CS406: Compilers

Spring 2021

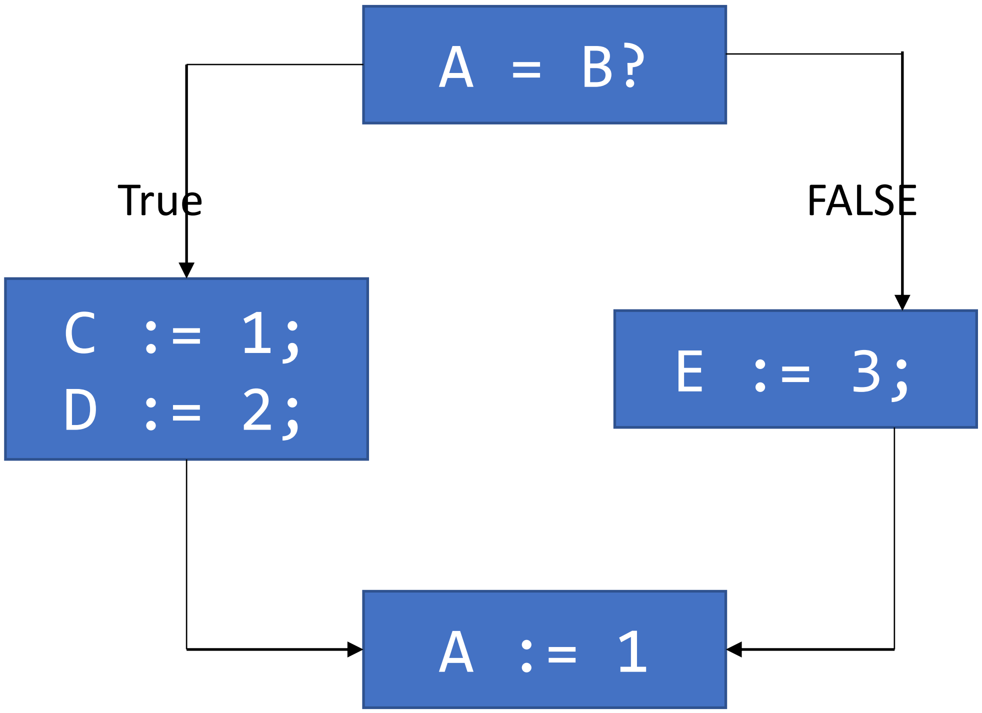
Week 12: Control Flow Graphs, Data Flow Analysis

Basic Blocks and Flow Graphs

- Basic Block
  - Maximal sequence of consecutive instructions with the following properties:
    - The first instruction of the basic block is the *only entry point*
    - The last instruction of the basic block is either the halt instruction or the *only exit point*
- Flow Graph
  - Nodes are the basic blocks
  - Directed edge indicates which block follows which block

Basic Blocks and Flow Graphs - Example

```
if A = B then
   C := 1;
   D := 2;
else
   E := 3
fi
A := 1;
```



A data flow graph

Flow Graphs

- Capture how control transfers between basic blocks due to:
  - Conditional constructs
  - Loops
- Are necessary when we want optimize considering larger parts of the program
  - Multiple procedures
  - Whole program

Flow Graphs - Representation

- We need to label and track statements that are jump targets
  - **Explicit targets** – targets mentioned in jump statement
  - **Implicit targets** – targets that follow conditional jump statement
    - Statement that is executed if the branch is not taken
- Implementation
  - Linked lists for BBs
  - Graph data structures for flow graphs

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)
```

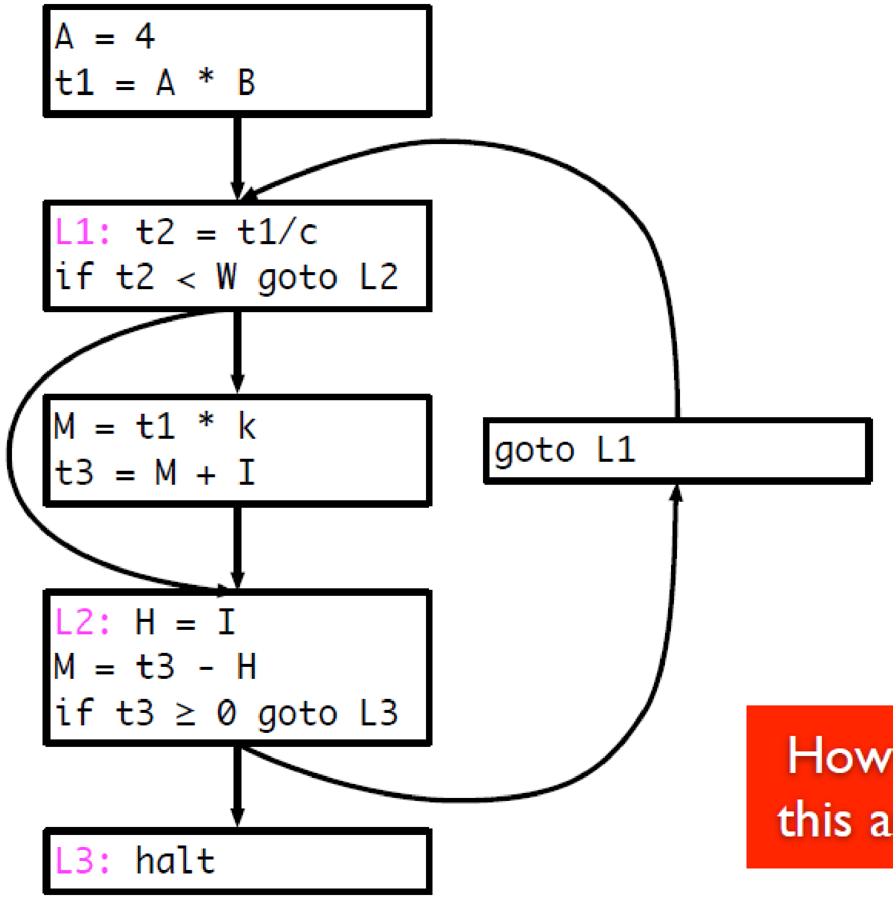
Slide Courtesy: Milind Kulkarni

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
```

Slide Courtesy: Milind Kulkarni

CFG for running example



How do we build this automatically?

Slide Courtesy: Milind Kulkarni

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
```

Slide Courtesy: Milind Kulkarni

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}
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}
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}
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}
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}
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}
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}
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}

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}
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}

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 =

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 =

Block(1) = ?

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 =

Block(1) = ?

Start from statement 2 and add till either the end or a leader is reached

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}   Block(1) = {1,2}

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}   Block(3) = ?
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}   Block(3) = {3,4}

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}   Block(5) = ?

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}   Block(5) = {5,6}

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} Block(7) = ?

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}   Block(7) = {7,8,9}

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}   Block(10) = ?
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}   Block(10) = {10}

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}   Block(11) = {11}

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

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

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

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
```

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

Edge from block 1 to block 2

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

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

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

Edge from block 2 to block 4

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

```
{{1,2},{3,4},{5,6},{7,8,9},{10},{11}}
```

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

Edge from block 2 to block 3

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

```
{{1,2},{3,4},{5,6},{7,8,9},{10},{11}}
```

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

Edge from block 3 to block 4

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

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

Edge from block 4 to block 6

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

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

Edge from block 4 to block 5

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

Putting edges in CFG

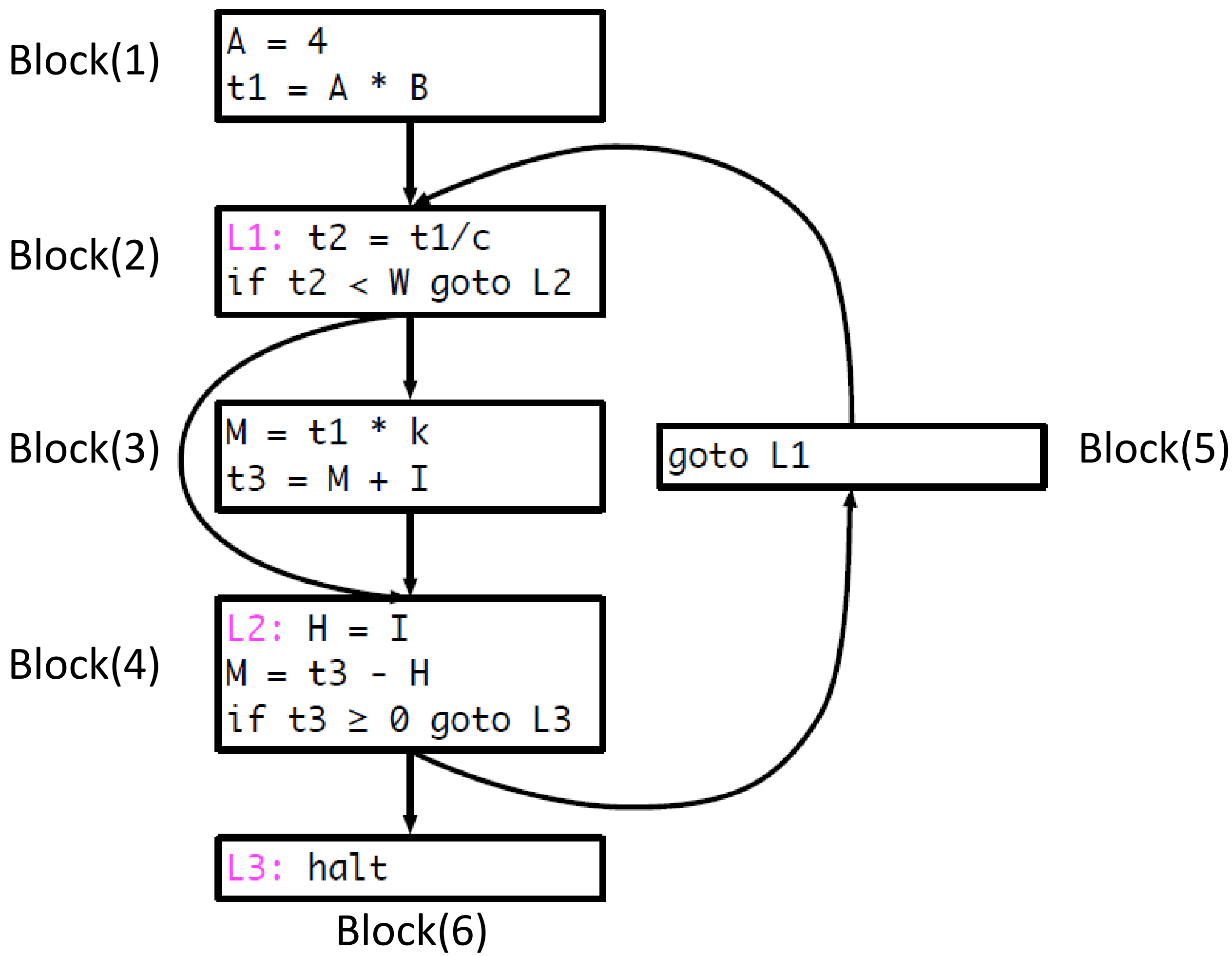
- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

```
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
```

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

Edge from block 5 to block 2

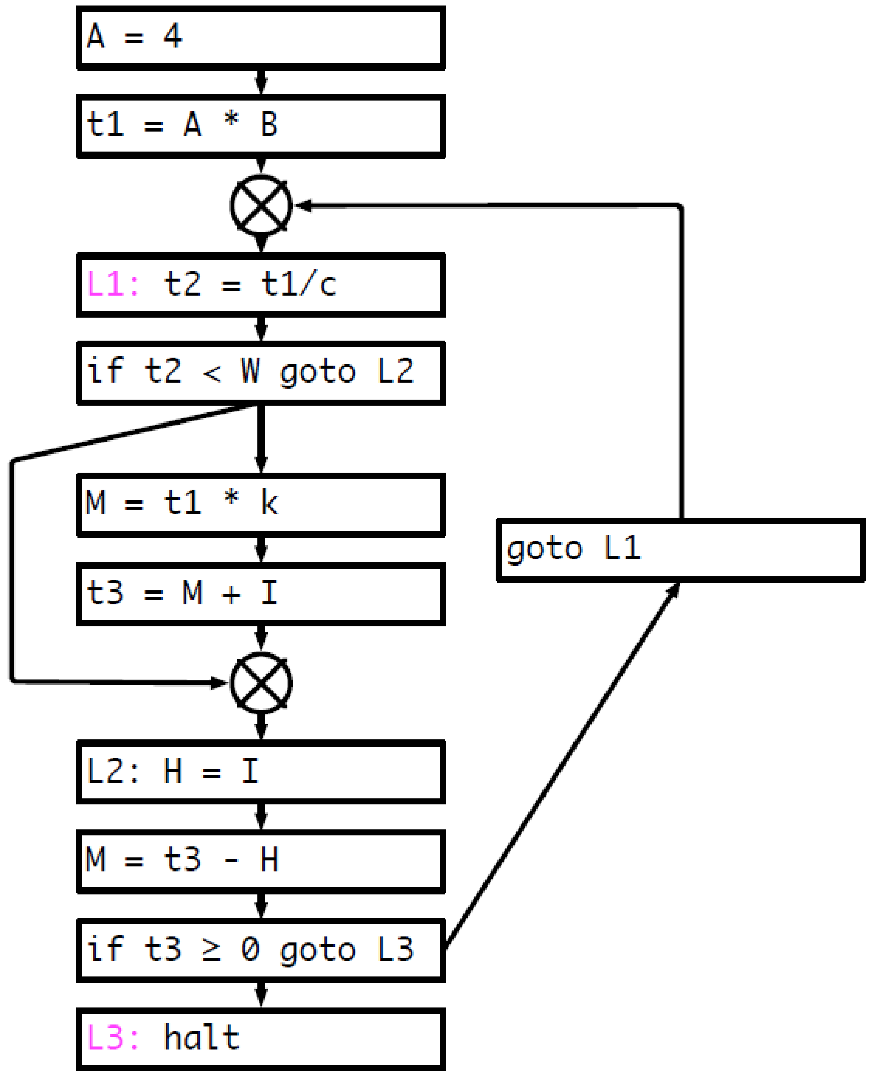
Result


Block(1)
A = 4
t1 = A * B
Block(2)
L1: t2 = t1/c
if t2 < W goto L2
Block(3)
M = t1 * k
t3 = M + I
Block(4)
L2: H = I
M = t3 - H
if t3 ≥ 0 goto L3
Block(5)
goto L1
L3: halt
Block(6)

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



Slide Courtesy: Milind Kulkarni

Control Flow Graphs - Use

- Why do we need CFGs? - Global Optimization
  - Optimizing compilers do global optimization (i.e. optimize beyond basic blocks)
    - Differentiating aspect of normal and optimizing compilers
  - E.g. loops are the most frequent targets of global optimization (because they are often the “hot-spots” during program execution)

how do we identify loops in CFGs?

Identify Loops in CFGs

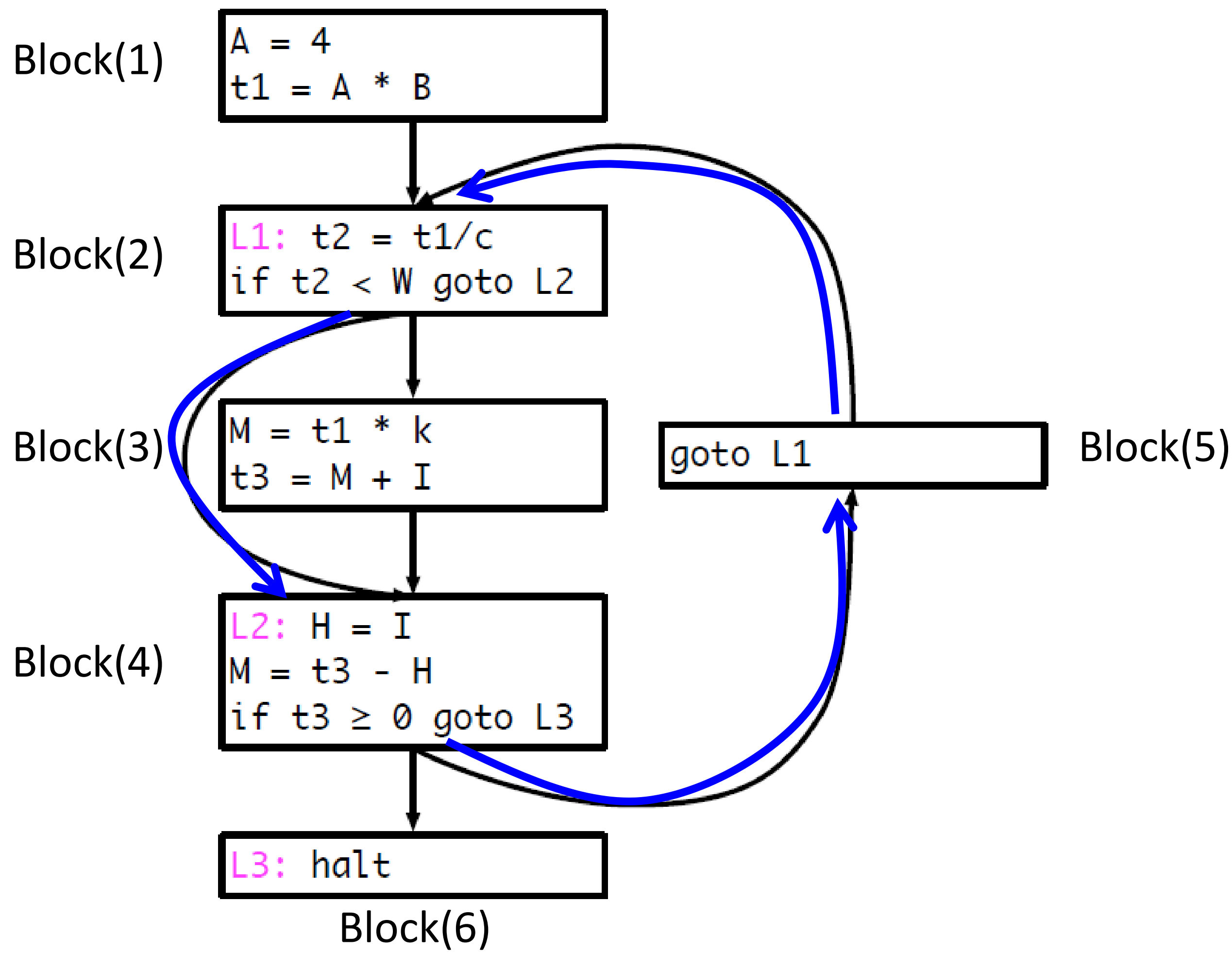
- Loops – **how do we identify loops in CFGs?**

  For a set of nodes, L, that belong to loop:

  1) There is a *loop entry node* such that any path from the *graph entry node* to any node in L goes through the *loop entry node.* i.e. no node in L has a predecessor that is outside L.

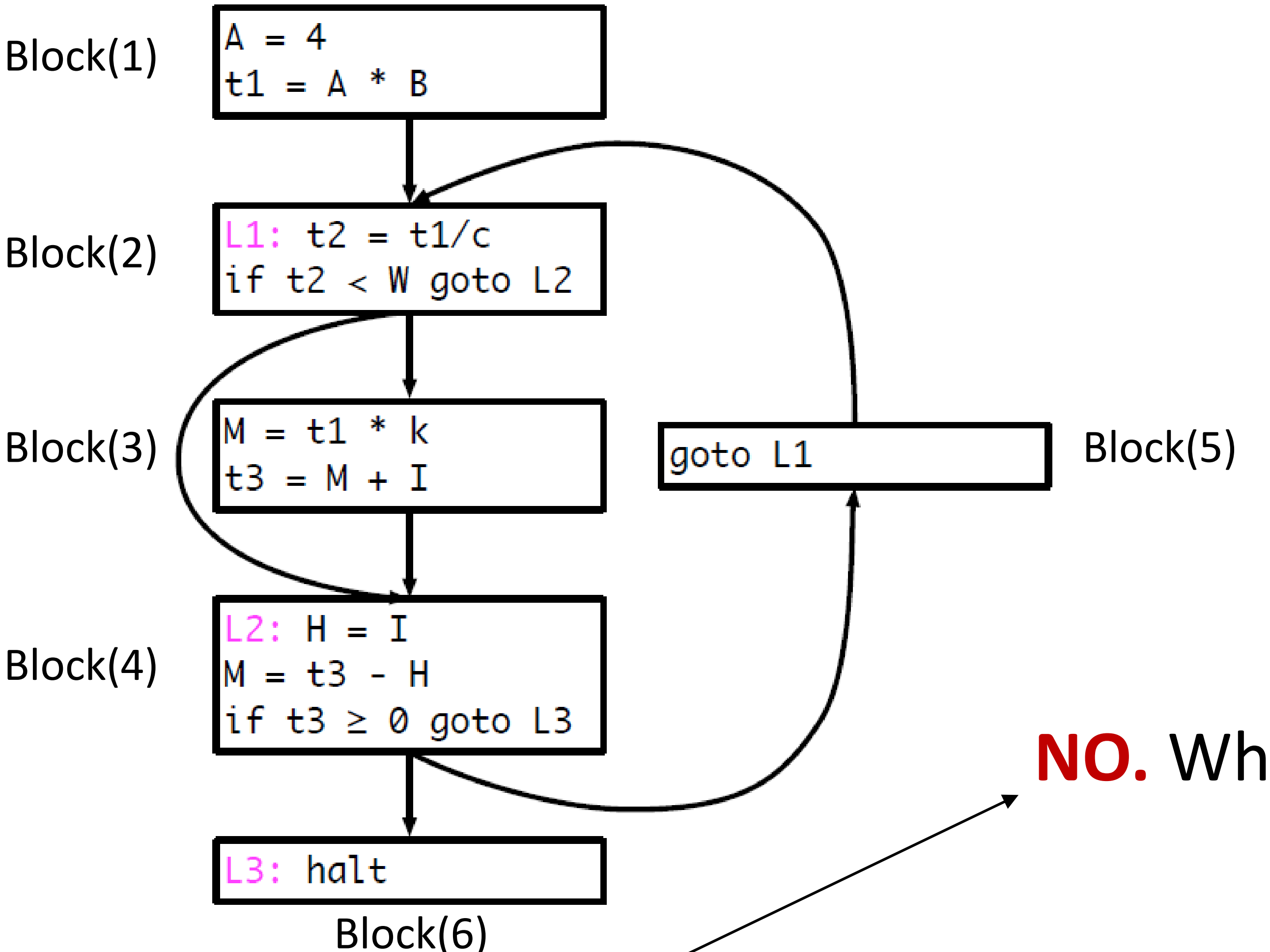
  2) *Every node in L* has a non-empty path, completely within L, to the entry of L.

Identify Loops in CFGs



Consider: {B2, B4, B5}. Is this a loop?, Are there other loops?

Identify Loops in CFGs

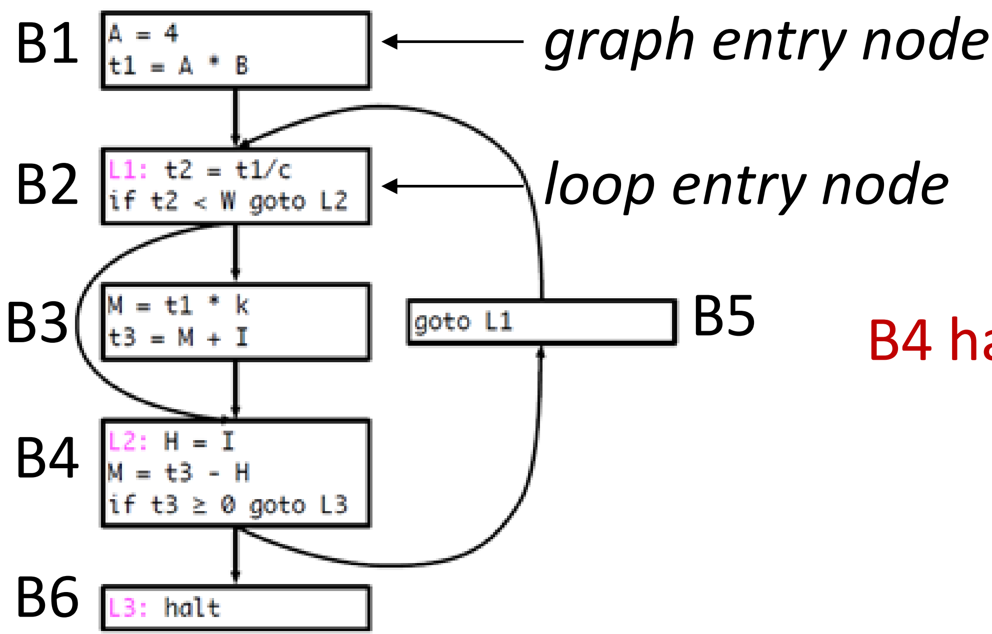


NO. Why?

Consider: {B2, B4, B5}. Is this a loop?, Are there other loops?

Identify Loops in CFGs

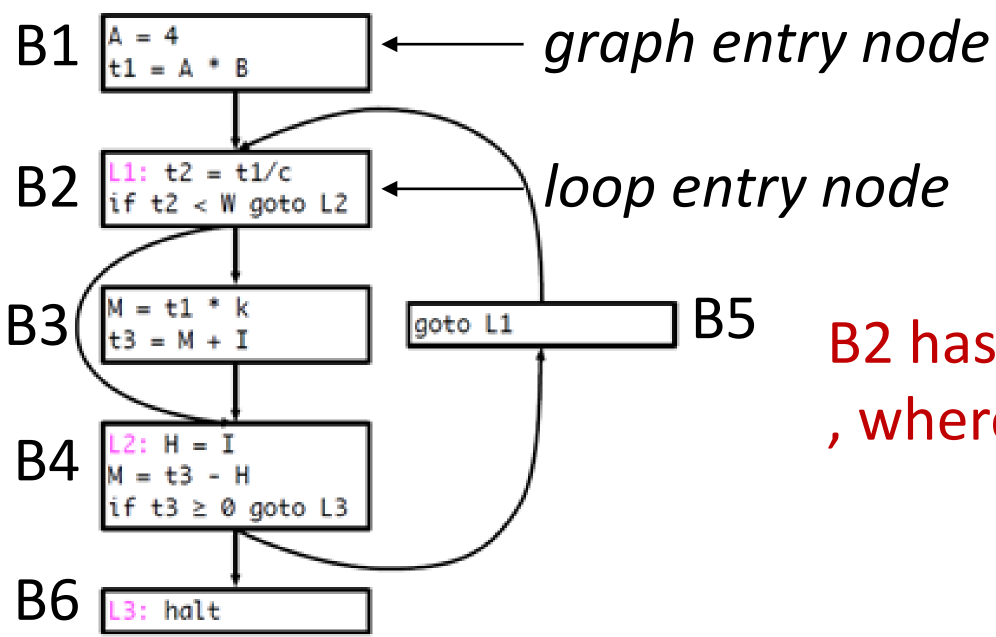
1) Is L={B2, B4, B5} a loop?. No. Consider:
   - There is a *loop entry node* such that any path from the *graph entry node* to any node in L goes through the *loop entry node*. i.e. no node in L has a predecessor that is outside L.



B4 has a predecessor B3 not in L

Identify Loops in CFGs

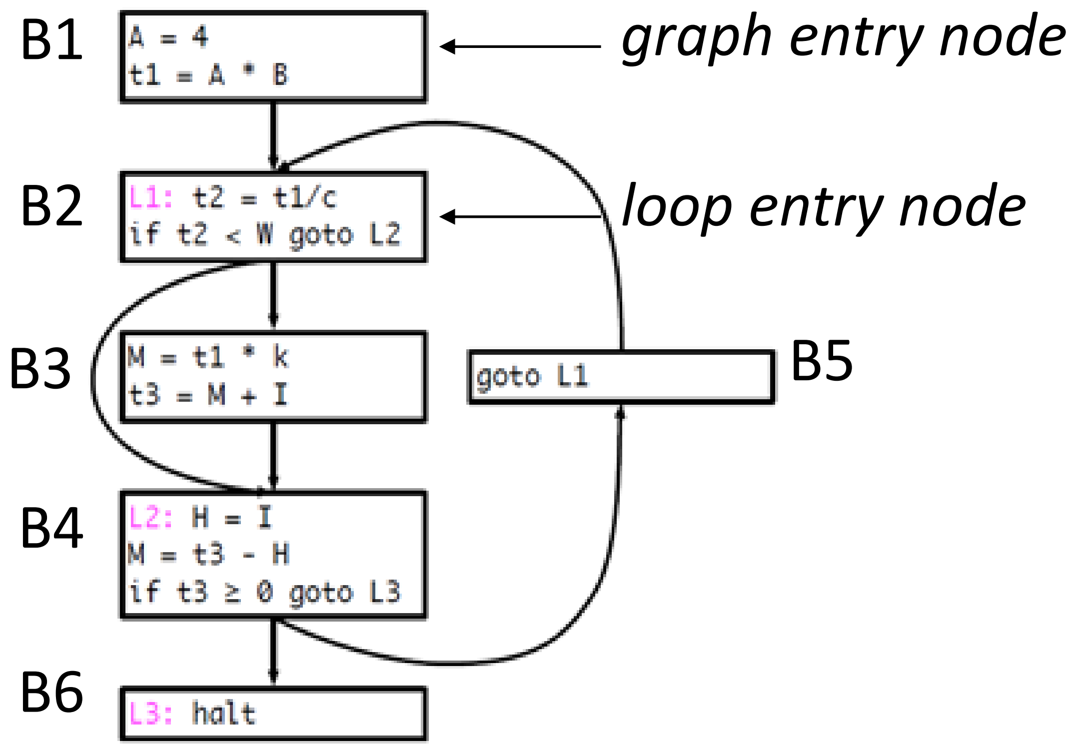
1) Is L={B2, B4, B5} a loop?. No. Consider:
   - *Every node in L* has a non-empty path, completely within L, to the entry of L.



B2 has a path B2->B3->B4->B5->B2 , where B3 is not in L

Identify Loops in CFGs

1) Is L={B2, B3, B4, B5} a loop?.



Optimize Loops

- Example - Code Motion

Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- Optimization: can move 10/I out of loop.

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- Optimization: can move 10/I out of loop
- What if I = 0?

Optimize Loops – Code Motion

- Should be careful while doing optimization of loops

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- Optimization: can move 10/I out of loop
- What if I = 0?
- What if I != 0 but loop executes zero times?

Optimization Criteria - Safety and Profitability

- Safety - is the code produced after optimization producing same result?
- Profitability - is the code produced after optimization running faster or uses less memory or triggers lesser number of page faults etc.

```
while J > I loop
   A(j) := 10/I;
   j := j + 2;
end loop;
```

- E.g. moving I out of the loop introduces exception (when I=0)
- E.g. if the loop is executed zero times, moving I out is not profitable

Optimize Loops -Identifying Invariant Expressions

- How do we identify expressions that can be moved out of the loop?
  - `LoopDef = {}` set of variables <u>defined</u> i.e. whose values are overwritten) in the loop body
  - `LoopUse = { }` ‘relevant’ variables <u>used</u> in computing an expression

```
Mark_Invariants(Loop L) {
   1. Compute LoopDef for L
   2. Mark as invariant all expressions,
      whose relevant variables don't belong
      to LoopDef
}
```

Optimize Loops -Identifying Invariant Expressions

- Example

LoopDef{}

```
for I = 1 to 100                    {A, J, K}
   for J = 1 to 100                 {A, J, K}
      for K = 1 to 100              {A, K}
         A[I][J][K] = (I*J)*K
```

Optimize Loops -Identifying Invariant Expressions

- Example

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100
         A[I][J][K] = (I*J)*K
```

Invariant Expressions

{ I*J, Addr(A[i][j]) }

For an array access, `A[m] => Addr(A) + m`

For 3D array above*, `Addr(A[I][J][K])` =

Addr(A)+(I*10000)-10000+(J*100)-100+K-1

*Assuming row-major ordering of storage

Optimize Loops -Identifying Invariant Expressions

- Example

Invariant Expressions

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100
         A[I][J][K] = (I*J)*K
```

→ `{ Addr(A[i]) }`

For an array access, `A[m] => Addr(A) + m`

For 3D array above*, `Addr(A[I][J][K]) =`

`Addr(A)+(I*10000)-10000+(J*100)-100+K-1`

*Assuming row-major ordering of storage

Optimize Loops -Factoring Invariant Expressions

- Move the invariant expressions identified

```
Factor_Invariants(Loop L) {
   Mark_Invariants(L);
   foreach expression E marked an invariant:
      1. Create a temporary T
      2. Replace each occurrence of E in L with T
      3. Insert T:=E in L's header code
         immediately after the first loop-
         termination test (i.e. after "j<!op> OUT" in slide 39,
         week9.pdf)
         // If loop is known to execute at least once,
         insert T:=E before LOOP:
}
```

Optimize Loops -Factoring Invariant Expressions

- Example

```
for I = 1 to 100
   for J = 1 to 100
      for K = 1 to 100
         A[I][J][K] = (I*J)*K
```

```
for I=1 to 100
   temp3=Addr(A[i])
   for J=1 to 100
      temp1=Addr(temp3(J))
      temp2=I*J
      for K=1 to 100
         temp1[K]=temp2*K
```

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case I: We can move `t = a op b` if the statement <u>dominates</u> all loop exits where `t` is live

A node a dominates node b if all paths to b must go through a

```
for (...) {
   if(*)
      a = 100
}
c=a
```

Cannot move `a=100` because it does not dominate `c=a` i.e. there is one path (when if condition is false) `c=a` can be reached without going `a=100`

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case II: We can move `t = a op b` if there is only definition of `t` in the loop

```
for (...) {
   if(*)
      a = 100
   else
      a = 200
}
```

Multiple definition of a

Optimize Loops -Factoring Invariant Expressions

- Expressions cannot always be moved out!

Case III: We can move `t = a op b` if `t` is not defined before the loop, where the definition reaches t's use after the loop

```
a=5
for (...) {
   a = 4+b
}
c=a
```

Definition of a in a=5 reaches `c=a`, which is defined after the loop

Optimize Loops –Strength Reduction

- Like strength reduction in peephole optimization
  - E.g. replace `a*2` with `a<<1`
- Applies to uses of induction variable in loops
  - Basic induction variable (`I`) – only definition within the loop is of the form `I = I ± S`, (S is loop invariant)

    `I` *usually determines number of iterations*
  - Mutual induction variable (`J`) – defined within the loop, its value is linear function of other induction variable, `I`, such that

    `J = I * C ± D` (C, D are loop invariants)

Optimize Loops –Strength Reduction

```
strength_reduce(Loop L) {
   Mark_Invariants(L);
   foreach expression E of the form I*C+D where I is
L's loop index and C and D are loop invariants
      1. Create a temporary T
      2. Replace each occurrence of E in L with T
      3. Insert T:=Io*C+D, where Io is the initial value of the
         induction variable, immediately before L
      4. Insert T:=T+S*C, where S is the step size, at the end of
         L's body
      }
```

Optimize Loops –Strength Reduction

- Suppose induction variable `I` takes on values I_o, I_o+S, I_o+2S, I_o+3S... in iterations 1, 2, 3, 4, and so on…
- Then, in consecutive iterations, Expression `I*C+D` takes on values

$$I_o*C+D$$
$$(I_o+S)*C+D = I_o*C+S*C+D$$
$$(I_o+2S)*C+D = I_o*C+2S*C+D$$
$$\ldots \qquad\qquad \ldots$$

- The expression changes by a constant `S*C`
- Therefore, we have replaced a * and + with a +

Optimize Loops – Strength Reduction

- Example (Applying to innermost loop)

```
for I = 1 to 100
    for J = 1 to 100
        for K = 1 to 100
            A[I][J][K] = (I*J)*K
```

```
for I=1 to 100
    temp3=Addr(A[i])
    for J=1 to 100
        temp1=Addr(temp3(J))
        temp2=I*J
        for K=1 to 100
            temp1[K]=temp2*K
```

```
. . .
temp2=I*J
temp4=temp2
for K=1 to 100
    temp1[K]=temp4
    temp4=temp4+temp2
```

```
//S=1
//C=temp2
```

Optimize Loops – Strength Reduction

- Exercise (Apply to intermediate loop)

```
for I=1 to 100
   temp3=Addr(A[i])
   for J=1 to 100
      temp1=Addr(temp3(J))
      temp2=I*J
      for K=1 to 100
         temp1[K]=temp2*K
```

```
. . .
temp2=I*J
temp4=temp2
for K=1 to 100
   temp1[K]=temp4
   temp4=temp4+temp2
```

```
// Induction var = J
// S = 1
// Expression = I * J
```

Optimize Loops – Strength Reduction

- Exercise (Apply to intermediate loop)

… …

```
. . .
temp5=I
for J=1 to 100
      temp1=Addr(temp3(J))
      temp2=temp5
      temp4=temp2
      for K=1 to 100
         temp1[K]=temp4
         temp4=temp4+temp2
      temp5=temp5+I
```

Optimize Loops – Strength Reduction

- Further strength reduction possible?

```
for I=1 to 100
   temp3=Addr(A[i])
   temp5=I
   for J=1 to 100
      temp1=Addr(temp3(J))
      temp2=temp5
      temp4=temp2
      for K=1 to 100
         temp1[K]=temp4
         temp4=temp4+temp2
      temp5=temp5+I
```

Optimize Loops – Loop Unrolling

- Modifying induction variable in each iteration can be expensive
- Can instead *unroll* loops and perform multiple iterations for each increment of the induction variable
- What are the advantages and disadvantages?

```
for (i = 0; i < N; i++)
  A[i] = ...
```

Unroll by factor of 4

```
for (i = 0; i < N; i += 4)
  A[i] = ...
  A[i+1] = ...
  A[i+2] = ...
  A[i+3] = ...
```


Slide Courtesy: Milind Kulkarni

Optimize Loops - Summary

- Low level optimization
  - Moving code around in a single loop
  - Examples: loop invariant code motion, strength reduction, loop unrolling
- High level optimization
  - Restructuring loops, often affects multiple loops
  - Examples: loop fusion, loop interchange, loop tiling

Slide Courtesy: Milind Kulkarni

Useful optimizations

- Common subexpression elimination (global)
    - Need to know which expressions are available at a point
- Dead code elimination
    - Need to know if the effects of a piece of code are never needed, or if code cannot be reached
- Constant folding
    - Need to know if variable has a constant value
- So how do we get this information?

Dataflow analysis

- Framework for doing compiler analyses to drive optimization
- Works across basic blocks
- Examples
  - Constant propagation: determine which variables are constant
  - Liveness analysis: determine which variables are live
  - Available expressions: determine which expressions are have valid computed values
  - Reaching definitions: determine which definitions could "reach" a use

Liveness – Recap..

X **defined** here

```
1: X = 10
……. .
N: Y = X + 5
```

X **used** here

X is <u>live</u> at 1

<u>..used in future</u>

- A variable X is live at statement S if:
  - There is a statement S’ that uses X
  - There is a path from S to S’
  - There are no intervening definitions of X

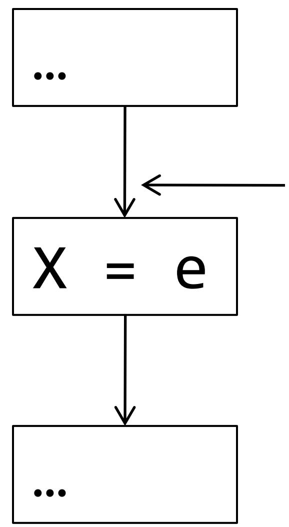
Liveness – Recap..

```
1: X = 10
2: X = Y + 2
   ….
N: Y = X + 5
```

X is <u>dead</u> at 1

- A variable X is dead at statement S if it is not live at S
  - What about …; `X = X + 1`?

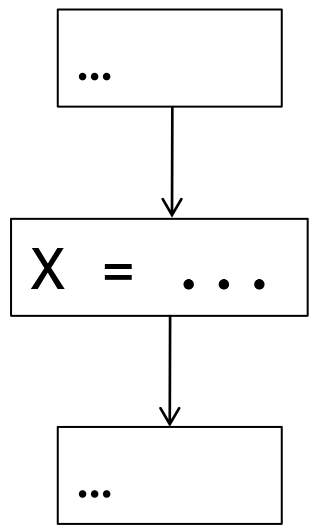
Liveness in a CFG



Given that `e` does not use `X`, `X` is *definitely dead* here (i.e. before the statement).

- Define a set `LiveIn(b)`, where b is a basic block, as: the set of all variables live at the entrance of a basic block

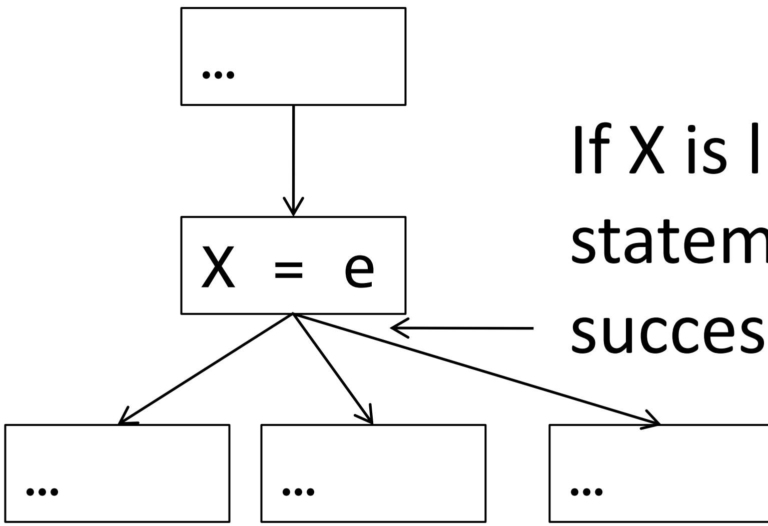
Liveness in a CFG



X is defined here

- Define a set `Def(b)`, where b is a basic block, as: the set of all variables that are defined within block b

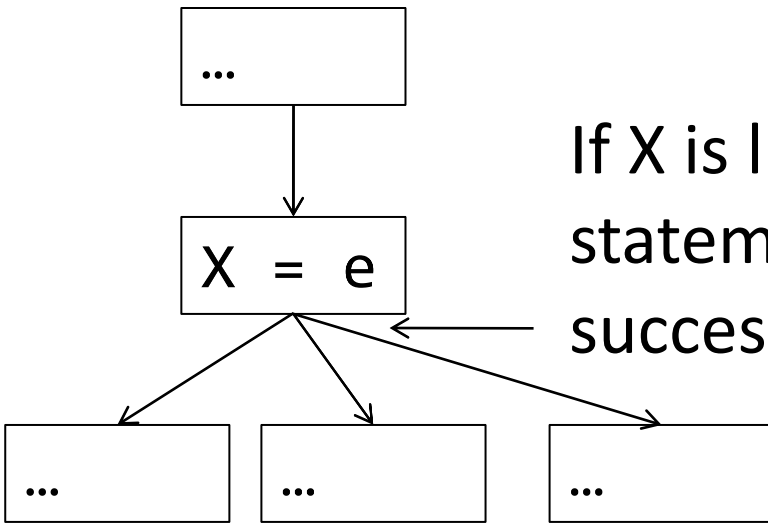
Liveness in a CFG



If X is live here (i.e. after the statement), X is used in *some* successor

- Define a set `LiveOut(b)`, where b is a basic block, as: the set of all variables live at the exit of a basic block

Liveness in a CFG

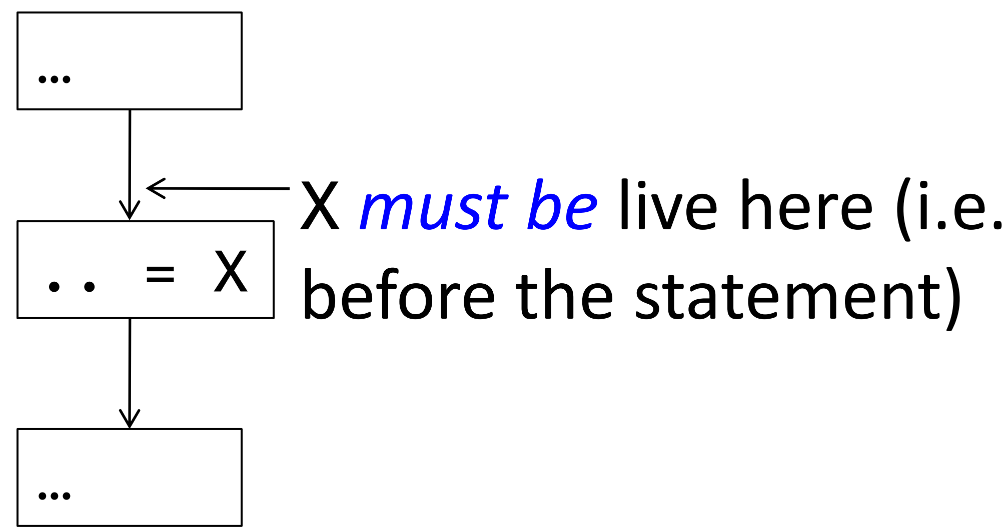


If X is live here (i.e. after the statement), X is used in *some* successor

- If S(b) is the set of all successors of b, then

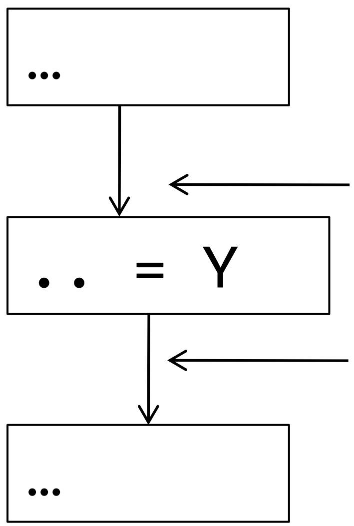
$$\text{LiveOut}(b) = \cup_{i \in S(b)} \text{LiveIn}(i)$$

Liveness in a CFG



- Define a set `LiveUse(b)`, where b is a basic block, as: the set of all variables that are used within block `b`. `LiveIn(b)` $\supseteq$ `LiveUse(b)`

Liveness in a CFG - Observation



X not live here / X is live here

If X is not live here / X is live here

- If a node neither uses nor defines X, the liveness property remains the same before and after executing the node

Liveness in a CFG

- If a variable is live on exit from b, it is either defined in b or live on entrance to b

$$\texttt{LiveIn(b)} \supseteq \texttt{LiveOut(b)} - \texttt{Def(b)}$$

- Under what scenarios can a variable be live at the entrance of a basic block?

Liveness in a CFG

- If a variable is live on exit from b, it is either defined in b or live on entrance to b

$$\texttt{LiveIn(b)} \supseteq \texttt{LiveOut(b)} - \texttt{Def(b)}$$

- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block

Liveness in a CFG

- If a variable is live on exit from b, it is either defined in b or live on entrance to b

$$\texttt{LiveIn(b)} \supseteq \texttt{LiveOut(b)} - \texttt{Def(b)}$$

- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block
  - OR the variable is live at exit and not defined within the block

Liveness in a CFG

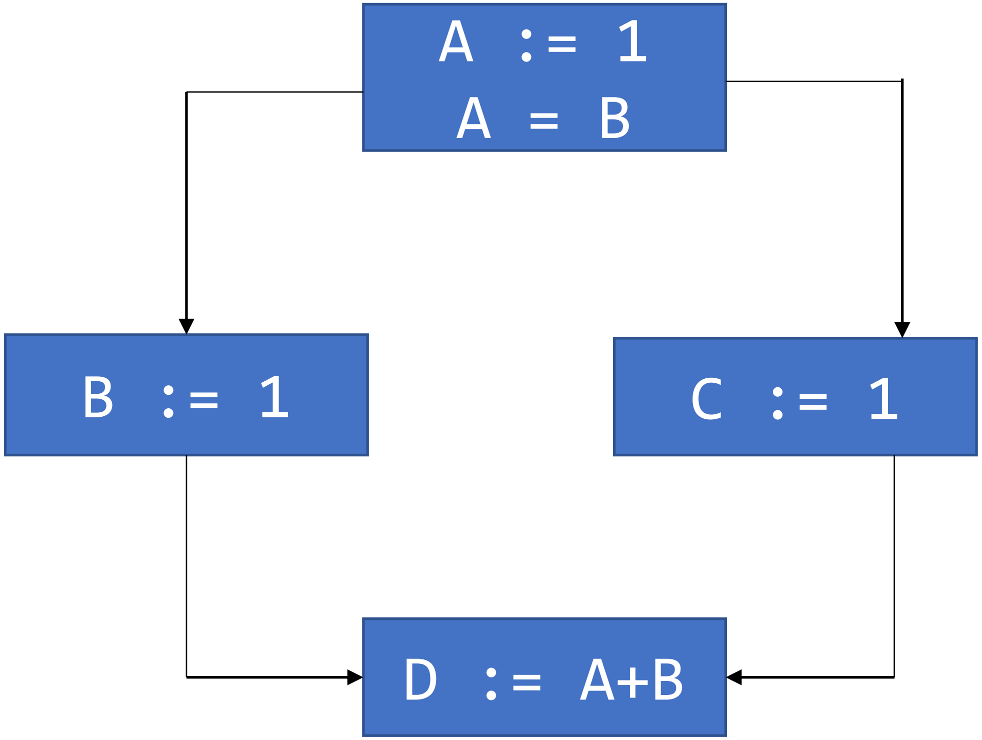
- Under what scenarios can a variable be live at the entrance of a basic block?
  - Either the variable is used in the basic block
  - OR the variable is live at exit and not defined within the block

```
LiveIn(b) = LiveUse(b) U  (LiveOut(b) –
Def(b))
```

Liveness in a CFG - Example

- Draw CFG for the code:

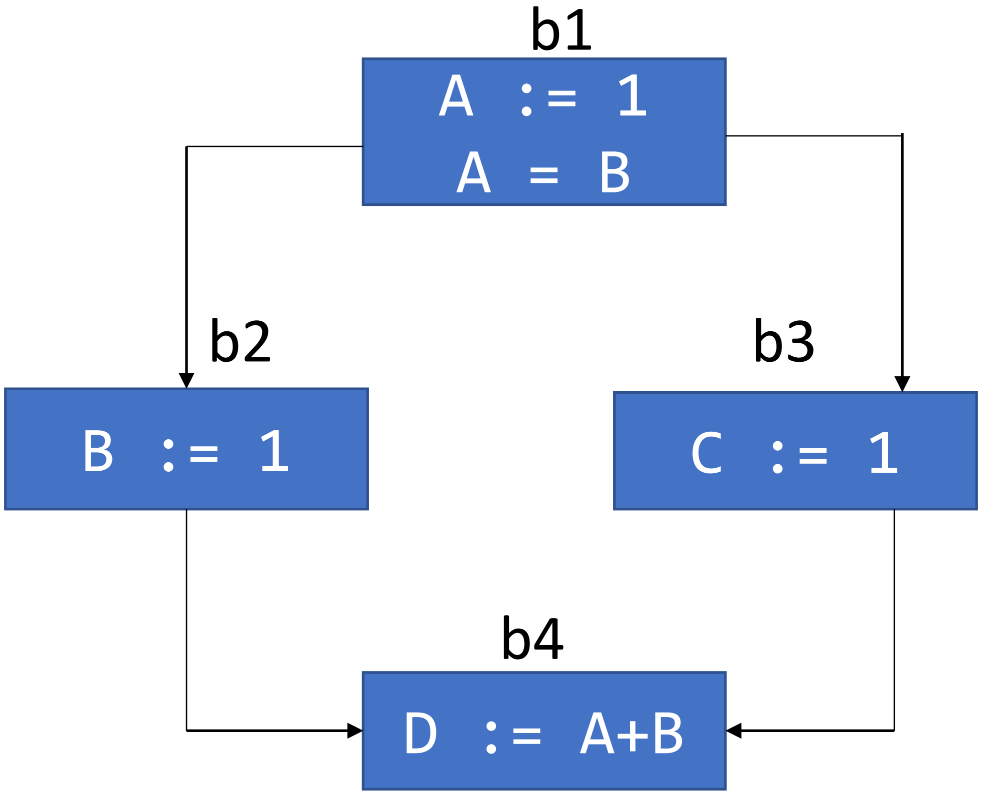
```
A:=1
if A=B then
   B:=1
else
   C:=1
endif
D:=A+B
```



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

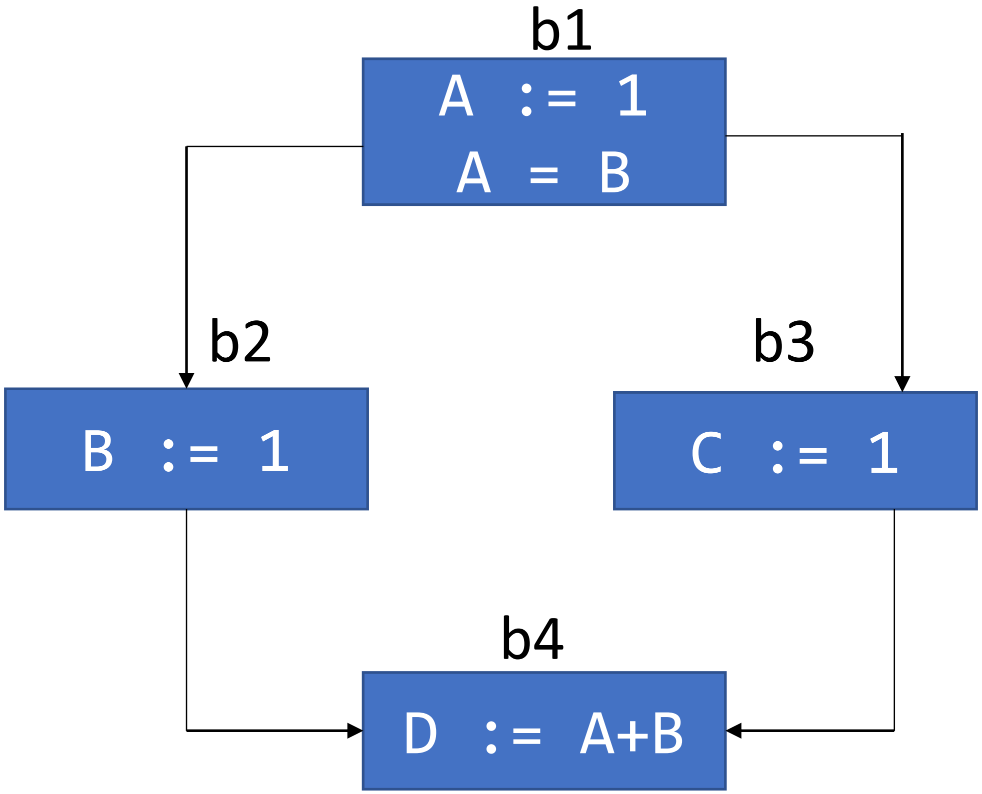
Block	Def	LiveUse
b1		
b2		
b3		
b4		



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

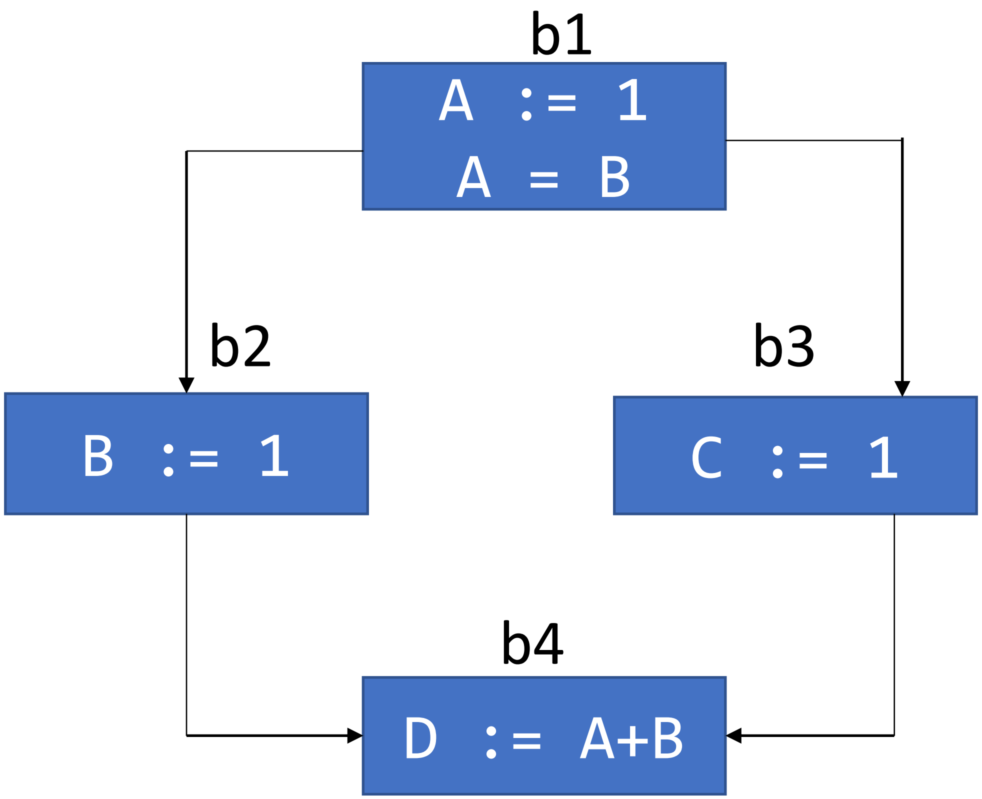
Block	Def	LiveUse
b1	{A}	{B}
b2		
b3		
b4		



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

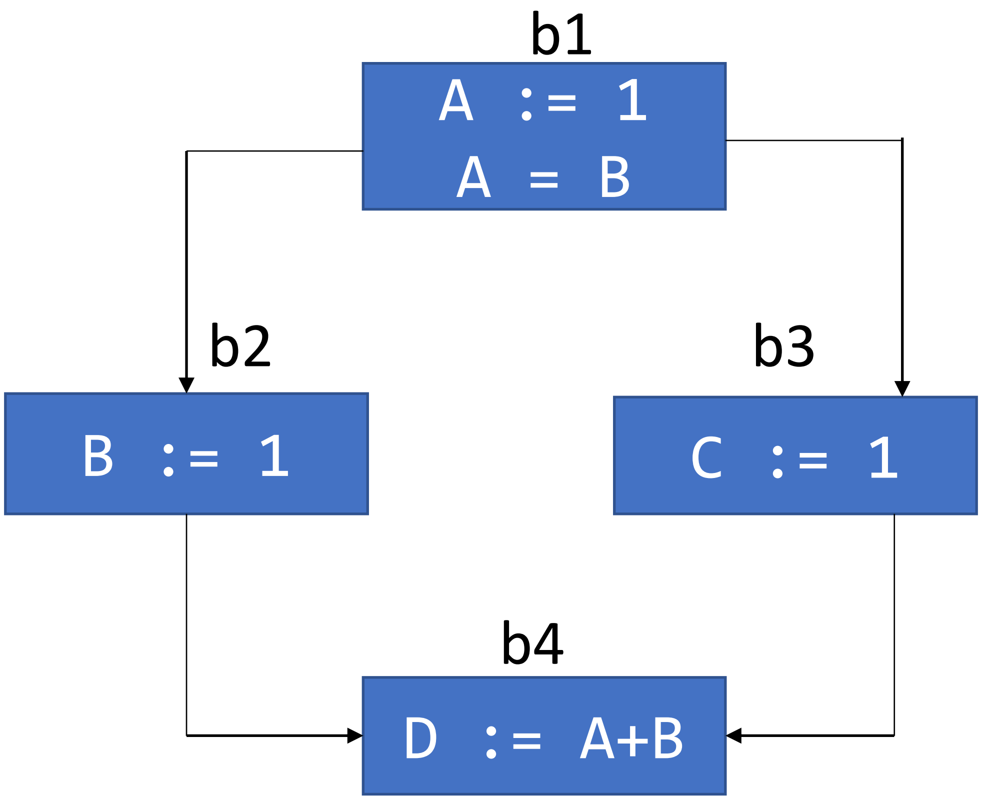
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3		
b4		



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

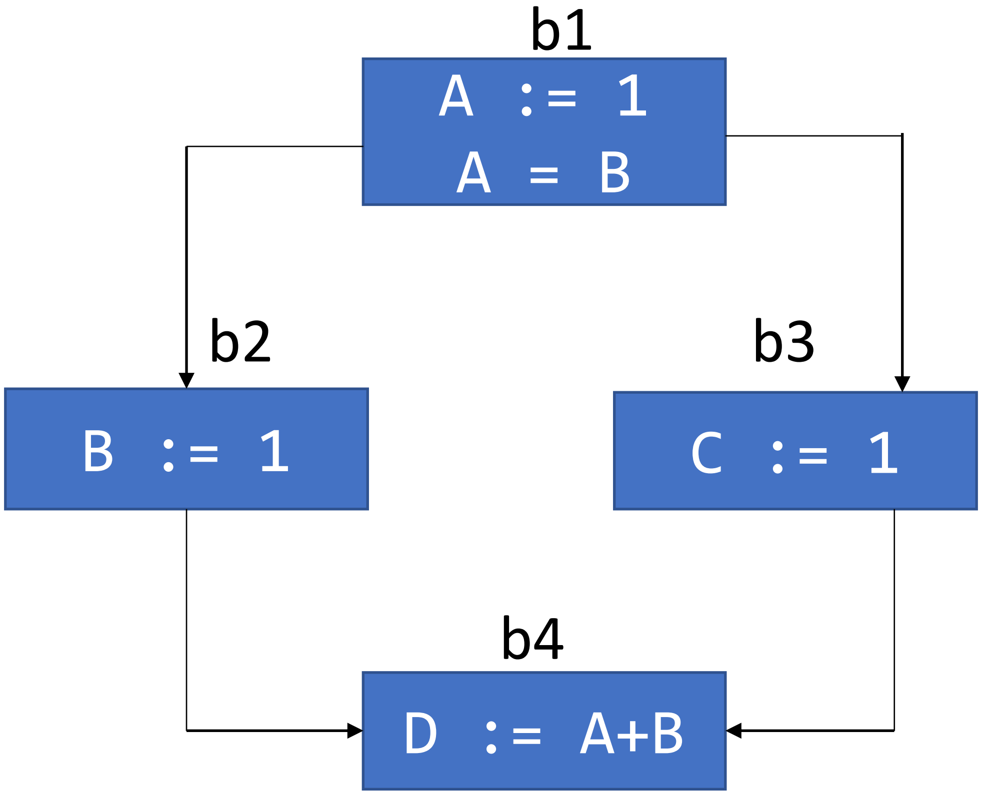
Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4		



Liveness in a CFG - Example

- Compute `Def(b)` and `LiveUse(b)` sets

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

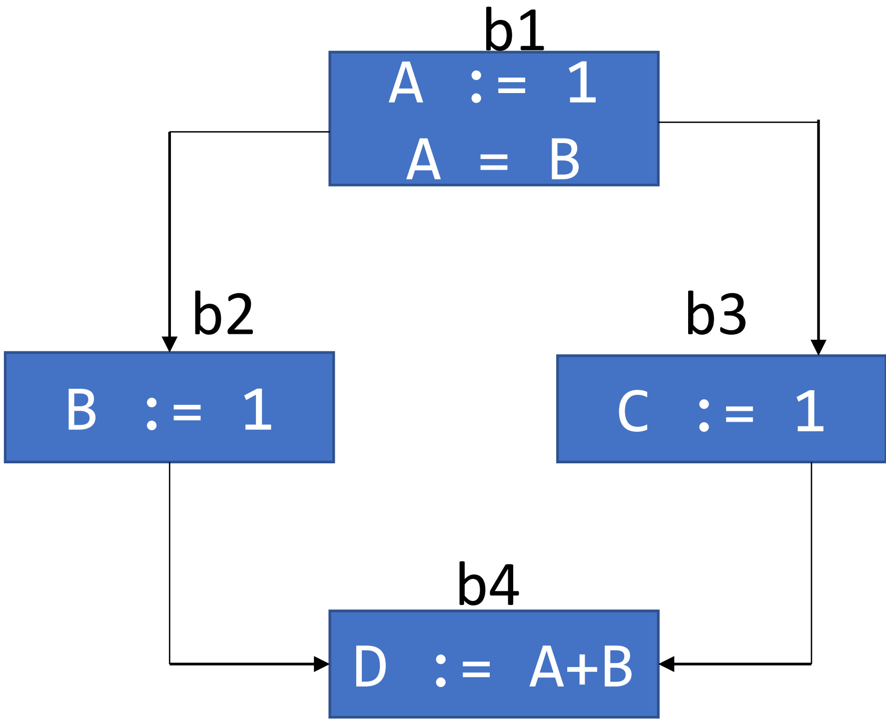


Liveness in a CFG - Example

- start from use of a variable to its definition.

Is this analysis going backward or forward w.r.t. control flow?

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

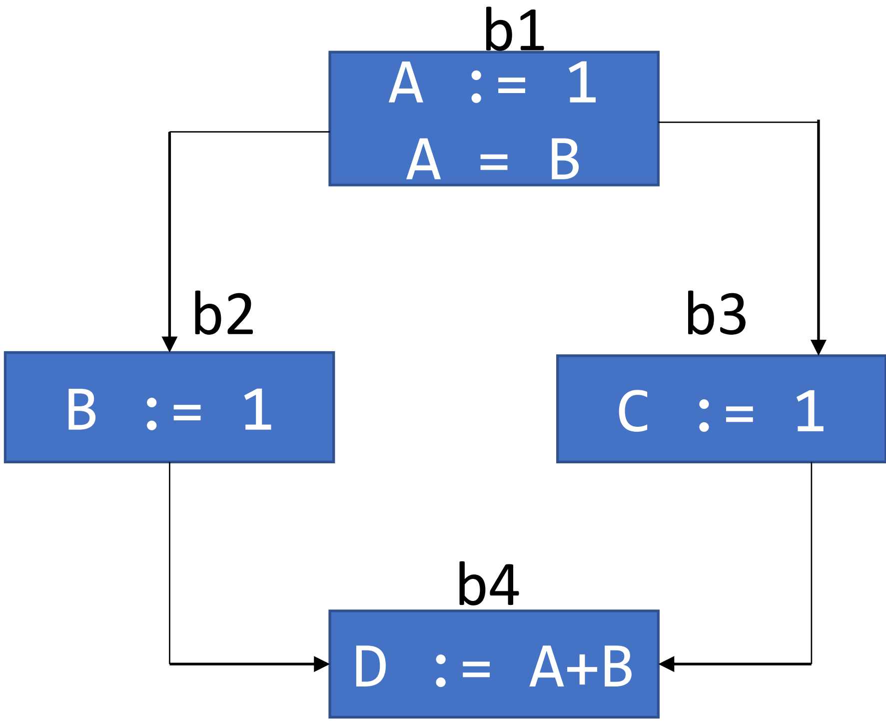


Liveness in a CFG - Example

- start from use of a variable to its definition.

Backward-flow problem

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}



Liveness in a CFG - Example

- Start from use of a variable to its definition.
- Compute:

```
LiveIn(b) = LiveUse(b) U  (LiveOut(b) – Def(b))
```

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Liveness in a CFG - Example

- Start from use of a variable to its definition.
- Compute:

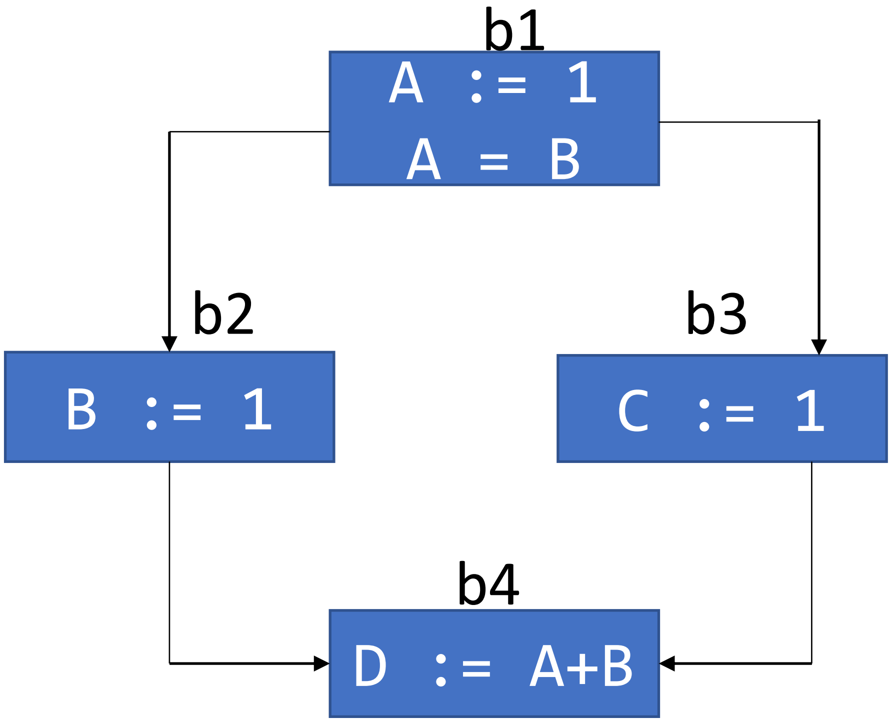
```
LiveIn(b) = LiveUse(b) U  (LiveOut(b) – Def(b))
```

Block	Def	LiveUse
b1	{A}	{B}
b2	{B}	{}
b3	{C}	{}
b4	{D}	{A,B}

Block	LiveIn	LiveOut
b1	{B}	{A,B}
b2	{A}	{A,B}
b3	{A,B}	{A,B}
b4	{A,B}	{}

Liveness in a CFG - Example

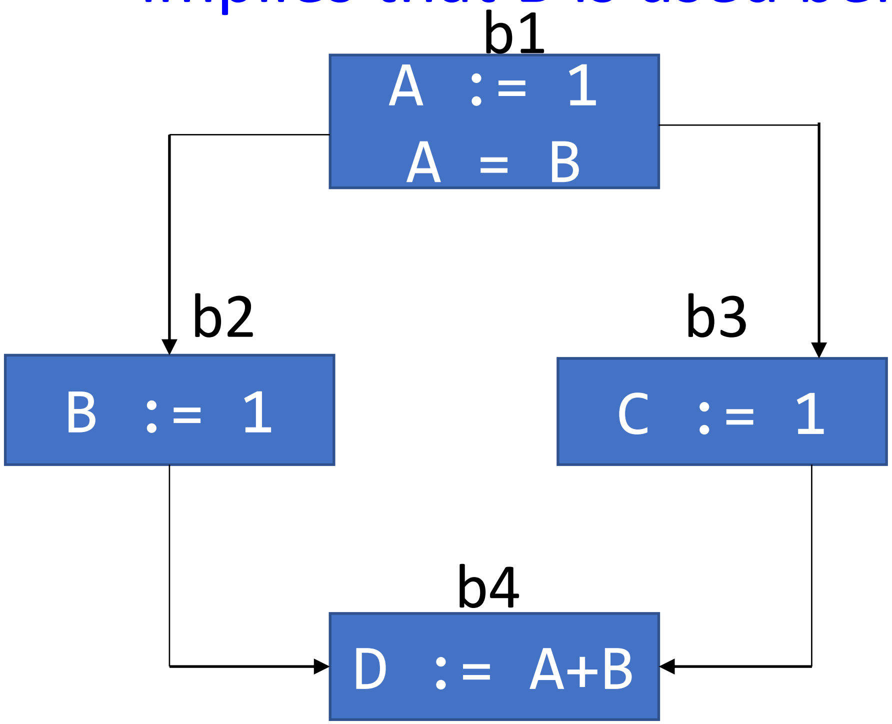
- Assume that the CFG below represents *your entire program*
    - What can you infer from the table?



Block	LiveIn	LiveOut
b1	{B}	{A,B}
b2	{A}	{A,B}
b3	{A,B}	{A,B}
b4	{A,B}	{}

Liveness in a CFG - Example

- Assume that the CFG below represents *your entire program*
    - Variable B is live in b1, the entry basic block of CFG. This implies that B is used before it is defined. An error!



Block	LiveIn	LiveOut
b1	{B}	{A,B}
b2	{A}	{A,B}
b3	{A,B}	{A,B}
b4	{A,B}	{}