Name: Adwait Hegde
Roll No: 2019130019
TE Comp (Batch-A)

EXPERIMENT 10

Aim:

Write a program to find Basic blocks and generate flow graph for the given three address code.

Theory:

Algorithm 9.1. Partition into basic blocks.

Input. A sequence of three-address statements.

Output. A list of basic blocks with each three-address statement in exactly one block.

Method.

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are the following.
   i) The first statement is a leader.
   ii) Any statement that is the target of a conditional or unconditional goto is a leader.
   iii) Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program. □

Example 9.3. Consider the fragment of source code shown in Fig. 9.7; it computes the dot product of two vectors `a` and `b` of length 20. A list of three-address statements performing this computation on our target machine is shown in Fig. 9.8.

```
begin
     prod := 0;
     i := 1;
     do begin
        prod := prod + a[i] * b[i];
        i := i + 1
     end
     while i <= 20
end
```

Fig. 9.7. Program to compute dot product.

Let us apply Algorithm 9.1 to the three-address code in Fig. 9.8 to determine its basic blocks. Statement (1) is a leader by rule (i) and statement (3) is a leader by rule (ii), since the last statement can jump to it. By rule (iii) the statement following (12) (recall that Fig. 9.13 is just a fragment of a program) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block. □

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]          /* compute a[i] */
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]          /* compute b[i] */
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

Fig. 9.8. Three-address code computing dot product.

Flow Graphs

We can add the flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a *flow graph*. The nodes of the flow graph are the basic blocks. One node is distinguished as *initial*; it is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence; that is, if

1. there is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2, or
2. B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.

We say that B_1 is a *predecessor* of B_2, and B_2 is a *successor* of B_1.

Example 9.4. The flow graph of the program of Fig. 9.7 is shown in Fig. 9.9. B_1 is the initial node. Note that in the last statement, the jump to statement (3) has been replaced by an equivalent jump to the beginning of block B_2. □

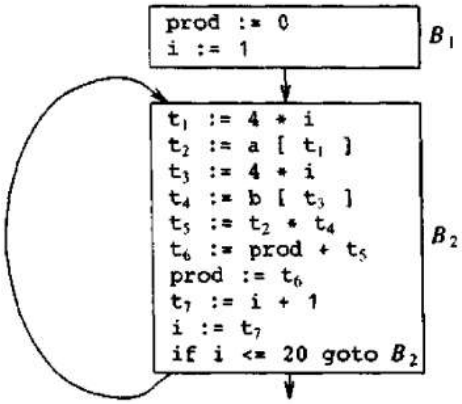


Fig. 9.9. Flow graph for program.

of the block, and by the lists of predecessors and successors of the block. An alternative is to make a linked list of the quadruples in each block. Explicit references to quadruple numbers in jump statements at the end of basic blocks can cause problems if quadruples are moved during code optimization. For example, if the block B_2 running from statements (3) through (12) in the intermediate code of Fig. 9.9 were moved elsewhere in the quadruple array or were shrunk, the (3) in `if i <= 20 goto (3)` would have to be changed. Thus, we prefer to make jumps point to blocks rather than quadruples, as we have done in Fig. 9.9.

It is important to note that an edge of the flow graph from block B to block B' does not specify the conditions under which control flows from B to B'. That is, the edge does not tell whether the conditional jump at the end of B (if there is a conditional jump there) goes to the leader of B' when the condition is satisfied or when the condition is not satisfied. That information can be recovered when needed from the jump statement in B.

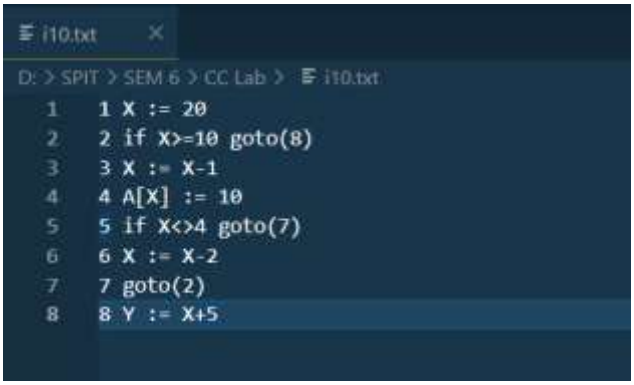
Code:

```
import re

inp=[]
f = open("i10.txt", "r")
for x in f:
    inp.append(x)

leaders = [1]
nodes=[]

for j, i in enumerate(inp):
    z = re.findall(r'goto\([0-9]\)',i)
    if len(z)>0:
        leader = int(z[0][5:-1])

        if 'if ' in i:
            nodes.append((j+1,j+2))
            print("Adding ",(j+1,j+2), "to nodes list")
        elif (j+1,j+2) in nodes:
            nodes.remove((j+1,j+2))
            print("Removing ",(j+1,j+2), "from nodes list")

        nodes.append((j+1,leader))
        print("Adding ",(j+1,leader), "to nodes list")

        if leader not in leaders:
            leaders.append(leader)
        if (leader-1,leader) not in nodes:
            print("Adding ",(leader-1,leader), "to nodes list")
            nodes.append((leader-1,leader))

        if j+2 not in leaders:
            leaders.append(j+2)

print()
print("====NODES====")
nodes = list(set(nodes))
print(nodes,'\n')
print("===LEADER===")
leaders.sort()
print(leaders,'\n')

blocks=[]
```

```
s = 1
for i in leaders[1:]:
    blocks.append(inp[s-1:i-1])
    s=i
blocks.append(inp[s-1:])


for i, b in enumerate(blocks):
    print("Block: B"+str(i+1),"->",b)


adjmat = [[] for i in blocks]

def getblock(ip):
    res = [-1,-1]
    global leaders
    for i, lead in enumerate(leaders):
        if ip[0]>= lead:
            res[0] = i+1
        if ip[1]>= lead:
            res[1] = i+1
    return res

for i in nodes:
    res = getblock(i)
    adjmat[res[0]].append(res[1])

print()
print("===ADJ.MAT.===")
print(adjmat[1:])
print()

print("--------OUTPUT---------")
for i, block in enumerate(blocks):
    for b in block:
        if 'goto' in b:
            z=re.findall(r'goto\([0-9]\)',b)
            kkk = int(z[0][5:-1])
            ind = b.find('goto(')
            print(f"{b[:ind+5]} B{getblock((kkk,kkk))[0]} )")
        else:
            print(b)
print("-----------------------")
```

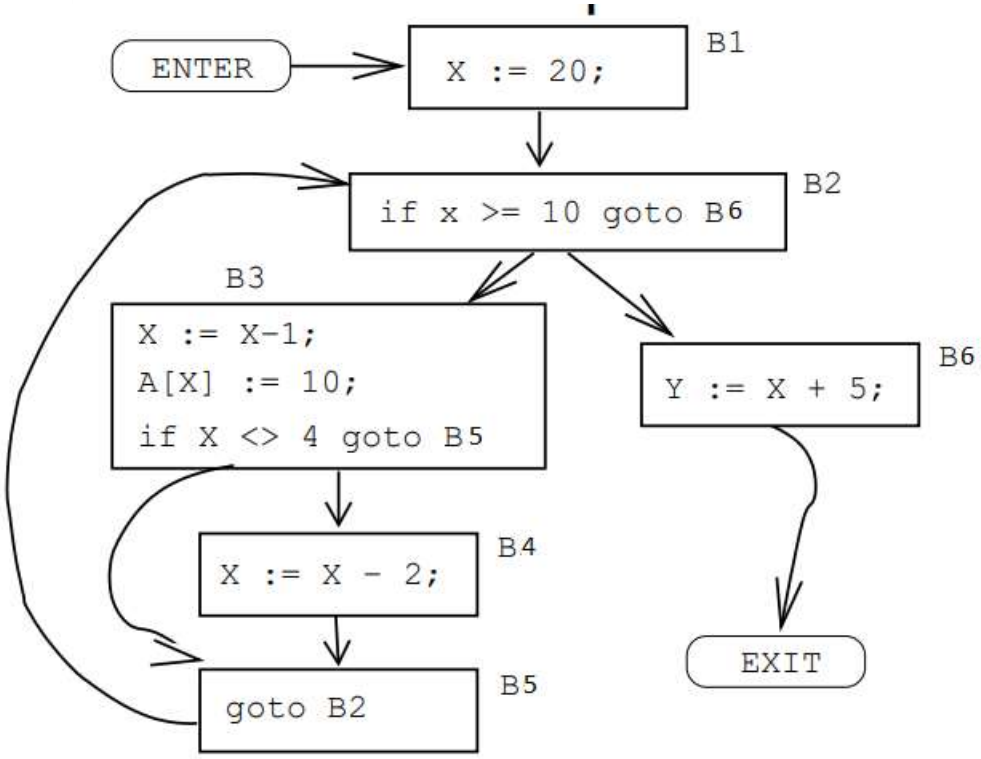
Result:

Input:



Output:

```
PS D:\SPIT\SEM 6\CC Lab> py ex10.py
Adding  (2, 3) to nodes list
Adding  (2, 8) to nodes list
Adding  (7, 8) to nodes list
Adding  (5, 6) to nodes list
Adding  (5, 7) to nodes list
Adding  (6, 7) to nodes list
Removing  (7, 8) from nodes list
Adding  (7, 2) to nodes list
Adding  (1, 2) to nodes list

====NODES====
[(1, 2), (5, 7), (2, 3), (6, 7), (5, 6), (7, 2), (2, 8)]

===LEADER===
[1, 2, 3, 6, 7, 8]

Block: B1 -> ['1 X := 20']
Block: B2 -> ['2 if X>=10 goto(8)']
Block: B3 -> ['3 X := X-1', '4 A[X] := 10', '5 if X<>4 goto(7)']
Block: B4 -> ['6 X := X-2']
Block: B5 -> ['7 goto(2)']
Block: B6 -> ['8 Y := X+5']

[[2], [3, 6], [5, 4], [5], [2]]

--------OUTPUT---------
1 X := 20
2 if X>=10 goto( B6 )
3 X := X-1
4 A[X] := 10
5 if X<>4 goto( B5 )
6 X := X-2
7 goto( B2 )
8 Y := X+5
-----------------------
```



Source: https://www2.cs.arizona.edu/~collberg/Teaching/453/2009/Handouts/Handout-15.pdf

Conclusion:

In this Experiment, I learned about the concept of Basic Blocks and Flow graphs and implemented an algorithm to find Basic blocks and generate flow graph for the given three address code.