CSc 553
Principles of Compilation

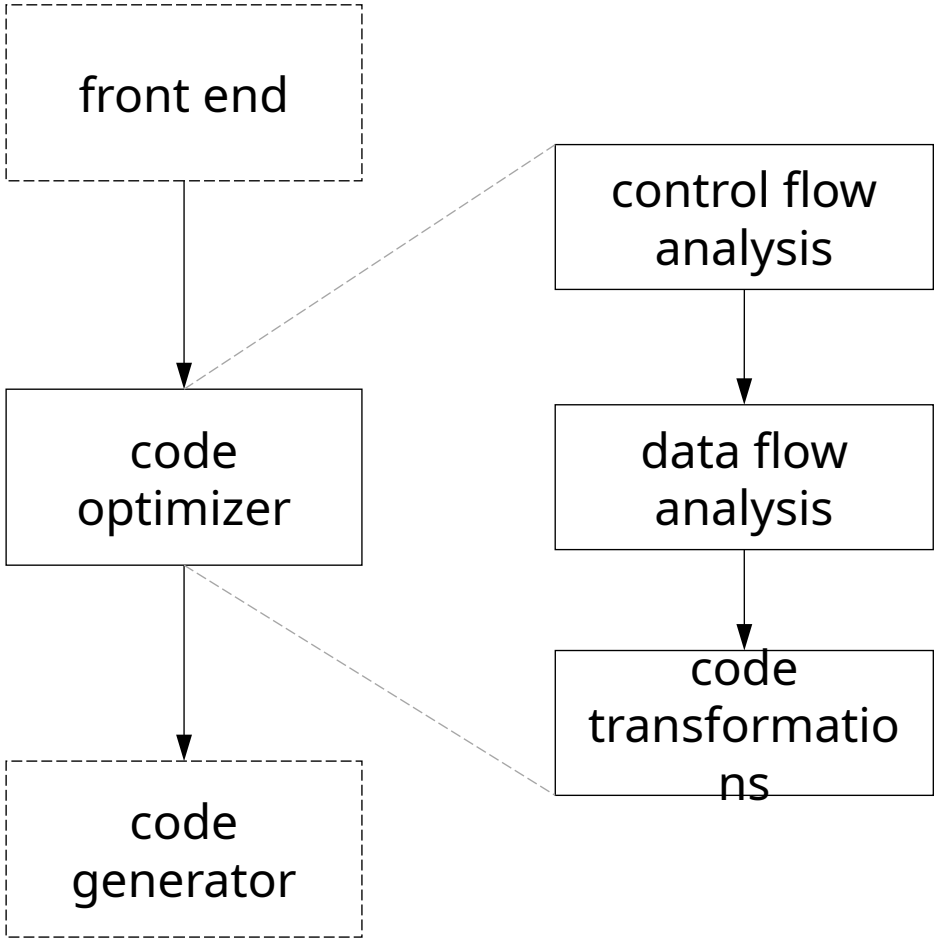
05. Program Analysis

Saumya Debray

The University of Arizona

Tucson, AZ 85721

Analysis and optimization: organization



Background

Background: Computability Theory

- Program analysis ≡ programs reasoning about programs
  - what is possible? what are the limits?

Background: Computability Theory

History:

- in the first half of the 20th century, a lot of researchers were trying to understand the capabilities and limits of mechanical computation
- different people approached these questions from different angles using different formalisms
  - Alan Turing: "automatic machines" (now called Turing machines)
  - Alonzo Church: Lambda calculus
  - Stephen Kleene: General recursive functions
  - ...

Background: Computability Theory

- These formalisms turned out to be computationally equivalent
  - each could simulate the others
- Consensus ["Church-Turing Thesis"]: These formalisms capture the notion of "algorithmically computable" functions
  - we can use these formalisms to study what can (and what cannot) be computed algorithmically

Undecidability

- Turing proved that some problems are *undecidable*
  - i.e.: there is no algorithm that will, for *all* instances of the problem:
    - correctly solve that problem instance; and
    - terminate
- This does not rule out:
  - algorithms that correctly solve *some* instances of that problem, and always terminate
  - computations that attempt to solve *all* instances of the problem, but sometimes don't terminate

Undecidability and program analysis

- **Rice's Theorem:** All nontrivial behavioral properties of programs are undecidable

  (a property is trivial if either all programs have it, or else no program has it)

- This means:

  - we can have program analyses that produce safe (but sometimes imprecise) analysis results and always terminate **

  - if we tried to always produce precise analysis results, the analysis would sometimes not terminate

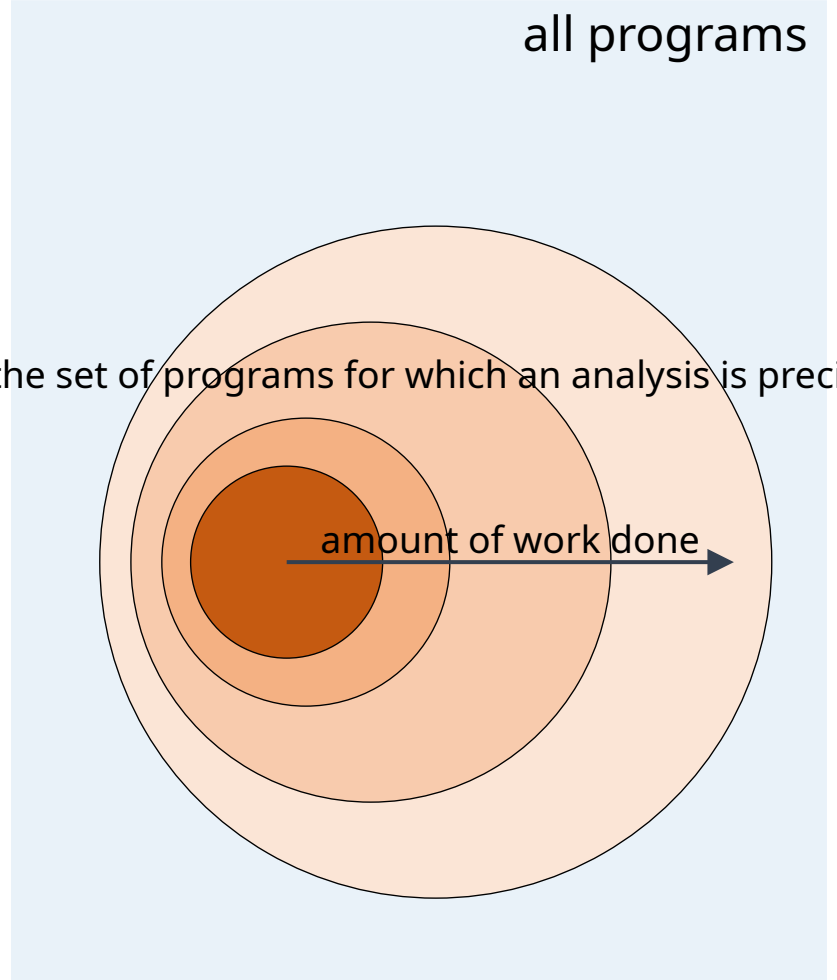
compilers *

* people hate it when the compiler goes into an infinite loop

** "safe" means that all possible executions are accounted for in the analysis results (however, imprecision arising from undecidability means that the analysis can also account for executions that cannot actually occur at runtime)

Program analysis and precision

- In general, we can expect precise results for some, but not all, input programs
- The more work an analysis does, the larger the set of inputs for which it can be precise



CONTROL FLOW ANALYSIS

Goals

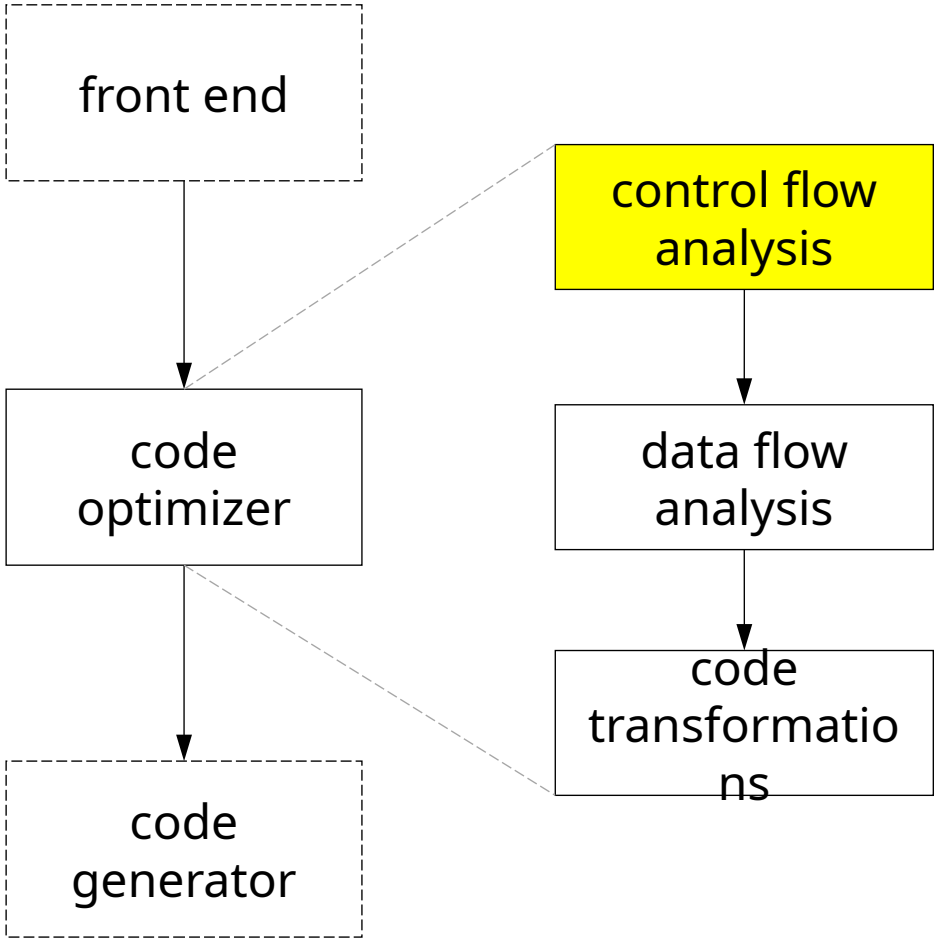
To obtain higher-level information about the possible control flow behaviors of the program, e.g.:

- *"which blocks are guaranteed to have been executed if control reaches some block B?"*
- *"which blocks are guaranteed to be executed once control reaches some block B?"*
- *"what is the loop structure of the code?"*

Criteria:

- must be "safe," i.e., must take into account all possible executions of the program

Analysis and optimization: organization



Dominators

Dominators

- Definition: A node d in a flow graph *G* dominates a node n (written "d dom n") iff every path from the entry node of *G* to n contains the node d.

- Facts:
  - every node dominates itself;
  - the "dom" relation is a partial order;
  - every node has a unique *immediate dominator*.

  ⇒ the dominator relationships in a flow graph form a tree (the "*dominator tree*.")

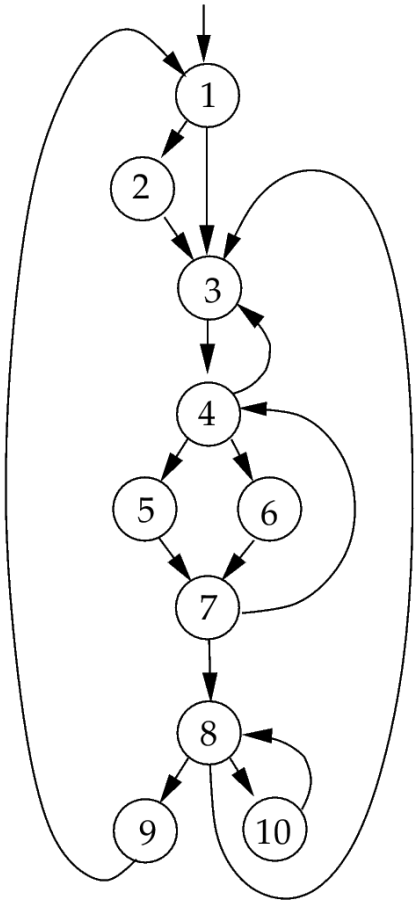
Immediate Dominator

x is an immediate dominator of y iff:

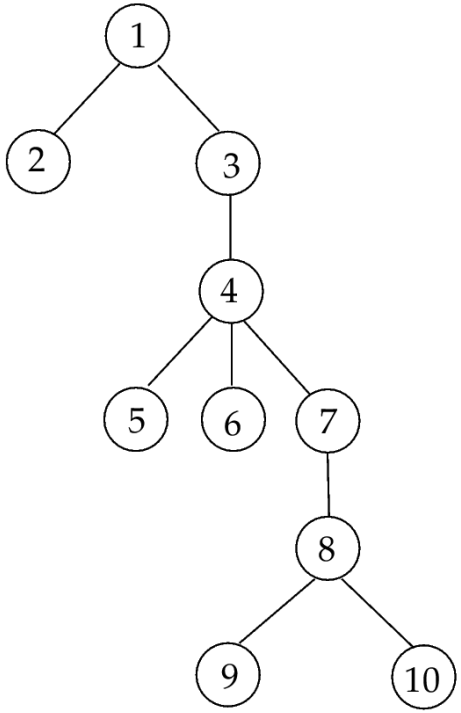
1) x dom y; and
2) there is no z (z != x) such that x dom z and z dom y

Dominator tree: Example

Control flow graph

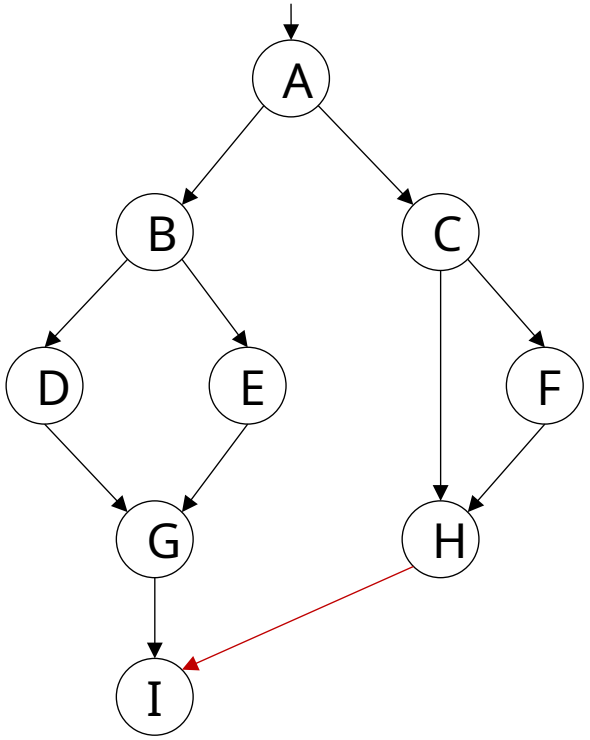


Dominator tree



EXERCISE

- What is the immediate dominator for:
  - F?
  - G?
  - H?
- What is the set of dominators for:
  - G?
  - H?
  - I?
- What happens to the set of dominators of node I if we add an edge H → I?



Finding dominators

Given: a flow graph *G* with set of nodes *N* and entry node n_0.

Algorithm:

1. for each node n, initialize $D(n) = \begin{cases} \{n_0\} & \text{if } n = n_0 \\ N & \text{otherwise} \end{cases}$
2. **repeat**:

   for each node $n \in N - \{n_0\}$ do:

   $$D(n) = \{n\} \cup (\bigcap_{p \in preds(n)} D(p))$$

   until there is no change to $D(n)$ for any node n.

Natural loops

Natural loops

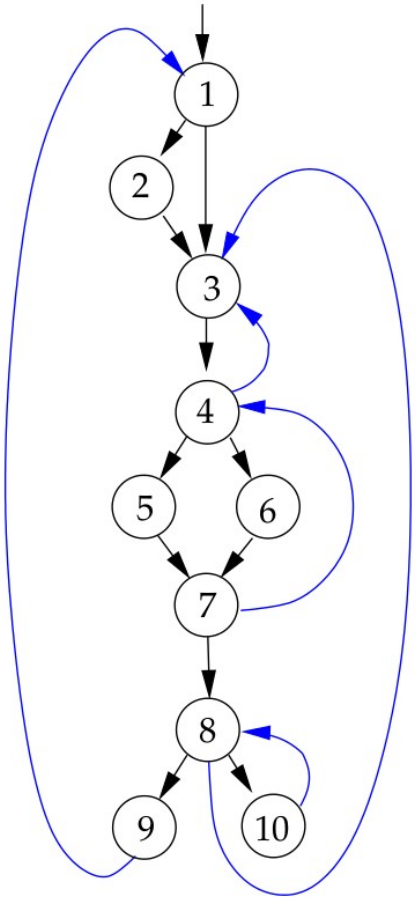
Definition: A loop in a flow graph is a set of nodes *N* satisfying:

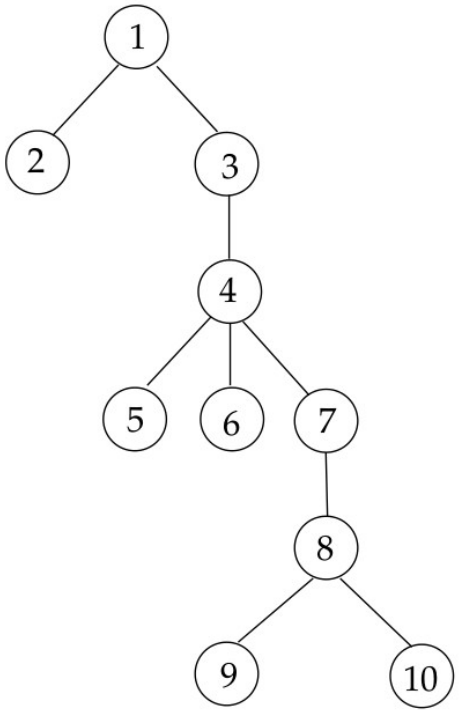
1. *N* has a single entry point (the “header”) that dominates every node in *N*; and
2. each node in *N* is reachable from every other node in *N*.

Back edges:

A back edge in a flow graph is an edge $a \rightarrow b$ where b dominates a.

Examples of back edges





back edges

Identifying natural loops

Problem:

Given a flow graph G and a back edge $e \equiv a \rightarrow b$ in G, find the natural loop associated with e

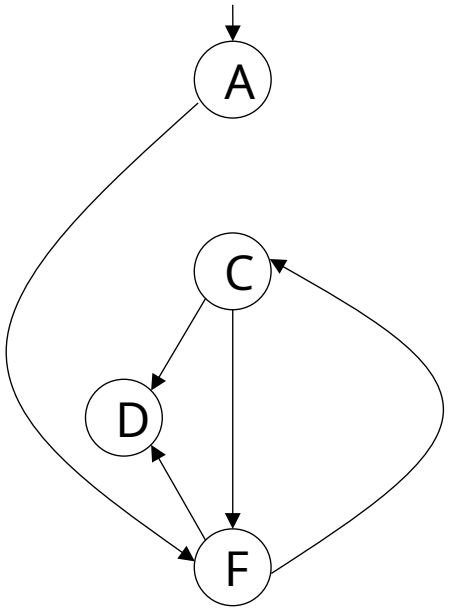
Algorithm:

```
stack = NULL; loop = { b };
insert(a, loop);
while stack not empty {
    m = pop(stack);
    for each predecessor p of m {
        insert(p, loop);
    }
}
```

```
procedure insert(node n,
                 nodeset L) {
    if ( n ∉ L ) {
        add n to L;
        push n on stack;
    }
}
```

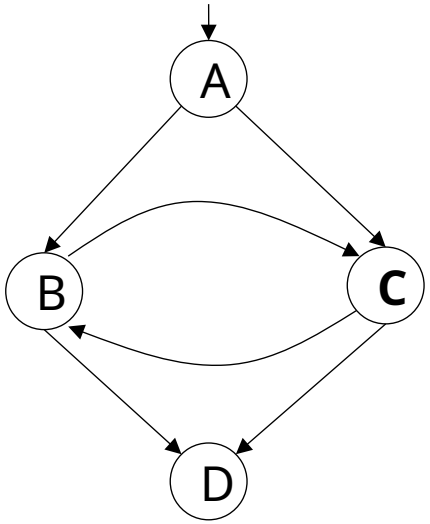
EXERCISE

- What are the back edges in this graph?
- For each back edge: which vertices are in its natural loop?



EXERCISE

- What are the back edges in this graph?
- For each back edge: which vertices are in its natural loop?

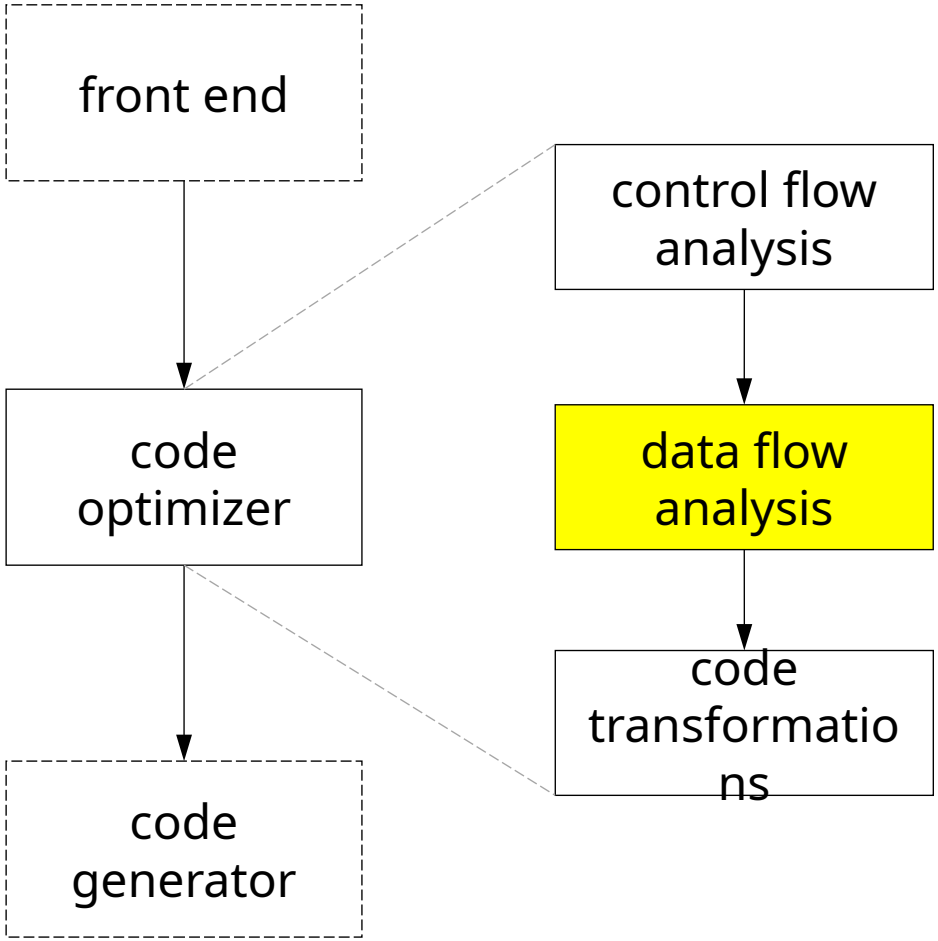


DATA FLOW ANALYSIS

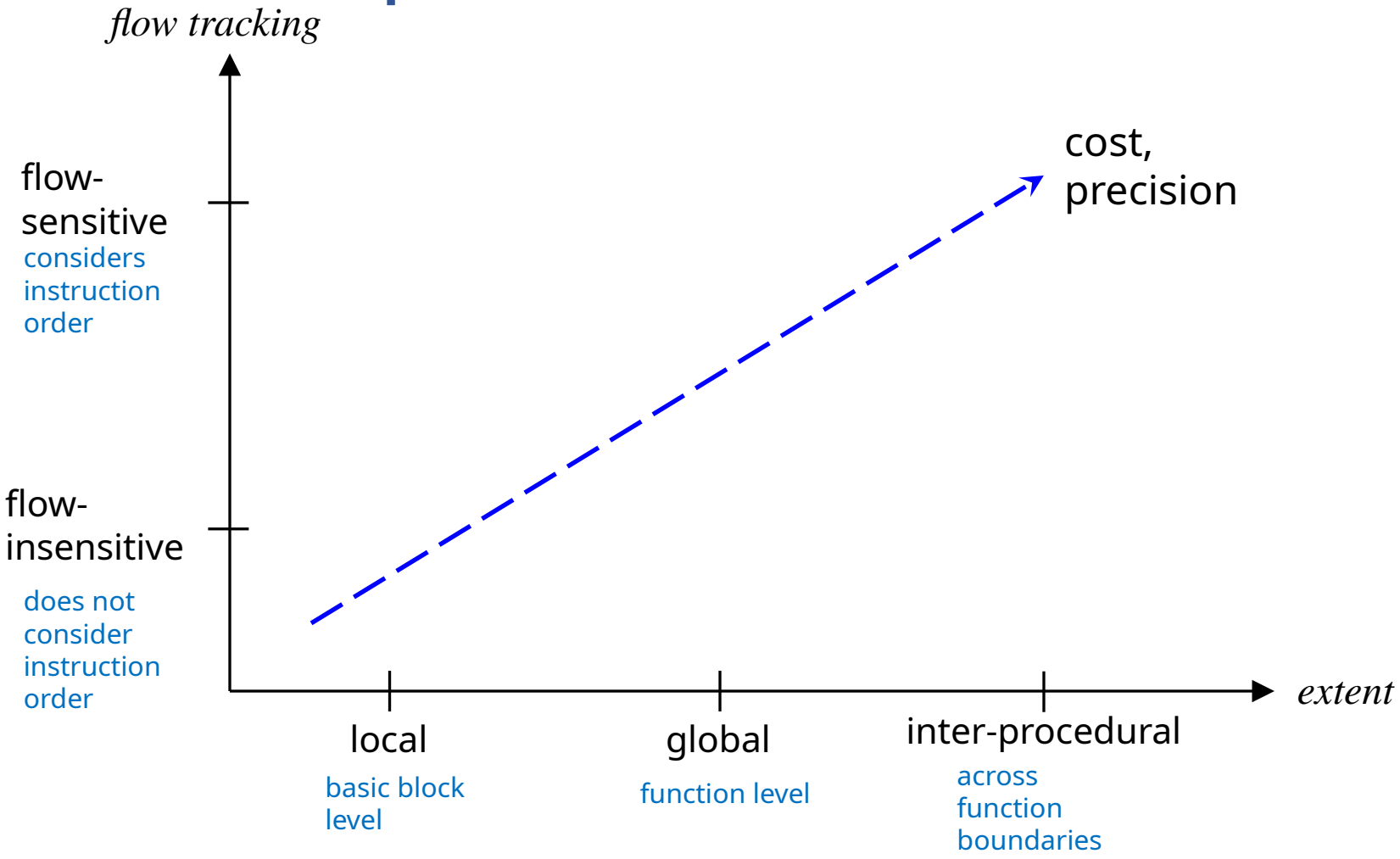
Dataflow analysis: goals

- To infer invariants about the runtime behavior of programs
  - *"what must be true, for any possible execution, whenever control reaches this point in the code?"*
  - this information is then used for various purposes, e.g., optimizing transformations, security analyses, …
- Criteria:
  - must be "safe," i.e., must take into account all possible executions of the program.

Analysis and optimization: organization



The Dataflow Analysis Landscape



Global dataflow analysis

- Set up equations relating properties at different program points in terms of properties at other “nearby” points.

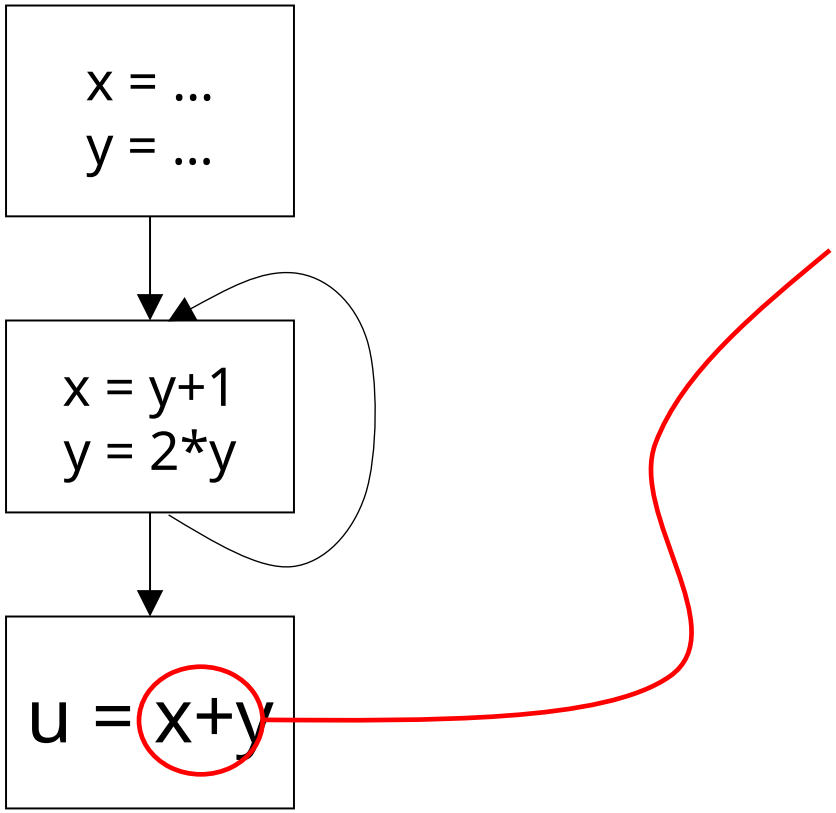
  Usually, for each block B:

  - one equation expresses how the property is affected by the instructions in B;
  - one equation captures the effect of the flow of values across basic block boundaries.

- We then solve these equations iteratively.

Analysis 1
Reaching definitions

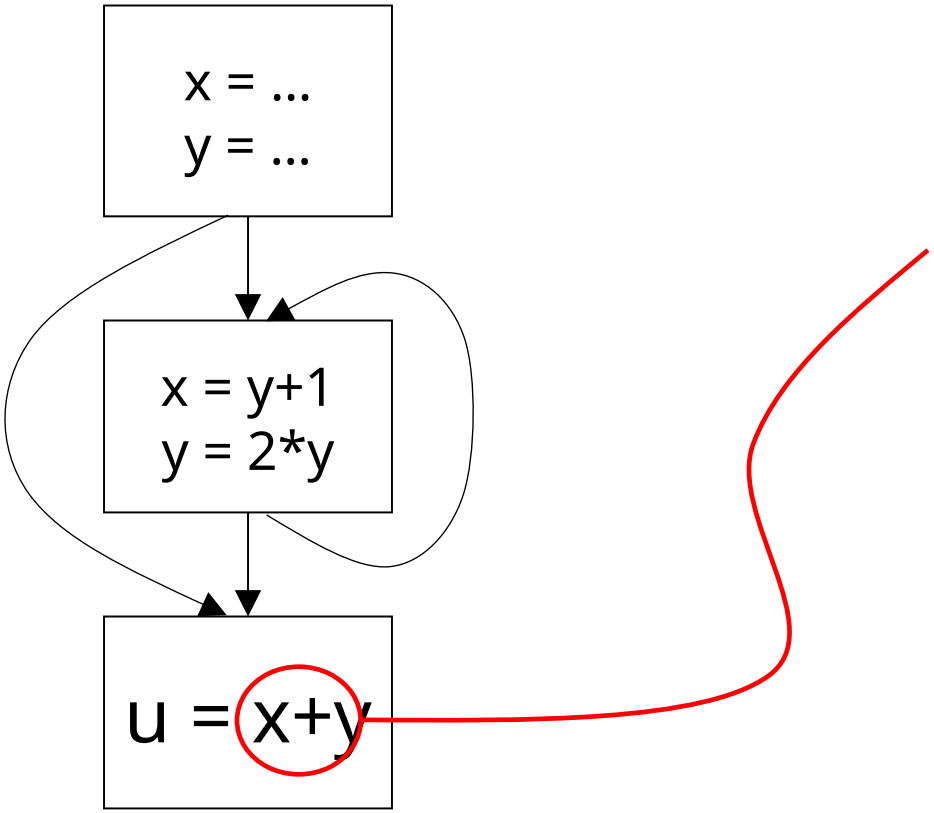
Reaching definitions



which instruction(s) could have assigned to x and y at this point?

which assignments to x and y "reach" this point?

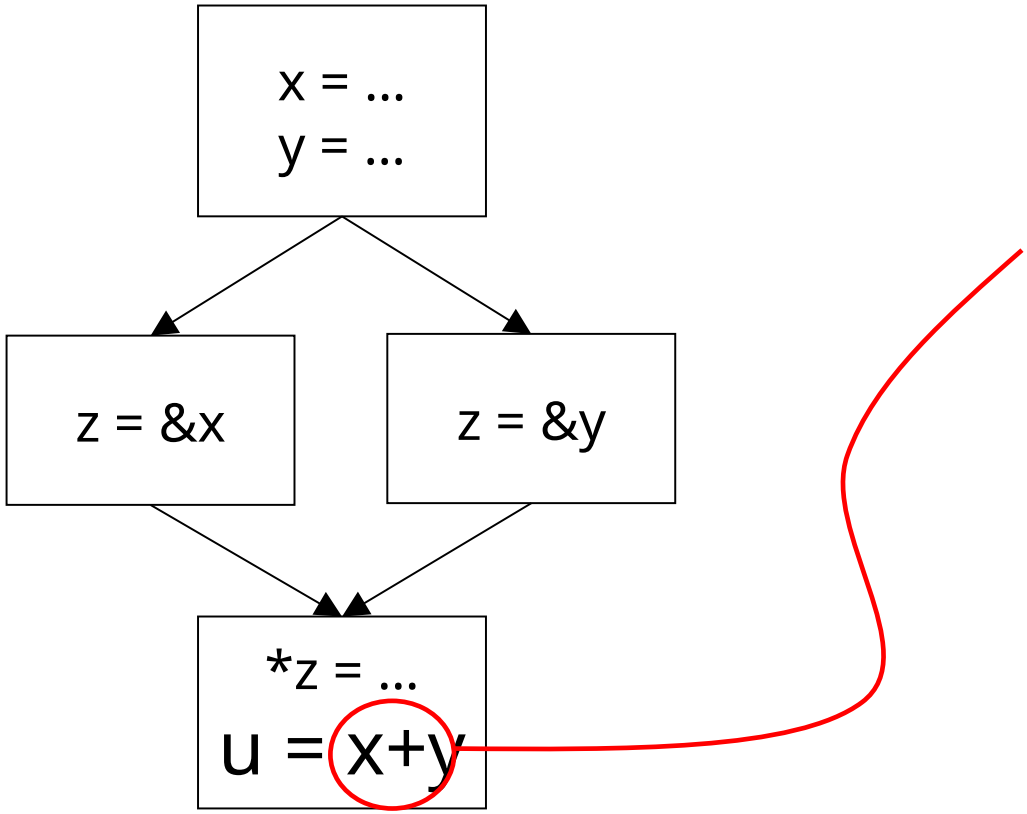
Reaching definitions



which instruction(s) could have assigned to x and y at this point?

which assignments to x and y "reach" this point?

Reaching definitions



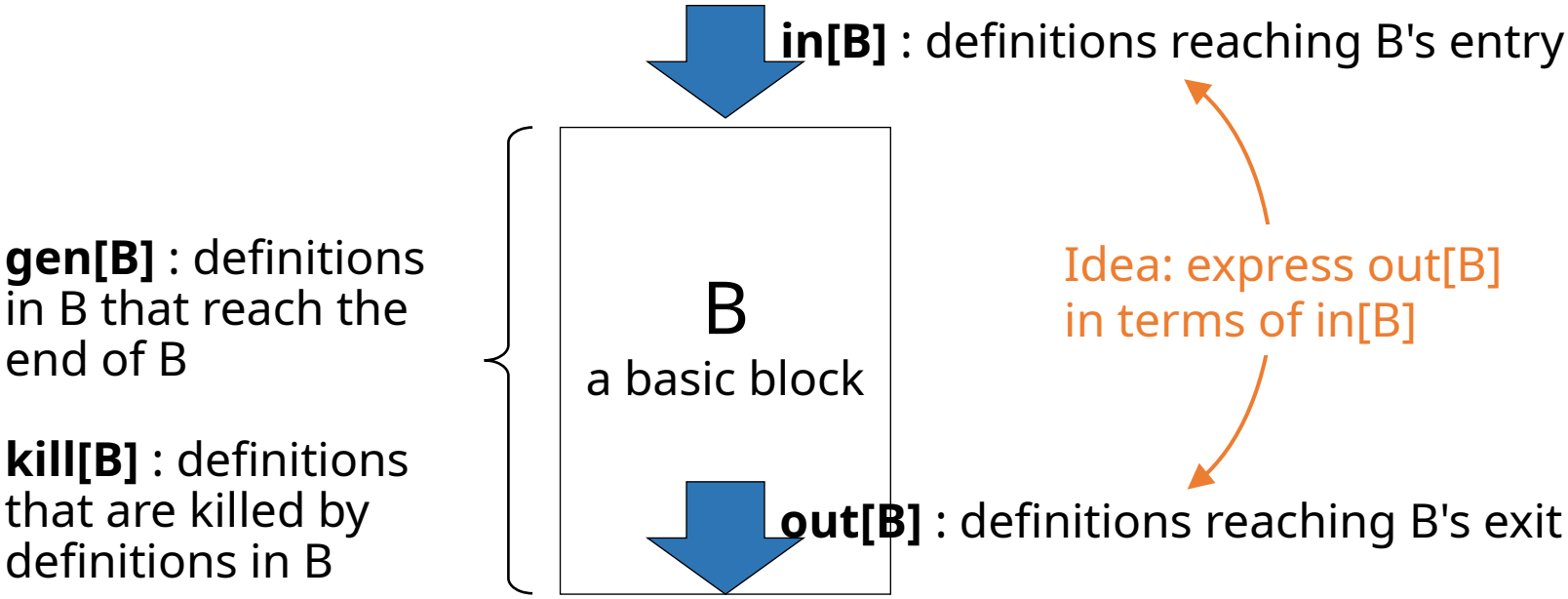
which instruction(s) could have assigned to x and y at this point?

which assignments to x and y "reach" this point?

Reaching definitions

- A *definition* of a variable x is an instruction that (may) assign a value to it
  - *unambiguous definitions*: definitely assign to x;
  - *ambiguous definitions*: may (or may not) assign to x, e.g., indirect stores, function calls
- A definition of x is *killed* along a path π if x is (definitely) redefined somewhere along π
- A definition d *reaches* a point p if there is some path π from d to p such that d is not killed along π

Computing reaching definitions



express how the code in B transforms in[B] to out[B]

- gen[B]: definitions added to in[B]
- kill[B]: definitions removed from in[B]

Reaching definitions: dataflow equations

$out[B] = gen[B] \cup (in[B] - kill[B])$

$in[B] = \cup \{ out[X] \mid X \text{ is a predecessor of } B \}$

gen[B]:	the set of definitions in B that reach the end of B
kill[B]:	the set of definitions (in the entire function) that are killed by definitions within B
in[B]:	the set of definitions that reach the entry to B
out[B]:	the set of definitions that reach the exit from B

Reaching definitions: dataflow equations

$out[B] = gen[B] \cup (in[B] - kill[B])$

$in[B] = \cup \{ out[X] \mid X \text{ is a predecessor of } B \}$

- Points to note:
  - out[B] is computed from in[B]

    in[B] computed from predecessors of B
    - *information is propagated along direction of control flow*
    - *"forward dataflow analysis"*
  - in the presence of loops, the equations become circular

Reaching definitions: *gen* and *kill* sets

For each block B: gen[B], kill[B] computed once, at the beginning, using information local to B:

1. $G = K = \varnothing$;
2. for each instruction I in B, going forward, do:
   - let I ≡ 'x = …', and D_x = {J | J is an instruction defining x};
   - $G = (G - D_x) \cup \{ I \}$
   - $K = (K \cup D_x) - \{ I \}$
3. gen[B] = *G* at the end of B;

   kill[B] = *K* at the end of B;

Solving dataflow equations

- Choose an initial approximation for in[B], out[B].
- Iteratively improve these approximations:
  - let in_i[B], out_i[B] be values obtained after iteration i.
  - recompute dataflow equations using these values:

$$out_{i+1}[B] = gen[B] \cup (in_i[B] - kill[B])$$

$$in_{i+1}[B] = \cup \{ out_i[X] \mid X \text{ is a predecessor of } B\}$$

until the equations converge, i.e., there is no change to in[B] or out[B] for any B.

Solving dataflow equations

```
for each block B, set out[B] = gen[B];    /* assumes in[B] = ⊘ */
chg = true
while ( chg ) {
  chg = false
  for each block B {
    in[B] = ∪ { out[X] | X a predecessor of B };
    oldout = out[B];
    out[B] = gen[B] ∪ (in[B] – kill[B]);
    if (out[B] ≠ oldout) chg = true;
  }
}
```

Implementation notes

- The sets being computed range over a fixed domain (variables, instructions, etc.)

  ⇒ *represent as bit vectors for efficiency.*

- Precompute gen[B], kill[B] as bit masks.
  - manipulate only in[B], out[B] during iteration.
- Storing dataflow information for every program point requires too much space:
  - store in[B] and/or out[B];
  - recompute values for points within B as needed.

Analysis 2
Liveness analysis

Liveness analysis

- ***Definition***: A variable is *live* at a program point p if its value is (may be) used at a later point q without getting redefined between p and q.
- Dataflow sets:
  - in[B], out[B] : the set of variables live at entry to/exit from B;
  - def[B]: the set of variables *definitely* assigned to within B before being used;
  - use[B]: the set of variables that *may be* used in B prior to being defined.

Liveness analysis: *def* and *use* sets

For each block B: def[B], use[B] are computed once, at the beginning, using information local to B:

1. $D = U = \emptyset$;
2. for each instruction I in B, going backward from the end, do: */* compute D, U immediately before I */*
   - let I ≡ ‘x = y ⊕ z’;
   - $D = (D \cup \{ x \}) - \{ y, z \}$;
   - $U = (U - \{ x \}) \cup \{ y, z \}$;
3. def[B] = *D* at the beginning of B;

   use[B] = *U* at the beginning of B.

Liveness analysis: equations and algorithm

in[B] = use[B] ∪ (out[B] – def[B])

out[B] = ∪ { in[X] | X is a successor of B }.

Algorithm:

1. for each basic block B, set in[B] = use[B];
2. **repeat**

   for each basic block B {

   out[B] = ∪ { in[X] | X is a successor of B };

   in[B] = use[B] ∪ (out[B] – def[B]);

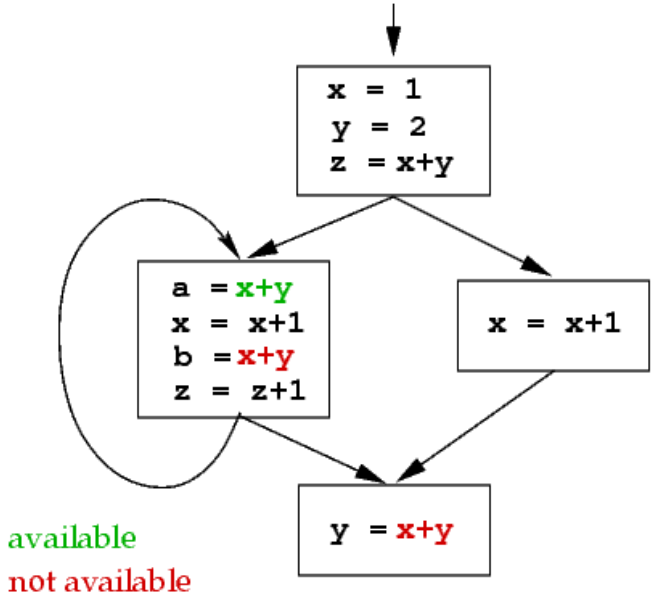
   }

until no change to in[B], out[B] for any B.

Analysis 3
Available expressions

Available expressions

- ***Definition***: An expression e is available at a point p if:
  - e is evaluated on every path from the entry node to p (incl. cyclic ones); and
  - the variables in e are not redefined after the last such evaluation.



If e is available, it need not be recomputed.

Available expressions: dataflow equations

- in[B], out[B]: sets of expressions available at B's entry/exit.
- gen[B]: the set of expressions e that are evaluated in B, s.t. no variable in e is redefined after the last evaluation of e in B.
- kill[B]: the set of expressions e in the program (function) s.t. B may redefine some variables in e without subsequently evaluating e.

$in[B_0] = \varnothing$ (B_0 is the entry node)

$in[B] = \cap \{ out[X] \mid X \text{ a predecessor of } B\}$ if $B \neq B_0$

$out[B] = gen[B] \cup (in[B] - kill[B])$

Available expressions: *gen* and *kill* sets

For each block B: gen[B], kill[B] computed once, at the beginning of the analysis, using information local to B:

1. $G = K = \varnothing$;
2. for each instruction I in B, going forward, do:
   - let I $\equiv$ 'x = y $\oplus$ z', and E_x = all expressions involving x;
   - $G = (G \cup \{\text{'y} \oplus \text{z'}\}) - E_x$;
   - $K = (K - \{\text{'y} \oplus \text{z'}\}) \cup E_x$;
3. gen[B] = *G* at the end of B;

   kill[B] = *K* at the end of B;

Available expressions: Algorithm

Let *U* be the set of all expressions appearing on the RHS of some instruction in the function.

1. $in[B_0] = \varnothing$;

   for each block B, set out[B] = *U* – kill[B];

2. **repeat**

   for each basic block B {

   in[B] = $\cap$ { out[X] | X a predecessor of B };

   out[B] = gen[B] $\cup$ (in[B] – kill[B]);

   }

   until no change to in[B], out[B] for any B.

Comparing different analyses

Reaching definitions vs. Available expressions

1. Modality:
   - d *reaches* p if it can reach p without getting killed along *some* path (∃-analysis).
     - ⇒ information merge operator: ∪
     - ⇒ initial estimates set too small.
   - e *available at* p if e evaluated, not killed along *every* path to p (∀-analysis).
     - ⇒ information merge operator: ∩
     - ⇒ initial estimates set too large.

2. Direction: both are *forward analyses*.
   - ⇒ out[B] computed from in[B] (using gen, kill);
   - ⇒ in[B] computed from out sets of predecessors of B.

Reaching definitions vs. Liveness Analysis

1. Modality: both are ∃-analyses:
   - merge operator: ∪
   - initial approximations are too small.
2. Direction:
   - reaching definitions: forward analysis
     - ⇒ out[B] defined in terms of in[B];
     - ⇒ in[B] defined in terms of out-sets of predecessors of B.
   - liveness analysis: backward analysis
     - ⇒ in[B] defined in terms of out[B]
     - ⇒ out[B] defined in terms of in-sets of succesors of B.

Families of dataflow analyses

- Analyses can be classified in terms of *modality* ($\forall$ or $\exists$) and *direction* (forward or backward).
- These aspects of an analysis define
  - the structure of the dataflow equations;
  - the merge operator;
  - the nature of the initial approximations.

	$\exists$	$\forall$
forward	reaching defns.	available exprs.
backward	variable liveness	?

Analysis 4
Constant propagation

Analysis 4. Constant propagation

- *Goal*: Identify each expression e that can be guaranteed to evaluate to some fixed constant n_e for all possible executions of the program.

- *Motivation*: Avoid runtime evaluation of expressions that can be evaluated at compile time. E.g.:

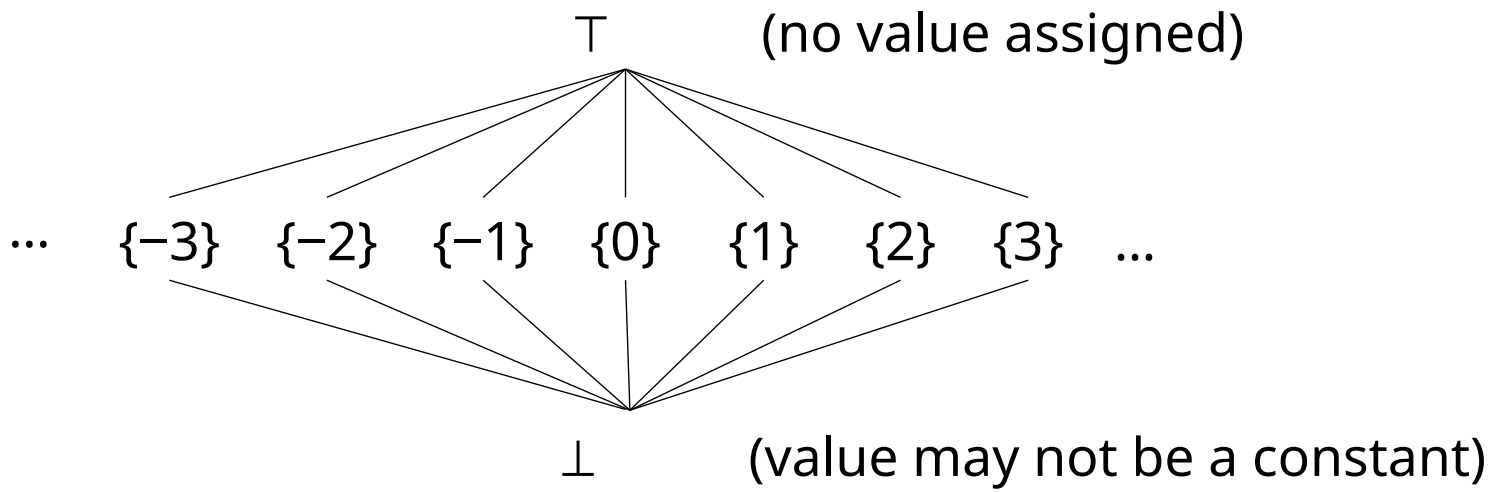
```
#define  N    100
...
p = (int *) malloc( N*N*sizeof(int) );
for (i = 0; i < N*N; i++) {
    p[i] = 0;
}
```

Constant propagation: Approach

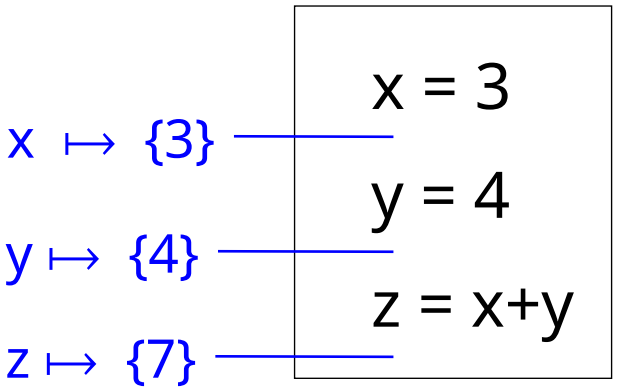
- Intuition: Compute an (over-)estimate of the set of values that can be taken on by an expression.
  - but we really only care about whether or not this set of values is a singleton

Constant propagation: Approach

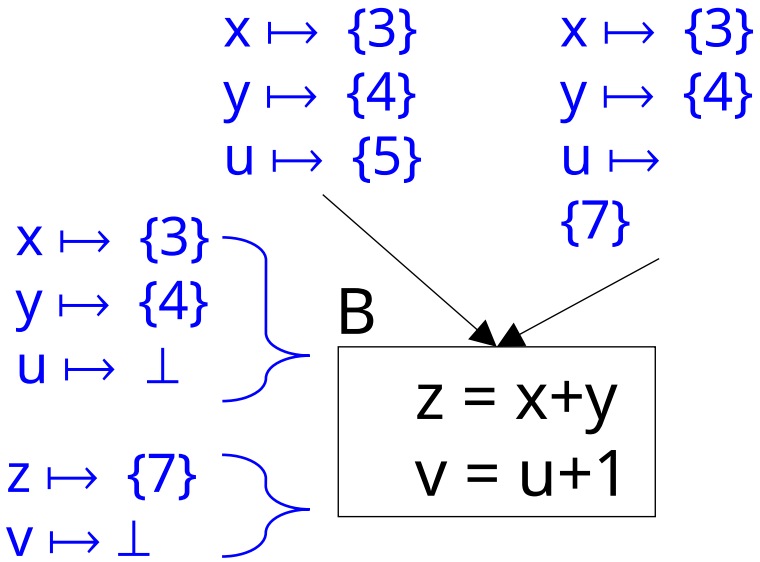
- Intuition: Compute an (over-)estimate of the set of values that can be taken on by an expression.
  - but we really only care about whether or not this set of values is a singleton



Constant propagation: Intuition



If all the variables in an expression have constant values, then the expression also has a constant value



A variable x is a constant n at the entry to B iff x is the same constant n at the exit of each predecessor of B

Constant propagation: Algorithm

- Initialization: for each variable v, set v $\mapsto \top$
- Iterate until no change:
  - for each instruction I $\equiv$ 'x = y $\oplus$ z' in each block in order:
    - if y $\mapsto \top$ or z $\mapsto \top$ before I, set x $\mapsto \top$ after I
    - else if y $\mapsto n_y$ and z $\mapsto n_z$ before I, and '$n_y \oplus n_z$' is a constant n_x, set x $\mapsto n_x$ after I
    - else set x $\mapsto \bot$ after I
  - propagate constants across block boundaries. At each block B:
    - for each variable x, if $\exists n_x$: x $\mapsto n_x$ at the end of each predecessor of B, then x $\mapsto n_x$ at entry to B