LAB REPORT-5
Subtopic: Code Optimization

A) Write code to identify the basic blocks and the flow graph of any intermediate code generated for any sample program written in the language you have developed.

B) Apply the optimization techniques (eg. dead code elimination, constant folding, copy propagation, induction variables, peephole optimization and optimization using next use information, etc.) over your intermediate code generated to improve the code.

C) i) Implement the recursive QuickSort algorithm in your language taking the first element as the pivot.
ii) Show all steps of compiler step by step: Lexical Analysis, Syntax and Semantic Analysis, Intermediate Code Generation, and Code Optimization.
iii) Show the flow graph and DAG for this code.
iv) Draw all steps for optimizing the code.

A) Write code to identify the basic blocks and the flow graph of any intermediate code generated for any sample program written in the language you have developed.

Code:

```
blocks = [[]]
next_block = [[]]
functions = {}
skip_f = False
idx = -1
returnto = []
labels = {}
mode="read" #read
function_trig = []
print("\n\nFLow graph")
while idx<len(quads):
    print(idx, quads[idx])
    idx+=1
    if idx>=len(quads):
        break
    ith_quad = quads[idx]
    if 'BeginF' == ith_quad[1]:
        functions[ith_quad[2]] = idx
        skip_f = True

    if 'EndF' == ith_quad[1]:
        print("Return to", returnto)
        if skip_f:
```

```
            skip_f = False
            continue
        else:
            idx = returnto.pop()
            blocks[-1].append(ith_quad)
            blocks.append([])
            continue
    if skip_f:
        continue
    if 'Label' == ith_quad[1]:
        if len(blocks[-1]) != 0:
            blocks.append([])
        labels[ith_quad[-1]] = len(blocks)-1
    if 'If False' == ith_quad[1]  or  'goto' == ith_quad[1]:
        blocks[-1].append(ith_quad)
        blocks.append([])
        continue
    blocks[-1].append(ith_quad)
    if 'Call' == ith_quad[1]:
        if ith_quad[2] not in function_trig:
            function_trig.append(ith_quad[2])
        else:
            continue
        if len(blocks[-1]) != 0:
            blocks.append([])
        returnto.append(idx)

        fname = ith_quad[2]
        blocks[-1].append(quads[functions[fname]])
        idx = functions[fname]

shift=0
for idx, block in enumerate(blocks):
    if len(block) == 0:
        blocks.pop(idx-shift)
        shift+=1
```

The code initialises a list of blocks to represent basic blocks within the program. Each block contains a list of quadruples. Additionally, it initialises dictionaries to store function indices (`functions`) and labels (`labels`).
Subsequently, the code iterates over each quadruple and performs several operations. It identifies function beginnings (`BeginF`) and ends (`EndF`) and stores their indices in the `functions` dictionary. Labels (`Label`) within the quadruples mark the beginning of new basic blocks. Conditional branches (`If False`) and unconditional jumps (`goto`) are utilised to split basic blocks based on control flow.
Furthermore, calls to functions (`Call`) are identified, and their indices are stored in the `function_trig` list. The code constructs the basic blocks by appending quadruples to the current block until it encounters a control flow change, such as a function call, branch, or jump. It handles these control flow changes by creating new blocks and updating the indices accordingly.
The code prints the flow graph during the processing by displaying each quadruple with its corresponding index. The generated flow graph can be further visualised using external libraries like NetworkX and Matplotlib.

Sample Code

```
import math
x=10
y=2
c=0
listX = [1,2,3,4]

def F1(A, B, C):
        while(listX[2]==y):
                z=x+y
                b=x-y
                if(z==b):
                        c=10+b
                else:
                        c=10+z
        return


m = F1(10, 10, 10)
print(m)
```

ICG output: Three address code

```
import math
T2 = 10
x = T2
T5 = 2
y = T5
T8 = 0
c = T8
Begin Function F1
T19 = listX[2]
T20 = y
T21 = T19 == T20
L0: If False T21 goto L1
T22 = x
T23 = y
T24 = T22 + T23
z = T24
T27 = x
T28 = y
T29 = T27 - T28
b = T29
T32 = z
T33 = b
T34 = T32 == T33
If False T34 goto L2
T35 = 10
T36 = b
T37 = T35 + T36
c = T37
goto L3
L2: T42 = 10
T43 = z
```

```
T44 = T42 + T43
c = T44
return
L3: goto L0
L1: End Function F1
Push Param 10
Push Param  10
Push Param  10
(T65)Call Function F1, 3
Pop Params for Function F1, 3
m = T65
T68 = m
Print T68

--------------------------------All Quads--------------------------------
0       import  math    -       -
1       =       10      -       T2
2       =       T2      -       x
3       =       2       -       T5
4       =       T5      -       y
5       =       0       -       T8
6       =       T8      -       c
7       BeginF  F1      -       -
8       ListIndex       listX   2       T19
9       =       y       -       T20
10      ==      T19     T20     T21
11      Label   -       -       L0
12      If False        T21     -       L1
13      =       x       -       T22
14      =       y       -       T23
15      +       T22     T23     T24
16      =       T24     -       z
17      =       x       -       T27
18      =       y       -       T28
19      -       T27     T28     T29
20      =       T29     -       b
21      =       z       -       T32
22      =       b       -       T33
23      ==      T32     T33     T34
24      If False        T34     -       L2
25      =       10      -       T35
26      =       b       -       T36
27      +       T35     T36     T37
28      =       T37     -       c
29      goto    -       -       L3
30      Label   -       -       L2
31      =       10      -       T42
32      =       z       -       T43
33      +       T42     T43     T44
34      =       T44     -       c
35      return  -       -       -
36      Label   -       -       L3
37      goto    -       -       L0
38      Label   -       -       L1
```

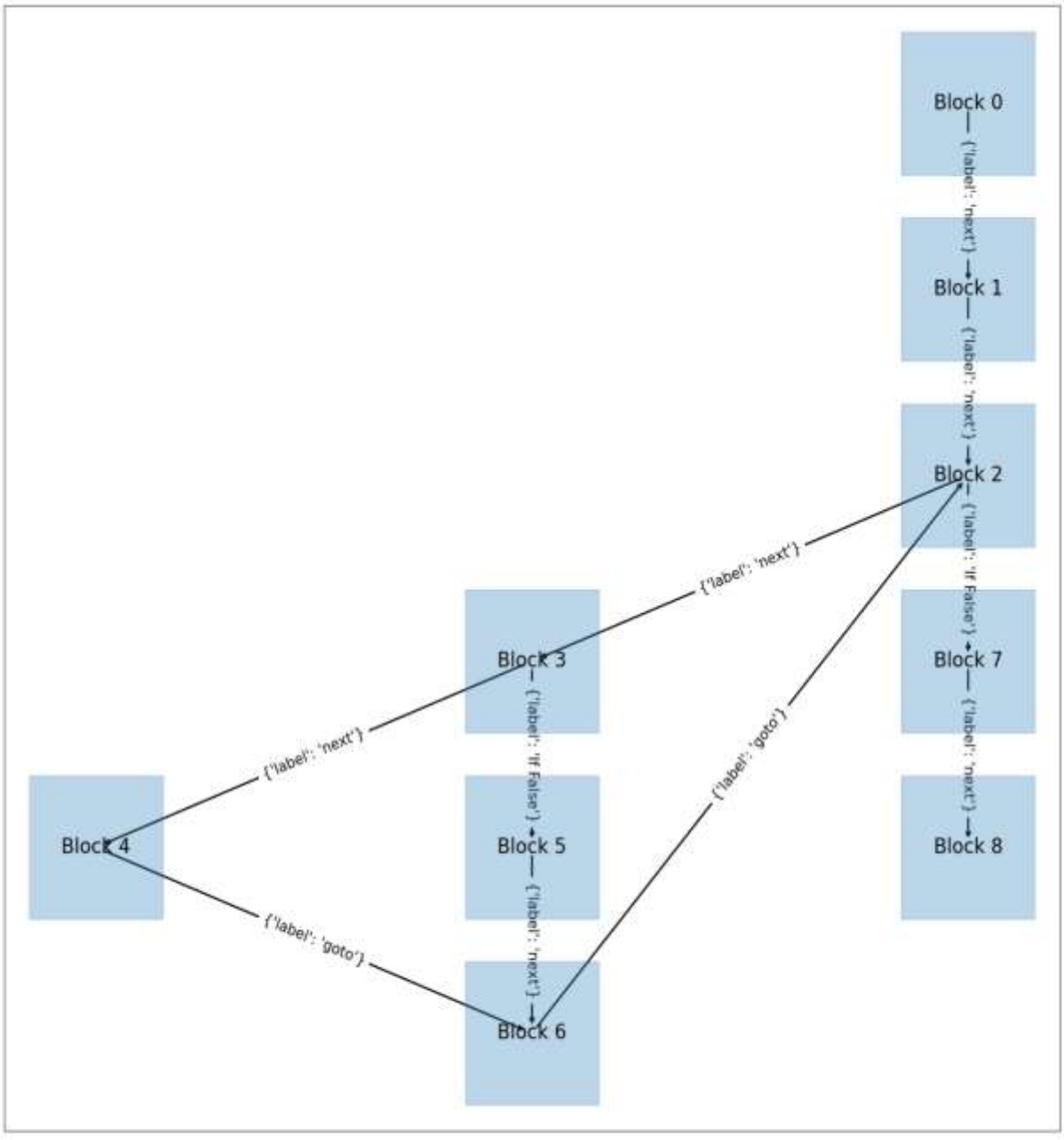
```
39      EndF    F1      -       -
40      Param   10      -       -
41      Param    10     -       -
42      Param    10     -       -
43      Call    F1      3       T65
44      =       T65     -       m
45      =       m       -       T68
46      Print   T69     -       -
--------------------------------------------------------------------------
```

Basic blocks:

```
('Block 0:'
 "['0', 'import', 'math', '-', '-']" - leader
 "['1', '=', '10', '-', 'T2']"
 "['2', '=', 'T2', '-', 'x']"
 "['3', '=', '2', '-', 'T5']"
 "['4', '=', 'T5', '-', 'y']"
 "['40', 'Param', '10', '-', '-']"
 "['41', 'Param', ' 10', '-', '-']"
 "['42', 'Param', ' 10', '-', '-']"
 "['43', 'Call', 'F1', '3', 'T61']")
==============
('Block 1:'
 "['7', 'BeginF', 'F1', '-', '-']" - leader
 "['8', 'ListIndex', 'listX', '2', 'T15']"
 "['9', '=', 'y', '-', 'T16']"
 "['10', '==', 'T15', 'T16', 'T17']")
==============
('Block 2:'
 "['11', 'Label', '-', '-', 'L0']" - leader
 "['12', 'If False', 'T17', '-', 'L1']")
==============
('Block 3:'
 "['13', '=', 'x', '-', 'T18']" - leader
 "['14', '=', 'y', '-', 'T19']"
 "['15', '+', 'T18', 'T19', 'T20']"
 "['16', '=', 'T20', '-', 'z']"
 "['17', '=', 'x', '-', 'T23']"
 "['18', '=', 'y', '-', 'T24']"
 "['19', '-', 'T23', 'T24', 'T25']"
 "['20', '=', 'T25', '-', 'b']"
 "['21', '=', 'z', '-', 'T28']"
 "['22', '=', 'b', '-', 'T29']"
 "['23', '==', 'T28', 'T29', 'T30']"
 "['24', 'If False', 'T30', '-', 'L2']")
==============
"Block 4:['29', 'goto', '-', '-', 'L3']" - leader
==============
"Block 5:
['30', 'Label', '-', '-', 'L2'] - leader
['35', 'return', '-', '-', '-']"
==============
"Block 6:
```

['36', 'Label', '-', '-', 'L3'] - leader
['37', 'goto', '-', '-', 'L0']"
==========
"Block 7:['38', 'Label', '-', '-', 'L1'] - leader
['39', 'EndF', 'F1', '-', '-']"
==========
('Block 8:'
 "['44', '=', 'T61', '-', 'm']" - leader
 "['45', '=', 'm', '-', 'T64']"
 "['46', 'Print', 'T64', '-', '-']")
==========

Control Flow graph



B) Apply the optimization techniques (eg. dead code elimination, constant folding, copy propagation, induction variables, peephole optimization, etc.) over your intermediate code generated to improve the code.

Code:

```
class ICG_Optimization:
   def __init__(self, l) -> None:
      self.l = l
      self.lineno = l[0]
      self.op = l[1]
      l[2:] = [None if i=='-' else (i.strip() if i!=None else None) for i in l[2:]]
      self.op1 = l[2]
      self.op2 = l[3]
      self.result = l[4]
   def __repr__(self) -> str:
      return f"{self.l[0]}: {self.op}, {self.op1}, {self.op2}, {self.result}"

def check_str(a):
   return a[0]=="" and a[-1]==""

def constant_folding(block):
        for idx, i in enumerate(block):
                if i.op1 != None and i.op2 != None:
                        if (i.op1.isdigit() or check_str(i.op1)) and (i.op2.isdigit() or
                        check_str(i.op2)):
                                print("Constant folding", i)
                                block[idx] = ICG_Optimization([i.lineno, "=",
                                str(eval(f"{i.op1} {i.op} {i.op2}")), None, i.result])
        return block
```

Constant folding is implemented through the `constant_folding` function, which identifies arithmetic expressions where both operands are constants and computes their values at compile time. If both operands are constants, the expression is evaluated, and the result is substituted for the expression within the block. These optimisations help reduce unnecessary computations and improve the efficiency of the code.

```
def constant_propagation_replace(block, idx, term, value):
        for i in range(idx, len(block)):
                if block[i].op1 == term:
                        block[i].op1 = value
                if block[i].op2 == term:
                        block[i].op2 = value
                if block[i].result == term:
                        break
        return block
def constant_propagation(block):
        for idx, i in enumerate(block):
                if i.result !=None:
                        if i.op2 == None and i.op1 is not None :
                                if i.op1.isdigit() or (i.op1[0]=="" and i.op1[-1]==""):
```

```
                                block = constant_propagation_replace(block, idx+1,
                                i.result, i.op1)
                                continue
                        elif i.op1 == None and i.op2 is not None:
                                if i.op2.isdigit() or (i.op2[0]=="" and i.op2[-1]==""):
                                block = constant_propagation_replace(block, idx+1,
                                i.result, i.op1)
                                continue
                        elif i.op1 == None and i.op2 == None:
                                continue
        return block
```

Constant propagation is achieved through the `constant_propagation` function, which iterates through the given block of code and replaces variables with continuous values if possible. It checks each instruction in the block and updates the result if both operands are constants. The `constant_propagation_replace` function is used internally to replace occurrences of the specified term with the given value within a specific range of instructions.

```
def deadCodeEliminations(blocks):
    in_use = {}
    for idx, block in enumerate(blocks):
        for idx_l, linel in enumerate(block):
            if linel.op1:
                if not linel.op1.isdigit() or (linel.op1[0]=="" and linel.op1[-1]==""):
                    in_use[linel.op1] = True
            if linel.op2:
                if not linel.op2.isdigit() or (linel.op2[0]=="" and linel.op2[-1]==""):
                    in_use[linel.op2] = True
            if linel.result and linel.op == '=':
                if not linel.result.isdigit() or (linel.result[0]=="" and linel.result[-1]==""):
                    in_use[linel.result] = False
                else:
                    continue
        blocks[idx] = block
    pprint(in_use)
    for idx, block in enumerate(blocks):
        offset = 0
        for idx_l, linel in enumerate(block):
            if linel.result:
                if not in_use[linel.result]:
                    block.pop(idx_l-offset)
                    offset+=1
    return blocks
before_copy = deepcopy(tot_icg)
while True:
    tot_icg = deadcodeeliminations(tot_icg)
    for i in range(len(tot_icg)):
        tot_icg[i] = constant_folding(tot_icg[i])
        tot_icg[i] = constant_propagation(tot_icg[i])
    if tot_repr(tot_icg) == tot_repr(before_copy):
        break
    else:
        before_copy = deepcopy(tot_icg)
```

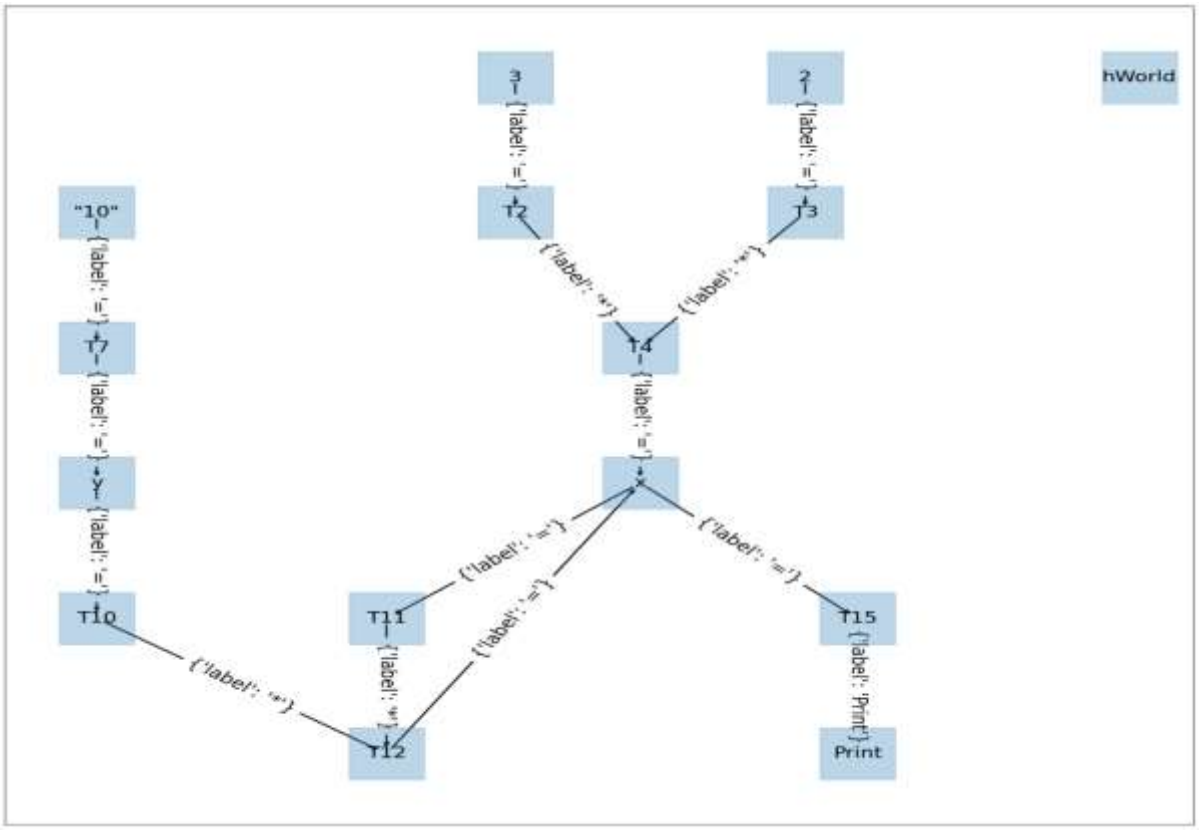
The deadCodeEliminations function iterates over each block and each line within the block, identifying variables and constants that are used and unused within the code. It then removes the unused variables from the blocks. Additionally, within each iteration, it applies constant folding and constant propagation optimizations to further improve the efficiency of the code.

Sample input:

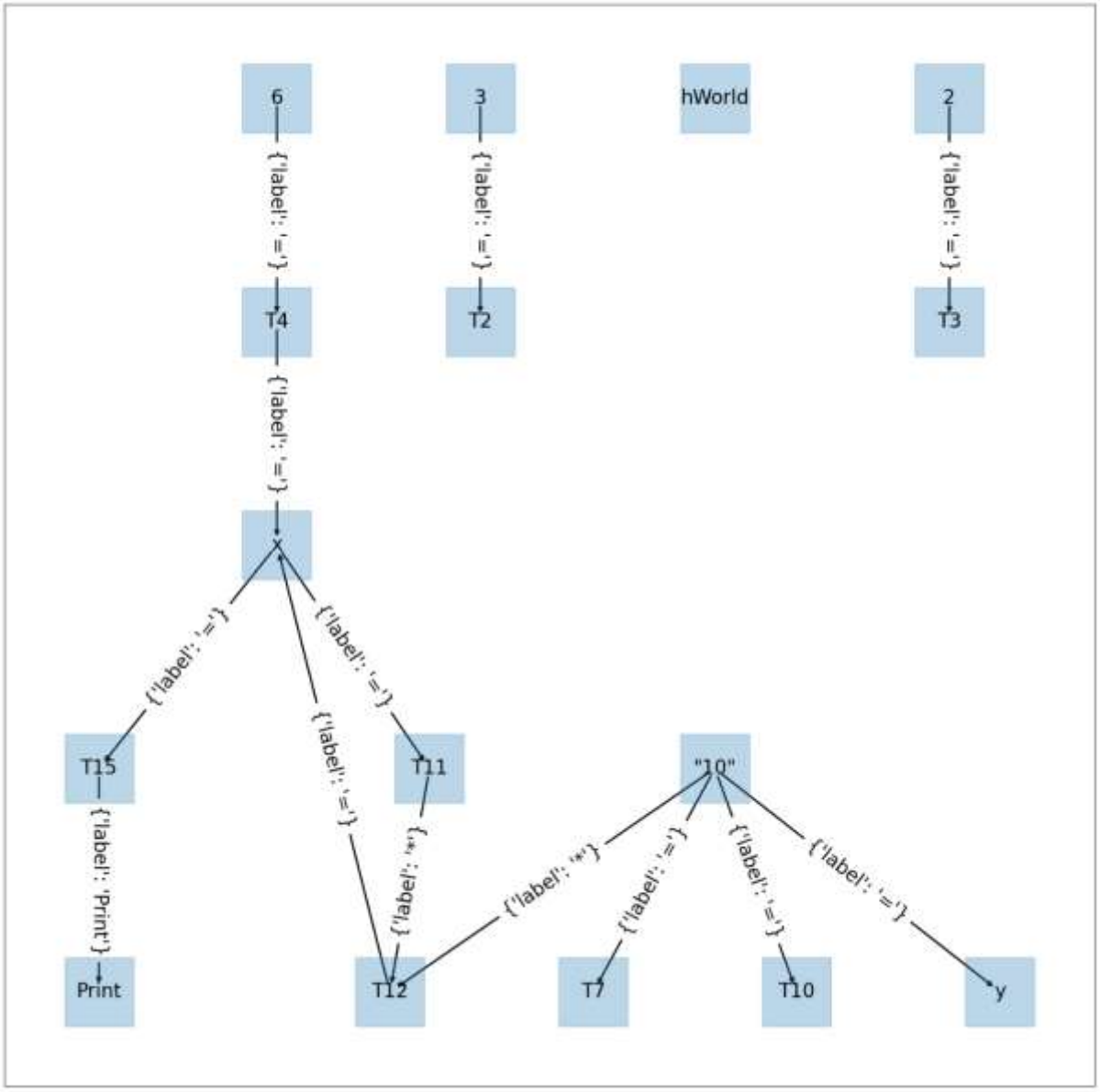
```
import hWorld
x=3*2
y="10"
x=y*x
print(x)
```

Three address code (Before optimization):

```
--------------------------------All Quads--------------------------------
0       import hWorld   -       -
1       =       3       -       T2
2       =       2       -       T3
3       *       T2      T3      T4
4       =       T4      -       x
5       =       "10"    -       T7
6       =       T7      -       y
7       =       y       -       T10
8       =       x       -       T11
9       *       T10     T11     T12
10      =       T12     -       x
11      =       x       -       T15
12      Print   T15     -       -
-------------------------------------------------------------------------
```



After 1 pass of copy propagation:

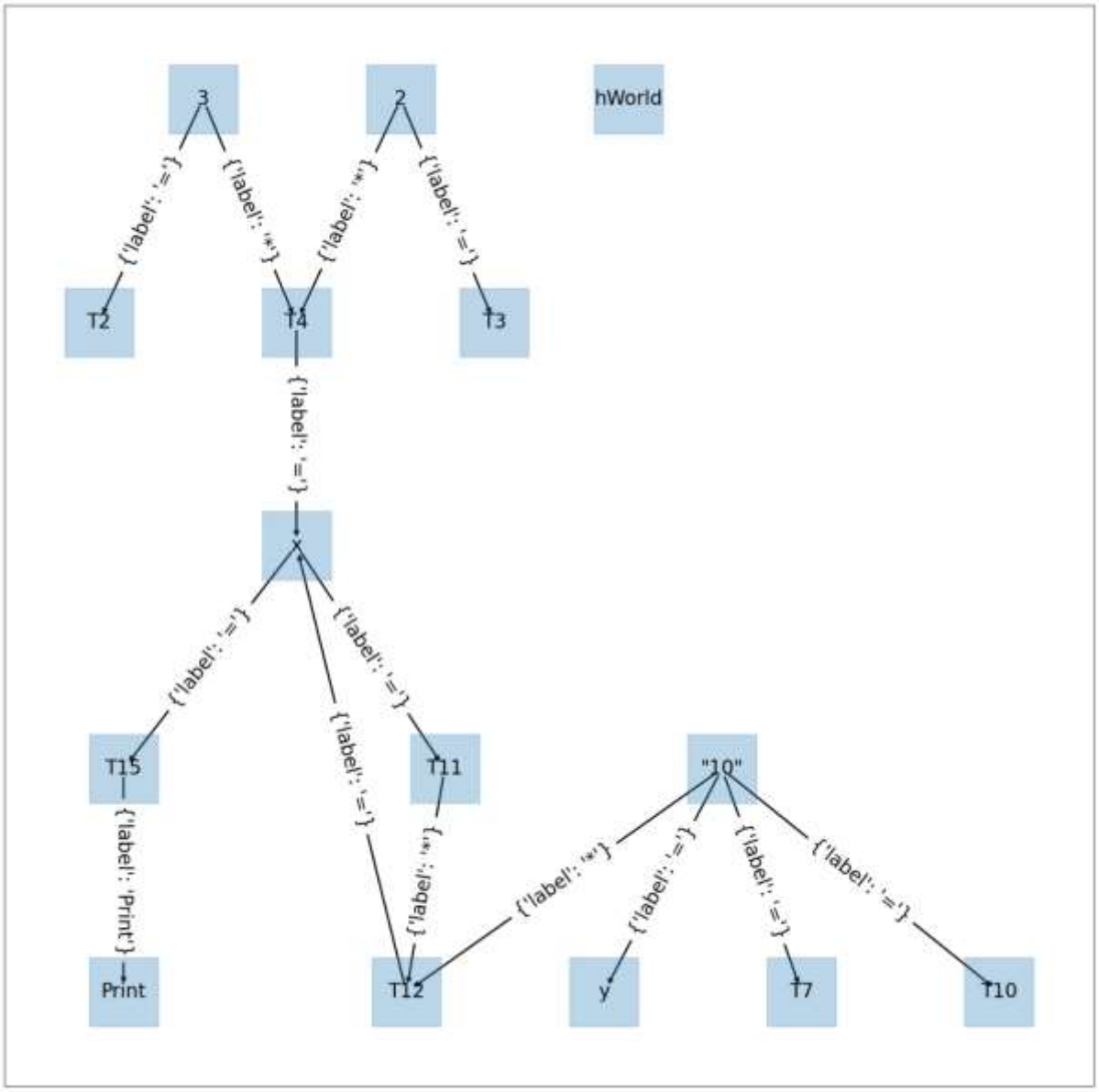


After 1 pass of copy propagation with 1 pass of constant folding:

[0: import, hWorld, None, None,
1: =, 3, None, T2,
2: =, 2, None, T3,
3: =, 6, None, T4,
4: =, T4, None, x,
5: =, "10", None, T7,
6: =, "10", None, y,
7: =, "10", None, T10,
8: =, x, None, T11,
9: *, "10", T11, T12,
10: =, T12, None, x,
11: =, x, None, T15,
12: Print, T15, None, None]

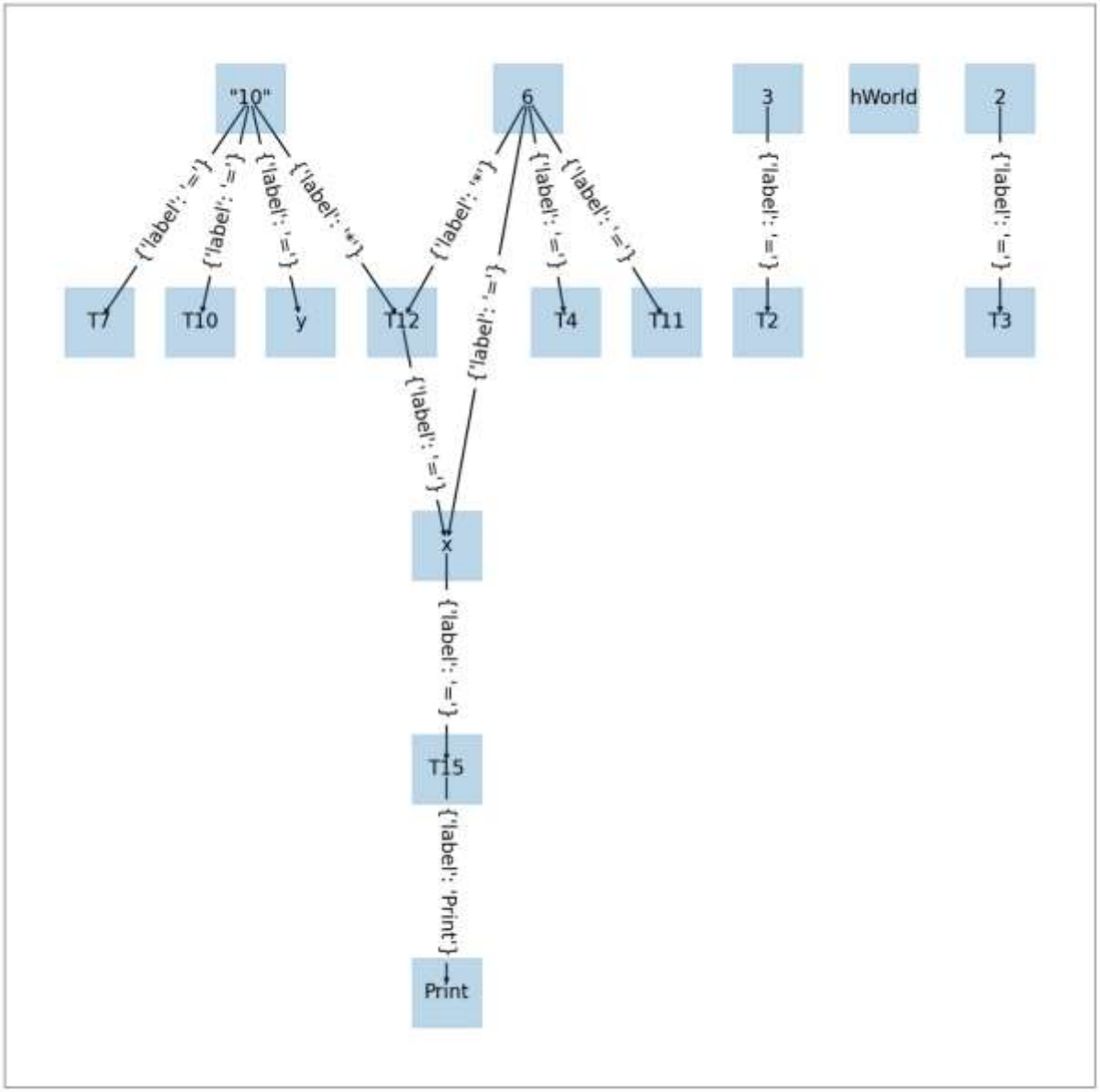
If done multiple time with copy propagation alternatively (no dead code elimination):

pass 2



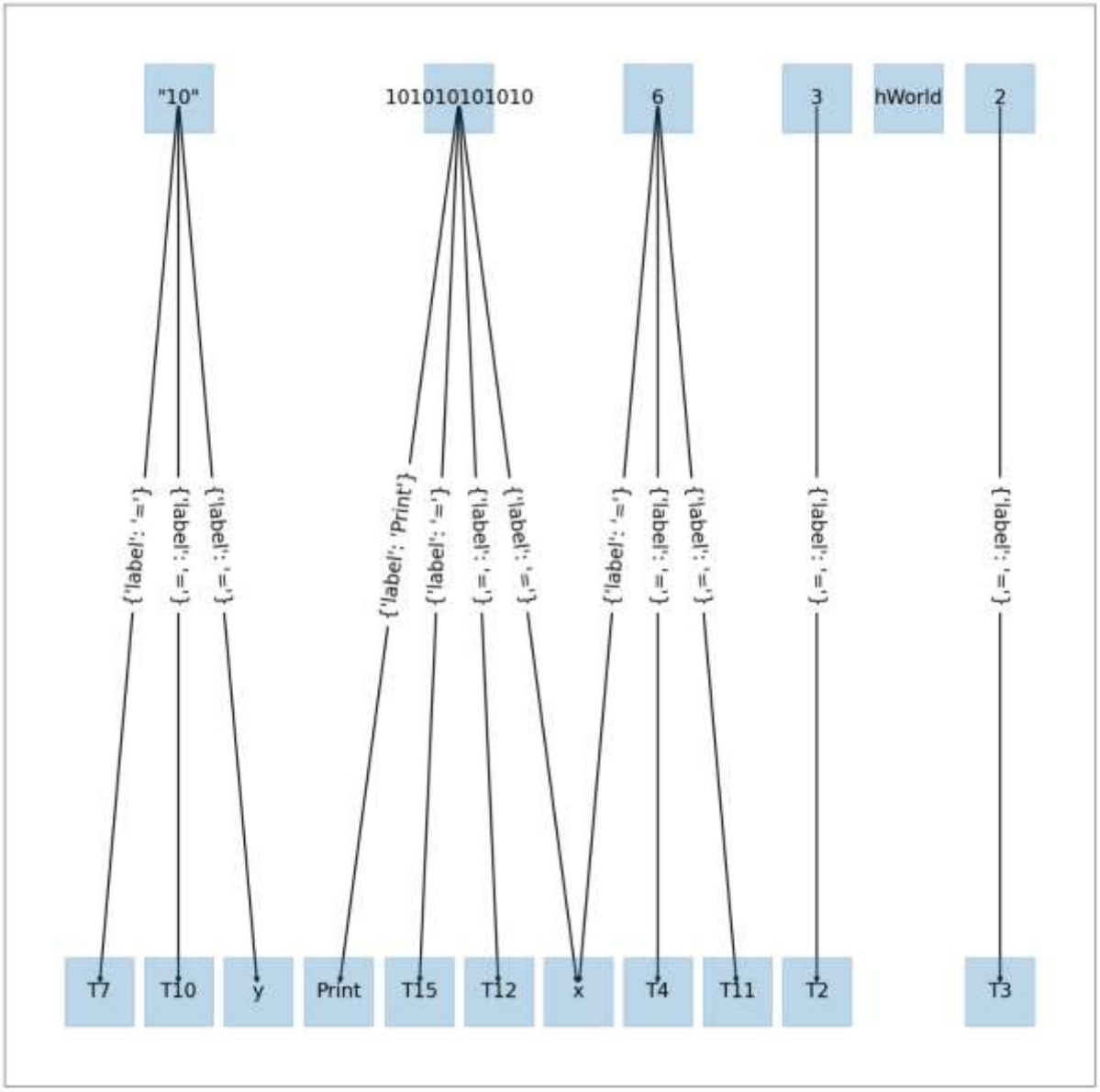
[[0: import, hWorld, None, None,
1: =, 3, None, T2,
2: =, 2, None, T3,
3: *, 3, 2, T4,
4: =, T4, None, x,
5: =, "10", None, T7,
6: =, "10", None, y,
7: =, "10", None, T10,
8: =, x, None, T11,
9: *, "10", T11, T12,
10: =, T12, None, x,
11: =, x, None, T15,
12: Print, T15, None, None]]

pass 3:



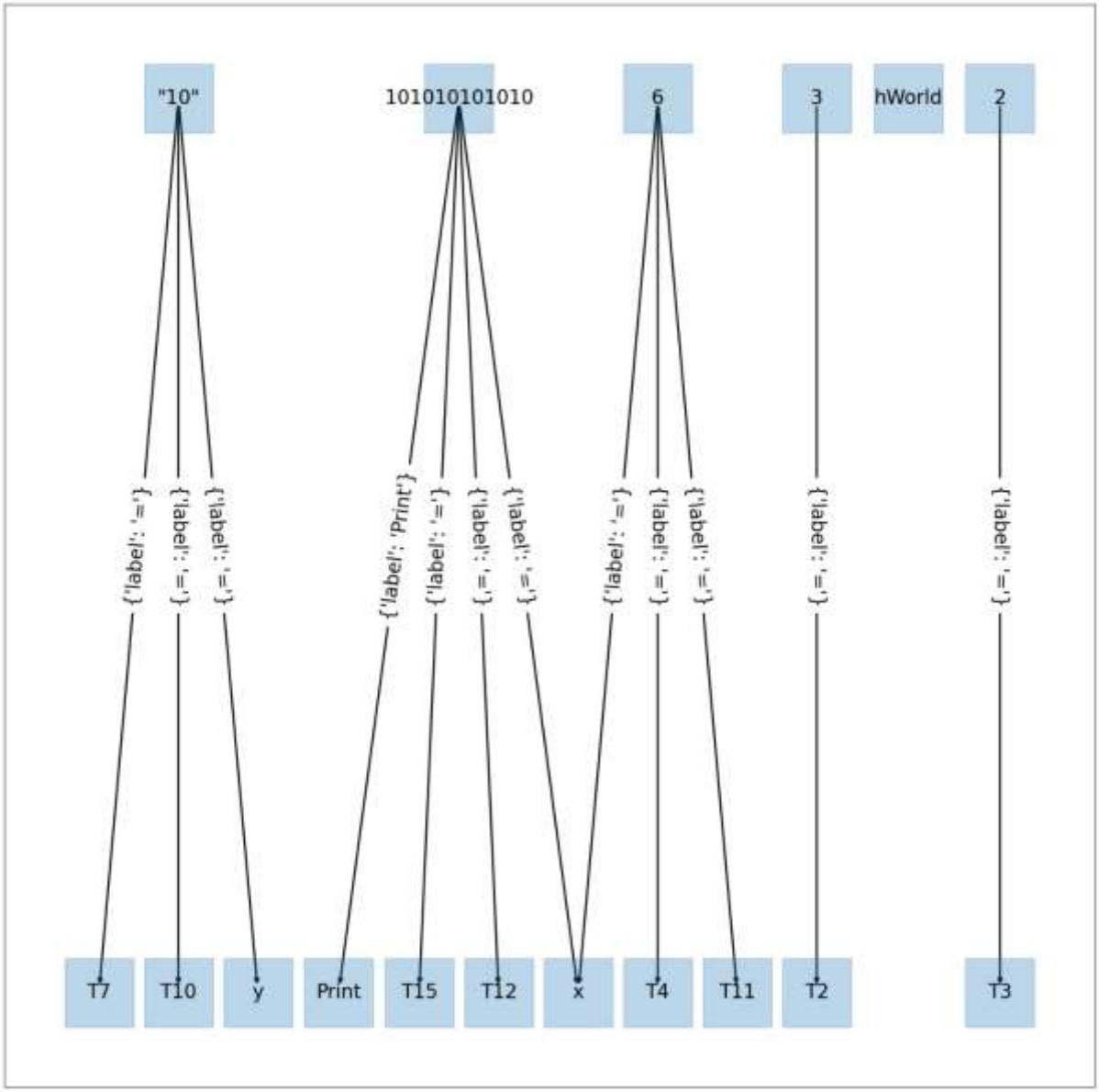
[[0: import, hWorld, None, None,
1: =, 3, None, T2,
2: =, 2, None, T3,
3: =, 6, None, T4,
4: =, 6, None, x,
5: =, "10", None, T7,
6: =, "10", None, y,
7: =, "10", None, T10,
8: =, 6, None, T11,
9: *, "10", 6, T12,
10: =, T12, None, x,
11: =, x, None, T15,
12: Print, T15, None, None]]

pass 4:



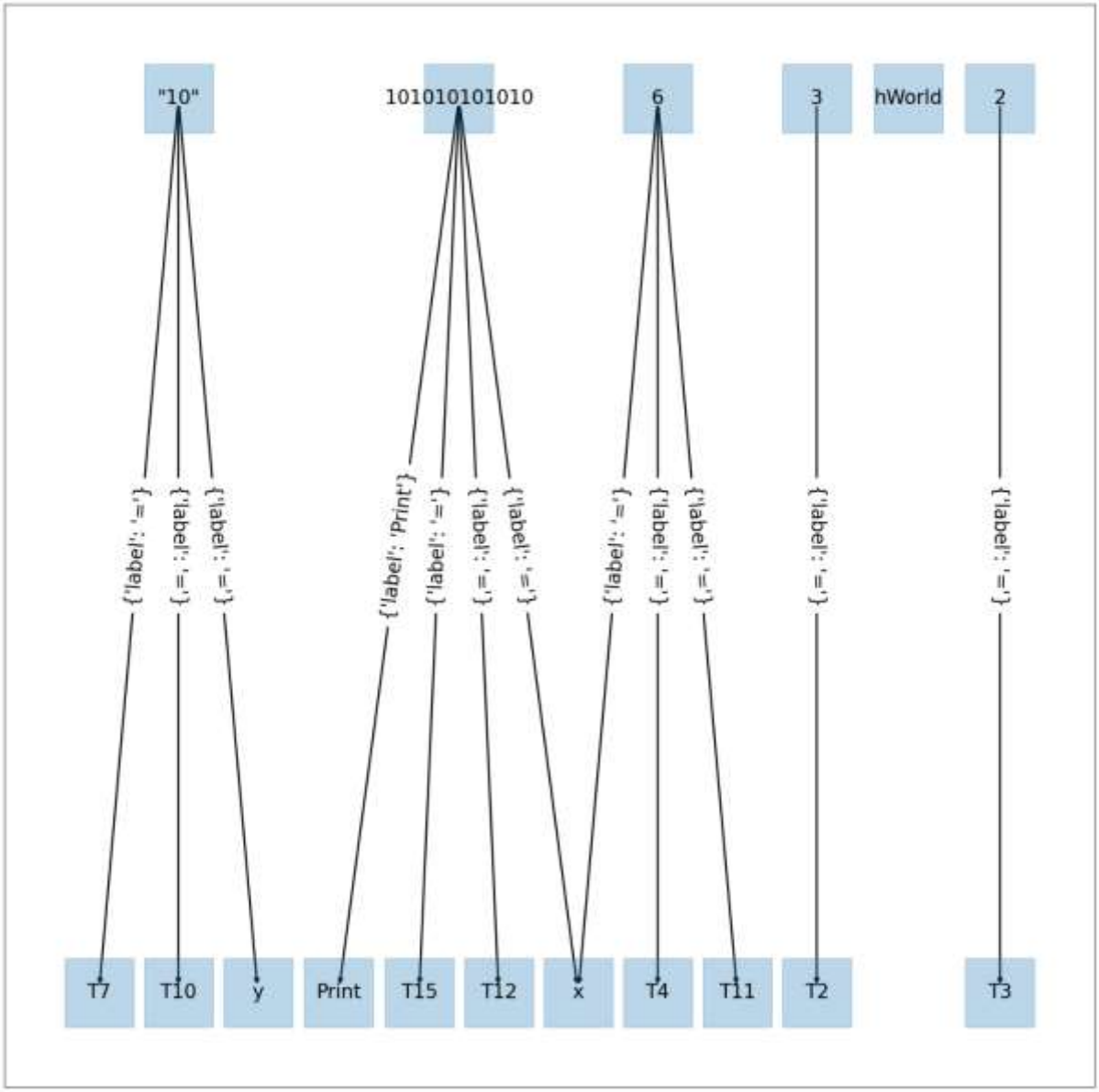
[[0: import, hWorld, None, None,
1: =, 3, None, T2,
2: =, 2, None, T3,
3: =, 6, None, T4,
4: =, 6, None, x,
5: =, "10", None, T7,
6: =, "10", None, y,
7: =, "10", None, T10,
8: =, 6, None, T11,
9: *, "10", 6, T12,
10: =, T12, None, x,
11: =, x, None, T15,
12: Print, T15, None, None]]

pass 5:



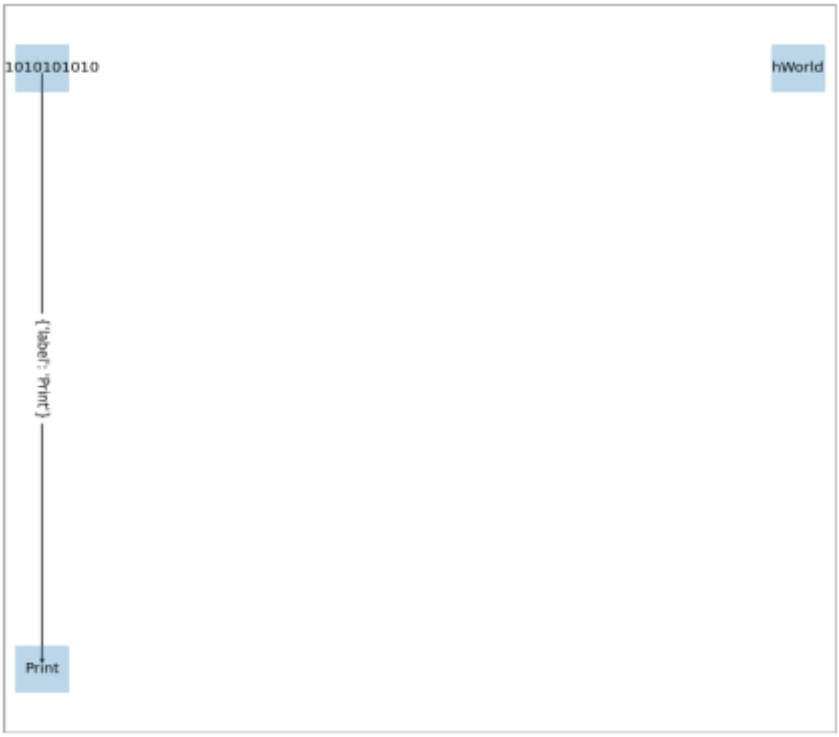
[[0: import, hWorld, None, None,
1: =, 3, None, T2,
2: =, 2, None, T3,
3: =, 6, None, T4,
4: =, 6, None, x,
5: =, "10", None, T7,
6: =, "10", None, y,
7: =, "10", None, T10,
8: =, 6, None, T11,
9: =, 101010101010, None, T12,
10: =, 101010101010, None, x,
11: =, 101010101010, None, T15,
12: Print, 101010101010, None, None]]

Final pass:



[[0: import, hWorld, None, None,
1: =, 3, None, T2,
2: =, 2, None, T3,
3: =, 6, None, T4,
4: =, 6, None, x,
5: =, "10", None, T7,
6: =, "10", None, y,
7: =, "10", None, T10,
8: =, 6, None, T11,
9: =, 101010101010, None, T12,
10: =, 101010101010, None, x,
11: =, 101010101010, None, T15,
12: Print, 101010101010, None, None]]

After performing copy propagation and constant folding, with dead code elimination:



Three address code (After optimization):

[[0: import, hWorld, None, None, 12: Print, 101010101010, None, None]]

```
import hWorld
print("101010101010")
```

Peephole Optimization:

Code:

```
def remove_unreachable(tot_icg):
	reachable = {}
	for idx, block in enumerate(tot_icg):
		for idx_l, linel in enumerate(block):
			if linel.op == 'goto':
				reachable[linel.result] = True
			if linel.op == 'If False':
				reachable[linel.result] = True
	print(reachable)
	offset = 0
	for idx, block in enumerate(tot_icg):
		if block[0].op == 'Label':
			if block[-1].result not in reachable:
				tot_icg.pop(idx-offset)
				offset+=1
	return tot_icg
```

```
def peephole(tot_icg):
	tot_icg = remove_unreachable(tot_icg)
	for idx, block in enumerate(tot_icg):
		offset = 0
		for idx_l, linel in enumerate(block):
			if linel.op == 'If False':
				if linel.op1 in ['True', 'False']:
					if linel.op1 == 'False':
						block[idx_l] = ICG([linel.lineno, 'goto', linel.result, None, None])
					else:
						block.pop(idx_l-offset)
						offset+=1
	tot_icg = remove_unreachable(tot_icg)
	return tot_icg
```

The remove_unreachable function identifies unreachable code blocks by traversing the intermediate code and marking the target labels of 'goto' and 'If False' statements as reachable. It then removes any code blocks whose labels are not marked as reachable, effectively eliminating dead code paths.
The peephole function further optimizes the intermediate code by applying peephole optimization. It first removes unreachable code using the remove_unreachable function. Then, it iterates through the code blocks, identifying 'If False' statements with constant conditions ('True' or 'False'). If found, it replaces the 'If False' statement with a corresponding 'goto' statement, effectively bypassing unnecessary conditional checks.

Sample input:

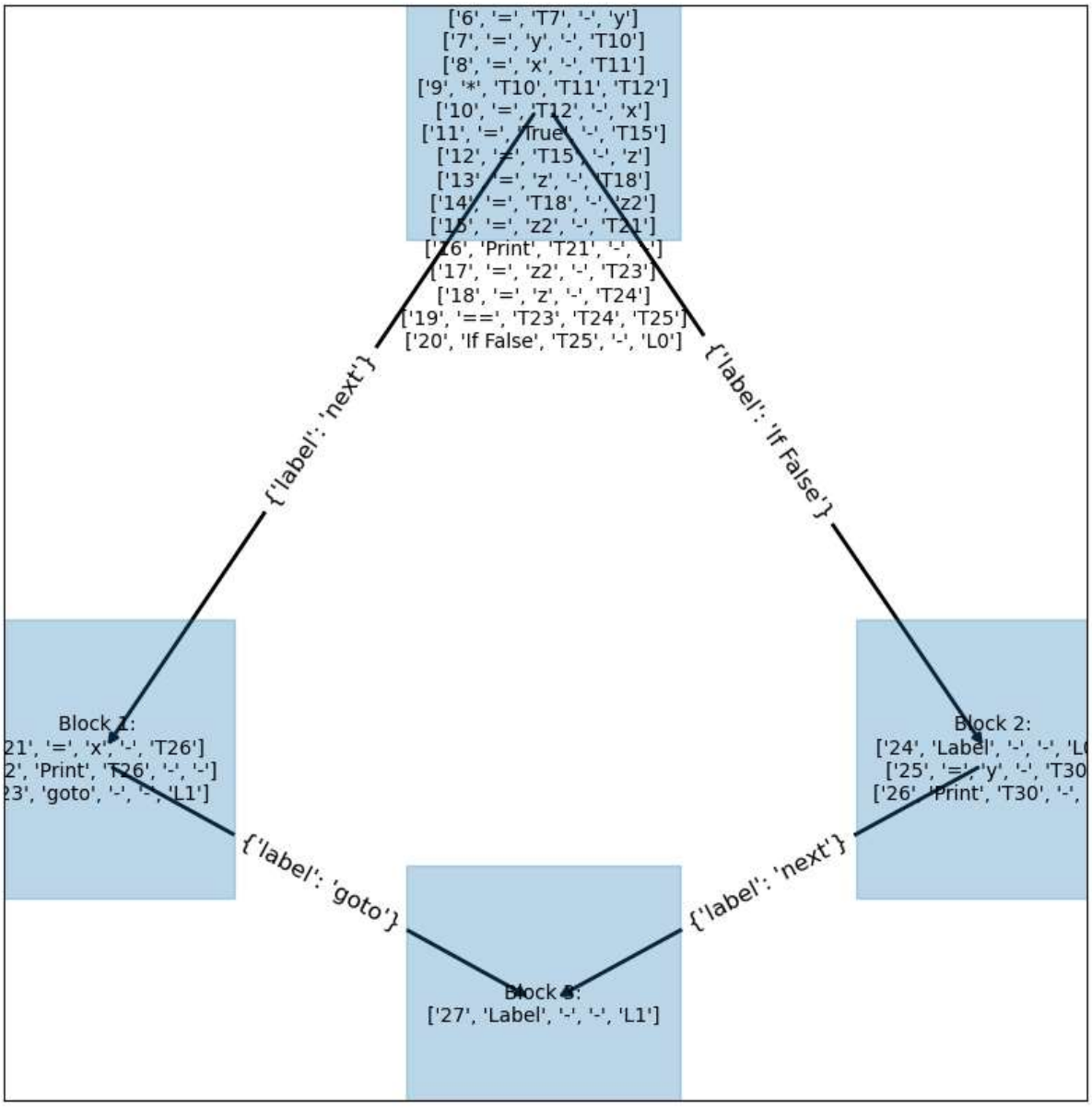
```
import hWorld
x=3*2
y="10"
x=y*x
z=True
z2=z
print(z2)

if(z2==z):
	print(x)
else:
	print(y)
```

Three address code (Before optimization):

```
--------------------------------All Quads--------------------------------
0	import hWorld	-	-
1	=	3	-	T2
2	=	2	-	T3
3	*	T2	T3	T4
4	=	T4	-	x
5	=	"10"	-	T7
6	=	T7	-	y
7	=	y	-	T10
8	=	x	-	T11
9	*	T10	T11	T12
10	=	T12	-	x
```

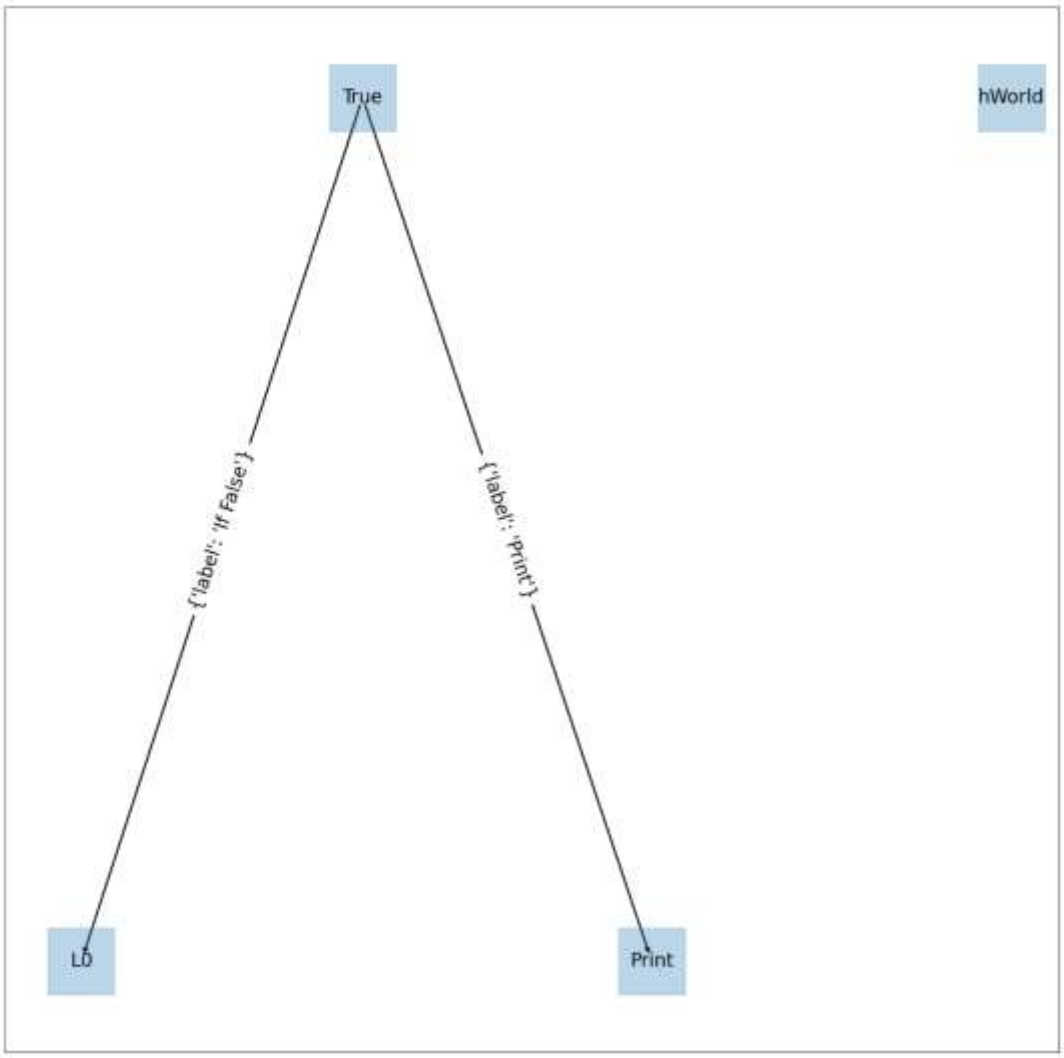
11	=	True	-	T15
12	=	T15	-	z
13	=	z	-	T18
14	=	T18	-	z2
15	=	z2	-	T21
16	Print	T21	-	-
17	=	z2	-	T23
18	=	z	-	T24
19	==	T23	T24	T25
20	If False	T25	-	L0
21	=	x	-	T26
22	Print	T26	-	-
23	goto	-	-	L1
24	Label	-	-	L0
25	=	y	-	T30
26	Print	T30	-	-
27	Label	-	-	L1



Three address code (After optimization):

After copy propagation, constant folding, dead code elimination, we get:

```
[[0: import, hWorld, None, None,
  16: Print, True, None, None,
  20: If False, True, None, L0],
 [21: =, x, None, T26, 22: Print, T26, None, None, 23: goto, None, None, L1],
 [24: Label, None, None, L0, 25: =, y, None, T30, 26: Print, T30, None, None],
 [27: Label, None, None, L1]]
```

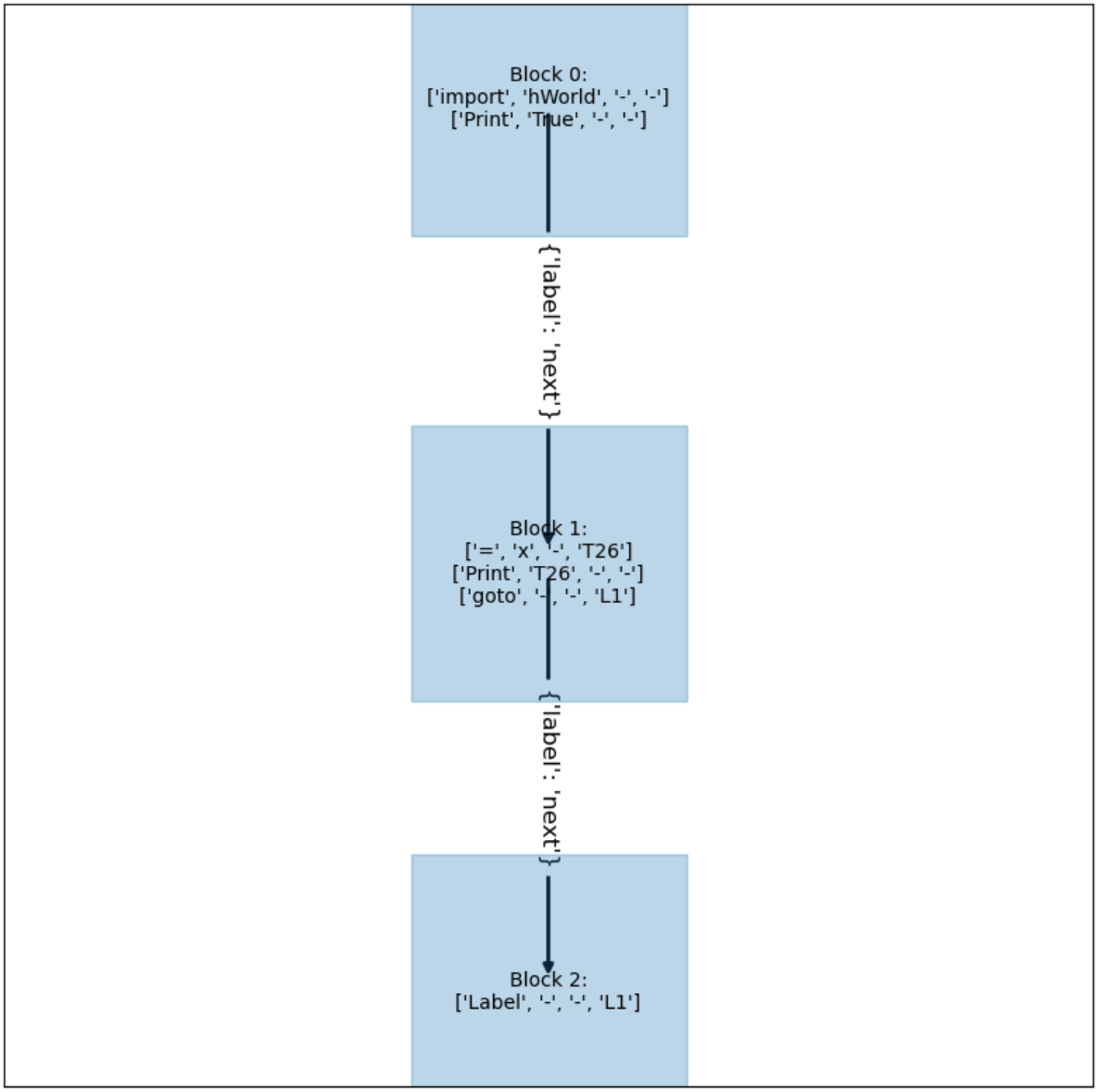


After peephole:

```
[[['import', 'hWorld', '-', '-'], ['Print', 'True', '-', '-']],
 [['=', 'x', '-', 'T26'],
  ['Print', 'T26', '-', '-'],
  ['goto', '-', '-', 'L1']],
 [['Label', '-', '-', 'L1']]]
```

Code after Peephole Optimization

```
import hWorld
z=True
z2=True
print(True)
if(z2==z):
        print("101010101010")
```



C) i) Implement the recursive QuickSort algorithm in your language taking the first element as the pivot.

```
array = [7,6,10,5,9,2,1,15,7]
#QuickSort algo implementation with first element as pivot

def quick_sort(array, low, high):
    if(low < high):
        pivot = array[low]
        start = low + 1
        end = high
        while True:
            while start <= end and array[end] >= pivot:
                end = end - 1
            while start <= end and array[start] <= pivot:
                start = start + 1
            if start <= end:
                temp = array[start]
                temp1 = array[end]
                array[start] = temp1
                array[end] = temp
            else:
                break
        temp = array[low]
        temp1 = array[end]
        array[low] = temp1
        array[end] = temp
        idx = end-1
        quick_sort(array, start, idx)
        idx = end+1
        quick_sort(array, idx, end)
    return

quick_sort(array, 0, 8)
```

ii) Show all steps of compiler step by step: Lexical Analysis, Syntax and Semantic Analysis, Intermediate Code Generation, and Code Optimization.

Lexical Analysis Output – Symbol Table

Scope	Name	Type	Declaration	Last Used Line
(0, 1)	1	Constant	1	10
(0, 1)	low	Identifier	1	32
(0, 1)	10	Constant	2	10
(0, 1)	x	Identifier	2	2
(0, 1)	y	Identifier	3	3
(0, 1)	high	Identifier	4	17
(0, 1)	temp	Identifier	5	27
(0, 1)	temp1	Identifier	6	26
(0, 1)	idx	Identifier	7	7
(0, 1)	start	Identifier	8	8
(0, 1)	end	Identifier	9	9
(0, 1)	6	Constant	10	10
(0, 1)	5	Constant	10	10
(0, 1)	9	Constant	10	10
(0, 1)	2	Constant	10	10
(0, 1)	15	Constant	10	10
(0, 1)	7	Constant	10	10
(0, 1)	array	ListTypeID	10	33
(0, 1)	quick_sort	Func_Name	13	13
(0, 1)	T0	ICGTempVar	-1	-1
(0, 1)	T1	ICGTempVar	-1	-1
(0, 1)	T4	ICGTempVar	-1	-1
(0, 1)	T7	ICGTempVar	-1	-1
(0, 1)	T10	ICGTempVar	-1	-1
(0, 1)	T11	ICGTempVar	-1	-1
(0, 1)	T14	ICGTempVar	-1	-1
(0, 1)	T15	ICGTempVar	-1	-1
(0, 1)	T18	ICGTempVar	-1	-1
(0, 1)	T19	ICGTempVar	-1	-1
(0, 1)	T22	ICGTempVar	-1	-1
(0, 1)	T23	ICGTempVar	-1	-1
(0, 1)	T26	ICGTempVar	-1	-1
(0, 1)	T27	ICGTempVar	-1	-1
(0, 1)	T30	ICGTempVar	-1	-1
(0, 1)	T31	ICGTempVar	-1	-1
(0, 1)	T45	ICGTempVar	-1	-1
(0, 1)	T46	ICGTempVar	-1	-1
(0, 1)	T47	ICGTempVar	-1	-1
(0, 1)	L0	ICGTempLabel	-1	-1
(0, 1)	T50	ICGTempVar	-1	-1
(0, 1)	T53	ICGTempVar	-1	-1
(0, 1)	T54	ICGTempVar	-1	-1
(0, 1)	T55	ICGTempVar	-1	-1
(0, 1)	T58	ICGTempVar	-1	-1
(0, 1)	T61	ICGTempVar	-1	-1
(0, 1)	L1	ICGTempLabel	-1	-1
(0, 1)	L2	ICGTempLabel	-1	-1

(0, 1)	T62	ICGTempVar	-1	-1
(0, 1)	T63	ICGTempVar	-1	-1
(0, 1)	T64	ICGTempVar	-1	-1
(0, 1)	T67	ICGTempVar	-1	-1
(0, 1)	T68	ICGTempVar	-1	-1
(0, 1)	T69	ICGTempVar	-1	-1
(0, 1)	T70	ICGTempVar	-1	-1
(0, 1)	L3	ICGTempLabel	-1	-1
(0, 1)	L4	ICGTempLabel	-1	-1
(0, 1)	T71	ICGTempVar	-1	-1
(0, 1)	T72	ICGTempVar	-1	-1
(0, 1)	T73	ICGTempVar	-1	-1
(0, 1)	T76	ICGTempVar	-1	-1
(0, 1)	T77	ICGTempVar	-1	-1
(0, 1)	T78	ICGTempVar	-1	-1
(0, 1)	T81	ICGTempVar	-1	-1
(0, 1)	T82	ICGTempVar	-1	-1
(0, 1)	T83	ICGTempVar	-1	-1
(0, 1)	T84	ICGTempVar	-1	-1
(0, 1)	L5	ICGTempLabel	-1	-1
(0, 1)	L6	ICGTempLabel	-1	-1
(0, 1)	T85	ICGTempVar	-1	-1
(0, 1)	T86	ICGTempVar	-1	-1
(0, 1)	T87	ICGTempVar	-1	-1
(0, 1)	T90	ICGTempVar	-1	-1
(0, 1)	T91	ICGTempVar	-1	-1
(0, 1)	T92	ICGTempVar	-1	-1
(0, 1)	L7	ICGTempLabel	-1	-1
(0, 1)	T95	ICGTempVar	-1	-1
(0, 1)	T100	ICGTempVar	-1	-1
(0, 1)	L8	ICGTempLabel	-1	-1
(0, 1)	T119	ICGTempVar	-1	-1
(0, 1)	T134	ICGTempVar	-1	-1
(0, 1)	T145	ICGTempVar	-1	-1
(0, 1)	T146	ICGTempVar	-1	-1
(0, 1)	T147	ICGTempVar	-1	-1
(0, 1)	T154	ICGTempVar	-1	-1
(0, 1)	T155	ICGTempVar	-1	-1
(0, 1)	T156	ICGTempVar	-1	-1
(0, 1)	T157	ICGTempVar	-1	-1
(0, 1)	T164	ICGTempVar	-1	-1
(0, 1)	T190	ICGTempVar	-1	-1
(0, 2)	0	Constant	41	41
(0, 2)	8	Constant	41	41
(1, 3)	pivot	Identifier	15	21
(1, 3)	1	Constant	16	36
(1, 3)	start	Identifier	16	35
(1, 3)	end	Identifier	17	37
(1, 3)	True	Constant	18	18
(1, 3)	temp	Identifier	30	33
(1, 3)	temp1	Identifier	31	32
(1, 3)	idx	Identifier	34	37
(3, 5)	1	Constant	20	20
(3, 5)	end	Identifier	20	27

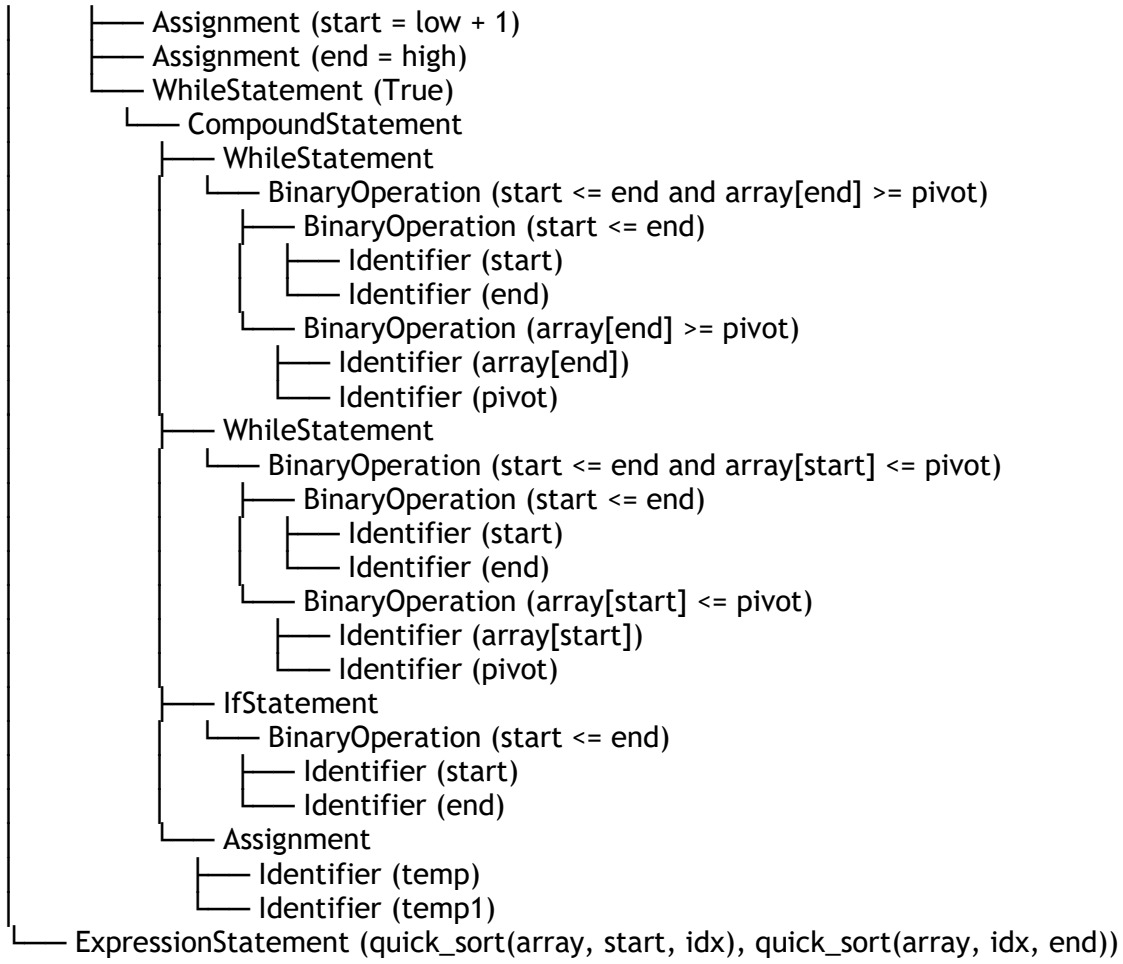
```
(3, 25) 1       Constant        22              22
(3, 25) start   Identifier      22              22
(3, 125)        temp    Identifier      24              24
(3, 125)        temp1   Identifier      25              25
```

Syntax Analysis – parse tree

```
Start
    └── FuncDef (quick_sort)
        ├── Identifier (array)
        ├── Identifier (low)
        ├── Identifier (high)
        └── CompoundStatement
            ├── IfStatement (low < high)
            │   └── CompoundStatement
            │       ├── Assignment (pivot = array[low])
            │       ├── Assignment (start = low + 1)
            │       ├── Assignment (end = high)
            │       └── WhileStatement (True)
            │           └── CompoundStatement
            │               ├── WhileStatement (start <= end and array[end] >= pivot)
            │               │   └── CompoundStatement
            │               │       └── Assignment (end = end - 1)
            │               ├── WhileStatement (start <= end and array[start] <= pivot)
            │               │   └── CompoundStatement
            │               │       └── Assignment (start = start + 1)
            │               ├── IfStatement (start <= end)
            │               │   └── CompoundStatement
            │               │       └── CompoundStatement
            │               │           ├── Assignment (temp = array[start])
            │               │           │   └── Assignment (array[start] = array[end]; array[end] =
temp)
            │               └── BreakStatement
            └── ExpressionStatement
                └── FunctionCall (quick_sort)
                    ├── Identifier (array)
                    ├── Identifier (start)
                    └── Identifier (idx)
```

Syntax Analysis – AST

```
Start
└── FuncDef (quick_sort)
    ├── Identifier (array)
    ├── Identifier (low)
    ├── Identifier (high)
    └── CompoundStatement
        ├── IfStatement
        │   ├── BinaryOperation (low < high)
        │   │   ├── Identifier (low)
        │   │   └── Identifier (high)
        │   └── CompoundStatement
        │       ├── Assignment (pivot = array[low])
```



ICG Output – three address code (quadruples)

```
T0 = 1
T1 = - T0
low = T1
T4 = 10
x = T4
T7 = 10
y = T7
T10 = 1
T11 = - T10
high = T11
T14 = 1
T15 = - T14
temp = T15
T18 = 1
T19 = - T18
temp1 = T19
T22 = 1
T23 = - T22
idx = T23
T26 = 1
T27 = - T26
start = T27
```

```
T30 = 1
T31 = - T30
end = T31
Begin Function quick_sort
T45 = low
T46 = high
T47 = T45 < T46
If False T47 goto L0
T50 = array[low]
pivot = T50
T53 = low
T54 = 1
T55 = T53 + T54
start = T55
T58 = high
end = T58
T61 = True
L1: If False T61 goto L2
T62 = start
T63 = end
T64 = T62 <= T63
T67 = array[end]
T68 = pivot
T69 = T67 >= T68
T70 = T64 and T69
L3: If False T70 goto L4
T71 = end
T72 = 1
T73 = T71 - T72
end = T73
T76 = start
T77 = end
T78 = T76 <= T77
T81 = array[start]
T82 = pivot
T83 = T81 <= T82
T84 = T78 and T83
L5: If False T84 goto L6
T85 = start
T86 = 1
T87 = T85 + T86
start = T87
T90 = start
T91 = end
T92 = T90 <= T91
If False T92 goto L7
T95 = array[start]
temp = T95
T100 = array[end]
temp1 = T100
array[start] = temp1
array[end] = temp
goto L8
L7: goto L7
```

```
T119 = array[low]
temp = T119
L8: goto L5
L6: goto L3
L4: T134 = array[end]
temp1 = T134
array[low] = temp1
array[end] = temp
T145 = end
T146 = 1
T147 = T145 - T146
idx = T147
Push Param array
Push Param  array
Push Param  start
(T154)Call Function quick_sort, 3
Pop Params for Function quick_sort, 3
T155 = end
T156 = 1
T157 = T155 + T156
idx = T157
Push Param array
Push Param  array
Push Param  idx
(T164)Call Function quick_sort, 3
Pop Params for Function quick_sort, 3
return
goto L1
L2: L0: End Function quick_sort
Push Param array
Push Param  array
Push Param  0
(T190)Call Function quick_sort, 3
Pop Params for Function quick_sort, 3
```

	OP	ARG1	ARG2	RES
0	=	1	-	T0
1	-	T0	-	T1
2	=	T1	-	low
7	=	1	-	T10
8	-	T10	-	T11
9	=	T11	-	high
10	=	1	-	T14
11	-	T14	-	T15
12	=	T15	-	temp
13	=	1	-	T18
14	-	T18	-	T19
15	=	T19	-	temp1
19	=	1	-	T26
20	-	T26	-	T27
21	=	T27	-	start
22	=	1	-	T30
23	-	T30	-	T31

```
24    =     T31    -     end
25    BeginF  quick_sort    -     -
26    =     low    -     T45
27    =     high    -     T46
28    <     T45    T46    T47
29    If False     T47    -     L0
30    ListIndex     array   low    T50
31    =     T50    -     pivot
32    =     low    -     T53
33    =     1     -     T54
34    +     T53    T54    T55
35    =     T55    -     start
36    =     high    -     T58
37    =     T58    -     end
38    =     True    -     T61
39    Label   -     -     L1
40    If False     T61    -     L2
41    =     start   -     T62
42    =     end    -     T63
43    <=     T62    T63    T64
44    ListIndex     array   end    T67
45    =     pivot   -     T68
46    >=     T67    T68    T69
47    and    T64    T69    T70
48    Label   -     -     L3
49    If False     T70    -     L4
50    =     end    -     T71
51    =     1     -     T72
52    -     T71    T72    T73
53    =     T73    -     end
54    =     start   -     T76
55    =     end    -     T77
56    <=     T76    T77    T78
57    ListIndex     array   start   T81
58    =     pivot   -     T82
59    <=     T81    T82    T83
60    and    T78    T83    T84
61    Label   -     -     L5
62    If False     T84    -     L6
63    =     start   -     T85
64    =     1     -     T86
65    +     T85    T86    T87
66    =     T87    -     start
67    =     start   -     T90
68    =     end    -     T91
69    <=     T90    T91    T92
70    If False     T92    -     L7
71    ListIndex     array   start   T95
72    =     T95    -     temp
73    ListIndex     array   end    T100
74    =     T100    -     temp1
75    ListAssign     start   temp1   array
76    ListAssign     end    temp    array
77    goto    -     -     L8
```

```
78      Label   -       -       L7
87      ListIndex       array   end     T134
88      =       T134    -       temp1
89      ListAssign      low     temp1   array
90      ListAssign      end     temp    array
95      Param   array   -       -
96      Param   array   -       -
97      Param   start   -       -
98      Call    quick_sort      3       T154
103     Param   array   -       -
104     Param   array   -       -
105     Param   idx     -       -
106     Call    quick_sort      3       T164
107     return  -       -       -
108     goto    -       -       L1
109     Label   -       -       L2
110     Label   -       -       L0
111     EndF    quick_sort      -       -
112     Param   array   -       -
113     Param   array   -       -
114     Param   0       -       -
115     Call    quick_sort      3       T190
```

---

iii) Show the flow graph and DAG for this code.

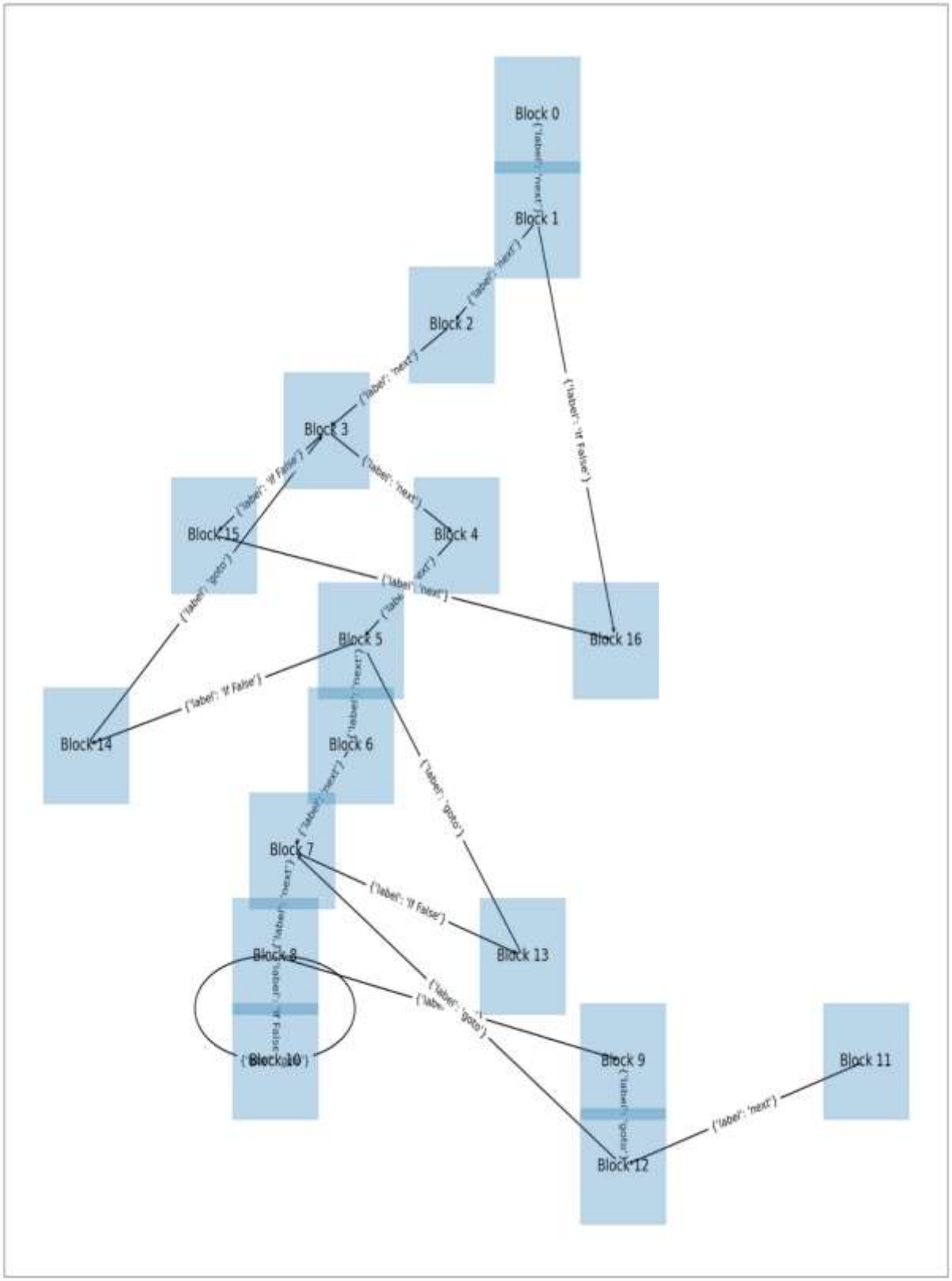
Basic Blocks

```
('Block 0:\n'
 "['0', '=', '1', None, 'T10']\n" - leader
 "['1', '-', 'T10', None, 'T11']\n"
 "['2', '=', 'T11', None, 'low']\n"
 "['7', '=', '1', None, 'T20']\n"
 "['8', '-', 'T20', None, 'T21']\n"
 "['9', '=', 'T21', None, 'high']\n"
 "['10', '=', '1', None, 'T24']\n"
 "['11', '-', 'T24', None, 'T25']\n"
 "['12', '=', 'T25', None, 'temp']\n"
 "['13', '=', '1', None, 'T28']\n"
 "['14', '-', 'T28', None, 'T29']\n"
 "['15', '=', 'T29', None, 'temp1']\n"
 "['106', 'Param', 'array', None, None]\n"
 "['107', 'Param', 'array', None, None]\n"
 "['108', 'Param', '0', None, None]\n"
 "['109', 'Call', 'quick_sort', '3', 'T182']")
==============
('Block 1:\n'
 "['19', 'BeginF', 'quick_sort', None, None]\n"- leader
 "['20', '=', 'low', None, 'T37']\n"
 "['21', '=', 'high', None, 'T38']\n"
 "['22', '<', 'T37', 'T38', 'T39']\n"
 "['23', 'If False', 'T39', None, 'L0']")
==============
('Block 2:\n'
```

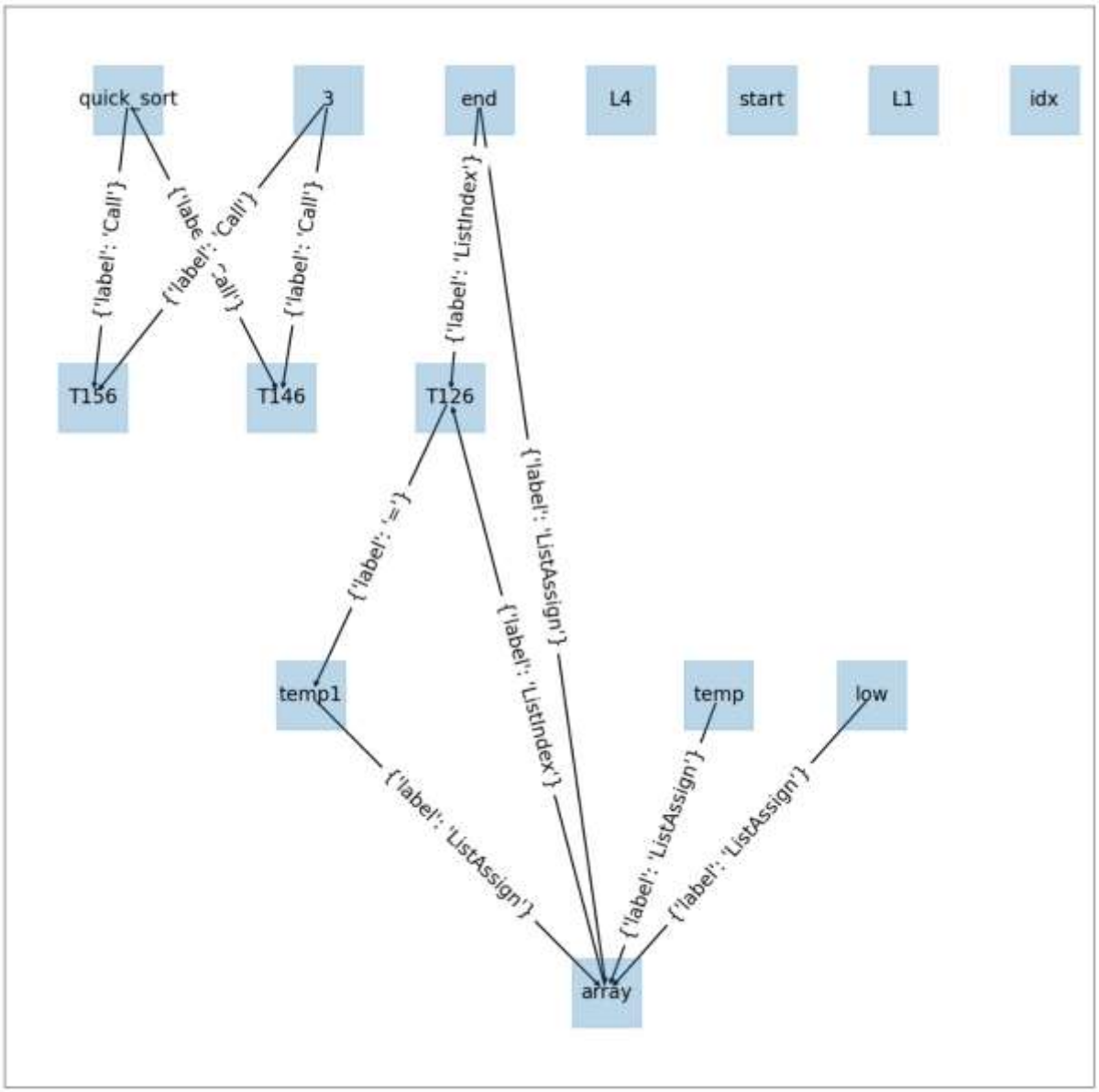
```
 "['24', 'ListIndex', 'array', 'low', 'T42']\n"- leader
 "['25', '=', 'T42', None, 'pivot']\n"
 "['26', '=', 'low', None, 'T45']\n"
 "['27', '=', '1', None, 'T46']\n"
 "['28', '+', 'T45', 'T46', 'T47']\n"
 "['29', '=', 'T47', None, 'start']\n"
 "['30', '=', 'high', None, 'T50']\n"
 "['31', '=', 'T50', None, 'end']\n"
 "['32', '=', 'True', None, 'T53']")
==============
('Block 3:\n'
 "['33', 'Label', None, None, 'L1']\n"- leader
 "['34', 'If False', 'T53', None, 'L2']")
==============
('Block 4:\n'
 "['35', '=', 'start', None, 'T54']\n"- leader
 "['36', '=', 'end', None, 'T55']\n"
 "['37', '<=', 'T54', 'T55', 'T56']\n"
 "['38', 'ListIndex', 'array', 'end', 'T59']\n"
 "['39', '=', 'pivot', None, 'T60']\n"
 "['40', '>=', 'T59', 'T60', 'T61']\n"
 "['41', 'and', 'T56', 'T61', 'T62']")
==============
('Block 5:\n'
 "['42', 'Label', None, None, 'L3']\n"- leader
 "['43', 'If False', 'T62', None, 'L4']")
==============
('Block 6:\n'
 "['44', '=', 'end', None, 'T63']\n"- leader
 "['45', '=', '1', None, 'T64']\n"
 "['46', '-', 'T63', 'T64', 'T65']\n"
 "['47', '=', 'T65', None, 'end']\n"
 "['48', '=', 'start', None, 'T68']\n"
 "['49', '=', 'end', None, 'T69']\n"
 "['50', '<=', 'T68', 'T69', 'T70']\n"
 "['51', 'ListIndex', 'array', 'start', 'T73']\n"
 "['52', '=', 'pivot', None, 'T74']\n"
 "['53', '<=', 'T73', 'T74', 'T75']\n"
 "['54', 'and', 'T70', 'T75', 'T76']")
==============
('Block 7:\n'
 "['55', 'Label', None, None, 'L5']\n"- leader
 "['56', 'If False', 'T76', None, 'L6']")
==============
('Block 8:\n'
 "['57', '=', 'start', None, 'T77']\n"- leader
 "['58', '=', '1', None, 'T78']\n"
 "['59', '+', 'T77', 'T78', 'T79']\n"
 "['60', '=', 'T79', None, 'start']\n"
 "['61', '=', 'start', None, 'T82']\n"
 "['62', '=', 'end', None, 'T83']\n"
 "['63', '<=', 'T82', 'T83', 'T84']\n"
 "['64', 'If False', 'T84', None, 'L7']")
==============
```

('Block 9:\n'
 "['65', 'ListIndex', 'array', 'start', 'T87']\n"- leader
 "['66', '=', 'T87', None, 'temp']\n"
 "['67', 'ListIndex', 'array', 'end', 'T92']\n"
 "['68', '=', 'T92', None, 'temp1']\n"
 "['69', 'ListAssign', 'start', 'temp1', 'array']\n"
 "['70', 'ListAssign', 'end', 'temp', 'array']\n"
 "['71', 'goto', None, None, 'L8']")
==========
"Block 10:\n
['72', 'Label', None, None, 'L7']\n - leader
['73', 'goto', None, None, 'L7']"
==========
('Block 11:\n'
 "['74', 'ListIndex', 'array', 'low', 'T111']\n"- leader
 "['75', '=', 'T111', None, 'temp']")
==========
"Block 12:\n
['76', 'Label', None, None, 'L8']\n- leader
['77', 'goto', None, None, 'L5']"
==========
"Block 13:\n
['78', 'Label', None, None, 'L6']\n - leader
['79', 'goto', None, None, 'L3']"
==========
('Block 14:\n'
 "['80', 'Label', None, None, 'L4']\n" - leader
 "['81', 'ListIndex', 'array', 'end', 'T126']\n"
 "['82', '=', 'T126', None, 'temp1']\n"
 "['83', 'ListAssign', 'low', 'temp1', 'array']\n"
 "['84', 'ListAssign', 'end', 'temp', 'array']\n"
 "['89', 'Param', 'array', None, None]\n"
 "['90', 'Param', 'array', None, None]\n"
 "['91', 'Param', 'start', None, None]\n"
 "['92', 'Call', 'quick_sort', '3', 'T146']\n"
 "['97', 'Param', 'array', None, None]\n"
 "['98', 'Param', 'array', None, None]\n"
 "['99', 'Param', 'idx', None, None]\n"
 "['100', 'Call', 'quick_sort', '3', 'T156']\n"
 "['101', 'return', None, None, None]\n"
 "['102', 'goto', None, None, 'L1']")
==========
"Block 15:\n['103', 'Label', None, None, 'L2']" - leader
==========
('Block 16:\n'
 "['104', 'Label', None, None, 'L0']\n" - leader
 "['105', 'EndF', 'quick_sort', None, None]")
==========

Control Flow Graph for Quicksort:



DAG for Quicksort (recursive call block)



iv) Draw all steps for optimizing the code.

Optimizing the Quicksort algorithm involves enhancing its efficiency and reducing unnecessary overhead. Loop unrolling is a technique used to reduce loop overhead by executing multiple iterations of a loop in a single iteration. By manually expanding the loop body, we can reduce loop control and branching overhead. In the context of Quicksort, unrolling the partitioning loop can enhance cache utilization and reduce the number of comparisons, leading to improved performance.

Loop unrolling:

```
quicksort_unrolled(array, low, high):
    while low < high:
        if high - low < INSERTION_THRESHOLD:
```

```
        insertion_sort(array, low, high)
        return
    pivot_index = partition(array, low, high)
    if pivot_index - low < high - pivot_index:
        quicksort_unrolled(array, low, pivot_index - 1)
        low = pivot_index + 1
    else:
        quicksort_unrolled(array, pivot_index + 1, high)
        high = pivot_index - 1
```

In the pseudocode for loop unrolling, the quicksort_unrolled function incorporates a mechanism to switch between partitions based on their sizes to balance the workload and optimize performance.