code analysis *and* transformation

DFA Part 2

Simone Campanoni
simone.campanoni@northwestern.edu

Outline

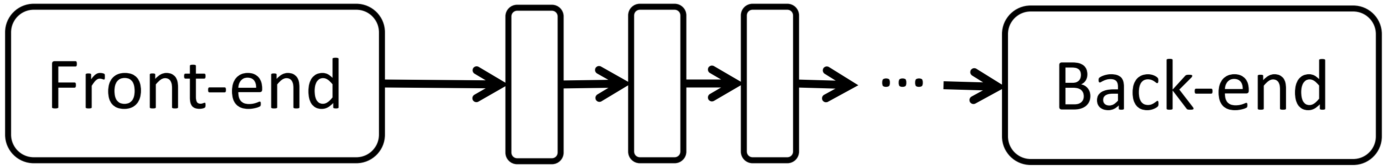
- More DFAs and related transformations
- DFAs without assumptions
- Other uses of DFA
- DFA implementation

Thinking about what constant propagation does

- 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



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.

```
1: x = y
2: a = 5
3: b = x + 3
```

Copy propagation →

```
1: x = y
2: a = 5
3: b = y + 3
```

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

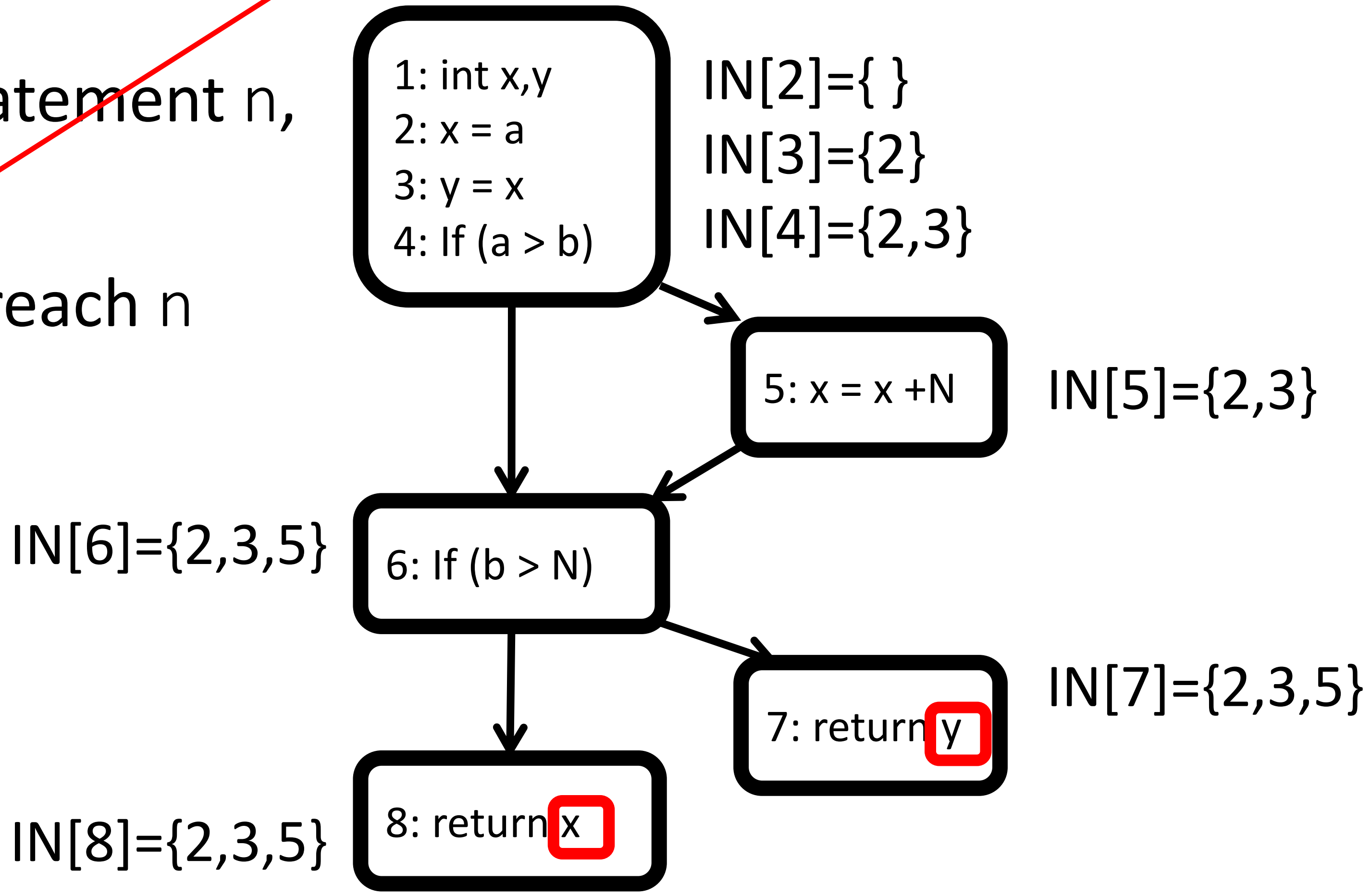
- For a use of variable v in statement n,

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)
- 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

- Beyond reaching definition:
  Now we are going to see other common DFAs

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)

1: y = …
2: x = y
3: return x

Copy propagation

1: y = …
~~2: x = y~~
3: return y

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

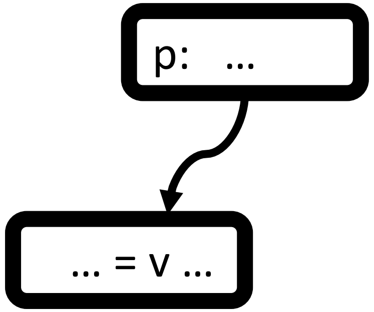
```
i-1: b = 42
i  : a = 5
i+1: return b
```

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
```

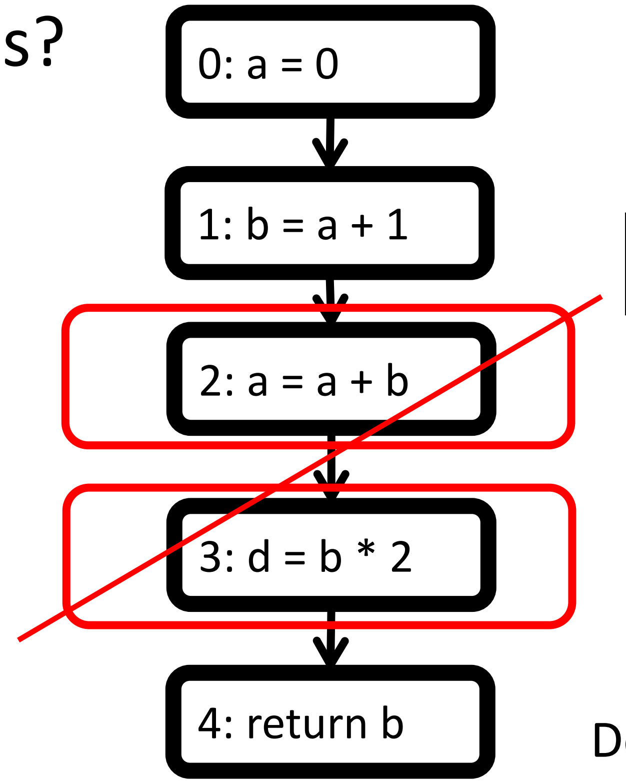
IN[i] = GEN[i] ∪(OUT[i] – KILL[i])

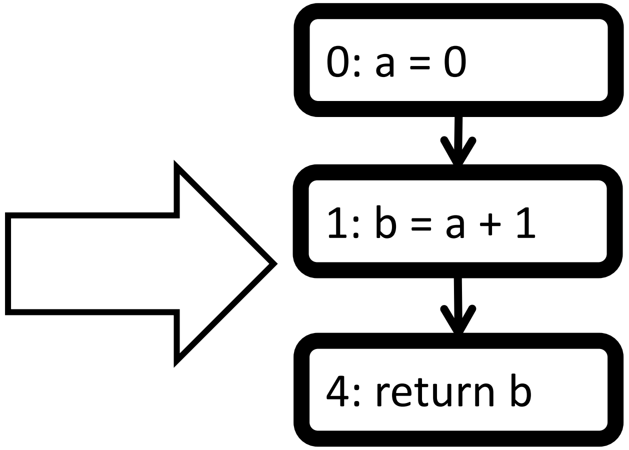
OUT[i] = $\cup_{s \text{ a successor of } i}$ IN[s]

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

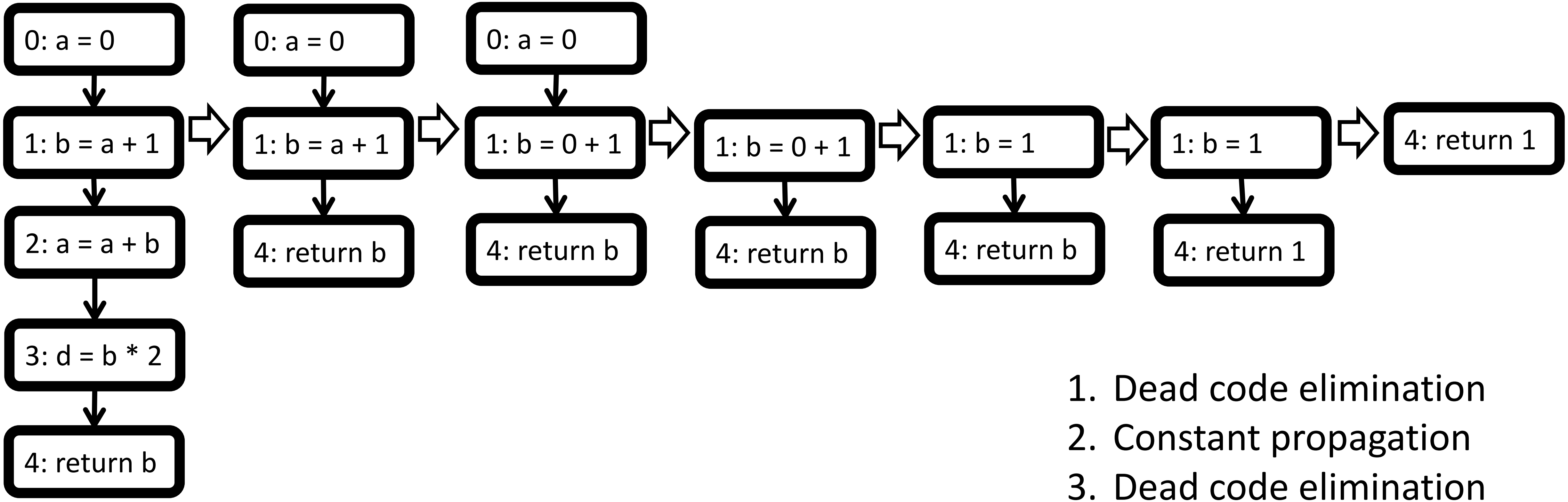
Is there dead-code?

Creating opportunities

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

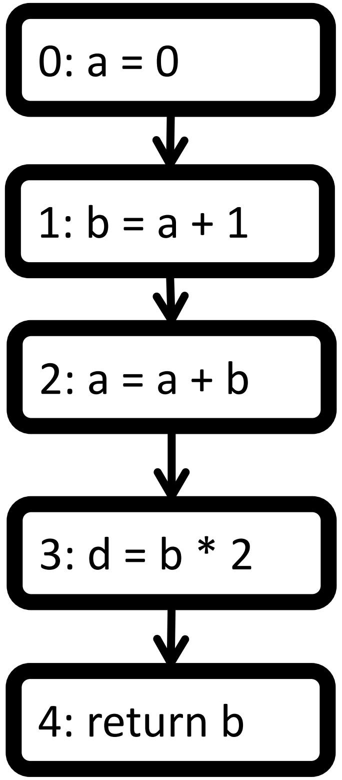
- They might look simple,
  but they can already optimize the code in interesting ways
  - Applying one often creates
    new optimization opportunities to the rest

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



With a combination of 3 “simple” transformations
- dead code elimination,
- constant propagation,
- constant folding

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
```

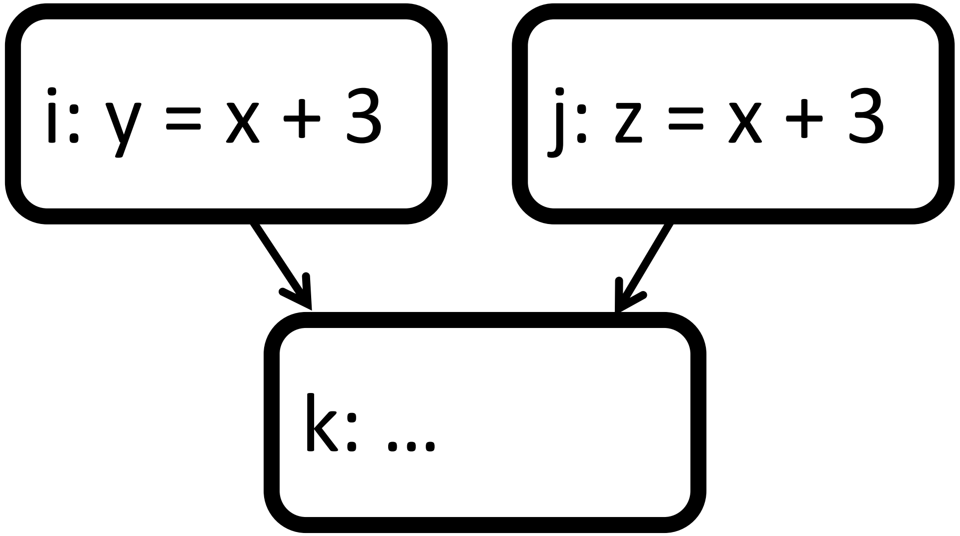
Do you see any redundancy?

Available expressions

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

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

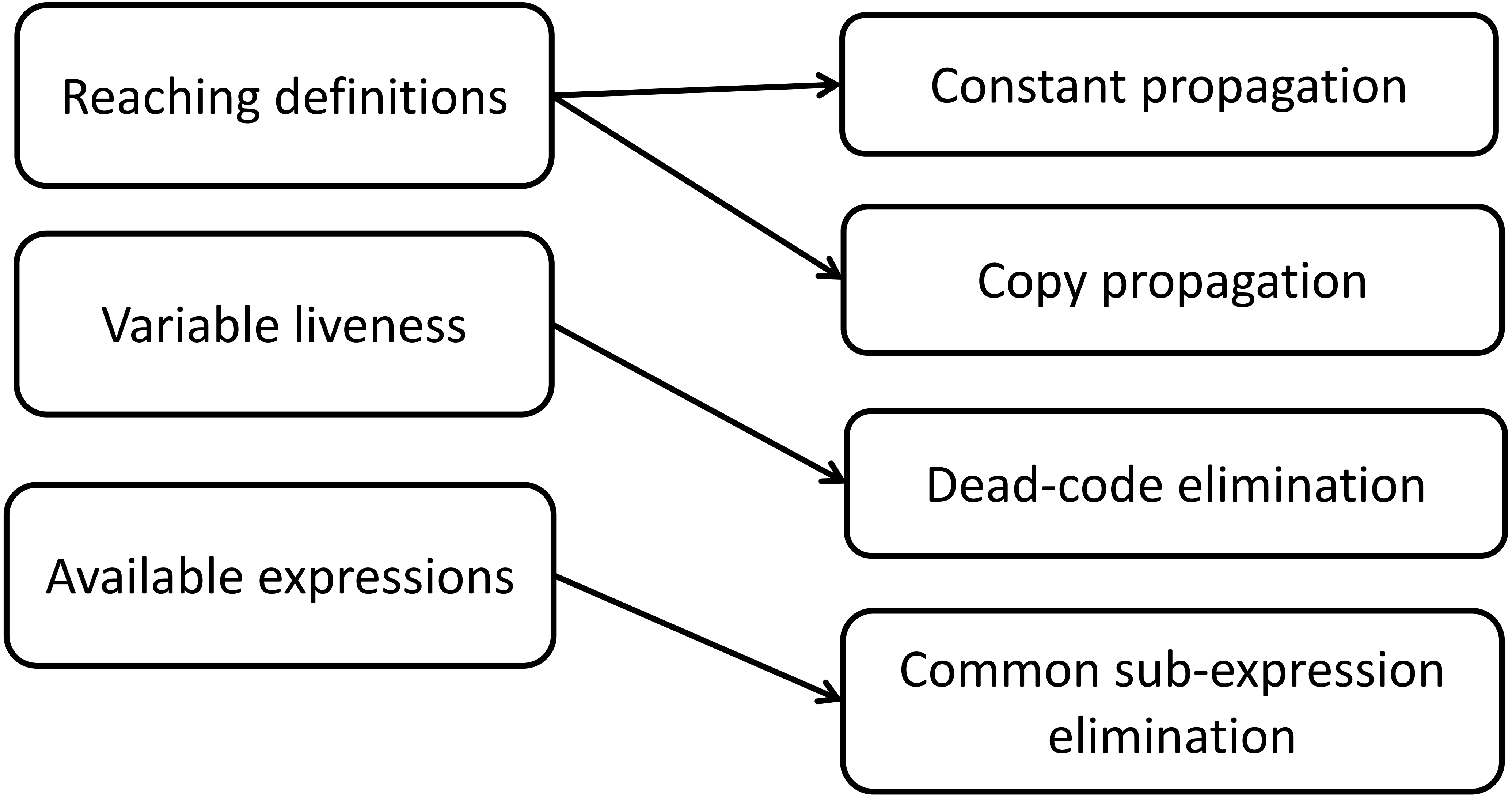
$OUT[i] = GEN[i] \cup (IN[i] - KILL[i])$



- How to use available expressions for eliminating redundant code?

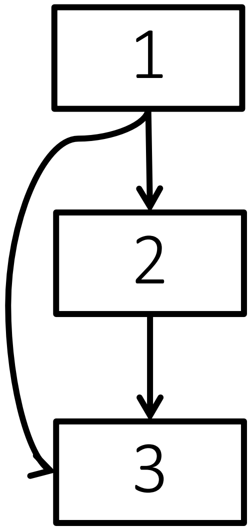
```
1: y = x + 3
2: b = x + 3
```

So far …

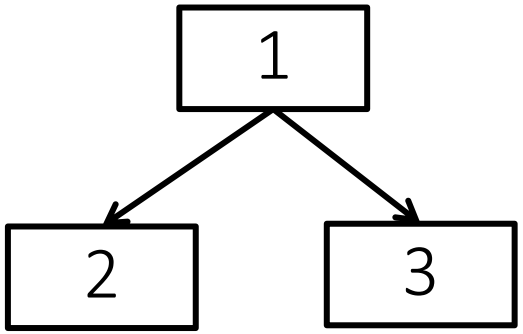

Reaching definitions
Variable liveness
Available expressions
Constant propagation
Copy propagation
Dead-code elimination
Common sub-expression elimination

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

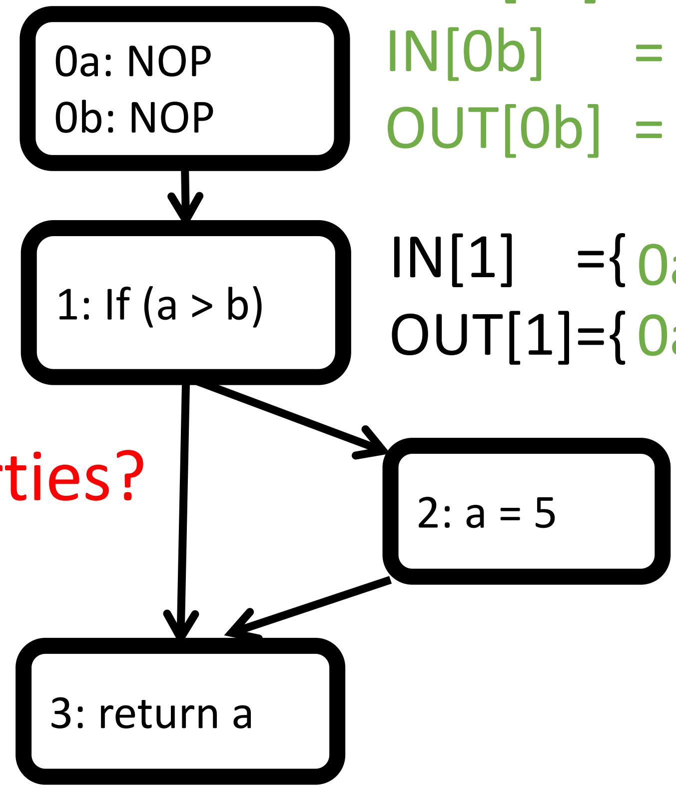
- More DFAs and related transformations
- DFAs without assumptions
- Other uses of DFA
- DFA implementation

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;
}
```

Can we exploit SSA properties?



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}

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?

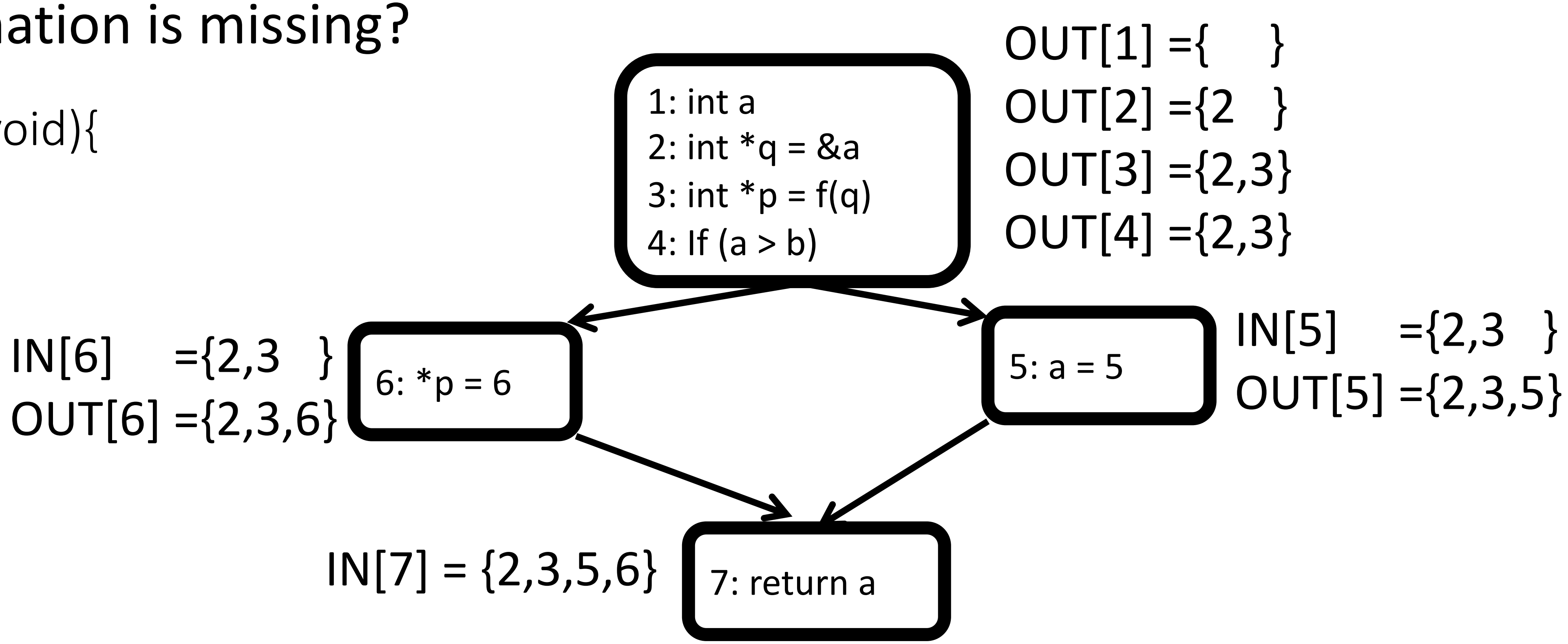
2. A C variable that includes a reference to a CAT variable cannot be given as argument to a call to a function.

What about escaped variables?

… let's compute the reaching definition analysis

Which information is missing?

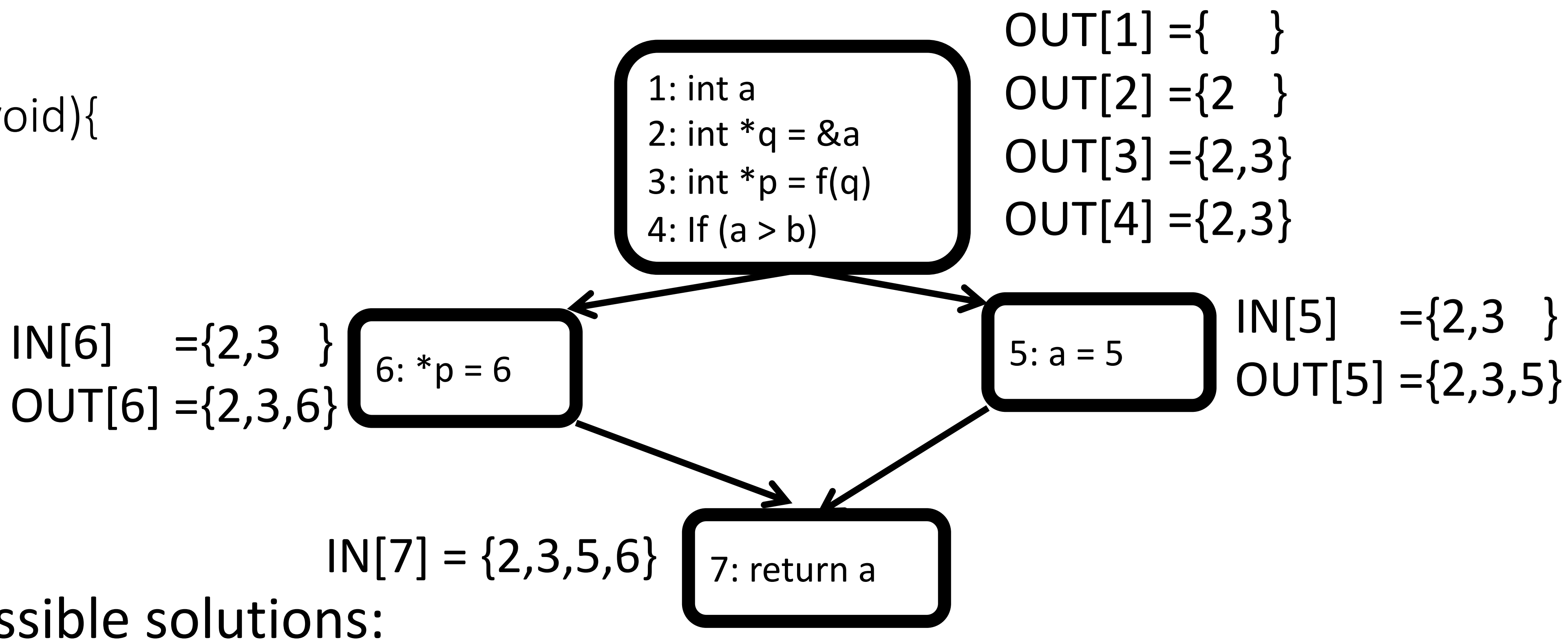
```
int myFunction (void){
  int a;
  int *p = f(&a);
  if (a > b){
      a = 5;
  } else {
      *p = 6;
  }
  return a;
}
```



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;
}
```



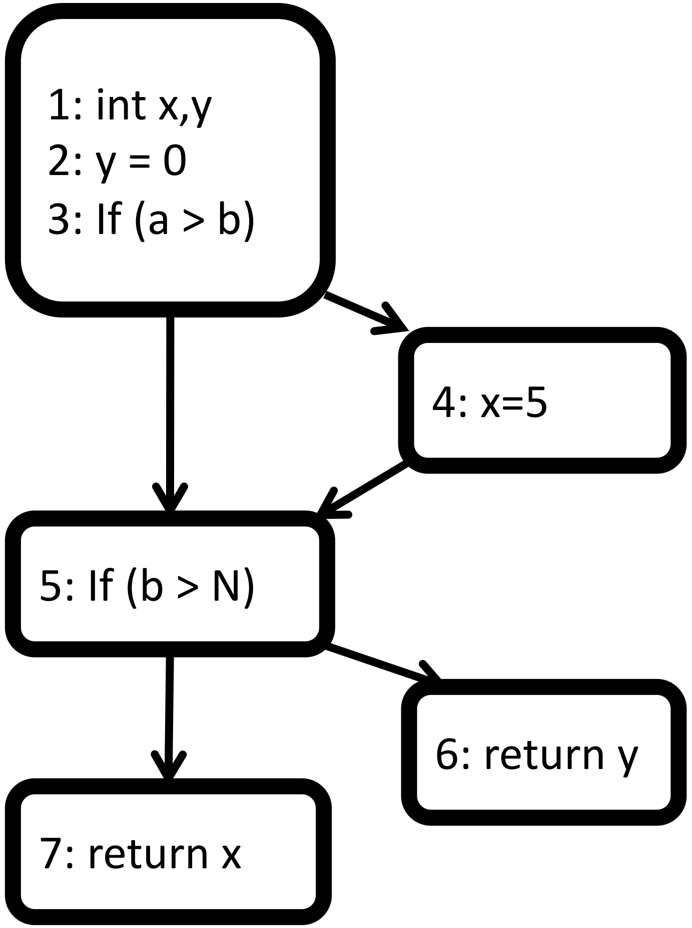
Possible solutions:

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

Outline

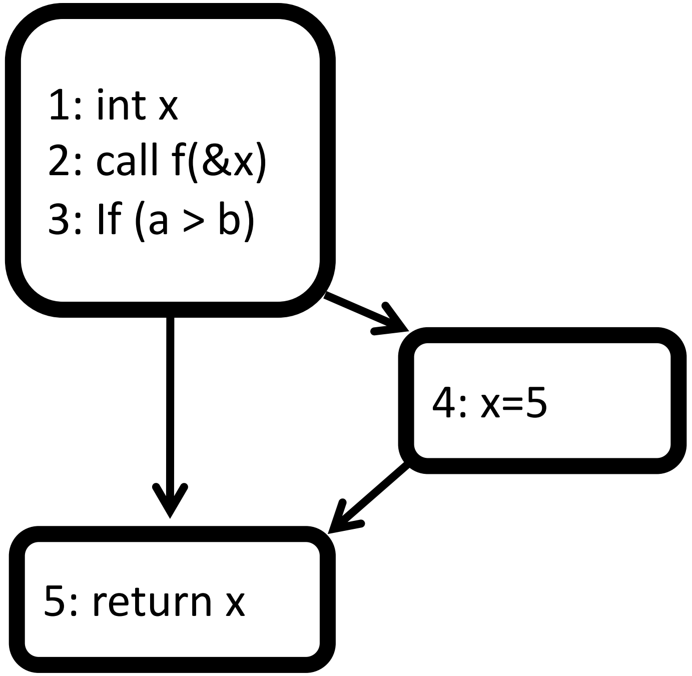
- More DFAs and related transformations
- DFAs without assumptions
- Other uses of DFA
- DFA implementation

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

Outline

- More DFAs and related transformations
- DFAs without assumptions
- Other uses of DFA
- DFA implementation

Forward DFA

for (each instruction *i*) IN[i] = OUT[i] = { };

do {

  for (each instruction *i*) {

    IN[*i*] = $fs_{p \text{ a predecessor of } i}$ (OUT[*p*])

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

  }

} while (changes to any OUT occur)

Backward DFA

for (each instruction *i*) IN[i] = OUT[i] = { };

do {

  for (each instruction *i*) {

    OUT[*i*] = $fs_{s \text{ a successor of } i}$ (IN[s])

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

  }

} while (changes to any IN   occur)

Now that we know DFAs and how to compute them,

let us look at how to reduce the computation time to compute them

Implementation aspects

for (each instruction *i*) IN[i] = OUT[i] = { };

do {

  for (each instruction *i*) {

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

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

  }

} while (changes to any OUT occur)

Hot code

- Memory representation of data flow values
  - Operations performed on them
  - What is an element in a set?

Optimization 1: bit-set

for (each instruction i) IN[i] = OUT[i] = { };

do {

for (each instruction i) {

IN[i] = $\boxed{\cup}_{p \text{ a predecessor of } i}$ OUT[p];

OUT[i] = GEN[i] $\boxed{\cup}$ (IN[i] $\boxed{-}$ KILL[i]);

}

} while (changes to any OUT occur)

Optimization 1: bit-sets

- Assign a bit to each element that might be in the set
  - Union: bitwise OR
  - Intersection: bitwise AND
  - Subtraction: bitwise NEGATE and AND

- Fast implementation
  - 64 elements packed to each word on today's commodity processors
  - AND and OR are single machine code instructions (single cycle latency)

llvm::BitVector llvm::SmallBitVector llvm::SparseBitVector

Can we further optimize the analysis?

for (each instruction i) IN[i] = OUT[i] = { };

do {

  for (each instruction i) {

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

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

  }

} while (changes to any OUT occur)

... that's a lot of iterations repeated for each while iteration

Are they all necessary for every while iteration?

```
do {
  for (each instruction i) {
    IN[i] = ∪_{p a predecessor of i} OUT[p];
    OUT[i] = GEN[i] ∪ (IN[i] – KILL[i]);
  }
} while (changes to any OUT occur)
```

First while-iteration

IN[i], OUT[i] — *Changed*

IN[j], OUT[j] — *Not changed*

IN[l], OUT[l]

Second while-iteration

IN[i], OUT[i]

IN[j], OUT[j]

IN[l], OUT[l]

Third while-iteration

IN[i], OUT[i]

IN[j], OUT[j]

IN[l], OUT[l]

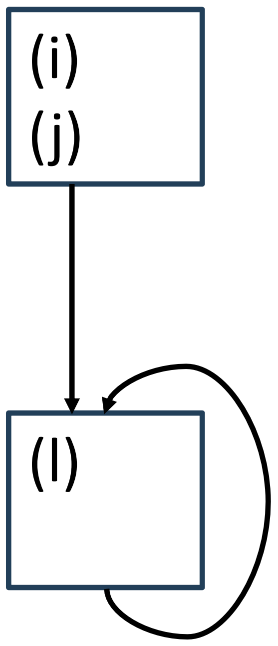
Are these necessary?

Forth while-iteration

IN[i], OUT[i]

IN[j], OUT[j]

IN[l], OUT[l]



Optimization 2: work list

```
for (each instruction i)  IN[i] = OUT[i] = { };
do {
 for (each instruction i) {
   IN[i] = ∪p a predecessor of i OUT[p];
   OUT[i] = GEN[i] ∪ (IN[i] – KILL[i]);
 }
} while (changes to any OUT occur)
```

Optimization 2: work list

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY)  OUT[i] = { };
workList = all instructions
while (workList isn't empty)
  i = pick and remove an instruction from workList
  oldOUT = OUT[i]
  IN[i] = ∪_{p a predecessor of i} OUT[p];
  OUT[i]= GEN[i] ∪ (IN[i] – KILL[i]);
  if (oldOut != OUT[i]) workList = workList U {all successors of i}
}
```

First while-iteration IN[i], OUT[i] *Changed*

IN[j], OUT[j] *Not changed*

IN[l], OUT[l]

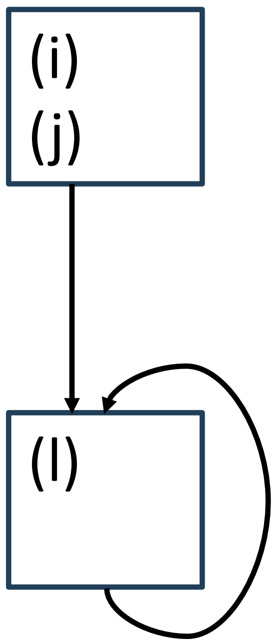
Second while-iteration IN[i], OUT[i]

IN[j], OUT[j]

IN[l], OUT[l]

Third while-iteration IN[l], OUT[l]

Forth while-iteration IN[l], OUT[l]



Can we further optimize it?

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY)  OUT[i] = { };
workList = all instructions
while (workList isn't empty)
  i = pick and remove an instruction from workList
  oldOUT = OUT[i]
  IN[i] = ∪_{p a predecessor of i} OUT[p];
  OUT[i]= GEN[i] ∪ (IN[i] – KILL[i]);
  if (oldOut != OUT[i]) workList = workList U {all successors of i}
}
```

First while-iteration IN[i], OUT[i]
IN[j], OUT[j]
IN[l], OUT[l]

Second while-iteration IN[i], OUT[i]
IN[j], OUT[j]
IN[l], OUT[l]

Third while-iteration IN[l], OUT[l]

Forth while-iteration IN[l], OUT[l]

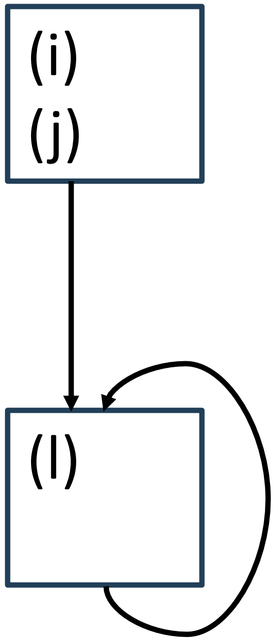
First while-iteration IN[l], OUT[l]
IN[j], OUT[j]
IN[i], OUT[i]

Second while-iteration IN[l], OUT[l]
IN[j], OUT[j]
IN[i], OUT[i]

Third while-iteration IN[l], OUT[l]
IN[j], OUT[j]
IN[i], OUT[i]

Forth while-iteration IN[l], OUT[l]

Fifth while-iteration IN[l], OUT[l]



Changed
Not changed

Optimization 3: evaluation order

```
OUT[ENTRY] = { };
for (each instruction i other than ENTRY)  OUT[i] = { };
workList = all instructions
while (workList isn't empty)
  i = pick and remove an instruction from workList
  oldOUT = OUT[i]
  IN[i] = U_{p a predecessor of i} OUT[p];
  OUT[i]= GEN[i] ∪ (IN[i] – KILL[i]);
  if (oldOut != OUT[i]) workList = workList U {all successors of i}
}
```

Optimization 4: basic blocks

for (each instruction *i*) IN[i] = OUT[i] = { };

do {

  for (each instruction *i*) {

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

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

  }

} while (changes to any OUT occur)

Is this always necessary ?

Optimization 4: basic blocks

for (each basic block B) IN[B] = OUT[B] = { };

do {

for (each basic block B) {

$IN[B] = \cup_{P \text{ a predecessor of } B} OUT[P];$

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

}

} while (changes to any OUT occur)

Contains **all** definitions in block B
that are **visible** immediately after B

```
i0: v1 = 5
i1: v2 = v1 + 1
i2: v1 = 42
```

GEN[B]={i1,i2}

i1 is not visible outside B

Optimization 4: basic blocks

for (each basic block B) IN[B] = OUT[B] = { };

do {

  for (each basic block B) {

    $IN[B] = \cup_{P \text{ a predecessor of } B} OUT[P]$;

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

  }

} while (changes to any OUT occur)

Suggestion: if you are going to implement these optimizations, then either

- skip this one or
- keep it to be the last one

Contains **all** definitions killed by instructions in block B

Contains **all** definitions in block B that are **visible** immediately after B

Optimization 4: basic blocks

```
for (each basic block B)  IN[B] = OUT[B] = { };
do {
  for (each basic block B) {
    IN[B] = ∪_{P a predecessor of B} OUT[P];
    OUT[B] = GEN[B] ∪ (IN[B] – KILL[B]);
  }
} while (changes to any OUT occur)
… // propagate IN[B] through the instructions within B
  // without computing IN[B.first()] and OUT[B.last()]
  // because IN[B.first()] == IN[B]; OUT[B.last()] == OUT[B]
```

Optimization 4: basic blocks

```
… // propagate IN[B] through the instructions within B
f = B.first() ; IN[f] = IN[B];
OUT[f] = GEN[f] ∪ (IN[f] – KILL[f]);
t = f;
while (t != B.last()){
   tNext = t.next();
   IN[tNext] = OUT[t];
   OUT[tNext] = GEN[tNext] ∪ (IN[tNext] – KILL[tNext]);
   t = tNext;
}
```

$\boxed{OUT[B] = GEN[B] \cup (IN[B] - KILL[B]);}$

Optimization 4: basic blocks

```
f = B.first() ; IN[f] = IN[B];
if (f != B.last()) OUT[f] = GEN[f] ∪ (IN[f] – KILL[f]);
t = f;
while (t != B.last()){
   tNext = t.next();
   IN[tNext] = OUT[t];
   if (tNext != B.last()) OUT[tNext] = GEN[tNext] ∪ (IN[tNext] – KILL[tNext]);
   t = tNext;
}
```

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?

Always have faith in your ability

Success will come your way eventually

Best of luck!