CS406: Compilers
Spring 2021

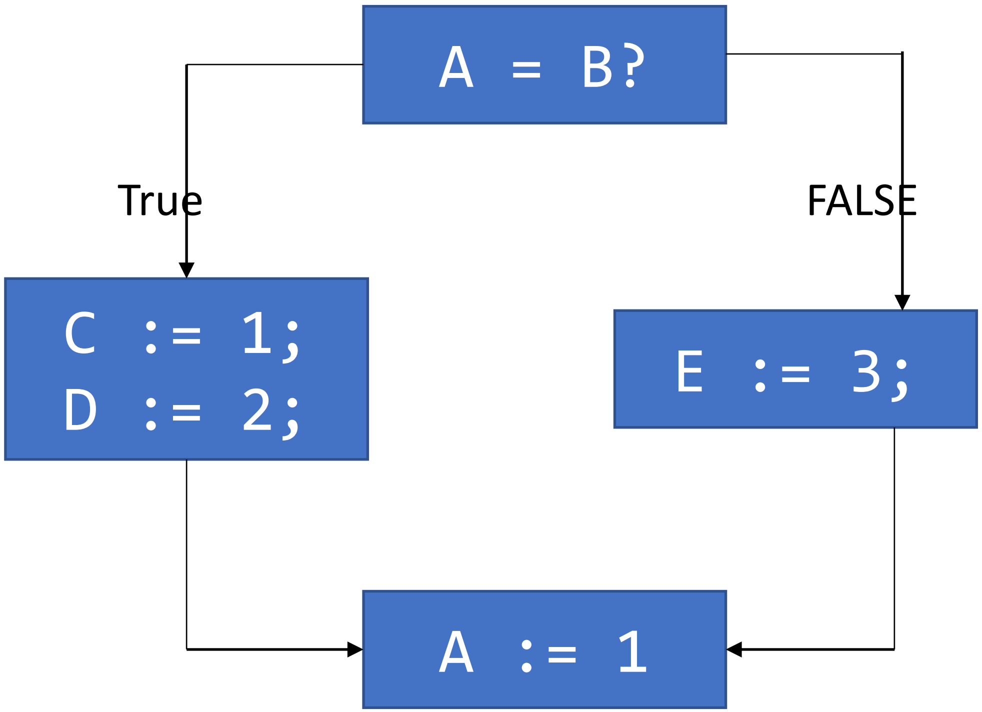
Week 12: Control Flow Graphs, Data Flow Analysis

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)
```

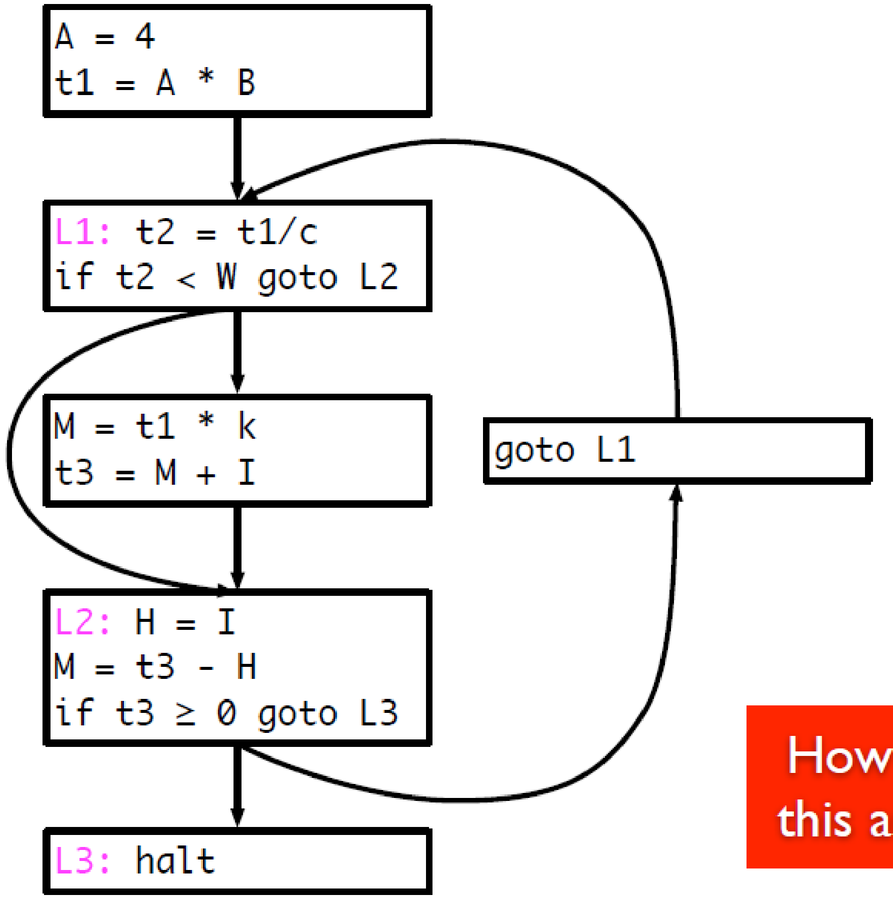

Slide Courtesy: Milind Kulkarni

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)* = i^{th} 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 =

Slide Courtesy: Milind Kulkarni

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

{{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 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}}

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 4 to block 5

Putting edges in CFG

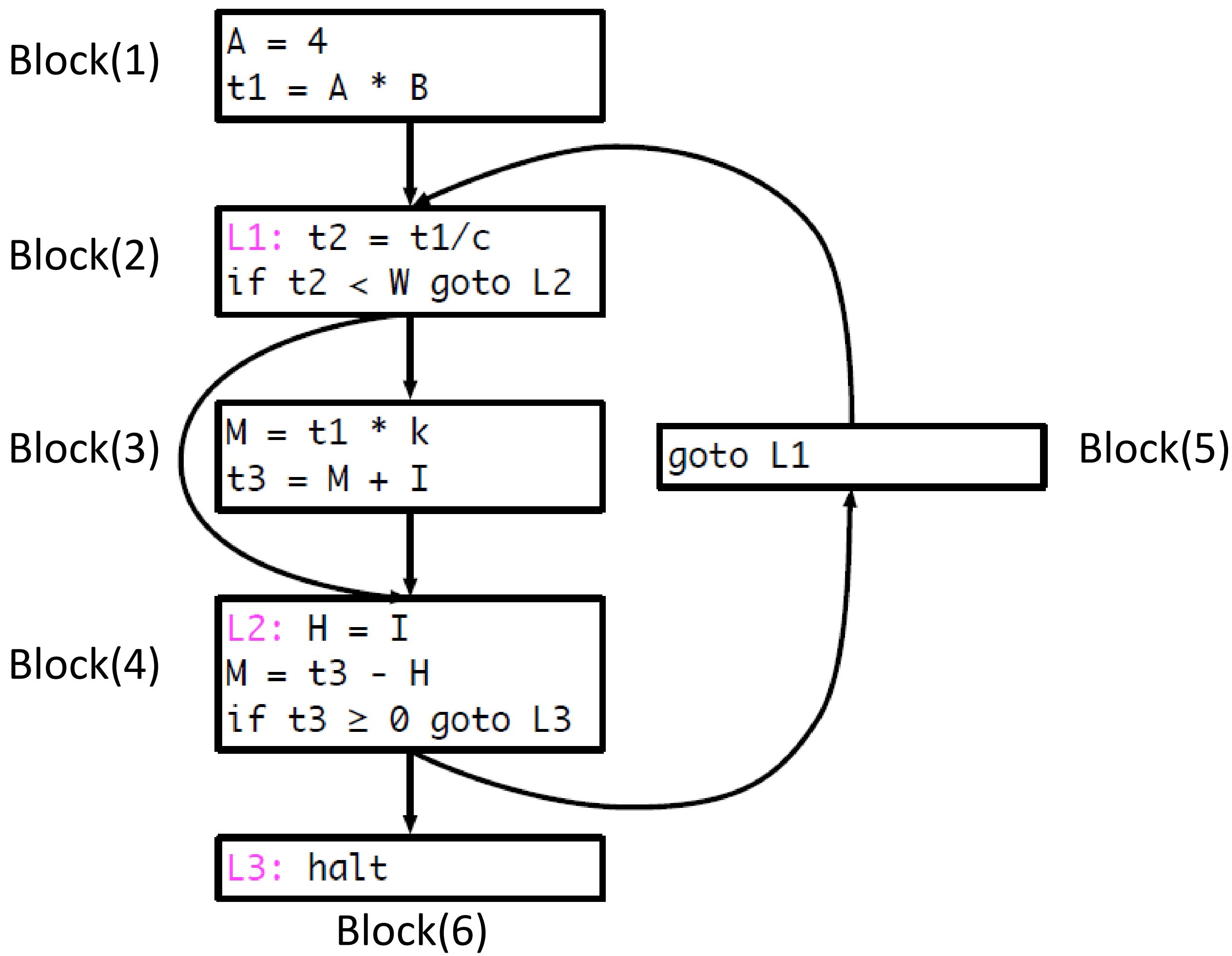
- There is a directed edge from B_1 to B_2 if
  - There is a branch from the last statement of B_1 to the first statement (leader) of B_2
  - B_2 immediately follows B_1 in program order and B_1 does not end with an unconditional branch
- Input: *block*, a sequence of basic blocks
- Output: The CFG

```
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

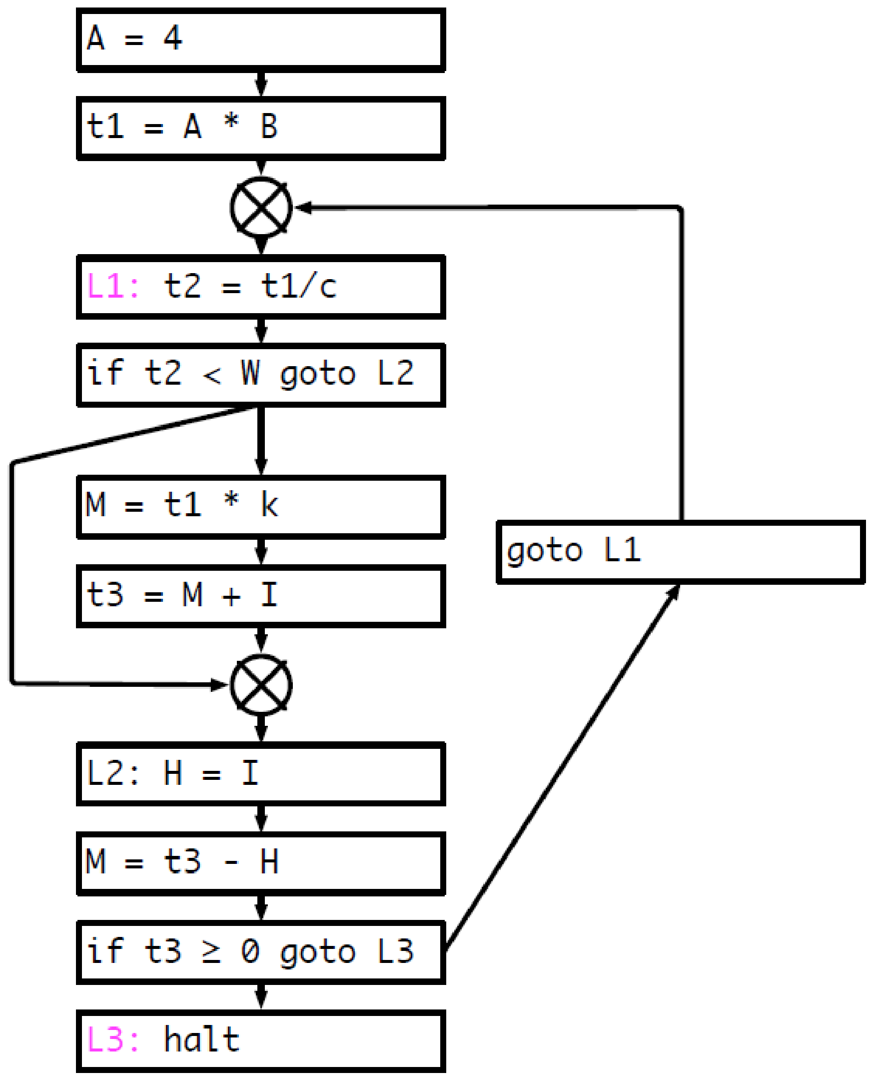


Block(1)
A = 4
t1 = A * B
Block(2)
L1: t2 = t1/c
if t2 < W goto L2
Block(3)
M = t1 * k
t3 = M + I
goto L1
Block(5)
Block(4)
L2: H = I
M = t3 - H
if t3 ≥ 0 goto L3
L3: halt
Block(6)

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

- Global Optimization i.e. beyond basic block
  - Differentiating aspect of normal and optimizing compilers
  - Most frequent targets of global optimization: Loops

how do we identify loops in CFGs?

Loops in CFGs

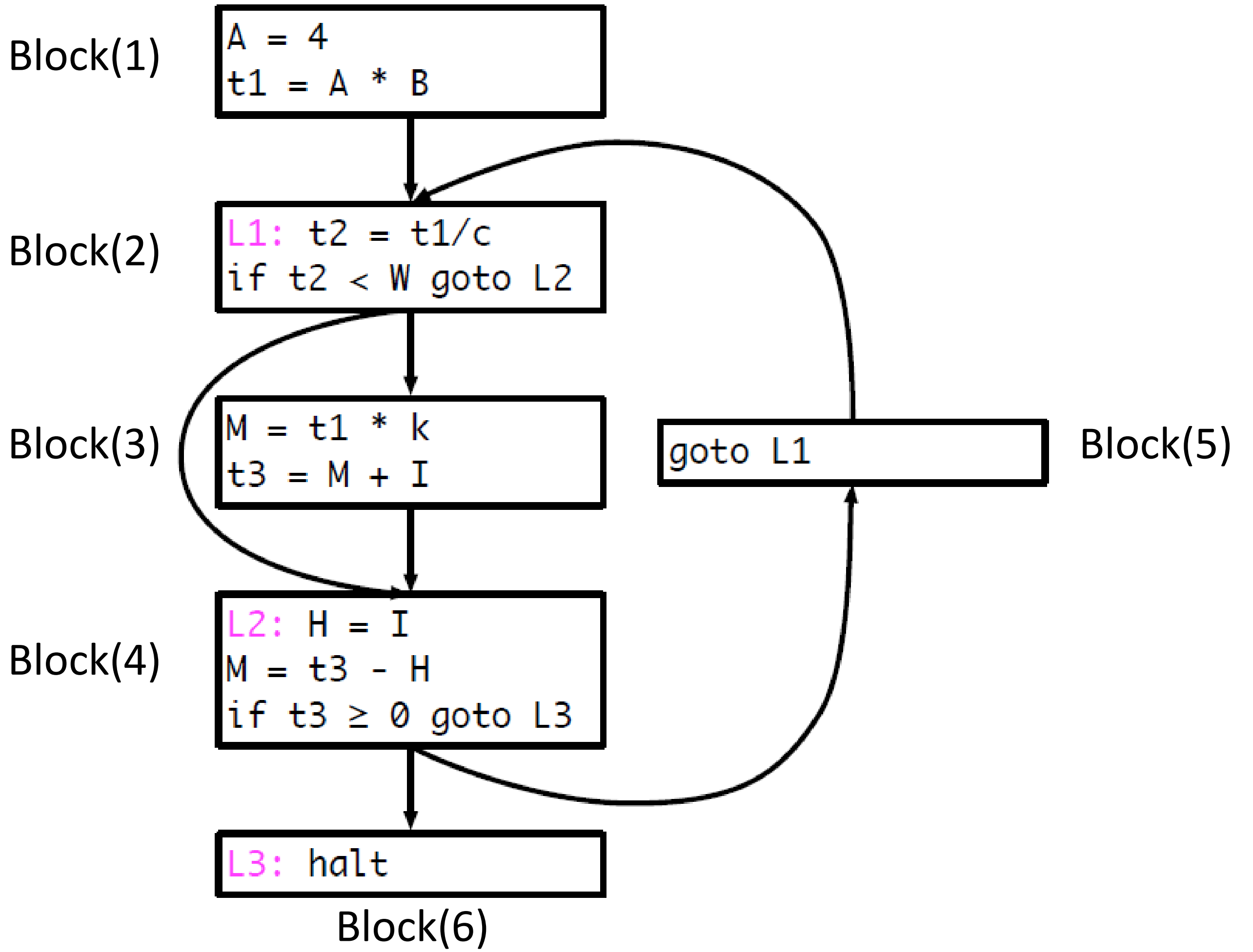
- Loops – **how do we identify loops in CFGs?**

  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.

CFGs - Loops



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

CFGs - Loops

- Should be careful while doing optimization of loops

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

CFGs - Loops

- 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.

CFGs - Loops

- 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?

CFGs - Loops

- 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?

Safety and Profitability

- Safety
  - Is the code produced after optimization producing same result?
  - E.g. moving I out of the loop introduced exception

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

- Profitability
  - Is the code produced after optimization running faster or uses less memory or triggers lesser number of page faults etc.
  - E.g. if the loop is executed zero times, moving I out is not profitable

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

Example: Constant Propagation and Dead Code Elimination

```
X = 1
Y = X + 2
Z = Y + A
```

⇨

```
X = 1
Y = 1 + 2
Z = Y + A
```

⇨

~~X = 1~~

```
Y = 1 + 2
Z = Y + A
```

Constant Propagation

Dead Code Elimination

Example: constant propagation

- Goal: determine when variables take on constant values
- Why? Can enable many optimizations
  - Constant folding

```
x = 1;                                       x = 1;
y = x + 2;                                   y = 3;
if (x > z) then y = 5          ---->         if (x > z) then y = 5
... y ...                                    ... y ...
```

  - Create dead code

```
x = 1;                                       x = 1;
y = x + 2;                                   y = 3; //dead code
if (y > x) then y = 5          ---->         if (true) then y = 5 //simplify!
... y ...                                    ... y ...
```

Exercise – Constant Propagation

```
1.  X := 2
2.  Label1:
3.  Y := X + 1
4.  if Z > 8 goto Label2
5.  X := 3
6.  X := X + 5
7.  Y := X + 5
8.  X := 2
9.  if Z > 10 goto Label1
10. X := 3
11. Label2:
12. Y := X + 2
13. X := 0
14. goto Label3
15. X := 10
16. X := X + X
17. Label3:
18. Y := X + 1
```

Which lines using X could be replaced with a constant value? (apply only constant propagation)

How can we find constants?

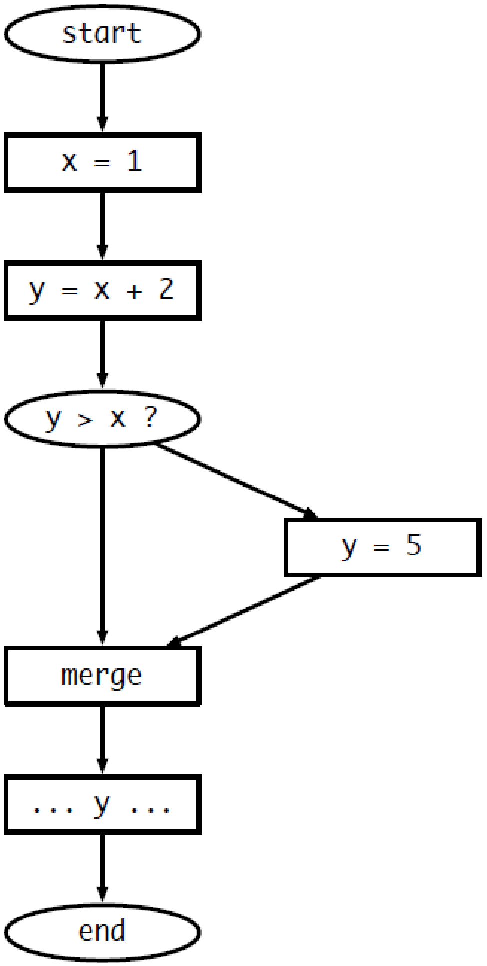
- Ideal: run program and see which variables are constant
  - Problem: variables can be constant with some inputs, not others – need an approach that works for all inputs!
  - Problem: program can run forever (infinite loops?) – need an approach that we know will finish
- Idea: run program *symbolically*
  - Essentially, keep track of whether a variable is constant or not constant (but nothing else)

Overview of algorithm

- Build control flow graph
  - We'll use statement-level CFG (with merge nodes) for this
- Perform symbolic evaluation
  - Keep track of whether variables are constant or not
- Replace constant-valued variable uses with their values, try to simplify expressions and control flow

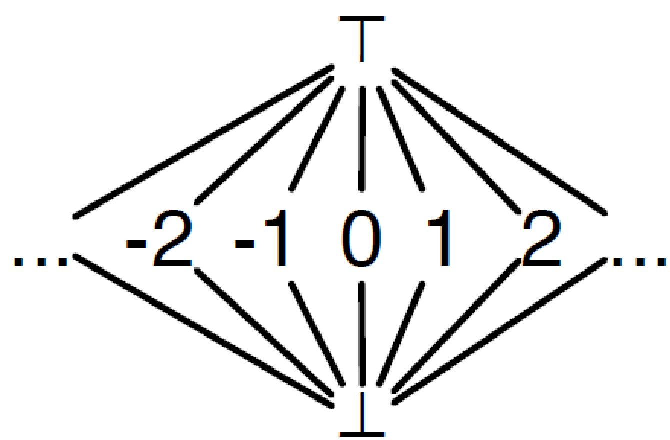
Build CFG

```
x = 1;
y = x + 2;
if (y > x) then y = 5;
... y ...
```



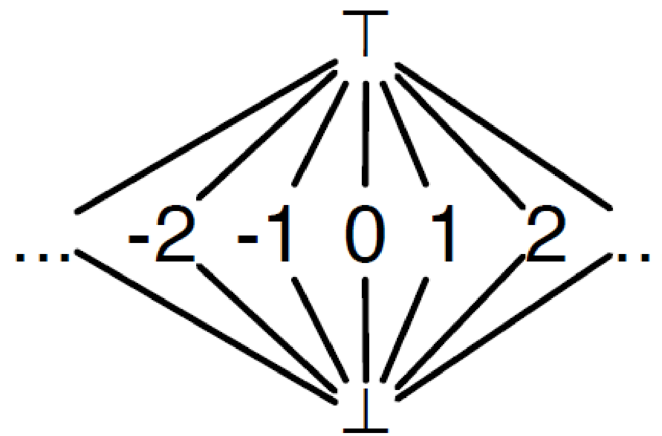
Symbolic evaluation

- Idea: replace each value with a symbol
    - constant (specify which), no information, definitely not constant
- Can organize these possible values in a *lattice*
    - Set of possible values, arranged from least information to most information



Symbolic evaluation

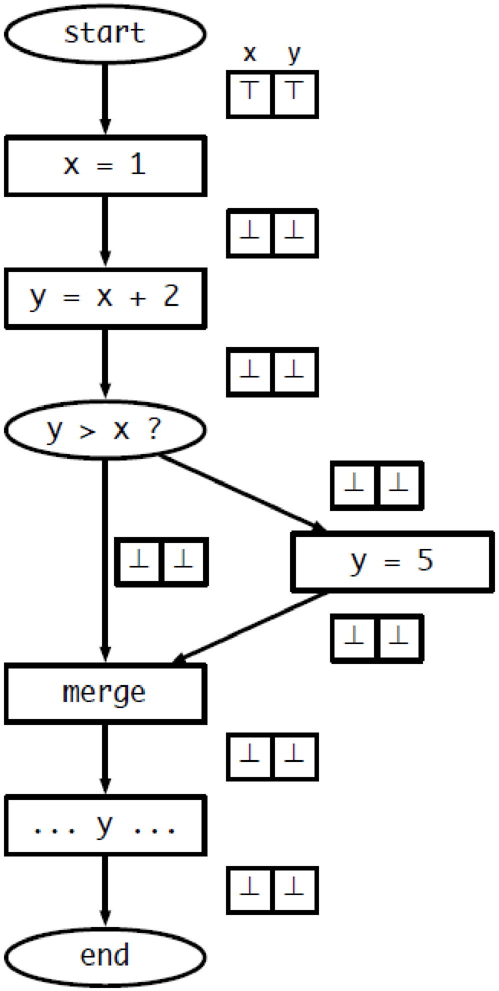
- Evaluate expressions symbolically: eval(e, V_{in})
  - If e evaluates to a constant, return that value. If any input is $\top$ (or $\bot$), return $\top$ (or $\bot$)
    - Why?
- Two special operations on lattice
  - meet(a, b) – highest value less than or equal to both a and b
  - join(a, b) – lowest value greater than or equal to both a and b



Join often written as a $\sqcup$ b
Meet often written as a $\sqcap$ b

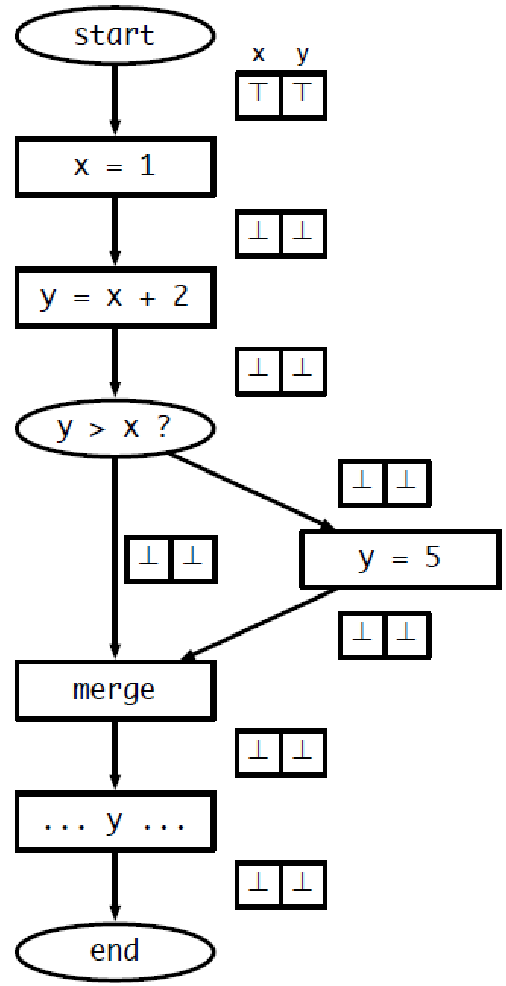
Putting it together

- Keep track of the symbolic value of a variable at every program point (on every CFG edge)
  - State vector
- What should our initial value be?
  - Starting state vector is all $\top$
    - Can't make any assumptions about inputs – must assume not constant
  - Everything else starts as $\perp$, since we have no information about the variable at that point



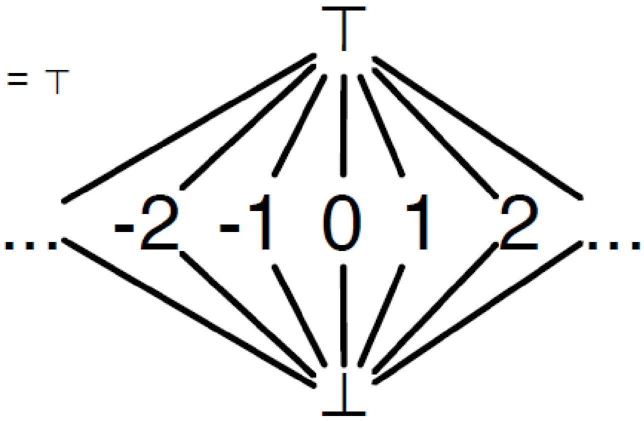
Executing symbolically

- For each statement `t = e` evaluate `e` using V_{in}, update value for `t` and propagate state vector to next statement
- What about switches?
  - If `e` is true or false, propagate V_{in} to appropriate branch
  - What if we can't tell?
    - Propagate V_{in} to both branches, and symbolically execute both sides
- What do we do at merges?



Handling merges

- Have two different V_{in}s coming from two different paths
- Goal: want new value for V_{in} to be *safe* (shouldn't generate wrong information), and we don't know which path we actually took
- Consider a single variable. Several situations:
  - $V_1 = \perp, V_2 = * \rightarrow V_{out} = *$
  - V_1 = constant x, $V_2 = x \rightarrow V_{out} = x$
  - V_1 = constant x, V_2 = constant y $\rightarrow V_{out} = \top$
  - $V_1 = \top, V_2 = * \rightarrow V_{out} = \top$
- Generalization:
  - $V_{out} = V_1 \sqcup V_2$

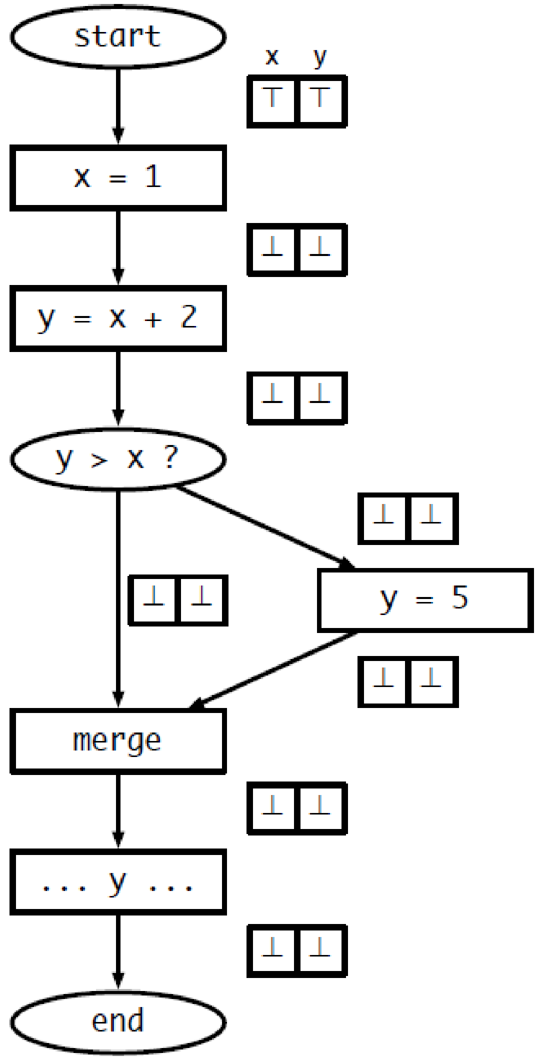


Result: worklist algorithm

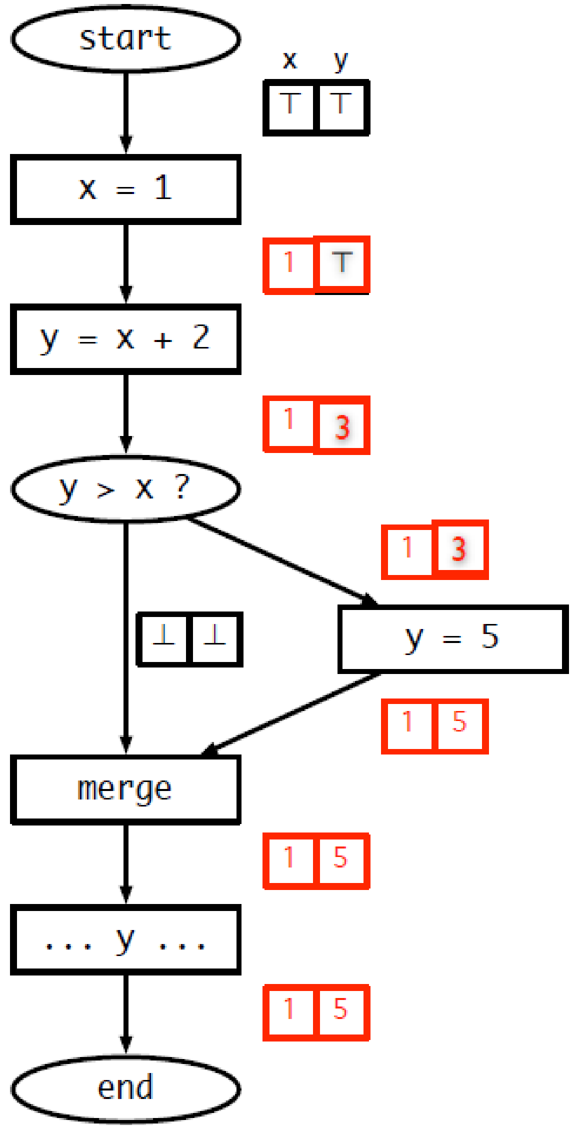
- Associate state vector with each edge of CFG, initialize all values to $\perp$, worklist has just start edge
  - While worklist not empty, do:

```
Process the next edge from worklist

Symbolically evaluate target node of edge using input state vector

If target node is assignment (x = e), propagate Vin[eval(e)/x] to
output edge

If target node is branch (e?)

 If eval(e) is true or false, propagate Vin to appropriate    output
edge

 Else, propagate Vin along both output edges

If target node is merge, propagate join(all Vin) to output edge

If any output edge state vector has changed, add it to worklist
```

Running example



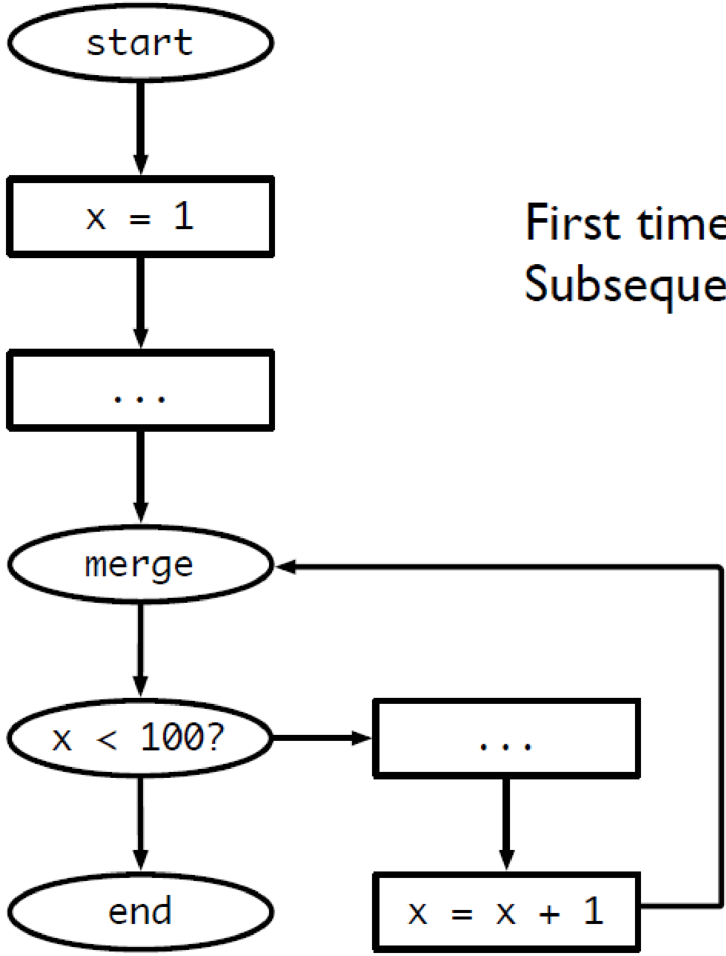
Running example



What do we do about loops?

- Unless a loop never executes, symbolic execution looks like it will keep going around to the same nodes over and over again
- Insight: if the input state vector(s) for a node don't change, then its output doesn't change
  - If input stops changing, then we are done!
- Claim: input will eventually stop changing. Why?

Loop example



First time through loop, x = 1
Subsequent times, x = $\top$

Complexity of algorithm

- V = # of variables, E = # of edges
- Height of lattice = 2 → each state vector can be updated at most $2 * V$ times.
- So each edge is processed at most $2 * V$ times, so we process at most $2 * E * V$ elements in the worklist.
- Cost to process a node: $O(V)$
- Overall, algorithm takes $O(EV^2)$ time

Question

- Can we generalize this algorithm and use it for more analyses?

Constant propagation

- Step 1: choose lattice (which values are you going to track during symbolic execution)?
  - Use constant lattice
- Step 2: choose direction of dataflow (if executing symbolically, can run program backwards!)
  - Run forward through program
- Step 3: create *transfer functions*
  - How does executing a statement change the symbolic state?
- Step 4: choose *confluence operator*
  - What do do at merges? For constant propagation, use join