

***Report on***

**“**Mini Compiler for If-Else and 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:***

|  |  |
| --- | --- |
| **<Arindaam Mandal>**  **<Siddhant Tripathi>**  **<Nipun Bhat>** | **PES1201800274>**  **<PES1201800198>**  **<PES1201801857>** |

*Under the guidance of*

|  |
| --- |
| **Preet Kanwal**  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)

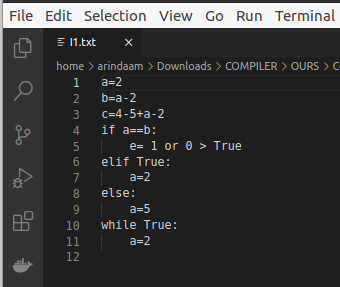
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

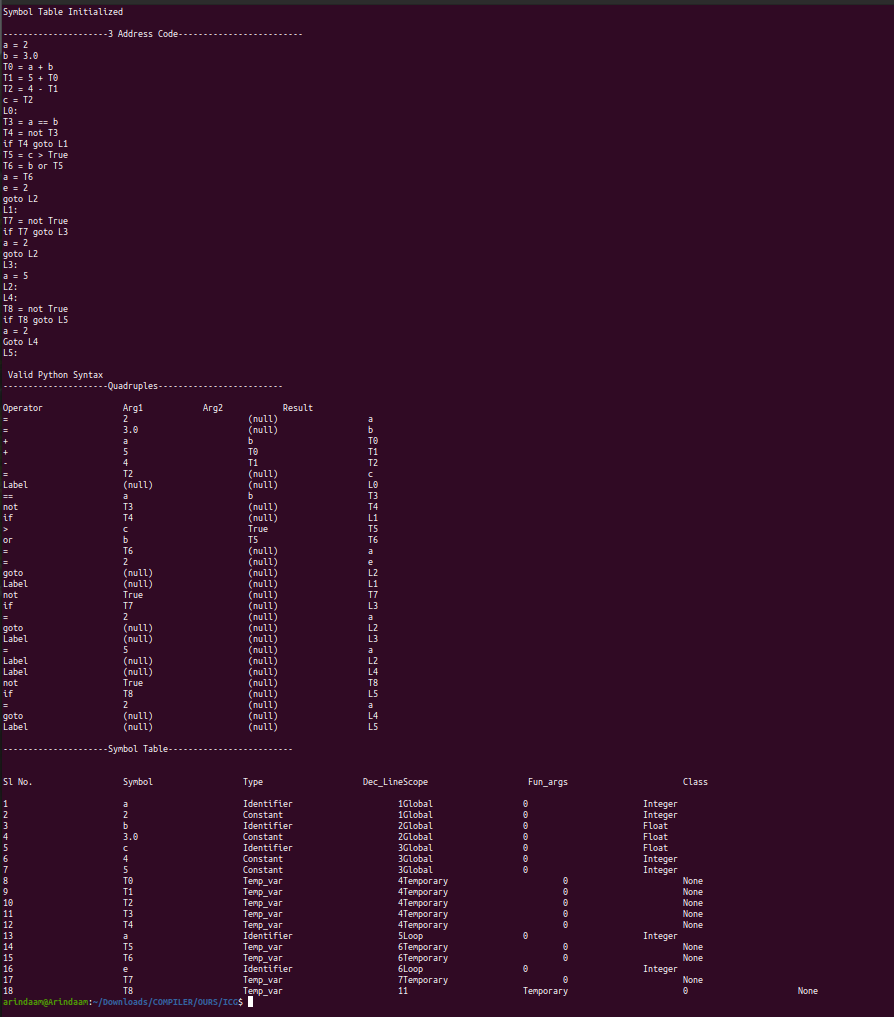
**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Title** | **Page No.** |
|  | INTRODUCTION (Mini-Compiler is built for which language. Provide sample input and output of your project) | **03** |
|  | ARCHITECTURE OF LANGUAGE:   * What all have you handled in terms of syntax and semantics for the chosen language. | **04** |
|  | LITERATURE SURVEY (if any paper referred or link used) | **05** |
|  | CONTEXT FREE GRAMMAR (which you used to implement your project) | **06** |
|  | DESIGN STRATEGY (used to implement the following)   * SYMBOL TABLE CREATION * INTERMEDIATE CODE GENERATION * CODE OPTIMIZATION * ERROR HANDLING | **07** |
|  | IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following):   * SYMBOL TABLE CREATION * INTERMEDIATE CODE GENERATION * CODE OPTIMIZATION * ERROR HANDLING * Provide instructions on how to build and run your program. | **09** |
|  | RESULTS AND possible shortcomings of your Mini-Compiler | **10** |
|  | SNAPSHOTS (of different outputs) | **11** |
|  | CONCLUSIONS | **19** |
|  | FURTHER ENHANCEMENTS | **19** |
| **REFERENCES/BIBLIOGRAPHY** | |  |

**Introduction**

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





**ARCHITECTURE OF THE LANGUAGE**

Python has a very flexible syntax and we have tried to incorporate as much as possible from our experience of using python into the grammar. We have not taken care of semicolons, so a semicolon will result in an error while parsing. All lines of code terminate upon seeing a newline character. We have taken care of the following:

* If-Elif-Else and While constructs
* Print Statements
* All arithmetic operators
* All Boolean operators except standalone ‘!’ (“!=” is taken care of)
* Single Line Comments (#)
* Multi Comment Line (""")
* Integers
* Floating Point Number

**Literature Survey**

Bison Documentation: [https://www.gnu.org/software/bison/manual](https://www.gnu.org/software/bison/manual/)/

Javatpoint: <https://www.javatpoint.com/lex>

**The Context Free Grammar**

Grammar Expression

StartDebugger StartParse;

constant T\_Number T\_String ;

term T\_ID| constant ;

StartParse finalStatements StartParse| T\_EndOfFile

Expressions arith\_exp | bool\_exp ;

basic\_stmt break\_stmt | import\_stmt | assign\_stmt | Expressions | print\_stmt;

arith\_exp term

| arith\_exp T\_Arop arith\_exp

|T\_OP arith\_exp T\_CP ;

bool\_exp T\_Binary

| T\_OP bool\_exp T\_CP

| arith\_exp T\_Relop arith\_exp

| bool\_exp T\_andor bool\_exp

| T\_Not bool\_exp

| Expressions T\_eq Expressions;

import\_stmt T\_Import T\_ID ;

break\_stmt T\_Break ;

assign\_stmt T\_ID T\_EQL value;

value Expressions|func\_call

print\_stmt T\_Print T\_OP Recur T\_CP;

finalStatements basic\_stmt T\_NL | cmpd\_stmt | func\_def| func\_call ;

Recur Expressions Recur| T\_Comma Expressions Recur|;

cmpd\_stmt if\_stmt | while\_stmt ;

if\_stmt T\_If bool\_exp T\_Cln T\_NL T\_id finalStatements suite elif\_stmts ;

elif\_stmts T\_Elif bool\_exp T\_Cln T\_NL T\_id

finalStatements suite elif\_stmts |T\_Else T\_Cln T\_NL T\_id finalStatements suite try|finalStatements;

try: finalStatements|;

suite: T\_ND finalStatements suite|T\_DD

while\_stmt T\_While bool\_exp T\_Cln T\_NL T\_id finalStatements suite;

args\_list T\_ID T\_Comma args\_list | T\_ID | ;

func\_def T\_Def T\_ID T\_OP args\_list T\_CP T\_Cln T\_NL T\_id finalStatements suite;

func\_call T\_ID T\_OP args\_list T\_CP T\_NL;

**Design Strategy**

**Symbol Table Generation:**

The symbol table is stored in a structure named record which is represented as below:

typedef struct record

    {

        char \*type;

        char \*name;

        int decLineNo;

        char\* scope;

        int fun;

        char\* classtype;

    } record;

The function void init() will initialise an instance of record of size 500\*sizeof(record) and prints:

printf("Symbol Table Initialized\n");

The symbol table is created using the insert\_Record() function. This function takes type, name, scope and class as parameters and assigns them to the corresponding values of the instance of record used as symbol table. The function is called in the grammar and the values of the parameters are defined in the calls to the function.

**Intermediate Code Generation:** The ICG is generated using the following strategy:

1. For arithmetic operations, the function codegen() is used, this function pushes all the RHS variables and arithmetic operators into a stack. It then prints the top, top-1 (operator) and top-2 and assigns them to a temporary variable. And these are then popped. This process continues until the stack is empty. The t-2 and t elements of the stack are passed as arguments to the quadruple.
2. For assignment operations, the codegen\_assign() function will print stack[t-2]=stack[t] and then pop the t, t-1 and t-2 element from the stack. In the quadruple generation, the arg2 is set to NULL.
3. The while loop and if statement are handled using the loop\_condition() and end\_block() functions.
4. The loop\_condition() generates the 3AC code for the if-else and while conditions and the label to goto when the condition fails.
5. The end\_block() condition at the end of each if-else-if-else block pops 2 variables from the stack. The first is stored in the pops variable and the second is printed in loop init in next iteration.

**Code Optimisation:** The code optimisation methods we used are:

1. **Constant Folding:** Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the integer literal 2 , but they may also be variables whose values are known at compile time. It is implemented using a dictionary, the quadruple values are split into arg1, arg2, op and res. If the op is an arithmetic or Boolean operator and the type of the arguments are digit, the dict[res] is assigned the evaluated value of the expression and it is appended to a constant folded list.
2. **Common Subexpression Evaluation:** common subexpression elimination (CSE) is a [compiler optimization](https://en.wikipedia.org/wiki/Compiler_optimization) that searches for instances of identical [expressions](https://en.wikipedia.org/wiki/Expression_(computer_science)) (i.e., they all evaluate to the same value), and analyzes whether it is worthwhile replacing them with a single variable holding the computed value. We iterate through the constant folded list. When we encounter an arithmetic or Boolean operation, we check from the start of the list to the ith position. If a match is found, the value of res is updated to the one that matched the expression before.
3. **Constant Propagation:** Constant propagation is the process of substituting the values of known constants in expressions at compile time. We have implemented it using the dictionary that was used in constant folding. If the operation is an arithmetic or Boolean operation and the arguments are digits or are present in the dictionary dict\_values, the result is evaluated and assigned to the temporary. In the case of an assignment statement, if arg1 is in the dictionary, the value is replaced with the value in the dictionary.
4. **DeadCode Elimination**: If a label is encountered which evaluates to false, all the following lines of code are skipped until the next label is encountered.

**Error Handling:** The following errors are reported with the line\_no and column\_number,

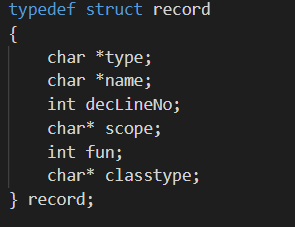
1. Syntax error: When the rules of the grammar are violated.
2. Invalid break: When a break statement is not inside a loop and has no meaning.
3. Variable out of scope: If a variable is used without declaration in scope.
4. String Datatype Error: If strings are operated with numerical or boolean types.
5. Function undeclared: When function is called without declaration
6. Function mismatched arguments: When the number of arguments in function call doesnot match the number of arguments in function definition.

**Implementation Details:**

The project was implemented using lex and bison, the code optimisation was written using a python script.

To Run the code:

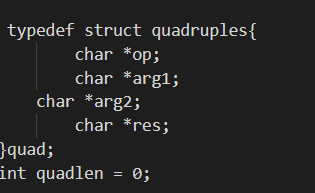
**Implementation of Symbol Table:**

****

An instance of the structure record is used to implement a symbol table.

**Intermediate Code Generation:**

The three-address code is generated using a stack called st.



The quadruples are generated using the following structure.

**Code Optimisation:**

A dictionary called dict\_values is used to store temporaries and variables that might need to be referred. The quad data structure is used to create the ConstantFoldedList, which is referred to by all the other optimisation methods.

**Error Handling:**

Inbuilt structures like YYLTYPE and YYSTYPE and variables yytext, yylineno, yylloc are used to print the position of the lexeme at fault.

Simple counters and appropriate grammar rules are used to detect the errors and yyerror function is used to display the correct error.

**Running the code:**

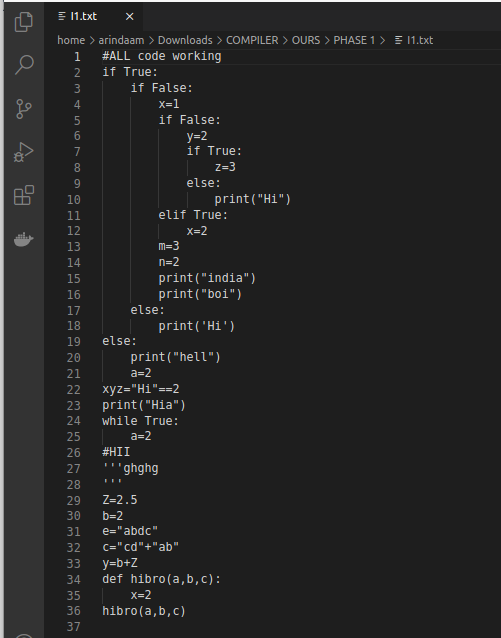
1. lex Scanner.y
2. bison -dy -v Parser.y
3. gcc lex.yy.c y.tab.c -g -ll -lm -w
4. ./a.out < input\_file.txt
5. python codeopt.py

**Results:**

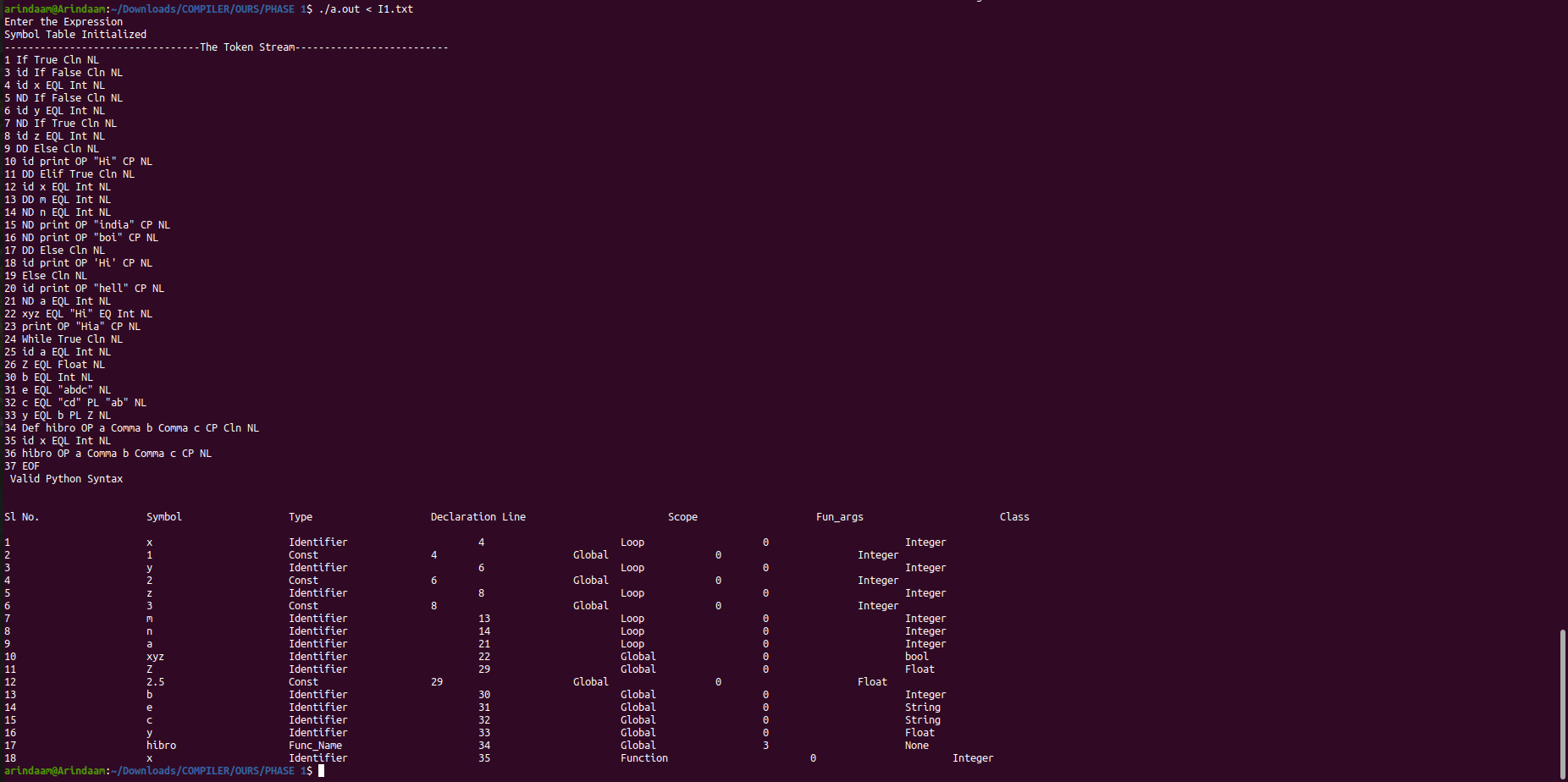
The mini-compiler is able to successfully handle the grammar and constructs mentioned and is also able to successfully generate ICG in 3ac and quadruple form and can perform constant folding, common subexpression elimination, constant propagation and dead-code elimination.

**SnapShots:**

1. **Input**

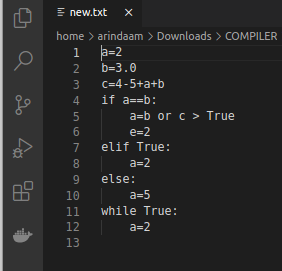
****

**Output:**

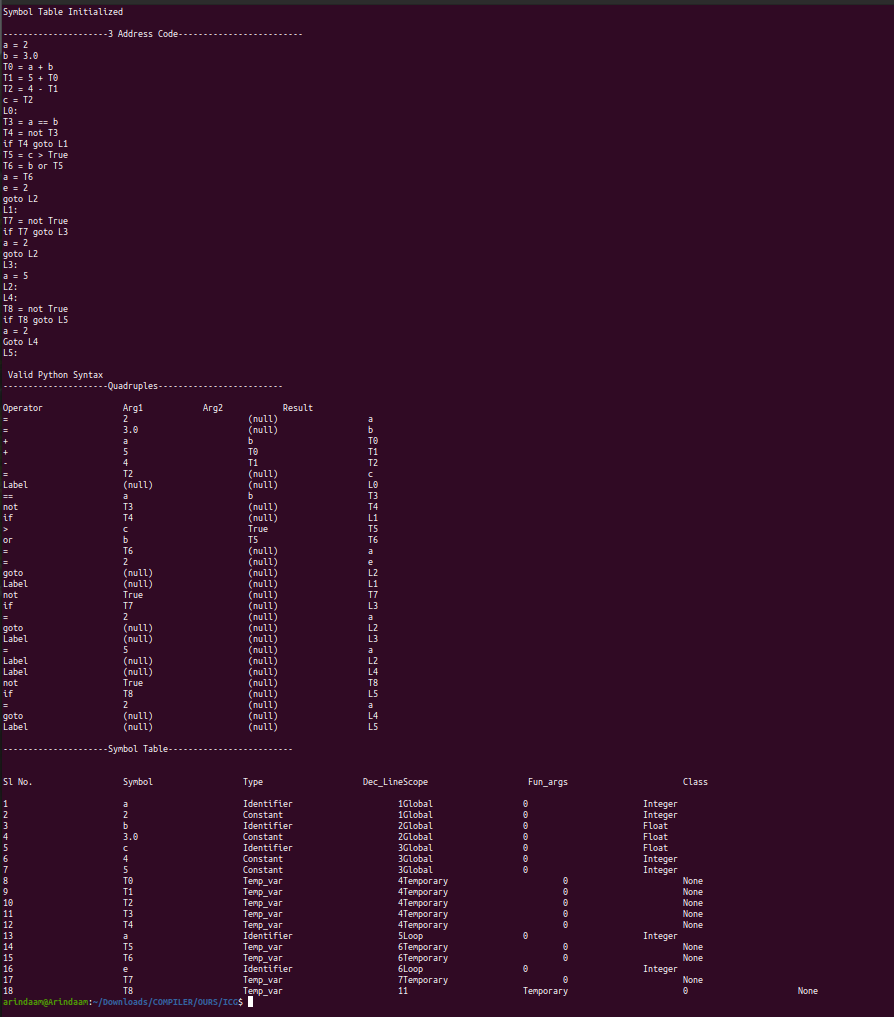
****

**Phase 2: ICG Generation**

**Input**

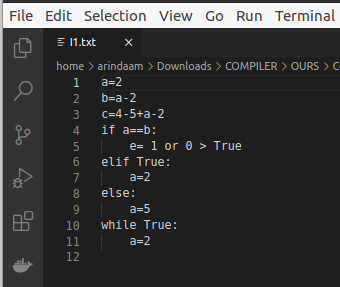
****

**Output:**

****

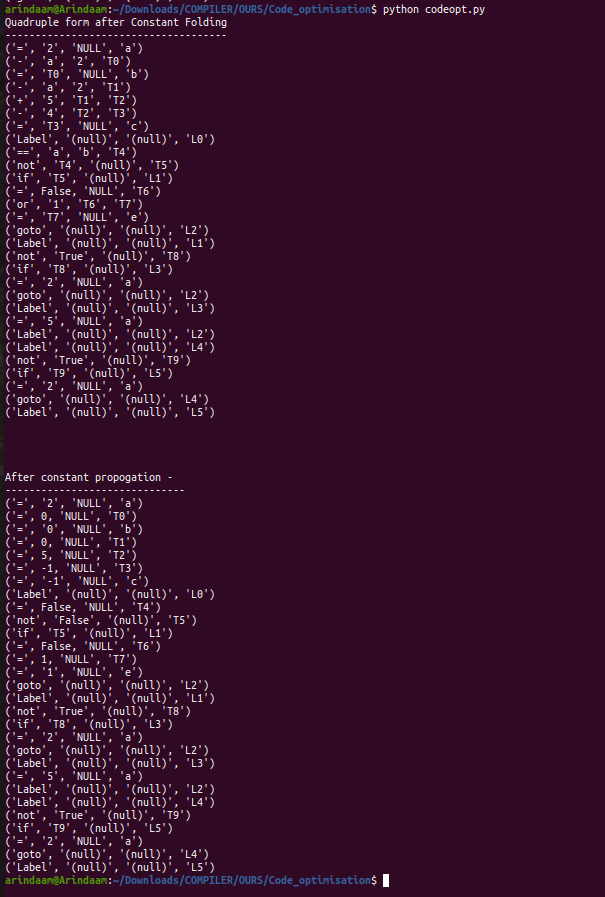
**Phase 3: Code Optimisation**

**Input:**

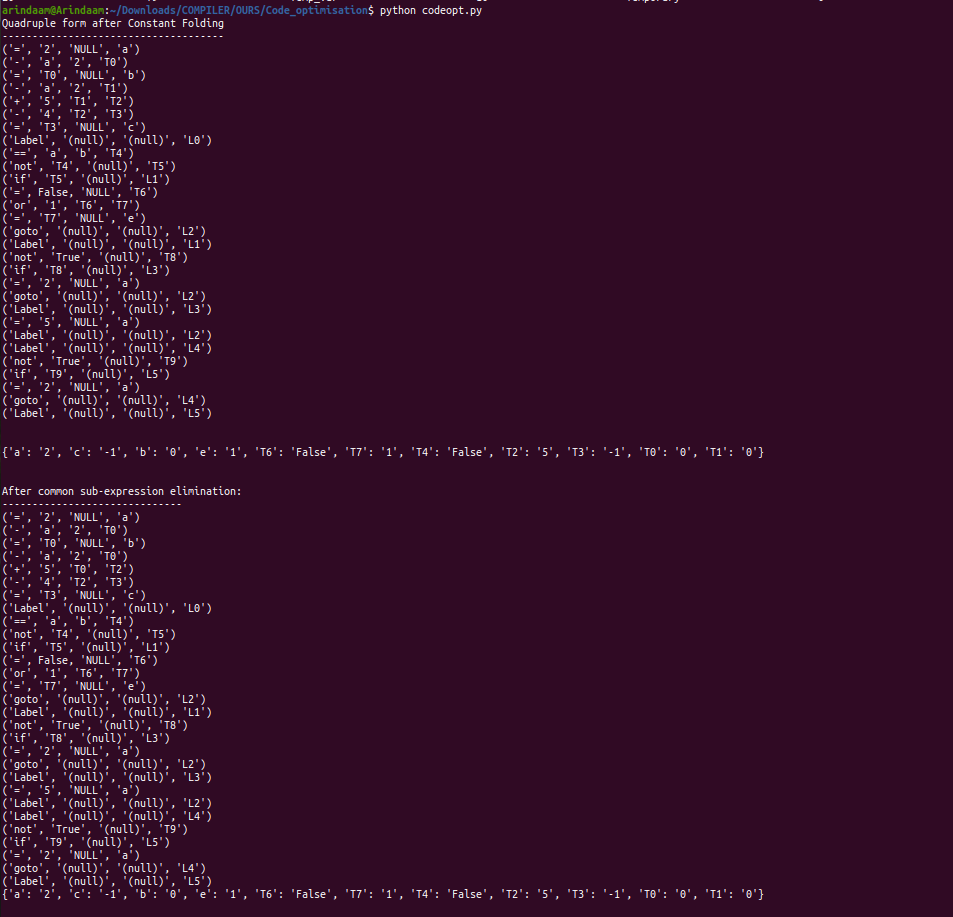
****

**Output:**

1. **Constant Folding and Propagation**

****

**2 Constant Folding and common sub-expression elimination**

****

**3. Constant Folding, Constant Propagation and dead code elimination.**

****

**Conclusion:**

Our mini-compiler is able to successfully handle all the listed grammar, generate ICG and perform code optimisation.

**Future Enhancements**

Our project can be enhanced to handle the for and do while constructs, data structures like arrays, lists and dictionaries may be handled as well.