continuously loop through the code for the removal of dead code until code can no longer be deleted. To check whether a quadruple represents dead code we can see if any arguments for any subsequent quads which were not deleted appear to the result parameter / element of that quad, If not, the quad represents dead code and marks 'I' with the value '-1.'
2. **Constant propagation:** All the variables which refer to a constant value are substituted for every occurrence.

1. **Constant folding:** Expressions with constant operands can be evaluated at compile time thus improving run time performance and reducing code size by avoiding evaluation at compile time.

1. **Loop invariant Motion:** Here constant expressions which are inside the loop are moved outside the loop . instead of executing the expression n times we execute 1 time because the result is always the same.

**RESULTS AND SHORTCOMINGS**

As a result, we have a mini compiler which scans the grammar of a basic python syntax and ultimately generates an optimised intermediate image.

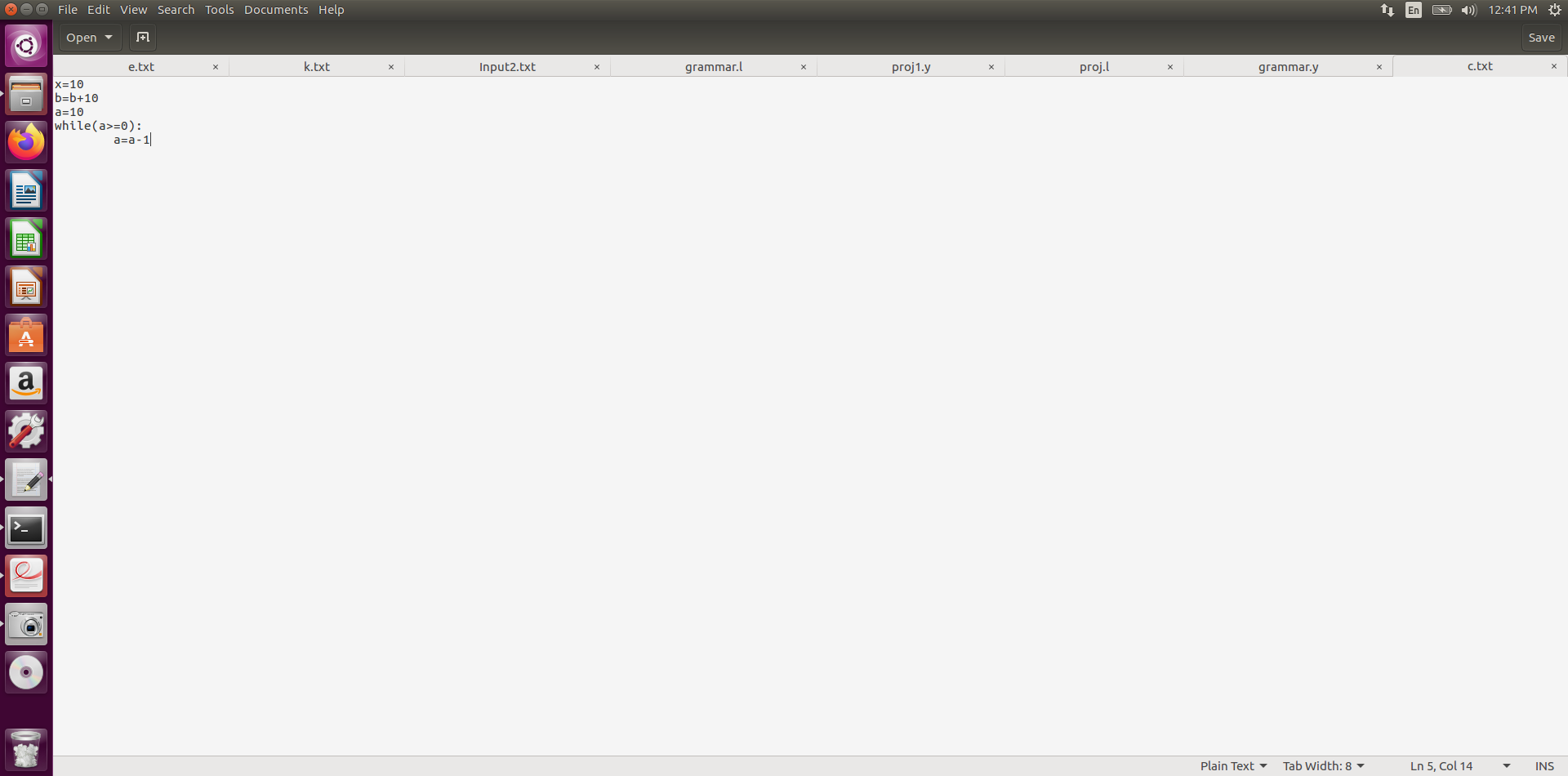
The areas where our mini compiler falls short are,

● Doesn’t handle semicolons.

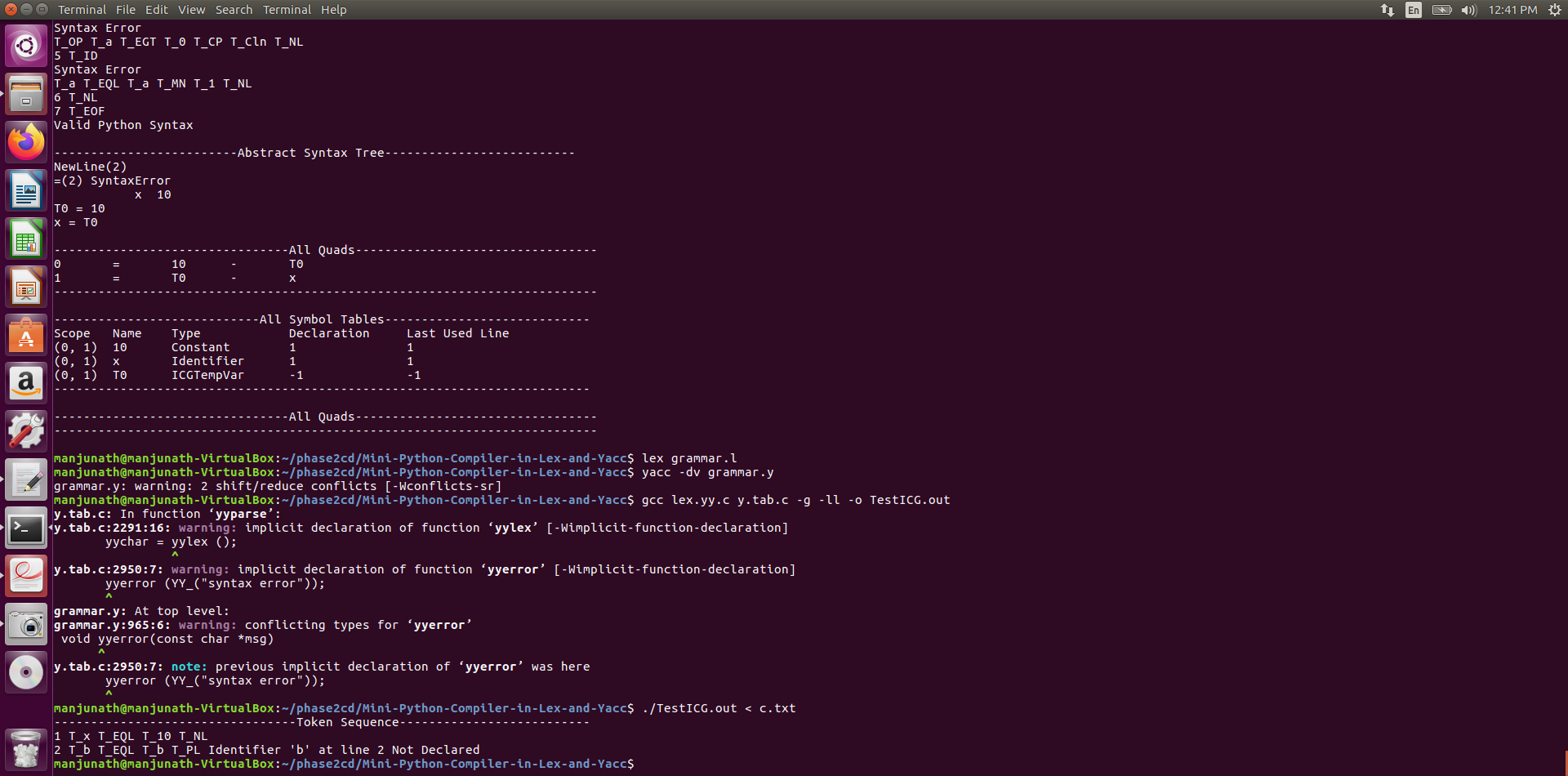
● Only one very basic optimization is implemented that does not reduce code density by a huge margin.

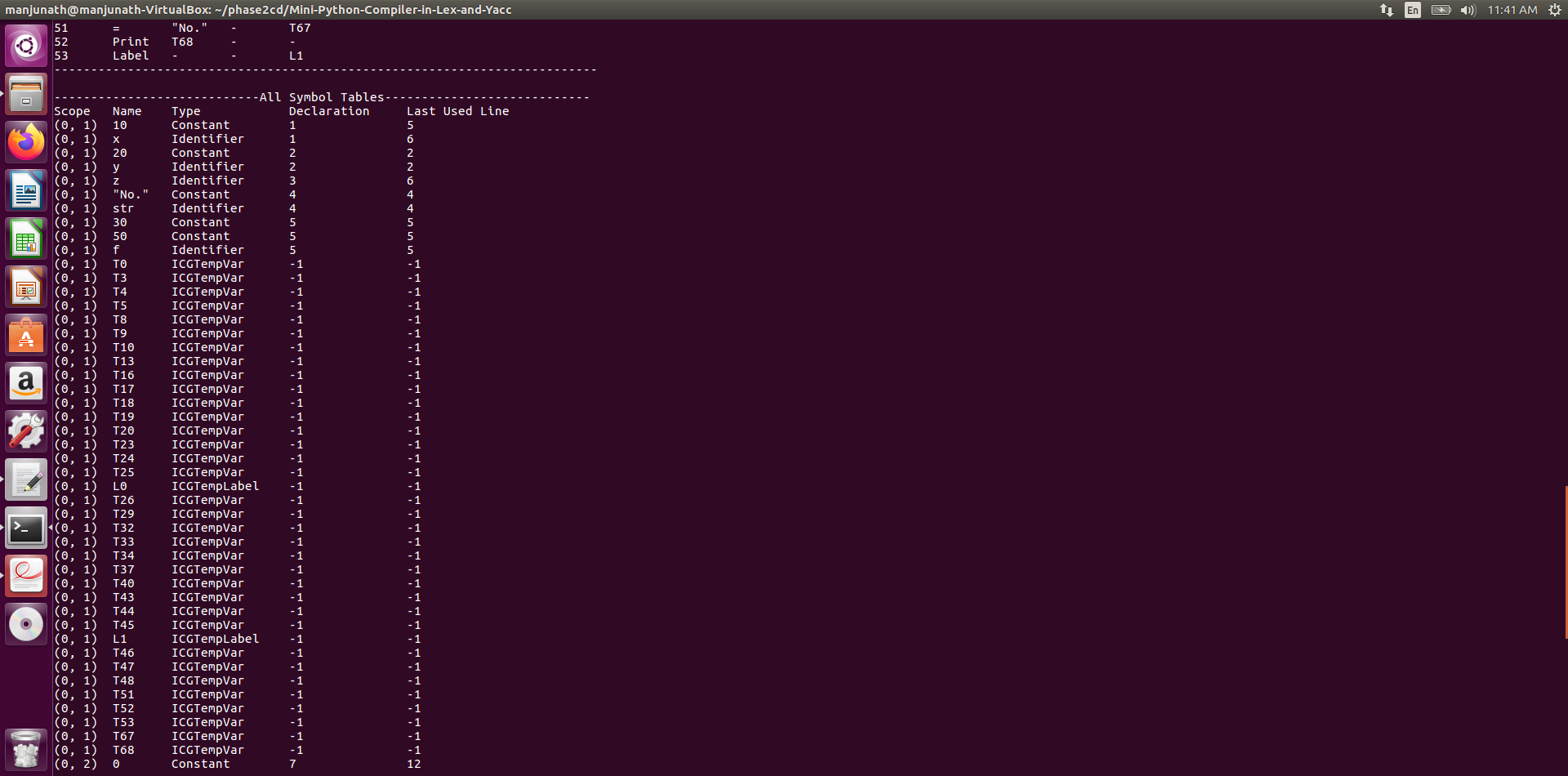
● The Program has a few memory leaks, although most of them have been taken care of.

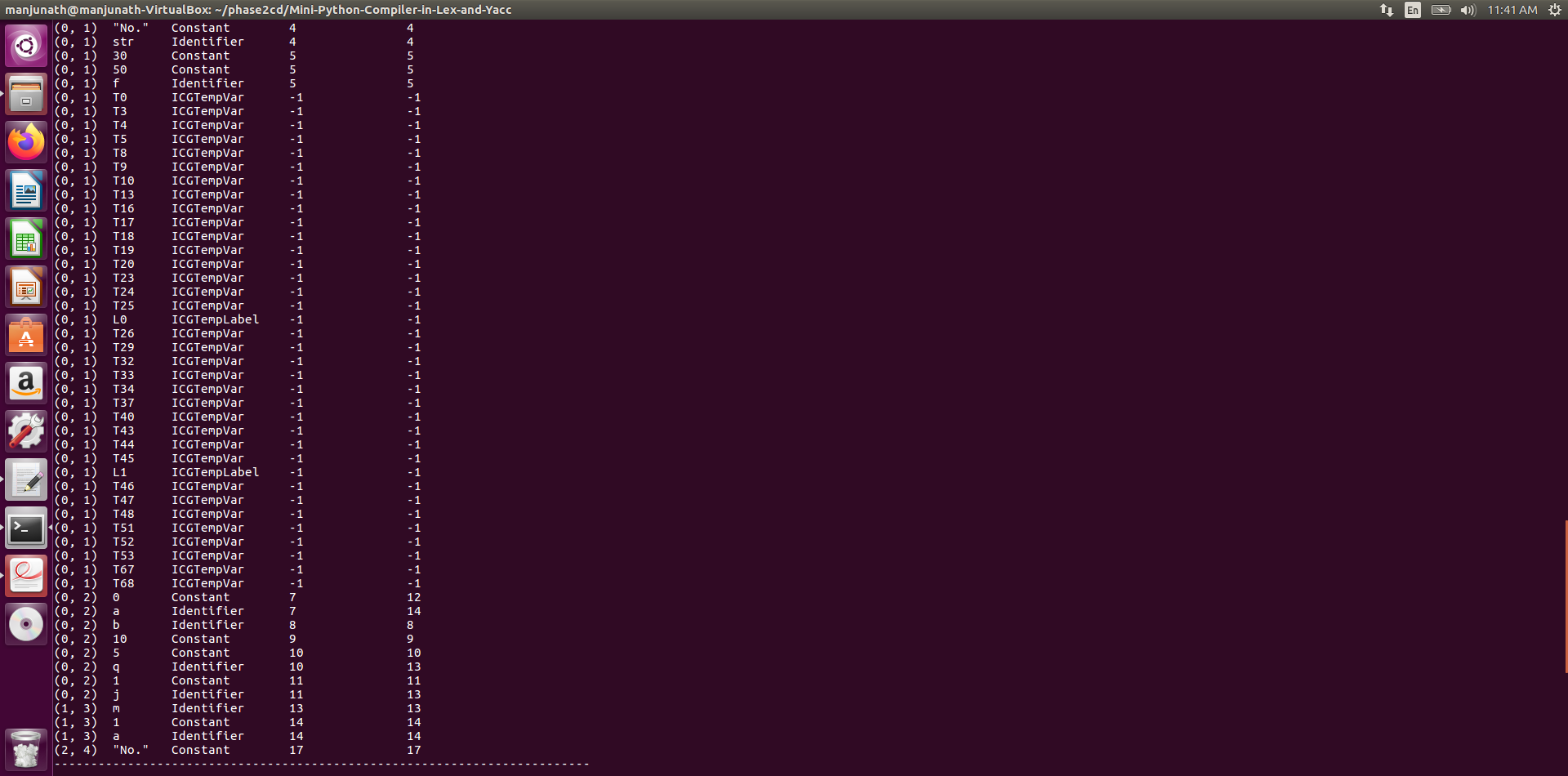
**SNAPSHOTS**



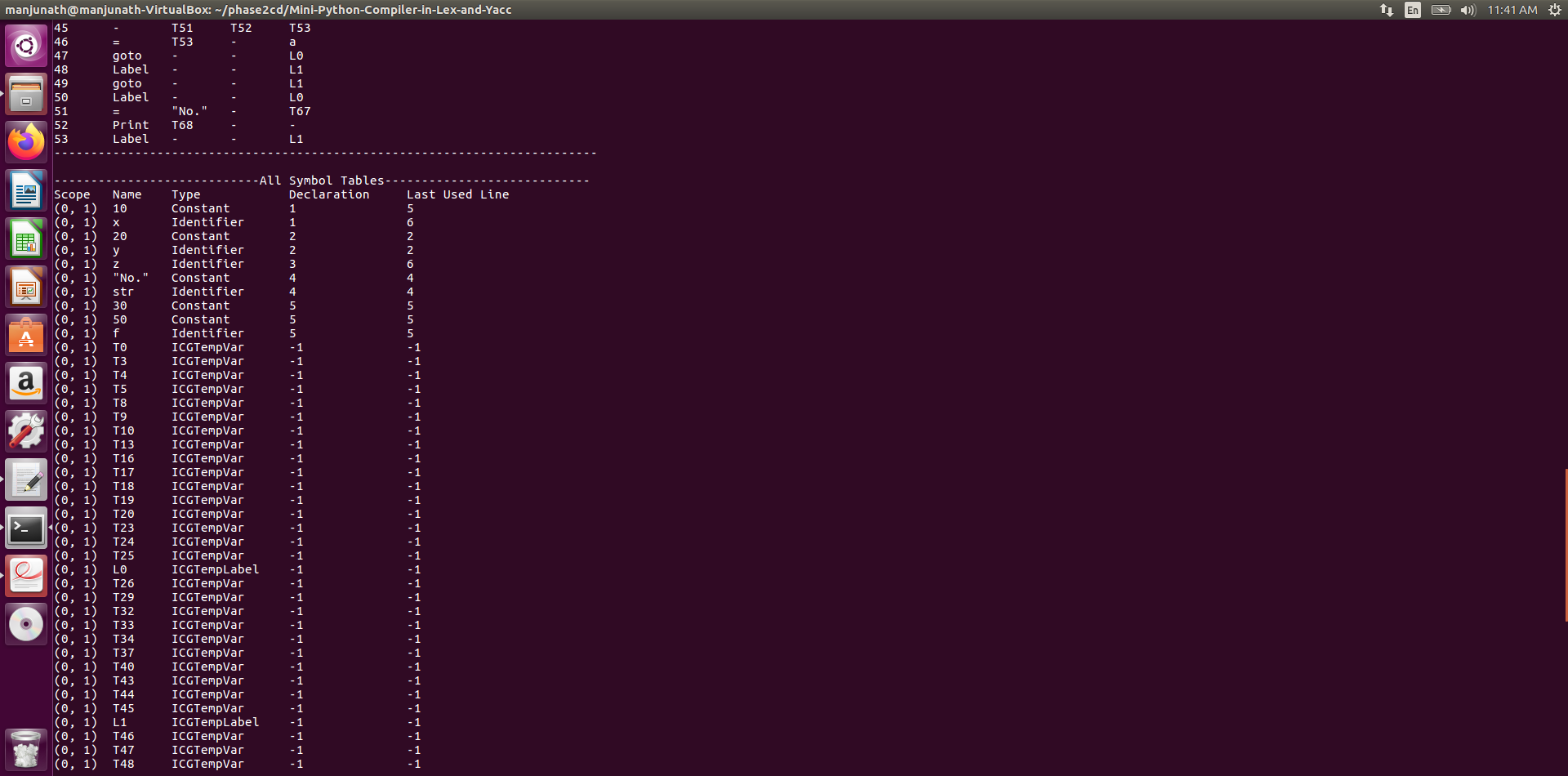
**Input with Semantic Error:**

**OutPut:**

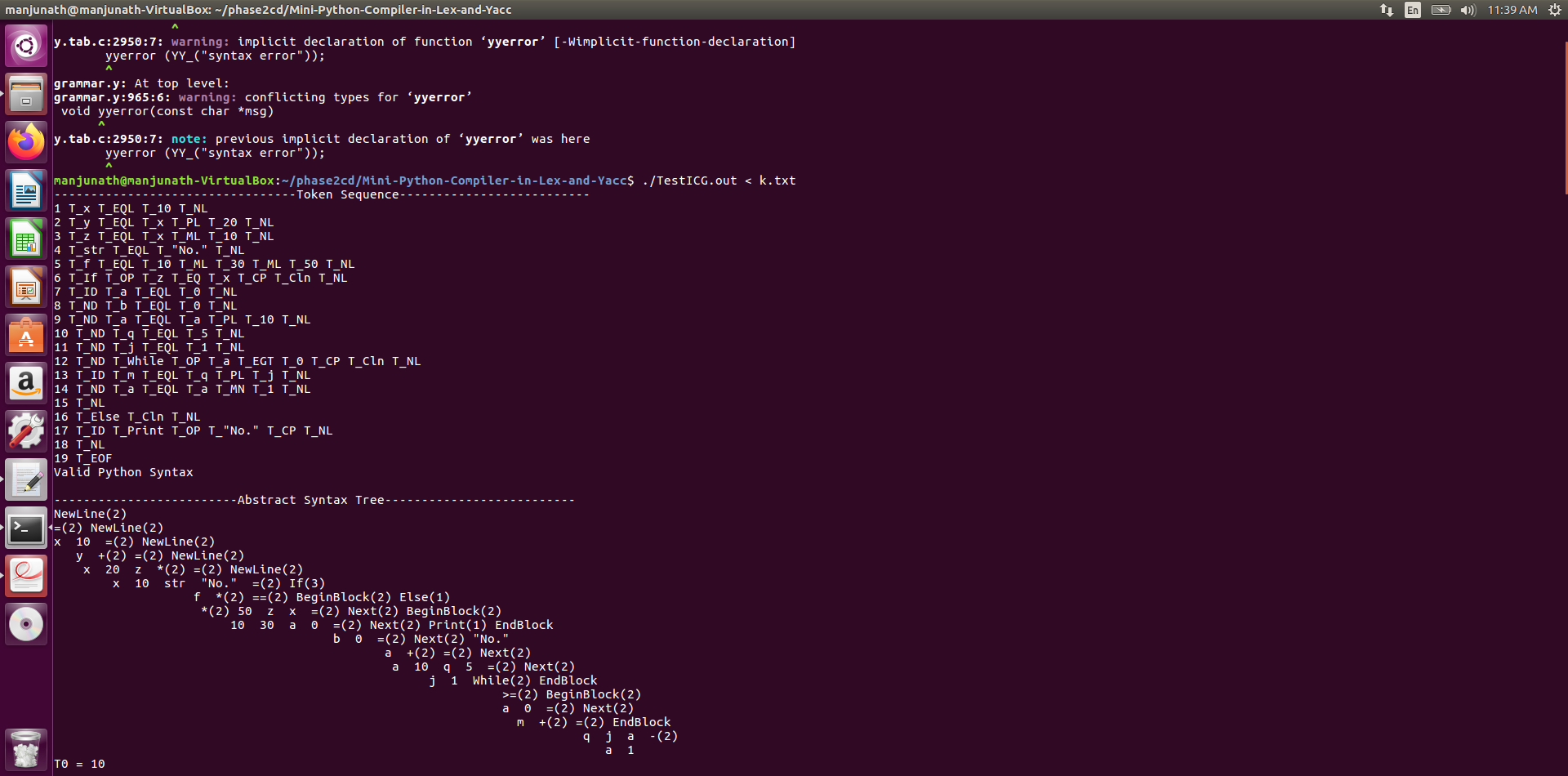
**Symbol Table :**

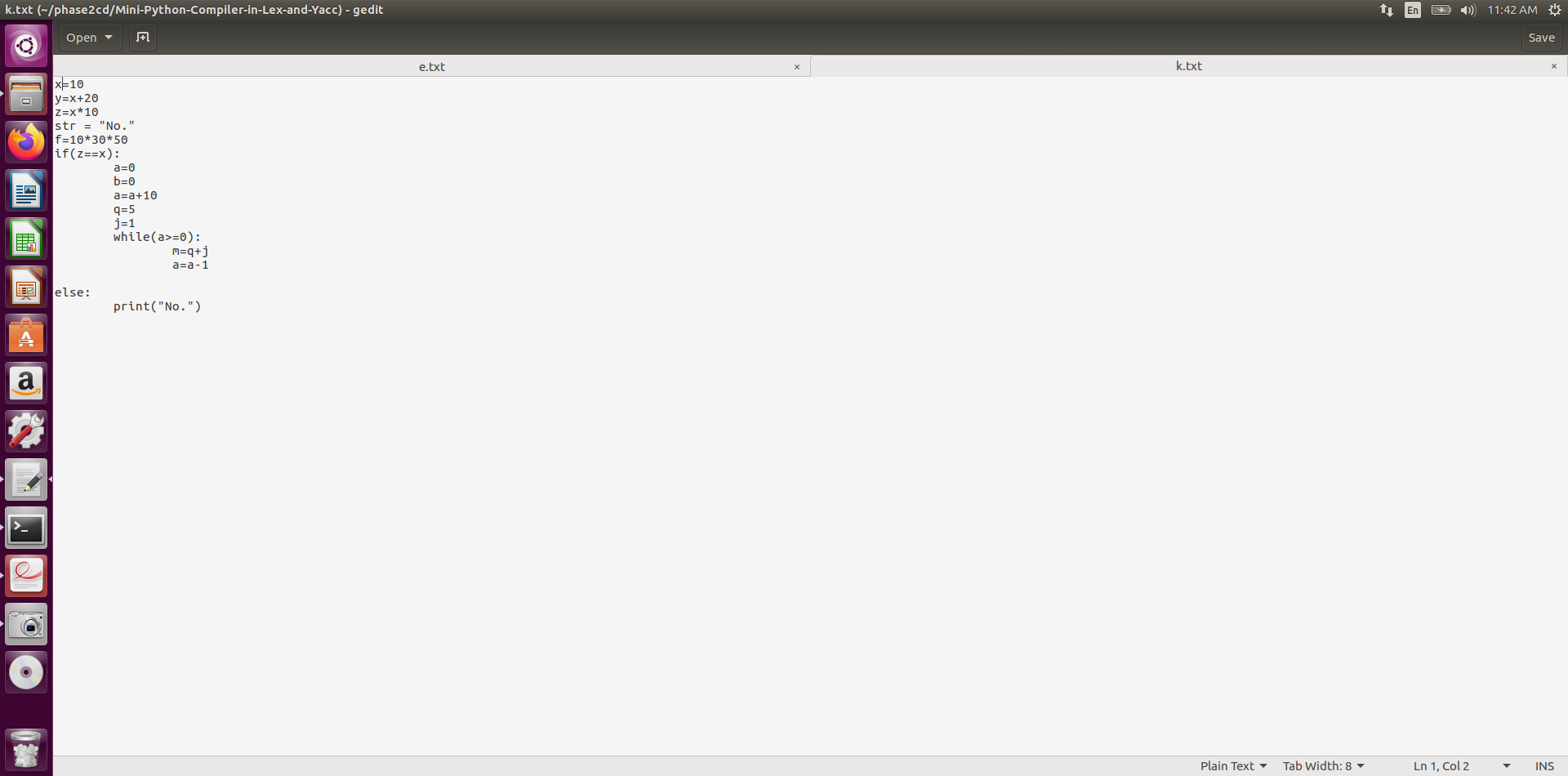
****

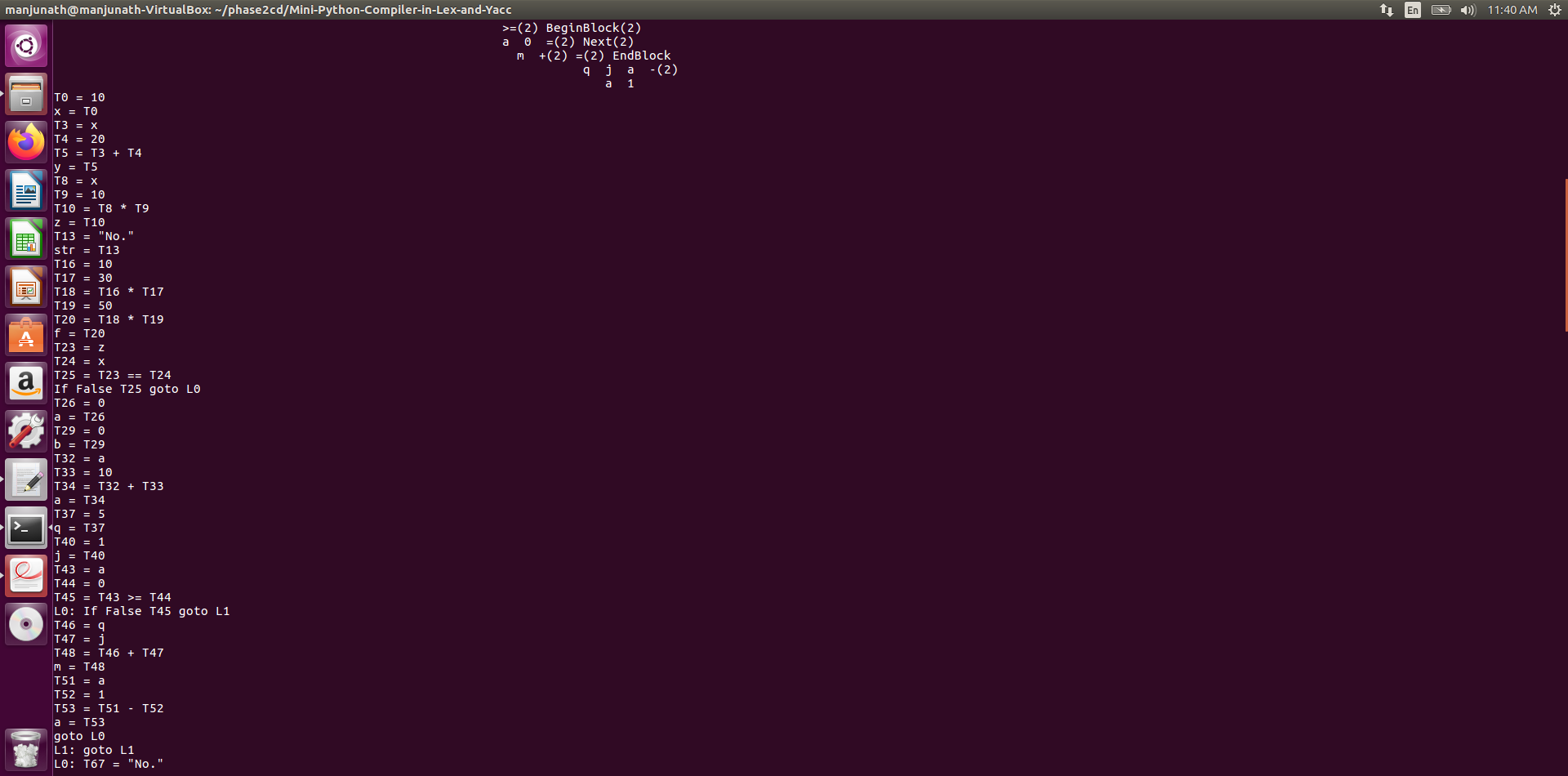
**Quadra tuple**

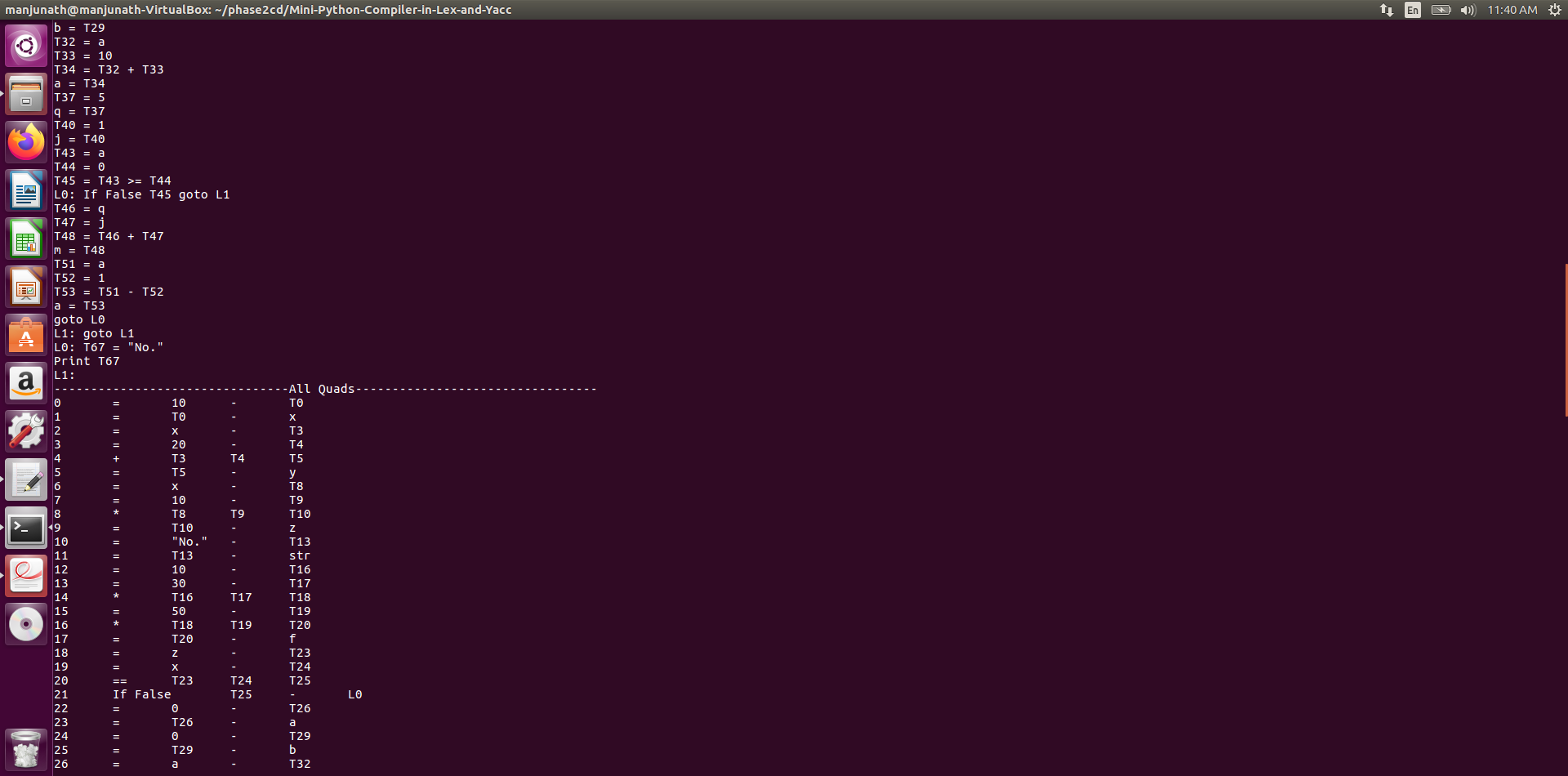
****

**TokenSequence-Abstract-syntax-tree**

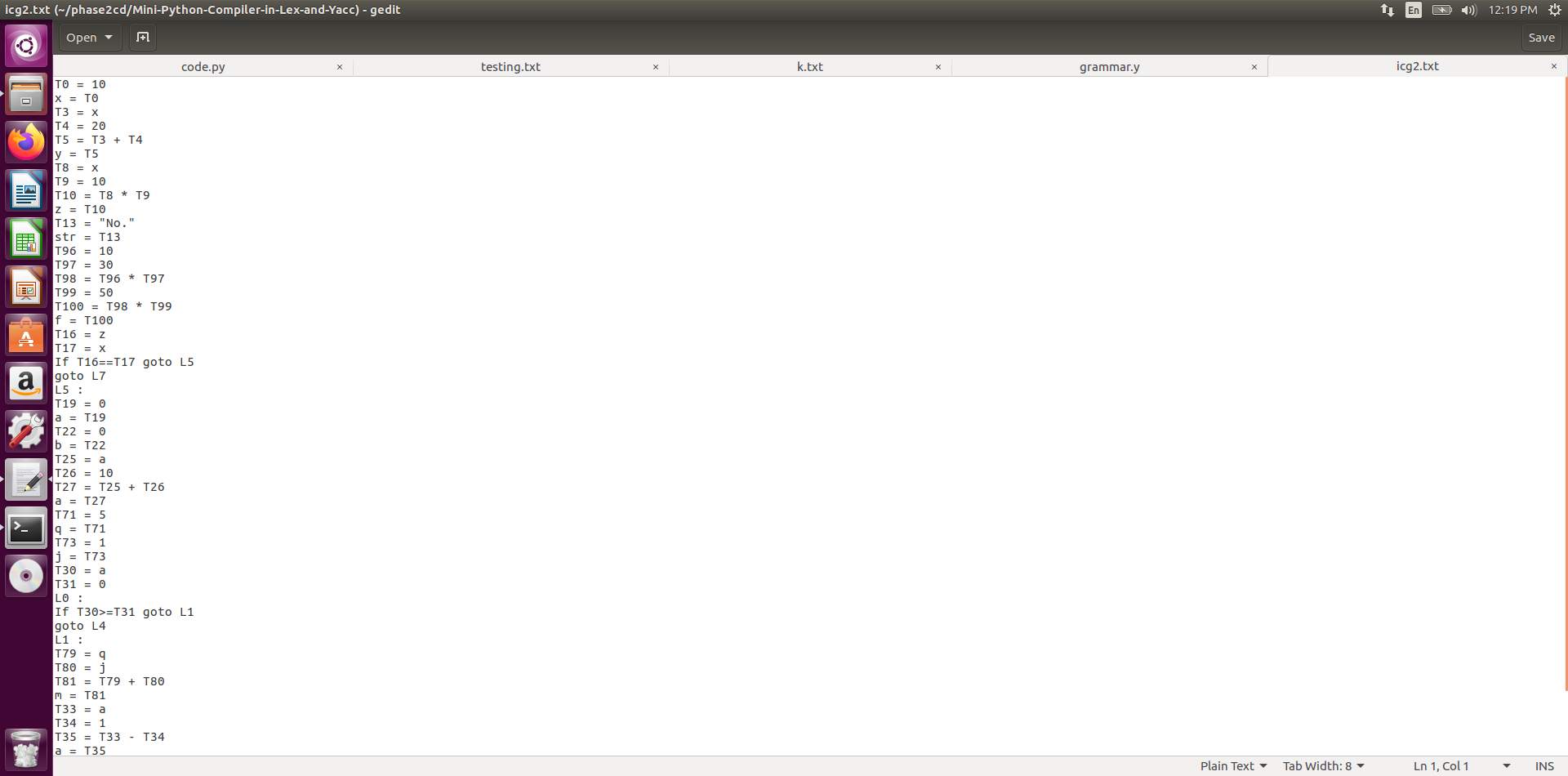
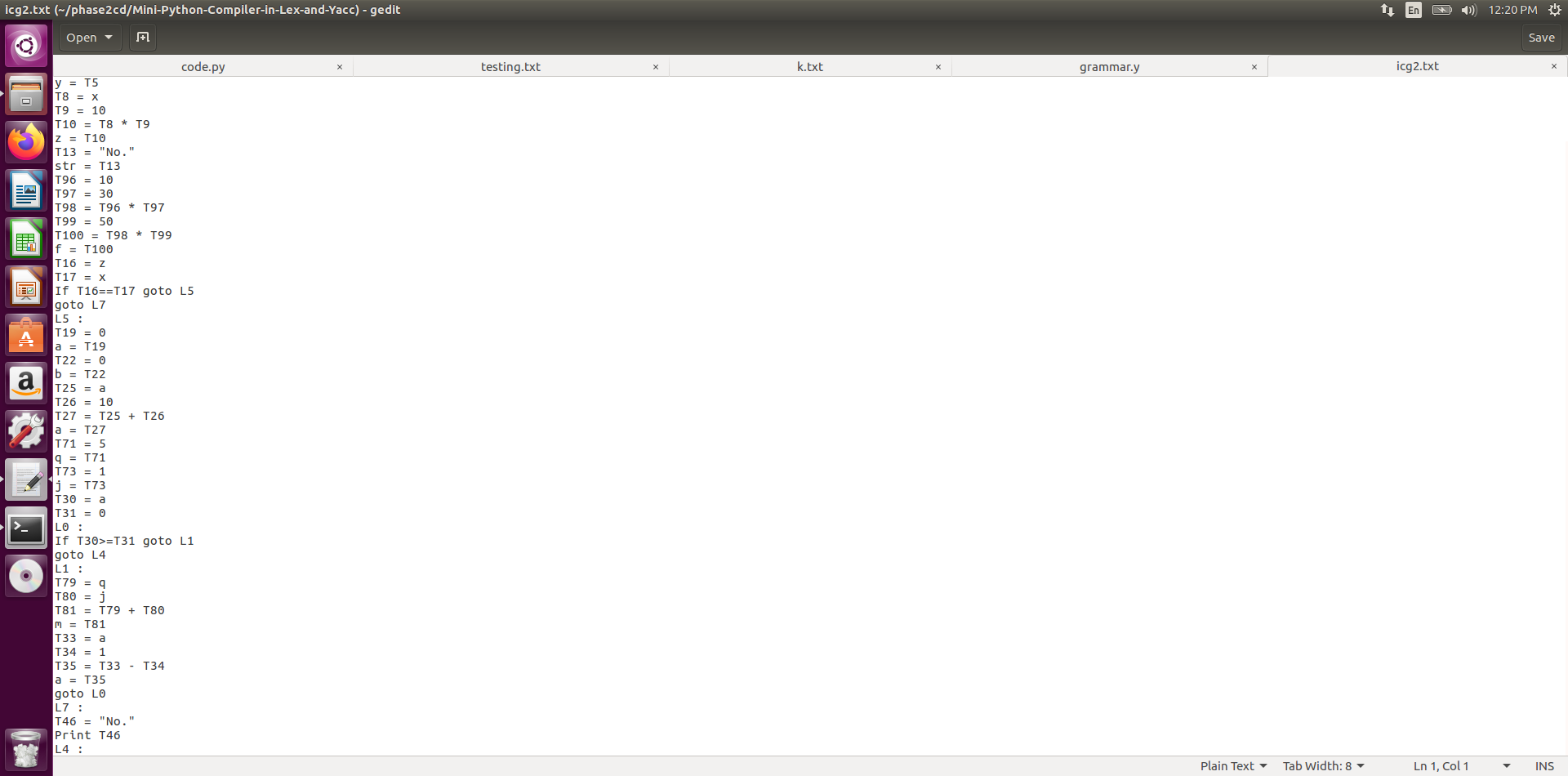


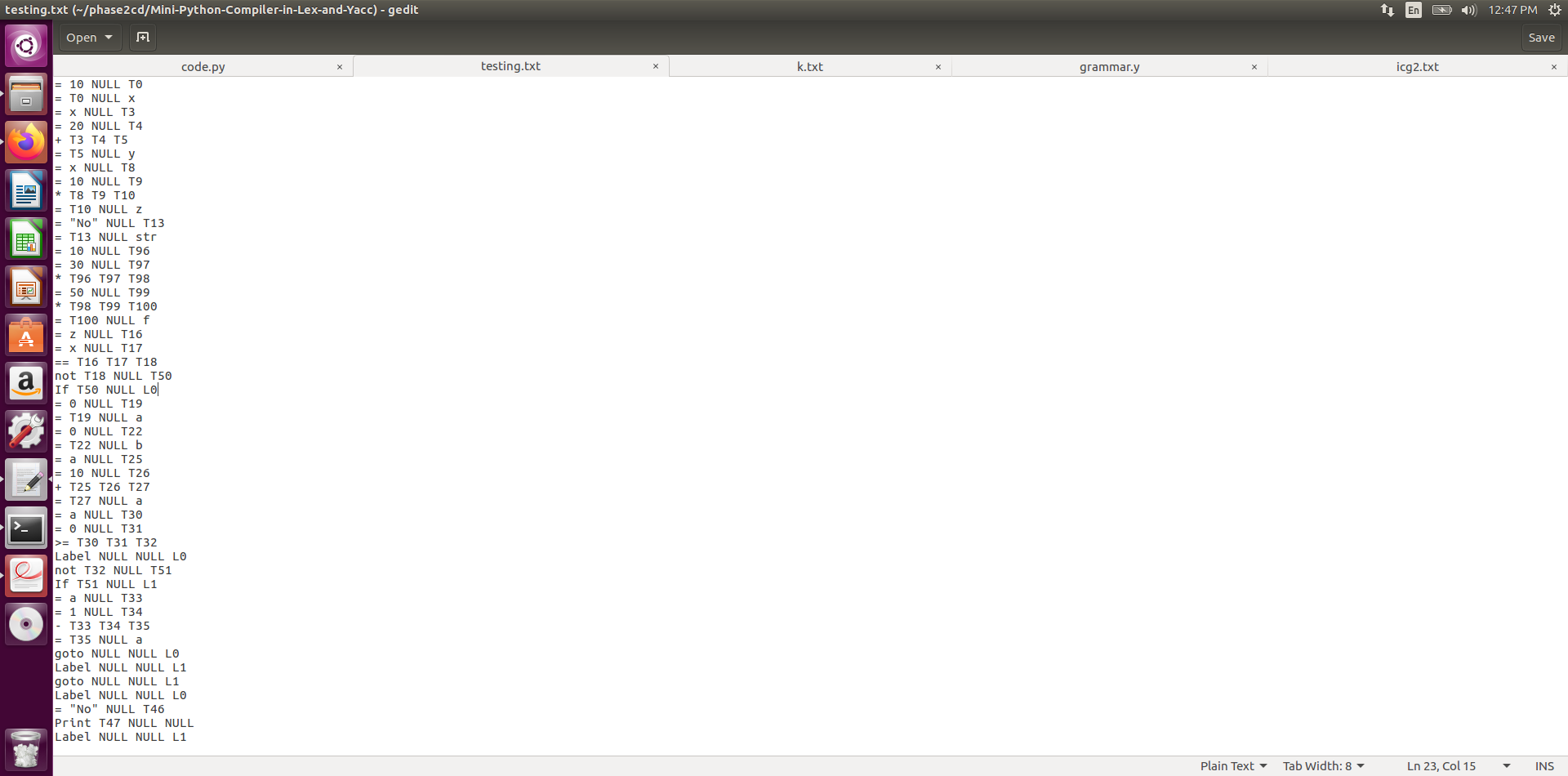
**Correct Input:**

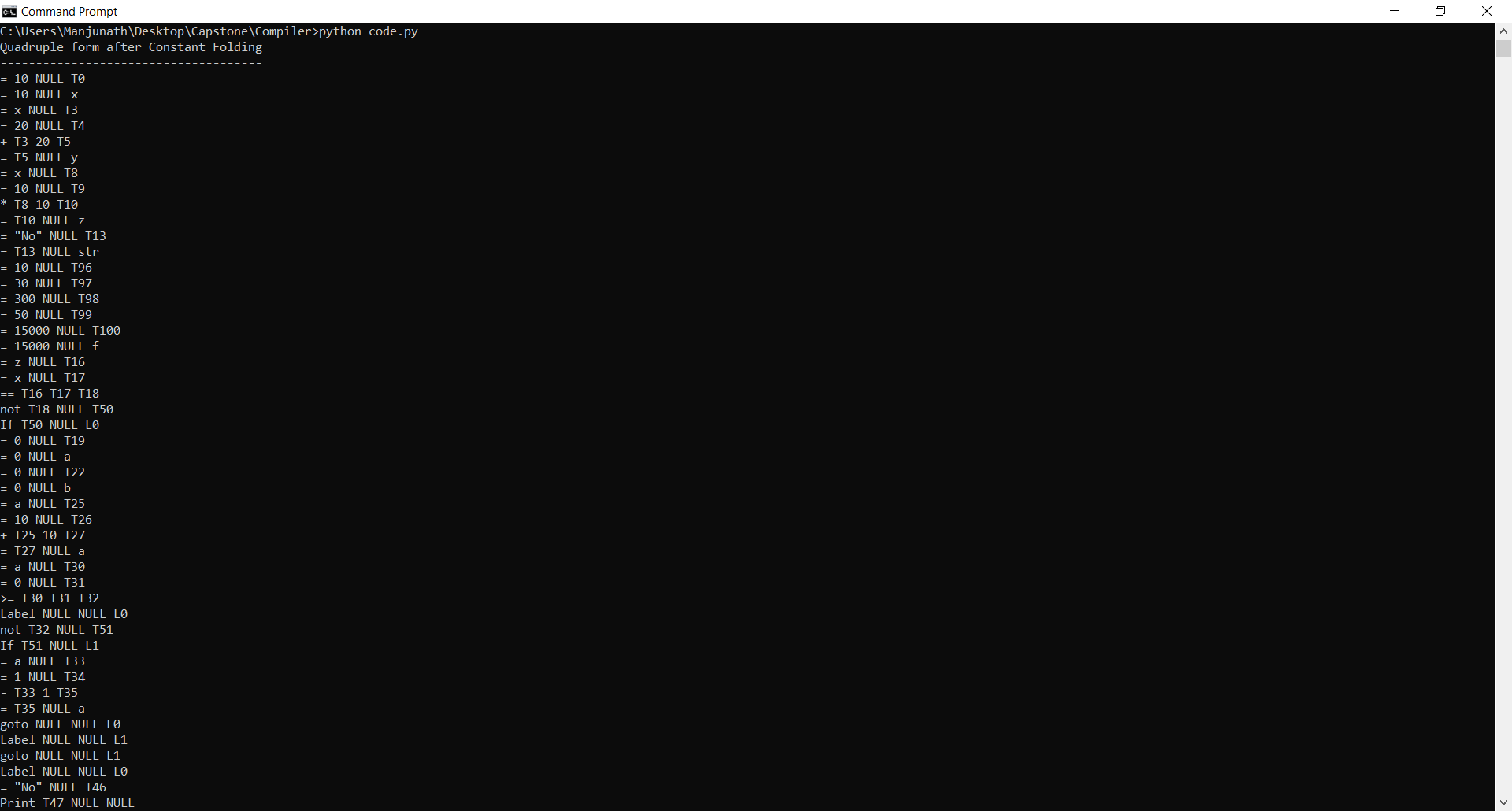
**ICG Output:**



**Input for Code Optimization:**

****

****

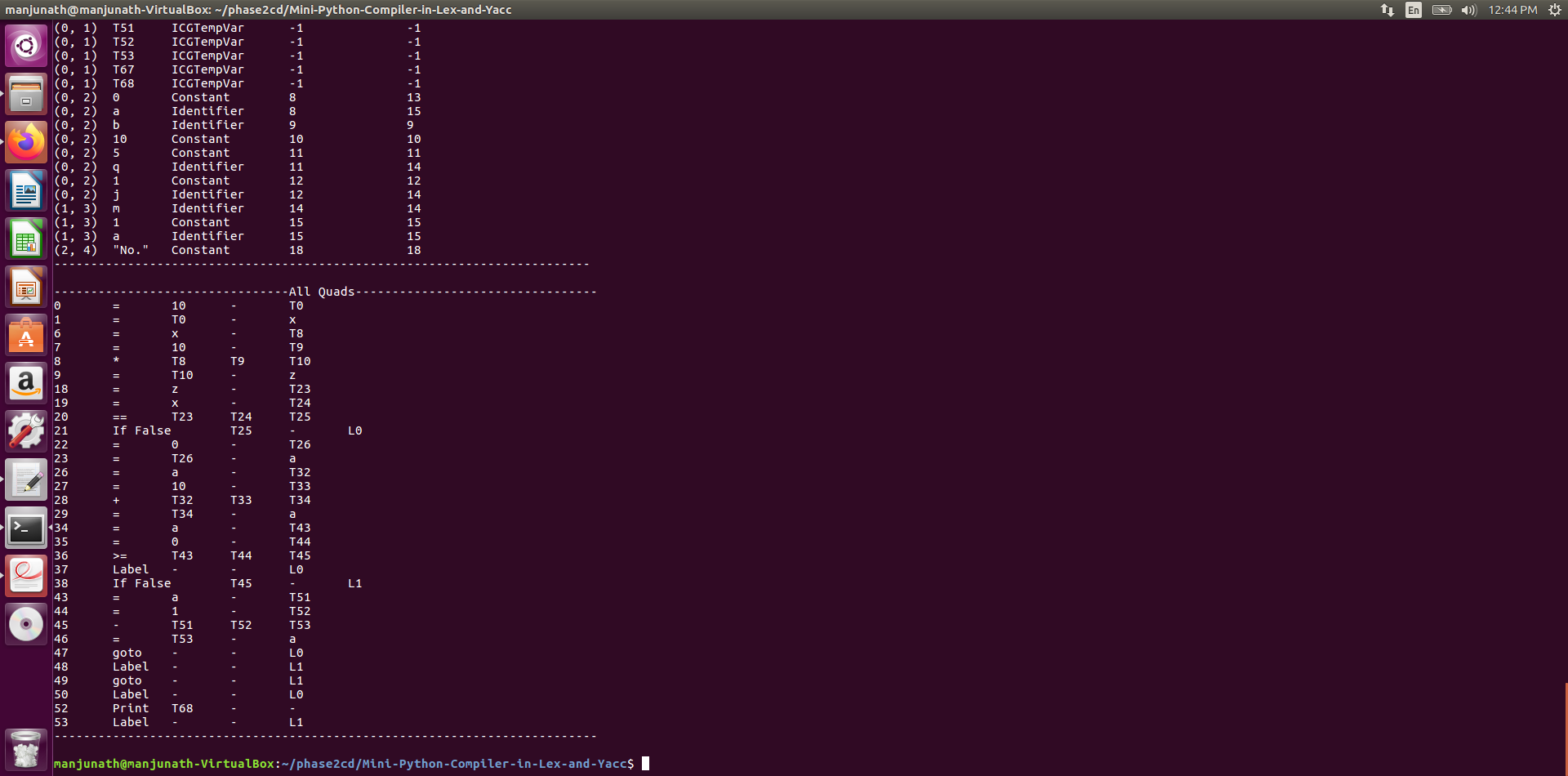
****

****

****

****

**Dead Code Optimization**

****

**CONCLUSION**

In Conclusion, A Compiler for Python was implemented. In addition to the constructs specified, the basic python constructs were implemented and function definitions and calls were supported. The compiler also reports the basic errors and gives the line number and column number. The Intermediate code was represented by quads which were later optimized to remove dead code.The compiler also checks for semantic errors .

**FURTHER ENHANCEMENTS**

1. More efficient optimization techniques
2. Support for ‘For’ Loops
3. Support for Do While Loops
4. Support for Nested Loops.