

*Report on*

“MINI PYTHON COMPILER WITH LEXER, PARSER, INTERMEDIATE CODE GENERATOR AND OPTIMIZATIONS TO HANDLE PYTHON CODE STRUCTURES WITH FOR LOOPS”

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

***Compiler Design Laboratory***

Bachelor of Technology

in

Computer Science & Engineering

***Submitted by:***

|  |  |
| --- | --- |
| **SINDHURA S** | **PES2201800374** |

***Under the guidance of:***

|  |
| --- |
| **Prof. Prajwala T R ma’am**  **Assistant Proffesor**  **Department Of Computer Science And Engineering**  **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 (Mini-Compiler is built for which language. Provide sample input and output of your project) | 01 |
|  | ARCHITECTURE OF LANGUAGE:   * What all have you handled in terms of syntax and semantics for the chosen language. | 02 |
|  | LITERATURE SURVEY (if any paper referred or link used) | 03 |
|  | 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 | |  |

1. **INTRODUCTION:**

In this project we have built a Mini-Compiler for Python language.

As a part of our Compiler Design Project, we have created a python compiler from scratch using the same Grammar Rules for Python.This Compiler is designed only for python code structures with loops and uses PLY tools and is written in Python itself.

1. **ARCHITECTURE OF LANGUAGE:**

THE MAIN CHALLENGE WAS TO DEAL WITH INDENTATION AS FAR THE LANGUAGE PYTHON IS CONCERNED.

The following things have been handled

-> User defined functions

-> For loop

-> while loop

-> Try and except clauses

-> If and else and elif statements

-> Lists

-> Binary operations and arithmetic operations

-> Import statements

-> Lambda functions

1. **LITERATURE SURVEY:**

<https://www.dabeaz.com/ply/ply.html>

[https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2 &cad=rja&uact=8&ved=2ahUKEwjx1qiy6-vhAhXGX30KHZLfCyQQFjAB egQIARAC&url=https%3A%2F%2Fwww.dabeaz.com%2Fply%2FPLYT](https://www.dabeaz.com/ply/PLYTalk.pdf)

[alk.pdf&usg=AOvVaw2R2Znb78j2Z84r7XbVKzfx](https://www.dabeaz.com/ply/PLYTalk.pdf)

<https://github.com/dabeaz/ply>

<http://compileroptimizations.com/category/constant_folding.htm>​l <http://compileroptimizations.com/category/dead_code_elimination.htm>

<http://jsonviewer.stack.hu/>

<https://github.com/pgbovine/python-parse-to-json>

1. **CONTEXT FREE GRAMMAR:**

S -> NV | SP | SaS

N -> n | dn | NP | NaN V -> vN | vS

P -> pN

<S> -> <NP> <VP> | <S> <PP> | <S> "and" <S> <NP> -> "n"

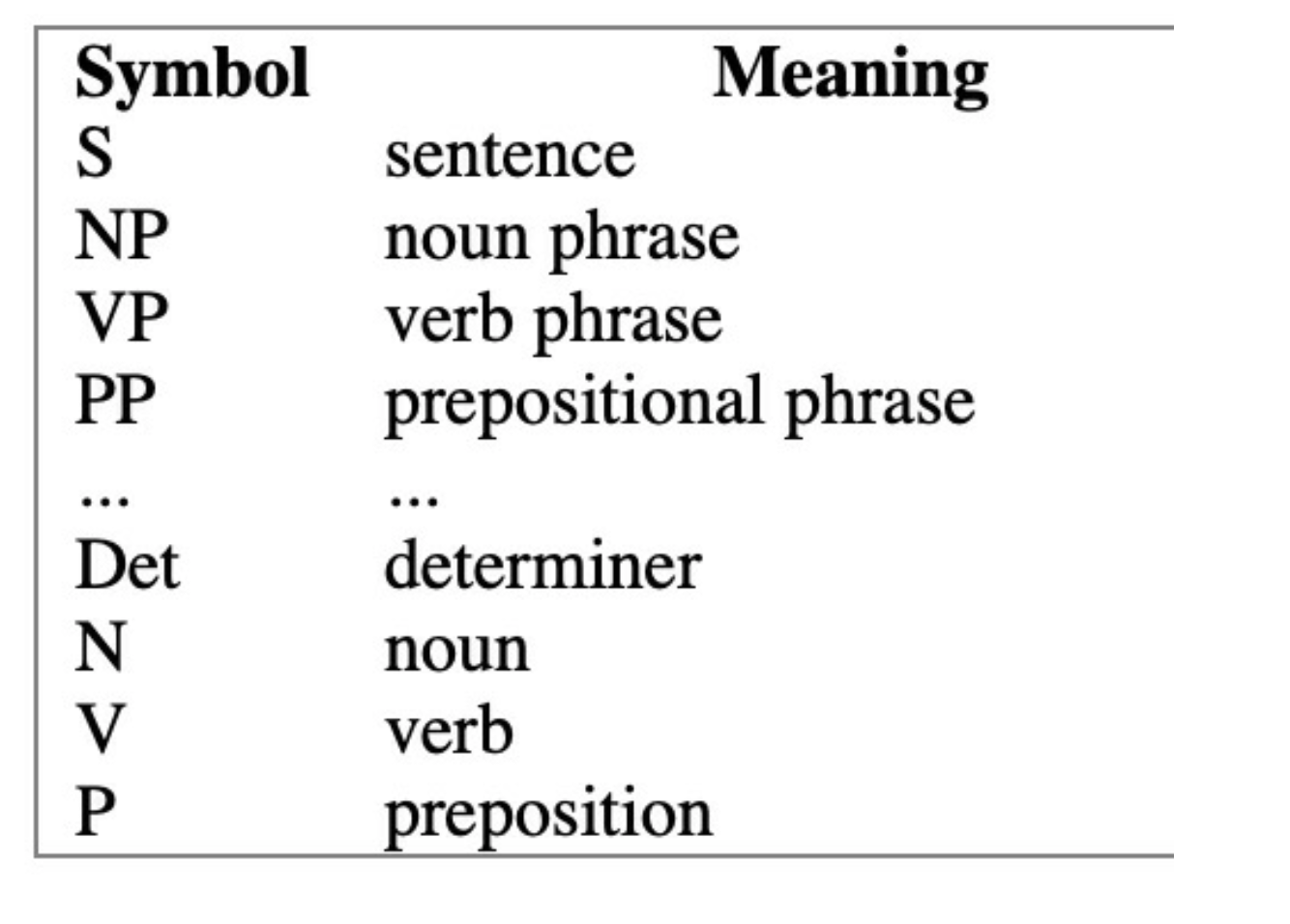
<NP> -> "det" "n"

<NP> -> <NP> <PP>

<NP> -> <NP> "and" <NP> <VP> -> "v" <NP>

<VP> -> "v" <S>

<PP> -> "p" <NP>



1. **DESIGN STRATEGY**

(Used to implement the following):

There are 4 stages of this project

1) THE LEXER (inclusive of ignoring singular and multiple comments)

2) PARSER (inclusive of identification handling)

3) INTERMEDIATE CODE GENERATION AND BUILDING AN ABSTRACT SYNTAX TREE

4) CODE OPTIMIZATION

* **SYMBOL TABLE CREATION**

Here the construction of symbol table takes scope and scope stack into consideration for assigning values to variables, assigning attributes and declaring identifiers.

It also takes care of the function default parameters. It is also responsible for assigning the width for different identifier types.

* **ABSTRACT SYNTAX TREE**

Here we are retrieving the parse tree from the parser and then we are converting the parse tree entries to json objects which then are fed to an online tree generator which gives us the constructed abstract syntax tree.

* **INTERMEDIATE CODE GENERATION**

We are storing the intermediate code in 3 address format - the quadraple format using registers. We are using a list of lists for storing the intermediate code.

We are handling the icg also for user defined function calls .

* **CODE OPTIMIZATION**

Here the two optimisations which we are performing are

->Constant Folding

->Dead Code Elimination.

The constructs being used here are lists, we are opening and storing the icg code in a list of lines and then we classify identifiers and numbers/constants seperately.

1. **IMPLEMENTATION DETAILS**

(TOOL AND DATA STRUCTURES USED in order to implement the following):

* **SYMBOL TABLE CREATION**

The following functions are implemented to build the symbol table ---->lookup,lookupScopeStack--- Looks for an identifier with the help of the scope numbers we generated in the lexical phase

->getcurrentscope

->addscope -> for user defined functions

->addidentifier

->addattribute,getattribute

->getattributefrom currentscope

->addattributeto currentscope

->getattributefromfunction list

->printst

->getwidthfromtype

->printsymboltablehistory

Structure

Entry: Scopename

Type

Return type

* **INTERMEDIATE CODE GENERATION**

Here we are using an array list or a list of lists to implement the intermediate code, we are using quadruple format to store the three address code and storing the quadraple of each instruction in a seperate list, then we concatenate the induvidual lists to a master list which gives us the whole code of the program.

Some functions being implemented are

->incrementQuad

->getnextQuad

->emit

->createnewFunctionCode

->printCode

* **CODE OPTIMIZATION**

The optimisations being implemented are Constant Folding and Dead-Code Elimination.

Main functions being implemented are: evalwrap() -> Evaluates each instruction.

fold constant() -> Does constant folding ,takes list of lines as function arguments and outputs the optimized code.

remove dead code() -> Removes dead code ,takes list of lines as function arguments and outputs the optimized code.

* **ERROR HANDLING** *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).

Syntax errors, parsing errors and value and name errors are being handled.

* **Provide instructions on how to build and run your program.**

Run the code with the following commands on your terminal:

python3 <filaname.py>

1. **RESULTS and possible shortcomings of your Mini-Compiler:**

**Test case 1:**

a=8

i=10 #Comment is to be cleaned

#if i < 15 :

# i++

#else :

# a++

def fun():

j=10

print("hi HEllo")

**Test case 2:**

//Single Comment

fn main(){

let a = 3 + 4 \* 9;

for x in 1..4 {

print!("hello");

print!("world");

}

let b = a + 4;

}

/\* Multiline Comments \*/

/// Generate library docs for the following item.

//! Generate library docs for the enclosing item.

**Test case 3:**

fn main(){

let a = 3 + 4 \* 9;

let f = a + 1;

while 1 < 4 {

let b = 3 + a;

if b > 10{

while a < 5 {

print!("hey");

}

let a = 1;

print!("TRUE");

c = 3 + 4\*5 - a;

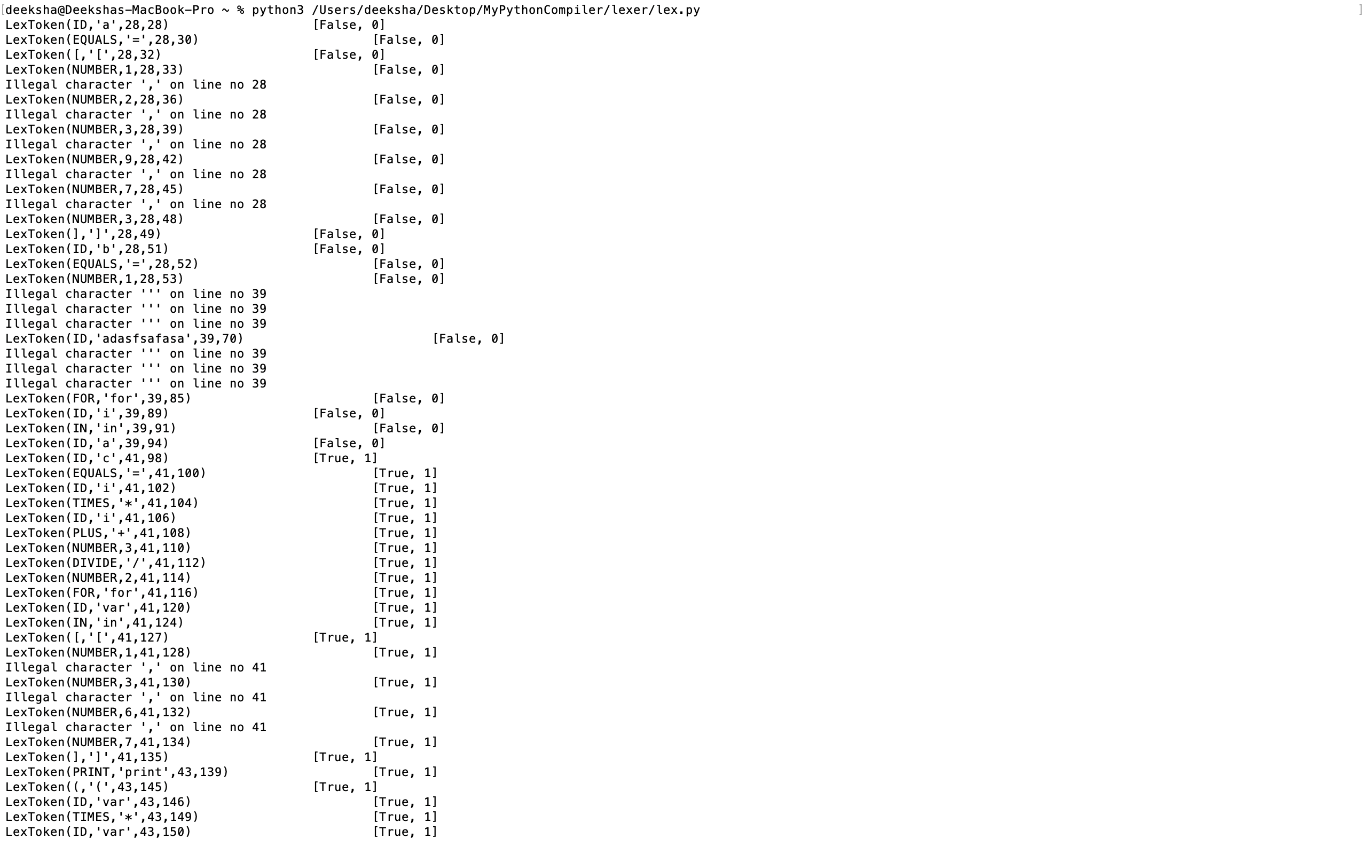
}

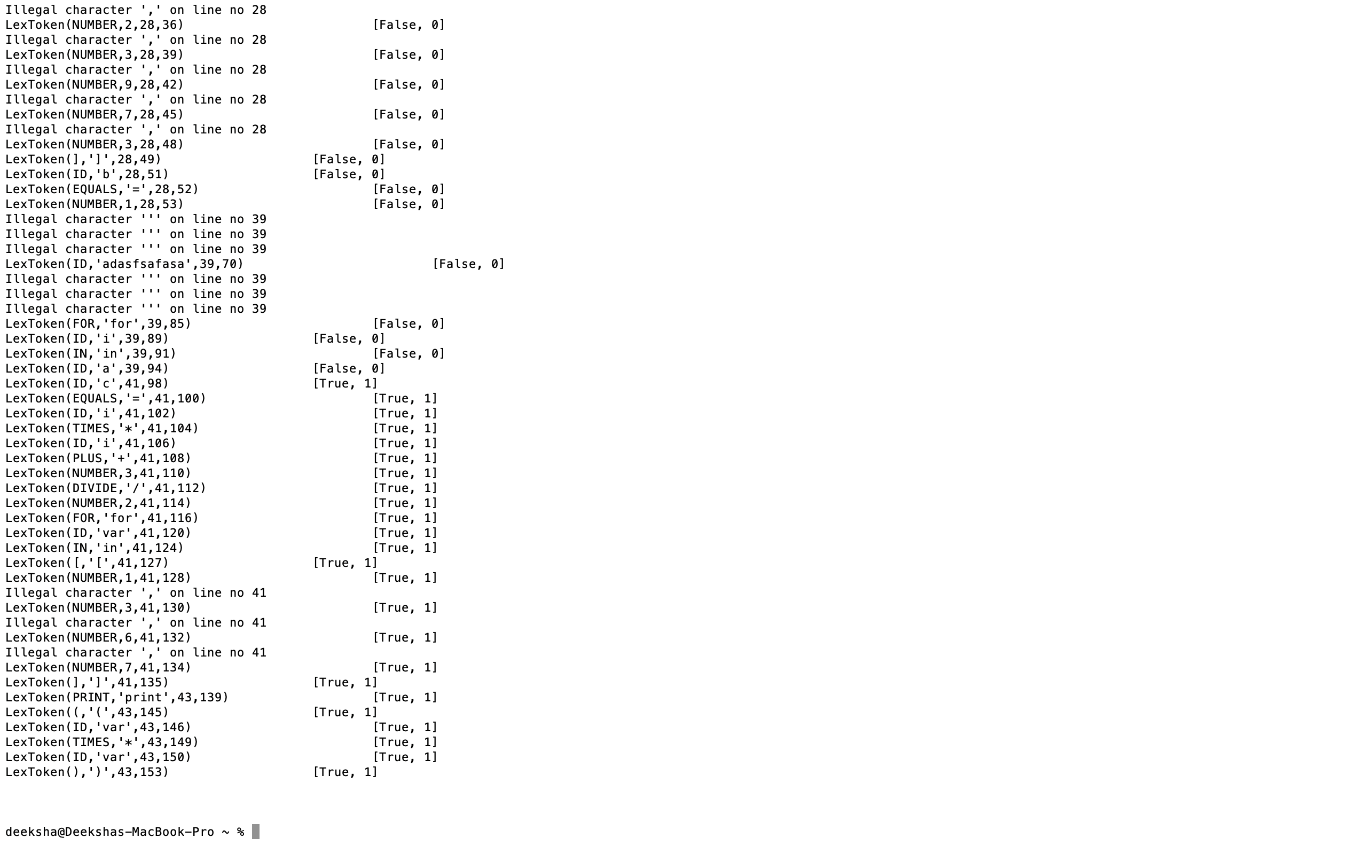
}

}

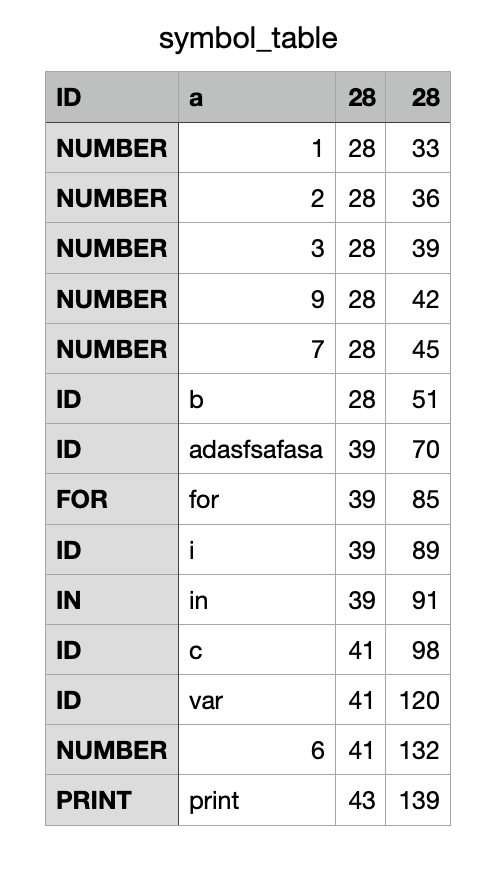
1. **SNAPSHOTS**

**LEXICAL ANALYSIS**

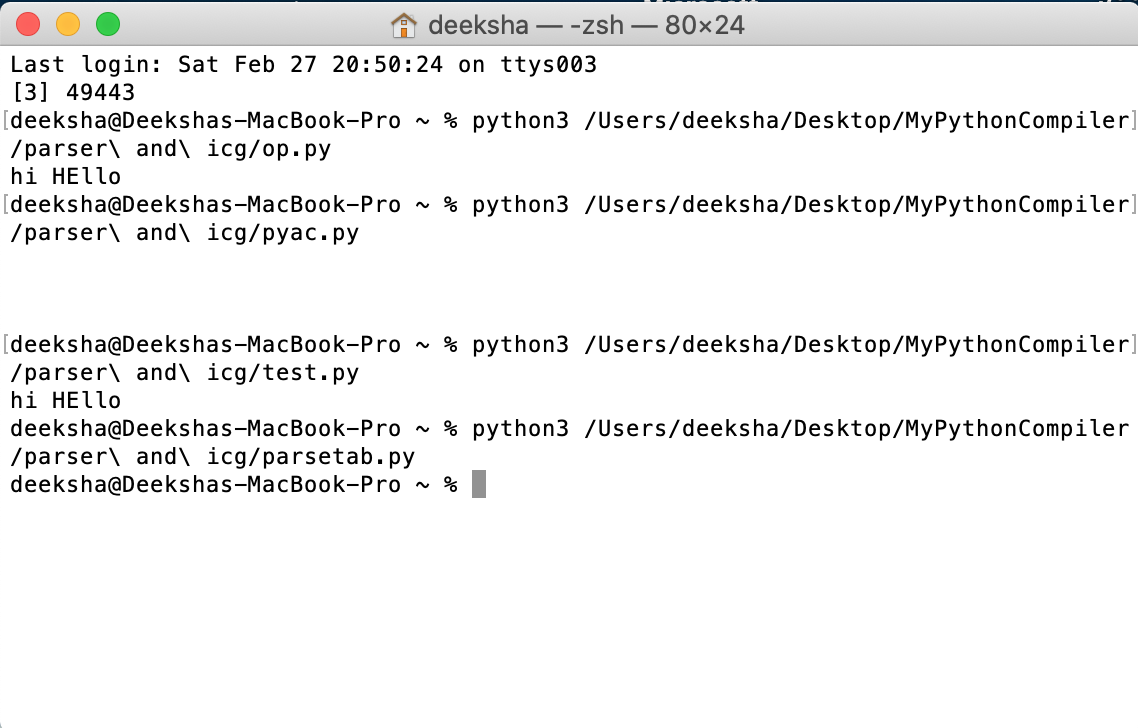
****

****

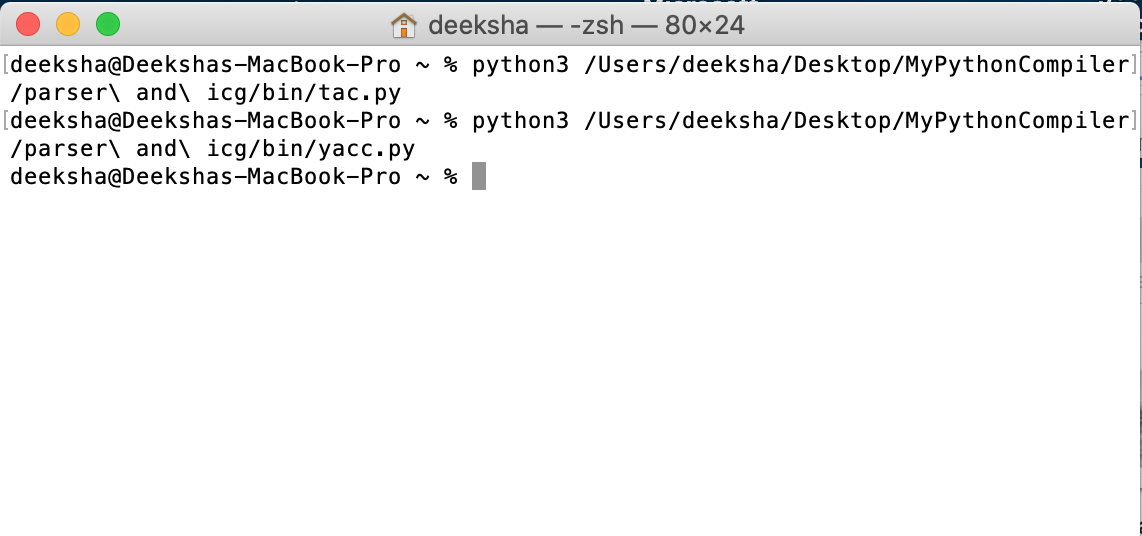
**SYMBOL TABLE**

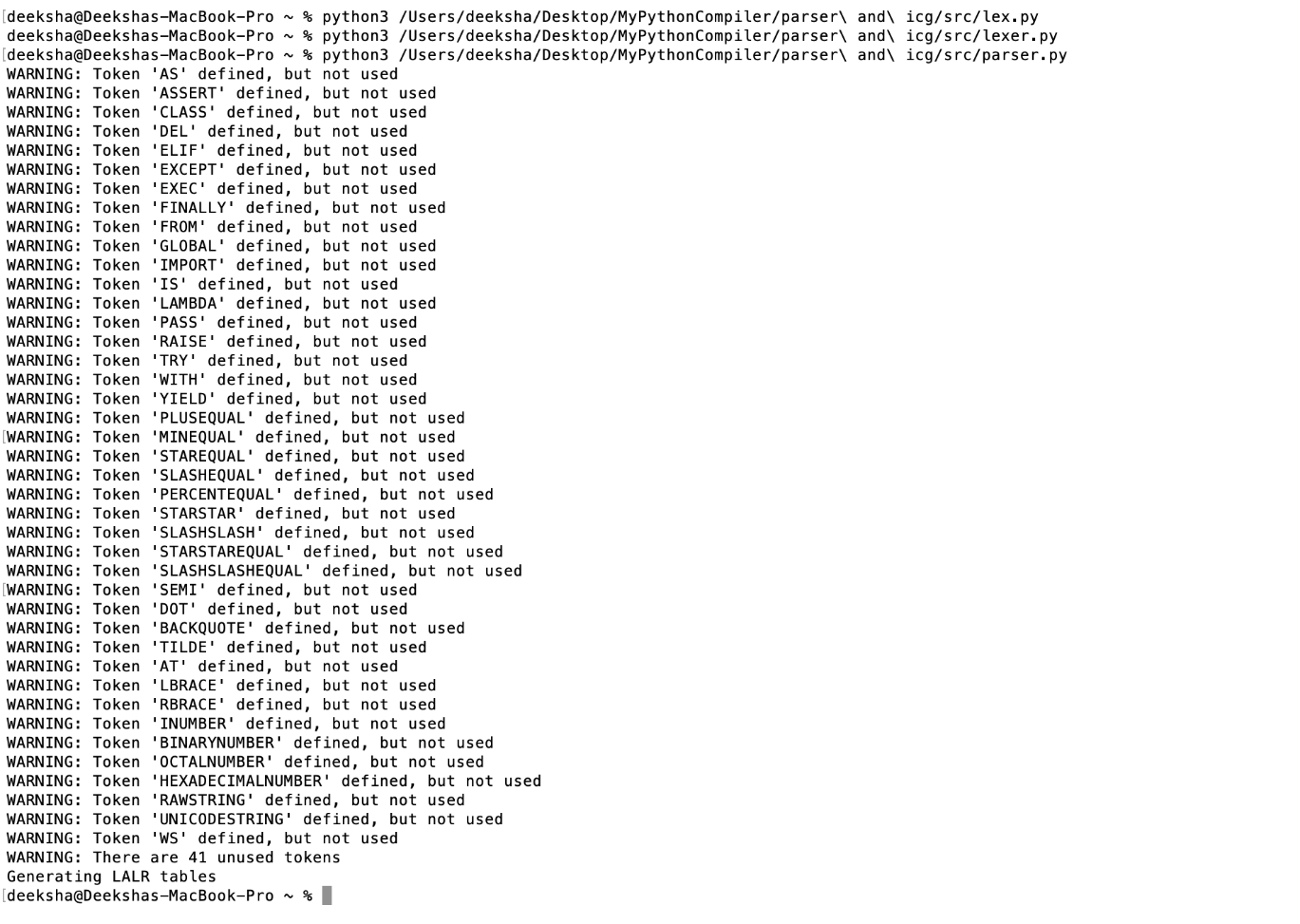
****

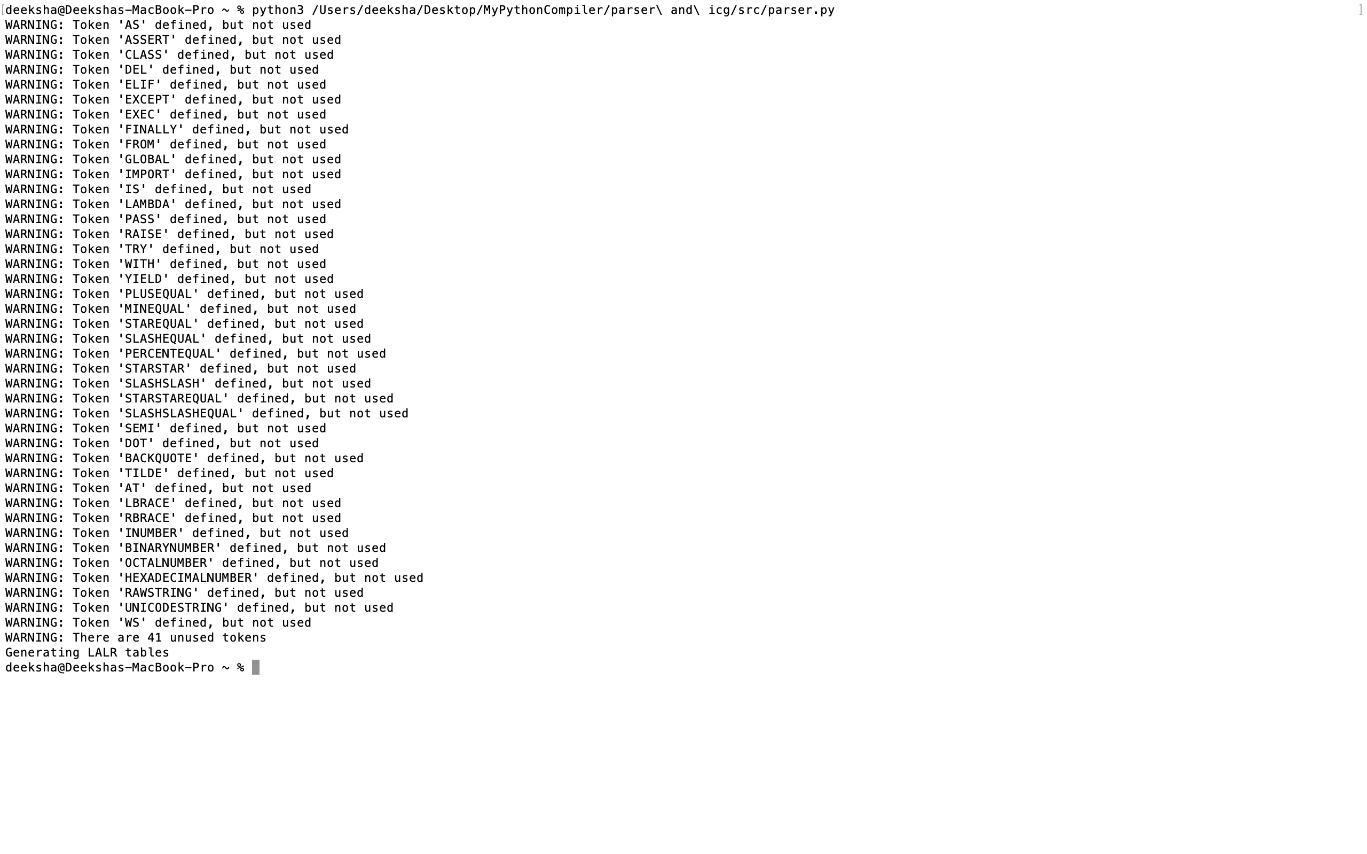
**PARSER:**

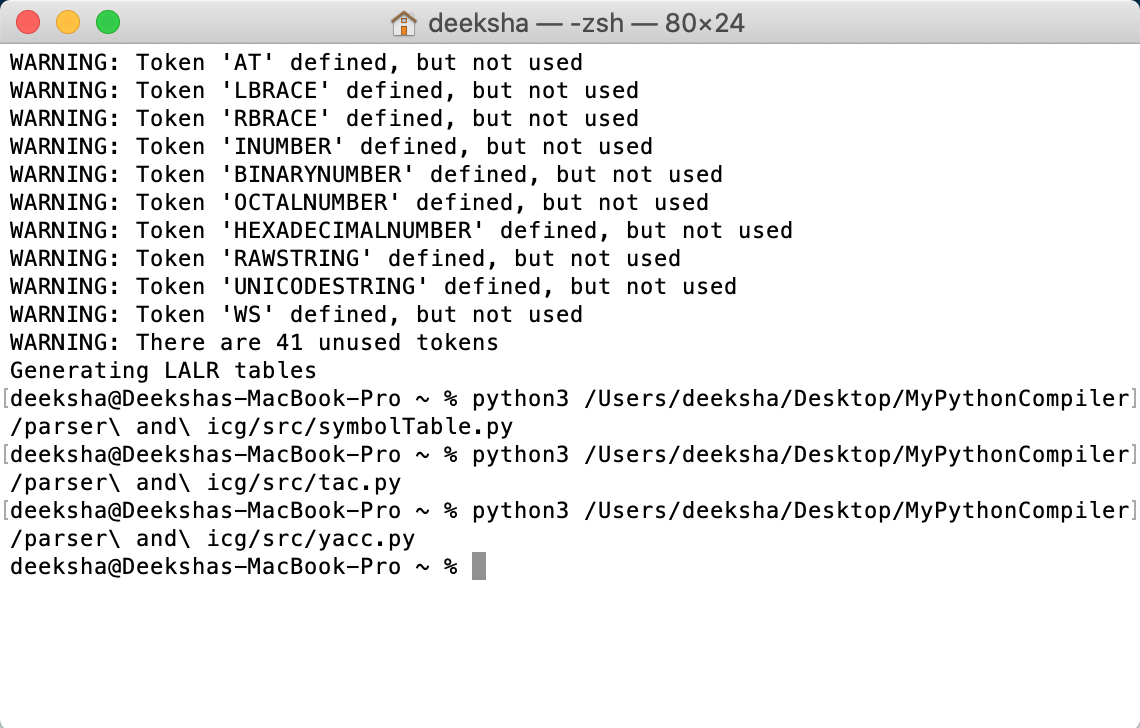
****

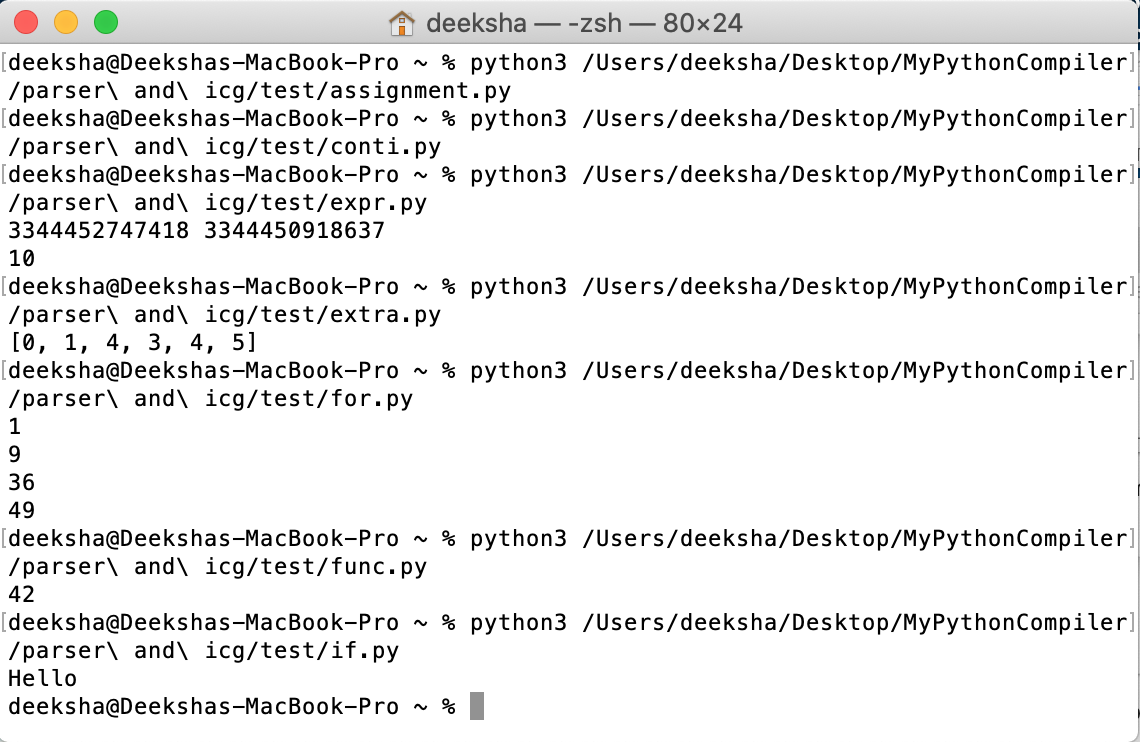
****

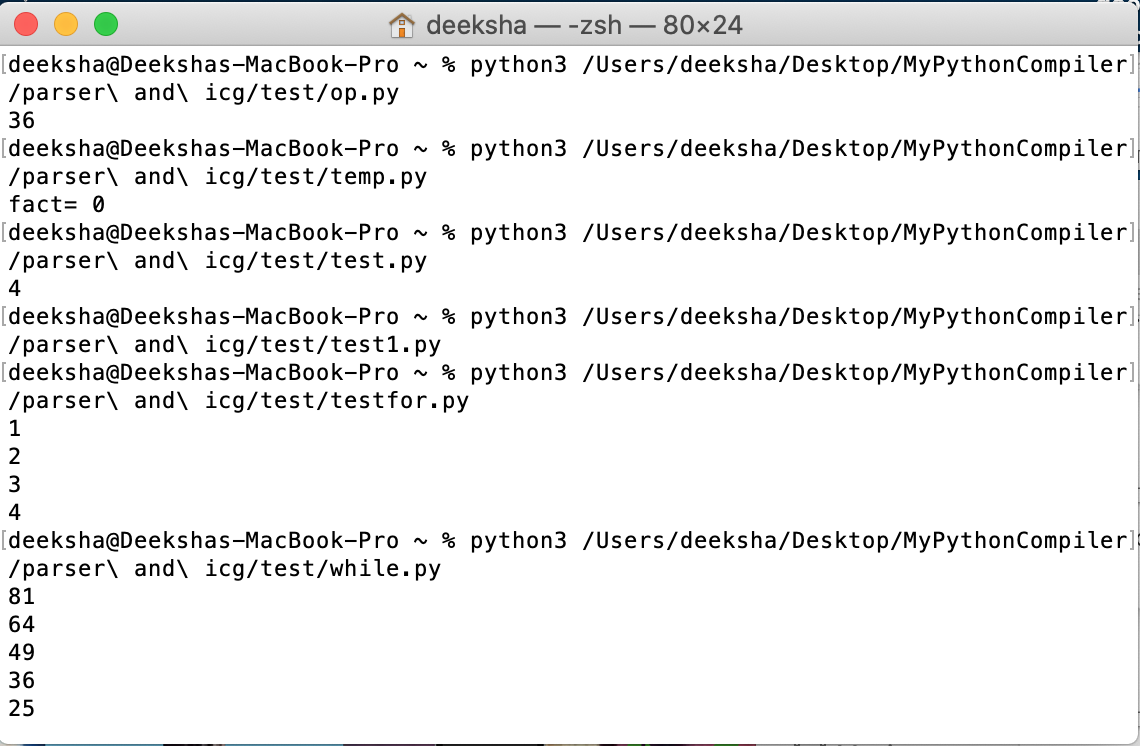
****

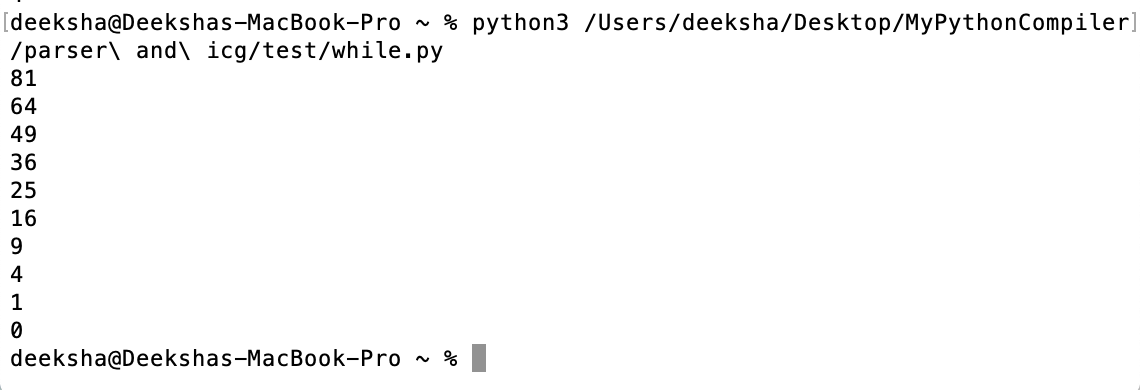
****

****

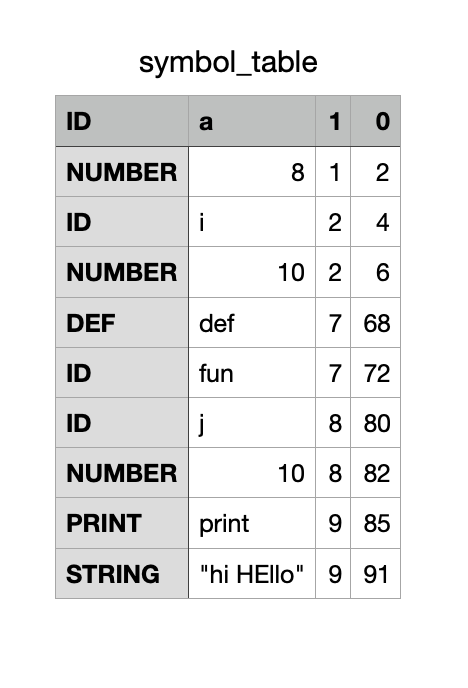
****

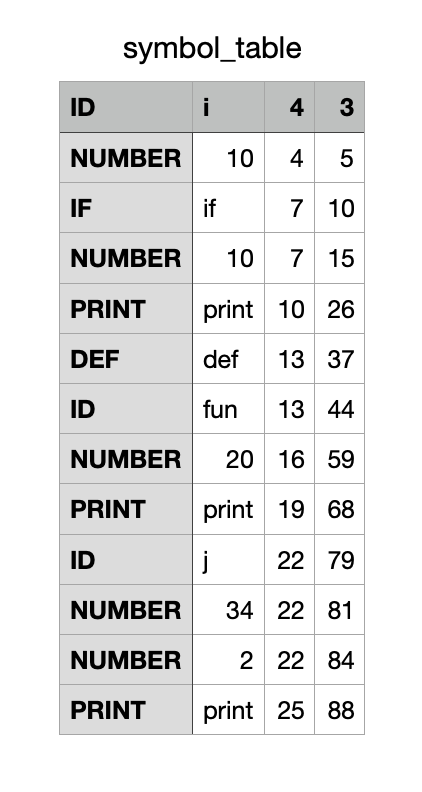
****

****

****

**SYMBOL TABLE**

****

****

## ABSTRACT SYNTAX TREE

## C:\Users\SINDHURA\Downloads\MyPythonCompiler-20210415T175734Z-001\MyPythonCompiler\ss\ast\1.png

## C:\Users\SINDHURA\Downloads\MyPythonCompiler-20210415T175734Z-001\MyPythonCompiler\ss\ast\2.png

## C:\Users\SINDHURA\Downloads\MyPythonCompiler-20210415T175734Z-001\MyPythonCompiler\ss\ast\3.pngC:\Users\SINDHURA\Downloads\MyPythonCompiler-20210415T175734Z-001\MyPythonCompiler\ss\ast\4.png

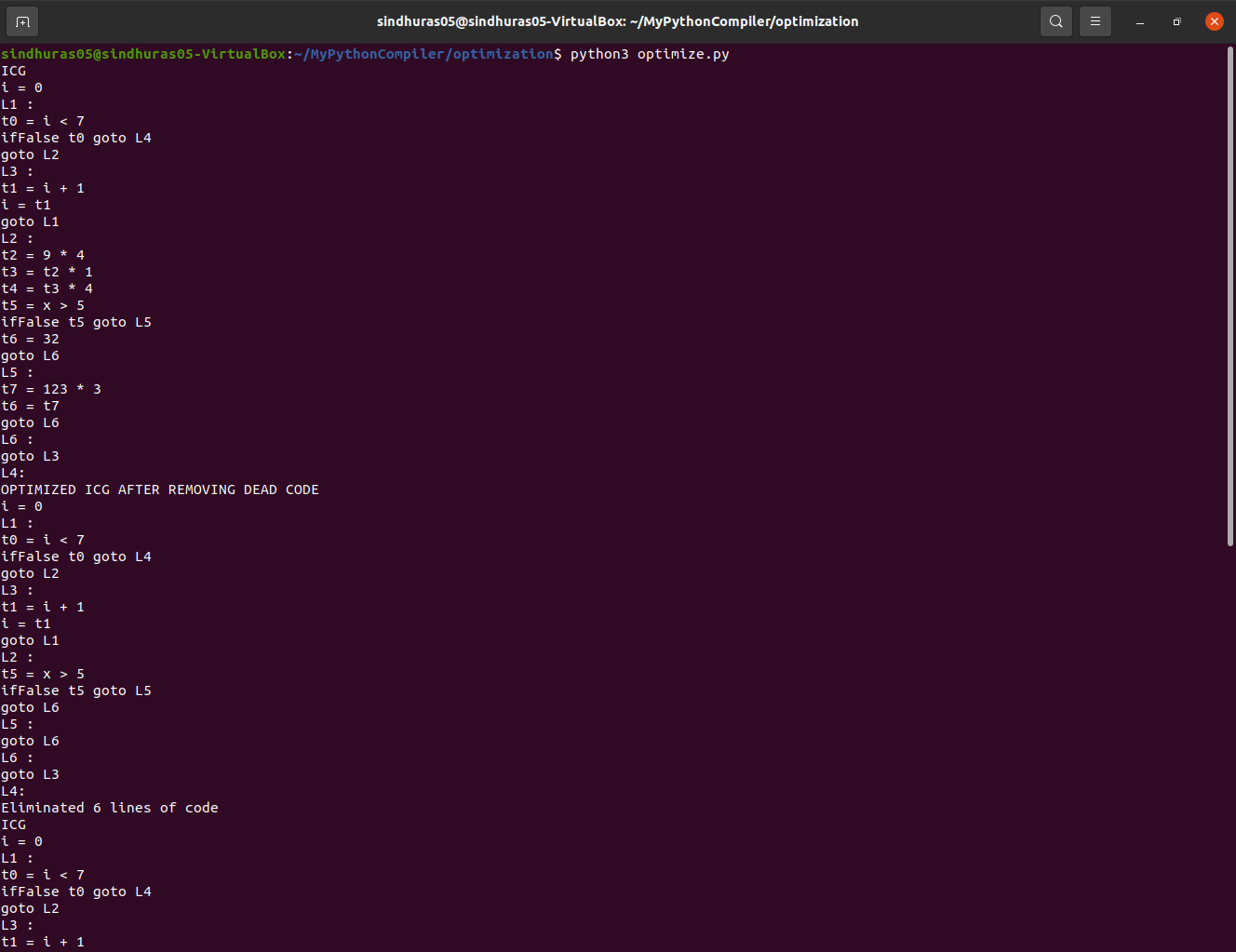
## C:\Users\SINDHURA\Downloads\MyPythonCompiler-20210415T175734Z-001\MyPythonCompiler\ss\ast\5.png

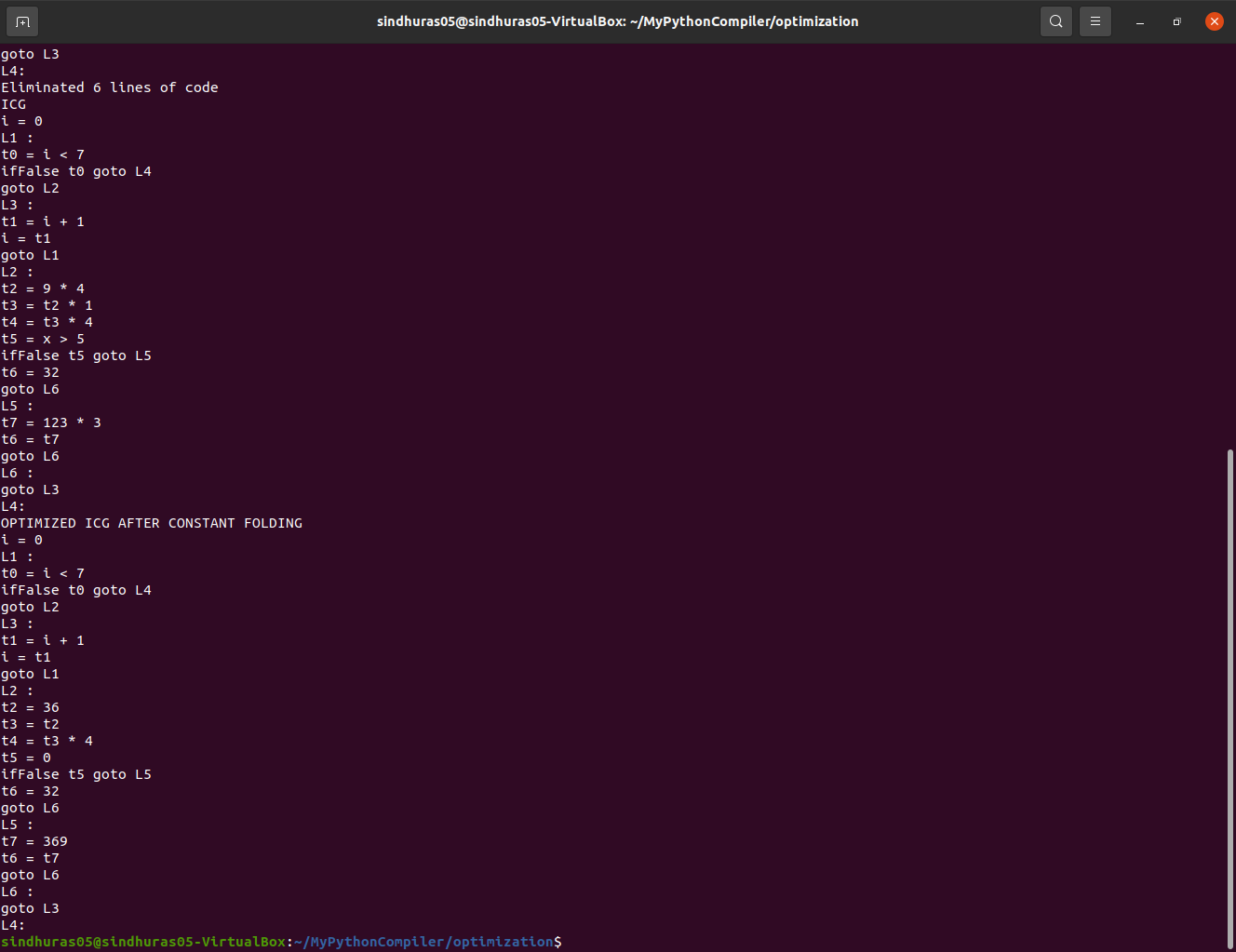
## C:\Users\SINDHURA\Downloads\MyPythonCompiler-20210415T175734Z-001\MyPythonCompiler\ss\ast\6.png

## C:\Users\SINDHURA\Downloads\MyPythonCompiler-20210415T175734Z-001\MyPythonCompiler\ss\ast\7.png

## C:\Users\SINDHURA\Downloads\MyPythonCompiler-20210415T175734Z-001\MyPythonCompiler\ss\ast\8.png

## CODE OPTIMIZATION





## CONCLUSIONS

A mini-compiler for python is successfully built. It is built via 4 phases: Lexical Analysis, Parsing, Intermediate Code Generation, Syntax Analysis and Code Optimisation.

Symbol Table, 3 address code and Abstract Syntax Tree has been generated.

Our compiler is able to identify and handle Value error, Key error, Indentation error, Syntax error.

## FURTHER ENHANCEMENTS

We can enhance it further by handling list comprehensions and classes. And we could include more optimization techniques to make it more accurate.