Control flow graphs

Moving beyond basic blocks

- Up until now, we have focused on single basic blocks
- What do we do if we want to consider larger units of computation
  - Whole procedures?
  - Whole program?
- Idea: capture *control flow* of a program
  - How control transfers between basic blocks due to:
    - Conditionals
    - Loops

Representation

- Use standard three-address code
- Jump targets are labeled
- Also label beginning/end of functions
- Want to keep track of *targets of jump statements*
  - Any statement whose execution may immediately follow execution of jump statement
  - *Explicit* targets: targets mentioned in jump statement
  - *Implicit* targets: statements that follow conditional jump statements
    - The statement that gets executed if the branch is not taken

Running example

```
A = 4
t1 = A * B
repeat {
  t2 = t1/C
  if (t2 ≥ W) {
    M = t1 * k
    t3 = M + I
  }
  H = I
  M = t3 - H
} until (T3 ≥ 0)
```

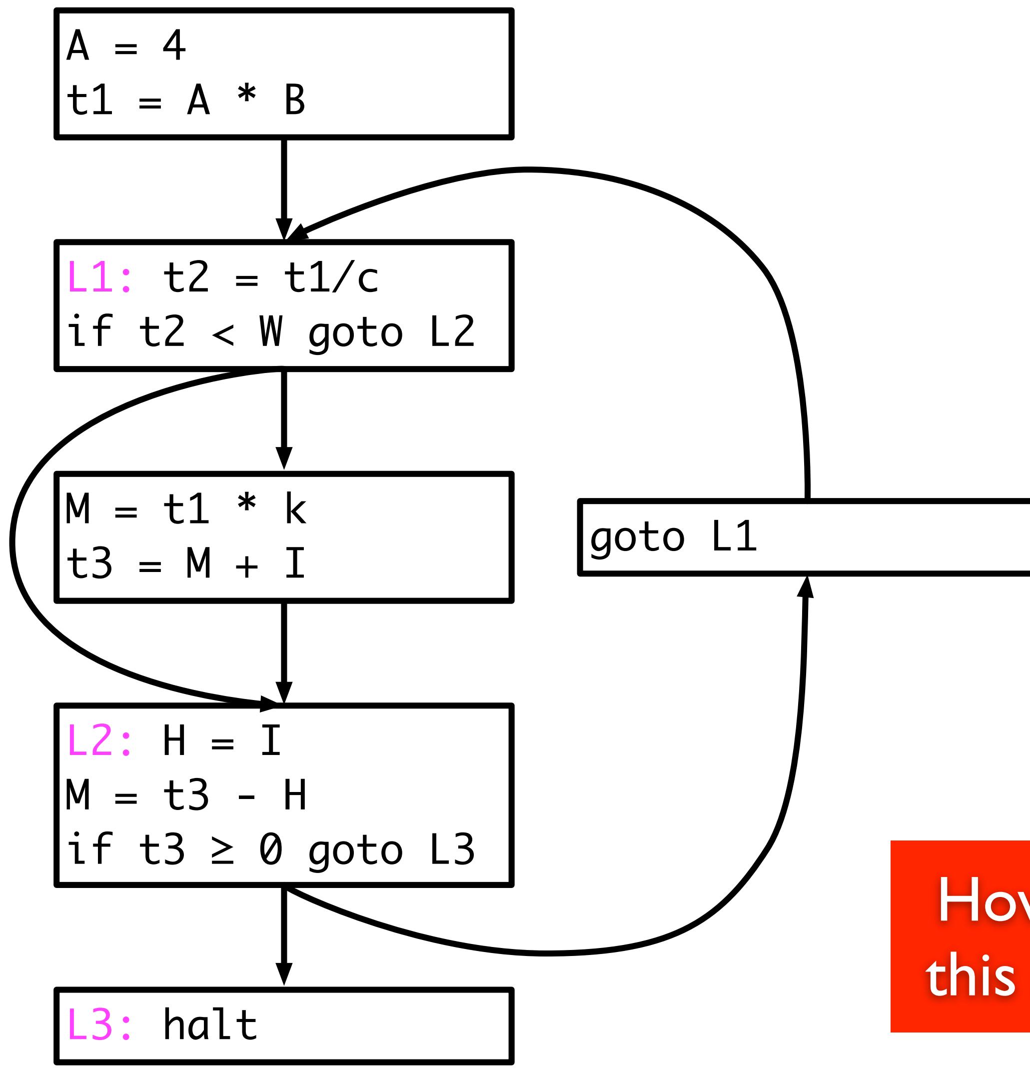
Running example

```
     A = 4
     t1 = A * B
L1:  t2 = t1 / C
     if t2 < W goto L2
     M = t1 * k
     t3 = M + I
L2:  H = I
     M = t3 - H
     if t3 ≥ 0 goto L3
     goto L1
L3:  halt
```

Control flow graphs

- Divides statements into *basic blocks*
- Basic block: a maximal sequence of statements $I_0, I_1, I_2, \ldots, I_n$ such that if I_j and I_{j+1} are two adjacent statements in this sequence, then
  - The execution of I_j is always immediately followed by the execution of I_{j+1}
  - The execution of I_{j+1} is always immediate preceded by the execution of I_j
- Edges between basic blocks represent potential flow of control

CFG for running example



How do we build this automatically?

Constructing a CFG

- To construct a CFG where each node is a basic block
  - Identify *leaders*: first statement of a basic block
  - In program order, construct a block by appending subsequent statements up to, but not including, the next leader
- Identifying leaders
  - First statement in the program
  - Explicit target of any conditional or unconditional branch
  - Implicit target of any branch

Partitioning algorithm

- Input: set of statements, *stat(i)* = ith statement in input
- Output: set of *leaders*, set of basic blocks where *block(x)* is the set of statements in the block with leader *x*
- Algorithm

```
leaders = {1}              //Leaders always includes first statement
for i = 1 to |n|           //|n| = number of statements
    if stat(i) is a branch, then
        leaders = leaders ∪ all potential targets
end for
worklist = leaders
while worklist not empty do
    x = remove earliest statement in worklist
    block(x) = {x}
    for (i = x + 1; i ≤ |n| and i ∉ leaders; i++)
        block(x) = block(x) ∪ {i}
    end for
end while
```

Running example

```
    A = 4
    t1 = A * B
L1: t2 = t1 / C
    if t2 < W goto L2
    M = t1 * k
    t3 = M + I
L2: H = I
    M = t3 - H
    if t3 ≥ 0 goto L3
    goto L1
L3: halt
```

Leaders =

Basic blocks =

Running example

```
1        A = 4
2        t1 = A * B
------------------------------
3   L1:  t2 = t1 / C
4        if t2 < W goto L2
------------------------------
5        M = t1 * k
6        t3 = M + I
------------------------------
7   L2:  H = I
8        M = t3 - H
9        if t3 ≥ 0 goto L3
------------------------------
10       goto L1
------------------------------
11  L3:  halt
```

Leaders = {1, 3, 5, 7, 10, 11}

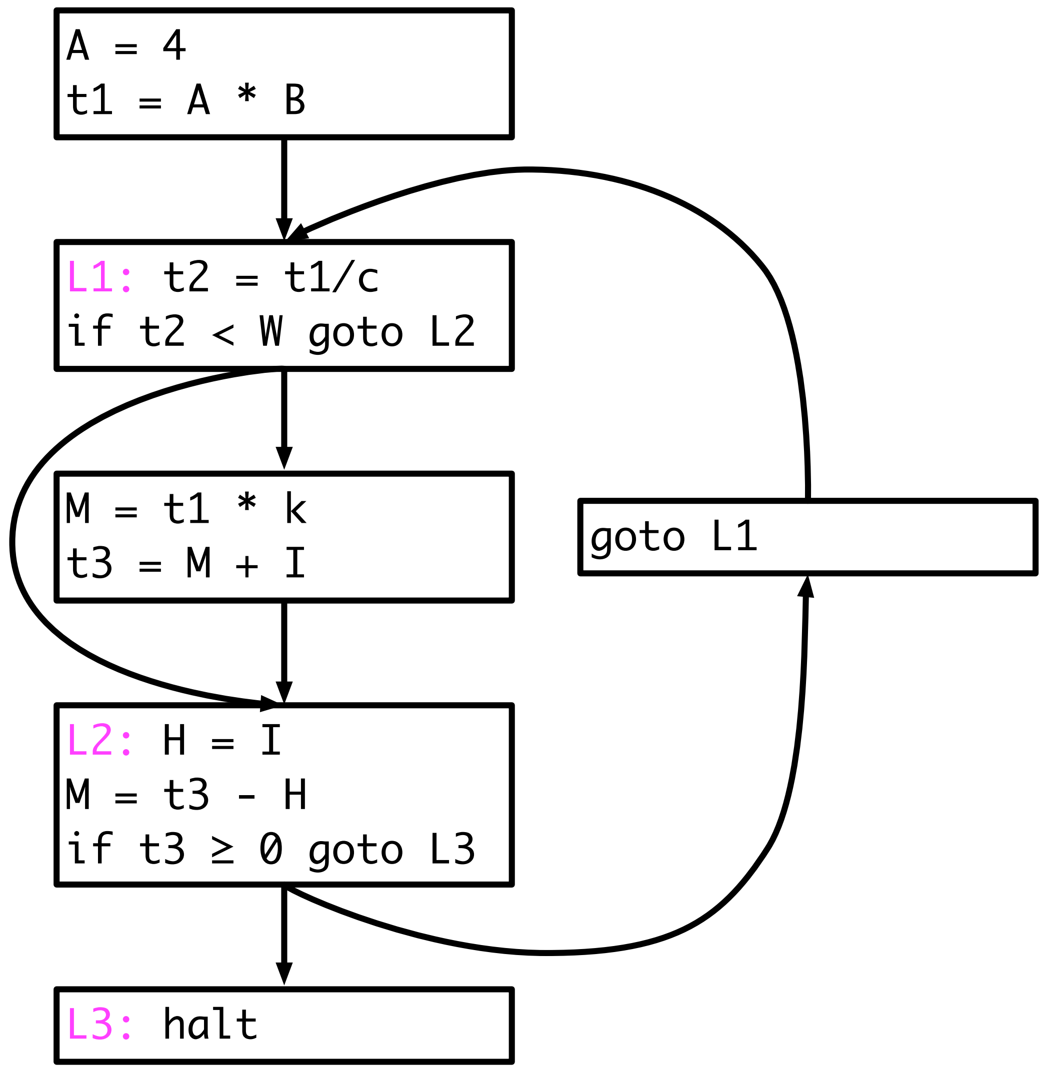
Basic blocks = { {1, 2}, {3, 4}, {5, 6}, {7, 8, 9}, {10}, {11} }

Putting edges in CFG

- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

```
for i = 1 to |block|
    x = last statement of block(i)
    if stat(x) is a branch, then
        for each explicit target y of stat(x)
            create edge from block i to block y
        end for
    if stat(x) is not unconditional then
        create edge from block i to block i+1
end for
```

Result


A = 4
t1 = A * B
L1: t2 = t1/c
if t2 < W goto L2
M = t1 * k
t3 = M + I
goto L1
L2: H = I
M = t3 - H
if t3 ≥ 0 goto L3
L3: halt

Discussion

- Some times we will also consider the *statement-level* CFG, where each node is a statement rather than a basic block
  - Either kind of graph is referred to as a CFG
- In statement-level CFG, we often use a node to explicitly represent *merging* of control
  - Control merges when two different CFG nodes point to the same node
- Note: if input language is *structured*, front-end can generate basic block directly
  - "GOTO considered harmful"

Statement level CFG