Lecture Overview

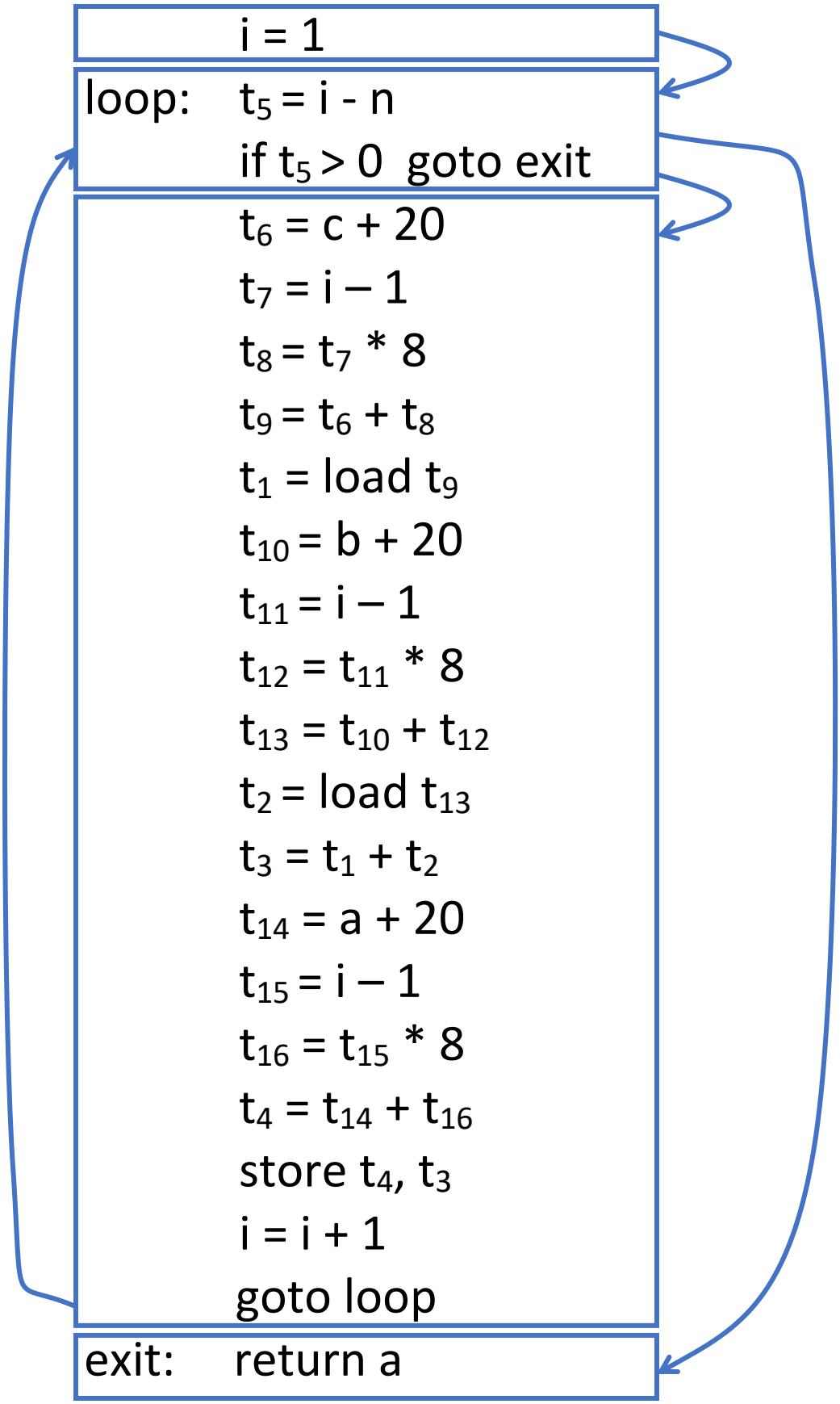
Optimization

- **Global Optimization**
  - **Live variables – dataflow**
  - **Possibly uninitialized variables**
  - **Global Code Placement**

[Chapter 8]

```
for i = 1 to n
      a[i] = b[i] + c[i]                    // arrays are 1..n
return a
```

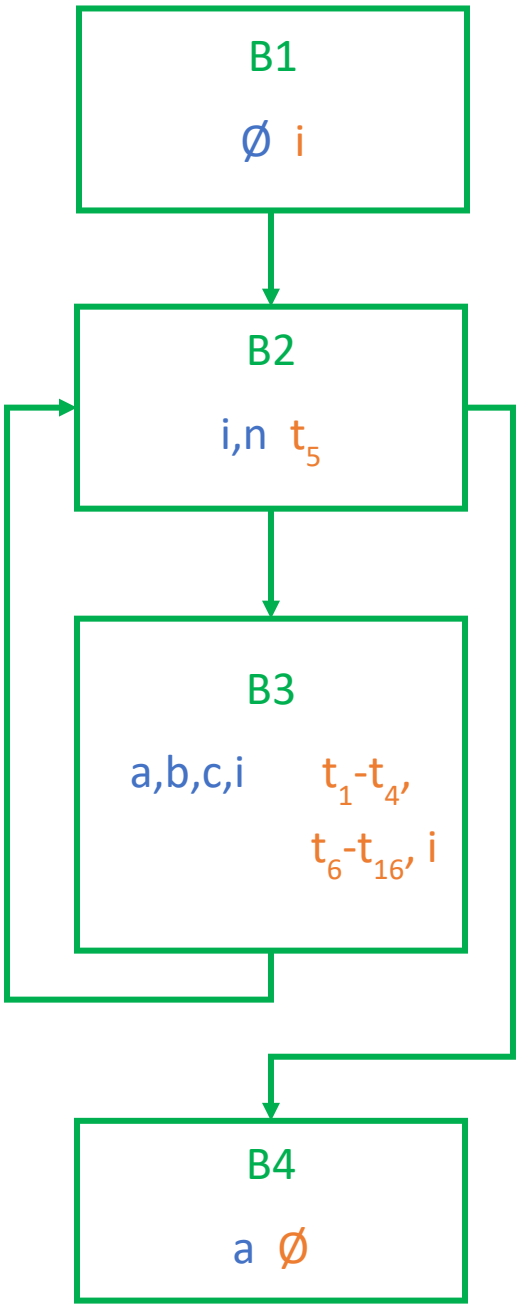
```
            i = 1
loop:       if i > n goto exit
            t1 = c[i]       // c + 20 (header) + (i-1)*8
            t2 = b[i]
            t3 = t1 + t2
            t4 = &a[i]
            store t4, t3
            i = i + 1
            goto loop
exit:       return a
```


i = 1
loop: $t_5 = i - n$
if $t_5 > 0$ goto exit
$t_6 = c + 20$
$t_7 = i - 1$
$t_8 = t_7 * 8$
$t_9 = t_6 + t_8$
t_1 = load t_9
$t_{10} = b + 20$
$t_{11} = i - 1$
$t_{12} = t_{11} * 8$
$t_{13} = t_{10} + t_{12}$
t_2 = load t_{13}
$t_3 = t_1 + t_2$
$t_{14} = a + 20$
$t_{15} = i - 1$
$t_{16} = t_{15} * 8$
$t_4 = t_{14} + t_{16}$
store t_4, t_3
i = i + 1
goto loop
exit: return a

Block 3:	UEVar	VarKills
$t_6 = c + 20$	c	t_6
$t_7 = i - 1$	c, i	t_6, t_7
$t_8 = t_7 * 8$	c, i	t_6, t_7, t_8
$t_9 = t_6 + t_8$	c, i	t_6, t_7, t_8, t_9
$t_1 =$ load t_9	c, i	t_1, t_6, t_7, t_8, t_9
$t_{10} = b + 20$	b, c, i	t_1, t_6, t_7, t_8, t_9, t_{10}
$t_{11} = i - 1$	b, c, i	t_1, t_6, t_7, t_8, t_9, t_{10}, t_{11}
$t_{12} = t_{11} * 8$	b, c, i	t_1, t_6-t_{12}
$t_{13} = t_{10} + t_{12}$	b, c, i	t_1, t_6-t_{13}
$t_2 =$ load t_{13}	b, c, i	t_1, t_2, t_6-t_{13}
$t_3 = t_1 + t_2$	b, c, i	t_1, t_2, t_3, t_6-t_{13}
$t_{14} = a + 20$	a,b,c,i	t_1, t_2, t_3, t_6-t_{14}
$t_{15} = i - 1$	a,b,c,i	t_1, t_2, t_3, t_6-t_{15}
$t_{16} = t_{15} * 8$	a,b,c,i	t_1, t_2, t_3, t_6-t_{16}
$t_4 = t_{14} + t_{16}$	a,b,c,i	t_1-t_4, t_6-t_{16}
store t_4, t_3	a,b,c,i	t_1-t_4, t_6-t_{16}
$i = i + 1$	a,b,c,i	t_1-t_4, t_6-t_{16}, i

```
for each block b
    init(b)


init(b)
      UEVar(b) = Ø
      VarKill(b) = Ø
      for i = 1 to b.numInstr     // instruction i is
                                  // x[i] = y[i] op[i] z[i]
            if y[i] ∉ VarKill(b)
                  add y[i] to UEVar(b)
            if z[i] ∉ VarKill(b)
                  add z[i] to UEVar(b)
            add x[i] to VarKill(b)
```

$$\text{LiveOut}(n) = \bigcup_{m \in \text{succ}(n)} (\text{UEVar}(m) \cup (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}))$$



LiveOut

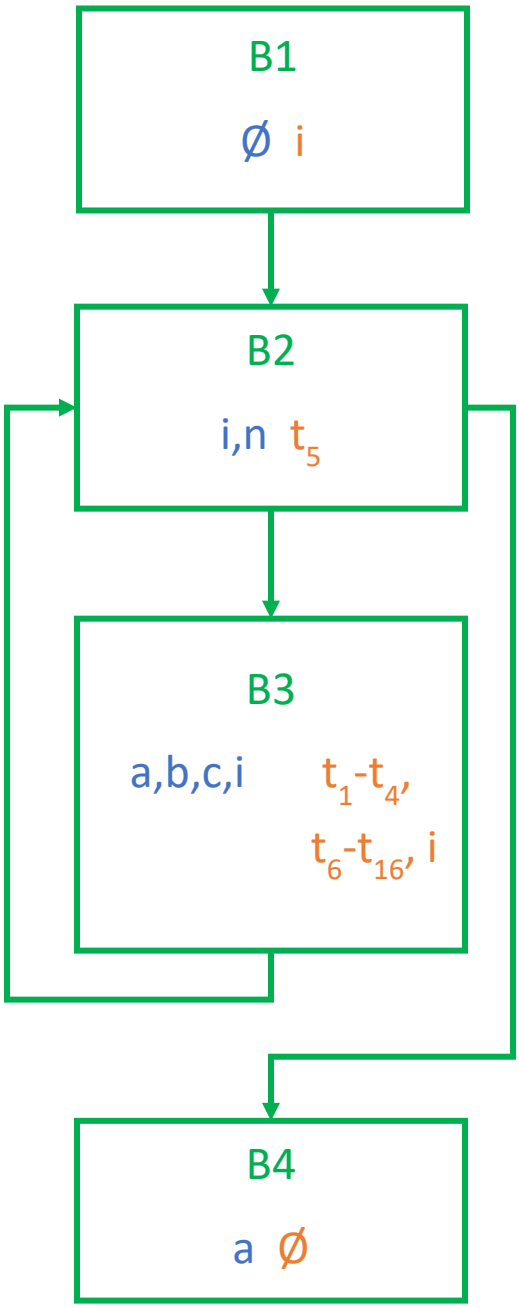
Ø
i,n

Ø
a,b,c,i

Ø
a,b,c,i,n

Ø
Ø

$$\text{LiveOut}(n) = \bigcup_{m \in \text{succ}(n)} (\text{UEVar}(m) \cup (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}))$$



LiveOut

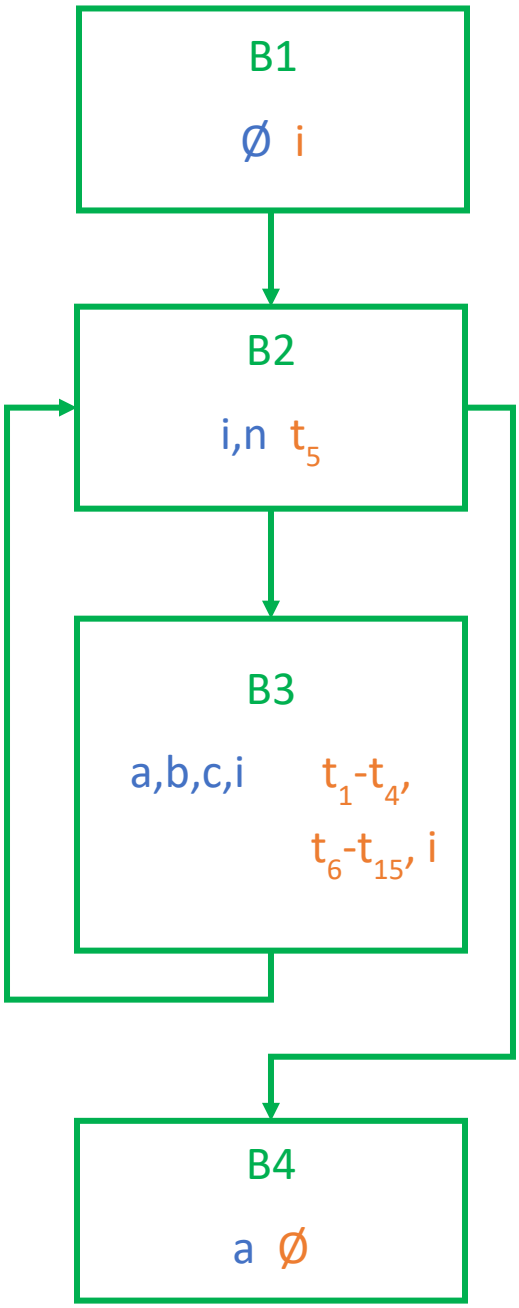
i,n
a,b,c,i,n

a,b,c,i
a,b,c,i,n

a,b,c,i,n
a,b,c,i,n

Ø
Ø

$$\text{LiveOut}(n) = \bigcup_{m \in \text{succ}(n)} (\text{UEVar}(m) \cup (\text{LiveOut}(m) \cap \overline{\text{VarKill}(m)}))$$



LiveOut

a,b,c,i,n
a,b,c,i,n

a,b,c,i,n
a,b,c,i,n

a,b,c,i,n
a,b,c,i,n

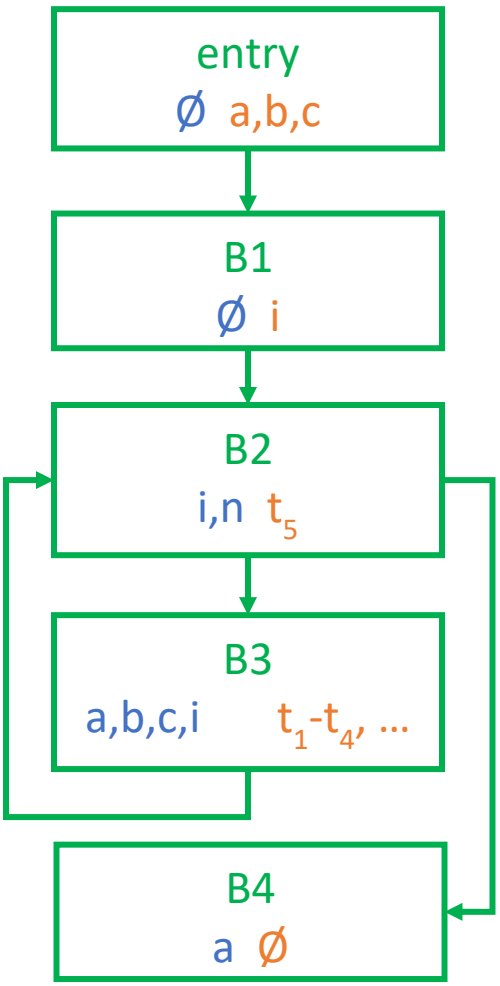
Ø
Ø

```
// CFG has N blocks numbered 0 to N-1

for i = 0 to N-1
      LiveOut(i) = Ø

changed = true
while (changed)
   changed = false
   for i = 0 to N-1
         recompute LiveOut(i) according to formula
         if LiveOut(i) changed
                  changed = true
```

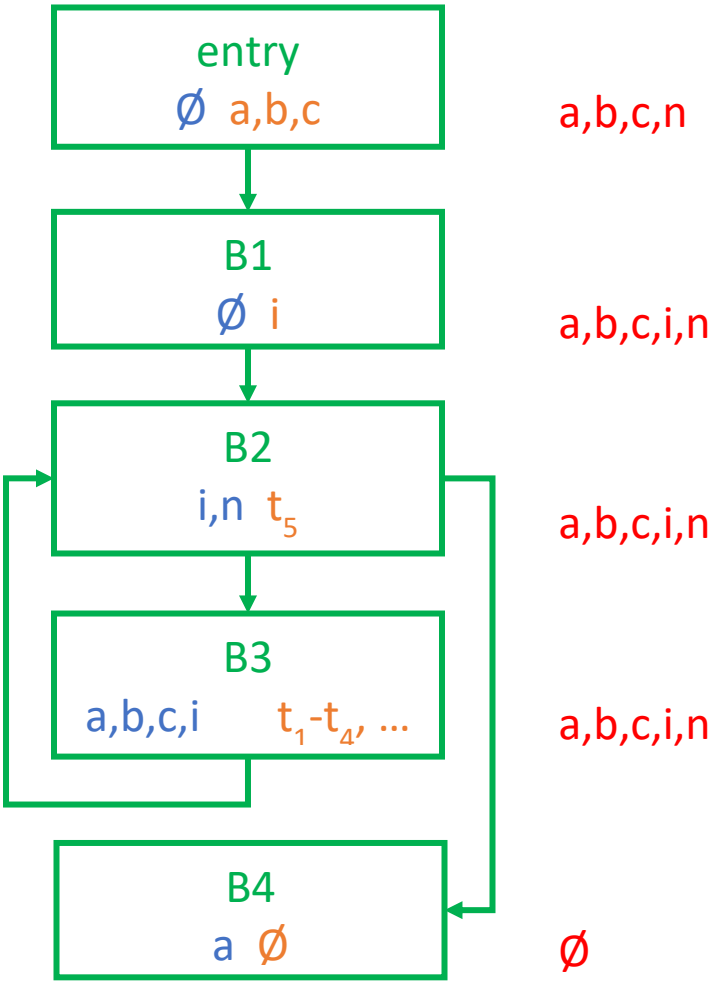
Finding Possibly Uninitialized Variable Uses

Some languages require the compiler to report when a variable is used before it is assigned a value. With LiveOut computation, this is straightforward.

First, we'll add a special "entry" node to the procedure CFG that contains "definitions" of the parameters only.



Then we do LiveOut analysis on the CFG. Any variable that is LiveOut of the entry node but not in its VarKills is potentially uninitialized.



(We can achieve the same result by adding yet another "pre-entry" node with Ø Ø and flagging any variable in its LiveOut.)

These are *potentially uninitialized* rather than simply *uninitialized* because of a few complicating factors.

1. If the variable is accessible via another name and initialized through that name, liveness analysis will not detect it. For instance:

```
…
p = &x;
*p = 0;
…
x = x + 1;
```

2. The variable may exist before the current procedure is invoked, and may have been initialized outside of the scope of the analysis. Static and global variables are examples.
3. Liveness analysis may discover a path from the procedure's entry to a use of a variable along which the variable is not defined. If, however, that path is not feasible at runtime, then the variable may be flagged as uninitialized even though no execution will ever use the uninitialized value. For instance:

```
int s;
int i = 1;
while (i < 10) {
      if (i == 1) {
            s = 0;
      }
      s = s + i++;
}
```

These illustrate a fundamental limit of dataflow analysis: it assumes that all paths through the CFG are feasible at runtime. That assumption can be *overly conservative*.

We'll say that an assumption is *conservative* if it leads us to fewer conclusions (lesser results of analysis) and fewer code modifications than what is theoretically possible **while respecting safety**. Conservative is good in compiler optimization, but one always wants to get as much as possible while still remaining conservative.
Saying something is *overly conservative* generally means that there are ways to do better. In this

case, with a deeper, more expensive analysis, we could show that there is no uninitialized use of s.

Note that we cannot determine exactly which variables are used before they are initialized, as this is equivalent to the halting problem.

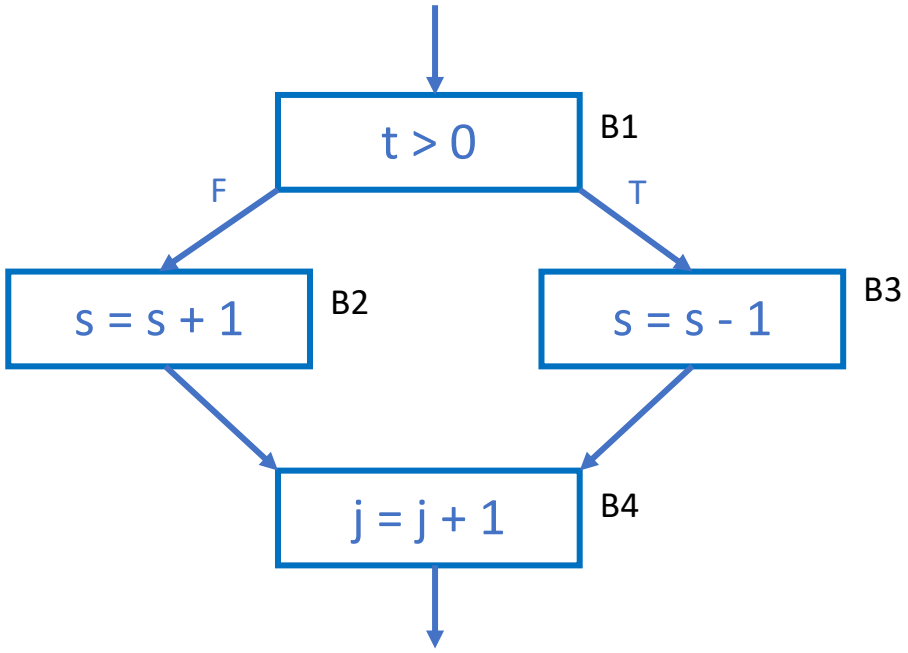
> Suppose we have an instance I of the halting problem, where I is a program with Halt instructions in it.
> We create a program I' by replacing each Halt instruction in I with the instruction
>
> t = s
>
> where s and t are variables that do not occur in I. Now we ask: are there any uninitialized uses of s in I' (when it is executed)? If we can answer this question exactly, then we have answered the halting problem on I.

Most dataflow questions end up being equivalent to halting, so we generally must make do with conservative estimations of exact results. Fortunately, in most cases, the conservative estimates are fairly effective.

But above all else, when using dataflow or any type of analysis, one must pay attention to **ensure safety** of any resulting code modifications.

Global Code Placement

Many processors have asymmetric branch costs; the cost of a fall-though branch is less than the cost of a taken branch:



Let's say that a fall-through has cost 0, an absolute jump has cost c_1, and a branch has cost c_2.

Furthermore, suppose we know something about the frequencies of T and F in the decision of Block B1. Let f(T) be the frequency of T, and f(F) be the frequency of false.

B1; B2; B3; B4

```
        if t > 0 goto pos:        f(T)*c2
neg:    s = s + 1
        goto join                 f(F)*c1
pos:    s = s – 1
join:   j = j + 1
        …
```

B1; B3; B2; B4

```
        if t <= 0 goto neg:       f(F)*c2
pos:    s = s - 1
        goto join                 f(T)*c1
neg:    s = s + 1
join:   j = j + 1
        …
```

B1; B2; B4; B3

```
        if t > 0 goto pos:        f(T)*c2
```

```
neg:    s = s + 1
join:   j = j + 1
        …
pos:    s = s – 1
        goto join                    f(T)*c1
```

```
B1; B3; B4; B2
        if t <= 0 goto neg:          f(F)*c2
pos:    s = s - 1
join:   j = j + 1
        …
neg:    s = s + 1
        goto join                    f(F)*c1
```

That's four different code placements with four different costs:

$f(F)*c_1 + f(T)*c_2$

$f(T)*c_1 + f(F)*c_2$

$f(T)*c_1 + f(T)*c_2$

$f(F)*c_1 + f(F)*c_2$

The compiler can choose whichever of these is estimated to be the smallest cost.

For instance, suppose we have f(F) = 1 and f(T) = 100. Then we should definitely choose the ordering B1; B3; B4; B2, at a cost of $c_1 + c_2$. The worst option from the above is B1; B2; B4; B3, at a cost of $100(c_1 + c_2)$. That's a big difference.

This sort of situation arises a lot in practice:

```
for( i = 1 to n) {
    …
}
```

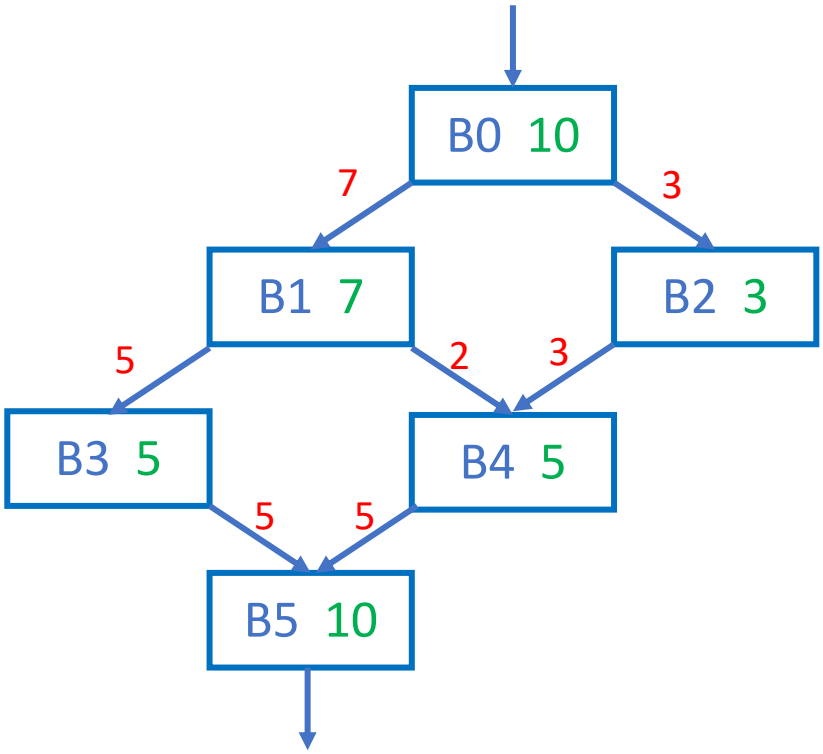
The conditional in this loop will take the branch into the loop body n times and take the branch out to the code afterwards 1 time. If n is known or assumed to be relatively large, the way we place the basic blocks could have a large influence on the time this loop takes.

Compilers can sometimes get frequency estimates along CFG edges by loop analysis. Other times, it is not so clear.

Another approach to getting frequency estimates is to compile the program, run it, and use frequencies that the running program actually generates (this information is called a profile). This then requires a second compilation using the profile to guide code placement. To get profiles from running programs, there are three main approaches.

1. Instrumented executables. In this scheme, the compiler generates code to count specific events, such as procedure entries or taken branches. At runtime, the data is written to an external file and processed offline by another tool.
2. Timer interrupts. A tool interrupts the program execution at frequent, regular intervals. The tool constructs a histogram of Program Counter values where the interrupts occurred. Post-processing constructs the profile information.
3. Performance counters. Many processors have counters to record hardware events, such as total cycles, cache misses, or taken branches. If these are available, they can be used to construct highly-accurate profiles.

CFG edge data is considered to be higher-quality than CFG vertex data. Consider the following CFG, where edge frequencies are shown in red and vertex frequencies are shown in green.



The most-often-executed path in this CFG is B0-B1-B3-B5. Using edge data, a compiler would place B3 as the fall-through option for B1. However, using vertex data, when trying to decide the fall-through

option for B1, both B3 and B4 look equally good, and it could end up choosing B4 as the fall-through.

Collecting and using edge frequencies is superior.

Hot Paths

To determine how to lay out code, a compiler can construct a set of hot paths—CFG paths that contain the most-frequently executed edges.

Hot paths can be constructed in a greedy fashion by considering the edges in order of decreasing frequency:

```
for each block b
    make a degenerate chain d for b
    d.priority = ∞
```

```
P = 0
for each CFG edge (x,y)  x ≠ y in decreasing freq. order
   if x is the tail of a chain a and
      y is the head of a chain b
         t = a.priority
         u = b.priority
         c = a followed by (x,y) followed by b
         c.priority = min(t, u, P++)
```

Edge	Set of Chains	P
–	$(B0)_\infty$ $(B1)_\infty$ $(B2)_\infty$ $(B3)_\infty$ $(B4)_\infty$ $(B5)_\infty$	0
B0, B1	$(B0, B1)_0$ $(B2)_\infty$ $(B3)_\infty$ $(B4)_\infty$ $(B5)_\infty$	1
B3, B5	$(B0, B1)_0$ $(B2)_\infty$ $(B3, B5)_1$ $(B4)_\infty$	2
B4, B5	–	2
B1, B3	$(B0, B1, B3, B5)_0$ $(B2)_\infty$ $(B4)_\infty$	3
B0, B2	–	3
B2, B4	$(B0, B1, B3, B5)_0$ $(B2, B4)_3$	4
B1, B4	–	4

Once these hot paths are constructed, code layout can be done with a straightforward worklist algorithm.

```
t = chain headed by CFG entry node
```

```
Worklist = {t}
while !Worklist.isEmpty()
    remove a chain c of lowest priority from WorkList
    for each block x in c in chain order
        place x at the end of the executable code
    for each block x in c
        for each edge (x, y) where y is unplaced
            t = chain containing y
            Worklist = Worklist ∪ {t}
```

The results of applying this algorithm to our example are:

Step	WorkList	Code Layout
–	$(B0\ B1\ B3\ B5)_0$	
1	$(B2\ B4)_3$	B0 B1 B3 B5
2	Ø	B0 B1 B3 B5 B2 B4