

Optimization 1

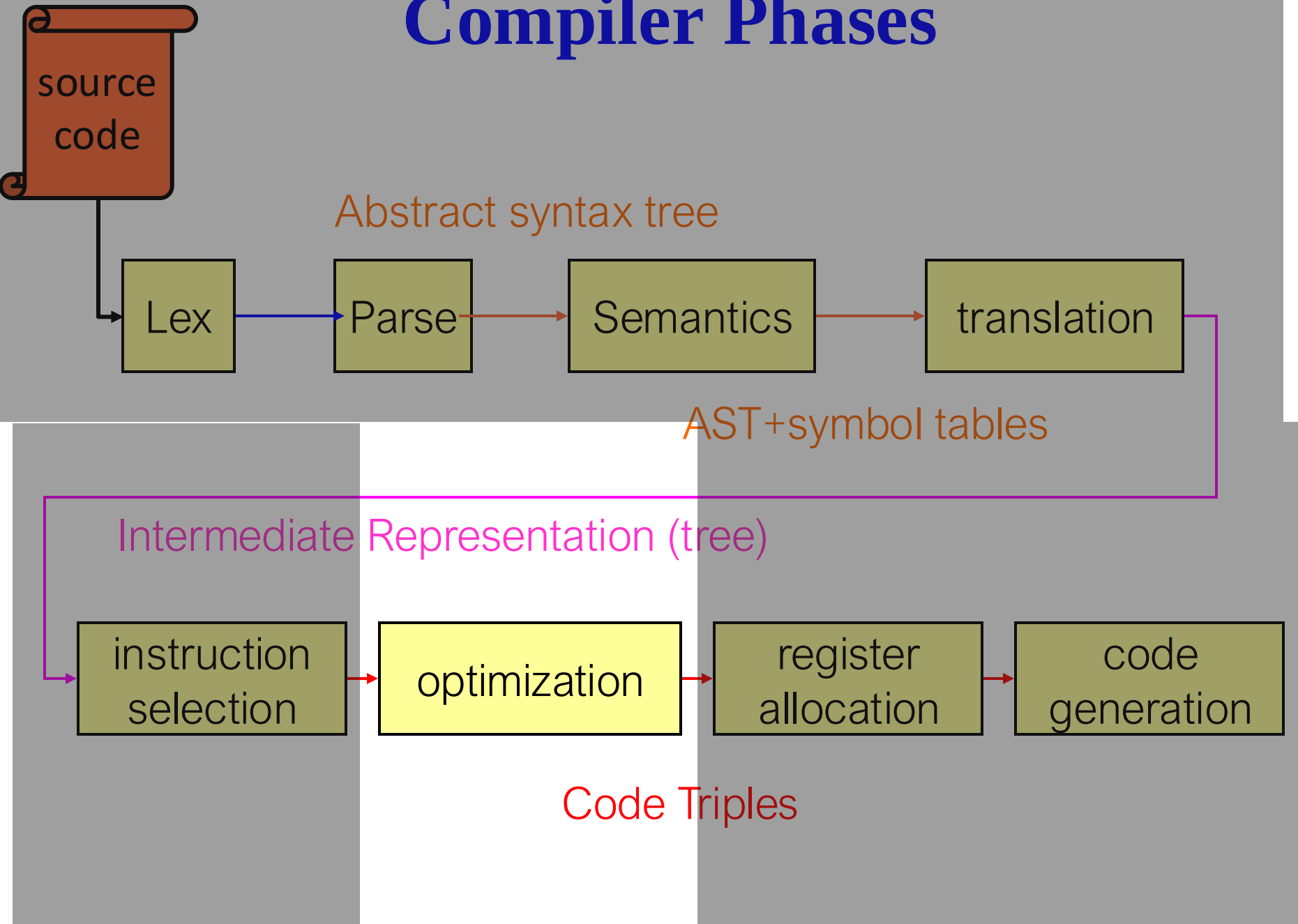
C

15-411/15-611 Compiler Design

Seth Copen Goldstein

March 20, 2025

Compiler Phases



Scope of Optimizations

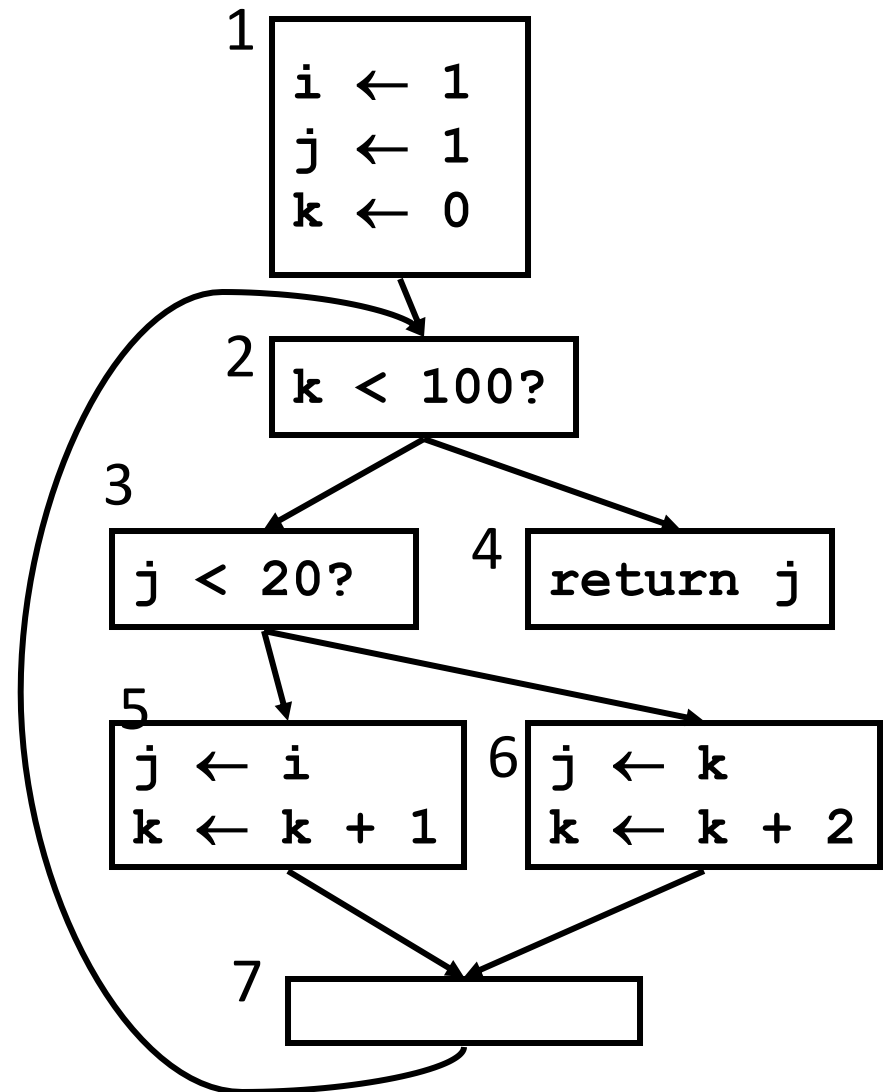
- Peephole
- Local
- Global
- Whole File
- Whole Program

Plan

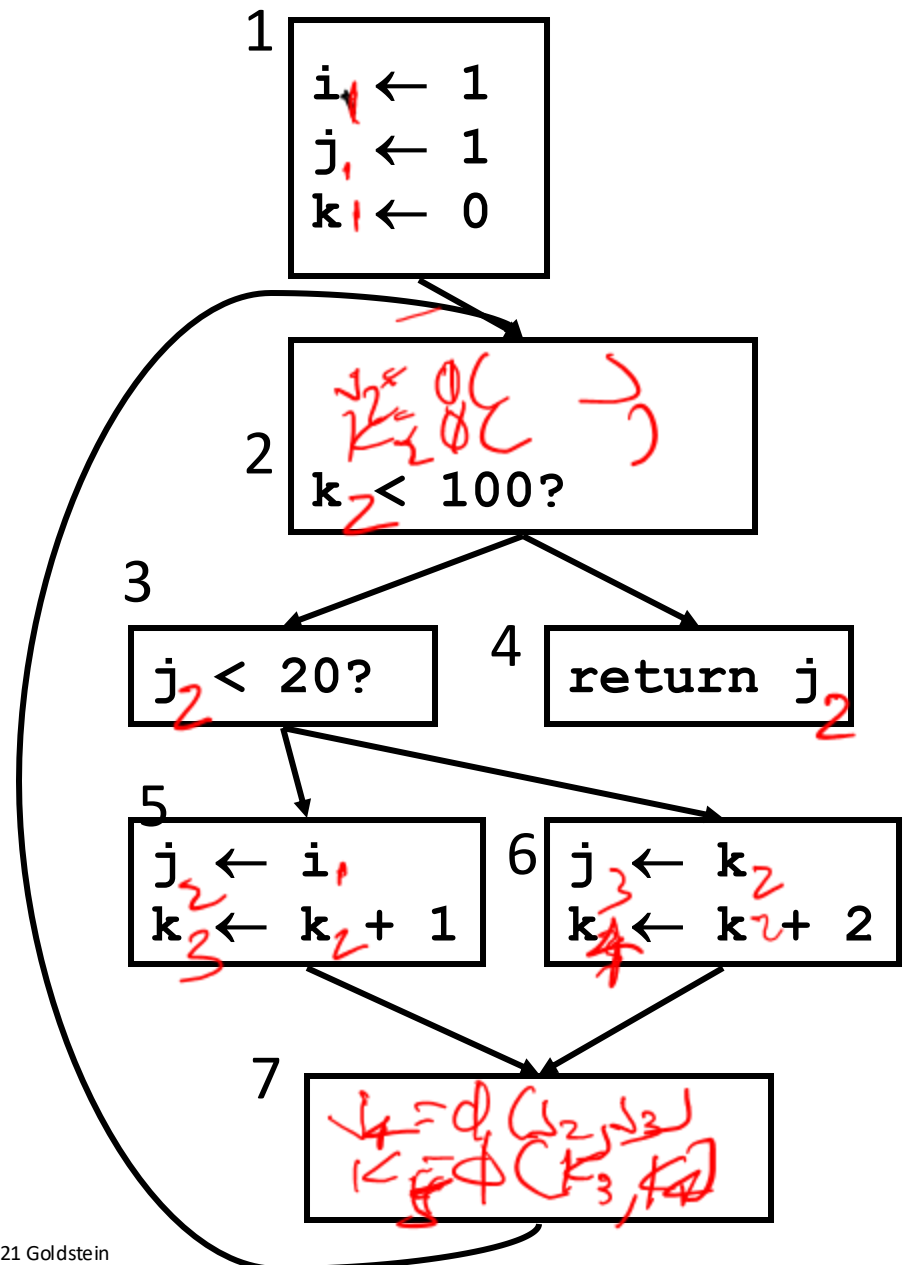
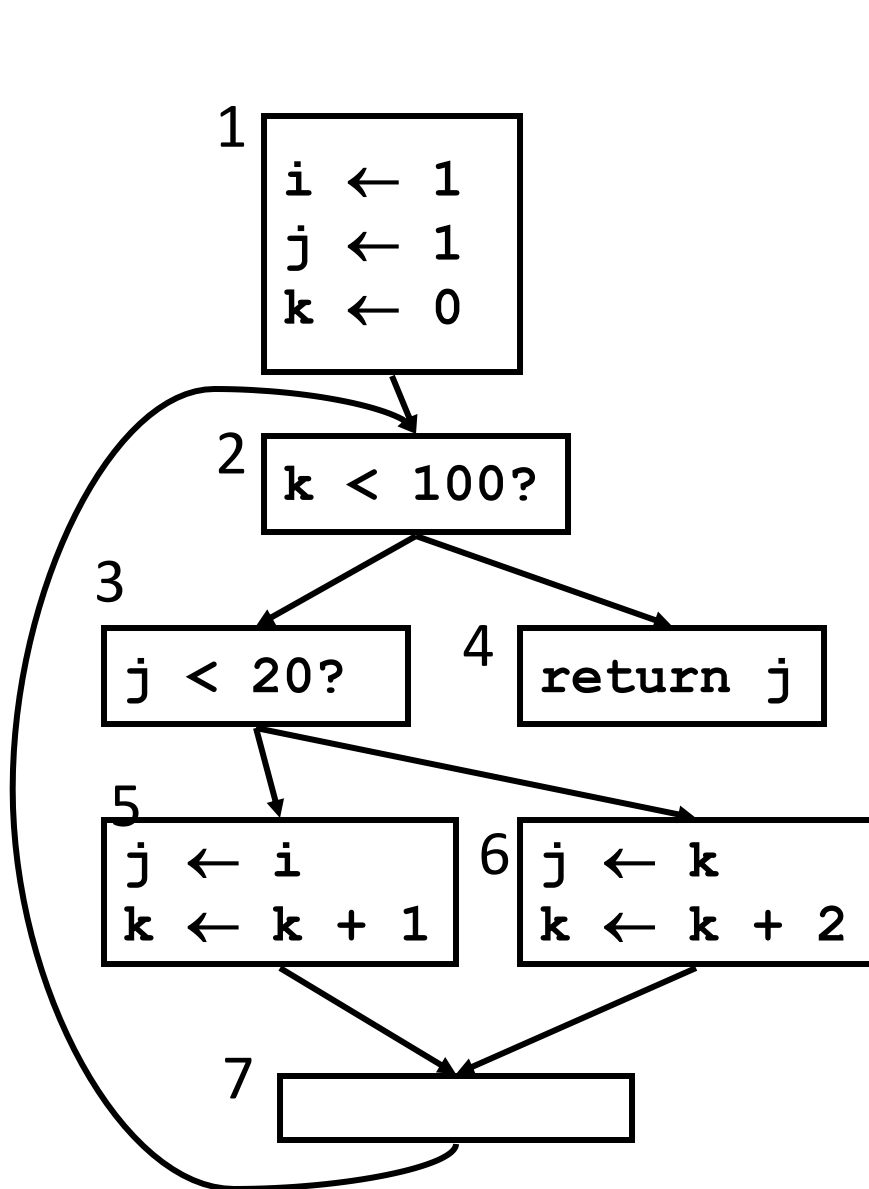
- (Re-)Motivation of SSA
 - SCCP
 - ADCE
 - Control Dependence and Dominance
- Introduction to Peephole
- Finding Loops
- Simple Loop-based Optimizations

Lets optimize the following:

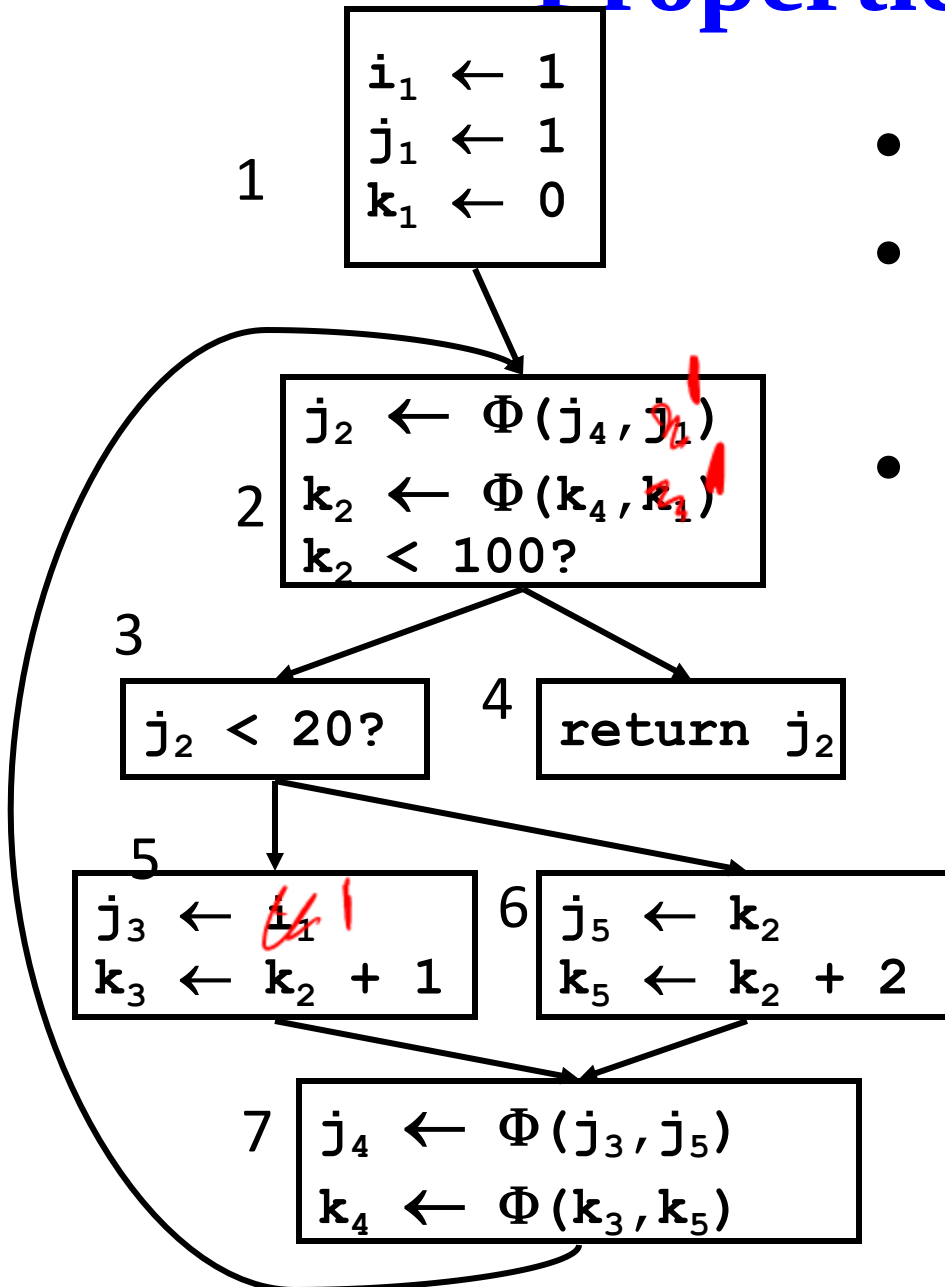
```
i=1;  
j=1;  
k=0;  
while (k<100) {  
    if (j<20) {  
        j=i;  
        k++;  
    } else {  
        j=k;  
        k+=2;  
    }  
}  
return j;
```



First, turn into SSA

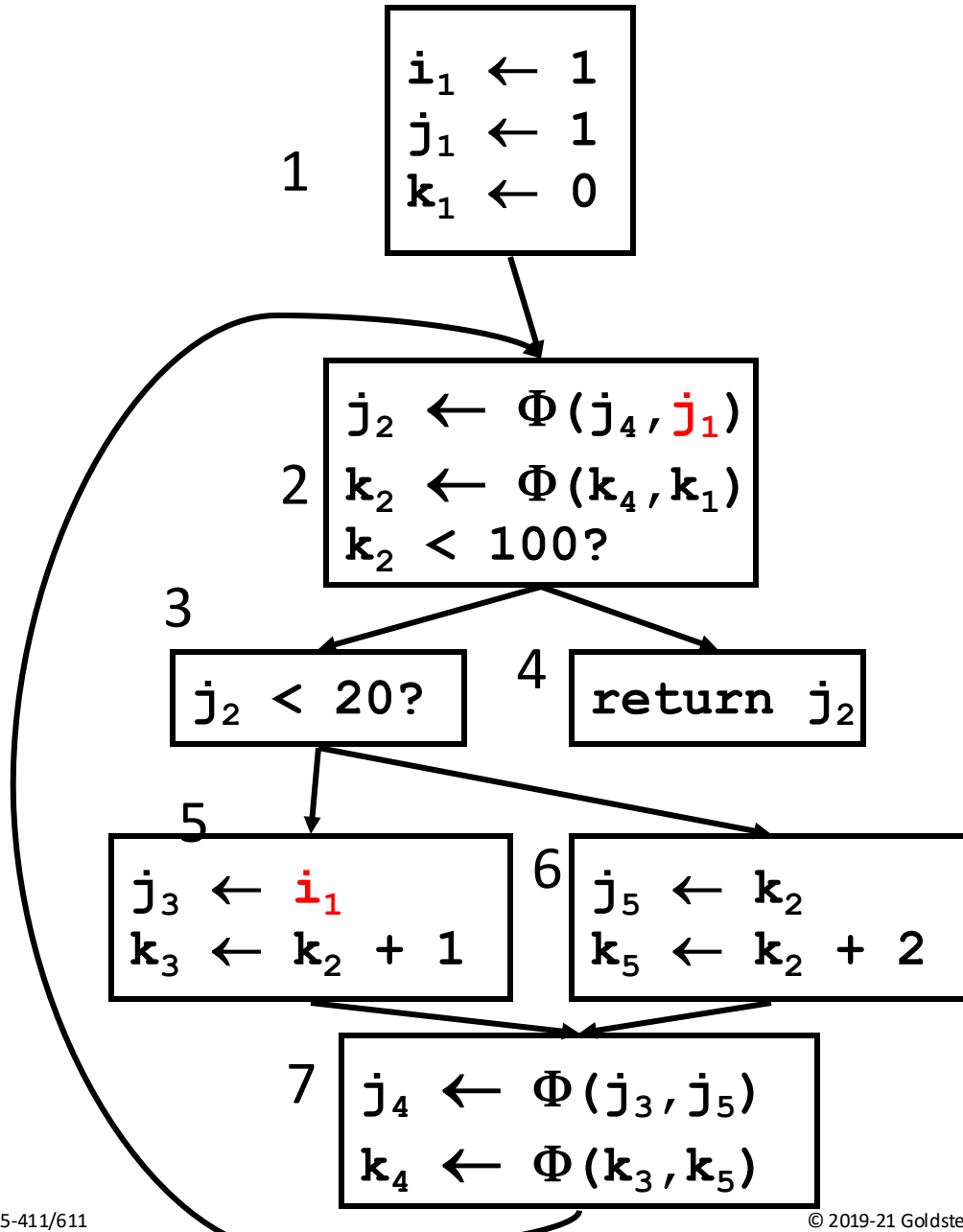


Properties of SSA



- Only 1 assignment per variable
- definitions dominate uses
- Can we use this to help with constant propagation?

Constant Propagation

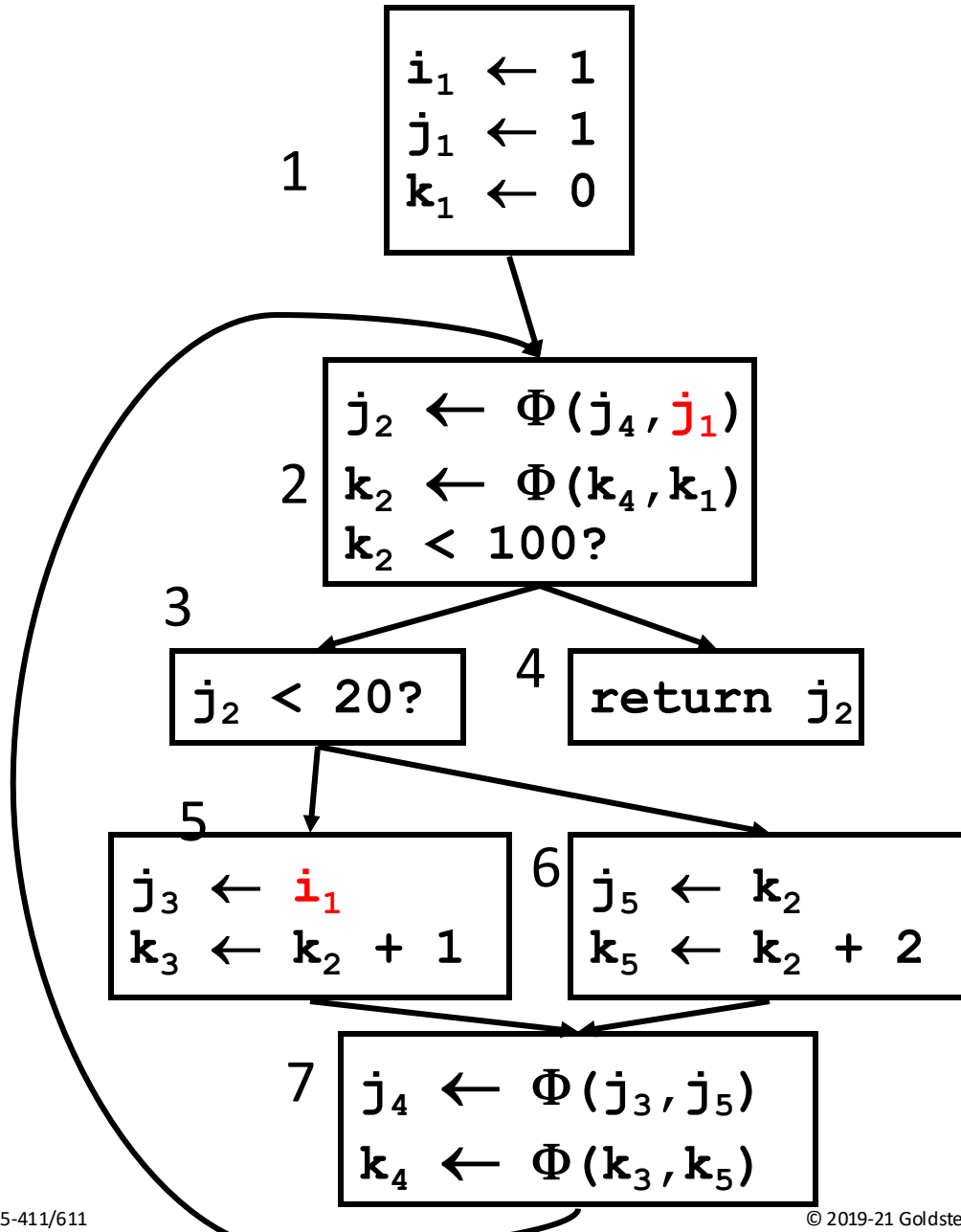


Constant Propagation

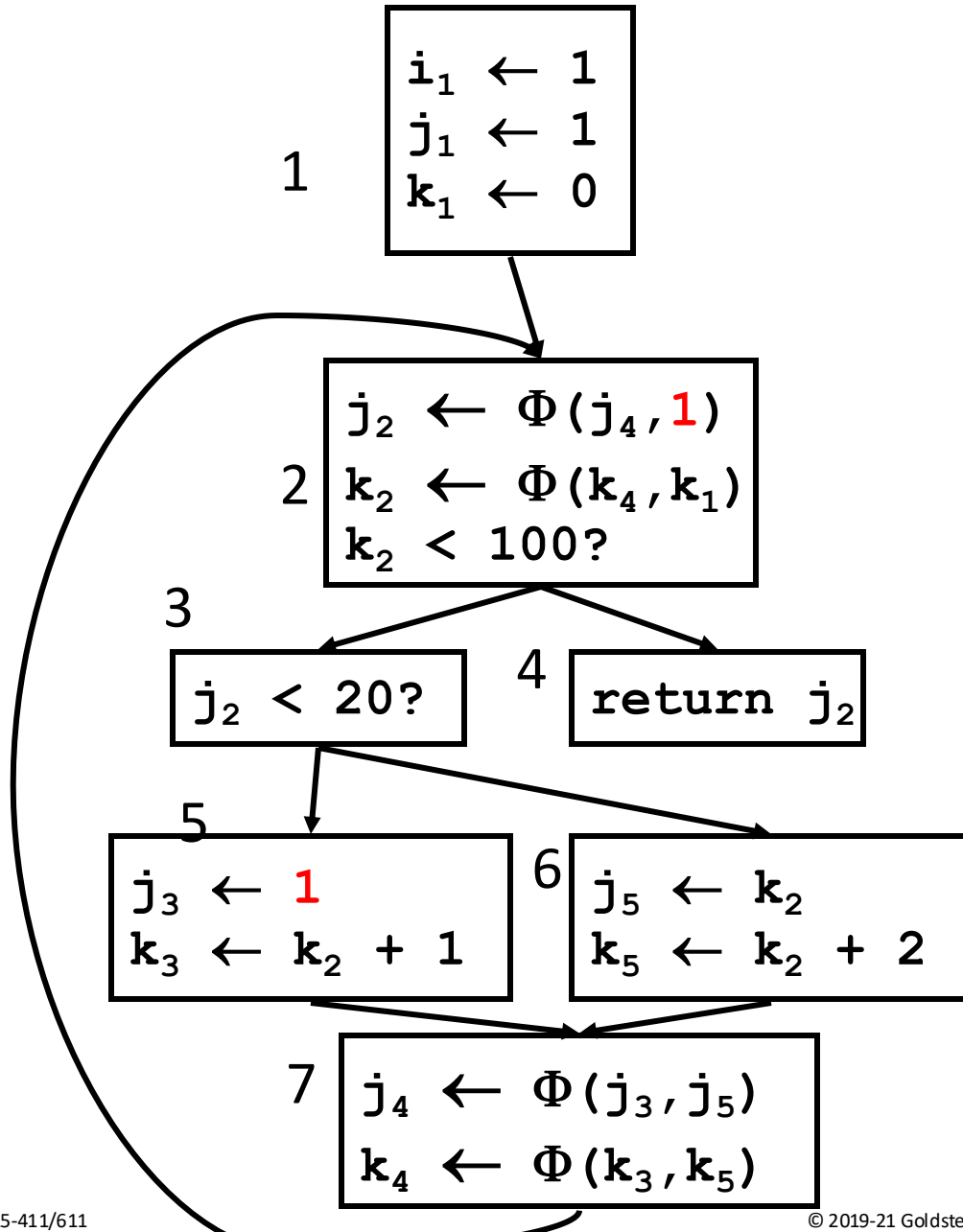
- If " $v \leftarrow c$ ", replace all uses of v with c
- If " $v \leftarrow \Phi(c, c, c)$ " replace all uses of v with c
 $W \leftarrow$ list of all defs

```
while !W.isEmpty {  
    Stmt S  $\leftarrow$  W.removeOne  
    if S has form " $v \leftarrow \Phi(c, \dots, c)$ "  
        replace S with  $V \leftarrow c$   
    if S has form " $v \leftarrow c$ " then  
        delete S  
    foreach stmt U that uses  $v$ ,  
        replace  $v$  with  $c$  in U  
    W.add(U)  
}
```

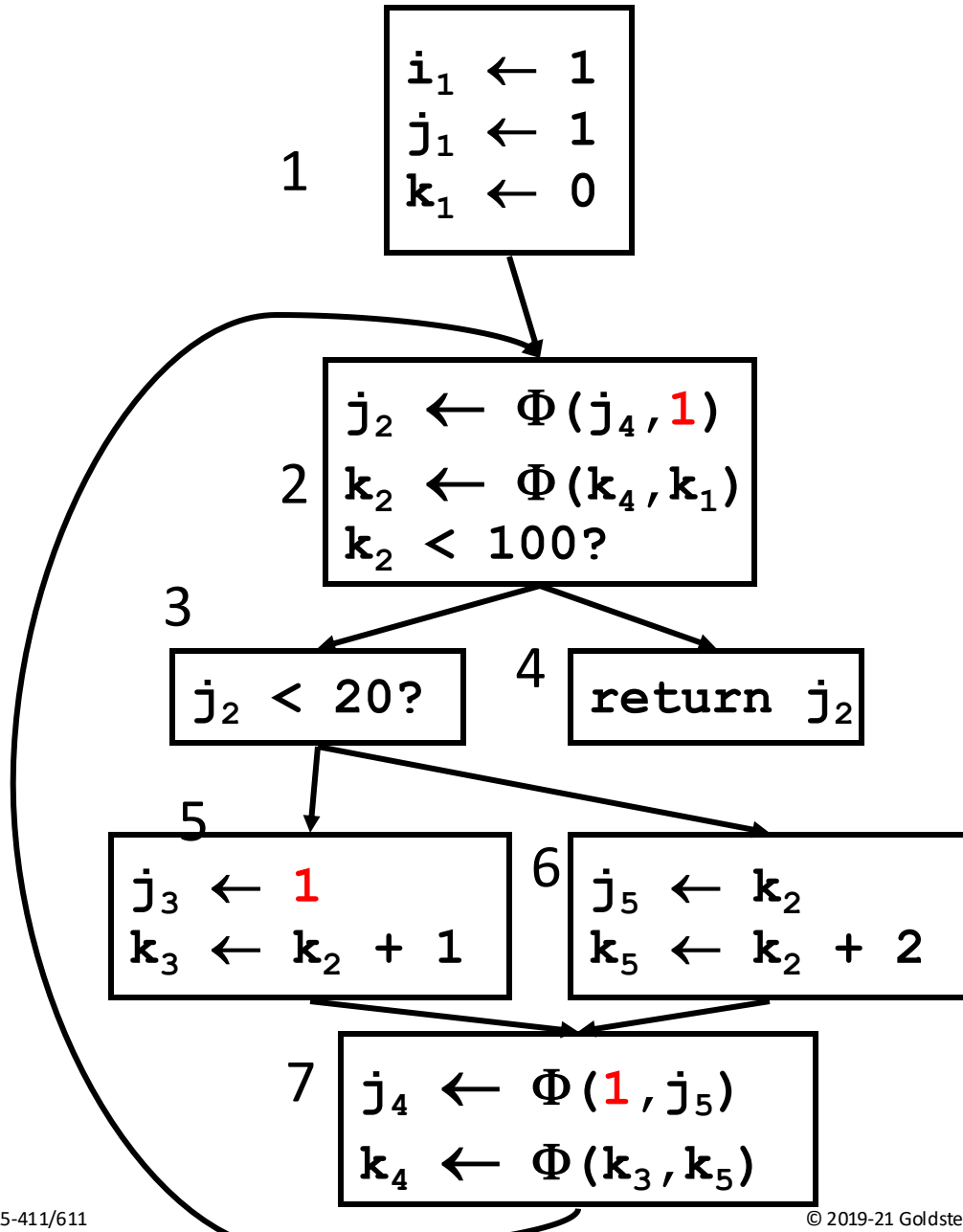
Constant Propagation



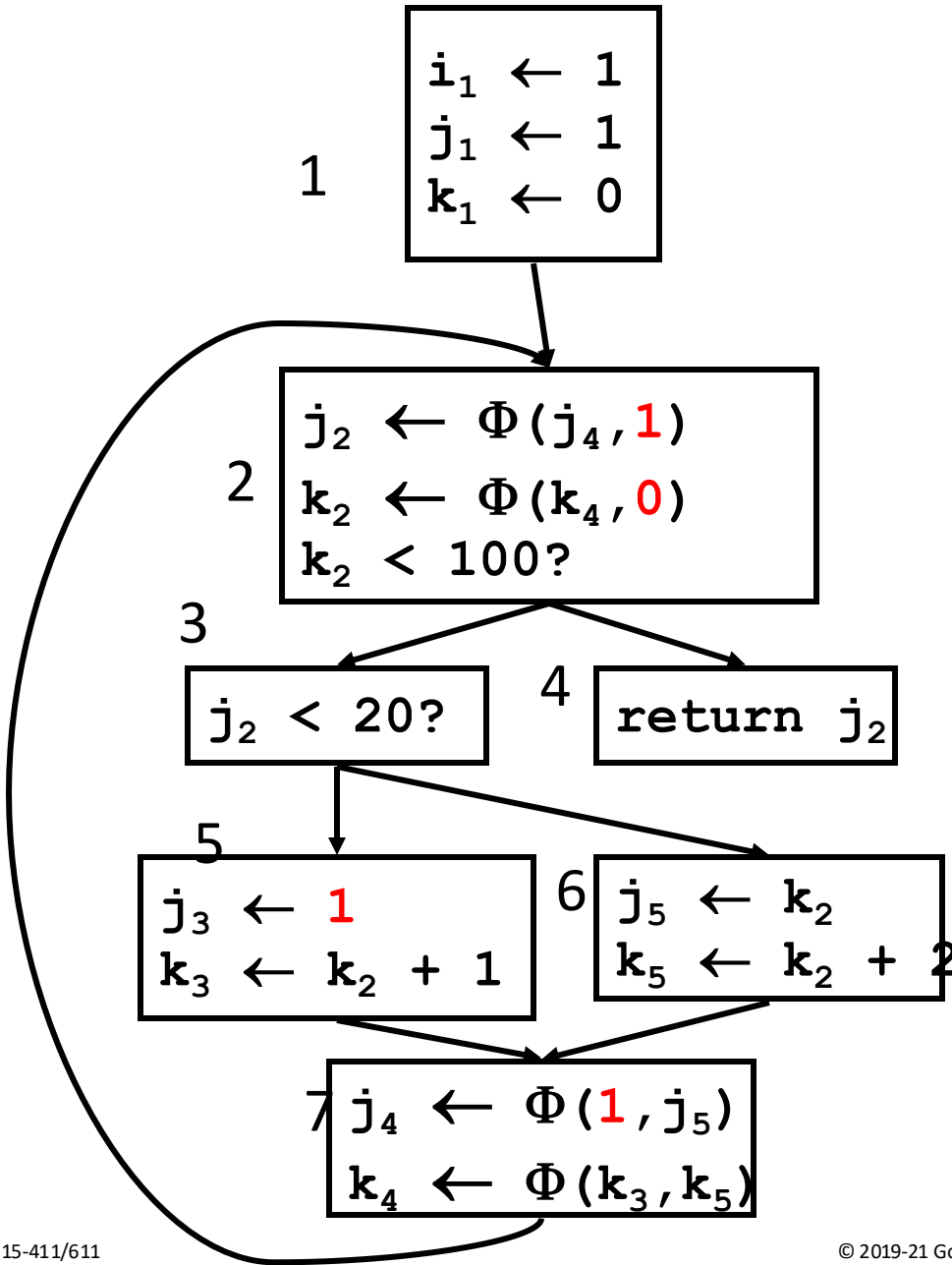
Constant Propagation



Constant Propagation

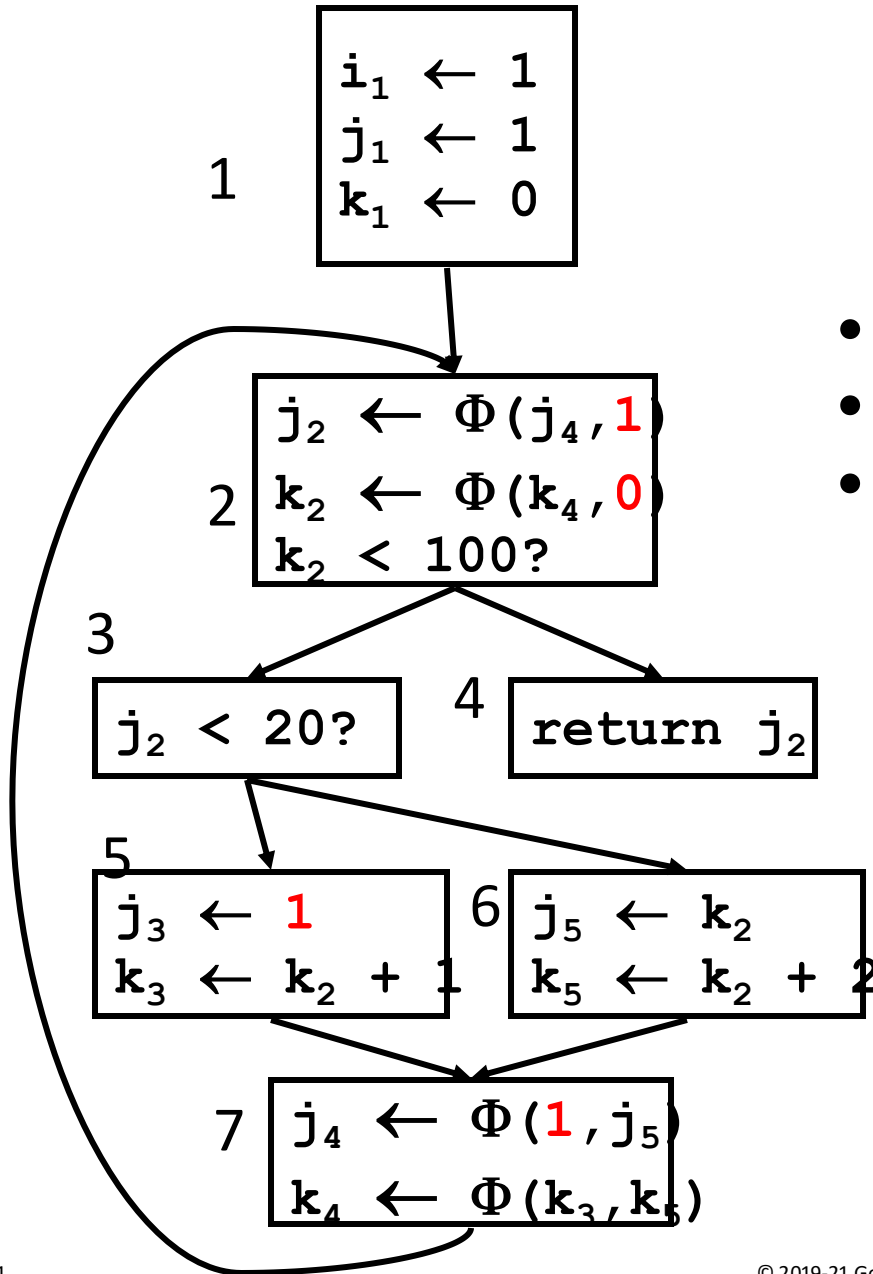


Constant Propagation



But, so what?

Conditional Constant Propagation



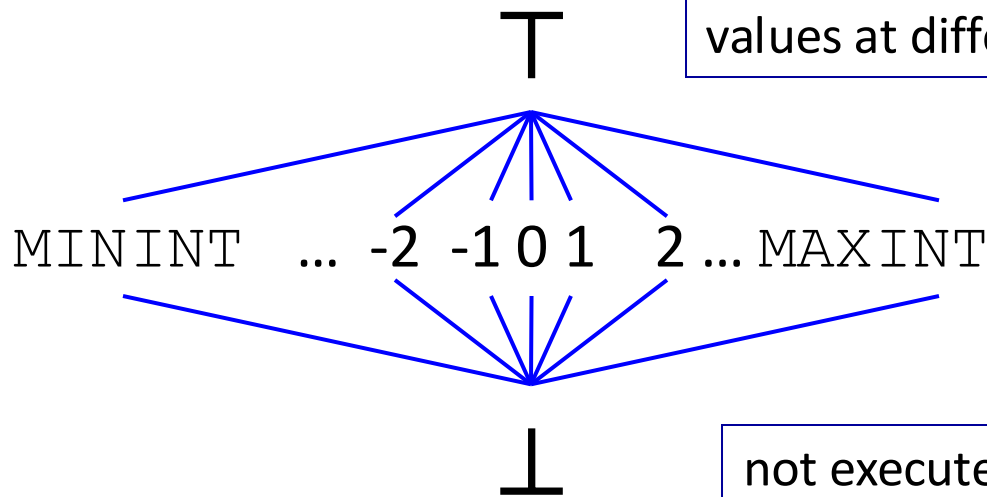
- Does block 6 ever execute?
- Simple CP can't tell
- CCP can tell:
 - Assumes blocks don't execute until proven otherwise
 - Assumes values are constants until proven otherwise

CCP data structures & lattice

Keep track of:

- Blocks (assume unexecuted until proven otherwise)
- Variables (assume not executed, only with proof of assignments of a non-constant value do we assume not constant)

Use a lattice for variables:

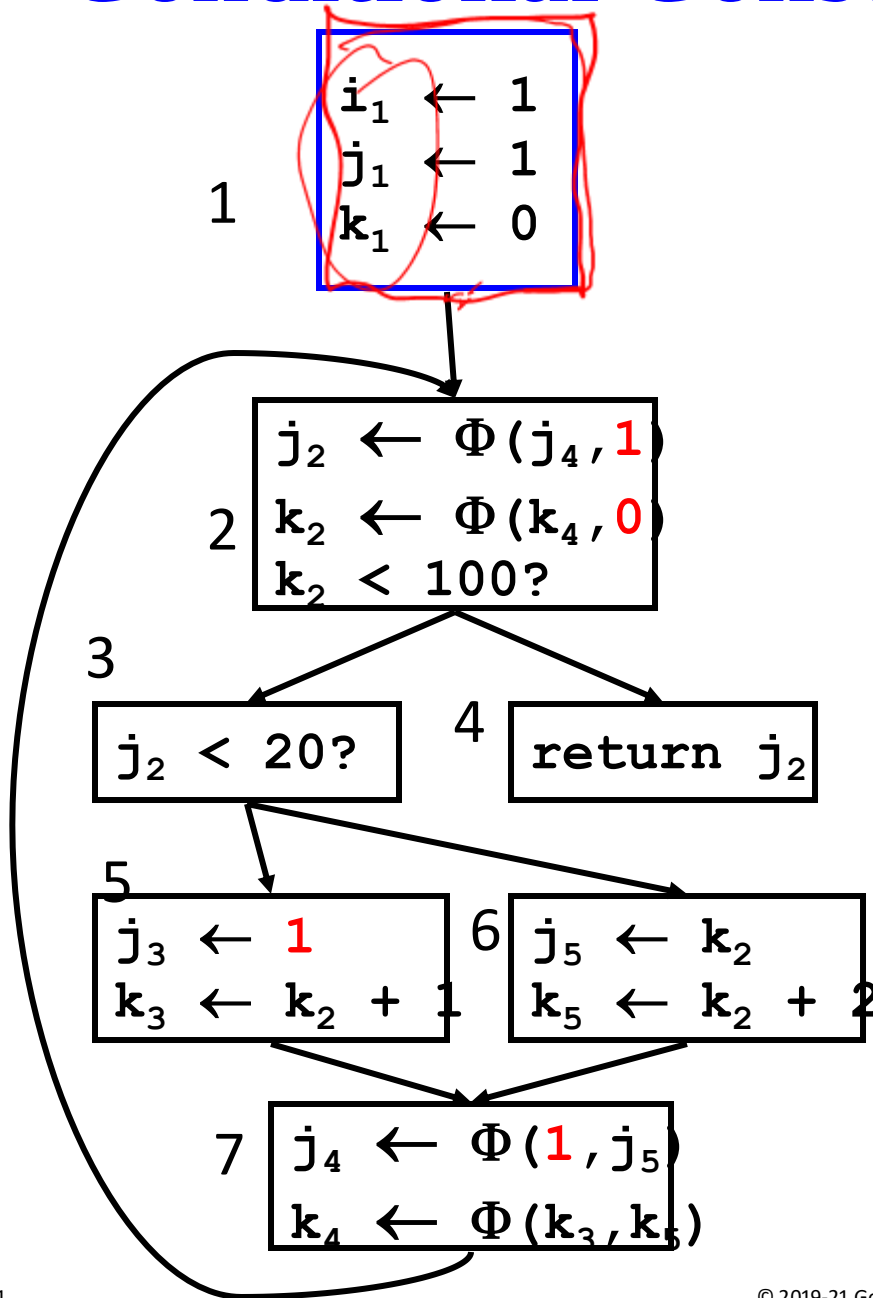


have evidence that variable can hold different values at different times

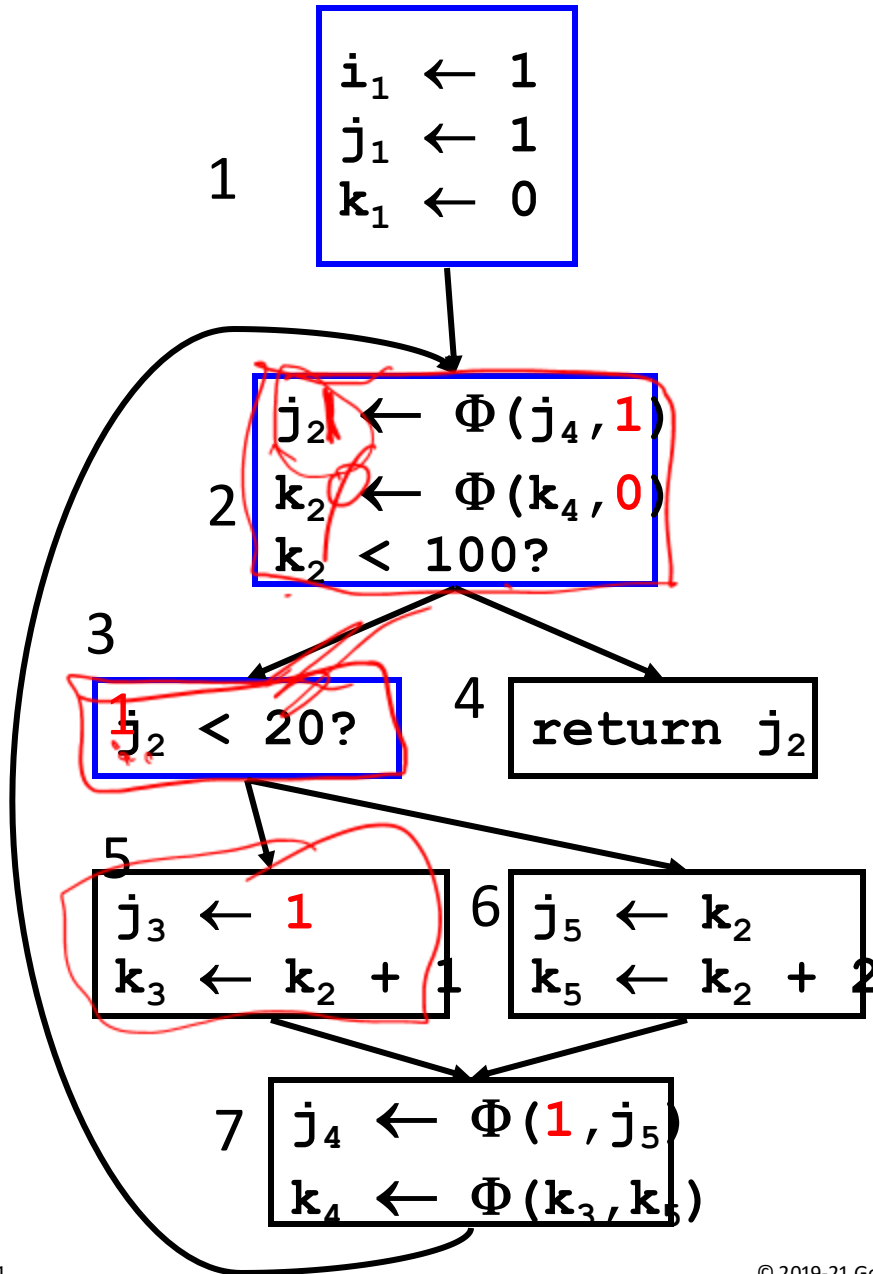
evidence that the var has been assigned a constant

not executed

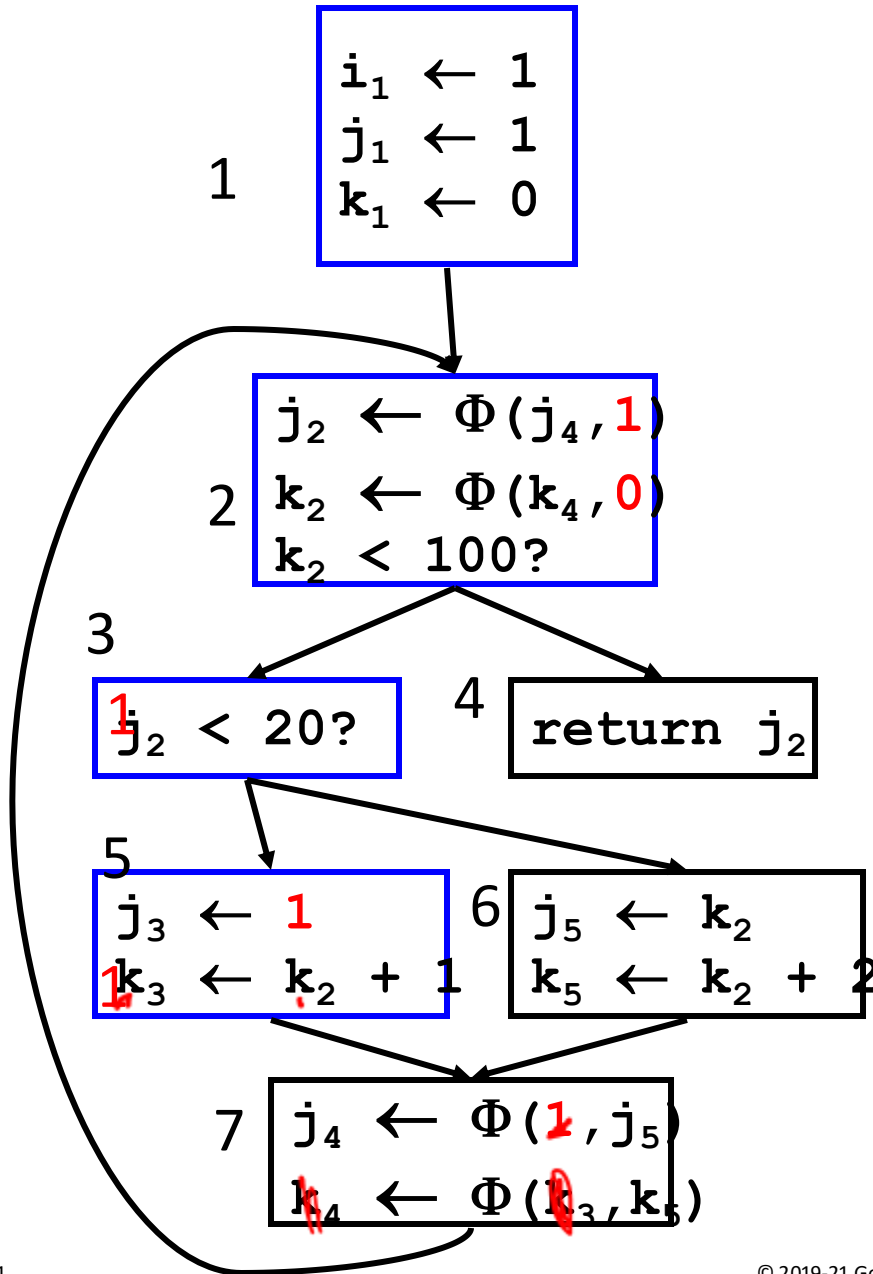
Conditional Constant Propagation



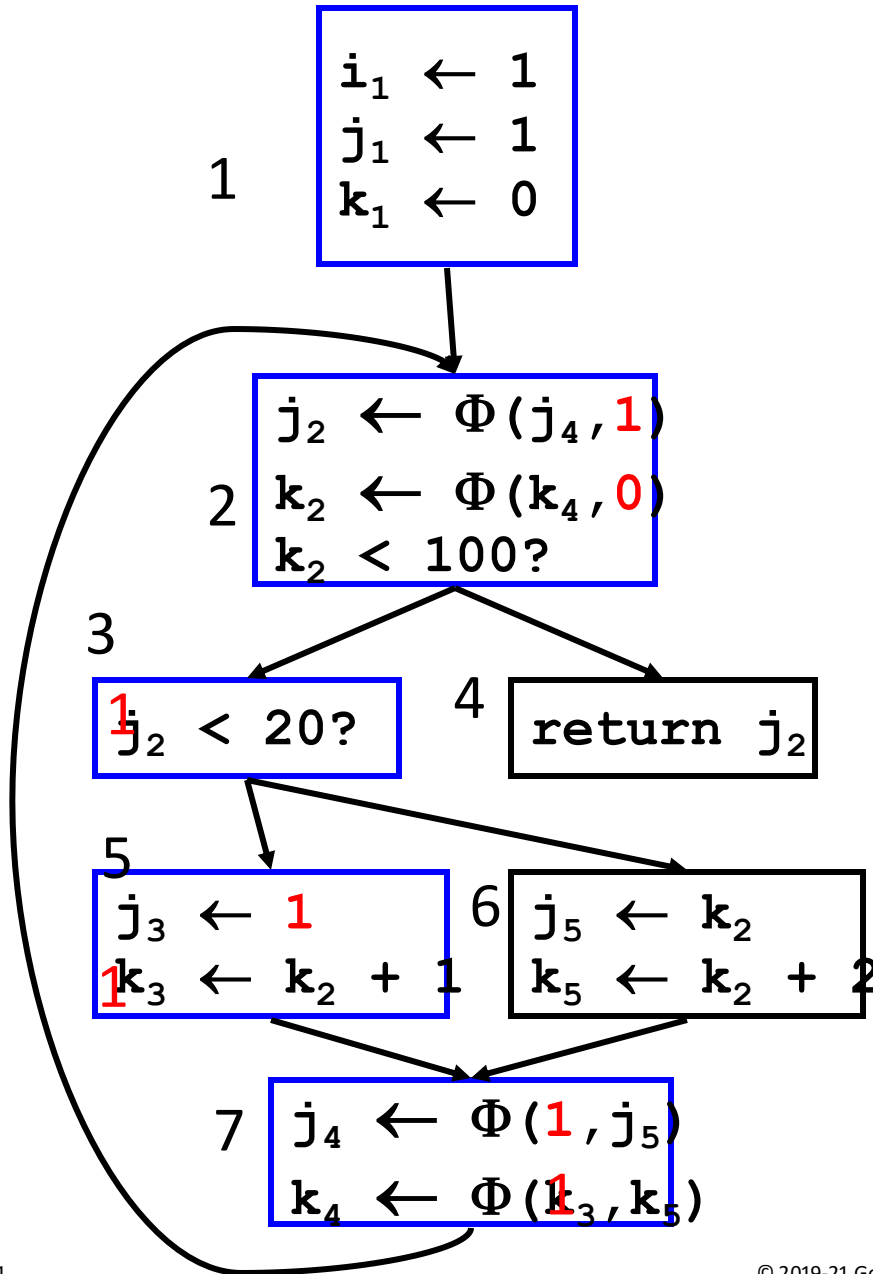
Conditional Constant Propagation



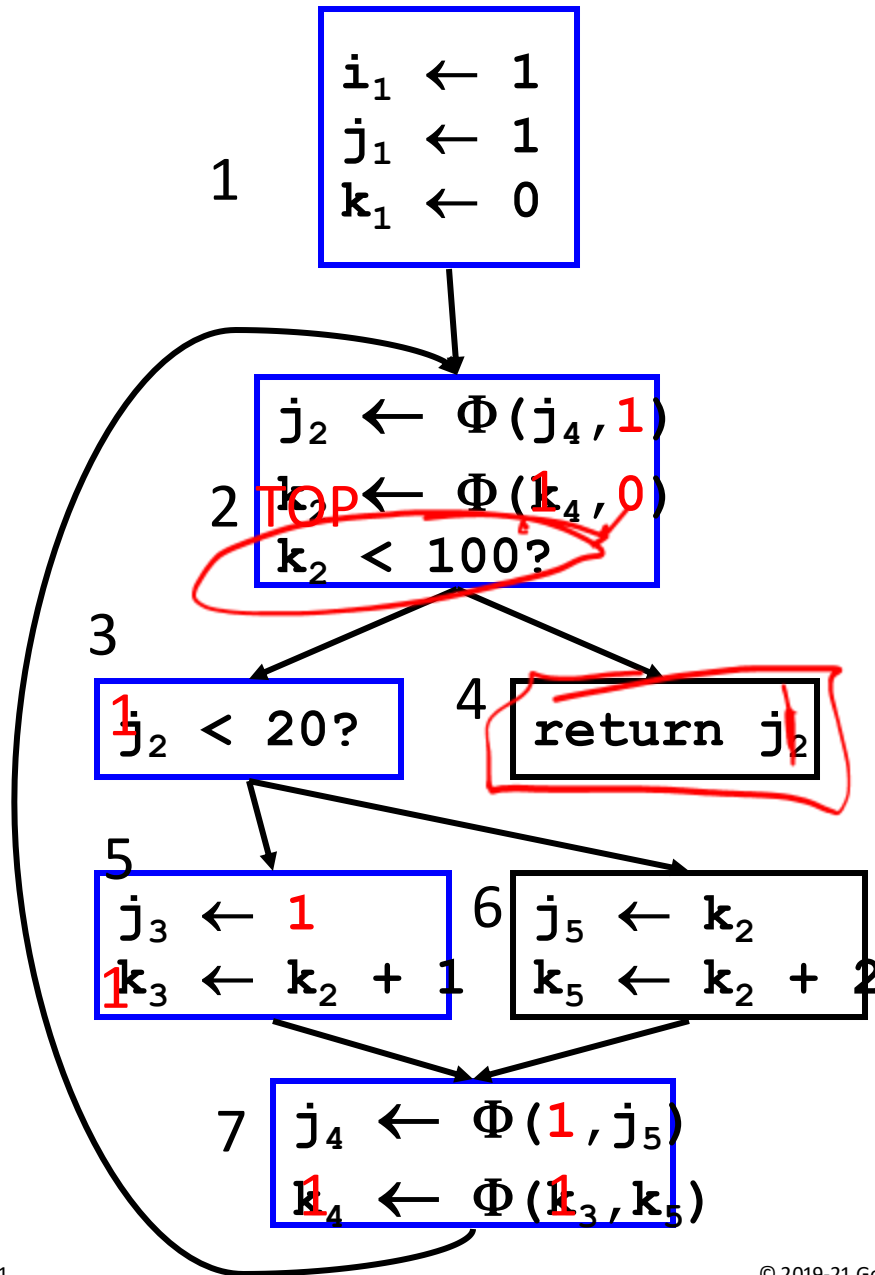
Conditional Constant Propagation



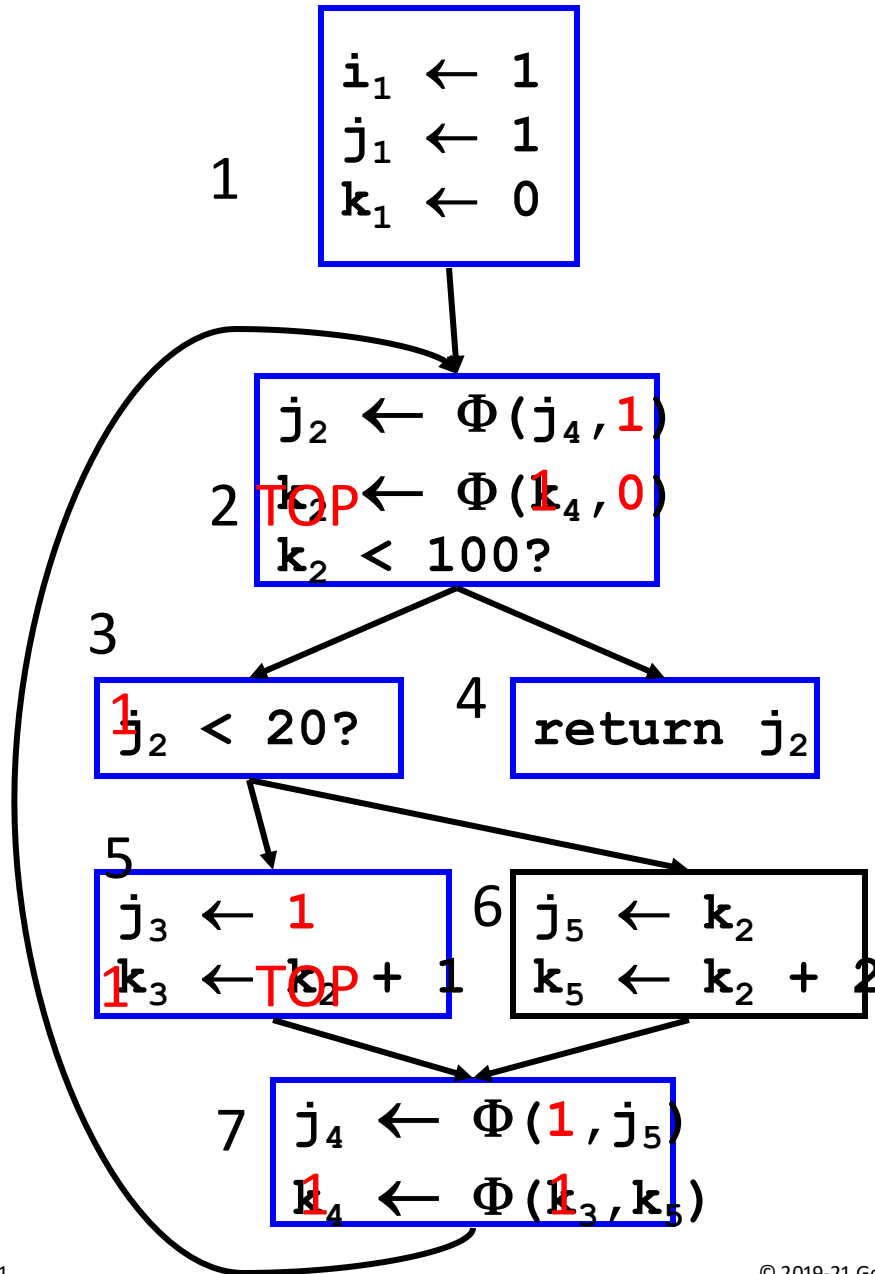
Conditional Constant Propagation



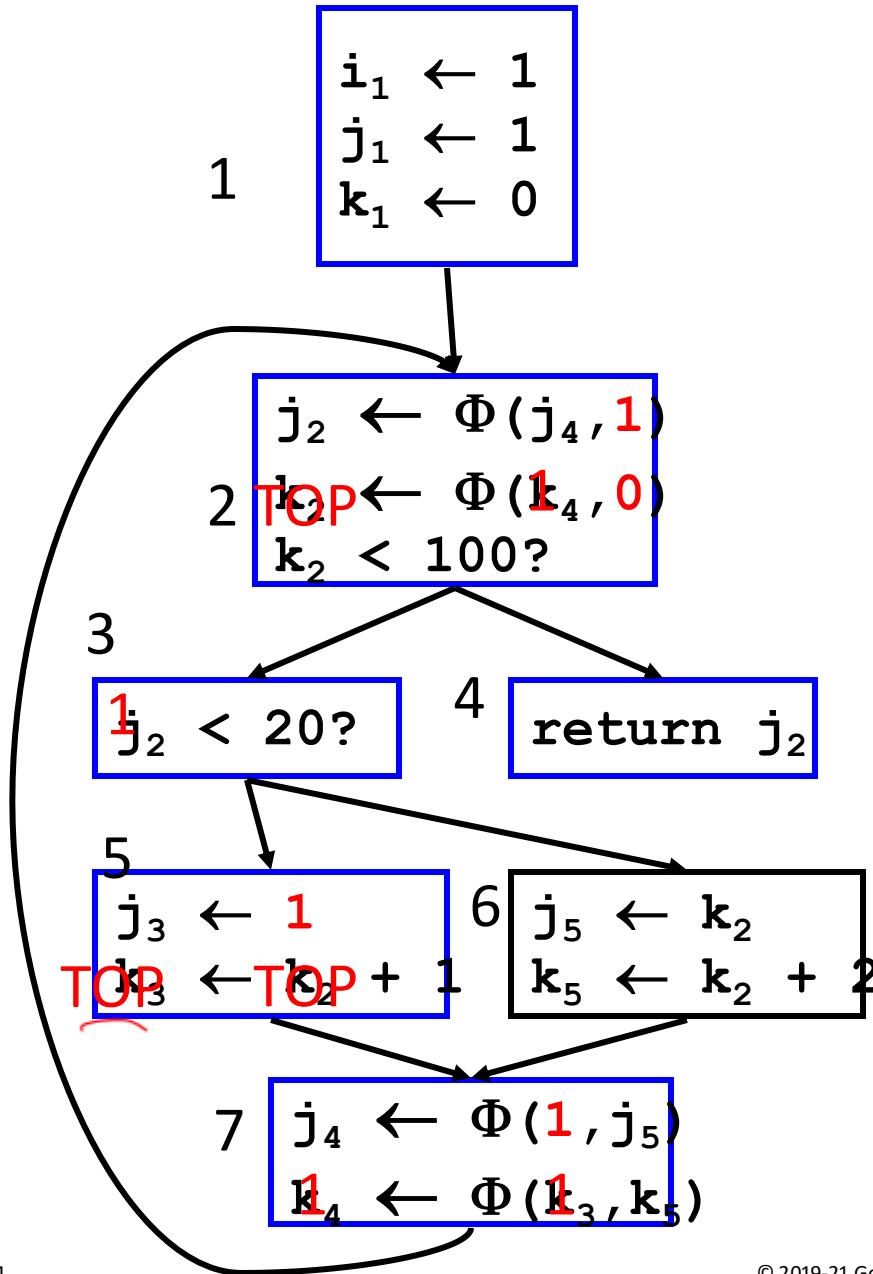
Conditional Constant Propagation



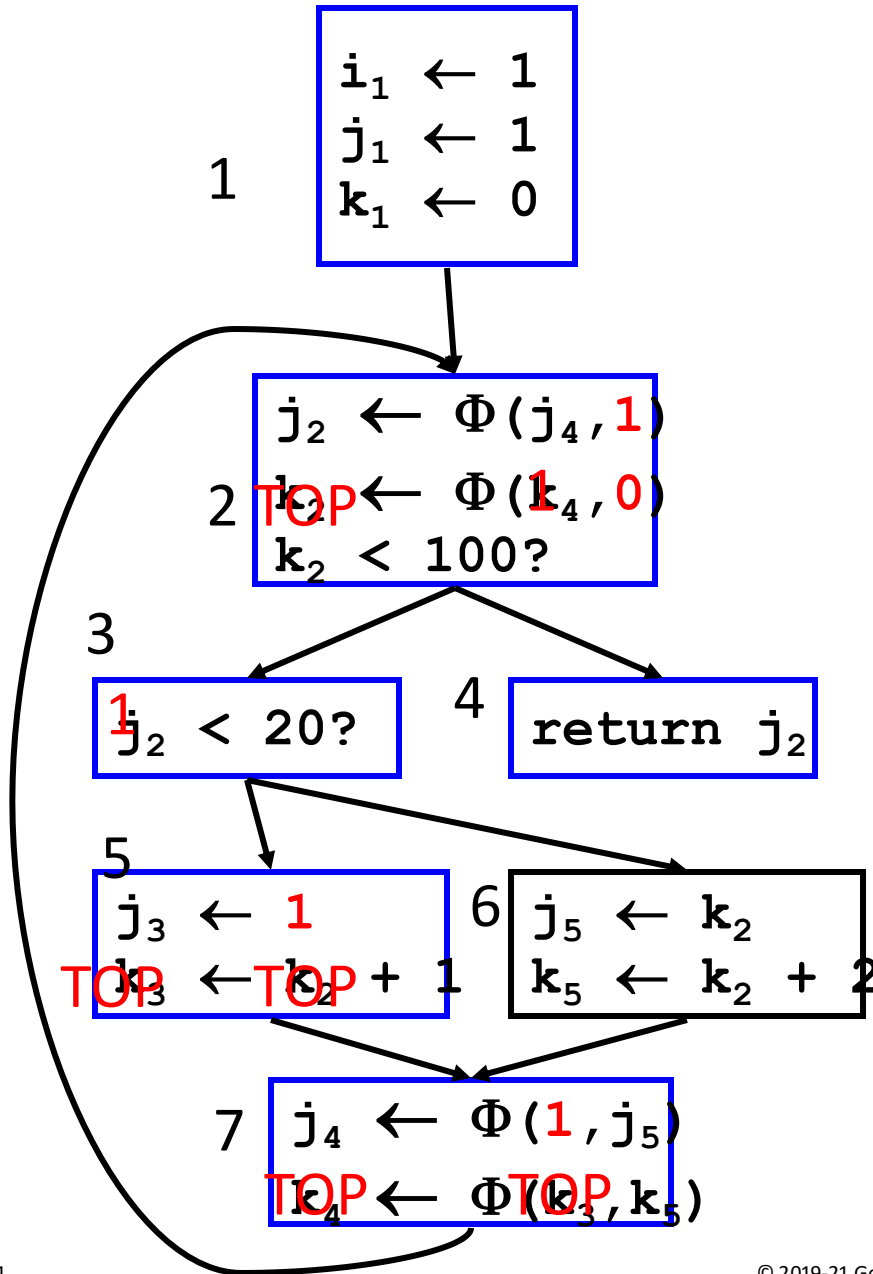
Conditional Constant Propagation



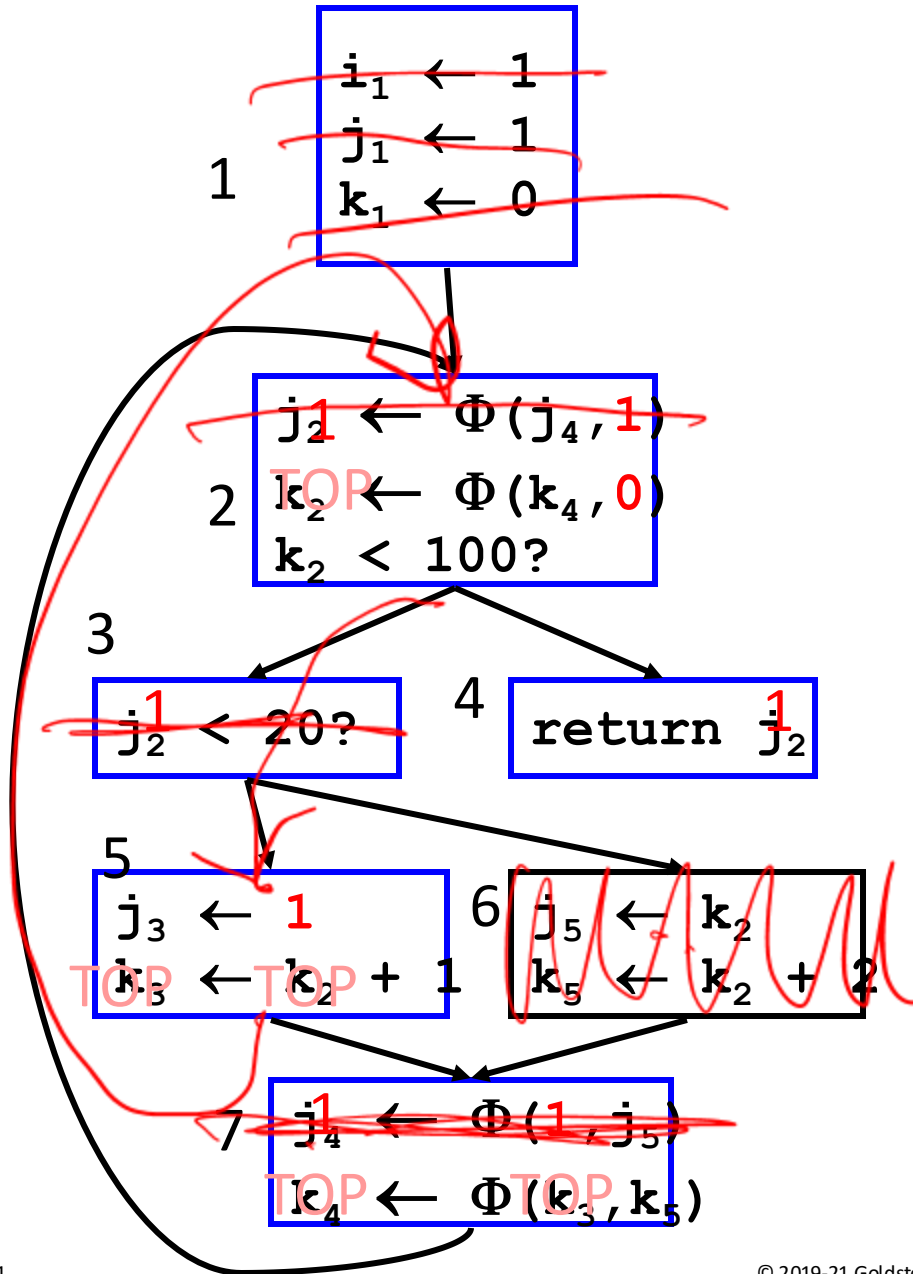
Conditional Constant Propagation



Conditional Constant Propagation

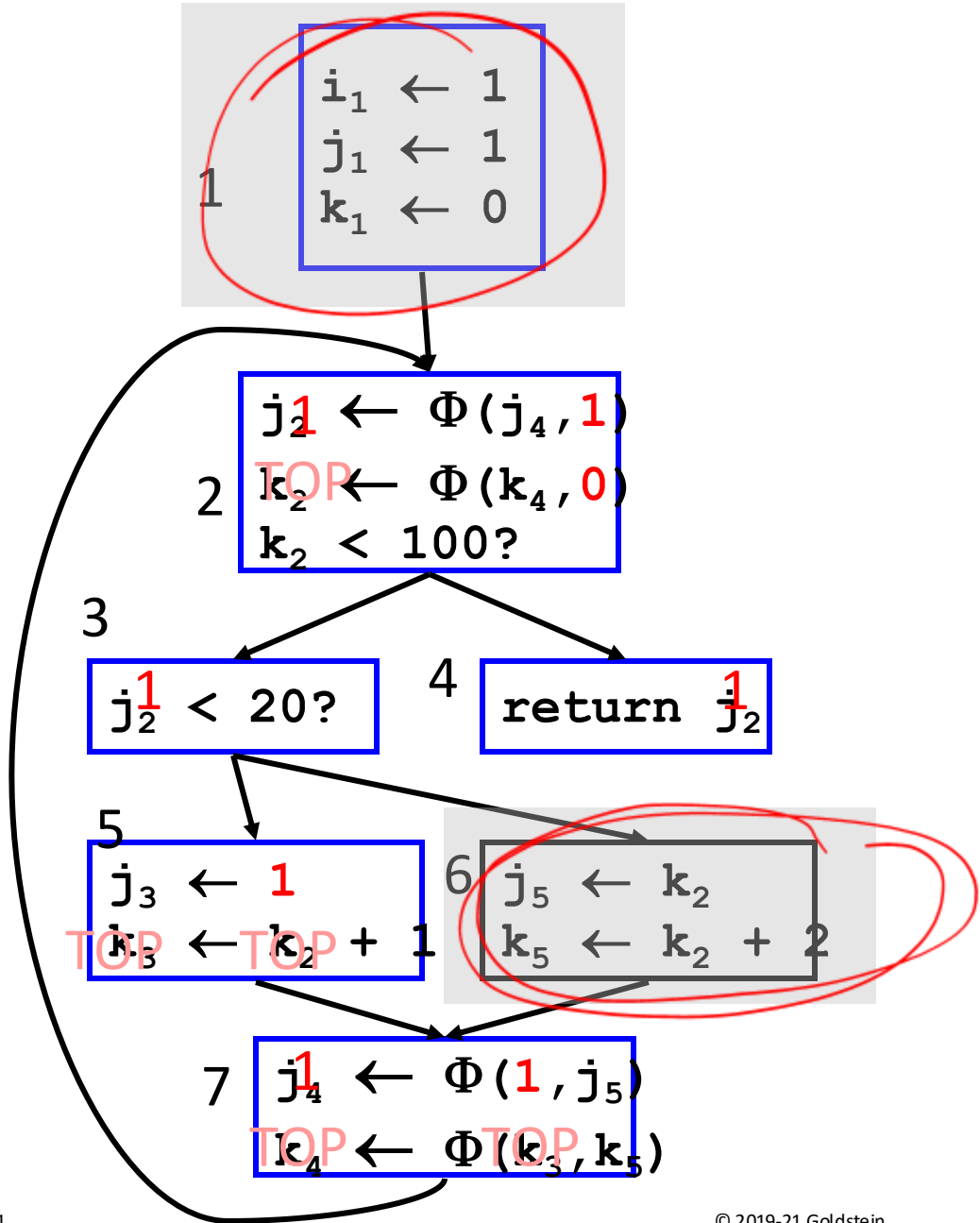


Conditional Constant Propagation



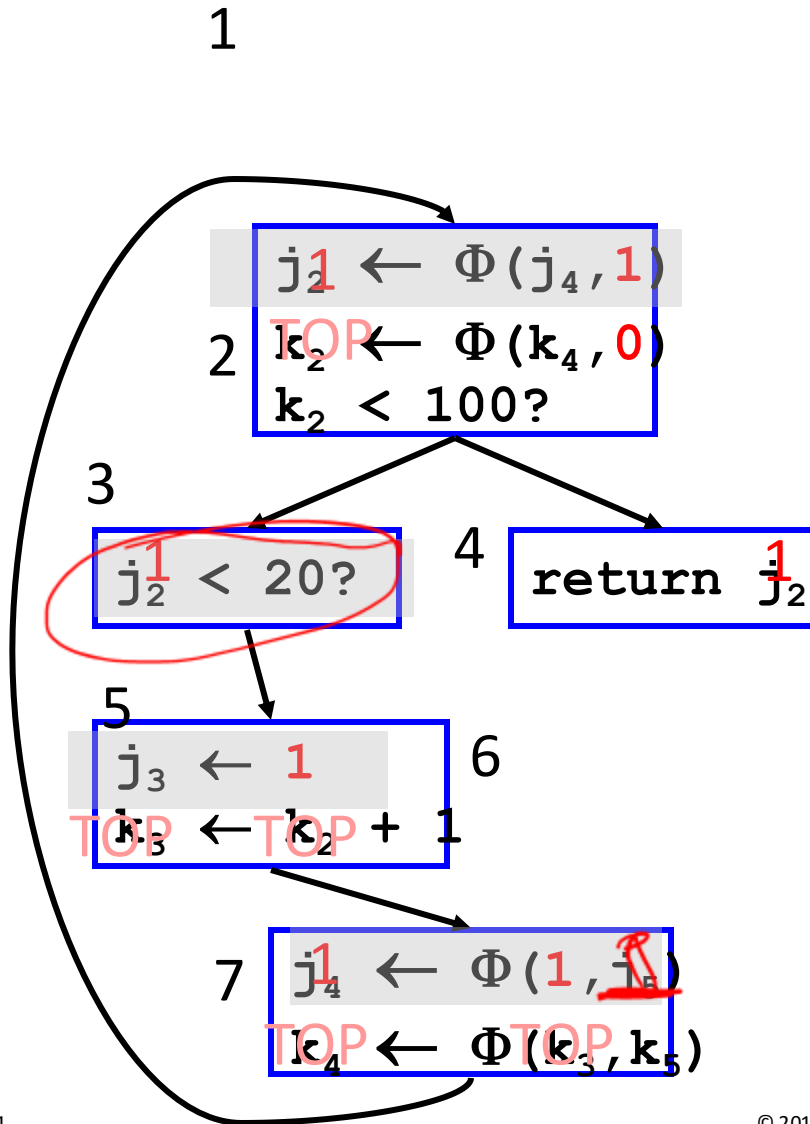
Now What?

Conditional Constant Propagation



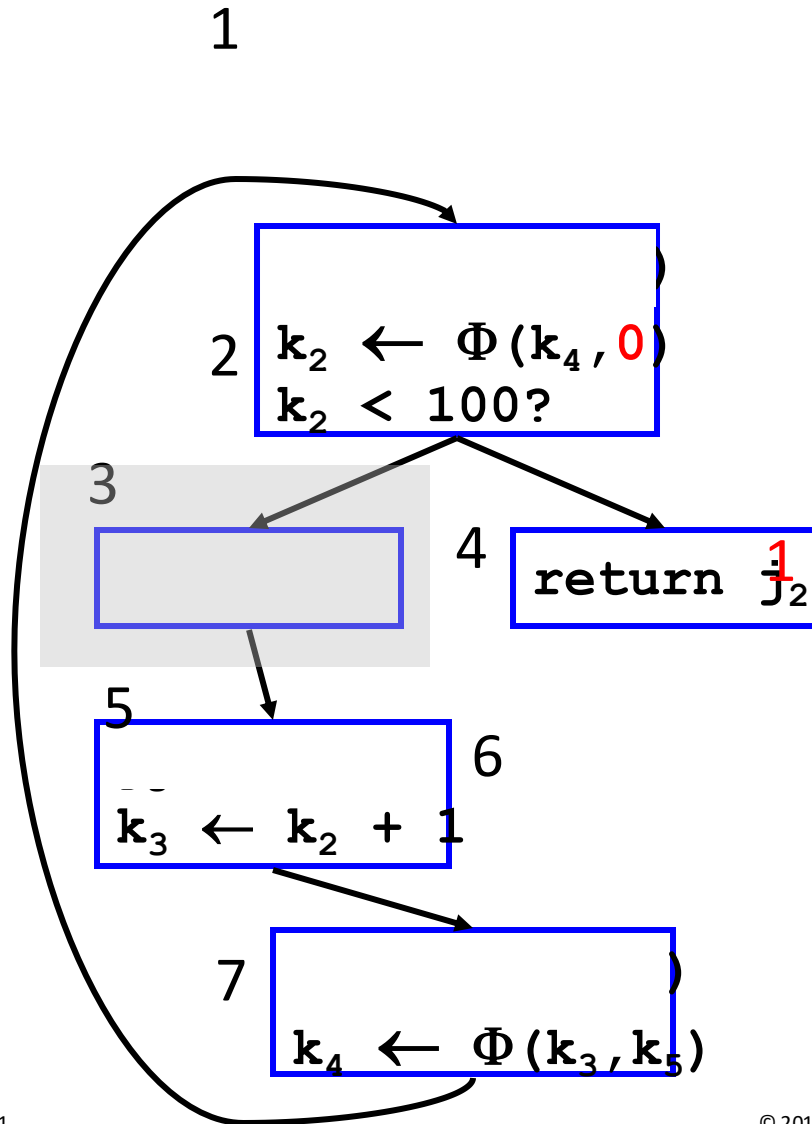
Now What?

Conditional Constant Propagation



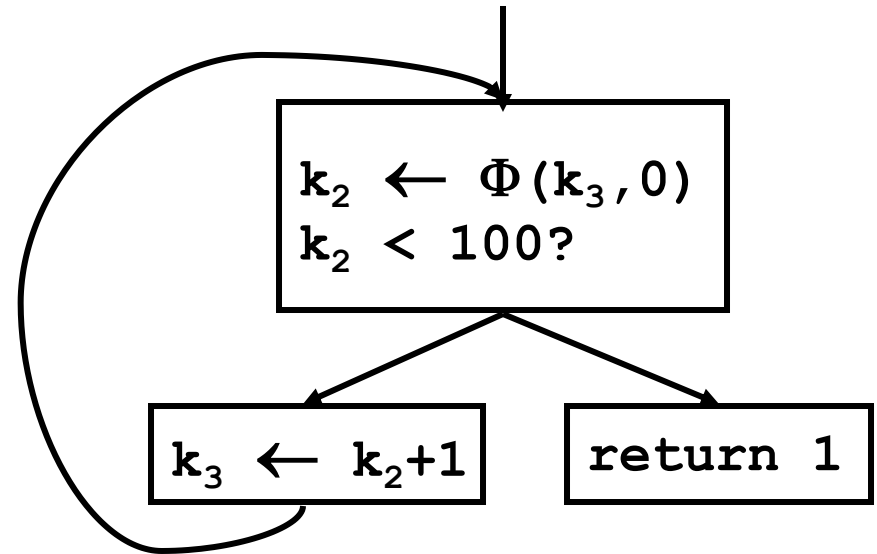
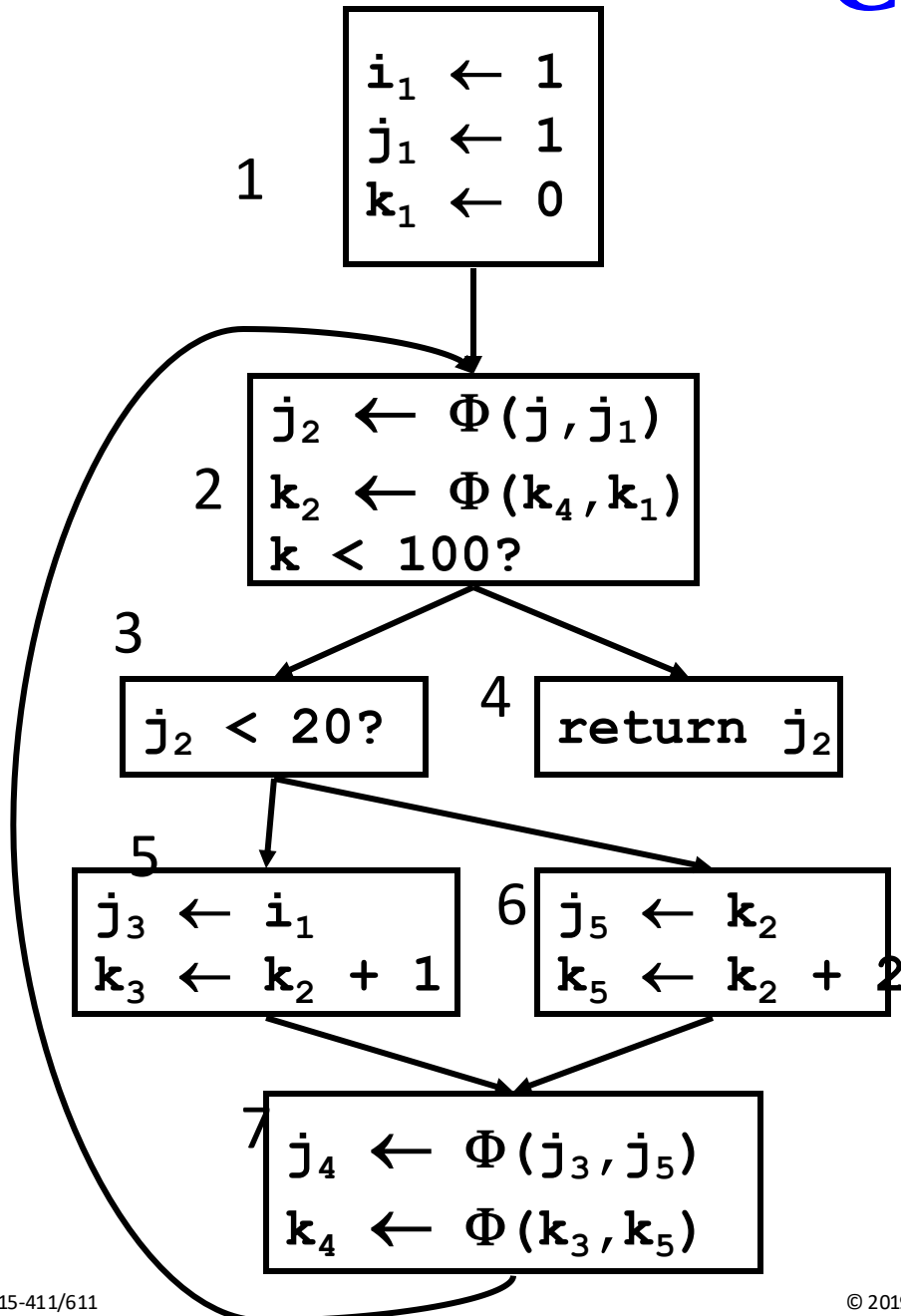
Now What?

Conditional Constant Propagation



Now What?

CCP



Just a taste of the kinds of optimizations that SSA enables.

ADCE

Dead Code Elimination

```
W <- list of all defs
```

```
while !W.isEmpty {
```

```
  Stmt S <- W.removeOne
```

```
  if |S.users| != 0 then continue
```

```
  if S.hasSideEffects() then continue
```

```
  foreach def in S.definers {
```

```
    def.users <- def.users - {S}
```

```
    if |def.users| == 0 then
```

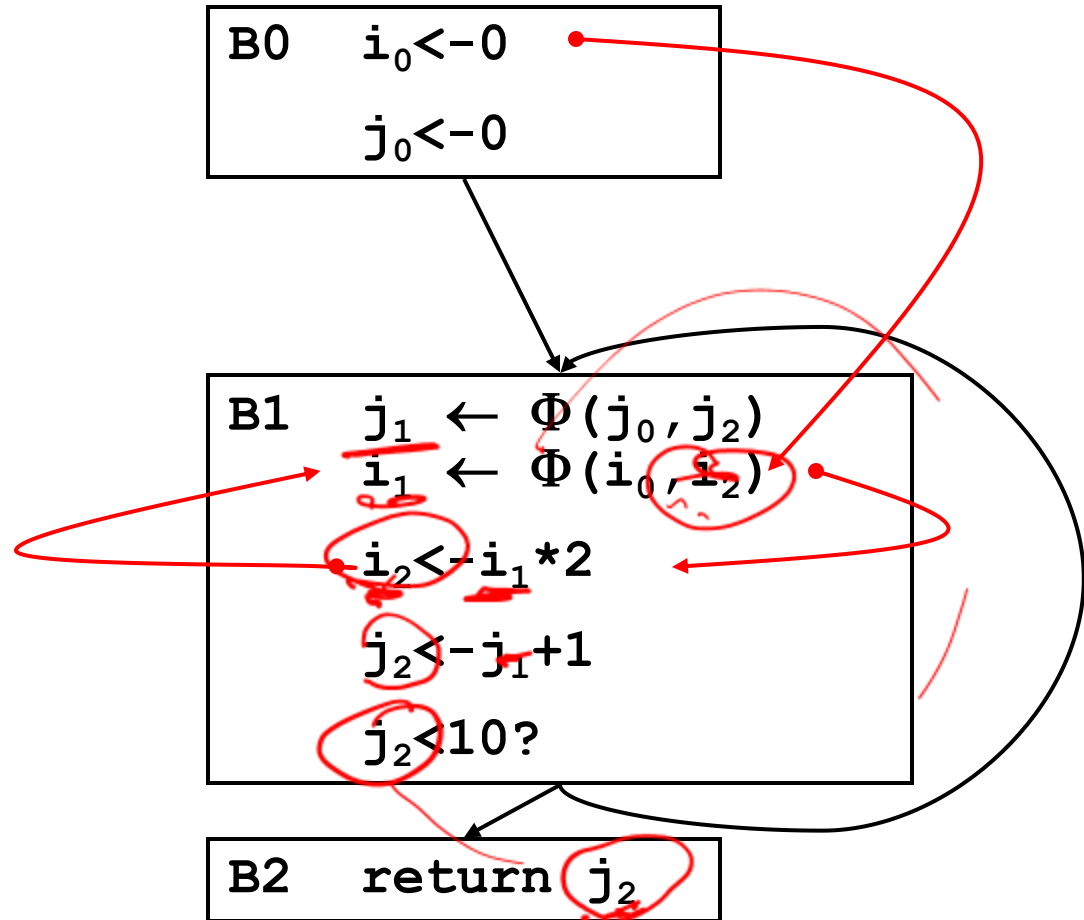
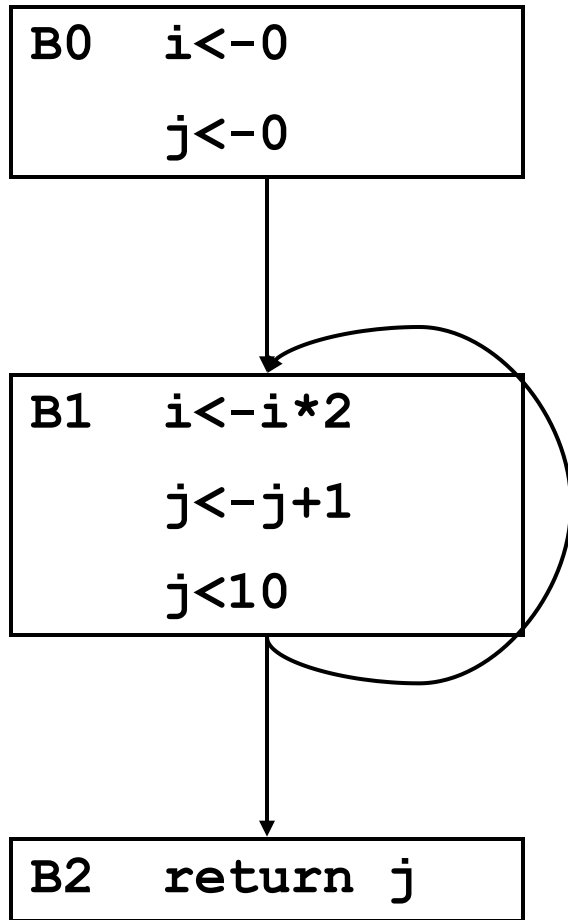
```
      W <- W UNION {def}
```

```
  }
```

```
}
```

Since we are using SSA, this
is just a list of all variable
assignments.

Example DCE



Standard DCE leaves Zombies!

Aggressive Dead Code Elimination

Assume a stmt is dead until proven otherwise.

init:

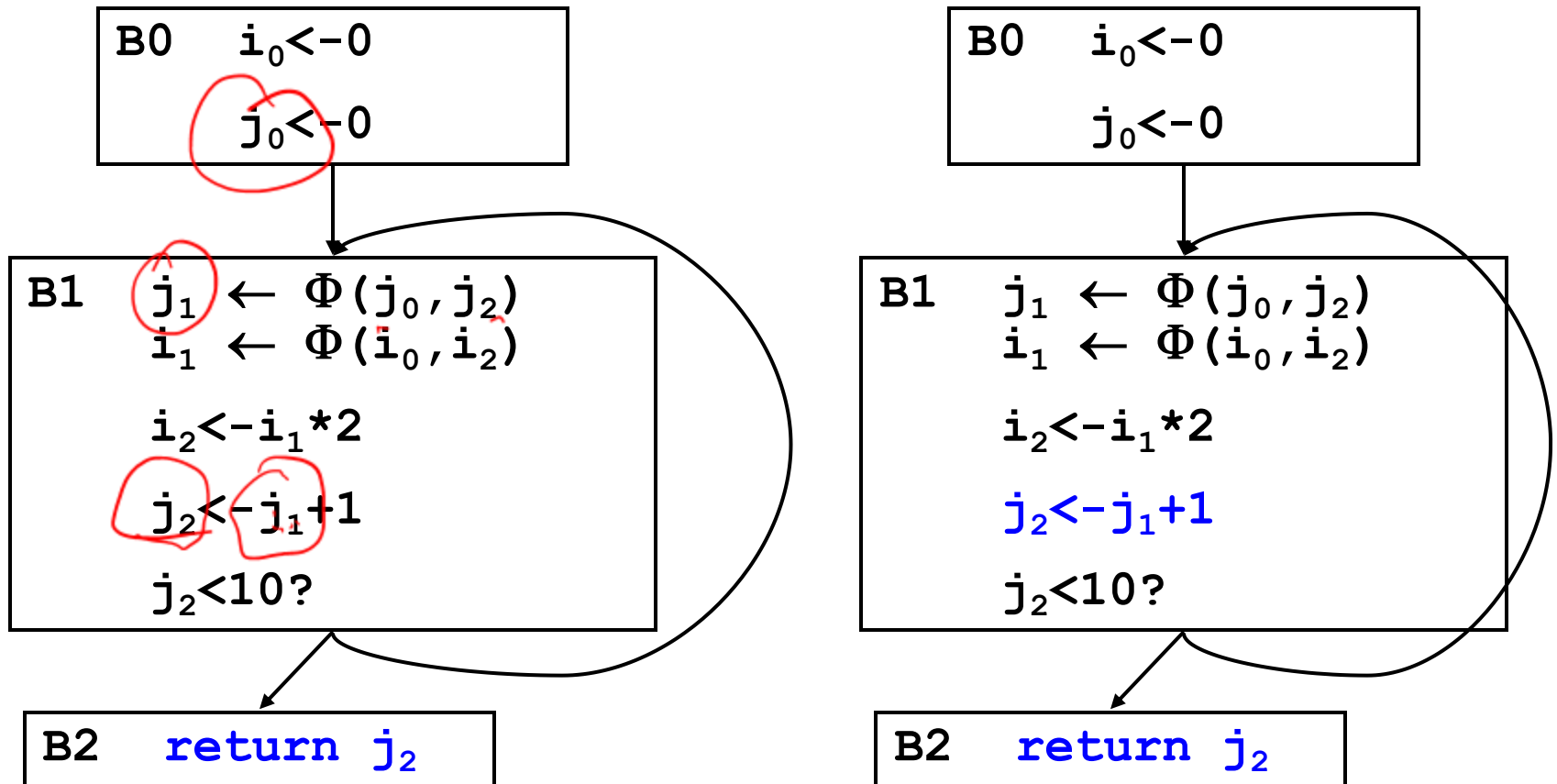
mark as live all stmts that have side-effects:

- I/O
- stores into memory
- returns
- calls a function that MIGHT have side-effects

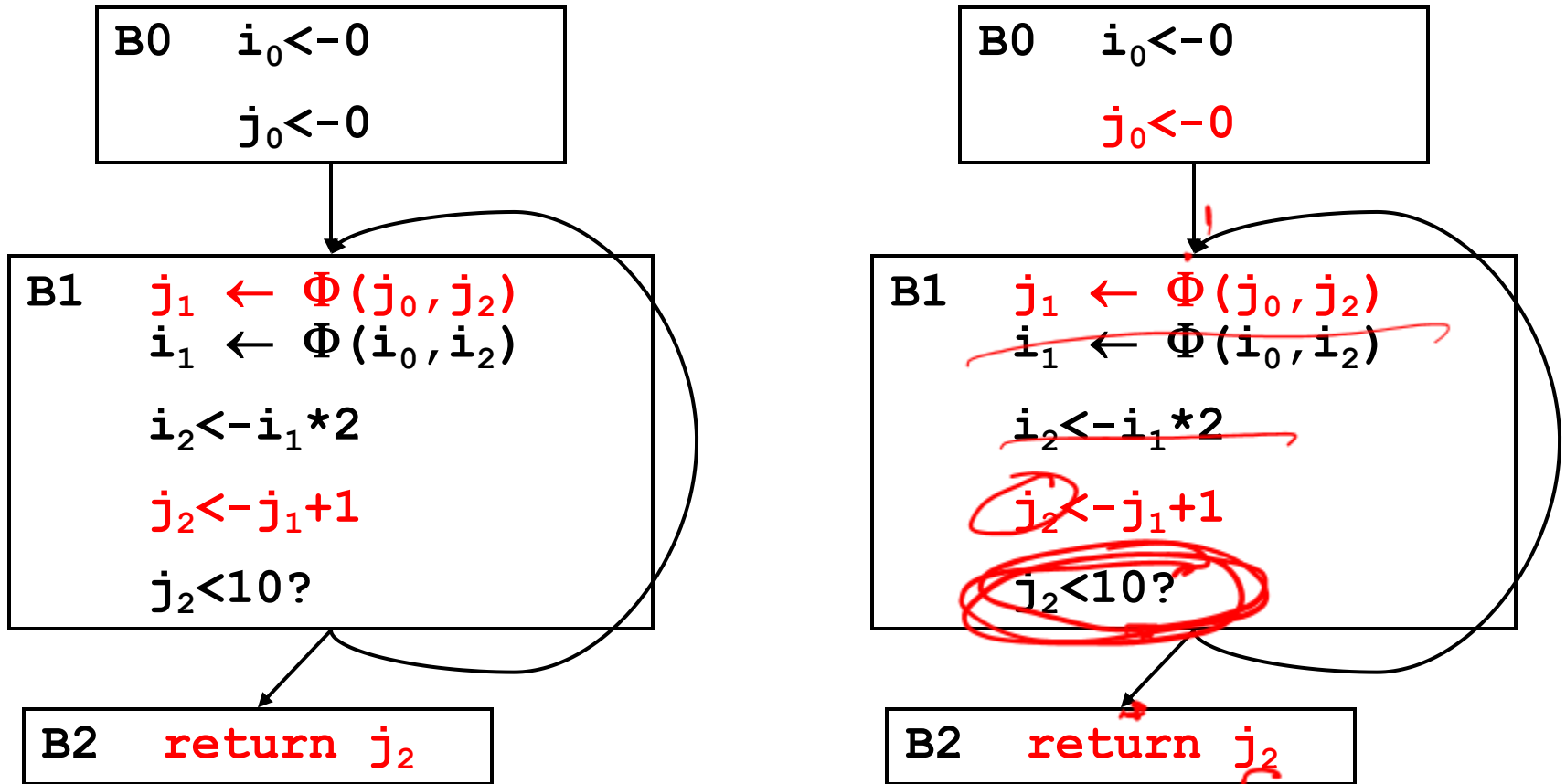
As we mark S live, insert S.defs into W

```
while (|W| > 0) {  
  S <- W.removeOne()  
  if (S is live) continue;  
  mark S live, insert S.defs into W  
}
```


Example DCE



Example DCE

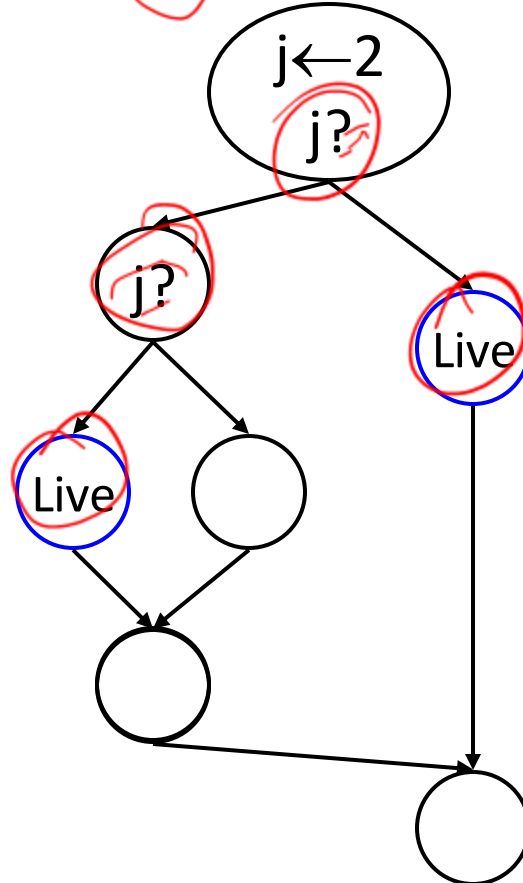


Problem!

Fixing ADCE

- If S is live, then

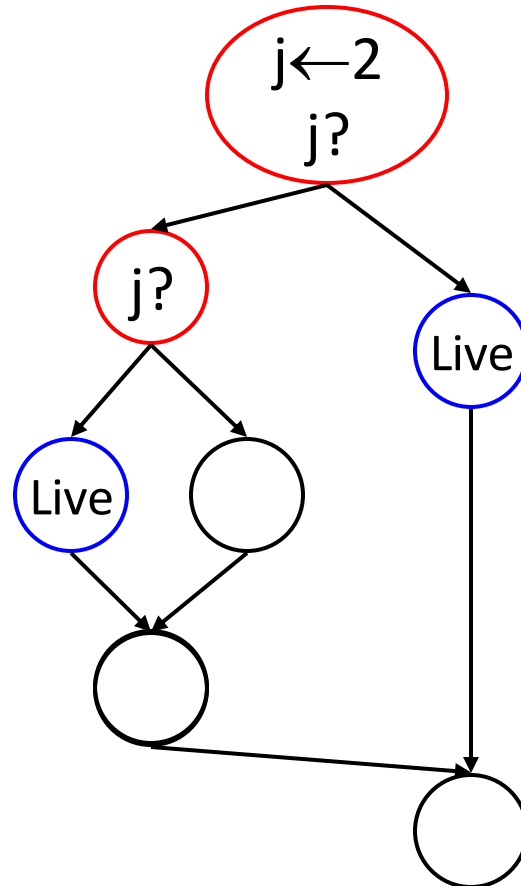
If T determines if S can execute, T should be live



Fixing DCE

- If S is live, then

If T determines if S can execute, T should be live

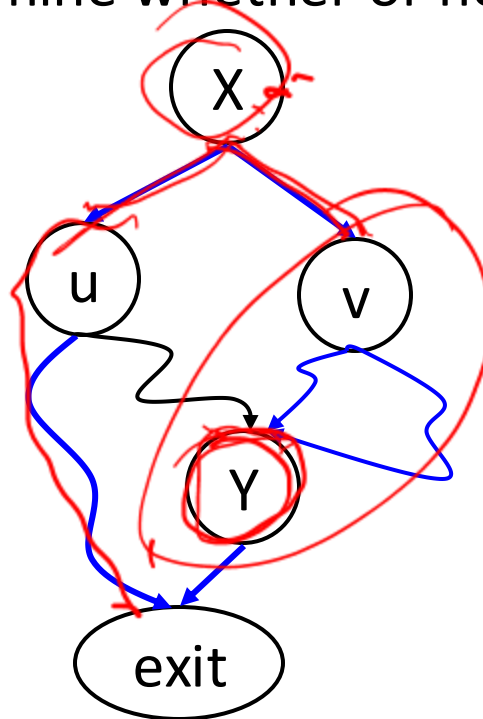


Control Dependence

Y is control-dependent on X if

- X branches to u and v
- $\exists$ a path $u \rightarrow \text{exit}$ which does not go through Y
- $\forall$ paths $v \rightarrow \text{exit}$ go through Y

IOW, X can determine whether or not Y is executed.

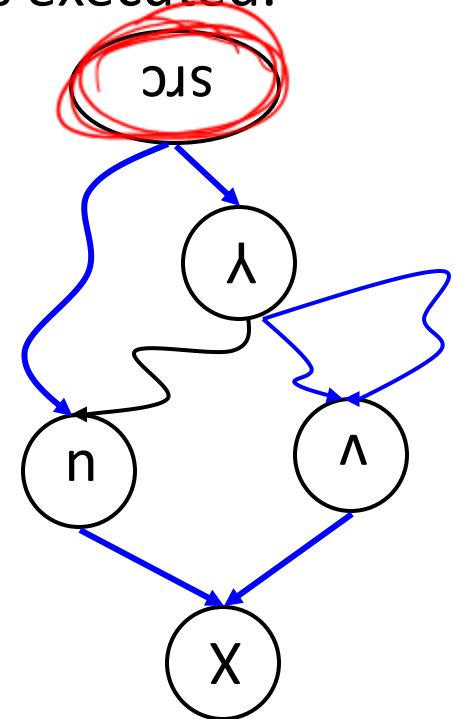
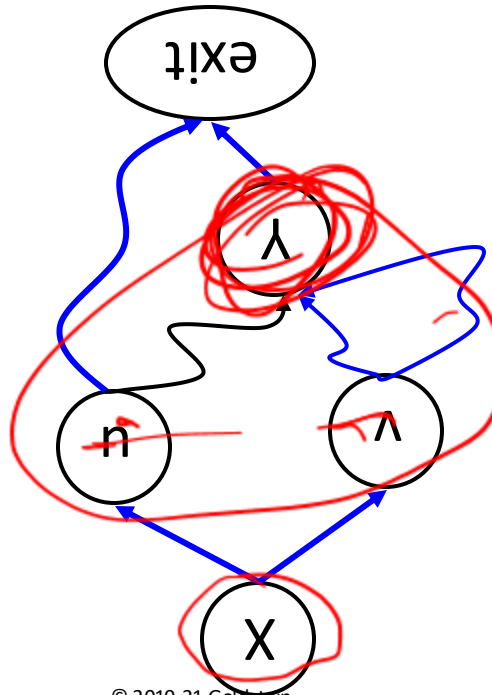
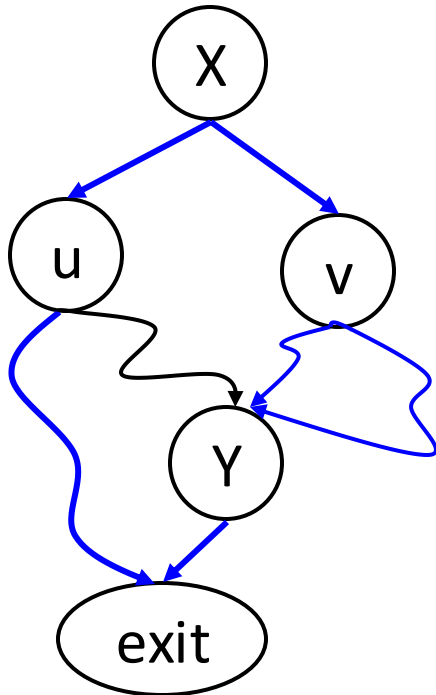


Finding the CDG

Y is control-dependent on X if

- X branches to u and v
- $\exists$ a path $u \rightarrow \text{exit}$ which does not go through Y
- $\forall$ paths $v \rightarrow \text{exit}$ go through Y

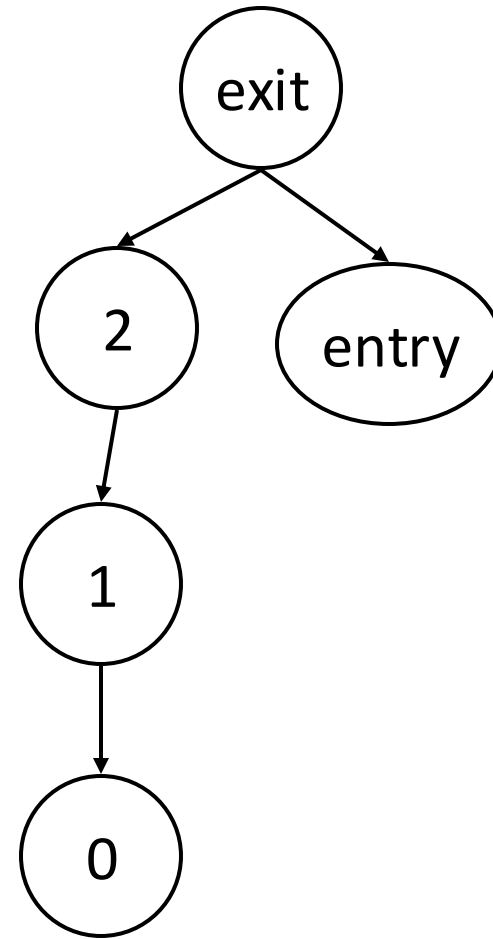
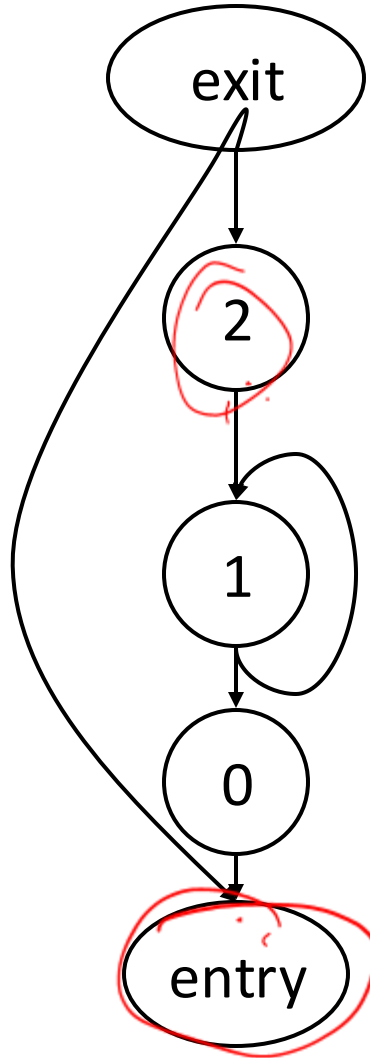
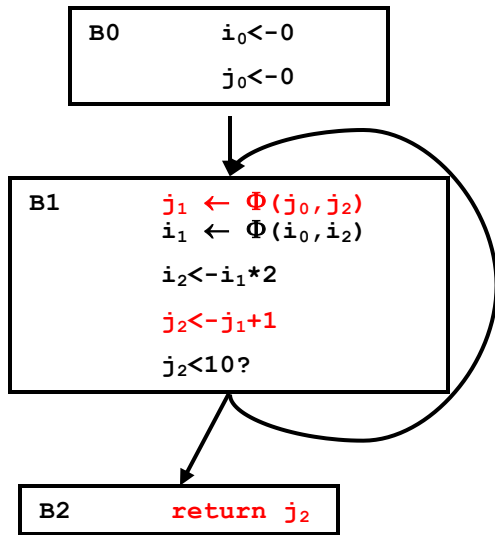
IOW, X can determine whether or not Y is executed.



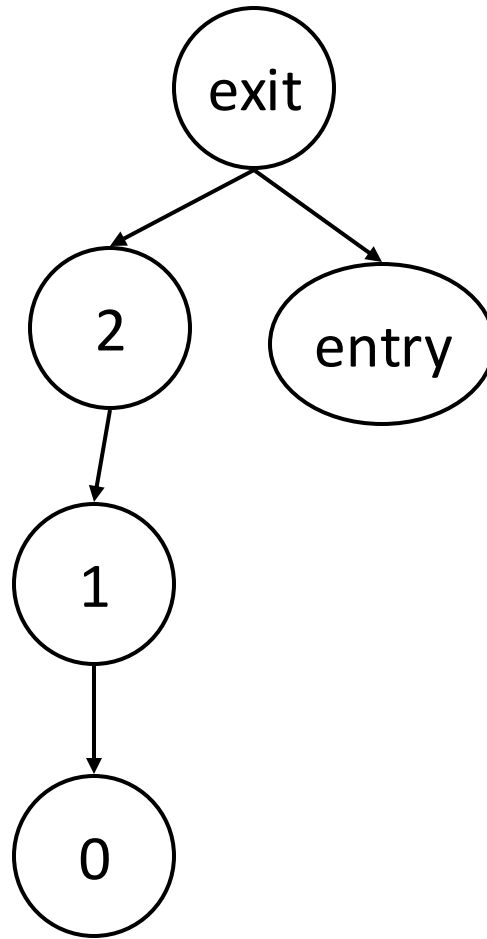
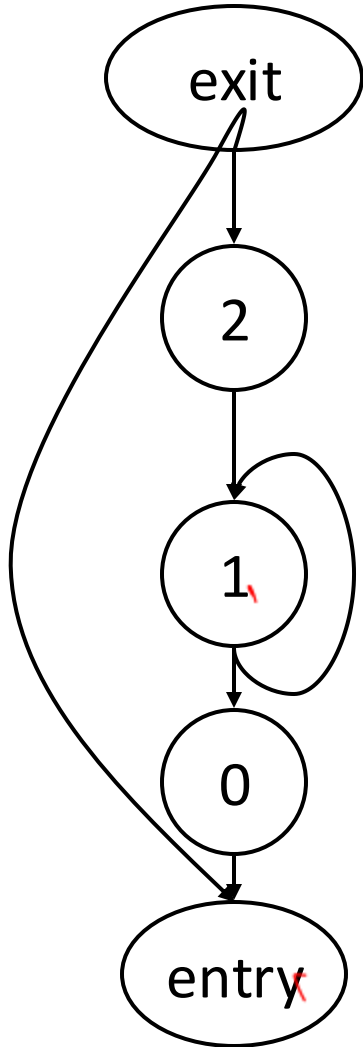
Finding the CDG

- Construct CFG
- Add entry node and exit node
- Add (entry, exit)
- Create G' , the reverse CFG
- Compute D-tree in G' (post-dominators of G)
- Compute $DF_{G'}(y)$ for all $y \in G'$ (post-DF of G)
- Add $(x, y) \in G$ to CDG if $x \in DF_{G'}(y)$

CDG of example



CDG of example



DF

exit:	{}
2:	{entry}
1:	{1, entry}
0:	{entry}
entry:	{}

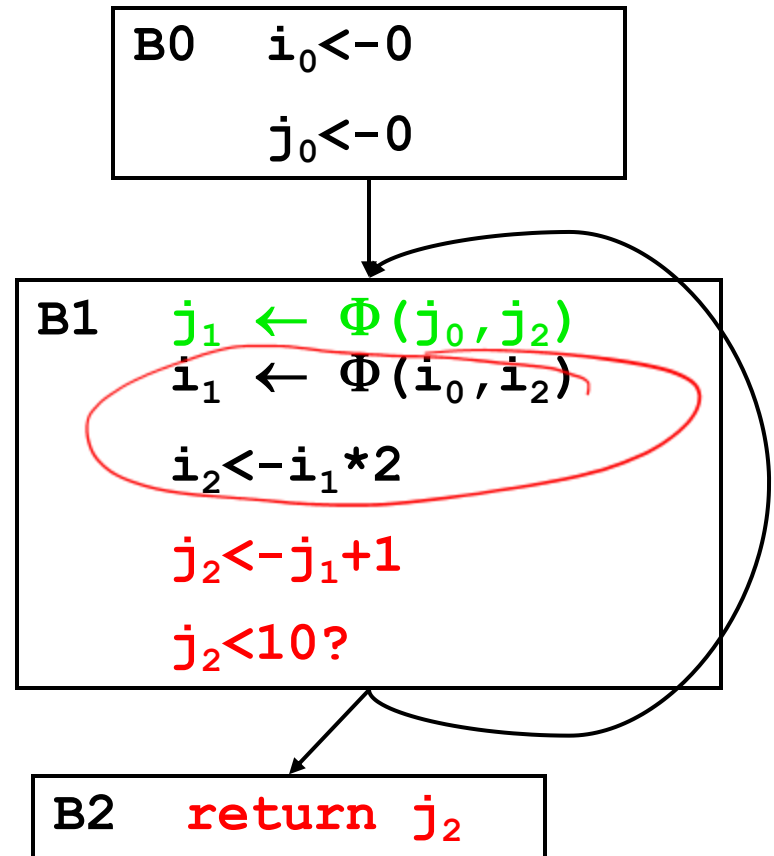
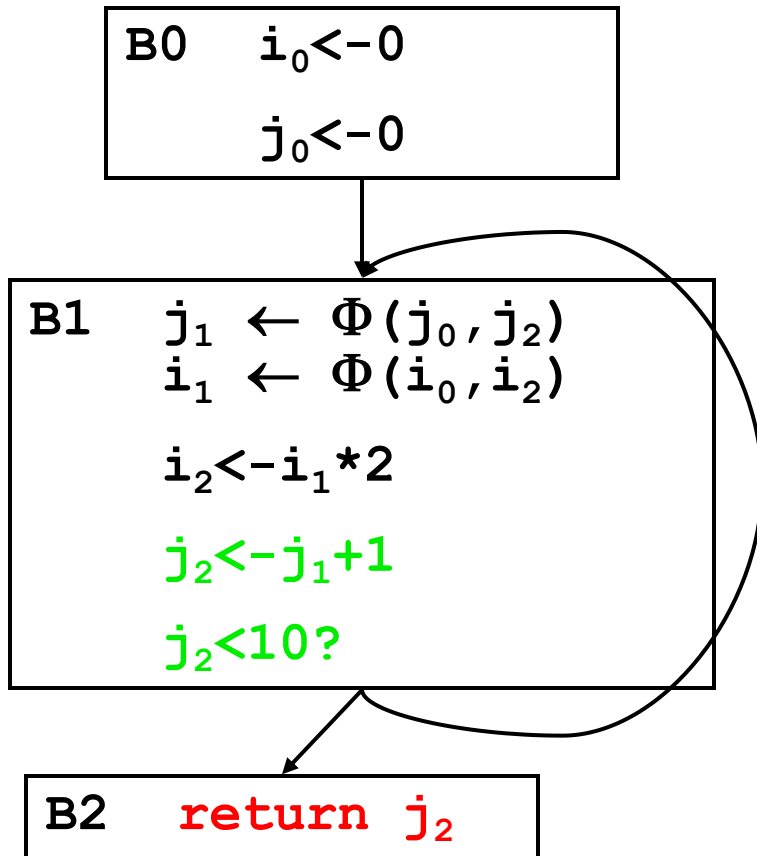
Red annotations: A red circle around the set {entry} for node 2. A red line with arrows pointing to the sets {1, entry} and {entry} for nodes 1 and 0 respectively.

Aggressive Dead Code Elimination

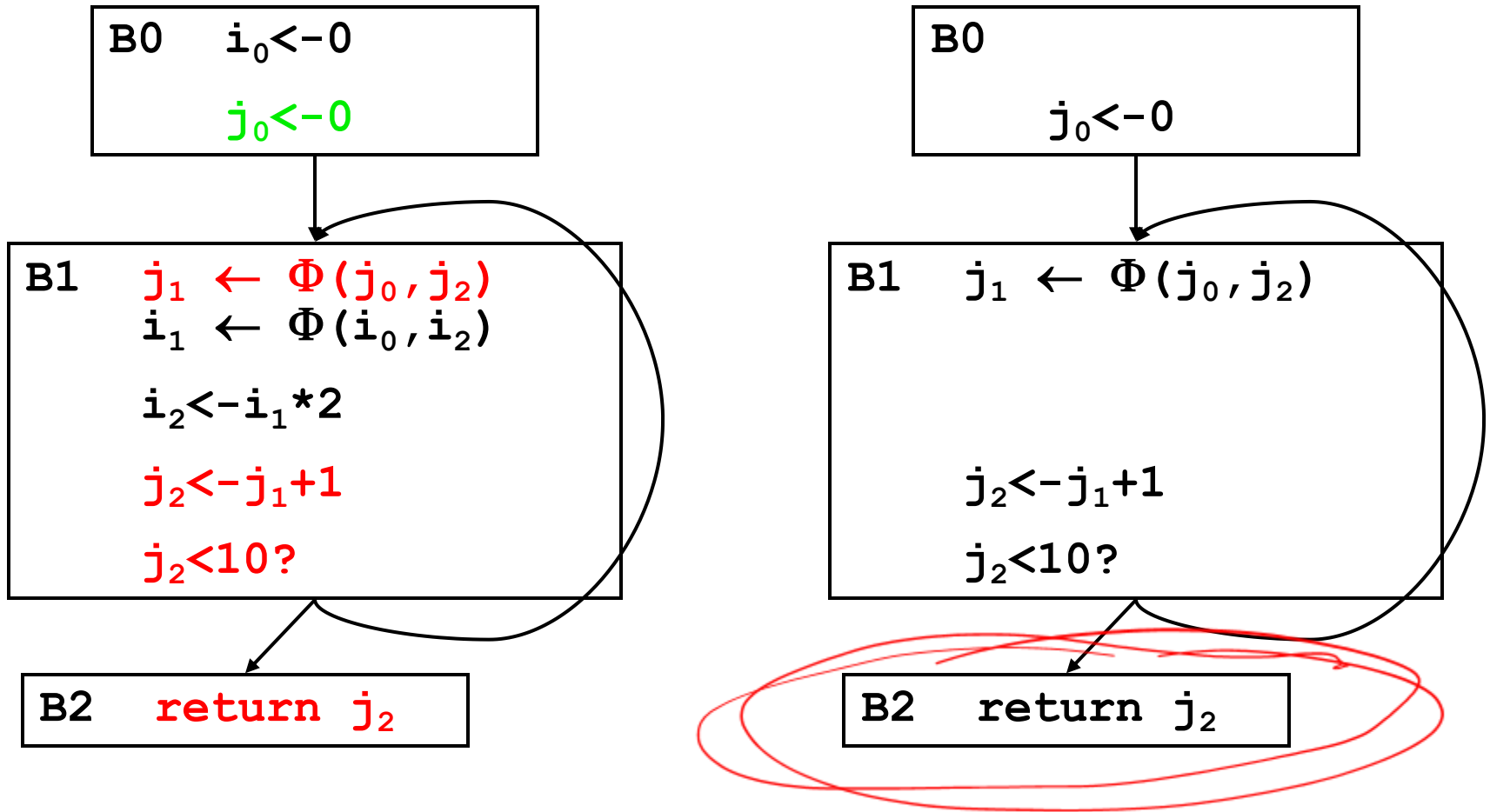
Assume a stmt is dead until proven otherwise.

```
while (|W| > 0) {  
    S ← W.removeOne()  
    if (S is live) continue;  
    mark S live, insert  
        - forall operands, S.operand.definers into W  
        - S.CD-1 into W  
}
```

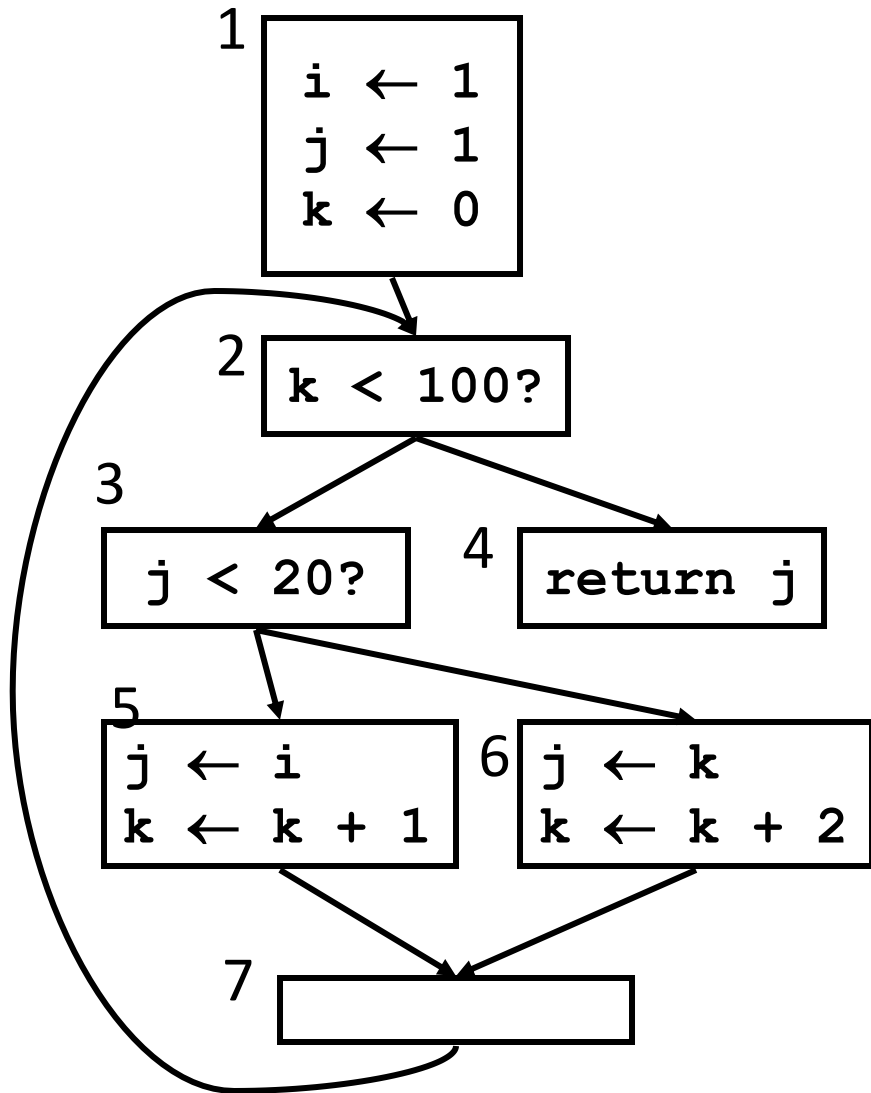
Example DCE



Example DCE

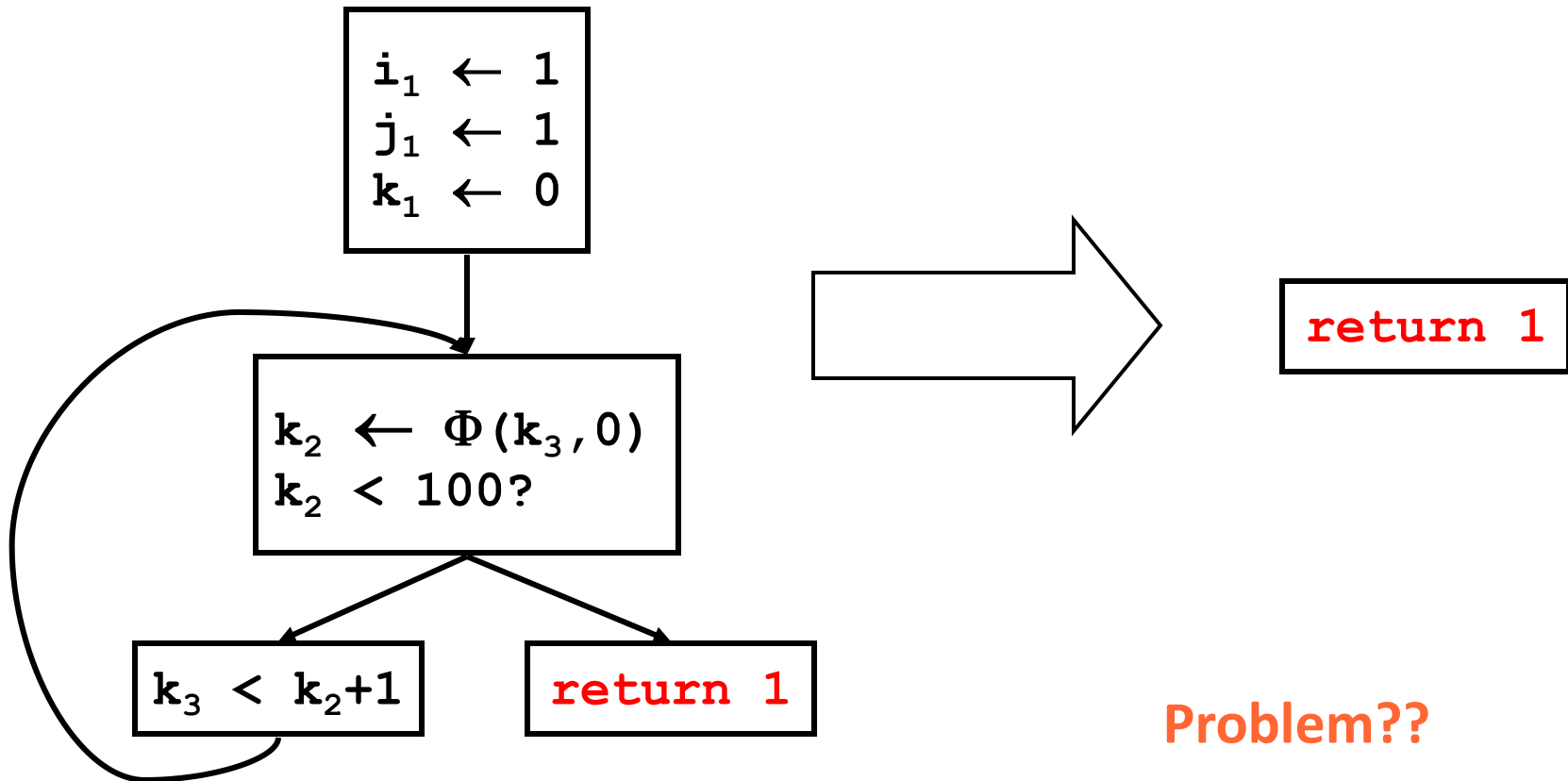


CCP Example



- Does block 6 ever execute?
- Simple CP can't tell
- CCP can tell:
 - Assumes blocks don't execute until proven otherwise
 - Assumes Values are constants until proven otherwise

CCP -> DCE

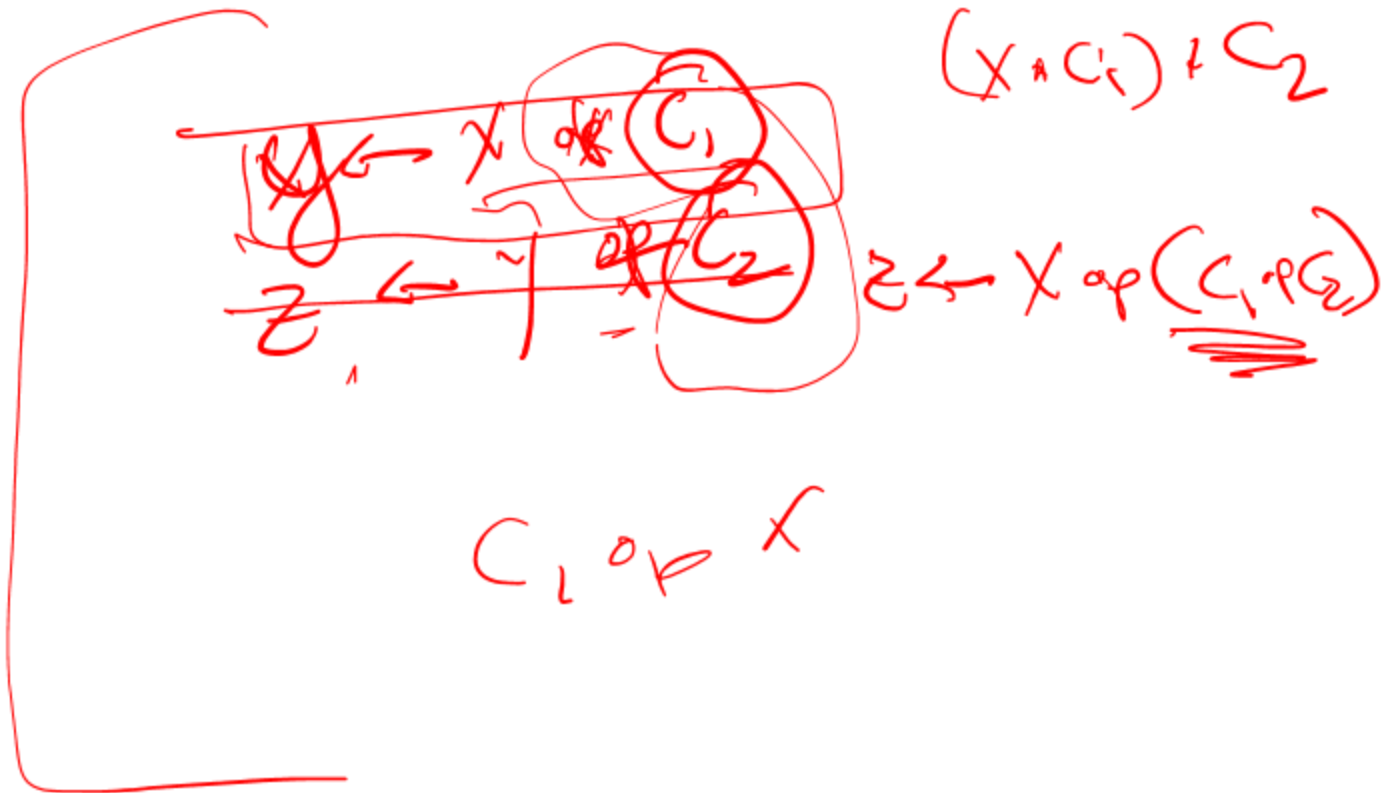


Problem??

Some Peephole Optimizations

- Constant Folding
- Strength Reduction
- Null Sequence Elimination
- ... (hundreds and hundreds of these)

Constant Folding



Strength Reduction (AKA Algebraic Simplification)

$$x \cdot 5 \Rightarrow \text{leaf } (x, x, \cancel{x})$$

~~$x + x + x$~~

$$x \cdot 2^n \quad x \ll n$$

$$x / 2^n \quad \begin{matrix} x \cdot 1 \\ x \cdot \phi \end{matrix} \quad \cancel{x} \gg n$$

$x + 1 \geq x$

Null Sequence Elimination

$X \leftarrow X$

~~$X \leftarrow M[x]$~~
 $M[x] \leftarrow X$
 ~~$X \leftarrow M[x]$~~ $\rightarrow$ $L2:$ $\begin{array}{l} \text{goto } L2 \\ \} \end{array}$

Today

- What is a loop?
- Control Flow Analysis
- Finding Loops
- Natural Loops
- Reducibility
- Classic Loop Optimizations
 - LICM
 - Induction Variable Elimination

Common loop optimizations

- | | |
|--|--|
| <ul style="list-style-type: none">• Hoisting of loop-invariant computations<ul style="list-style-type: none">– pre-compute before entering the loop• Elimination of induction variables<ul style="list-style-type: none">– change $p=i*w+b$ to $p=b, p+=w$, when w, b invariant• Loop unrolling<ul style="list-style-type: none">– to improve scheduling of the loop body | Scalar opts,
DF analysis,
Control flow
analysis |
| <ul style="list-style-type: none">• Software pipelining<ul style="list-style-type: none">– To improve scheduling of the loop body• Loop permutation<ul style="list-style-type: none">– to improve cache memory performance | Requires
understan
ding data
dependen
cies |

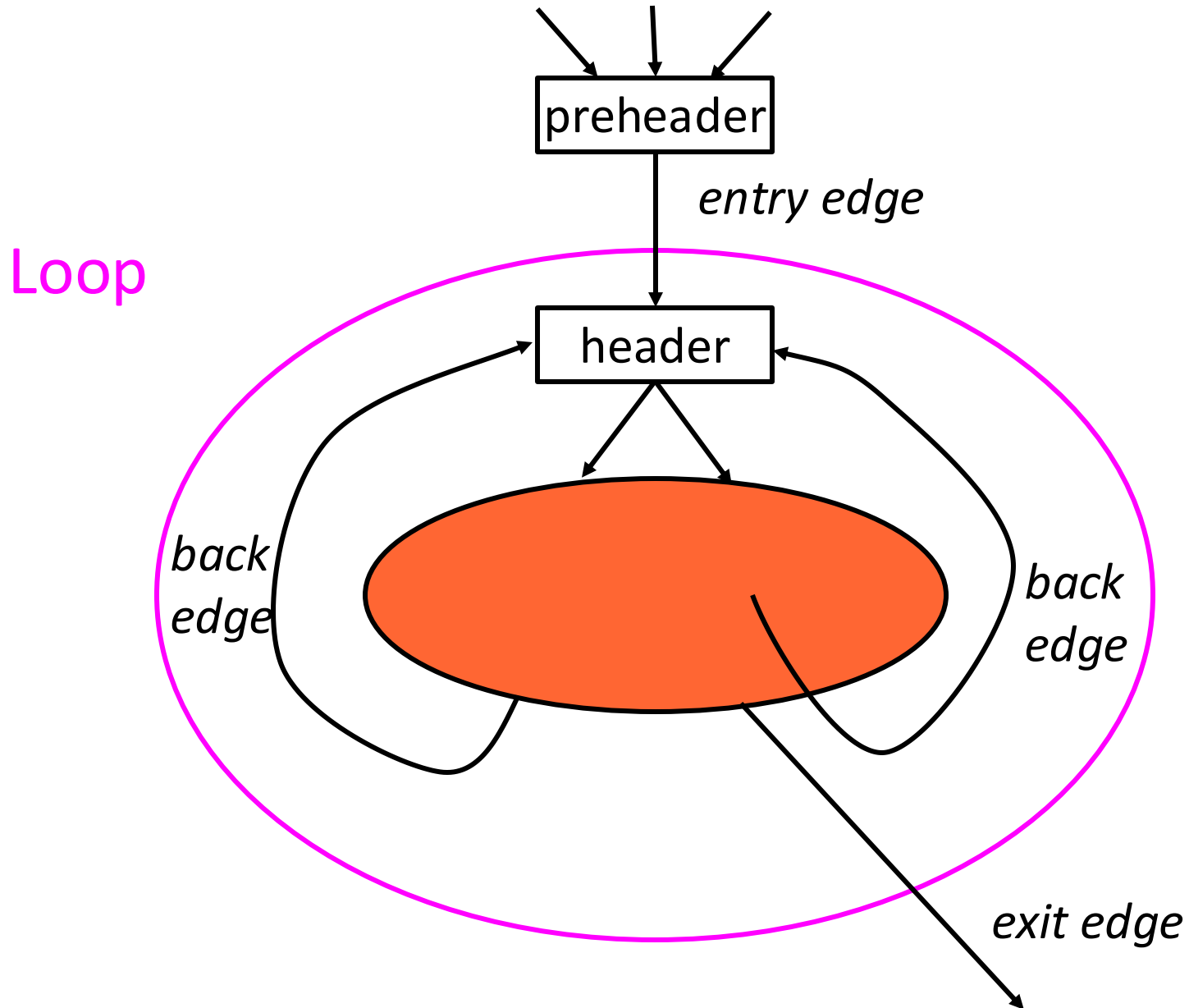
Loops are Key

- Loops are **extremely** important
 - the “90-10” rule
- Loop optimization involves
 - understanding control-flow structure
 - Understanding data-dependence information
 - sensitivity to side-effecting operations
 - extra care in some transformations such as register spilling

Finding Loops

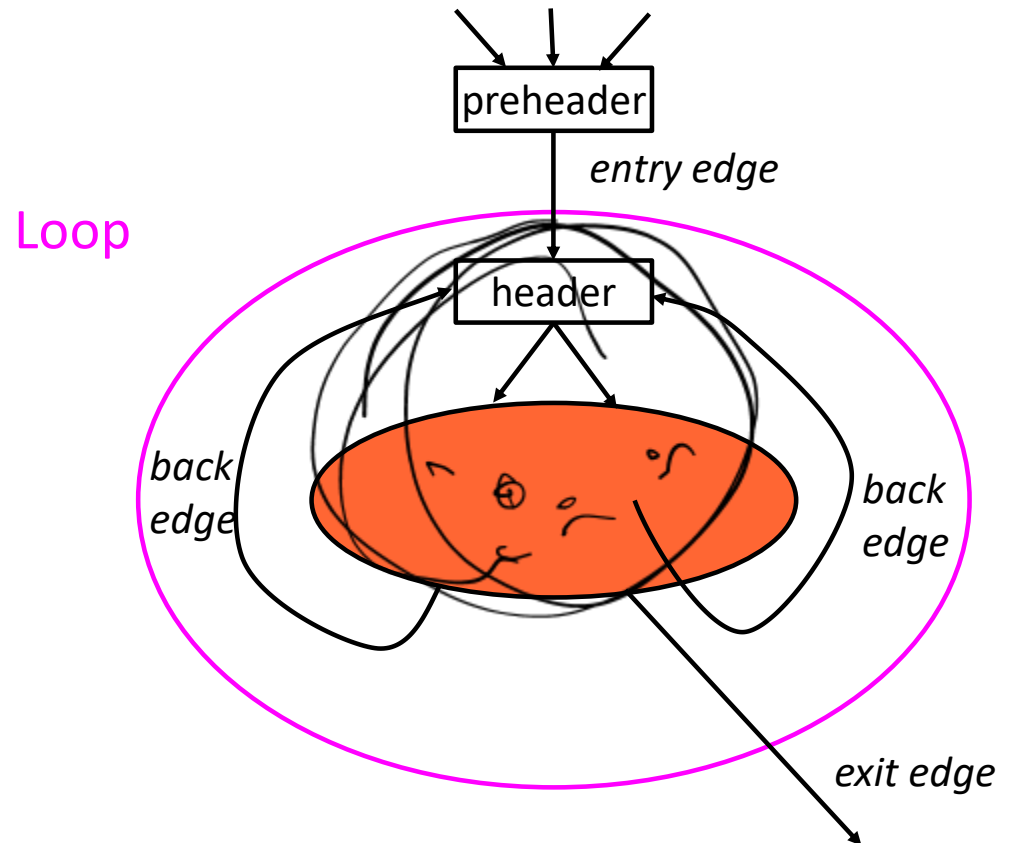
- To optimize loops, we need to find them!
- Could use source language loop information in the abstract syntax tree...
- **BUT:**
 - There are multiple source loop constructs: for, while, do-while, even goto in C
 - Want IR to support different languages
 - Ideally, we want a single concept of a loop so all have same analysis, same optimizations
 - **Solution:** dismantle source-level constructs, then re-find loops from fundamentals

A Loop



Loop Terminology

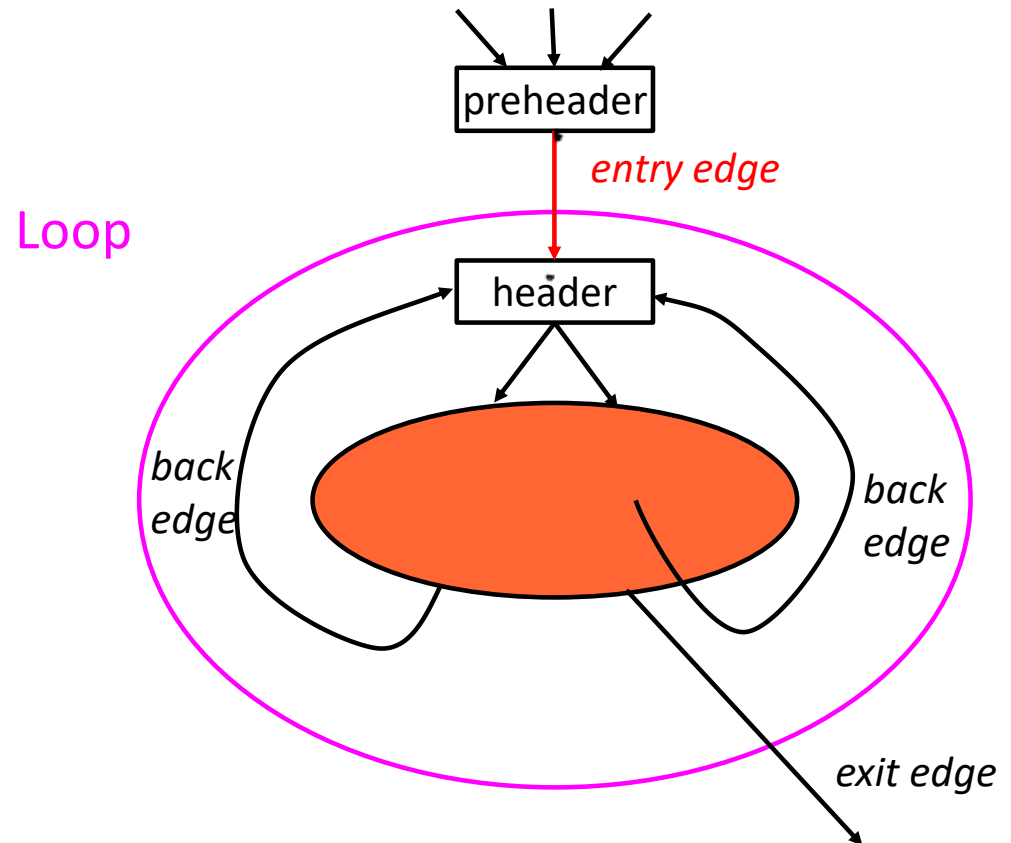
Loop: Strongly Connected Component of CFG



Loop Terminology

Loop: Strongly Connected Component of CFG

Entry Edge: tail not in loop, head in loop.

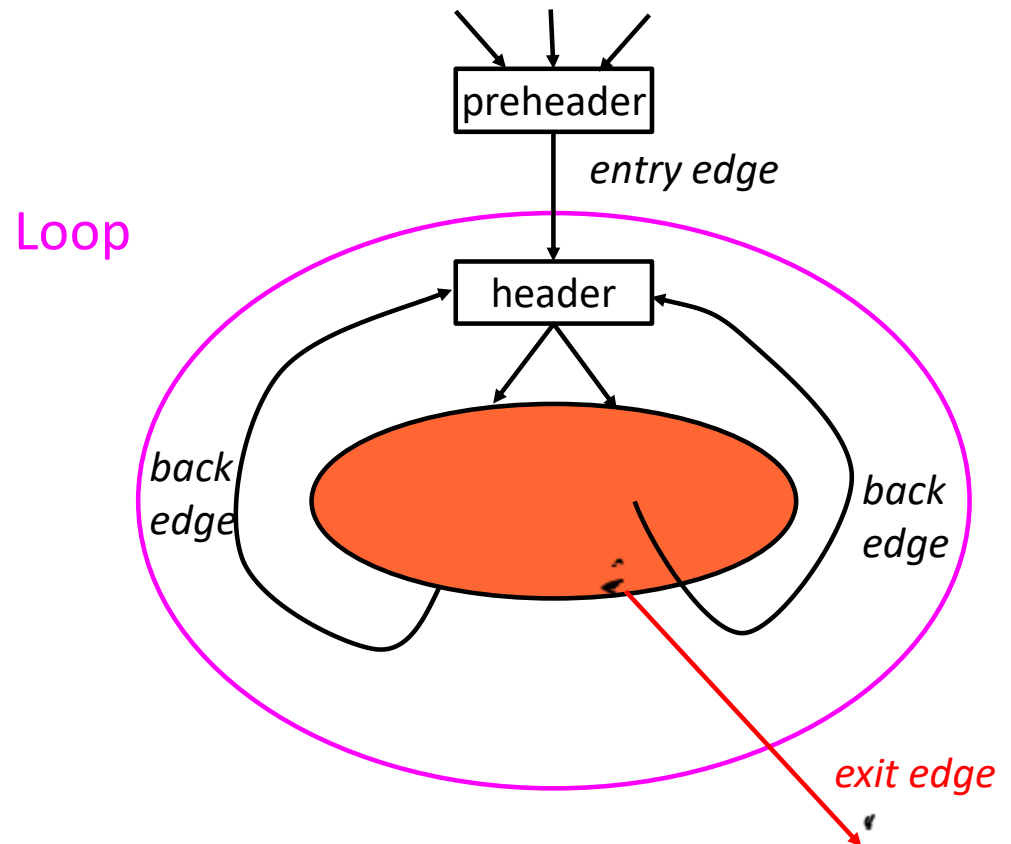


Loop Terminology

Loop: Strongly Connected Component of CFG

Entry Edge: tail not in loop, head in loop.

Exit edge: tail in loop, head not in loop



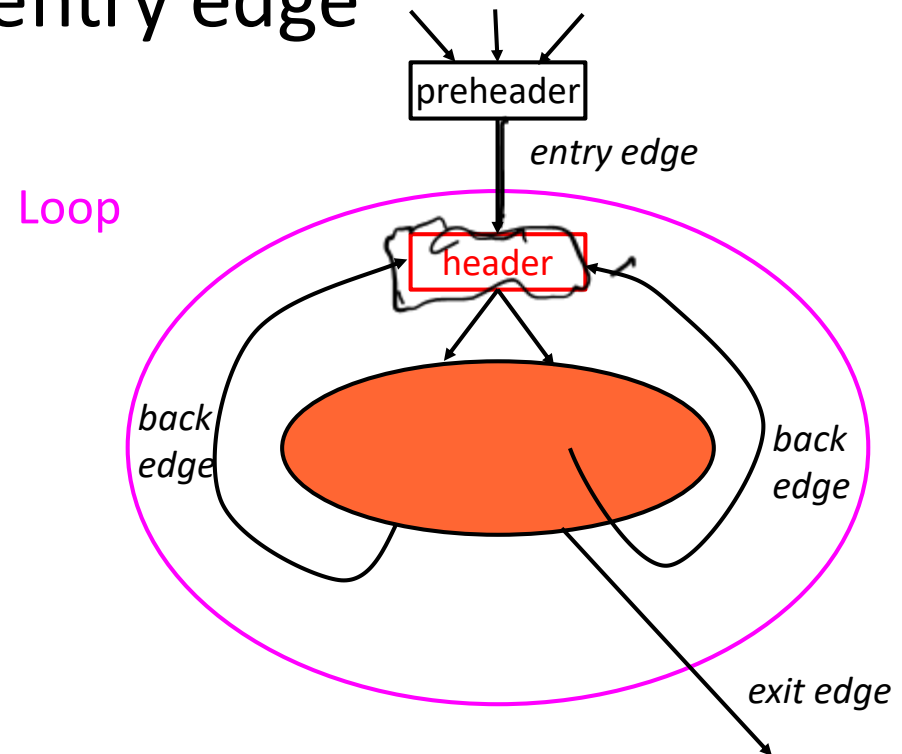
Loop Terminology

Loop: Strongly Connected Component of CFG

Entry Edge: tail not in loop, head in loop.

Exit edge: tail in loop, head not in loop

Loop Header: target of entry edge



Loop Terminology

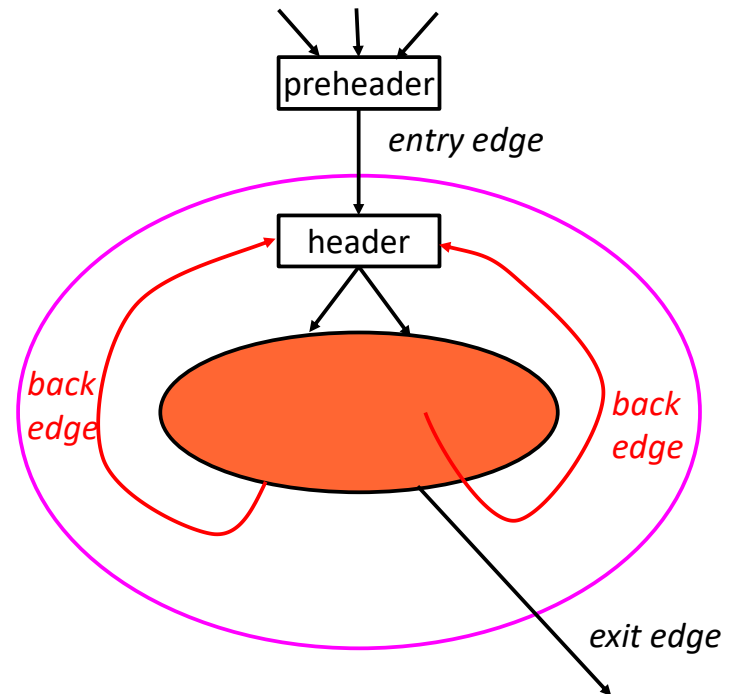
Loop: Strongly Connected Component of CFG

Entry Edge: tail not in loop, head in loop.

Exit edge: tail in loop, head not in loop

Loop Header: target of entry edge

Back Edge: target is header,
source is in loop



Loop Terminology

Loop: Strongly Connected Component of CFG

Entry Edge: tail not in loop, head in loop.

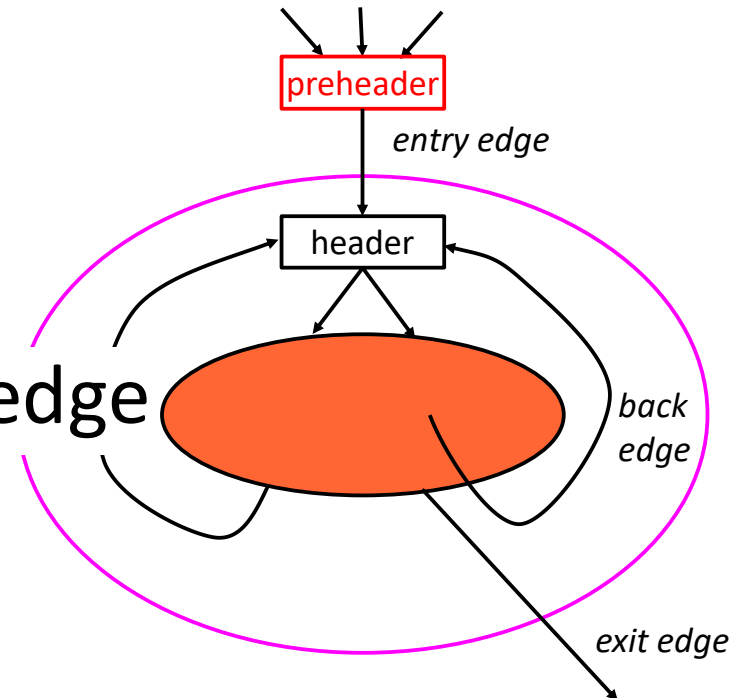
Exit edge: tail in loop, head not in loop

Loop Header: target of entry edge

Back Edge: target is header,
source is in loop

Preheader:

Source of the only entry edge



(You may have to add a preheader.)

Loop Terminology

Loop: Strongly Connected Component of CFG

Entry Edge: tail not in loop, head in loop.

Exit edge: tail in loop, head not in loop

Loop Header: target of entry edge

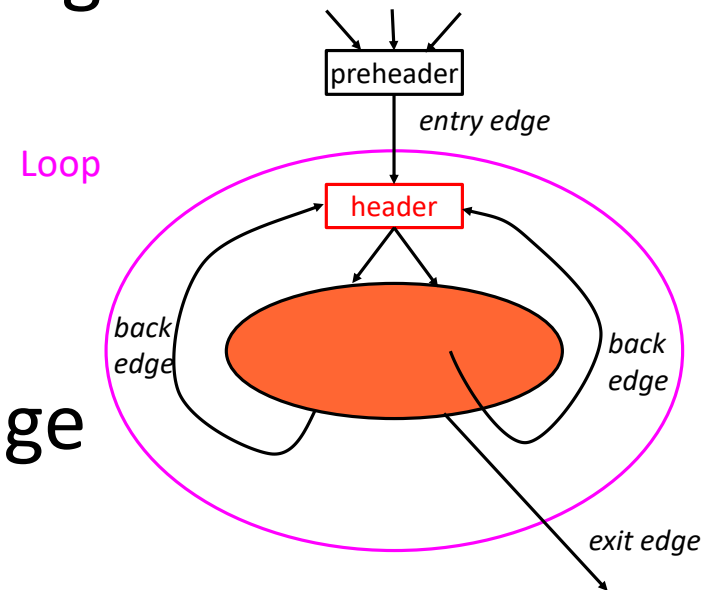
Back Edge: target is header,
source is in loop

Preheader:

Source of the only entry edge

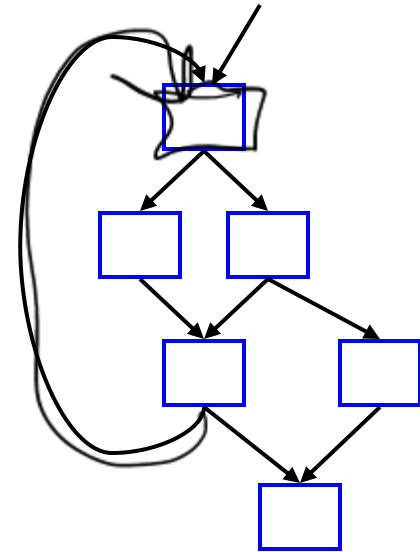
Natural Loop:

A Loop with only a single loop header



Finding Loops

- To optimize loops, we need to find them!
- Specifically:
 - loop-header node(s)
 - nodes in a loop that have immediate predecessors not in the loop
 - back edge(s)
 - control-flow edges to previously executed nodes
 - all nodes in the loop body



Control-flow analysis

- Many languages have goto and other complex control, so loops can be hard to find in general
- Determining the control structure of a program is called **control-flow analysis**
- Based on the notion of **dominators**



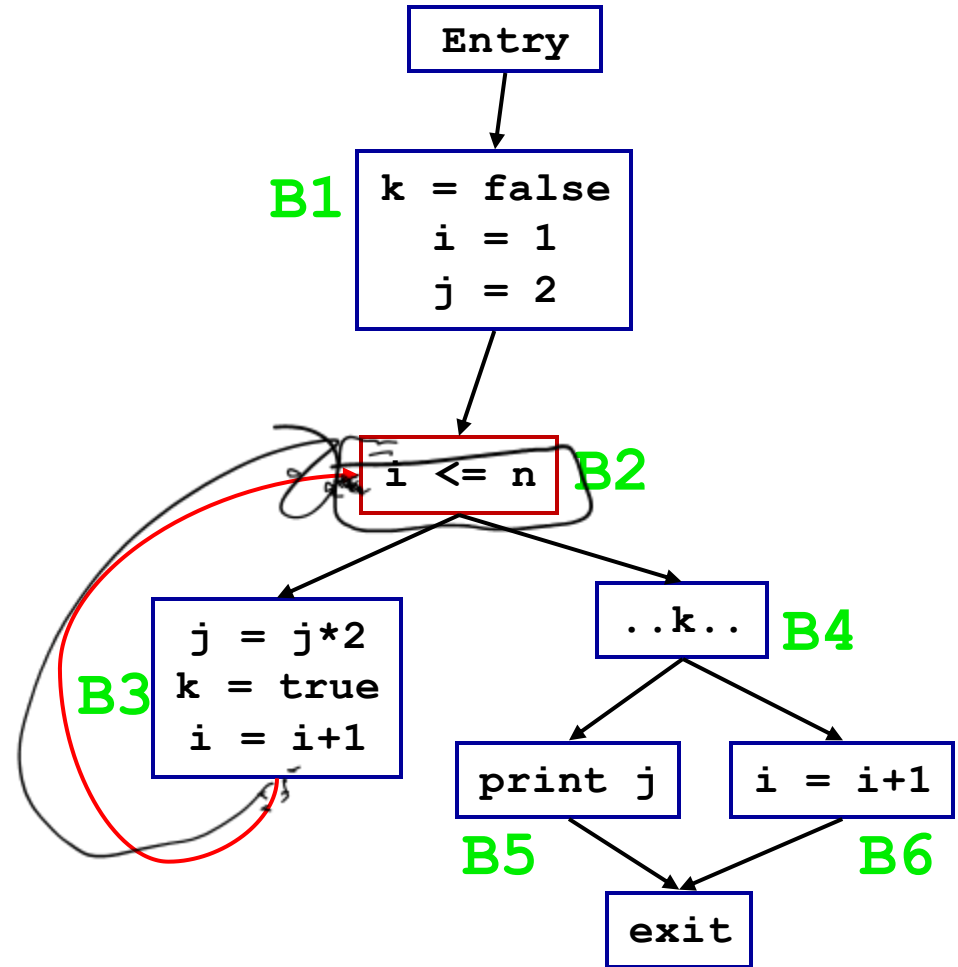
Recall: Dominators

- $a \text{ dom } b$
 - node a dominates b if every possible execution path from entry to b includes a
- $a \text{ sdom } b$
 - a strictly dominates b if $a \text{ dom } b$ and $a \neq b$

- $a \text{ idom } b$
 - a immediately dominates b if $a \text{ sdom } b$, AND there is no c such that $a \text{ sdom } c$ and $c \text{ sdom } b$

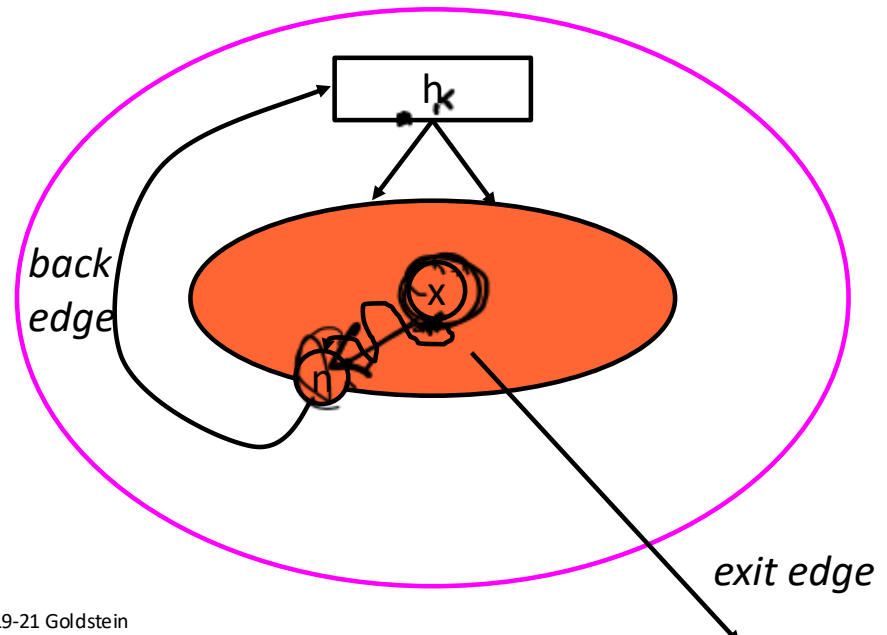
Back edges and loop headers

- A control-flow edge from node B3 to B2 is a **back edge** if $B2 \text{ dom } B3$
- Furthermore, in that case node B2 is a **loop header**



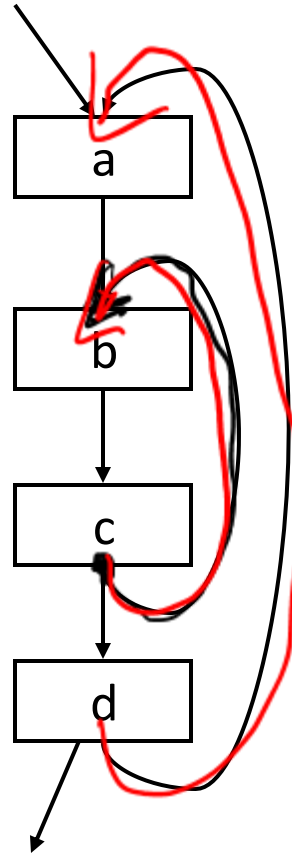
Natural loop

- Consider a back edge from node n to node h
- The natural loop of $n \rightarrow h$ is the set of nodes L such that for all $x \in L$:
 - $h \text{ dom } x$ and
 - there is a path from x to n not containing h



Examples

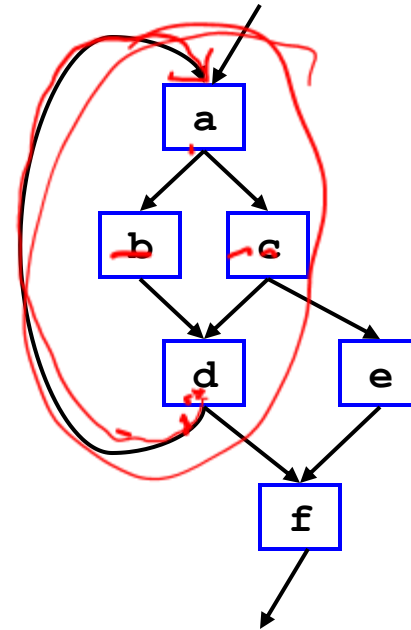
Simple example:



(often it's more complicated, since a FOR loop found in the source code might need an if/then guard)

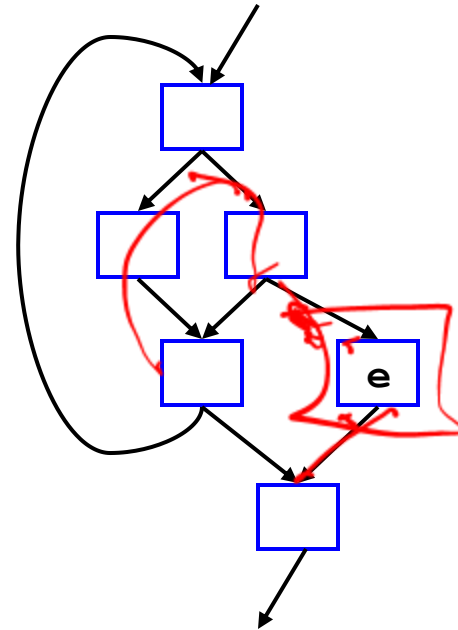
Examples

Try this:



Examples

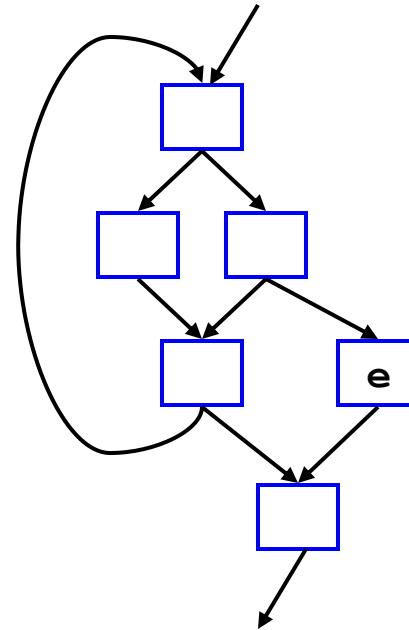
```
for (..) {  
  if {  
    ...  
  } else {  
    ...  
    if (x) {  
      e;  
      break;  
    }  
  }  
}
```



Examples

```
for (...) {  
  if {  
    ...  
  } else {  
    ...  
    if (x) {  
      e;  
      break;  
    }  
  }  
}
```

lexically, in loop,
but not in natural
loop

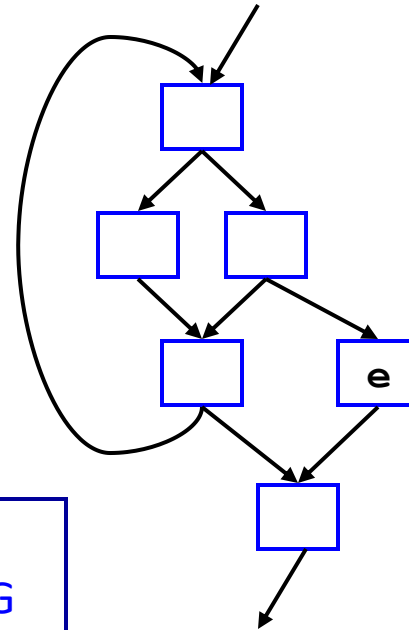


Examples

```
for (...) {  
  if {  
    ...  
  } else {  
    ...  
    if (x) {  
      e;  
      break;  
    }  
  }  
}
```

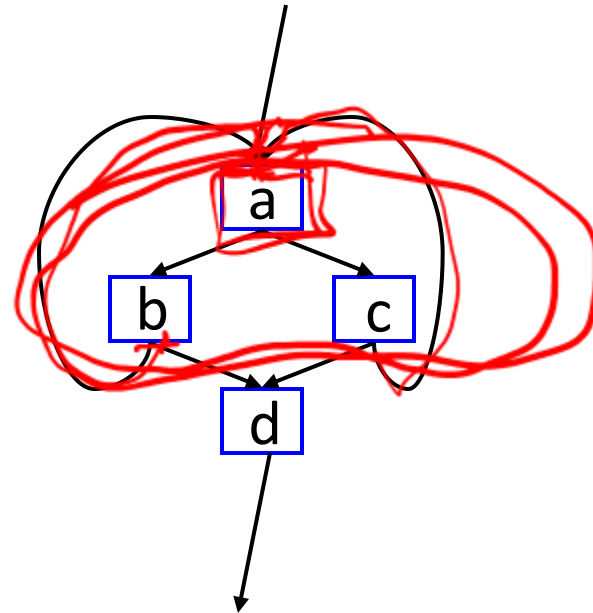
lexically, in loop,
but not in natural
loop

and another
reason why CFG
analysis is
preferred over
source/AST loops



Examples

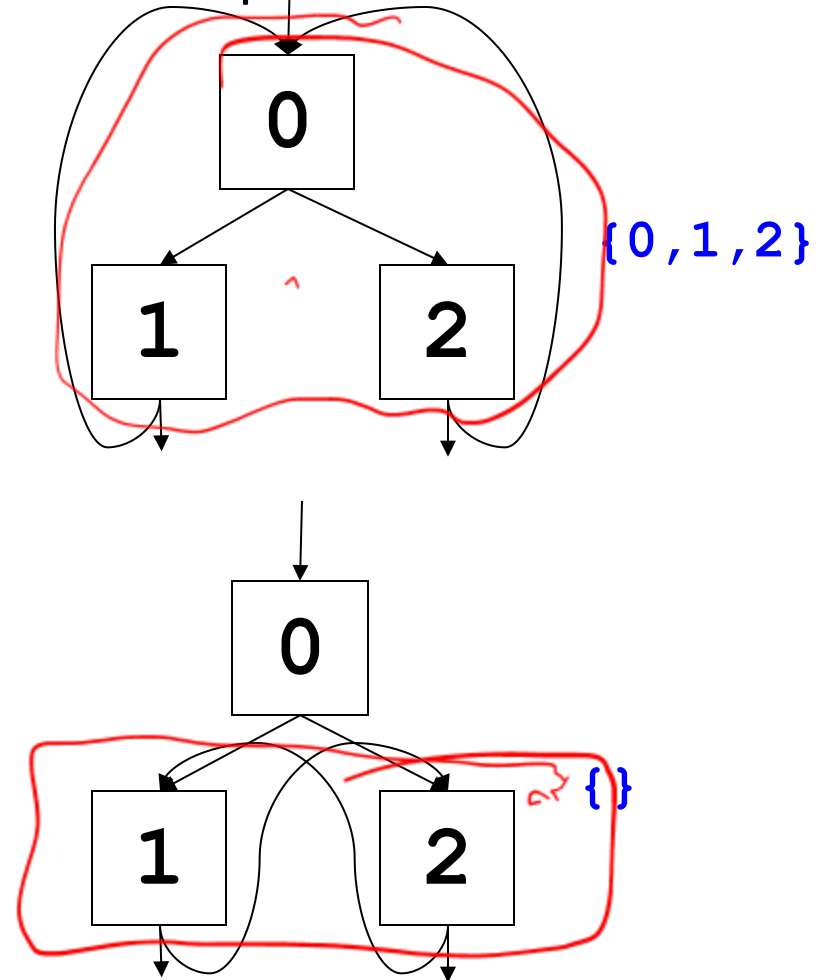
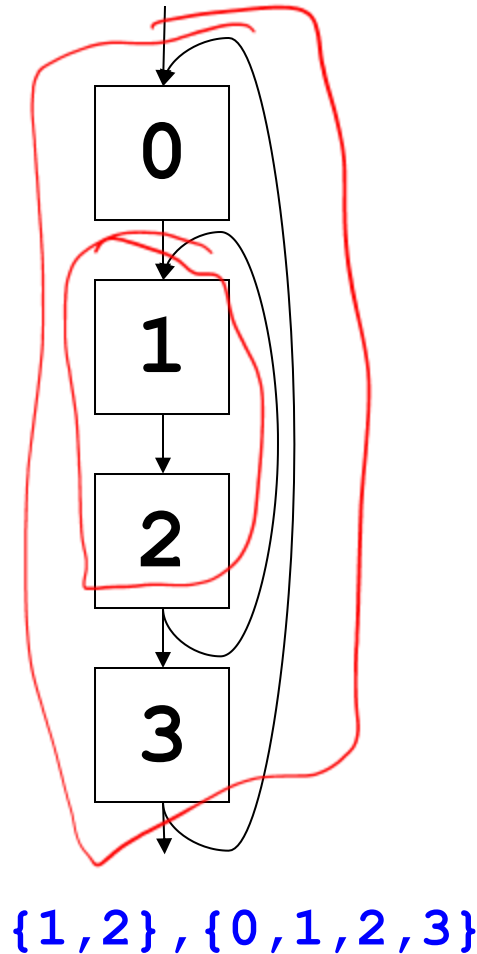
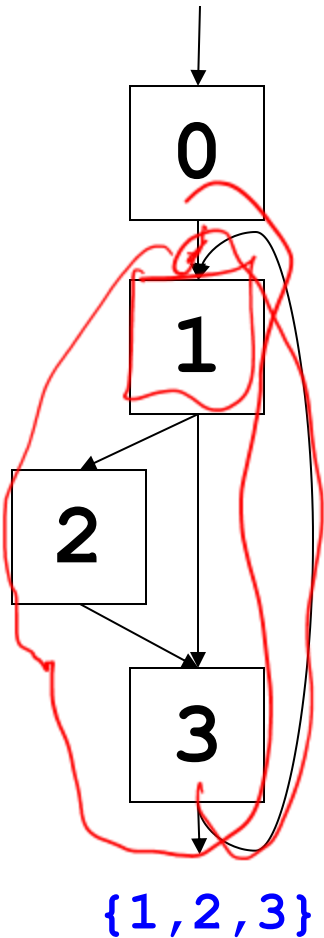
- Yes, it can happen in C



Natural Loops

One loop per header..

What are the natural loops?



Nested Loops

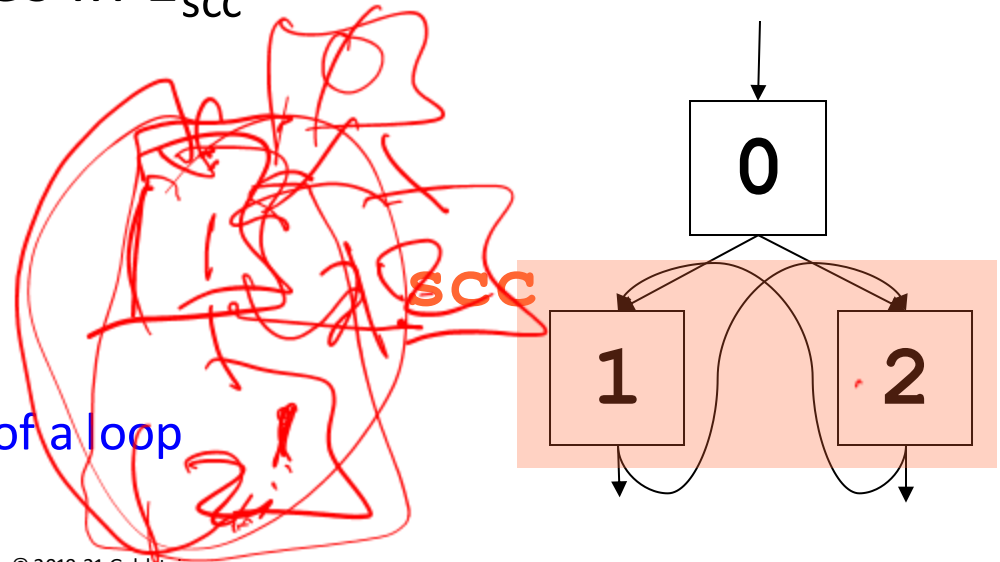
- Unless two natural loops have the same header, they are either disjoint or *nested* within each other
- If *A* and *B* are loops (sets of blocks) with headers *a* and *b* such that $a \neq b$ and $b \in A$
 - $B \subset A$
 - loop *B* is *nested* within *A*
 - *B* is the *inner loop*
- Can compute the loop-nest tree

General Loops

- A more general looping structure is a *strongly connected component* of the control flow graph
 - subgraph $\langle N_{\text{scc}}, E_{\text{scc}} \rangle$ such that

every block in N_{scc} is reachable from every other node using only edges in E_{scc}

Not very useful definition of a loop



Reducible Flow Graphs

There is a special class of flow graphs, called **reducible flow graphs**, for which several code-optimizations are especially easy to perform.

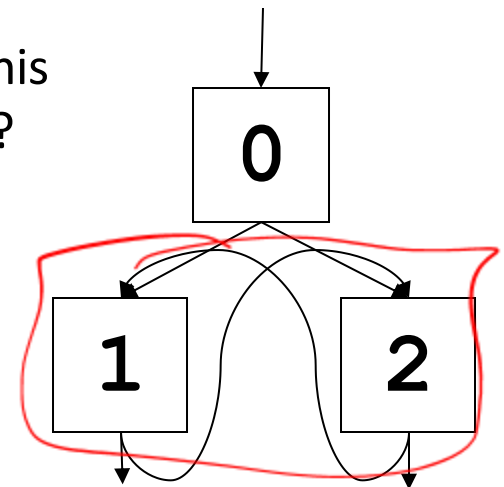
In reducible flow graphs **loops are unambiguously defined and dominators can be efficiently computed.**

Reducible flow graphs

Definition: A flow graph G is **reducible** iff we can partition the edges into two disjoint groups, **forward edges** and **back edges**, with the following two properties.

1. The forward edges form an acyclic graph in which every node can be reached from the initial node of G .
2. The back edges consist only of edges whose heads dominate their tails.

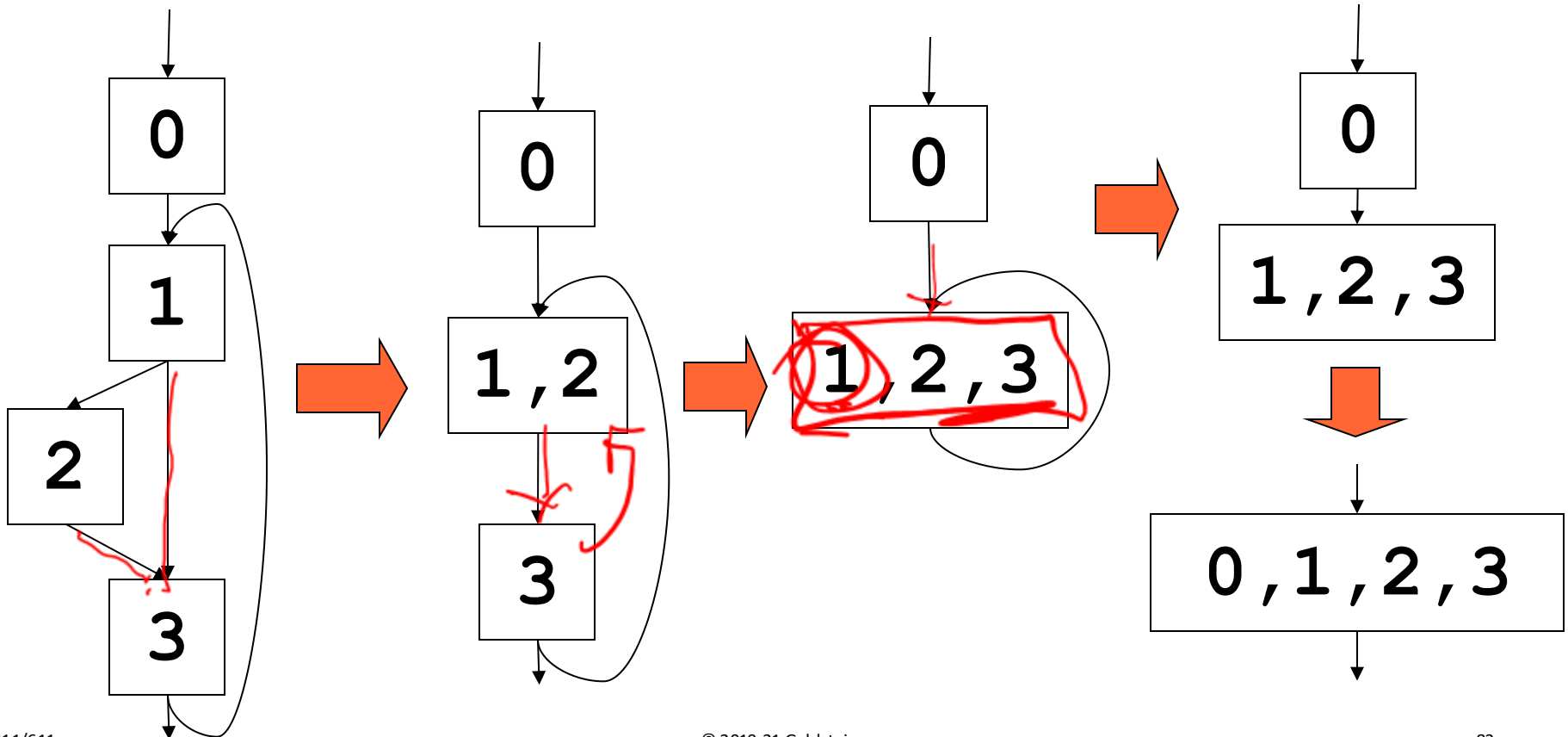
Why isn't this reducible?



*This flow graph has **no back edges**. Thus, it would be reducible if the entire graph were acyclic, which is not the case.*

Alternative definition

- Definition:** A flow graph G is **reducible** if we can repeatedly collapse (reduce) together blocks (x,y) where x is the only predecessor of y (ignoring self loops) until we are left with a single node



Properties of Reducible Flow Graphs

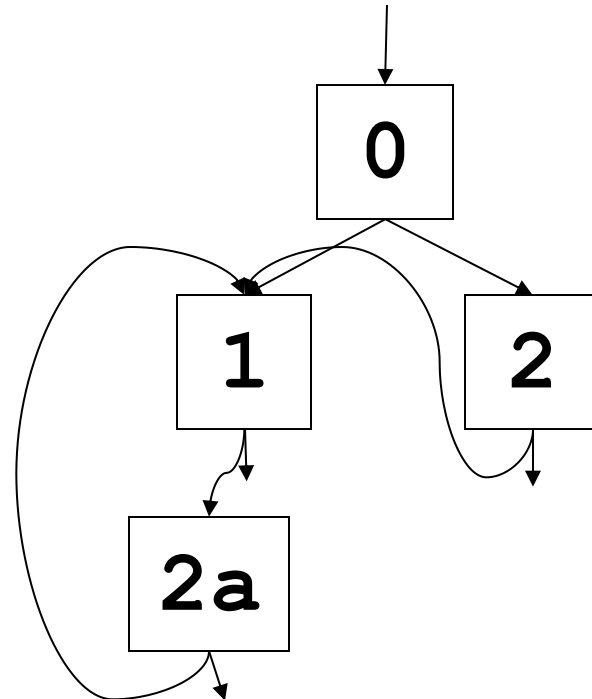
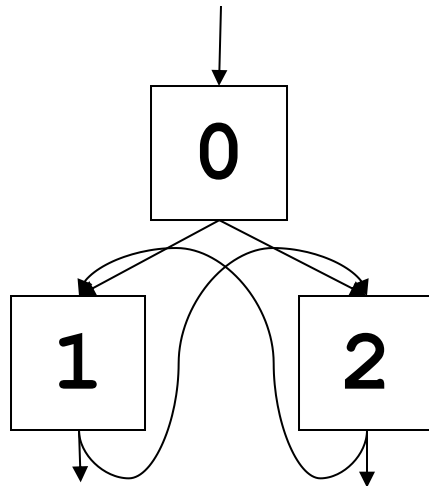
- In a reducible flow graph,
all loops are natural loops
- Can use DFS to find loops
- Many analyses are more efficient
 - polynomial versus exponential

Good News

- Most flow graphs are reducible
- Languages prohibit irreducibility
 - goto free C
 - Java
 - C0 ☺
- Programmers usually don't use such constructs even if they're available
 - >90% of old Fortran code reducible

Dealing with Irreducibility

- Don't
- Can split nodes and duplicate code to get reducible graph
 - possible exponential blowup
- Other techniques...



Loop optimizations: Hoisting of loop-invariant computations