Report on

"Mini-Compiler for 'for' and 'while' constructs"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by **TEAM NO. 14*:***

Akash K	**PES1201801760**
Rajdeep Sengupta	**PES1201800144**
L. Anvesh reddy	**PES1201801299**

Under the guidance of

Prof. Madhura V.
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

INTRODUCTION

A mini-compiler is created for this course Compiler Design UE18CS351. The constructs allotted to us are ‘for’ and ‘while’. The compiler is designed for Python language version 3 or above. The compiler handles all the basic assignment statements, relational operations, comments, for loops and while loops.

Sample Input:

```
a=10
b=a
c=b+1

# this is a comment line

for i in range(10):
   x=20
   y='hello'

x=3
```

Sample Output:

The output of the above code contains the lexical tokens, symbol table, updated symbol table, the three address code before and after optimization. Also, the three address code is written into a separate file.

ARCHITECTURE OF LANGUAGE

The library used in this project is PLY (python lex yacc). This library makes the implementation of a compiler easy with its inbuilt tools. It has some very important and useful features:

- implemented in python
- uses LR parsing which is suited for larger grammars
- supports empty productions, precedence rules, error recovery and support for ambiguous grammars

Two files are written namely lex.py and yacc.py.

Lex file

The lex file contains the generation of tokens and removing the comment lines from the code. It also has the initial creation of the symbol table.

Yacc file

The yacc file contains the defined grammars and semantic rules. The symbol table is updated thereafter. Also, it has three address code and quadruple table generation.

All the optimizations are also done in the yacc file. The optimizations contain copy propagation, constant folding, constant propagation and dead-code elimination.

CONTEXT FREE GRAMMAR

The following includes only some of the grammar rules, rest is present in the code

assignments

```
'''expression : ID EQUAL expression
                    | ID EQUAL EMP_LIST
                    | ID EQUAL EMP_TUPLE
                    | ID EQUAL EMP_SET
                    | ID EQUAL STR_CONST
                    | ID EQUAL ID
                    '''
```

addition expressions

```
'''expression : ID PLUS EQUAL term
                    | factor PLUS factor
                    | expression PLUS term
                    | ID PLUS factor
                    | factor PLUS ID
                    | ID PLUS ID
                    | expression PLUS ID
                    | ID PLUS expression
                    '''
'expression : term'
```

term and factor

```
'''term : term MULTIPLY factor
            | ID MULTIPLY ID
            | ID MULTIPLY factor
            | factor MULTIPLY ID
            | ID MULTIPLY term
            | term MULTIPLY ID
            | factor MULTIPLY factor
    '''
'term : term DIVIDE factor'
'term : factor'
'''factor : INT
            | FLOAT'''
'factor : PARANOPEN expression PARANCLOSE'
```

loops

```
'''expression : whileloop
                    | forloop
                    | forloop2
                    | whileloop2'''
```

Here in the above grammar, forloop2 refers to nested for loop(2 levels) and whileloop2 refers to nested while loop(2 levels)

while loop

```
'''whileloop :  WHILE PARANOPEN ID EQUAL factor PARANCLOSE COLON
                | WHILE PARANOPEN ID EQUAL STR_CONST PARANCLOSE COLON
                | WHILE PARANOPEN ID EQUAL ID PARANCLOSE COLON
                | WHILE PARANOPEN ID GREATER factor PARANCLOSE COLON
                | WHILE PARANOPEN ID GREATER STR_CONST PARANCLOSE COLON
                | WHILE PARANOPEN ID GREATER ID PARANCLOSE COLON
                | WHILE PARANOPEN ID GREATEREQ factor PARANCLOSE COLON
              | WHILE PARANOPEN ID GREATEREQ STR_CONST PARANCLOSE COLON
                | WHILE PARANOPEN ID GREATEREQ ID PARANCLOSE COLON
                | WHILE PARANOPEN ID LESSER factor PARANCLOSE COLON
                | WHILE PARANOPEN ID LESSER STR_CONST PARANCLOSE COLON
                | WHILE PARANOPEN ID LESSER ID PARANCLOSE COLON
                | WHILE PARANOPEN ID LESSEREQ factor PARANCLOSE COLON
               | WHILE PARANOPEN ID LESSEREQ STR_CONST PARANCLOSE COLON
                | WHILE PARANOPEN ID LESSEREQ ID PARANCLOSE COLON
                | WHILE BOOL COLON
                | WHILE STR_CONST COLON
                | WHILE factor COLON
                | WHILE ID COLON'''
```

The above grammar is for outer while loop. Grammar for nested while loop is present in the code file

for loop

```
'''forloop : FOR ID IN RANGE PARANOPEN term PARANCLOSE COLON
                    | FOR ID IN RANGE PARANOPEN LEN PARANOPEN ID
PARANCLOSE PARANCLOSE COLON
                    | FOR ID IN STR_CONST COLON
                    | FOR ID IN ID COLON
                    | FOR ID IN PARANOPEN STR_CONST PARANCLOSE COLON
                 '''
```

comment

```
'''expression : HASH expression
                    | HASH ID
                    | HASH STR_CONST'''
```

print statement

```
'''expression : PRINT PARANOPEN STR_CONST PARANCLOSE
                    | PRINT PARANOPEN ID PARANCLOSE
                    | PRINT PARANOPEN term PARANCLOSE'''
```

DESIGN STRATEGY

Symbol Table:

<***identifier, type, line number, value, scope***>

Variables are added initially in the lex file and are further updated in the yacc file after syntax and semantic checking.

Quadruple Table:

<***result, operator, argument1, argument2***>

The three address code is generated while parsing the grammar. Then this is used further to generate the quadruple table. This table is updated after optimizations.

IMPLEMENTATION DETAILS

Symbol Table:

In the lex.py file, functions are defined with their regular expressions. If they match, then the lexer value is stored and returned. Using these tokens, the initial symbol table is created. For each operation, a function is written, for eg. assignment, addition, subtraction etc., and the grammar is defined for each function. Inside the function, the updated value is computed and updated in the symbol table.

The scope is stored in an incremental way. This means whenever a loop starts, the counter value is incremented and all the values defined in that loop have counter value.

A dictionary is maintained with keys as scope which has values as list of dictionaries as variable names and values as their type and line number.

Three Address Code:

For each function in yacc.py file, the grammar is defined and a **tac** variable is appended with four fields **(Result, Operator, Argument1, Argument2)**. Later on at the end of the code, this variable is parsed and the formatted three address code is written into a new text file.

Optimizations:

1. Copy propagation:

   CODE:

   ```
   a=10
   x=a
   y=x+2
   ```

   OPTIMIZED CODE:

   ```
   y=a+10
   ```

2. Constant folding

   CODE:

   ```
   a=10+2*3
   ```

   OPTIMIZED CODE:

   ```
   a=16
   ```

3. Constant propagation

   CODE:

   ```
   a=10
   b=a+20
   ```

   OPTIMIZED CODE:

   ```
   b=10+20
   ```

4. Dead-code elimination

   CODE:

```
a=10
b=20
c=30
d=a+c
```

   OPTIMIZED CODE:

```
a=10
c=30
d=a+c
```

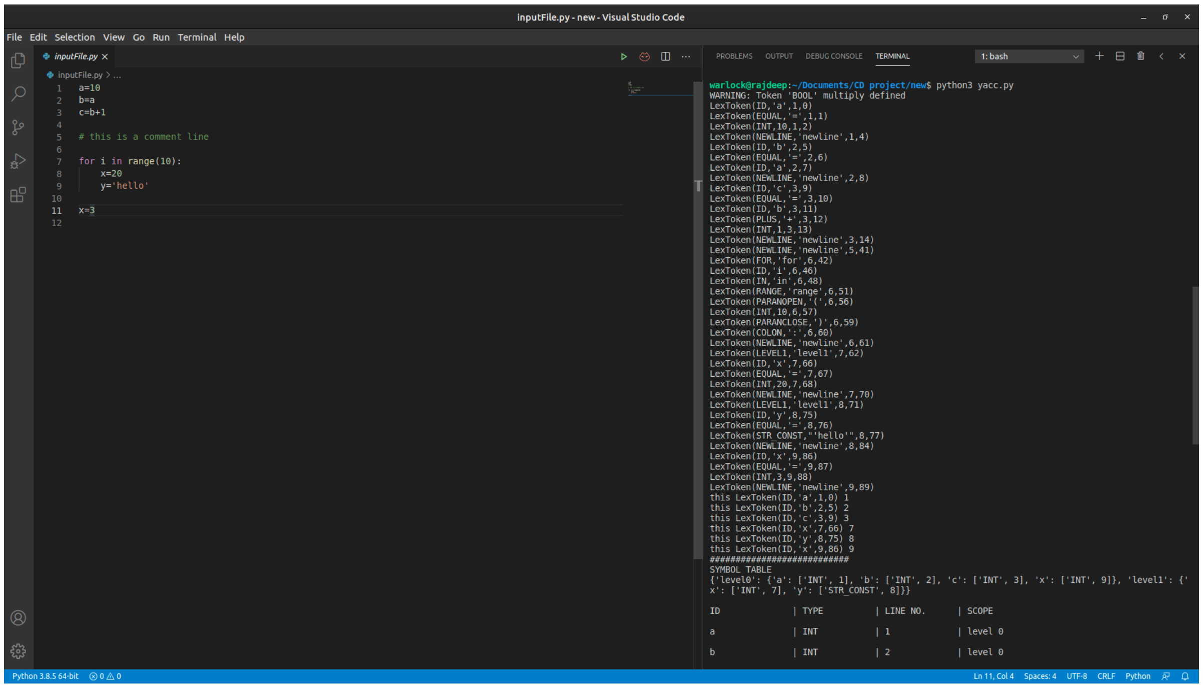
RESULTS

The following jobs are performed successfully:

- syntax checking
- generation of symbol table while lexical analysis
- updation of symbol table after parsing and checking semantic rules
- three address code generation
- generation of quadruple table (result, operator, arg1, arg2)
- generation of optimized three address code

Some important screenshots can be found below in the Snapshots section.

SNAPSHOTS

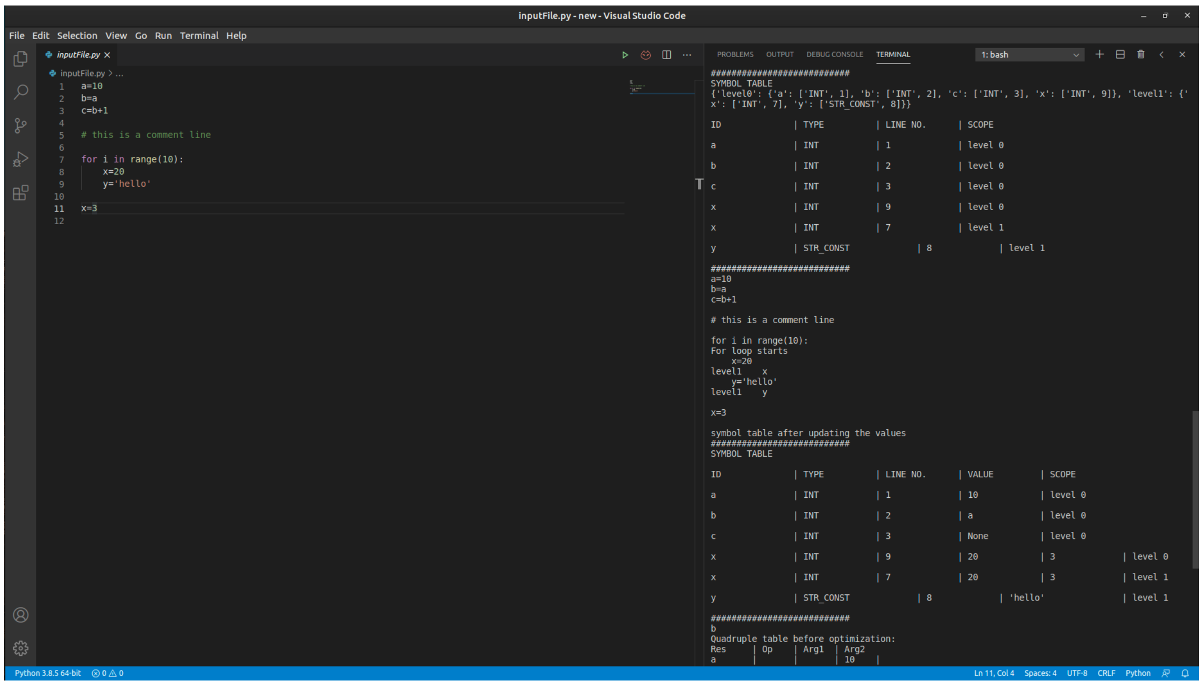


inputFile.py - new - Visual Studio Code

```python
a=10
b=a
c=b+1

# this is a comment line

for i in range(10):
    x=20
    y='hello'

x=3

```

```
warlock@rajdeep:~/Documents/CD project/new$ python3 yacc.py
WARNING: Token 'BOOL' multiply defined
LexToken(ID,'a',1,0)
LexToken(EQUAL,'=',1,1)
LexToken(INT,10,1,2)
LexToken(NEWLINE,'newline',1,4)
LexToken(ID,'b',2,5)
LexToken(EQUAL,'=',2,6)
LexToken(ID,'a',2,7)
LexToken(NEWLINE,'newline',2,8)
LexToken(ID,'c',3,9)
LexToken(EQUAL,'=',3,10)
LexToken(ID,'b',3,11)
LexToken(PLUS,'+',3,12)
LexToken(INT,1,3,13)
LexToken(NEWLINE,'newline',3,14)
LexToken(NEWLINE,'newline',5,41)
LexToken(FOR,'for',6,42)
LexToken(ID,'i',6,46)
LexToken(IN,'in',6,48)
LexToken(RANGE,'range',6,51)
LexToken(PARANOPEN,'(',6,56)
LexToken(INT,10,6,57)
LexToken(PARANCLOSE,')',6,59)
LexToken(COLON,':',6,60)
LexToken(NEWLINE,'newline',6,61)
LexToken(LEVEL1,'level1',7,62)
LexToken(ID,'x',7,66)
LexToken(EQUAL,'=',7,67)
LexToken(INT,20,7,68)
LexToken(NEWLINE,'newline',7,70)
LexToken(LEVEL1,'level1',8,71)
LexToken(ID,'y',8,75)
LexToken(EQUAL,'=',8,76)
LexToken(STR_CONST,"'hello'",8,77)
LexToken(NEWLINE,'newline',8,84)
LexToken(ID,'x',9,86)
LexToken(EQUAL,'=',9,87)
LexToken(INT,3,9,88)
LexToken(NEWLINE,'newline',9,89)
this LexToken(ID,'a',1,0) 1
this LexToken(ID,'b',2,5) 2
this LexToken(ID,'c',3,9) 3
this LexToken(ID,'x',7,66) 7
this LexToken(ID,'y',8,75) 8
this LexToken(ID,'x',9,86) 9
###########################
SYMBOL TABLE
{'level0': {'a': ['INT', 1], 'b': ['INT', 2], 'c': ['INT', 3], 'x': ['INT', 9]}, 'level1': {'
x': ['INT', 7], 'y': ['STR_CONST', 8]}}

ID              | TYPE          | LINE NO.      | SCOPE

a               | INT           | 1             | level 0

b               | INT           | 2             | level 0
```

Screenshot 1: Generation of stream of tokens for the input file

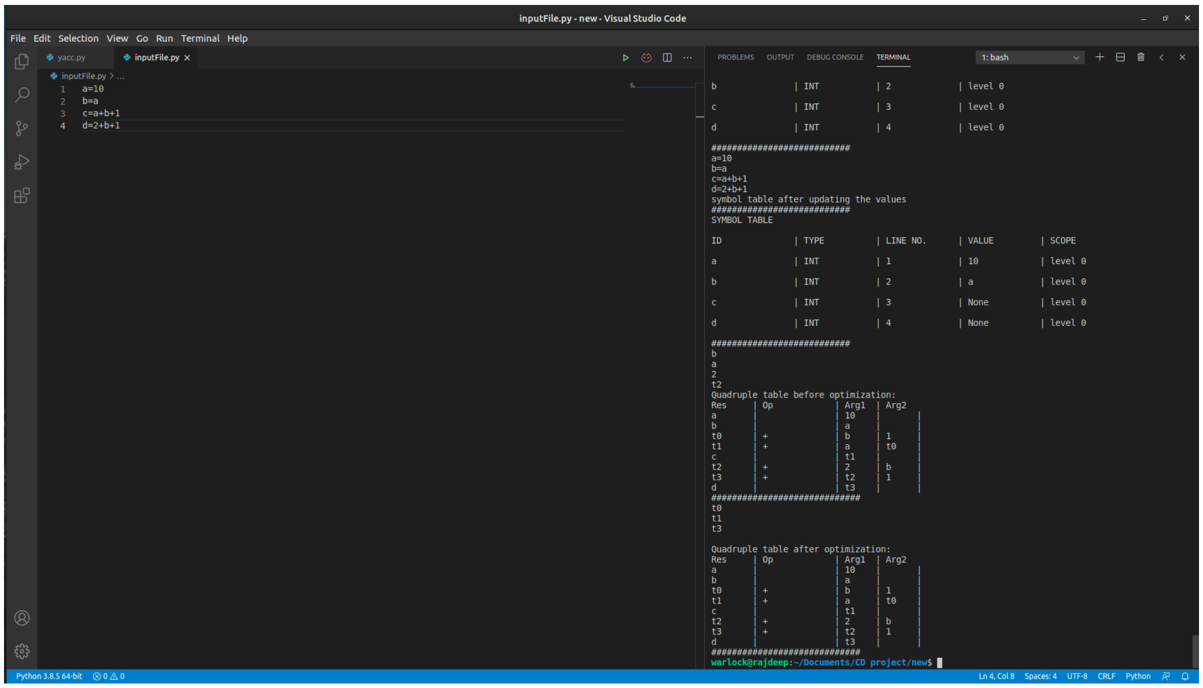


inputFile.py - new - Visual Studio Code

```python
a=10
b=a
c=b+1

# this is a comment line

for i in range(10):
    x=20
    y='hello'

x=3

```

```
###########################
SYMBOL TABLE
{'level0': {'a': ['INT', 1], 'b': ['INT', 2], 'c': ['INT', 3], 'x': ['INT', 9]}, 'level1': {'
x': ['INT', 7], 'y': ['STR_CONST', 8]}}

ID              | TYPE          | LINE NO.      | SCOPE

a               | INT           | 1             | level 0

b               | INT           | 2             | level 0

c               | INT           | 3             | level 0

x               | INT           | 9             | level 0

x               | INT           | 7             | level 1

y               | STR_CONST             | 8             | level 1

###########################
a=10
b=a
c=b+1

# this is a comment line

for i in range(10):
For loop starts
    x=20
level1    x
    y='hello'
level1    y

x=3

symbol table after updating the values
###########################
SYMBOL TABLE

ID              | TYPE          | LINE NO.      | VALUE         | SCOPE

a               | INT           | 1             | 10            | level 0

b               | INT           | 2             | a             | level 0

c               | INT           | 3             | None          | level 0

x               | INT           | 9             | 20            | 3             | level 0

x               | INT           | 7             | 20            | 3             | level 1

y               | STR_CONST             | 8             | 'hello'               | level 1

###########################
b
Quadruple table before optimization:
Res     | Op    | Arg1  | Arg2
a       |       |       | 10    |
```

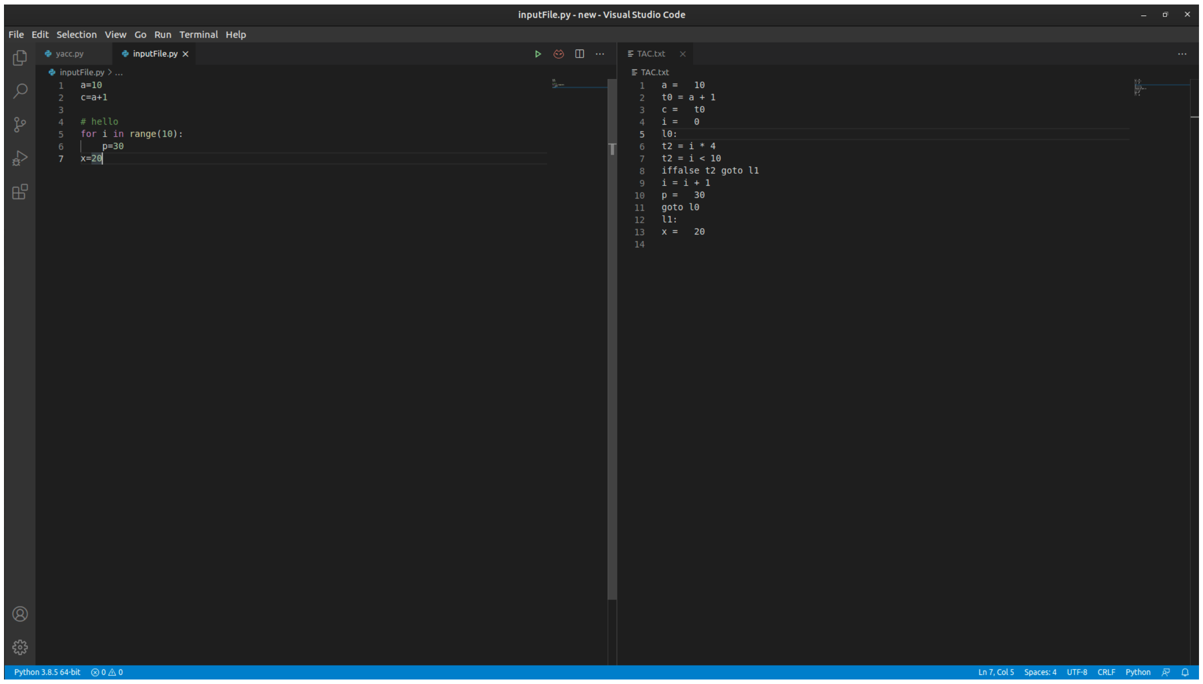
Screenshot 2: Initial symbol table and updated symbol table



```
a=10
b=a
c=a+b+1
d=2+b+1
```

```
b               | INT           | 2             | level 0

c               | INT           | 3             | level 0

d               | INT           | 4             | level 0

###########################
a=10
b=a
c=a+b+1
d=2+b+1
symbol table after updating the values
###########################
SYMBOL TABLE

ID              | TYPE          | LINE NO.      | VALUE         | SCOPE

a               | INT           | 1             | 10            | level 0

b               | INT           | 2             | a             | level 0

c               | INT           | 3             | None          | level 0

d               | INT           | 4             | None          | level 0

###########################
b
a
2
t2
Quadruple table before optimization:
Res     | Op            | Arg1  | Arg2
a       |               | 10    |       |
b       |               | a     |       |
t0      | +             | b     | 1     |
t1      | +             | a     | t0    |
c       |               | t1    |       |
t2      | +             | 2     | b     |
t3      | +             | t2    | 1     |
d       |               | t3    |       |
#############################
t0
t1
t3

Quadruple table after optimization:
Res     | Op            | Arg1  | Arg2
a       |               | 10    |       |
b       |               | a     |       |
t0      | +             | b     | 1     |
t1      | +             | a     | t0    |
c       |               | t1    |       |
t2      | +             | 2     | b     |
t3      | +             | t2    | 1     |
d       |               | t3    |       |
#############################
warlock@rajdeep:~/Documents/CD project/new$
```

Screenshot 3: Initial quadruple table and after optimization

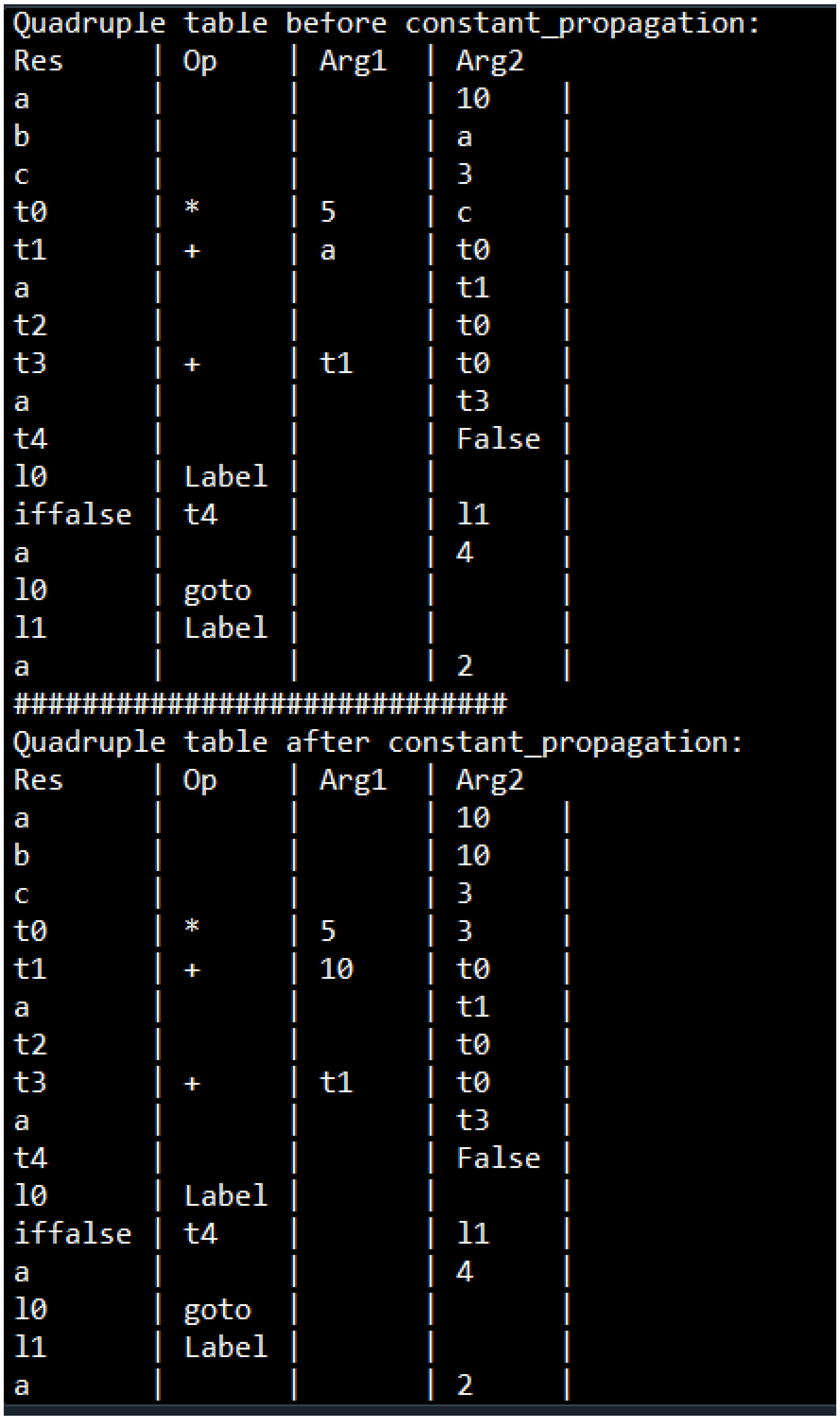


```
a=10
c=a+1

# hello
for i in range(10):
    p=30
x=20
```

```
a =   10
t0 = a + 1
c =   t0
i =   0
l0:
t2 = i * 4
t2 = i < 10
iffalse t2 goto l1
i = i + 1
p =   30
goto l0
l1:
x =   20
```

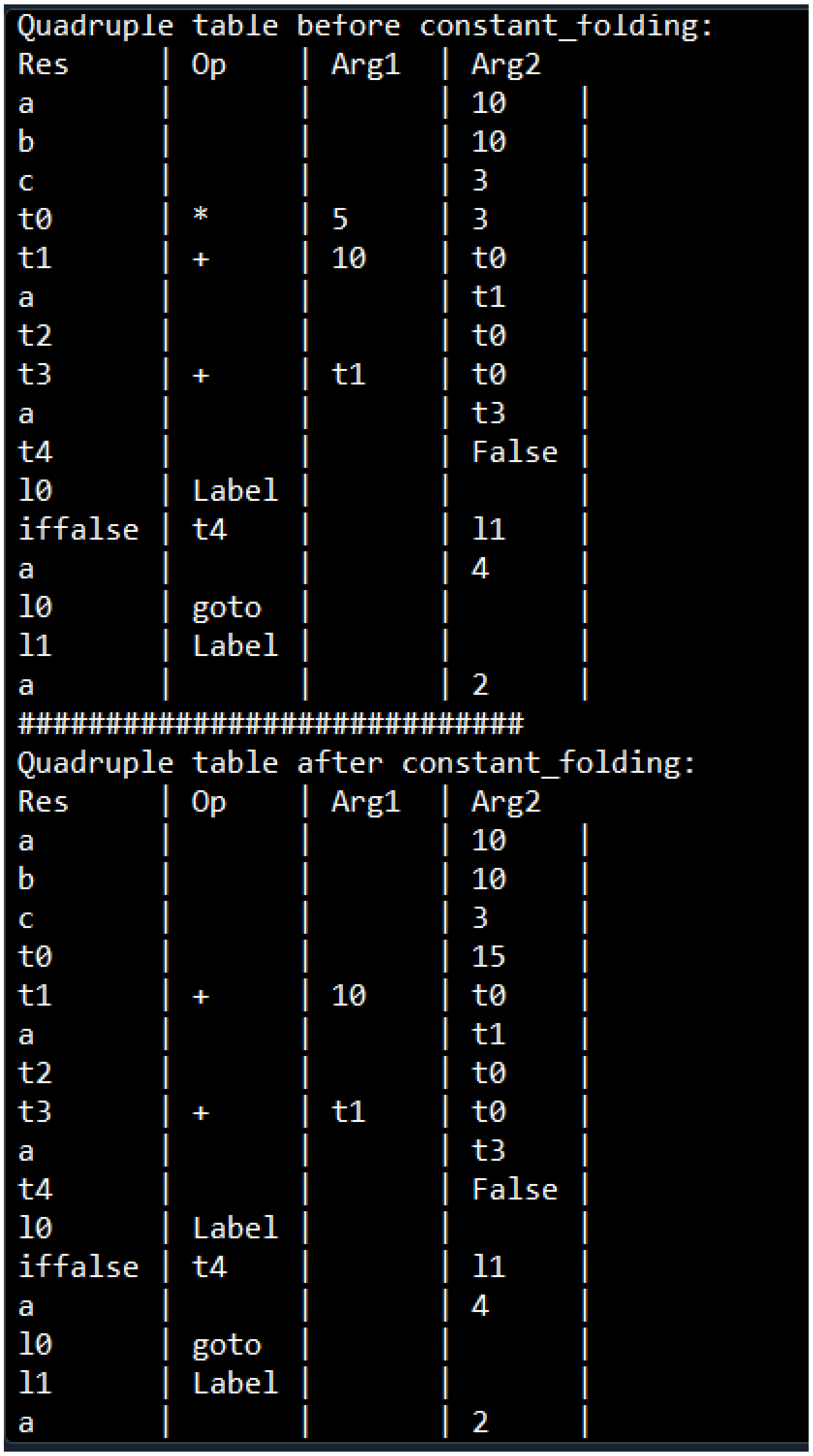
Screenshot 4: Three address code written into a separate text file

BELOW SCREENSHOTS EXPLAIN EACH OPTIMIZATION CLEARLY



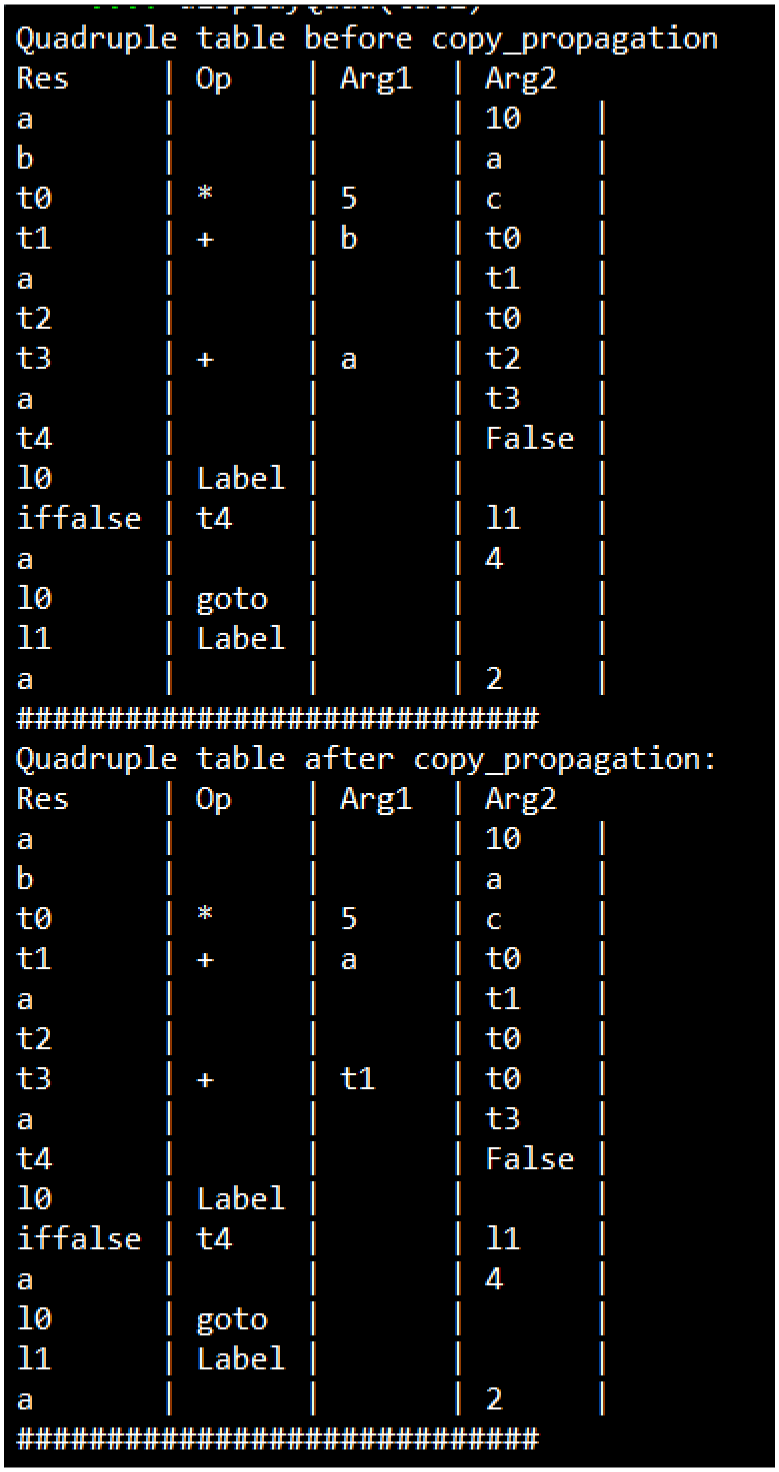
```
Quadruple table before constant_propagation:
Res      | Op    | Arg1  | Arg2
a        |       |       | 10    |
b        |       |       | a     |
c        |       |       | 3     |
t0       | *     | 5     | c     |
t1       | +     | a     | t0    |
a        |       |       | t1    |
t2       |       |       | t0    |
t3       | +     | t1    | t0    |
a        |       |       | t3    |
t4       |       |       | False |
l0       | Label |       |       |
iffalse  | t4    |       | l1    |
a        |       |       | 4     |
l0       | goto  |       |       |
l1       | Label |       |       |
a        |       |       | 2     |
##############################
Quadruple table after constant_propagation:
Res      | Op    | Arg1  | Arg2
a        |       |       | 10    |
b        |       |       | 10    |
c        |       |       | 3     |
t0       | *     | 5     | 3     |
t1       | +     | 10    | t0    |
a        |       |       | t1    |
t2       |       |       | t0    |
t3       | +     | t1    | t0    |
a        |       |       | t3    |
t4       |       |       | False |
l0       | Label |       |       |
iffalse  | t4    |       | l1    |
a        |       |       | 4     |
l0       | goto  |       |       |
l1       | Label |       |       |
a        |       |       | 2     |
```

Screenshot 5: Constant propagation



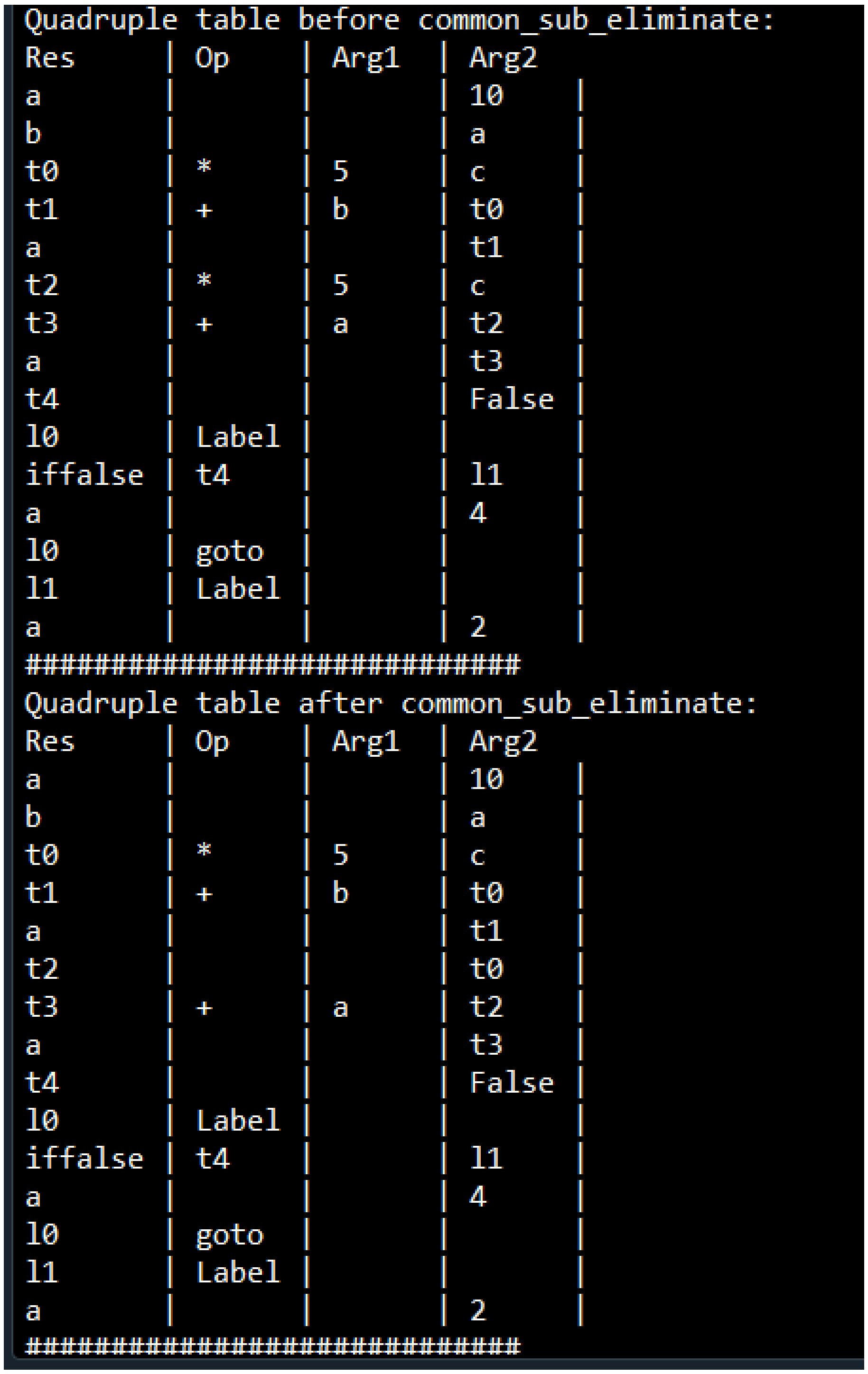
```
Quadruple table before constant_folding:
Res      | Op    | Arg1  | Arg2
a        |       |       | 10    |
b        |       |       | 10    |
c        |       |       | 3     |
t0       | *     | 5     | 3     |
t1       | +     | 10    | t0    |
a        |       |       | t1    |
t2       |       |       | t0    |
t3       | +     | t1    | t0    |
a        |       |       | t3    |
t4       |       |       | False |
l0       | Label |       |       |
iffalse  | t4    |       | l1    |
a        |       |       | 4     |
l0       | goto  |       |       |
l1       | Label |       |       |
a        |       |       | 2     |
##############################
Quadruple table after constant_folding:
Res      | Op    | Arg1  | Arg2
a        |       |       | 10    |
b        |       |       | 10    |
c        |       |       | 3     |
t0       |       |       | 15    |
t1       | +     | 10    | t0    |
a        |       |       | t1    |
t2       |       |       | t0    |
t3       | +     | t1    | t0    |
a        |       |       | t3    |
t4       |       |       | False |
l0       | Label |       |       |
iffalse  | t4    |       | l1    |
a        |       |       | 4     |
l0       | goto  |       |       |
l1       | Label |       |       |
a        |       |       | 2     |
```

Screenshot 6: Constant folding



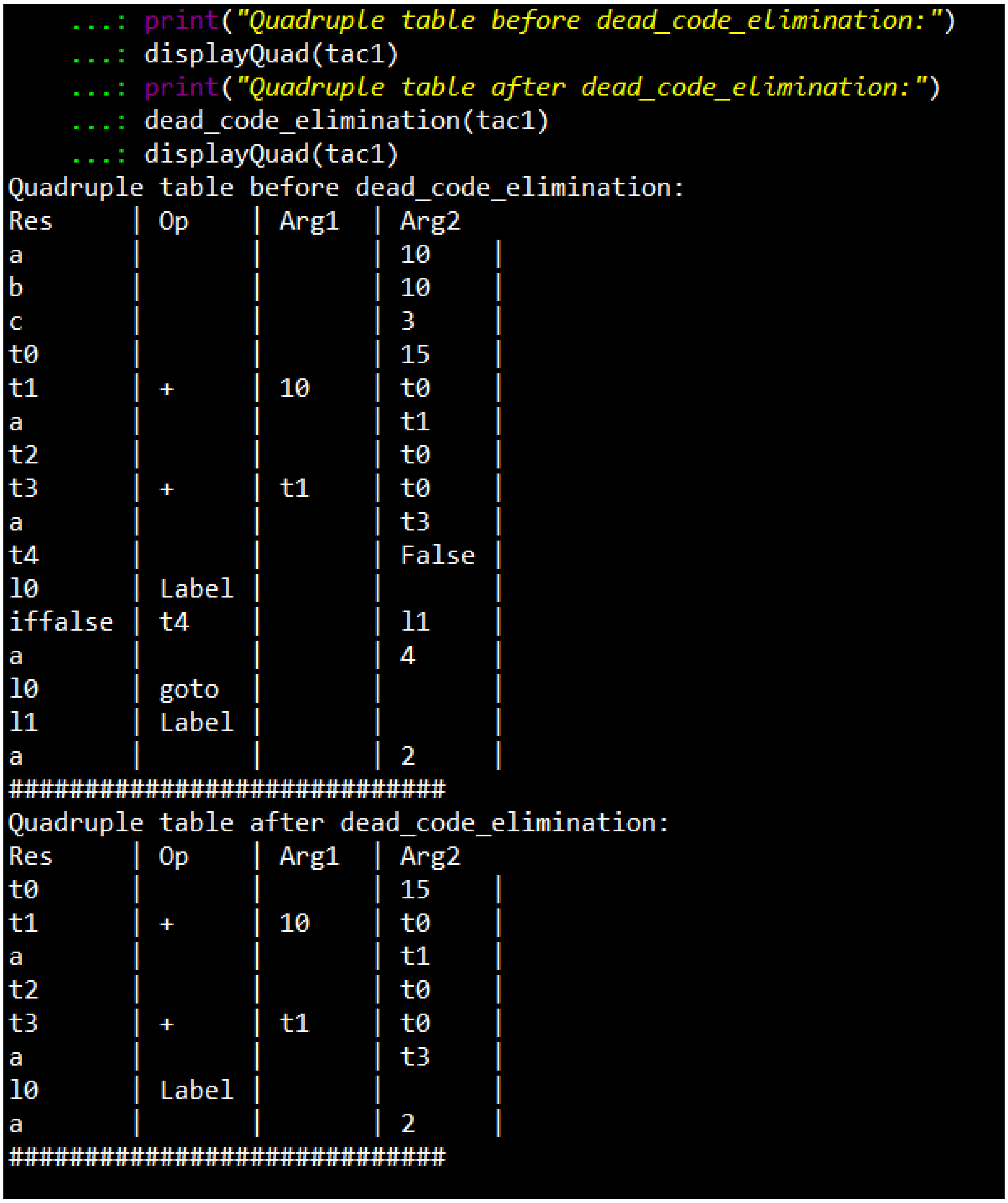
```
Quadruple table before copy_propagation
Res      | Op    | Arg1  | Arg2
a        |       |       | 10    |
b        |       |       | a     |
t0       | *     | 5     | c     |
t1       | +     | b     | t0    |
a        |       |       | t1    |
t2       |       |       | t0    |
t3       | +     | a     | t2    |
a        |       |       | t3    |
t4       |       |       | False |
l0       | Label |       |       |
iffalse  | t4    |       | l1    |
a        |       |       | 4     |
l0       | goto  |       |       |
l1       | Label |       |       |
a        |       |       | 2     |
##############################
Quadruple table after copy_propagation:
Res      | Op    | Arg1  | Arg2
a        |       |       | 10    |
b        |       |       | a     |
t0       | *     | 5     | c     |
t1       | +     | a     | t0    |
a        |       |       | t1    |
t2       |       |       | t0    |
t3       | +     | t1    | t0    |
a        |       |       | t3    |
t4       |       |       | False |
l0       | Label |       |       |
iffalse  | t4    |       | l1    |
a        |       |       | 4     |
l0       | goto  |       |       |
l1       | Label |       |       |
a        |       |       | 2     |
##############################
```

Screenshot 7: Copy propagation



```
Quadruple table before common_sub_eliminate:
Res      | Op    | Arg1  | Arg2
a        |       |       | 10    |
b        |       |       | a     |
t0       | *     | 5     | c     |
t1       | +     | b     | t0    |
a        |       |       | t1    |
t2       | *     | 5     | c     |
t3       | +     | a     | t2    |
a        |       |       | t3    |
t4       |       |       | False |
l0       | Label |       |       |
iffalse  | t4    |       | l1    |
a        |       |       | 4     |
l0       | goto  |       |       |
l1       | Label |       |       |
a        |       |       | 2     |
#############################
Quadruple table after common_sub_eliminate:
Res      | Op    | Arg1  | Arg2
a        |       |       | 10    |
b        |       |       | a     |
t0       | *     | 5     | c     |
t1       | +     | b     | t0    |
a        |       |       | t1    |
t2       |       |       | t0    |
t3       | +     | a     | t2    |
a        |       |       | t3    |
t4       |       |       | False |
l0       | Label |       |       |
iffalse  | t4    |       | l1    |
a        |       |       | 4     |
l0       | goto  |       |       |
l1       | Label |       |       |
a        |       |       | 2     |
#############################
```

Screenshot 8: Common subexpression elimination



```
    ...: print("Quadruple table before dead_code_elimination:")
    ...: displayQuad(tac1)
    ...: print("Quadruple table after dead_code_elimination:")
    ...: dead_code_elimination(tac1)
    ...: displayQuad(tac1)
Quadruple table before dead_code_elimination:
Res     | Op    | Arg1  | Arg2
a       |       |       | 10    |
b       |       |       | 10    |
c       |       |       | 3     |
t0      |       |       | 15    |
t1      | +     | 10    | t0    |
a       |       |       | t1    |
t2      |       |       | t0    |
t3      | +     | t1    | t0    |
a       |       |       | t3    |
t4      |       |       | False |
l0      | Label |       |       |
iffalse | t4    |       | l1    |
a       |       |       | 4     |
l0      | goto  |       |       |
l1      | Label |       |       |
a       |       |       | 2     |
#############################
Quadruple table after dead_code_elimination:
Res     | Op    | Arg1  | Arg2
t0      |       |       | 15    |
t1      | +     | 10    | t0    |
a       |       |       | t1    |
t2      |       |       | t0    |
t3      | +     | t1    | t0    |
a       |       |       | t3    |
l0      | Label |       |       |
a       |       |       | 2     |
#############################
```

Screenshot 9: Dead-code elimination

CONCLUSION

A mini-compiler is created for the constructs '**for'** and '**while'** for python language. This is done using PLY (python lex yacc) library.

FURTHER ENHANCEMENTS

Some of the future work and enhancements may include:

- error handling
- user defined data types
- function calls
- other constructs

REFERENCES

- https://www.dabeaz.com/ply/ply.html
- https://www.dabeaz.com/ply/PLYTalk.pdf
- https://www.dabeaz.com/ply/example.html
- http://www.dalkescientific.com/writings/NBN/parsing_with_ply.html