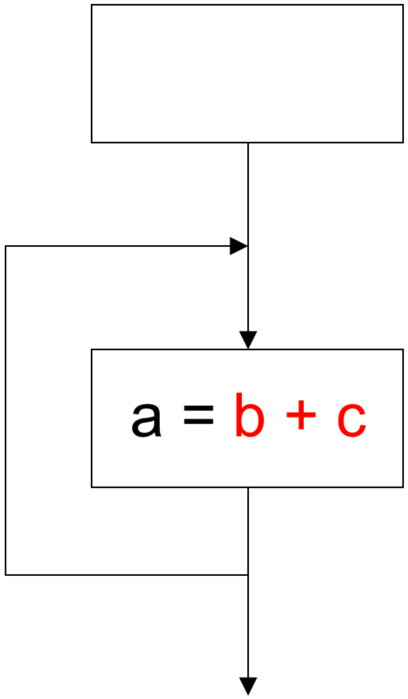
CS293S Lazy Code Motion

Yufei Ding

Slides adapted from Phillip B.
Gibbons and Todd C. Mowry

Loop-Invariant Expressions

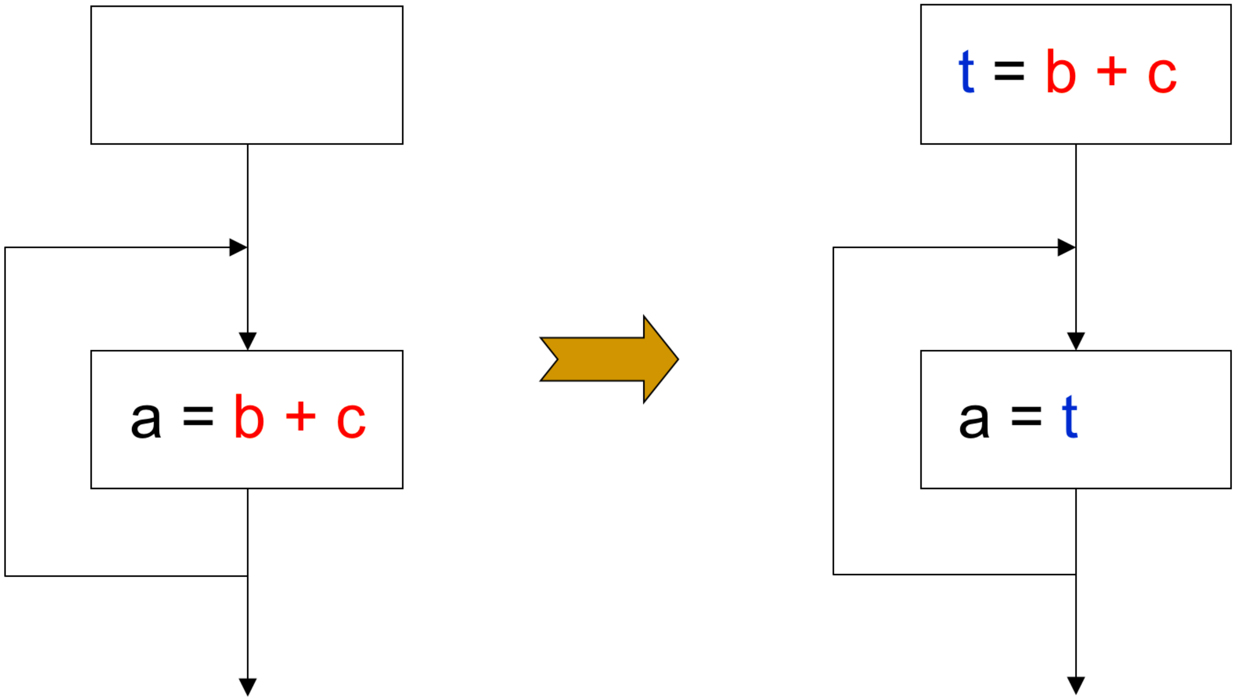
Given an expression (b+c) inside a loop,
– does the value of b+c change inside the loop?
– is the code executed at least once?



Loop invariant expressions are partially redundant

Loop-Invariant Expressions

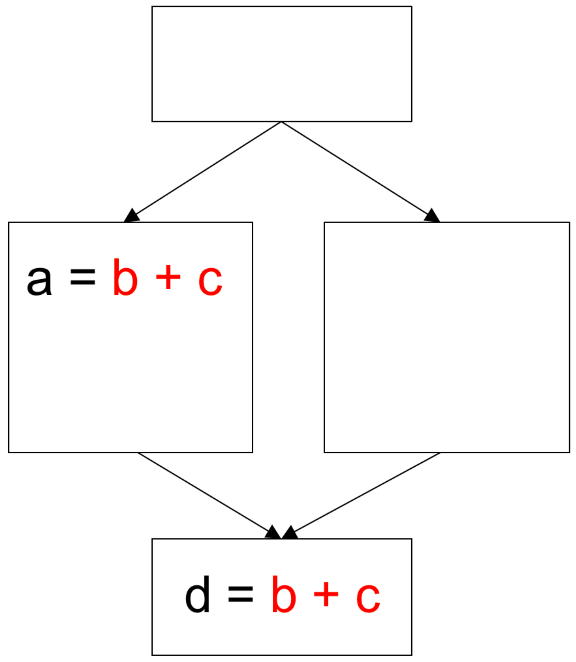
Given an expression (b+c) inside a loop,
– does the value of b+c change inside the loop?
– is the code executed at least once?



Loop invariant expressions are partially redundant

Partial Redundant Expressions

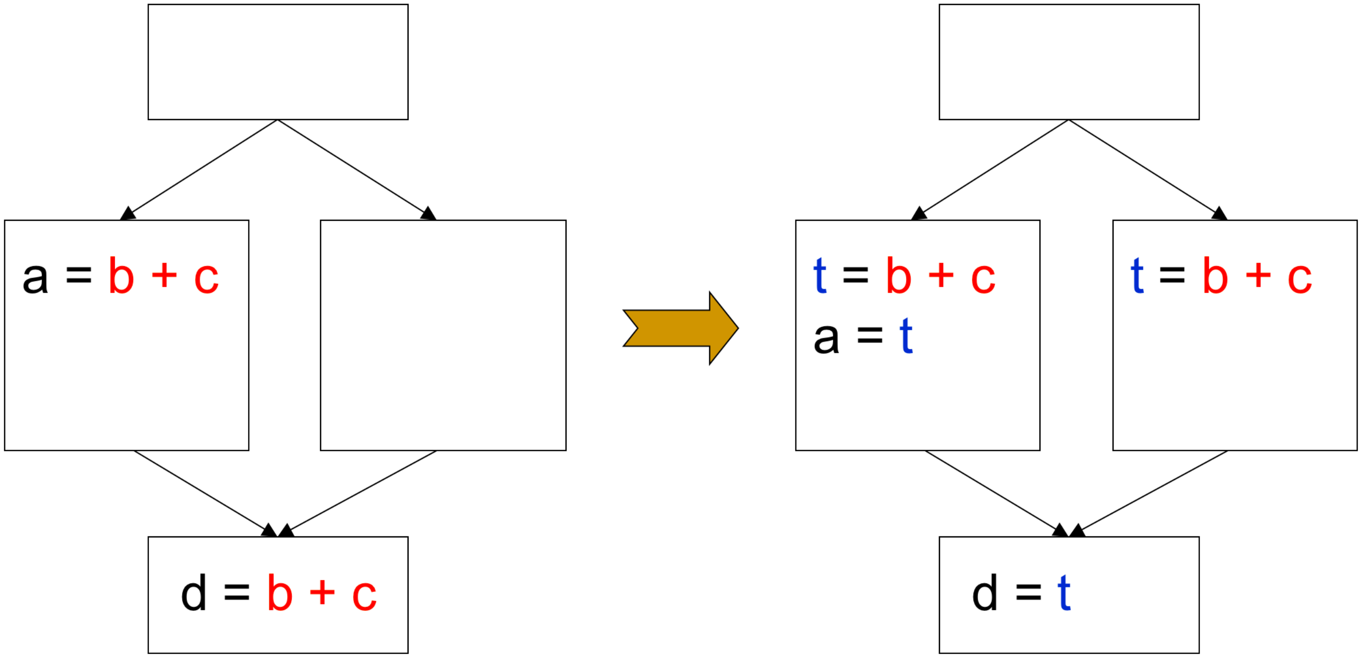
An expression is partially redundant at p if it is redundant along some, but not all, paths reaching p.



- Can we place calculations of b+c such that no path re-executes the same expression?

Partial Redundant Expressions

An expression is partially redundant at p if it is redundant along some, but not all, paths reaching p.



a = b + c

d = b + c

t = b + c
a = t

t = b + c

d = t

- Can we place calculations of b+c such that no path re-executes the same expression?

Partial-Redundancy Elimination

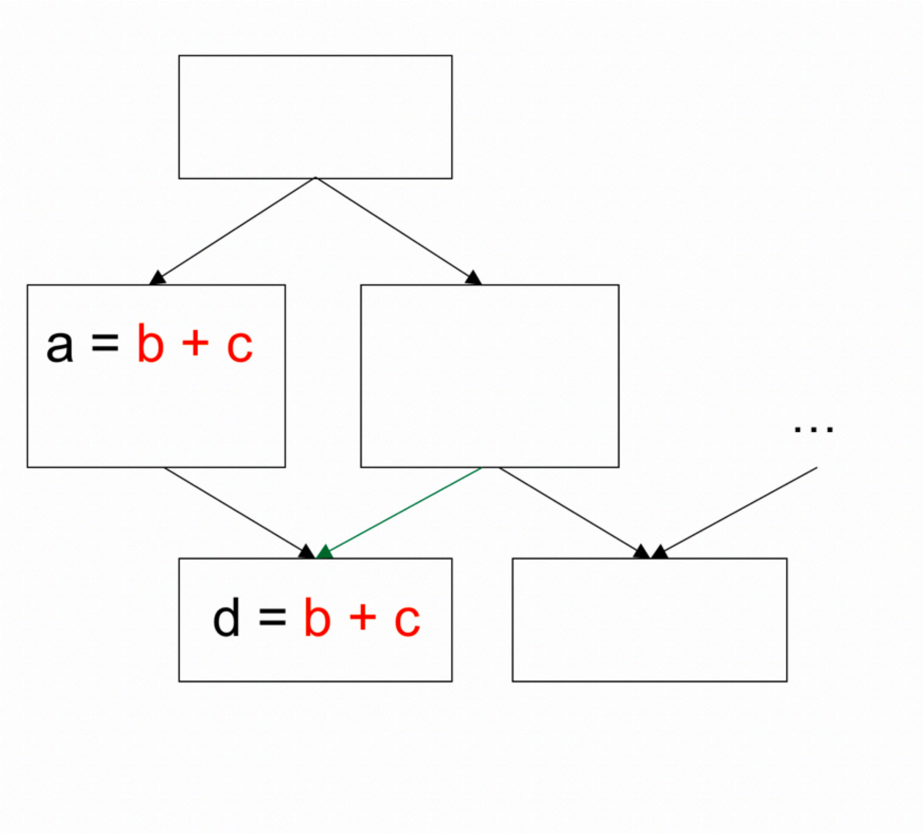
- Partial redundancy elimination performs code motion to Minimize the number of expression evaluations

- Major part of the work is figuring out where to operations

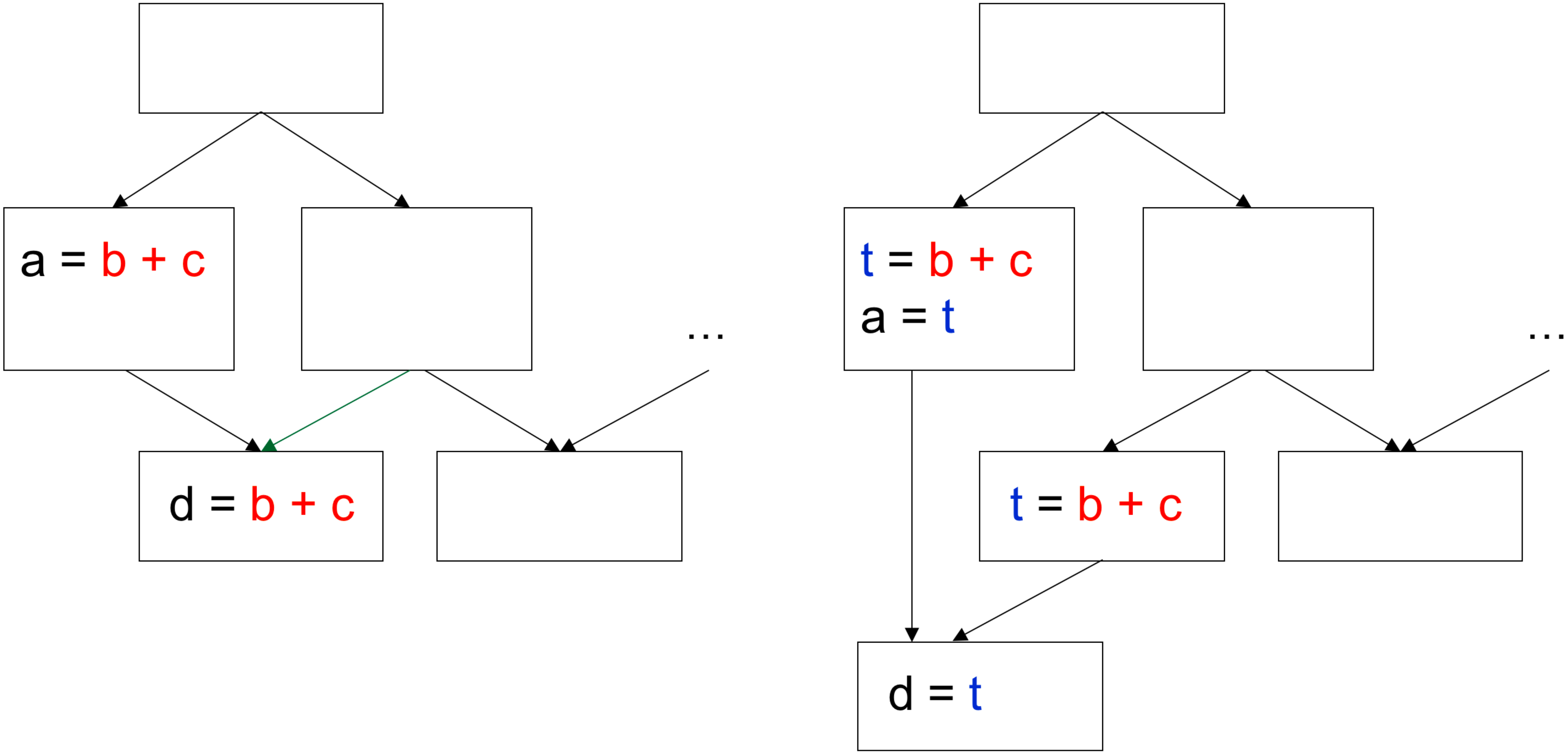
Goal: By moving around the places where an expression is evaluated and keeping the result in a temporary variable when necessary, we often can reduce the number of evaluations of this expression along many of the execution paths, while not increasing that number along any path.

Can All Redundancy Be Eliminated by code motion?

New blocks creation



New blocks creation



a = b + c

d = b + c

…

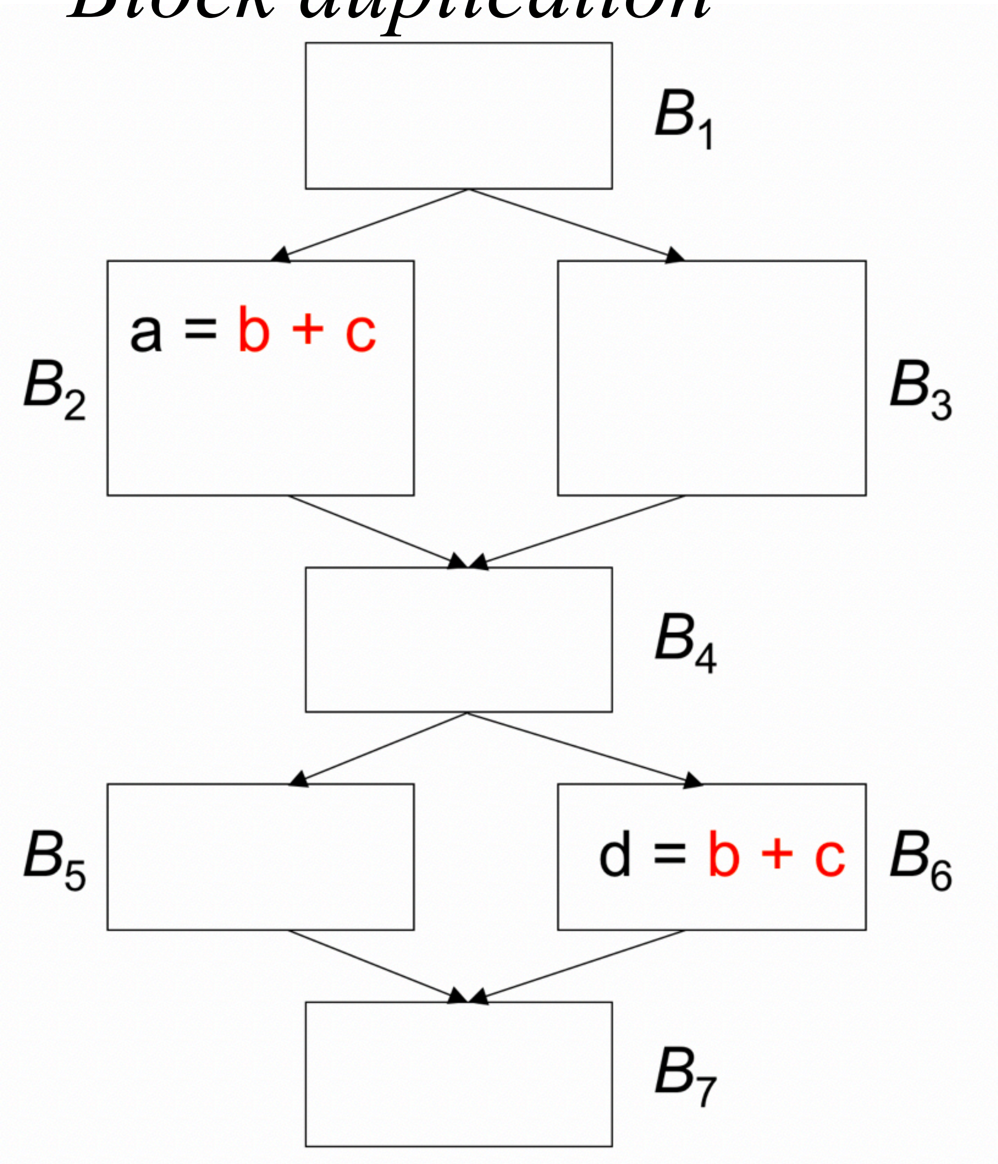
t = b + c
a = t

t = b + c

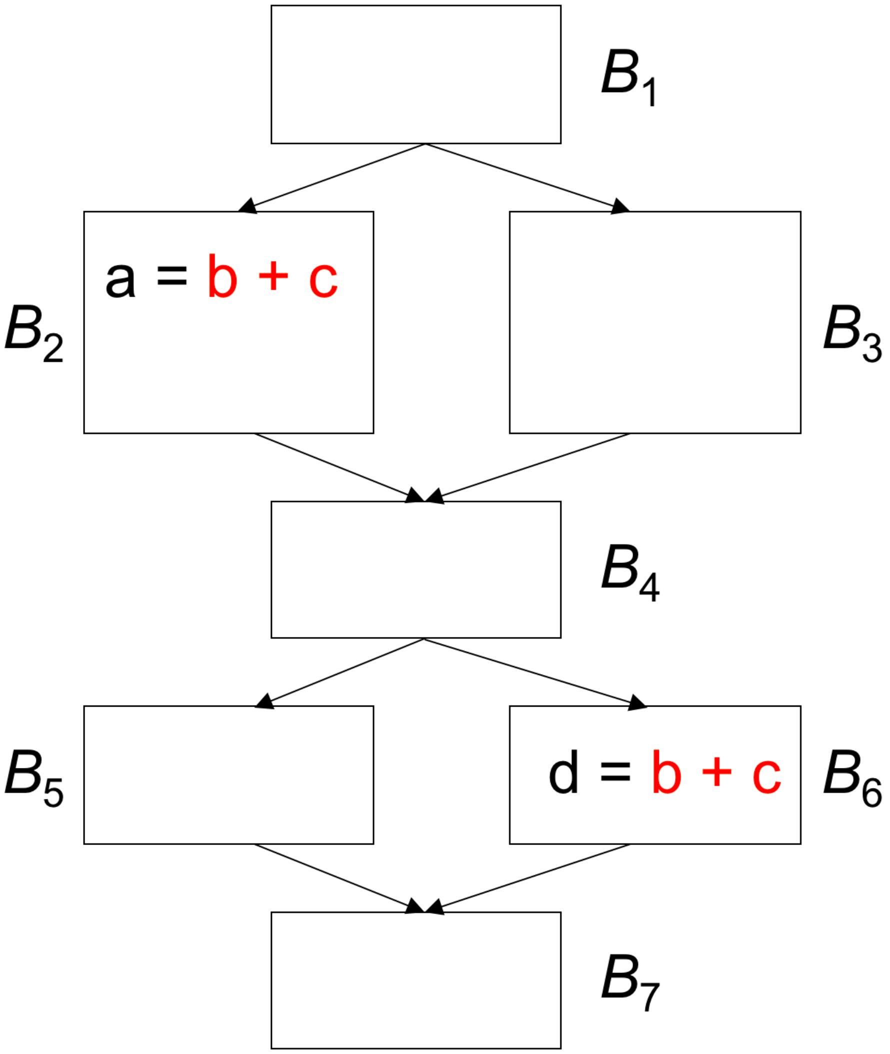
…

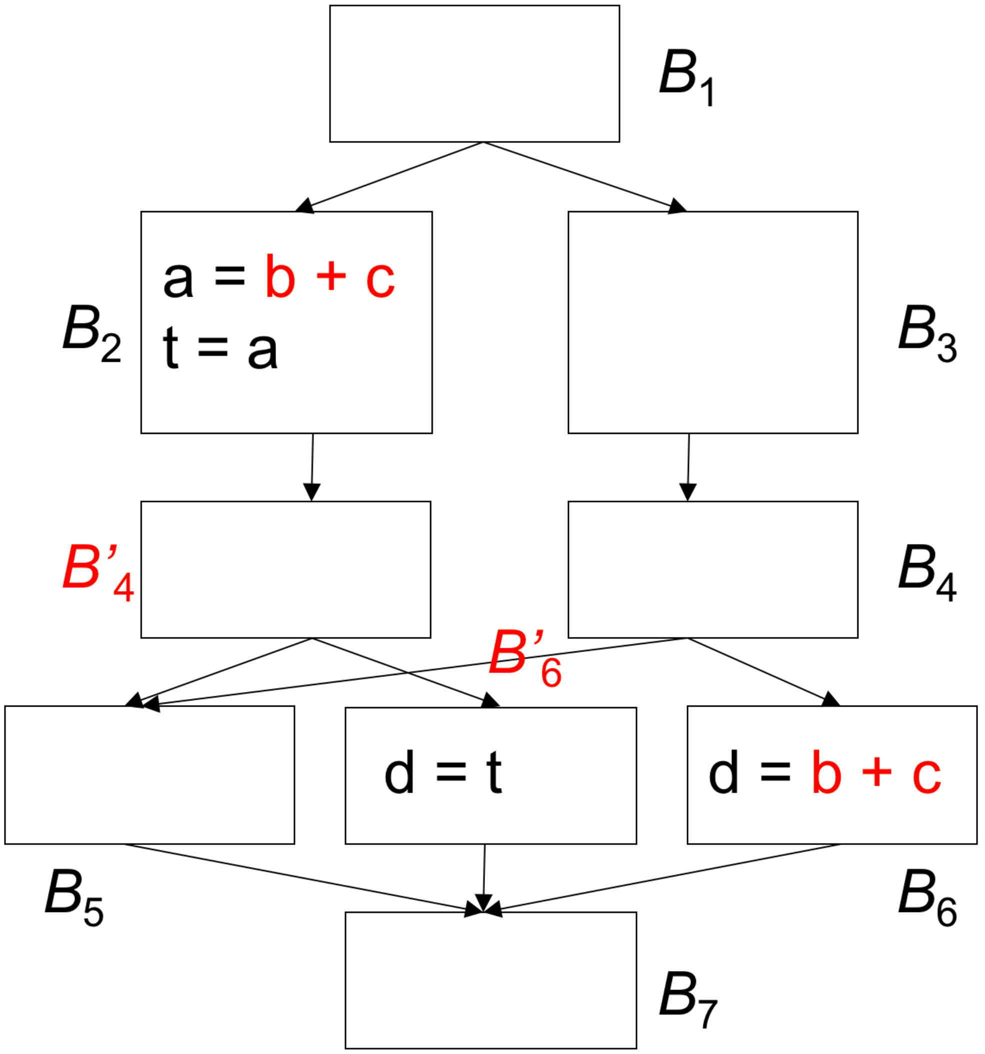
d = t

Block duplication



Block duplication





Can All Redundancy Be Eliminated by code motion?

- It is not possible to eliminate all redundant computations along every path, unless we are allowed to change the control flow graph by creating new blocks and duplicating blocks.
- New blocks creation: it can be used to break "critical edge", which is an edge leading from a node with more than one successor to a node with more than one predecessor.
- Block duplication: it can be used to isolate the path where redundancy is found.

References

1. E. Morel and C. Renvoise, “Global Optimization by Suppression of Partial Redundancies,” CACM 22 (2), Feb. 1979, pp. 96-103.
2. Knoop, Rüthing, Steffen, “Lazy Code Motion,” PLDI 92.
3. F. Chow, A Portable Machine-Independent Global Optimizer--Design and Measurements. Stanford CSL memo 83-254.
4. Dhamdhere, Rosen, Zadeck, “How to Analyze Large Programs Efficiently and Informatively,” PLDI 92.
5. K. Drechsler, M. Stadel, “A Solution to a Problem with Morel and Renvoise’s ‘Global Optimization by Suppression of Partial Redundancies,’” ACM TOPLAS 10 (4), Oct. 1988, pp. 635-640.
6. D. Dhamdhere, “Practical Adaptation of the Global Optimization Algorithm of Morel and Renvoise,” ACM TOPLAS 13 (2), April 1991.
7. D. Dhamdhere, “A Fast Algorithm for Code Movement Optimisation,” SIGPLAN Not. 23 (10), 1988, pp. 172-180.
8. S. Joshi, D. Dhamdhere, “A composite hoisting --- strength reduction transformation for global program optimisation,” International Journal of Computer Mathematics, 11 (1982), pp. 21-41, 111-126.

The Lazy-Code-Motion Problem

Three properties desirable from the partial redundancy elimination algorithm:

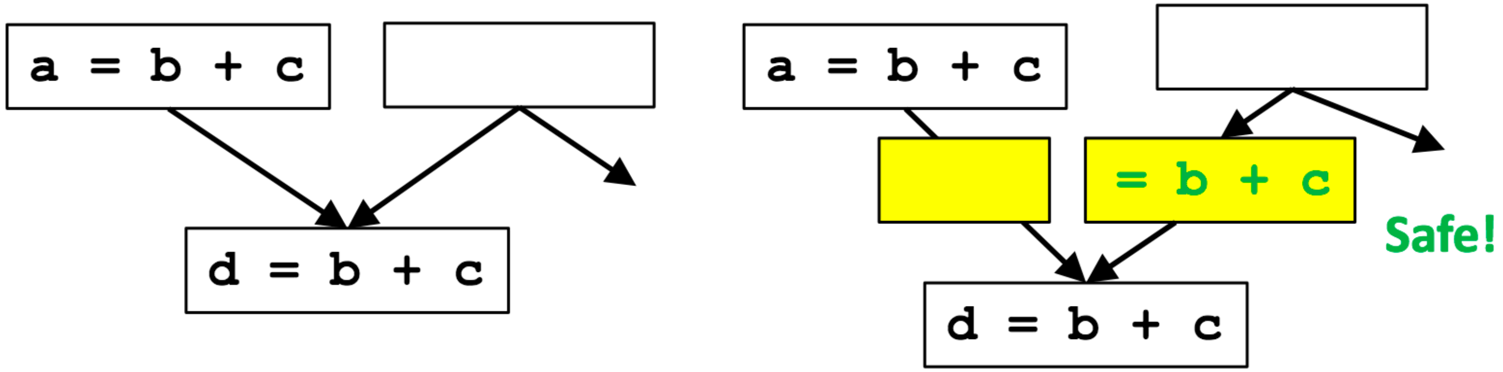
- All redundant computations of expressions that can be eliminated without block duplication are eliminated
- No extra computation is added.
- Expressions are computed at the latest possible time
  - Least register pressure.

Challenge: to systematically find the right places for inserting copy statements.

The Lazy-Code-Motion Problem

- Algorithm overview
  - Find all the anticipated expressions at each program point using a backward analysis
  - Find all the “available” expressions at each program point using a forward analysis.
  - Find the earliest point that an expression can be placed
  - Find all the “postponable” expressions at each program point using a forward analysis
  - Place expressions at those points where they can no longer be postponed
  - …

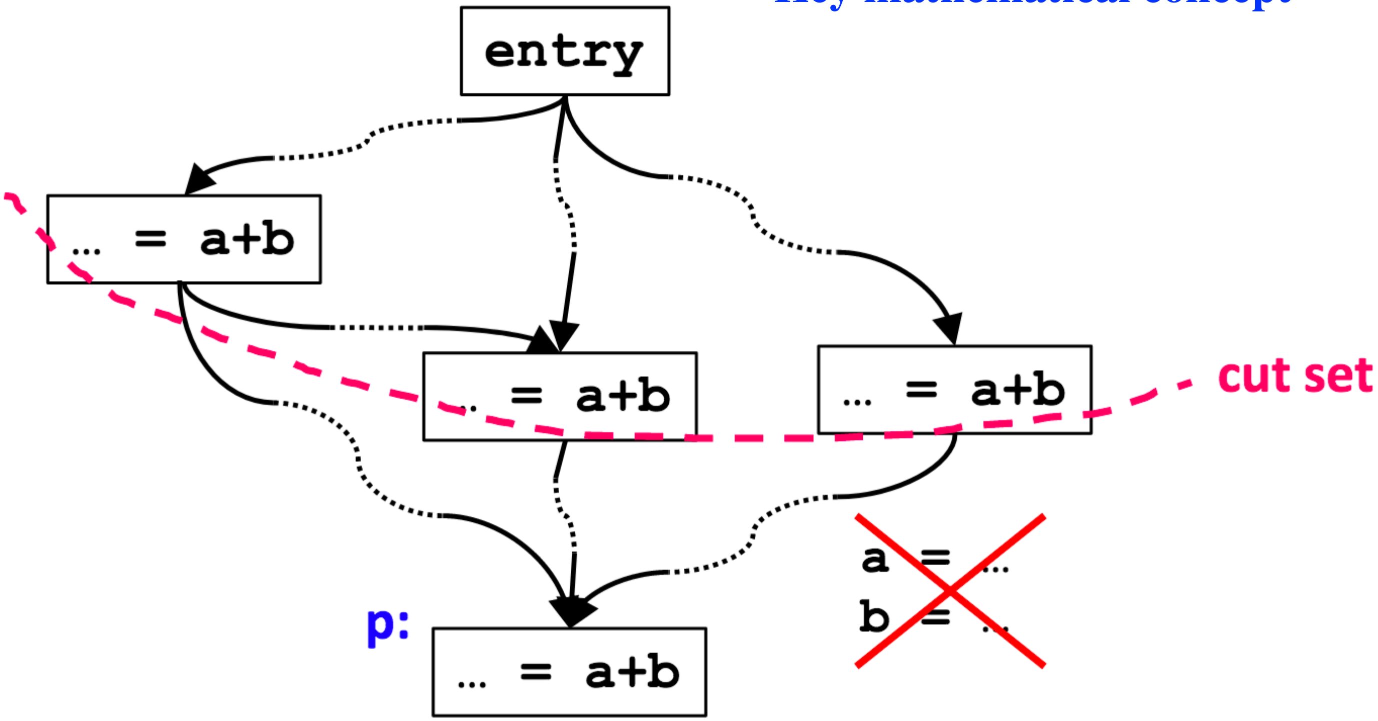
Preprocessing: Preparing the Flow Graph



- Modify the flow graph:
  - Ensure redundancy elimination power
    - Add a basic block for every edge that leads to a basic block with multiple predecessors (to ensure the)
  - Keep algorithm simple
    - Restrict placement of instructions to the beginning of a basic block
    - Consider each statement as its own basic block.

Full Redundancy: A Cut Set in a Graph

Key mathematical concept



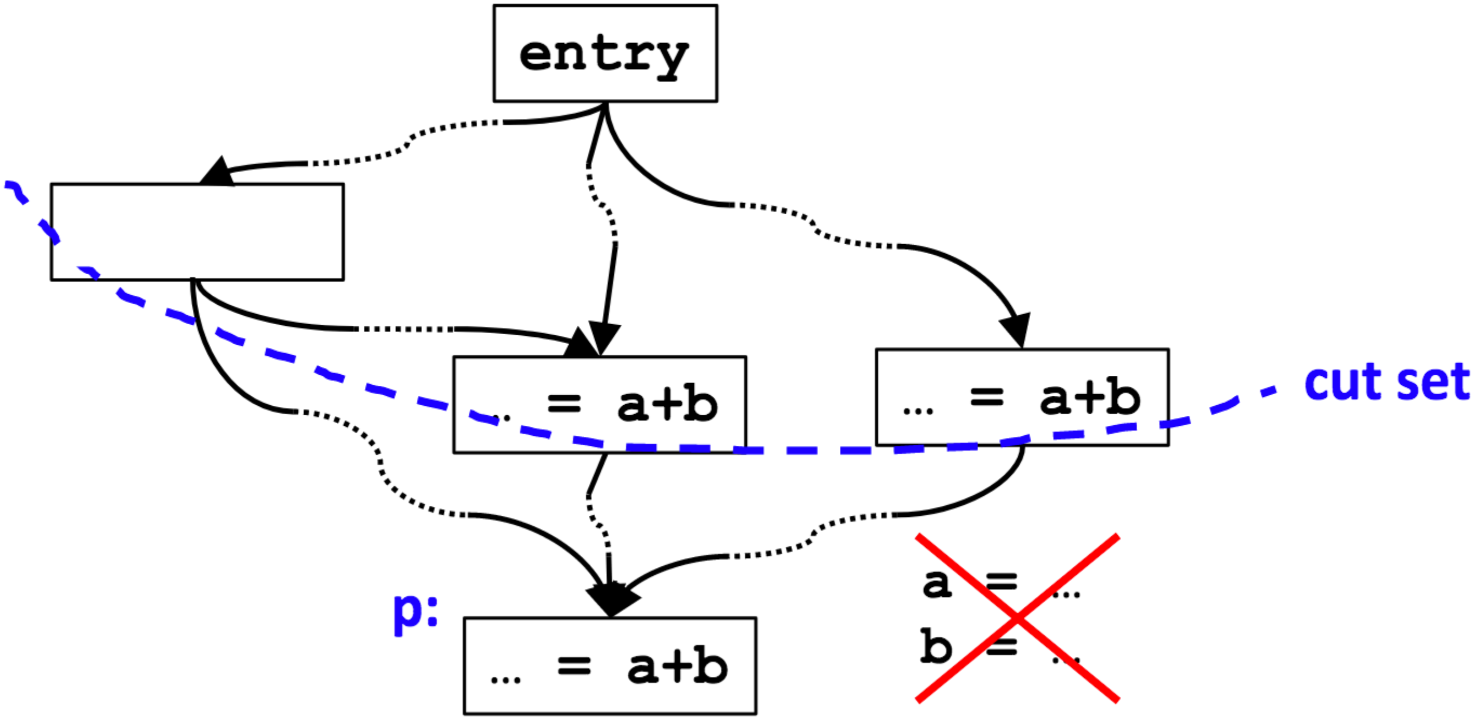
Full redundancy at p: expression a+b redundant on all paths

– a cut set: nodes that separate entry from p (could have multiple cut sets).

– each node in a cut set contains a calculation of a+b.

– a, b not redefined.

Partial Redundancy: Completing a Cut Set



Partial redundancy at p: redundant on some but not all paths

– Add operations to create a cut set containing a+b

– Note: Moving operations up can eliminate redundancy

Constraint on placement: no wasted operation

– Range where a+b is anticipated --> Choices

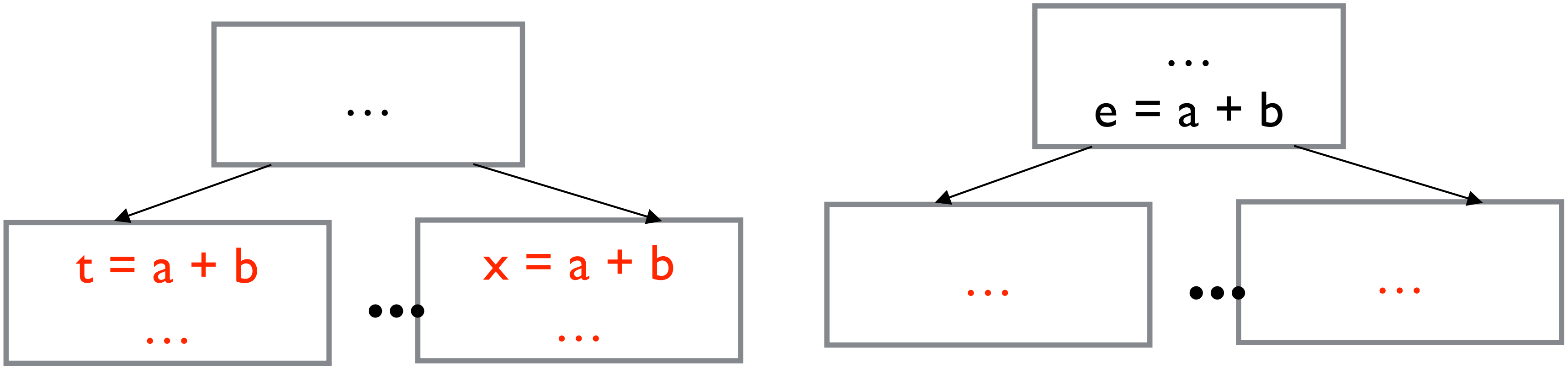
Anticipated (Very Busy) Expressions

- An expression is anticipated at point p if all paths leaving p eventually compute the expression from the values of the operands that are available at p.

- To ensure that no extra operations are executed, copies of an expression must be placed only at program points where the expression is anticipated (very busy).

Very Busy Expressions

- Def: e is a very busy expression at the exit of block b if
  - e is evaluated and used along every path that leaves b, and
  - evaluating e at the end of b produces the same result
- useful for code hoisting
- saves code space



Very Busy Expressions

- VERYBUSY(b) contains expressions that are very busy at <u>end</u> of b
- UEEXPR(b): up exposed expressions (i.e. expressions defined in b and not subsequently killed in b)
- EXPRKILL(b): killed expressions

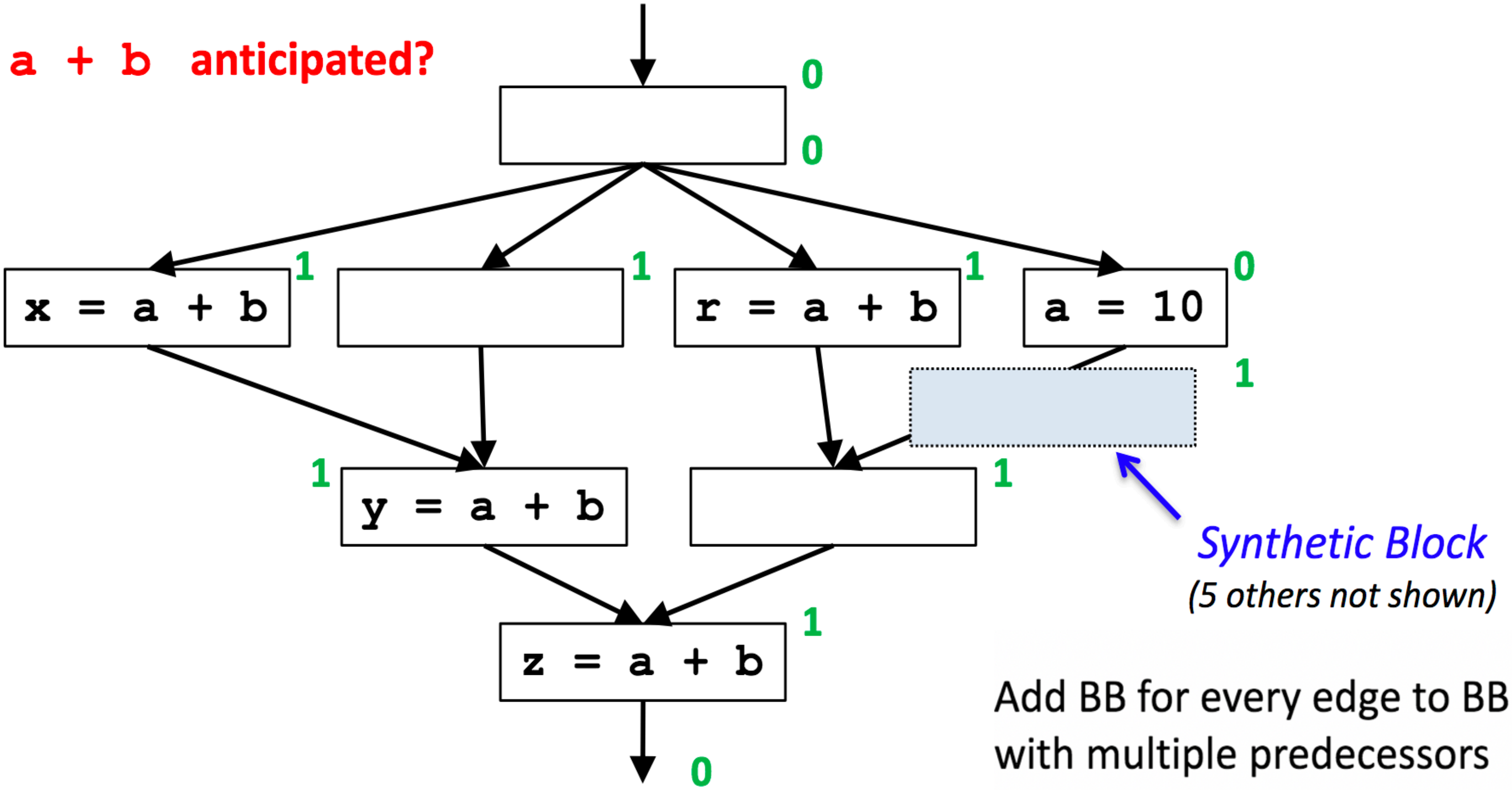
A backward flow problem, domain is the set of expressions

$$\text{VERYBUSY}(b) = \cap_{s \in \text{succ}(b)} \text{UEEXPR}(s) \cup (\text{VERYBUSY}(s) \cap \overline{\text{EXPRKILL}(s)})$$

$$\text{VERYBUSY}(n_f) = \emptyset$$

Example 1: where to insert/move the inst.?

Where is `a + b` anticipated?



Synthetic Block

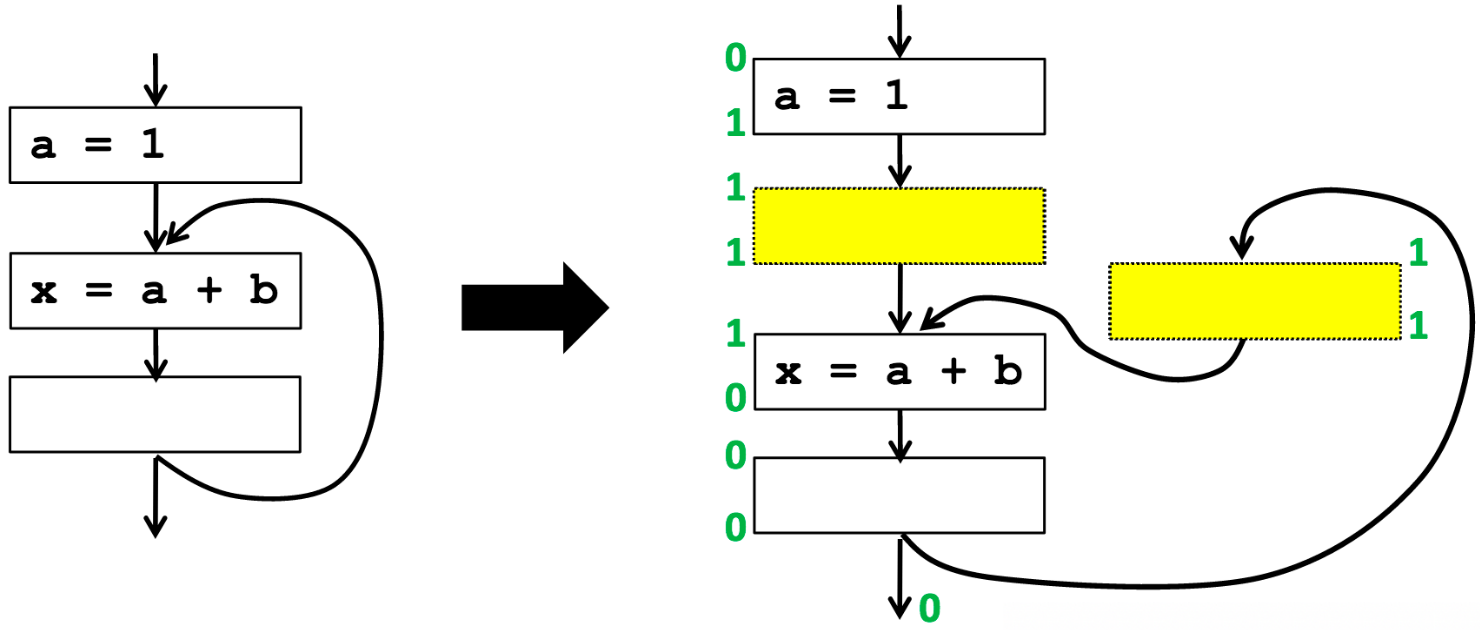
(5 others not shown)

Add BB for every edge to BB with multiple predecessors

What is the result if we insert t = a + b at the frontier of anticipation ?
i.e., those BBs for which a + b is anticipated to the entry of BB, but not anticipated to the entry of its parents.

Example 2: where to insert/move the inst.?

Where is `a + b` anticipated?



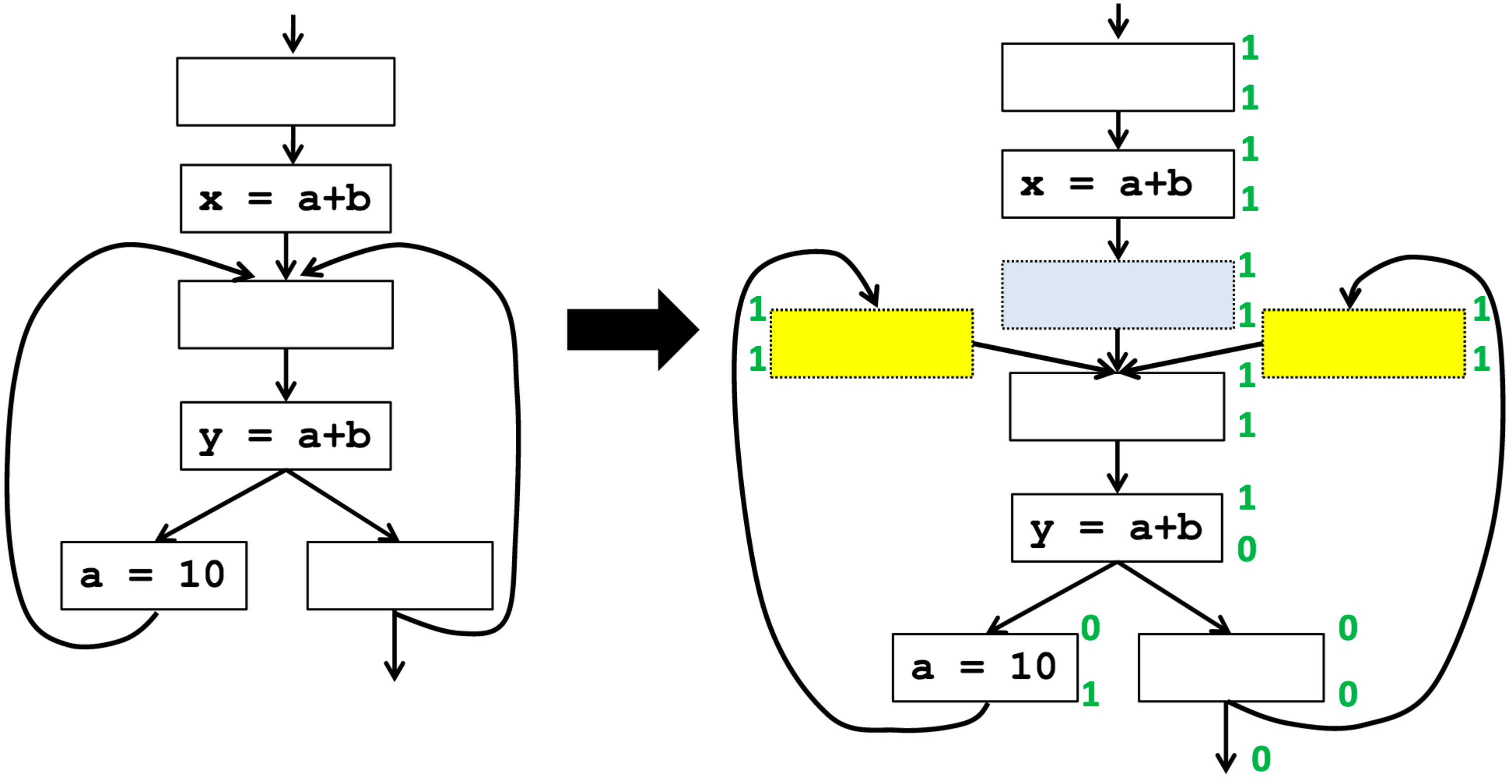
Add BB for every edge to BB with multiple predecessors

What is the result if we insert t = a + b at the frontier of anticipation ?

-- doesn’t eliminate redundancy within loop (why not?)

Example 3: where to insert/move the inst.?

Where is `a + b` anticipated?



- What is the result if we insert to the frontier of anticipation?
- What if we simply avoid insertion to BB in a loop?
- Where would we ideally like to insert “a+b” in this case

(will be) Available Expressions

- Pretend we calculate expression e whenever it is anticipated.
- e will be available at p if e has been "anticipated but not subsequently killed" on all paths reaching p

Direction	Forwards
Transfer function	$f_B(x) = (anticipated[B].in \cup x) - e\_kill_B$
Boundary	$\mathrm{OUT}[\mathrm{ENTRY}] = \varnothing$
Meet($\wedge$)	$\cap$
Equations	$\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$ $\mathrm{IN}[B] = \wedge_{P,\, pred(B)} \mathrm{OUT}[P]$

- e-kill$_B$ is the set of expressions any of whose operands are defined in B (a.k.a, ExpKill)

Where to insert?

- Any anticipated blocks

- First approximation: frontier between “not anticipated” & “anticipated”. It could already remove most of the PRE.

- How to find such anticipated frontier and exclude “those not needed blocks” discussed in previous loop examples?
  Final solution: Place expression at “anticipated” but not “will be available” blocks

earliest[b] = anticipated[b] - available[b]

Early Insertion Algorithm and Analysis

Algorithm:

For all basic block b, if x+y ϵ earliest[b]

- at beginning of b:

create a new variable t,

t = x+y,

- replace every original x+y in the CFG by t

Result:

- Maximized redundancy elimination (Placed as early as possible)
- But: register lifetimes?

The Lazy-Code-Motion Problem

- Algorithm overview
  - Find all the anticipated expressions at each program point using a backward analysis
  - Find all the "available" expressions at each program point using a forward analysis.
  - Find the earliest point that an expression can be placed
  - Find all the "postponable" expressions at each program point using a forward analysis
  - Place expressions at those points where they can no longer be postponed
  - …

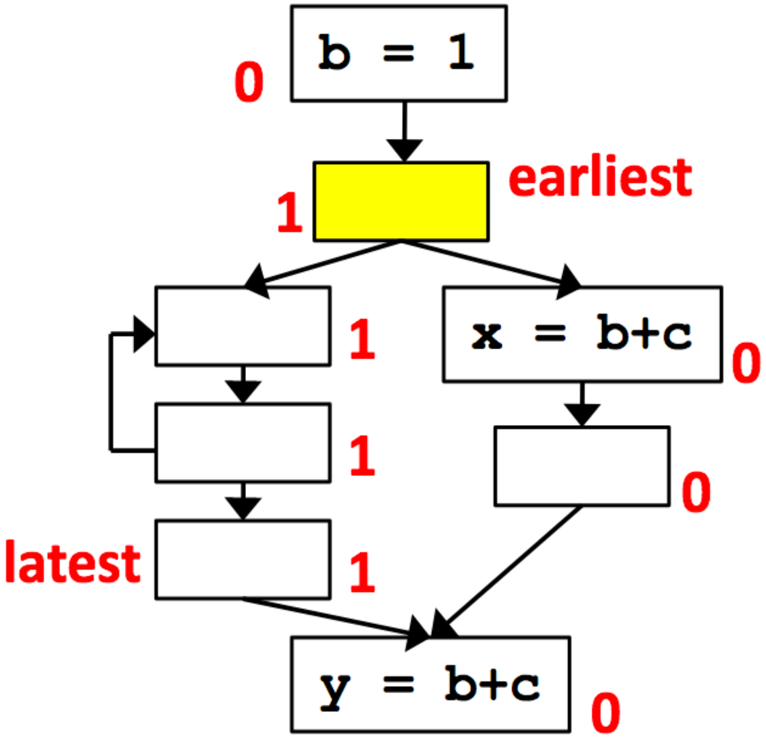
Why latest possible time?

- The values of expressions found to be redundant are usually held in registers until they are used

- Computing a value as late as possible minimizes its lifetime – the duration between the time the value is defined and the time it is last used

- Minimizing the lifetime of a value in turn minimizes the usage of a register

Postponable Expressions



☐ An expression e is postponable at a program point p if

– all paths leading to p have seen earliest placement of e

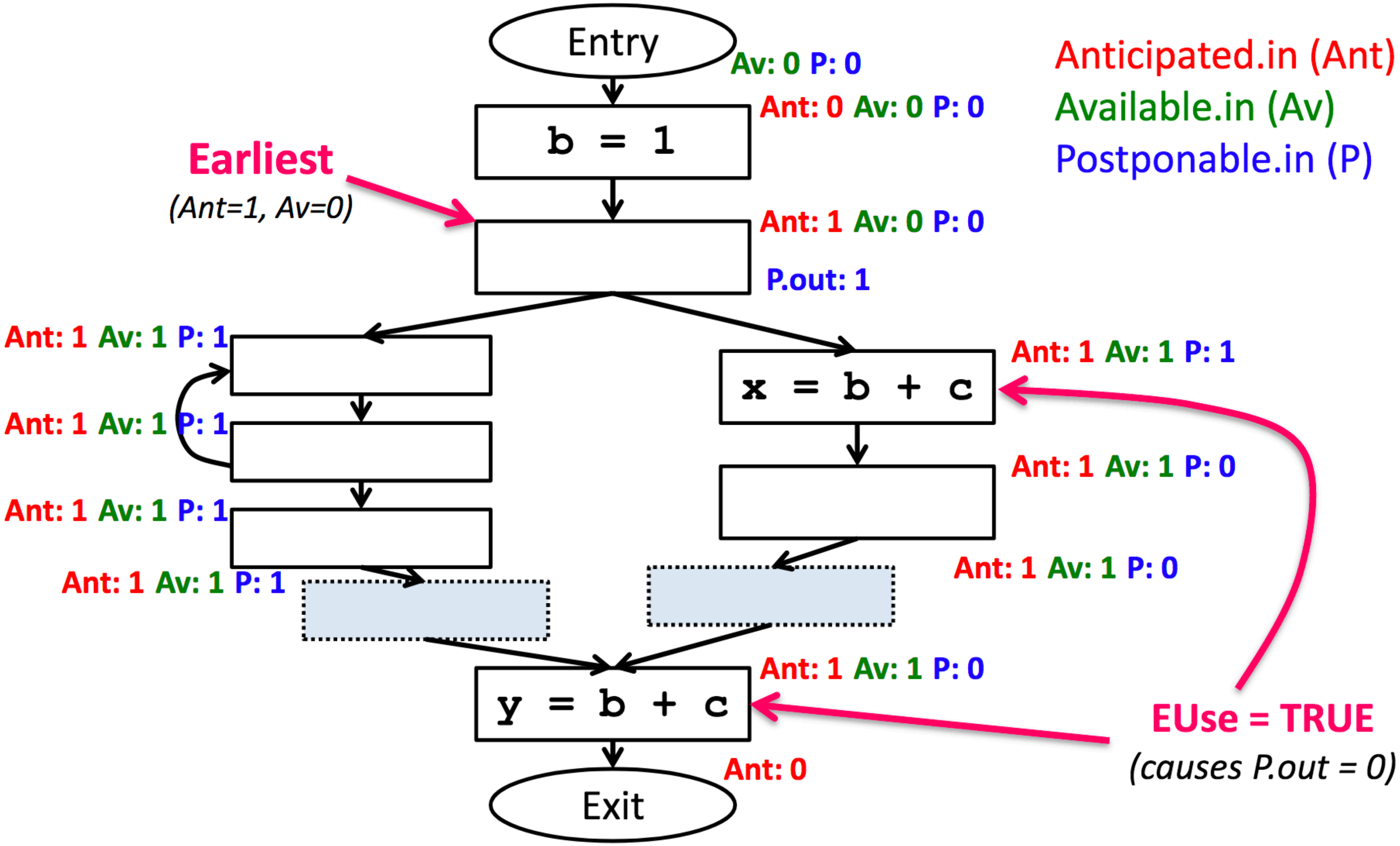
– but not a subsequent use

Postponable Expressions

Direction	Forwards
Transfer function	$f_B(x) = (earliest[B] \cup x) - e\_use_B$
Boundary	OUT[ENTRY] = $\varnothing$
Meet($\wedge$)	$\cap$
Equations	OUT[B] = f_B(IN[B]) IN[B] = $\wedge_{P,\, pred(B)}$ OUT[P]

- e-use$_B$ is the set of expressions computed but not subsequently killed (a.k.a., UEEXP).

Example Illustrating "Postponable"



Anticipated.in (Ant)

Available.in (Av)

Postponable.in (P)

Av: 0 P: 0

Ant: 0 Av: 0 P: 0

Earliest

(Ant=1, Av=0)

Ant: 1 Av: 0 P: 0

P.out: 1

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 1

Ant: 1 Av: 1 P: 0

Ant: 1 Av: 1 P: 0

Ant: 1 Av: 1 P: 0

Ant: 0

EUse = TRUE

(causes P.out = 0)

Ant.IN[i] = EUse[i] ∪ (Ant.OUT[i]-EKill[i])

Avail.OUT[i] = (Ant.IN[i] ∪ Avail.IN[i])-EKill[i]

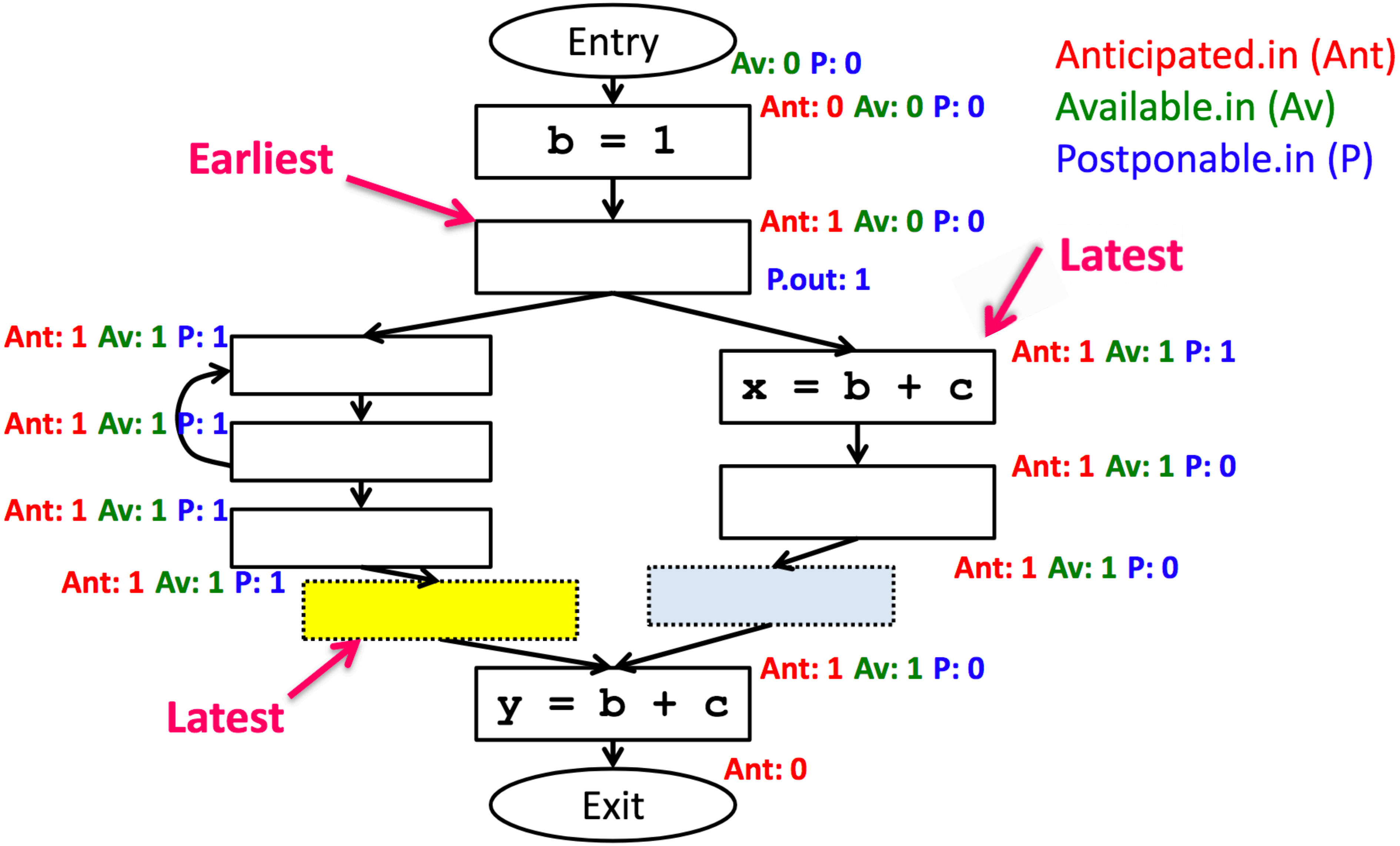
Post.OUT[i] = (Earliest[i] ∪ Post.IN[i])-EUse[i]

Latest: frontier at the end of "postponable" cut set

$$\text{latest}[b] = (\text{earliest}[b] \cup \text{postponable.in}[b]) \cap$$
$$(\text{EUse}_b \cup \neg(\bigcap_{s \in \text{succ}[b]}(\text{earliest}[s] \cup \text{postponable.in}[s])))$$

- OK to place expression: earliest or postponable
- Need to place at b if either
  - used in b or
  - not OK to place in one of its successors

Example Illustrating "Latest"



Anticipated.in (Ant)

Available.in (Av)

Postponable.in (P)

$$\text{latest}[b] = (\text{earliest}[b] \cup \text{postponable.in}[b]) \cap (\text{EUse}_b \cup \neg(\textstyle\bigcap_{s \in \text{succ}[b]}(\text{earliest}[s] \cup \text{postponable.in}[s])))$$