code analysis *and* transformation

More DFAs

Simone Campanoni
simone.campanoni@northwestern.edu

Outline

- Reaching definition and constant propagation
- More DFAs and related transformations
- DFAs without assumptions
- Other uses of DFA

Constant propagation: problem definition

Given a program, we would like to know
for every point in that program,
which variables have constant values, and which ones do not.

A variable has a constant value at a certain point in the CFG
if every execution that reaches that point
sees that variable holding the same constant value.

We are now going to implement
constant propagation automatically
and by relying only on reaching definition

```
i: x = 0
j: N = N + 1
k: Z = x + N
```

Reaching definition summary

- Reaching definition data-flow analysis computes IN[i] and OUT[i] for every instruction i
- IN[i] (OUT[i]) includes definitions that reach just before (just after) instruction i
- Each IN/OUT set contains a mapping for every variable in the program to a “value”

Constant propagation

- For a use of variable v by instruction n

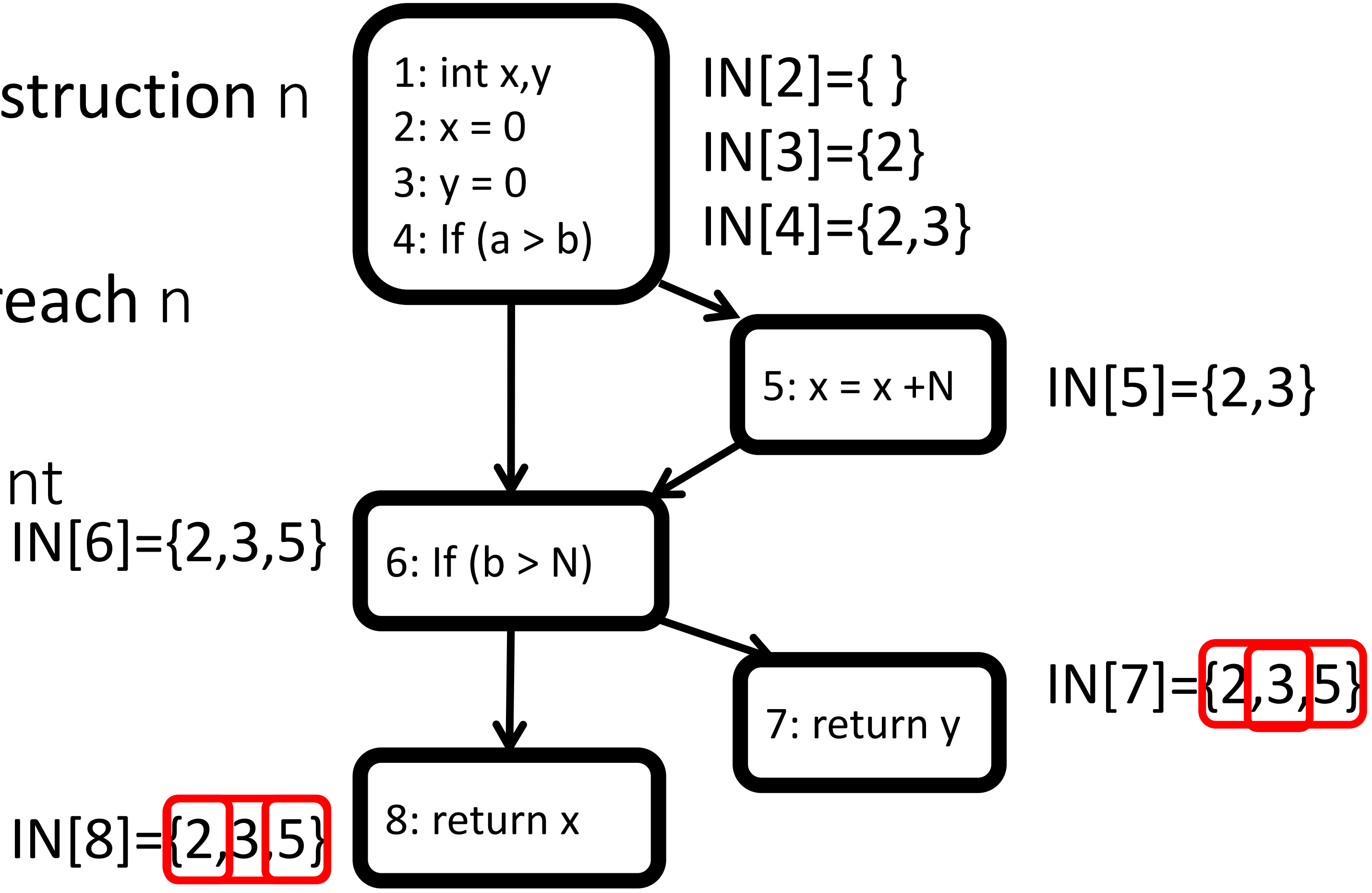
n: x = … v …

- If the definitions of v that reach n are all of the form

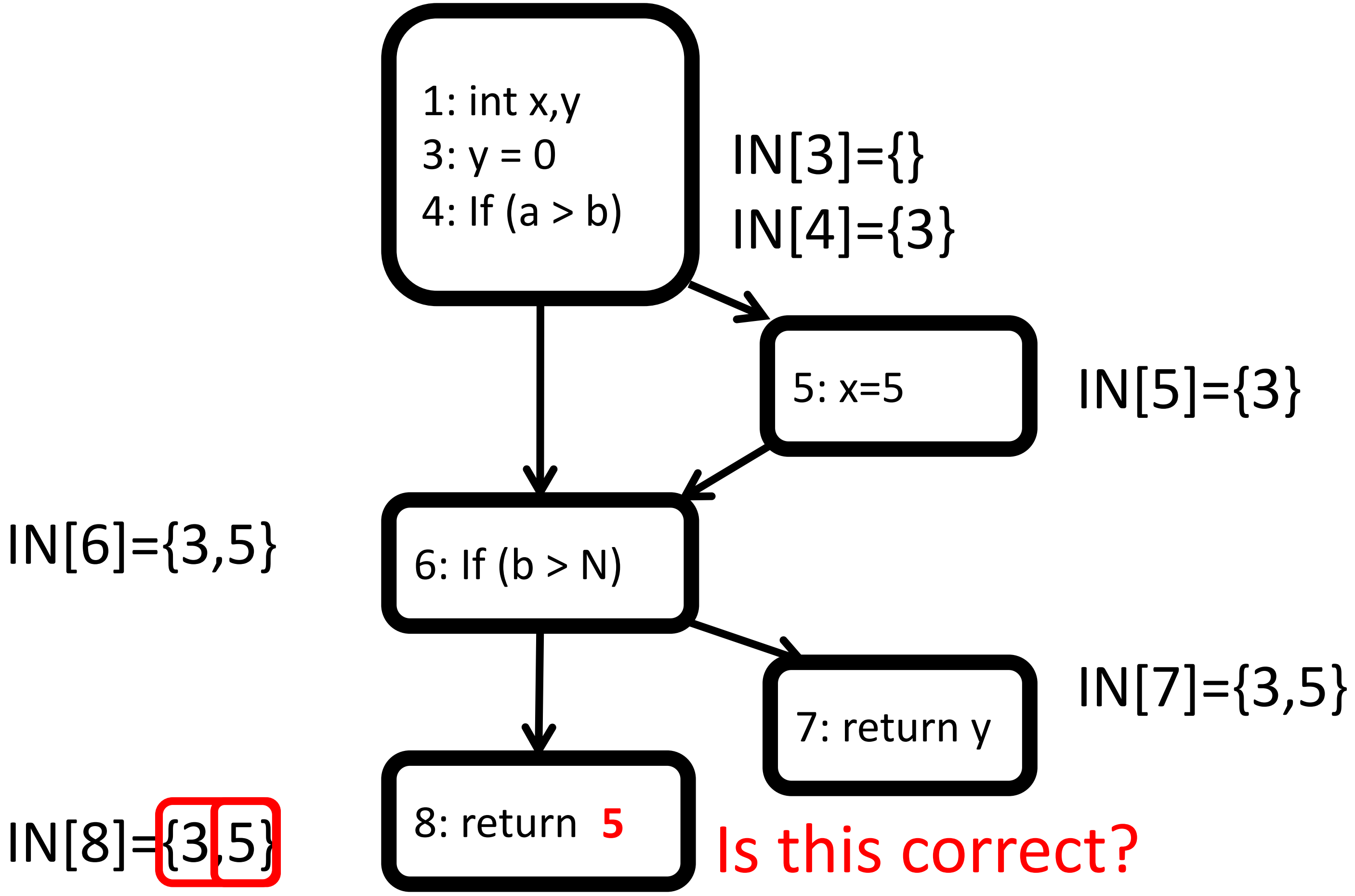
d: v = c //c is a generic constant

- then replace the uses of v in n with c

Do you see any problem?



Constant propagation problem?



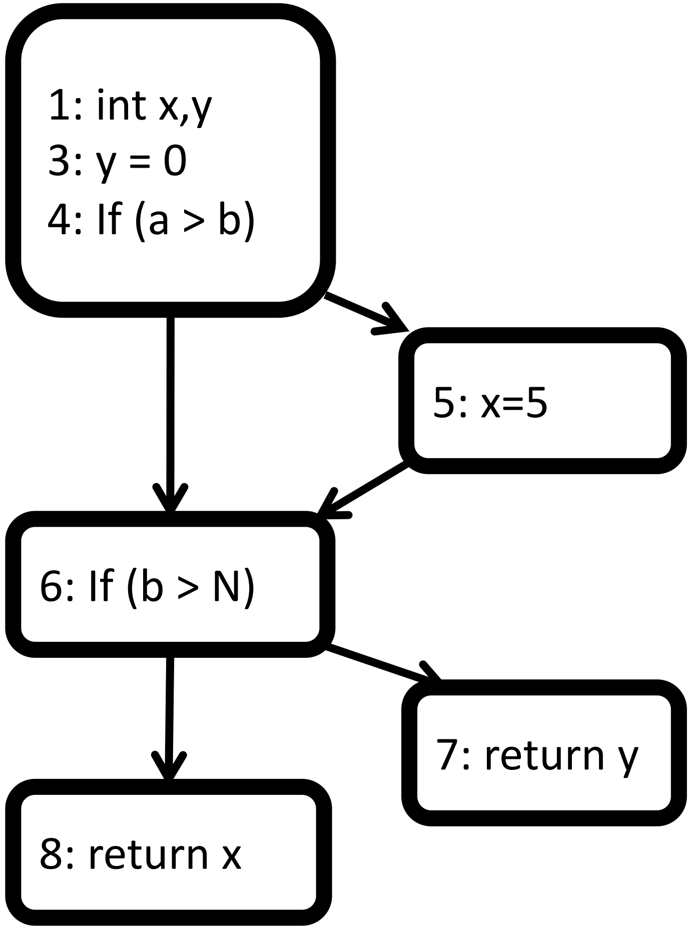
Undefined behavior: a funny interpretation

- **Undefined behavior** is the result of executing a program whose behavior is unpredictable

- Undefined behavior results in whatever compilers want the program being compiled to do

  even *to make demons fly out of your nose*
  - Undefined behavior is often referred to as *nasal demons*

Constant propagation problem?



Better solutions?

- Customize reaching definitions
- New analysis

Constant propagation for CAT

- **Undefined values enable optimizations**
- **What about in the CAT language?**
- CATData CAT_new (int64_t value);

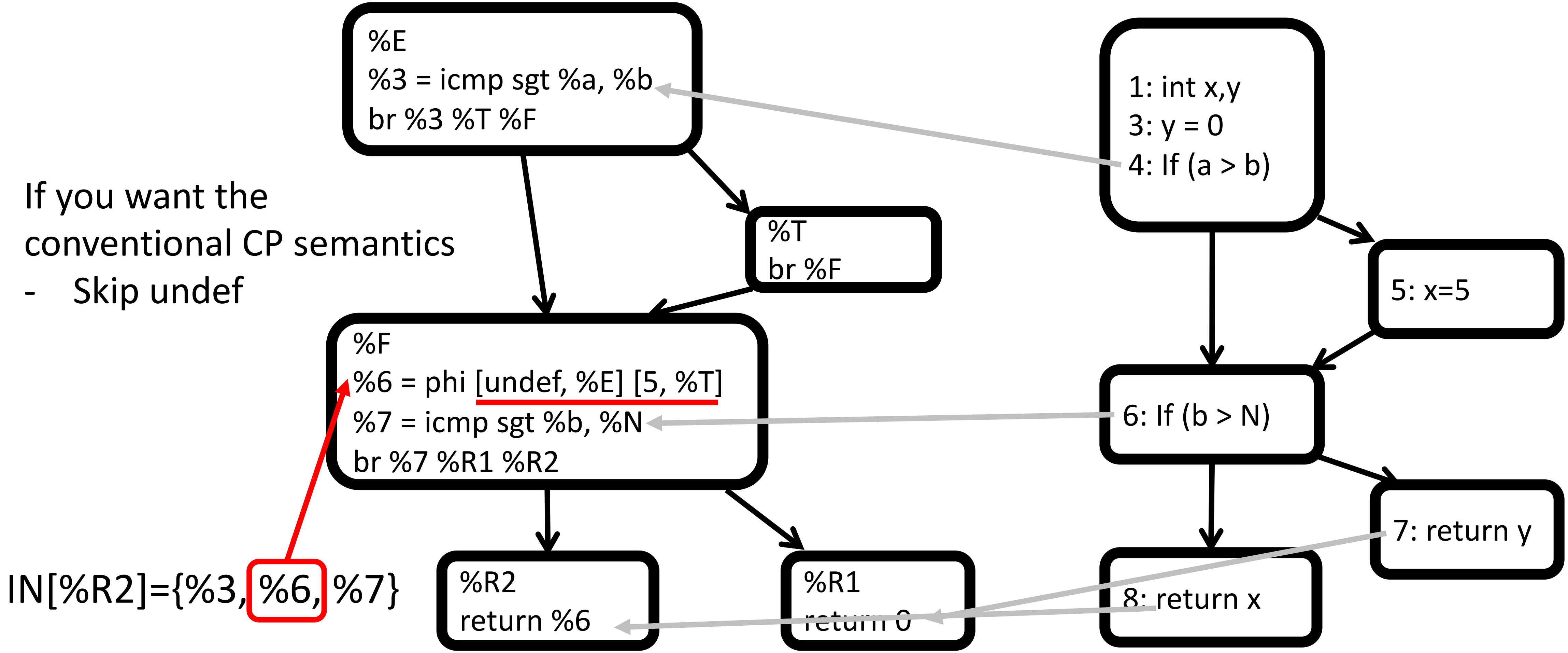
SSA simplifies transformations

- We learned constant propagation that relies on reaching definition
  - This transformation is correct for both SSA and non-SSA IRs
- Can we have a faster constant propagation for SSA IRs?
  - Yes
  - Let's first apply the previous constant propagation to an SSA IR to understand how to make it faster

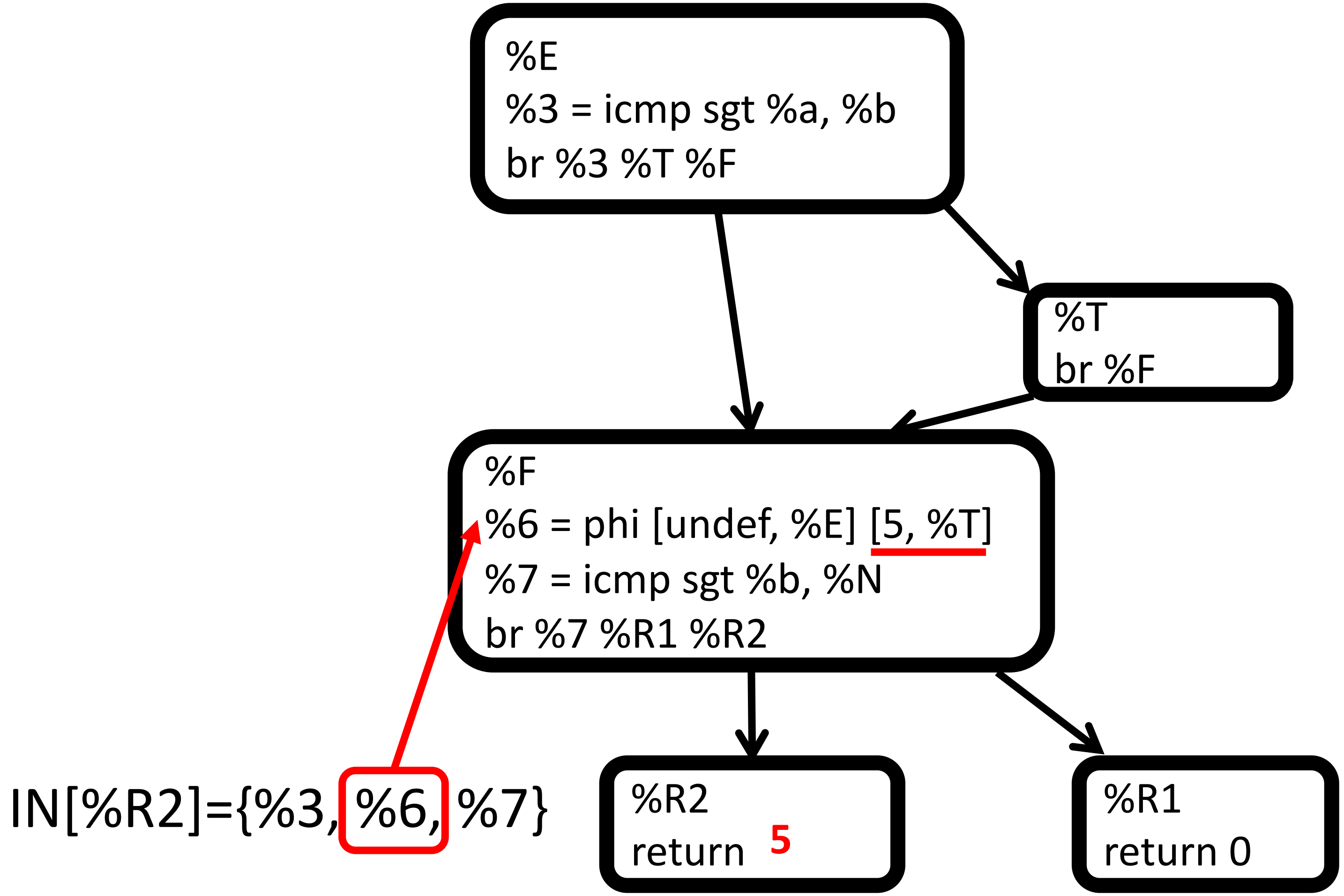
Constant propagation in SSA (in LLVM)

If you want the conventional CP semantics

- Skip undef



(Unnecessary thanks to SSA)
Constant propagation in SSA (in LLVM)

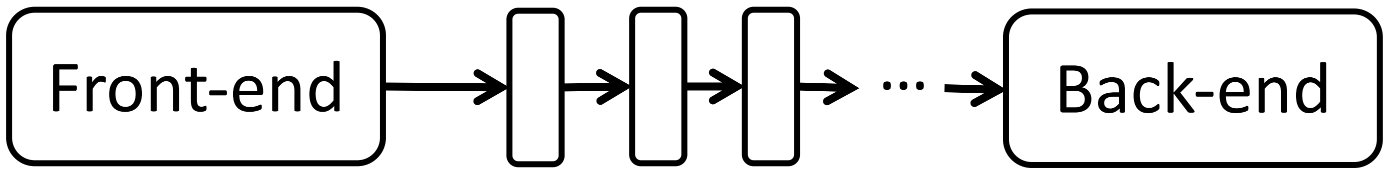


Thinking about what we have done

- What's the value of these propagations?
  - Constant propagation: less variable uses

    Redundant use of variables

- Redundancy is one of the
  main source of optimization in compilers

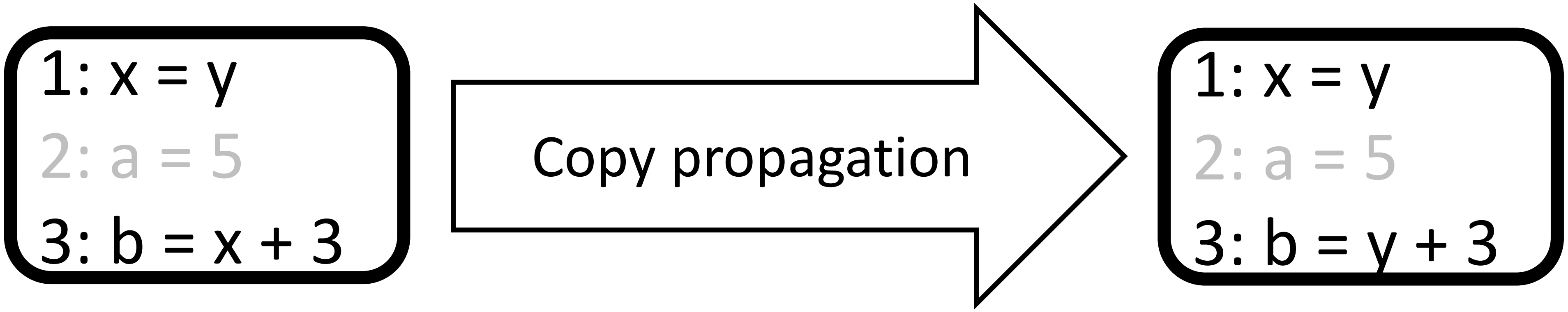


Outline

- Reaching definition and constant propagation
- More DFAs and related transformations
- DFAs without assumptions
- Other uses of DFA

Copy propagation: problem definition

Given a CFG, we would like to know
for every point in the program,
if a variable contains always the same value of another one.



How can we implement this transformation?

Reaching definition summary

- Reaching definition data-flow analysis computes IN[i] and OUT[i] for every instruction i
- IN[i] (OUT[i]) includes definitions that reach just before (just after) instruction i
- Each IN/OUT set contains a mapping for every variable in the program to a “value”;

Copy propagation

- For a use of variable v in statement n,

n: x = ... v ...

- If the definitions of v that reach n are all of the form

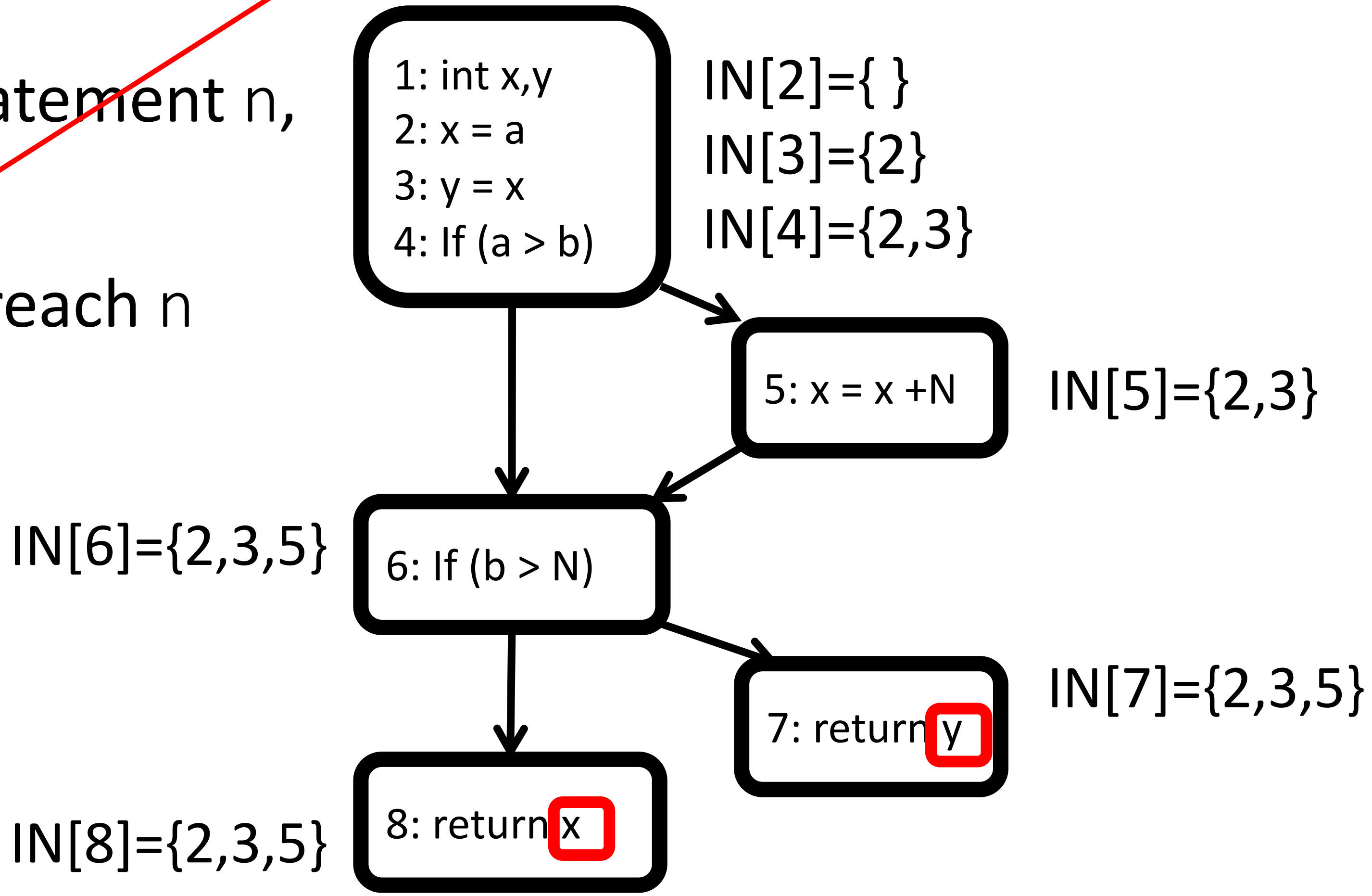
d: v = z [z is another variable]

- then replace the use of v in n with z

Do you see any problem?

How can we fix it?

(3 points, deadline = next class)

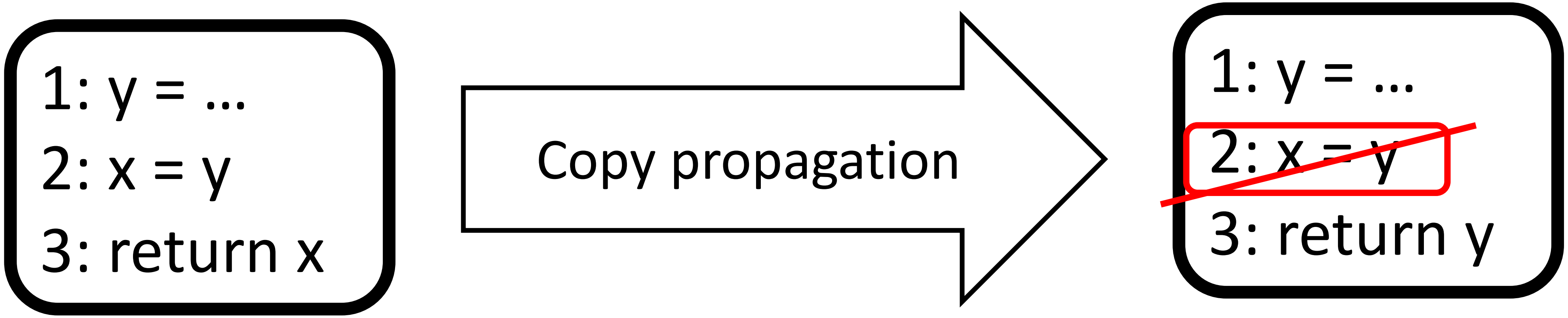


- Copy propagation relies on the same DFA of constant propagation
  - We got lucky
- Often, however, a new optimization often relies on a (or multiple) new data-flow analysis
  - It is important to learn how to define new and specialized DFAs

- Different DFAs have different
  - Data-flow values
  - Data-flow equations
  - Definitions of GEN and KILL sets
- Now we are going to see the most common examples

Dead code elimination: problem definition

Given a program, we would like to know statements/instructions that do not influence the program at all (i.e., dead code)



How can we identify dead code?

With a new data flow analysis called **liveness analysis**

Liveness analysis

A variable is live at a particular point in the program
if its value at that point will be used in the future (dead, otherwise)

- To compute liveness at a given point of a CFG,
  we need to look at instructions that will be executed next

- How to use variable liveness information for eliminating dead-code?
  - Dead-code:
    a side-effect free instruction i that defines a variable
    that is dead just after i

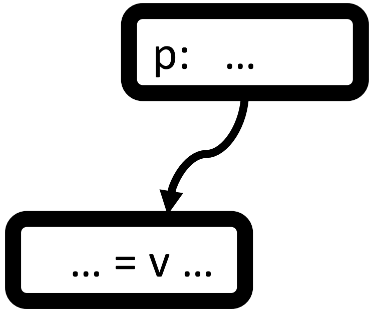
```
i-1: b = 42
i  : a = 5
i+1: return b
```

Liveness analysis

A variable is live at a particular point in the program
if its value at that point will be used in the future (dead, otherwise)

- Another use: register allocation
- A program contains an unbounded number of variables
  - Must execute on a machine with a bounded number of registers
  - Two variables can use the same register
    if they are never in use at the same time
- CS 322 Compiler Construction

Liveness analysis



A variable v is live at a given point of a program p if

- Exist a directed path from p to a use of v and
- that path does not contain any definition of v

- Is liveness data-flow analysis forward or backward?
  - Liveness flows backwards through the CFG, because the behavior at future nodes determines liveness at a given node

IN[*s*] = *fs*(OUT[*s*])

- What are the elements in data flow values? variables

GEN[i]= variables used by i KILL[i]= variable defined by i

```
i: a = 5
…
j: a = v + 1
…
k: x = a + 1
```

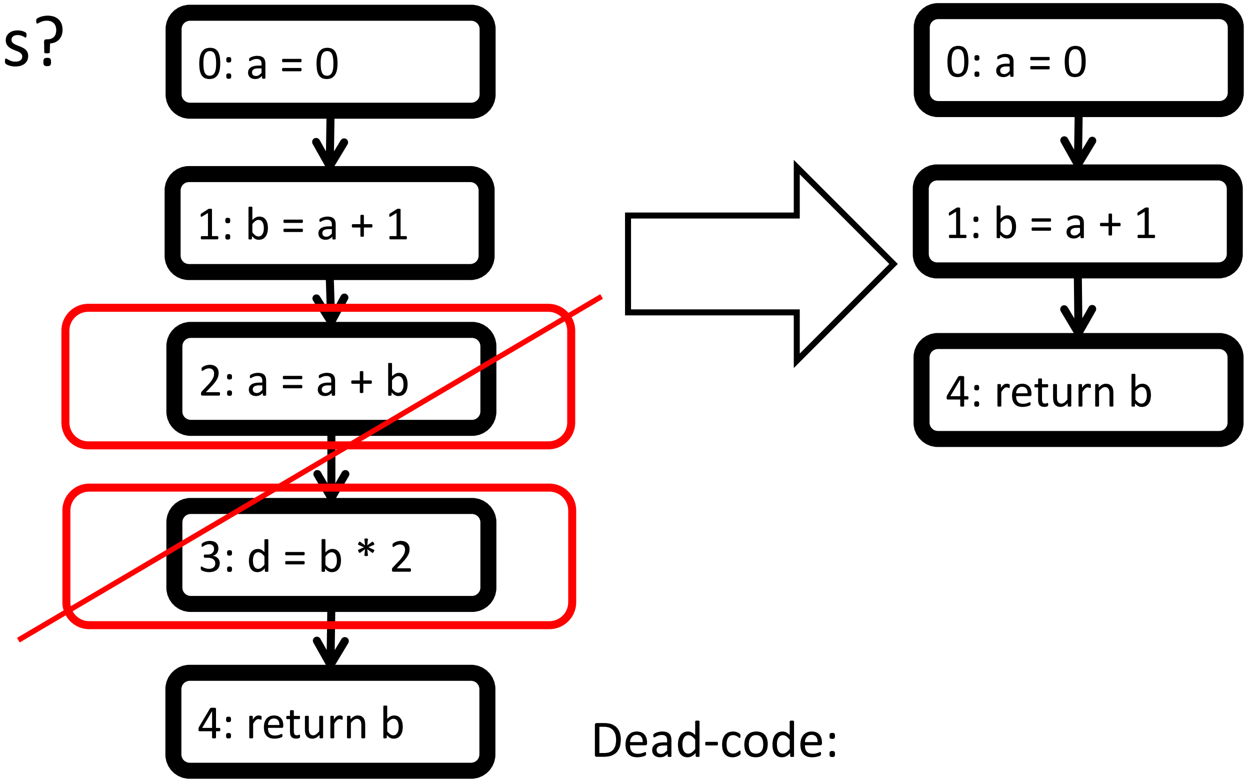
$IN[i] = GEN[i] \cup (OUT[i] - KILL[i])$

$OUT[i] = \cup_{s \text{ a successor of } i} IN[s]$

Example of variable liveness and dead-code elimination

What are in IN/OUT sets?

IN[0] = {}
OUT[0] = {a}
IN[1] = {a}
OUT[1] = {a, b}
IN[2] = {a, b}
OUT[2] = {b}
IN[3] = {b}
OUT[3] = {b}
IN[4] = {b}
OUT[4] = {}



Dead-code:
a side-effect free instruction i that defines a variable that is dead just after i

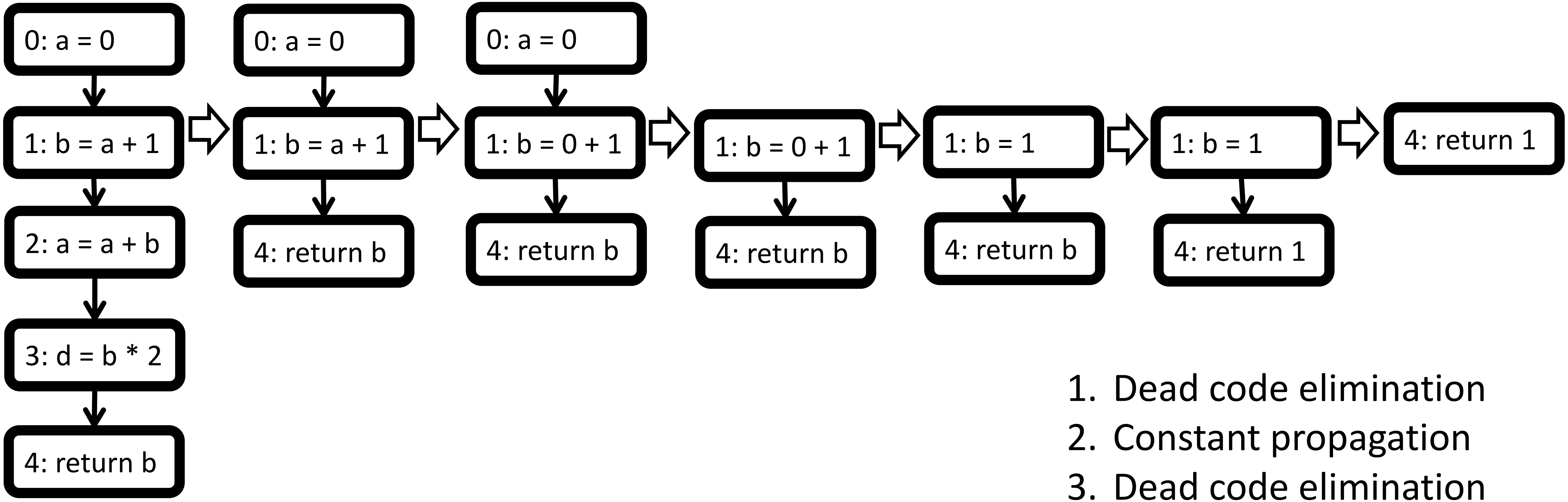
Is there dead-code?

Creating opportunities

- So far we saw
  - Dead code elimination
  - Constant propagation
  - Copy propagation

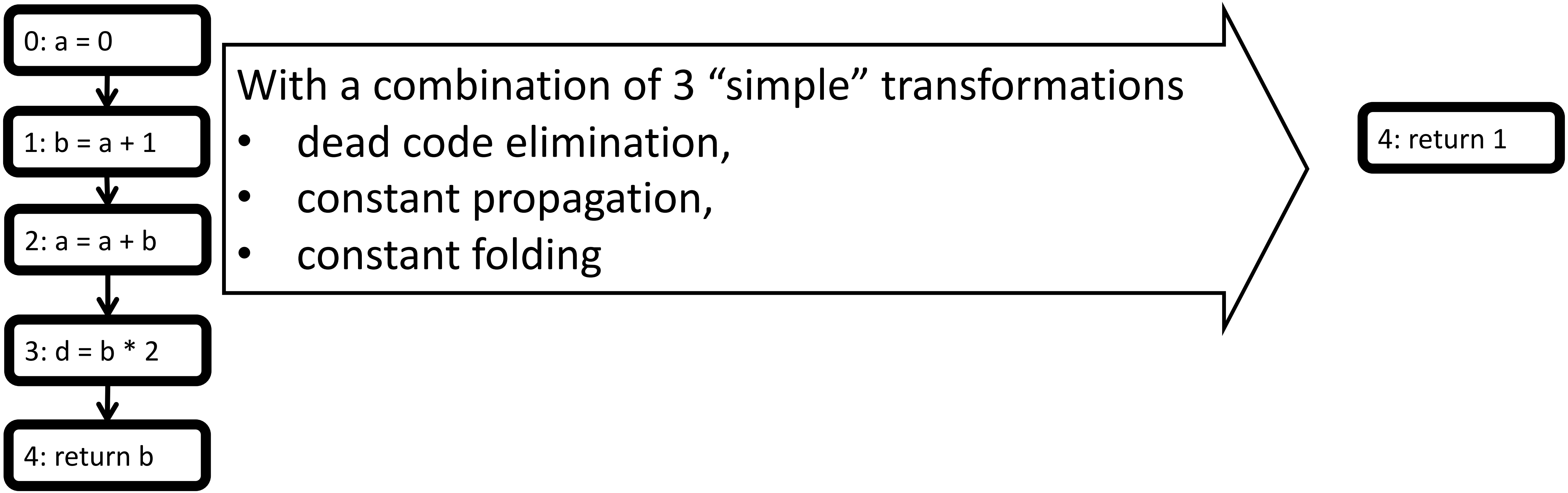
- They might look simple,
  but they can already optimize the code in interesting ways
  - Applying one often creates
    new optimization opportunities to the rest

Example of variable liveness and dead-code elimination



1. Dead code elimination
2. Constant propagation
3. Dead code elimination
4. Constant folding
5. Constant propagation
6. Dead code elimination

Example of variable liveness and dead-code elimination



Are there more transformations to remove more redundancy?

Common sub-expression elimination: problem definition

Given a program, we would like to know
for every point in the program,
which expressions are available

```
1: y = x + 3
2: b = x + 3
```

⇒

```
1: y = x + 3
2: b = y
```

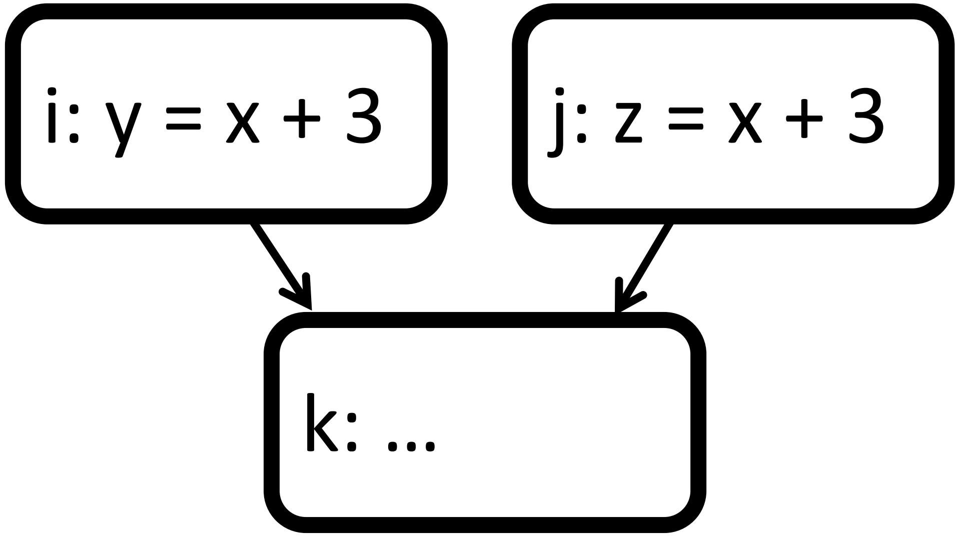
Do you see any redundancy?

Available expressions

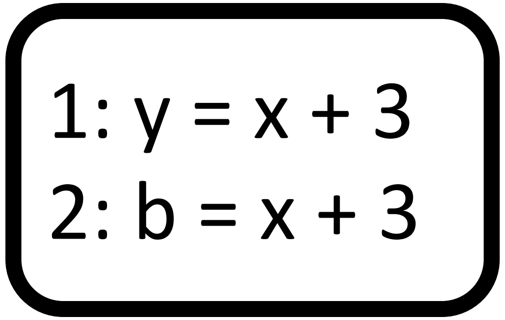
- What are the elements in data-flow sets?
- GEN and KILL?
- Forward or backward?
- IN and OUT?

$IN[i] = \cap_{p \text{ a predecessor of } i} OUT[p]$

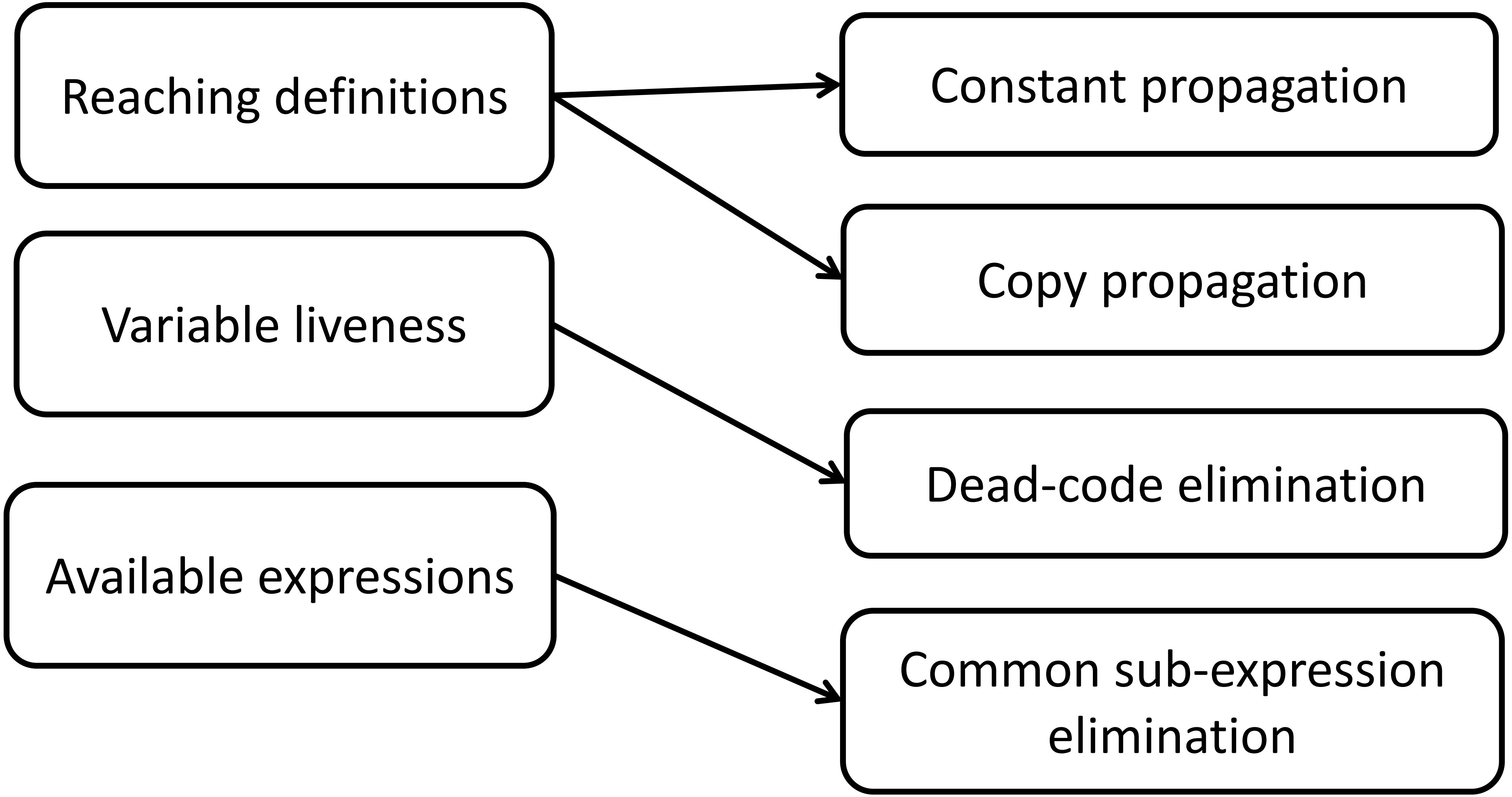
$OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$



- How to use available expressions for eliminating redundant code?

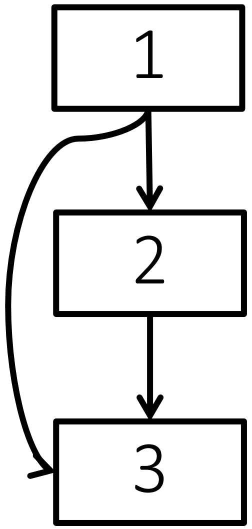


So far ...

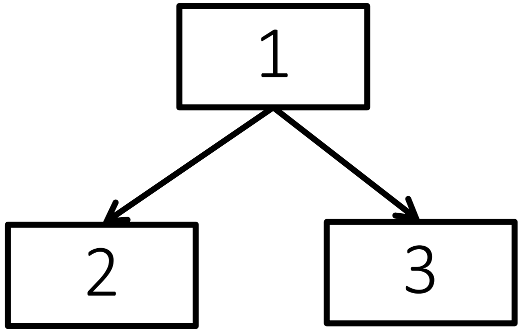

Reaching definitions
Variable liveness
Available expressions
Constant propagation
Copy propagation
Dead-code elimination
Common sub-expression elimination

Dominators

Definition: a basic block *d* dominates *n* in a CFG (*d dom n*) if every control flow from the start node to *n* goes through *d.* Every node dominates itself.



CFG



Dominators

What are the elements for data flow values?

GEN ? KILL ? IN ? OUT? (1 point)

Outline

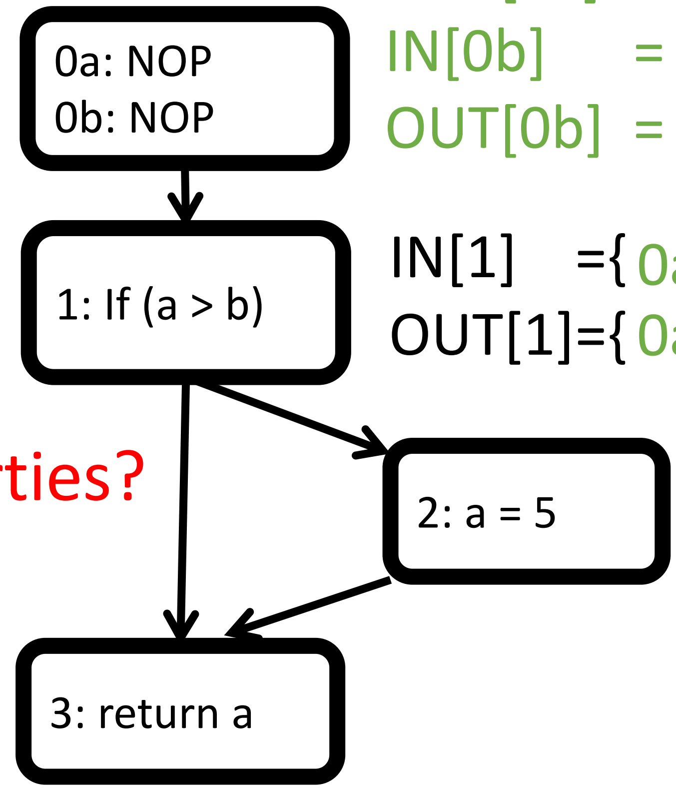
- Reaching definition and constant propagation
- More DFAs and related transformations
- DFAs without assumptions
- Other uses of DFA

What about function parameters?

... let's compute the reaching definition analysis

Which information is missing?

```
int myFunction (int a, int b){
   if (a > b){
        a = 5;
   }
    return a;
}
```



IN[0a] = { }
OUT[0a] = {0a}
IN[0b] = {0a}
OUT[0b] = {0a,0b}

IN[1] ={ 0a,0b}
OUT[1]={ 0a,0b}

IN[2] ={ 0a,0b}
OUT[2]={2, 0b }

IN[3] = {2,0a,0b}

Can we exploit SSA properties?

CP algorithm replaces "a" with "5" in instruction 3!

What about function parameters?

- But you didn't have to deal with this problem in your assignments so far
- Why?

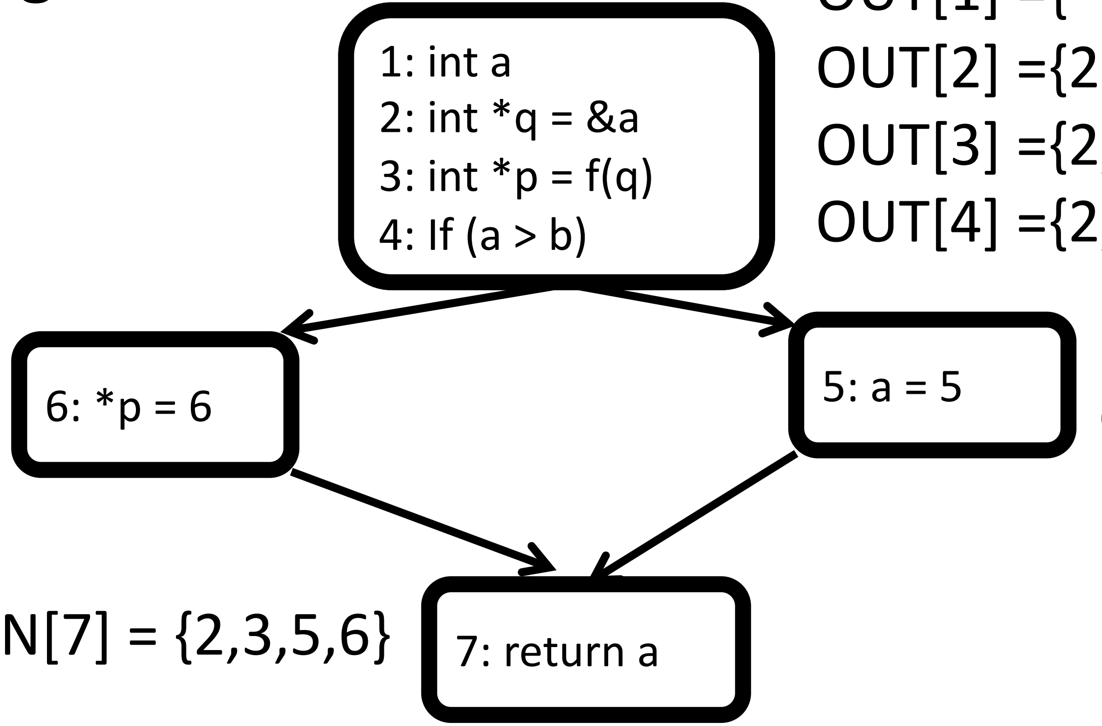
4. A C variable that includes a reference to a CAT variable does not escape the C function where it has been declared.

What about escaped variables?

… let's compute the reaching definition analysis

Which information is missing?

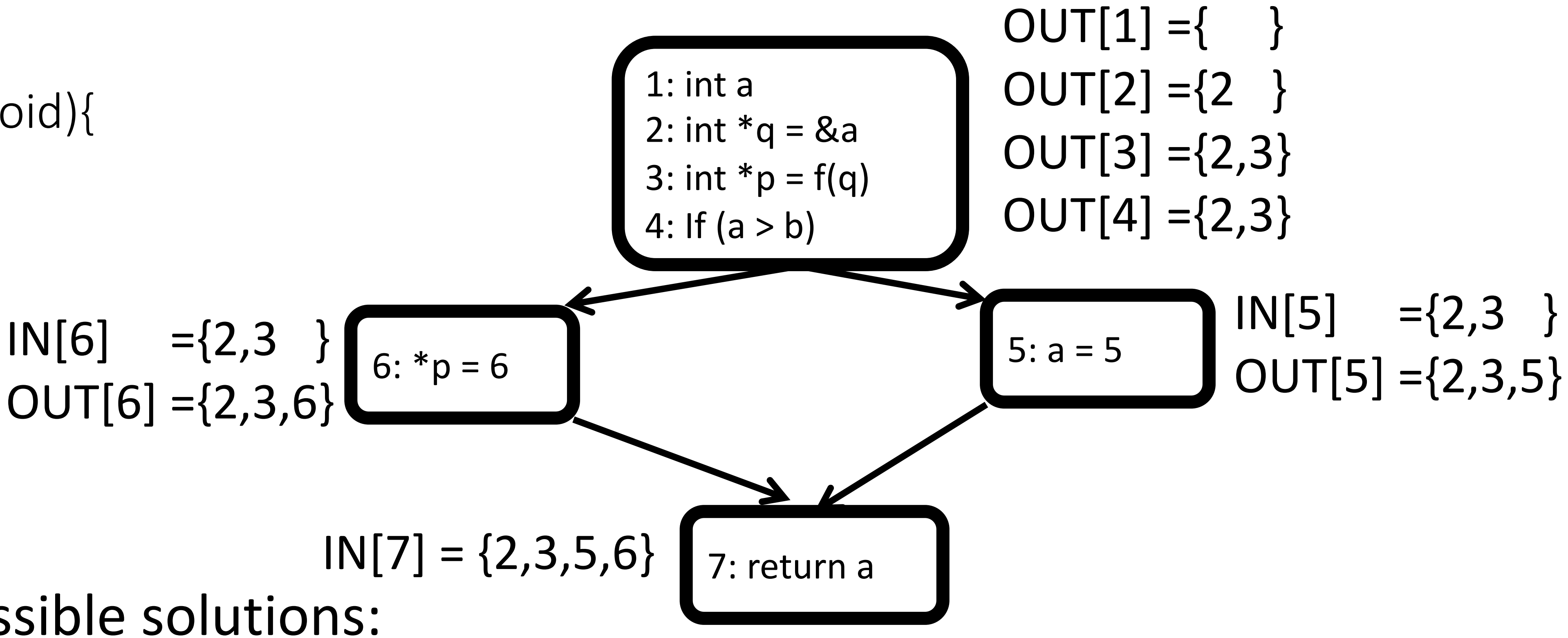
```
int myFunction (void){
  int a;
  int *p = f(&a);
  if (a > b){
      a = 5;
  } else {
      *p = 6;
  }
  return a;
}
```



CP algorithm replaces "a" with "5" in instruction 7!

What about escaped variables?

```
int myFunction (void){
  int a;
  int *p = f(&a);
  if (a > b){
     a = 5;
  } else {
     *p = 6;
  }
  return a;
}
```



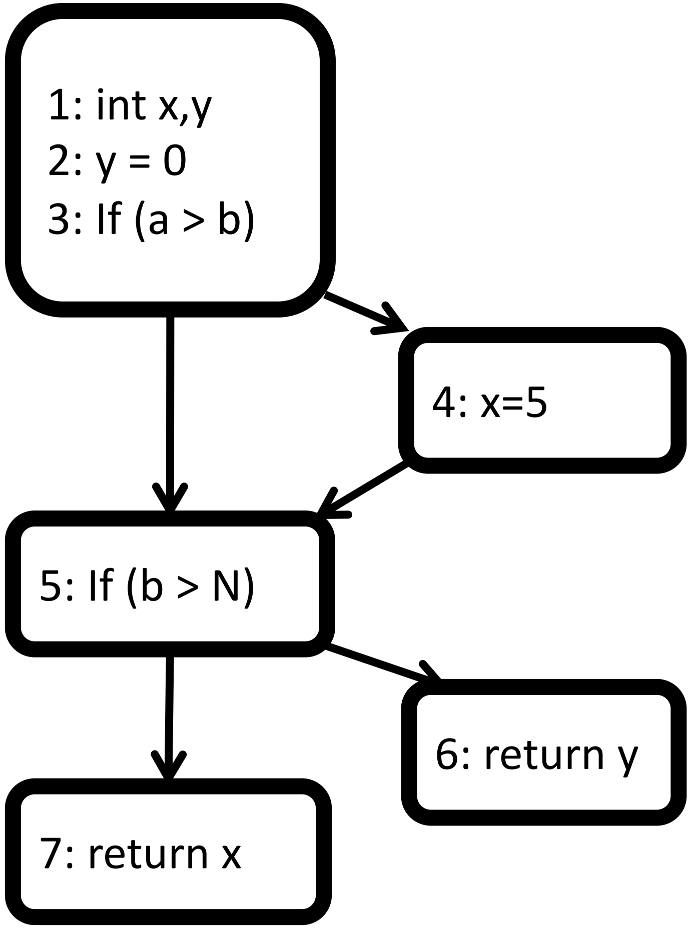
Possible solutions:

- Simple = skip escaped variables in CP
- Advanced = analyze how the memory is modified via pointers

Outline

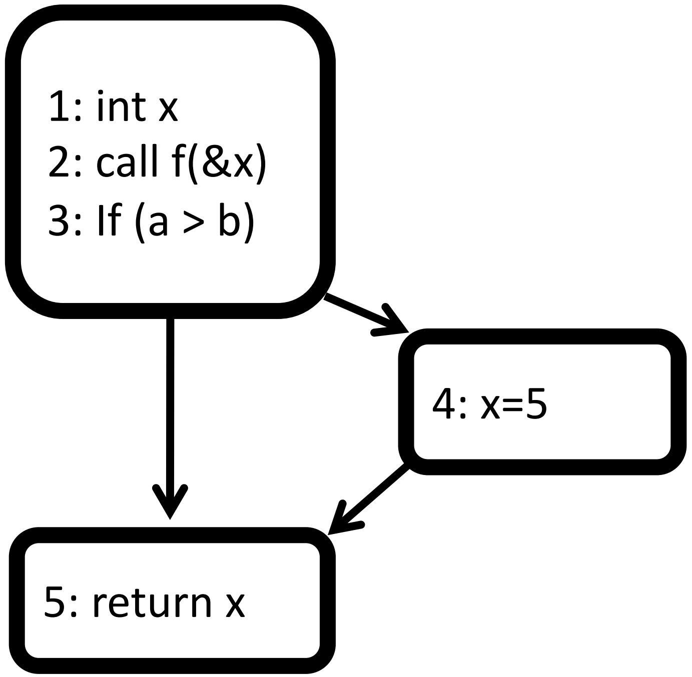
- Reaching definition and constant propagation
- More DFAs and related transformations
- DFAs without assumptions
- Other uses of DFA

Identifying software bugs



- “x” can be undefined at instruction 7
- Can we design an analysis to identify this problem and notify a developer about this bug?
- Let’s define precisely the problem
  - Conservativeness
- What are the data flow values?
- GEN[i] = ?
- KILL[i] = ?
- IN[i] and OUT[i] ?

Identifying software bugs (2)



- What about now?

- Let's define precisely the problem
  - Conservativeness
  - Warnings vs. errors

Data-flow analysis: food for thought

- Correctness: is the answer ALWAYS correct?
- Meaning: what is exactly the meaning of the answer?
- Precision: how good is the answer?
- Convergence:
  - Will the analysis ALWAYS terminate?
  - Under what conditions does the iterative algorithm converge?
- Speed: how long does it take to converge in the worst case?