Report on

"Python Mini Compiler"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Sahith Kurapati	**PES1201800032**
Revanth Babu P N	**PES1201800042**
Sreyans Bothra	**PES1201802012**

Under the guidance of

Preet Kanwal
Associate 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

1. INTRODUCTION

This mini compiler was implemented for the programming language **Python 3.x**. This mini compiler goes through multiple phases taking a python script as an input to the compiler and finally generating optimised 3 address code in the form of quads. The phases involved are lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and code optimisation.

Sample input:

```
 a=1
#Have to specify that this is a comment
b=2
c=3
P=8
print(a)
import d
a=a+1
a==5
a<=5
c=b//5

while True:
        print("hello")
        for i in range(10):
                print(i)
```

Sample Output(Non-optimized):

#	op	A1	A2	Res
1	=	1	-	t1
2	=	t1	-	a
3	=	2	-	t2
4	=	t2	-	b
5	=	3	-	t3
6	=	t3	-	c
7	=	8	-	t4
8	=	t4	-	P
9	PRINT	a	-	-
10	IMPORT	d	-	-
11	=	1	-	t5
12	+	a	t5	t6
13	=	t6	-	a
14	=	5	-	t7
15	==	a	t7	t8
16	=	5	-	t9
17	<=	a	t9	t10
18	=	5	-	t11
19	//	b	t11	t12
20	=	t12	-	c
21	LABEL	-	-	L4
22	IF	TRUE	-	L5
23	GOTO	-	-	L6
24	LABEL	-	-	L5
25	=	"hello"	-	t13
26	PRINT	t13	-	-
27	=	0	-	i
28	LABEL	-	-	L1
29	<	i	10	t14
30	IF	t14	-	L2
31	GOTO	-	-	L3
32	LABEL	-	-	L2
33	PRINT	i	-	-
34	+	i	1	t15
35	=	t15	-	i
36	GOTO	-	-	L1
37	LABEL	-	-	L3
38	GOTO	-	-	L4
39	LABEL	-	-	L6

2. ARCHITECTURE OF LANGUAGE

All the constructs handled in the architecture are:

- For loops
    - Range
    - Lists
    - Strings
    - Break
    - Continue
    - Pass

- While loops:
    - Conditional statements
    - Block code
    - Break
    - Continue
    - Pass

- Indentation and Dedentation

- Arithmetic expressions
    - Addition
    - Subtraction
    - Multiplication
    - Division
    - Modulus
    - Power
    - Floor division

- Boolean expressions
    - And
    - Not
    - Or

- Variables and Values
    - Integer
    - Float
    - String

- Import statements
    - Normal import
    - Import from

- Print statements

- Relational operators

- Single and Multi-line comments

3. LITERATURE SURVEY

The links referred to are:

- Python_Mini_Compiler - [Link]
- Anagha1999 GitHub - [Link]
- https://www.javatpoint.com/lex
- https://drive.google.com/drive/u/0/folders/1QfwpPEQIyLhyrDEoOMYCUSjxzUiaFxuW
- https://www.geeksforgeeks.org/introduction-to-yacc/
- https://www.youtube.com/playlist?list=PLkB3phqR3X43IRqPT0t1iBfmT5bvn198Z

4. CONTEXT FREE GRAMMAR

```
start_maro: start_karo T_EOF

start_karo
        : T_NL start_karo
        | stmt start_karo
        | T_EOF
term
        : T_String

math_term
        : T_ID
        | T_Real
        | T_Integer

stmt
        : simple_stmt
        | compound_stmt

simple_stmt
        : base_stmt

base_stmt
        : pass_stmt
        | delete_stmt
        | import_stmt
        | cobr_stmt
        | assign_stmt
        | print_stmt
        | printable_stmt

pass_stmt
        : T_Pass

delete_stmt
        : T_Del T_ID

import_stmt
        : T_Import T_ID
        | import_from

import_from
        : T_From T_ID T_Import T_ID end_import_from

end_import_from
        : T_Comma T_ID end_import_from
        | %empty

cobr_stmt
        : T_Break
        | T_Continue

assign_stmt
```

```
    : T_ID T_EQ printable_stmt

print_stmt
    : T_Print T_LP printable_stmt T_RP

printable_stmt
    : arith_stmt
    | bool_stmt
    | list_stmt

arith_stmt
    : arith_stmt T_Plus arith_stmt
    | arith_stmt T_Minus arith_stmt
    | arith_stmt T_Star arith_stmt
    | arith_stmt T_Divide arith_stmt
    | arith_stmt T_DDiv arith_stmt
    | arith_stmt T_Mod arith_stmt
    | T_LP arith_stmt T_RP
    | math_term


bool_stmt
    : bool_term T_Or bool_term
    | bool_term T_And bool_term
    | bool_term
    | T_Not bool_stmt
    | T_LP bool_stmt T_RP
    | arith_stmt comp_op arith_stmt

bool_term
    : term
    | T_True
    | T_False

comp_op
    : T_Lt
    | T_Gt
    | T_Deq
    | T_Lte
    | T_Gte

compound_stmt
    : for_stmt
    | while_stmt

for_stmt
    : T_For T_ID T_In range_stmt T_Cln block_code
    | T_For T_ID T_In list_stmt T_Cln block_code
    | T_For T_ID T_In term T_Cln block_code

range_stmt
    : T_Range T_LP T_Integer T_RP
    | T_Range T_LP T_ID T_RP
    | T_Range T_LP T_Integer T_Comma T_Integer T_RP
    | T_Range T_LP T_ID T_Comma T_ID T_RP
```

```
        | T_Range T_LP T_Integer T_Comma T_Integer T_Comma T_Integer T_RP
        | T_Range T_LP T_ID T_Comma T_ID T_Comma T_ID T_RP

list_stmt
        : T_Ls T_Rs
        | T_Ls args T_Rs

args
        : T_String items
        | T_Real items
        | T_Integer items
        | T_ID

items
        : T_Comma T_String items
        | T_Comma T_Real items
        | T_Comma T_Integer items
        | T_Comma T_ID items
        | %empty

while_stmt
        : T_While bool_stmt T_Cln block_code

block_code
        : base_stmt
        | T_NL T_IND stmt repeater T_DED

repeater
        : stmt repeater
        |T_NL stmt repeater
        | %empty
```

5. DESIGN STRATEGY

The Symbol Table creation:

- **Lex Phase:**
  - Scopewise Symbol table wherein each variable defined in a scope is displayed once the code exits out of the scope.
  - Stores the name of the variables, line declared and also the last line used.
- **Parser Phase:**
  - Single Symbol Table for all variables defined in the program.
  - Stores the name, scope (of last use) and also the value propagation.
  - Also stores temporaries. The scope of such variables is -1.

Intermediate Code Generation:

- It is done based on strings and concatenation of the strings.
- The code is generated in a recursive manner and finally string manipulation is done on this concatenated code to produce tab separated quads for the code based on various rules.

Code Optimization:

- Different files with functions for different optimizations.
- Handles:
  - Dead code elimination
  - Common subexpression elimination
  - Loop invariant code
  - Constant folding and constant propagation
- We go through the tsv file (QUADs format) and based on various rules defined in the files, code is optimized using code movement and also code deletion if needed.

Error Handling

- Based on lex rules, lexer sees if the lexeme is correctly written. If not, then it calls the yyerror function.
- Based on grammar rules, if a symbol shouldn't appear at a point in the code, the parser calls yyerror function.
- If an undefined variable is used to assign values to another variable, the parser throws an error and exits out of the program.

6. IMPLEMENTATION DETAILS

Symbol table:

- It is stored as a linear DS.
- Lexer side symbol table is a scope wise array of structs with each struct having 3 fields:
    - Name
    - Line Declared
    - Last Line Used
- Parser Side Symbol Table, an all scope symbol table, is also an array of structs with each of the structs having 3 fields:
    - Name
    - Scope
    - Value

Intermediate Code Generation:

- The parser phase creates a single concatenated Intermediate code for all the lines in the code. It is basically a string.
- Quads are made from this code using string manipulation and stored in a tab separated format in .tsv files.

Code optimization:

- Separate python files for each optimization.
- Take the quads generated from the parser phase and use the python code to perform optimization on them.

Error Handling:

- As soon as an error is seen, the parser/lexer calls the yyerror function and the code exits out.

Instructions to run:

```
lex lex_file.l
yacc -d parser_file.y
gcc lex.yy.c y.tab.c -ll
cat Code_Optimization/tests/test_main_all.py | ./a.out
python3 Code_Optimization/<optimization_file_name>.py
```

Note:

- lex_file.l is the lexer file, parser_file_with_value.y is the parser file, show.py is the test file, optimization_file.py has the optimization python functions.

7. RESULTS AND POSSIBLE SHORTCOMINGS

Results:

- The compiler works well all the way from the initial lexical phase up till the code optimisation phase where the optimized code is represented in the form of QUADs saved in a .tsv file.
- An input python test file goes through all 5 different phases namely, lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and code optimisation.
- All the expressions and scopes are also evaluated and kept track of just like it would in a real python compiler.

Shortcomings:

- Doesn’t work for loops after loops.

8. SNAPSHOTS

Screenshot of outputs:

- **Input Code File named as show.py**

```python
a=1
#Have to specify that this is a comment
b=2
c=3
P=8*5+6
e=8*5+6
print(a)
import d
a=a+1
a<=5
c=b//5

while True:
    print("hello")
    for i in range(10):
        print(i)
```

(Figure 8.1)

- **Commands to run lex and yacc using a shell script file called as work.sh**

```
lex lex_file.l
yacc -d parser_file_with_value.y
gcc lex.yy.c y.tab.c -ll
cat Code_Optimization/tests/show.py | ./a.out
```

(Figure 8.2)

- **The lines matched by the lexer file**

```
1)Matched : a = 1
Single Line Comment NL
3)
Matched : b = 2NL
4)
Matched : c = 3NL
5)
Matched : P = 8* 5+ 6NL
6)
Matched : e = 8* 5+ 6NL
7)
print (Matched : a )NL
8)
import Matched : d NL
9)
Matched : a = Matched : a + 1NL
10)
Matched : a <= 5NL
11)
Matched : c = Matched : b // 5
Empty Line NL
13)
while True : NL
14)print ("hello")
for Matched : i in range (10): NL
16)print (Matched : i )
```

(Figure 8.3)

- **The Scope-Wise symbol table from the lexer side showing the name, line number declared and last line number used of all the variables defined in the input file**

```
Deleting

=====================================

SCOPE: 2 Number of Tabs: 2
Name            |LineDeclared   |LastLineUsed|

======================================

Deleting

=====================================

SCOPE: 1 Number of Tabs: 1
Name            |LineDeclared   |LastLineUsed|
i               |15             |16     |

======================================


EOF-all at scope 0 tab 0

=====================================

SCOPE: 0 Number of Tabs: 0
Name            |LineDeclared   |LastLineUsed|
a               |1              |10     |
b               |3              |11     |
c               |4              |11     |
P               |5              |5      |
e               |6              |6      |
d               |8              |8      |

======================================

END
```

(Figure 8.4)

- **The parser side symbol table showing that the code is valid and hence accepted all the symbols defined, their most recently used scope and also the calculated value of each of the symbols**

```
Accepted Code : Valid

Name    |Scope  |Value
t1      |-1     |1
a       |0      |2
t2      |-1     |2
b       |0      |2
t3      |-1     |3
c       |0      |0
t4      |-1     |8
t5      |-1     |5
t6      |-1     |40
t7      |-1     |6
t8      |-1     |46
P       |0      |46
t9      |-1     |8
t10     |-1     |5
t11     |-1     |40
t12     |-1     |6
t13     |-1     |46
e       |0      |46
d       |0      |0
t14     |-1     |1
t15     |-1     |2
t16     |-1     |5
t17     |-1     |1
t18     |-1     |5
t19     |-1     |0
t20     |-1     |1
i       |1      |0
t21     |-1     |-1
t22     |-1     |-1
```

(Figure 8.5)

- **The Intermediate Code generated for the input file**

```
FINAL > Code_Optimization > non_optimized > show.txt
t1=1
a=t1
t2=2
b=t2
t3=3
c=t3
t4=8
t5=5
t6=t4*t5
t7=6
t8=t6+t7
P=t8
t9=8
t10=5
t11=t9*t10
t12=6
t13=t11+t12
e=t13
a
PRINT a
IMPORT d
a
t14=1
t15=a+t14
a=t15
a
t16=5
t17=a<=t16
b
t18=5
t19=b//t18
c=t19
L4 :
TRUE
IF (TRUE) GOTO L5
GOTO L6
L5 :
t20="hello"
PRINT t20
i=0
L1 :
 t21=i<10
IF (t21) GOTO L2
GOTO L3
L2 :
 i
PRINT i
t22=i+1
i=t22
GOTO L1
L3 :
GOTO L4
L6 :
```

(Figure 8.6)

- **Three Address Code(TAC) in Quadruple Format for the input file(text.tsv)**

```
FINAL > Code_Optimization > non_optimized > show.tsv
#   op  A1  A2  Res
1   =   1   -   t1
2   =   t1  -   a
3   =   2   -   t2
4   =   t2  -   b
5   =   3   -   t3
6   =   t3  -   c
7   =   8   -   t4
8   =   5   -   t5
9   *   t4  t5  t6
10  =   6   -   t7
11  +   t6  t7  t8
12  =   t8  -   P
13  =   8   -   t9
14  =   5   -   t10
15  *   t9  t10 t11
16  =   6   -   t12
17  +   t11 t12 t13
18  =   t13 -   e
19  PRINT   a   -   -
20  IMPORT  d   -   -
21  =   1   -   t14
22  +   a   t14 t15
23  =   t15 -   a
24  =   5   -   t16
25  <=  a   t16 t17
26  =   5   -   t18
27  //  b   t18 t19
28  =   t19 -   c
29  LABEL   -   -   L4
30  IF  TRUE    -   L5
31  GOTO    -   -   L6
32  LABEL   -   -   L5
33  =   "hello" -   t20
34  PRINT   t20 -   -
35  =   0   -   i
36  LABEL   -   -   L1
37  <   i   10  t21
38  IF  t21 -   L2
39  GOTO    -   -   L3
40  LABEL   -   -   L2
41  PRINT   i   -   -
42  +   i   1   t22
43  =   t22 -   i
44  GOTO    -   -   L1
45  LABEL   -   -   L3
46  GOTO    -   -   L4
47  LABEL   -   -   L6
```

(Figure 8.7)

- **Common Sub-Expression Elimination**

FINAL > Code_Optimization > CSE.py > ...

```python
'''
CODE LOGIC:
O(n^2) code
We loop between lines in quads and check if same line exists with different results
----------->Would work better with constant folded and propagated quads.
----------->Break the inner loop as soon as one of the args in the inner quad line is
            the result of another line in quads(Import and Print are exempted).
----------->If the found line has same operator and arguments as outer loop quad and the result
            is a temporary, delete the line and replace all occurences of that temporary with
            the result of the quad on which the search is being done.
'''
import re
fptr=open("non_optimized/show.tsv","r")
all_quads=fptr.readlines()[1:]
fptr.close()
all_quads = [(x[:-1].split("\t"))[1:] for x in all_quads]
todel=set()
print(len(all_quads))
for i in range(len(all_quads)-1):
    op,arg1,arg2,res=all_quads[i]
    if op=="PRINT" or op=="GOTO" or op=="LABEL":
        continue
    results=set()
    for j in range(i+1,len(all_quads)):
        opj,arg1j,arg2j,resj=all_quads[j]
        #if arg1 or arg2 is reassigned values and opj cant be Import or print because their result is -
        if (resj==arg1 or resj==arg2) and (opj!="IMPORT" and opj!="PRINT"):
            break
        #to remove a line : only temporaries need to be removed
        if op==opj and arg1==arg1j and arg2==arg2j and (re.search(r"^t[1-9]",resj)):
            results.add(resj)
            todel.add(j)
        if arg1j in results:
            all_quads[j][1]=res
        if arg2j in results:
            all_quads[j][2]=res
    #print(i,results)
all_quads=[all_quads[i] for i in (set(range(len(all_quads)))-todel)]
all_quads=["\t".join(i)+"\n" for i in all_quads]
all_quads=[str(i+1)+"\t"+all_quads[i] for i in range(len(all_quads))]
print(len(all_quads))
all_quads="".join(all_quads)
f=open("optimized/showCSE.tsv","w")
f.write("#\top\tA1\tA2\tRes\n")
f.write(all_quads)
f.close()
```

Python script -> CSE.py (Figure 8.8)

Output quads: As can be seen from Figure 8.7 and the next figure (Figure 8.9) the number of lines in quads has decreased from 47 to 39 using Common Subexpression Elimination. (Lines: 13-17, 21, 24 and 26 removed).

FINAL > Code_Optimization > optimized > showCSE.tsv

#	op	A1	A2	Res
1	=	1	-	t1
2	=	t1	-	a
3	=	2	-	t2
4	=	t2	-	b
5	=	3	-	t3
6	=	t3	-	c
7	=	8	-	t4
8	=	5	-	t5
9	*	t4	t5	t6
10	=	6	-	t7
11	+	t6	t7	t8
12	=	t8	-	P
13	=	t8	-	e
14	PRINT	a	-	-
15	IMPORT	d	-	-
16	+	a	t1	t15
17	=	t15	-	a
18	<=	a	t5	t17
19	//	b	t5	t19
20	=	t19	-	c
21	LABEL	-	-	L4
22	IF	TRUE	-	L5
23	GOTO	-	-	L6
24	LABEL	-	-	L5
25	=	"hello"	-	t20
26	PRINT	t20	-	-
27	=	0	-	i
28	LABEL	-	-	L1
29	<	i	10	t21
30	IF	t21	-	L2
31	GOTO	-	-	L3
32	LABEL	-	-	L2
33	PRINT	i	-	-
34	+	i	1	t22
35	=	t22	-	i
36	GOTO	-	-	L1
37	LABEL	-	-	L3
38	GOTO	-	-	L4
39	LABEL	-	-	L6

(Figure 8.9)

- ## Dead Code Elimination

FINAL > Code_Optimization > dead_code_elimination.py > ...

```python
'''
Code logic:
Loop through all the lines in the quads (i.e. while flag is True) at every iteration and for
every such line-see if there is no line where the result of the first line is used.
'''


import csv
import copy

PATH_TO_CSV = r"./non_optimized/show.tsv"
PATH_TO_OUTPUT_1 = r"./optimized/showDCE.tsv"

file_input = open(PATH_TO_CSV)
quads = list(csv.reader(file_input, delimiter='\t'))

def dead_code_elimination(quads):
    flag = True
    remove = True
    while(flag):
        flag = False
        for i in range(len(quads)):
            remove = True
            if(not (quads[i][4] == "-" or quads[i][1].lower() in ["label", "goto", "if false","if"])):
                for j in range(i+1,len(quads)):
                    if((quads[i][4] == quads[j][2] and quads[j][0] != "-1") or (quads[i][4] == quads[j][3] and quads[j][0] != "-1")):
                        remove = False
                if((remove == True) and (quads[i][0] != "-1")):
                    quads[i][0] = "-1"
                    flag = True
    return quads

quads_copy = copy.deepcopy(quads)
quads_output_1 = dead_code_elimination(quads_copy[1:])
quads_output_1.insert(0, quads[0])
quads_output_1=[i for i in quads_output_1 if i[0]!='-1']
#
index=1
for i in range(1,len(quads_output_1)):
    quads_output_1[i][0]=str(index)
    index+=1
print(quads_output_1)
file_output = open(PATH_TO_OUTPUT_1, "w")
csv_writer = csv.writer(file_output, delimiter='\t', lineterminator = "\n")
csv_writer.writerows(quads_output_1)
file_output.flush()
file_output.close()
file_input.close()
```

Python script -> dead_code_elimination.py (Figure 8.10)

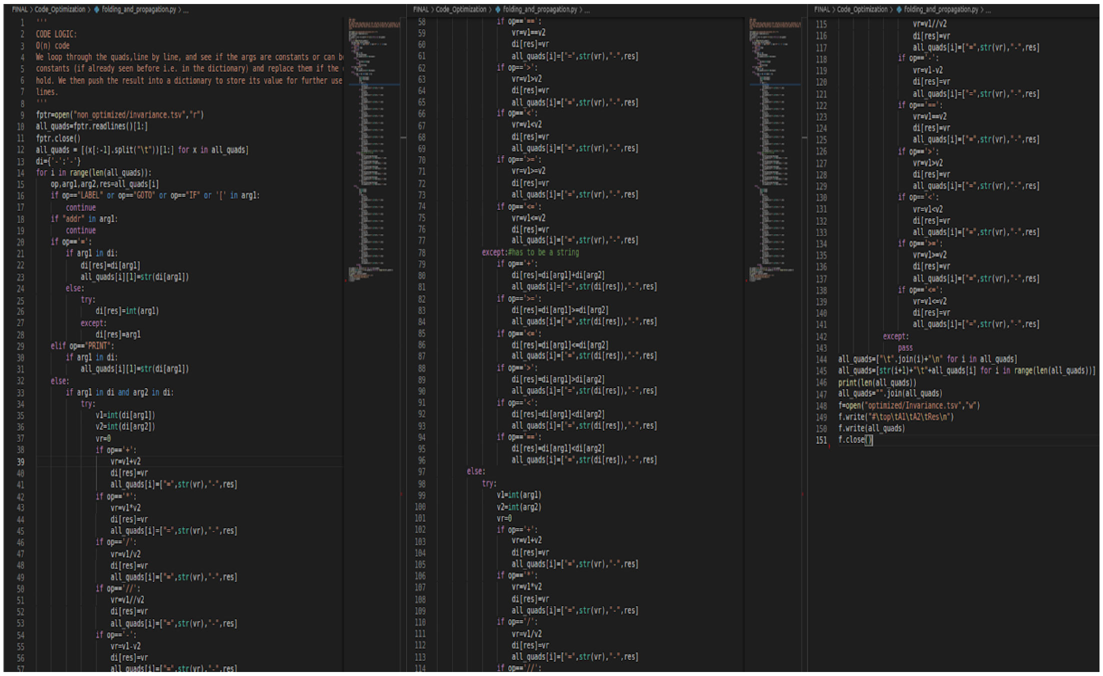
Output quads: As can be seen from Figure 8.7 and the next figure (Figure 8.11) the number of lines in quads has decreased from 47 to 23 using Dead Code Elimination. (Lines: 3-18, 21-28 removed).

FINAL > Code_Optimization > optimized > showDCE.tsv

```
#   op      A1      A2      Res
1   =       1       -       t1
2   =       t1      -       a
3   PRINT   a       -       -
4   IMPORT  d       -       -
5   LABEL   -       -       L4
6   IF      TRUE    -       L5
7   GOTO    -       -       L6
8   LABEL   -       -       L5
9   =       hello   -       t20
10  PRINT   t20     -       -
11  =       0       -       i
12  LABEL   -       -       L1
13  <       i       10      t21
14  IF      t21     -       L2
15  GOTO    -       -       L3
16  LABEL   -       -       L2
17  PRINT   i       -       -
18  +       i       1       t22
19  =       t22     -       i
20  GOTO    -       -       L1
21  LABEL   -       -       L3
22  GOTO    -       -       L4
23  LABEL   -       -       L6
```

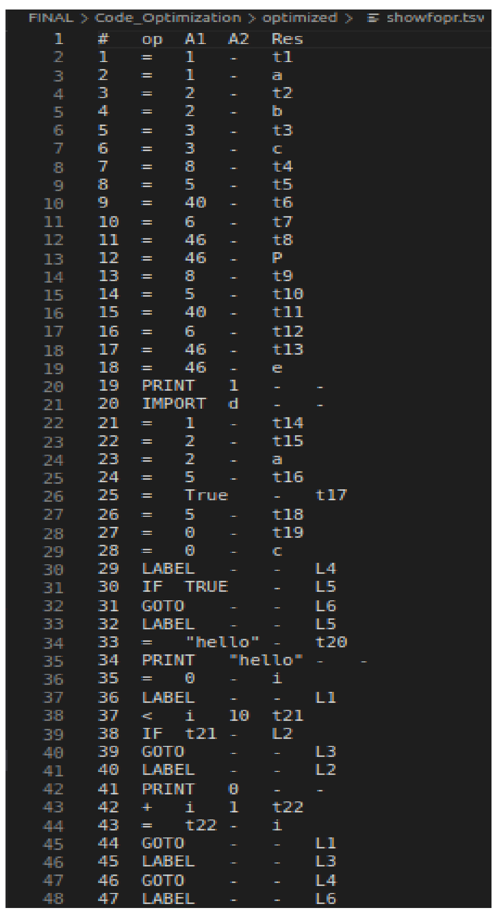
(Figure 8.11)

- **Constant Folding and Propagation**



Python script -> folding_and_propagation.py (Figure 8.12)

Output quads: As can be seen from Figure 8.7 and the next figure (Figure 8.13), constant values are being propagated and also folding occurs to next lines in the quad.



FINAL > Code_Optimization > optimized > showfopr.tsv

#	op	A1	A2	Res
1	=	1	-	t1
2	=	1	-	a
3	=	2	-	t2
4	=	2	-	b
5	=	3	-	t3
6	=	3	-	c
7	=	8	-	t4
8	=	5	-	t5
9	=	40	-	t6
10	=	6	-	t7
11	=	46	-	t8
12	=	46	-	P
13	=	8	-	t9
14	=	5	-	t10
15	=	40	-	t11
16	=	6	-	t12
17	=	46	-	t13
18	=	46	-	e
19	PRINT	1	-	-
20	IMPORT	d	-	-
21	=	1	-	t14
22	=	2	-	t15
23	=	2	-	a
24	=	5	-	t16
25	=	True	-	t17
26	=	5	-	t18
27	=	0	-	t19
28	=	0	-	c
29	LABEL	-	-	L4
30	IF	TRUE	-	L5
31	GOTO	-	-	L6
32	LABEL	-	-	L5
33	=	"hello"	-	t20
34	PRINT	"hello"	-	-
35	=	0	-	i
36	LABEL	-	-	L1
37	<	i	10	t21
38	IF	t21	-	L2
39	GOTO	-	-	L3
40	LABEL	-	-	L2
41	PRINT	8	-	-
42	+	i	1	t22
43	=	t22	-	i
44	GOTO	-	-	L1
45	LABEL	-	-	L3
46	GOTO	-	-	L4
47	LABEL	-	-	L6

(Figure 8.13)

- **Loop Invariant Code : Movement (even in nested loops)**

FINAL > Code_Optimization > loop_invariant_total.py > ...

```python
CODE LOGIC:
O(n) code
Loop Invariancy can be applied to loops only(duh)
    According to what we have developed IF(operator) signifies a loop with the result as the label to go to if the condition is true
It is a recursive code
    wherein we identify loops, mark the invariant code of the loop
    We first call the recurse function for nested loops
    Then remake the quads array by shifting and moving invariant code just outside the current loop.
In this optimization we iterate again and bring all of invariant code outside the main loop even in times of nested loops.
'''
fptr=open("optimized/showfopr.tsv","r")
all_quads=fptr.readlines()[1:]
fptr.close()
all_quads = [(x[:-1].split("\t"))[1:] for x in all_quads]
i=0
n=len(all_quads)
def recurse(i,n=n):#i is the index of the record
    global all_quads
    op,arg1,arg2,res=all_quads[i]
    if op=="IF" and arg1!="0" and arg1!="FALSE":#indicates loop in our code atleast
        label_index=-1
        for j in range(i-1,-1,-1):
            if all_quads[j][0].upper()=="LABEL":
                label_index=j
                break
        final_label=all_quads[i+1][-1]
        j=i+2
        invariants=[]
        while(j<n):
            opj,arg1j,arg2j,resj=all_quads[j]
            flag=j+1
            if opj.upper()=="LABEL" and resj==final_label:
                break
            try:
                if opj=="=" and (arg1j.startswith('"') or int(arg1j)):
                    invariants.append(all_quads[j])
                if opj.upper()=="IF":
                    flag=recurse(j)
            except:
                pass
            j=flag
        after=[all_quads[k] for k in range(label_index,j) if all_quads[k] not in invariants]
        all_quads=all_quads[:label_index]+invariants+after+all_quads[j:n]

        j=i+2
        while(j<n):
            opj,arg1j,arg2j,resj=all_quads[j]
            flag=j+1
            if opj.upper()=="LABEL" and resj==final_label:
                break
            try:
                if opj=="=" and (arg1j.startswith('"') or int(arg1j)):
                    invariants.append(all_quads[j])
            except:
                pass
            j=flag
        after=[all_quads[k] for k in range(label_index,j) if all_quads[k] not in invariants]#rejoining/redesigning quads array
        all_quads=all_quads[:label_index]+invariants+after+all_quads[j:n]
        return j+1
    return i+1
while i<n:
    i=recurse(i)
all_quads=["\t".join(i)+"\n" for i in all_quads]
all_quads=[str(i+1)+"\t"+all_quads[i] for i in range(len(all_quads))]
print(len(all_quads))
all_quads="".join(all_quads)
f=open("optimized/showinvariant.tsv","w")
f.write("#\top\tA1\tA2\tRes\n")
f.write(all_quads)
f.close()
```

Python script -> loop_invariant_total.py (Figure 8.14)

Output quads: As can be seen from Figure 8.7 and the next figure (Figure 8.15), loop invariant lines are moved above the loops, to the top most loop even in nested loops.

FINAL > Code_Optimization > optimized > showinvariant.tsv

#	op	A1	A2	Res
1	=	1	-	t1
2	=	1	-	a
3	=	2	-	t2
4	=	2	-	b
5	=	3	-	t3
6	=	3	-	c
7	=	8	-	t4
8	=	5	-	t5
9	=	40	-	t6
10	=	6	-	t7
11	=	46	-	t8
12	=	46	-	P
13	=	8	-	t9
14	=	5	-	t10
15	=	40	-	t11
16	=	6	-	t12
17	=	46	-	t13
18	=	46	-	e
19	PRINT	1	-	-
20	IMPORT	d	-	-
21	=	1	-	t14
22	=	2	-	t15
23	=	2	-	a
24	=	5	-	t16
25	=	True	-	t17
26	=	5	-	t18
27	=	0	-	t19
28	=	0	-	c
29	=	"hello"	-	t20
30	LABEL	-	-	L4
31	IF	TRUE	-	L5
32	GOTO	-	-	L6
33	LABEL	-	-	L5
34	PRINT	"hello"	-	-
35	=	0	-	i
36	LABEL	-	-	L1
37	<	i	10	t21
38	IF	t21	-	L2
39	GOTO	-	-	L3
40	LABEL	-	-	L2
41	PRINT	0	-	-
42	+	i	1	t22
43	=	t22	-	i
44	GOTO	-	-	L1
45	LABEL	-	-	L3
46	GOTO	-	-	L4
47	LABEL	-	-	L6

(Figure 8.15)

Loop Invariant Code and Movement for the python file:

```python
a="hello"
d=6
c=a
while True:
     e=a
     for i in range(10):
          b=d
          while False:
               f=5
```

Non-optimized Quad Code

#	op	A1	A2	Res
1	=	"hello"	-	t1
2	=	t1	-	a
3	=	6	-	t2
4	=	t2	-	d
5	=	a	-	c
6	LABEL	-	-	L7
7	IF	TRUE	-	L8
8	GOTO	-	-	L9
9	LABEL	-	-	L8
10	=	a	-	e
11	=	0	-	i
12	LABEL	-	-	L4
13	<	i	10	t4
14	IF	t4	-	L5
15	GOTO	-	-	L6
16	LABEL	-	-	L5
17	=	d	-	b
18	LABEL	-	-	L1
19	IF	FALSE	-	L2
20	GOTO	-	-	L3
21	LABEL	-	-	L2
22	=	5	-	t3
23	=	t3	-	f
24	GOTO	-	-	L1
25	LABEL	-	-	L3
26	+	i	1	t5
27	=	t5	-	i
28	GOTO	-	-	L4
29	LABEL	-	-	L6
30	GOTO	-	-	L7
31	LABEL	-	-	L9

Optimized Quad Code

FINAL > Code_Optimization > optimized > showinvariant.tsv

#	op	A1	A2	Res
1	=	"hello"	-	t1
2	=	"hello"	-	a
3	=	6	-	t2
4	=	6	-	d
5	=	"hello"	-	c
6	=	"hello"	-	e
7	=	6	-	b
8	=	5	-	t3
9	=	5	-	f
10	LABEL	-	-	L7
11	IF	TRUE	-	L8
12	GOTO	-	-	L9
13	LABEL	-	-	L8
14	=	0	-	i
15	LABEL	-	-	L4
16	<	i	10	t4
17	IF	t4	-	L5
18	GOTO	-	-	L6
19	LABEL	-	-	L5
20	LABEL	-	-	L1
21	IF	FALSE	-	L2
22	GOTO	-	-	L3
23	LABEL	-	-	L2
24	GOTO	-	-	L1
25	LABEL	-	-	L3
26	+	i	1	t5
27	=	t5	-	i
28	GOTO	-	-	L4
29	LABEL	-	-	L6
30	GOTO	-	-	L7
31	LABEL	-	-	L9

9. CONCLUSIONS

This python mini compiler goes through all 5 different phases of compilation of python code namely, lexical analysis, syntax analysis, semantic analysis, intermediate code generation, and code optimisation. The 2 main tools used to build this mini compiler are lex/flex and yacc/bison and python for optimization.

The lex tool was used to build the lexical analysis phase by using regex to match the lexemes and convert to tokens. Whereas the yacc tool was used to parse the grammar along with implementing actions for the context free grammar.

Both these files used many custom functions to keep track of various data structures to store the scope, value, line number, etc. These are then used to find and report errors during any of the 5 phases.

10. FURTHER ENHANCEMENTS

Some further enhancements that could be done are:

- Make the optimizations better and more robust.
- Make the grammar encompass many more constructs.