Partial Redundancy Elimination

Finding the Right Place to Evaluate Expressions

Four Necessary Data-Flow Problems

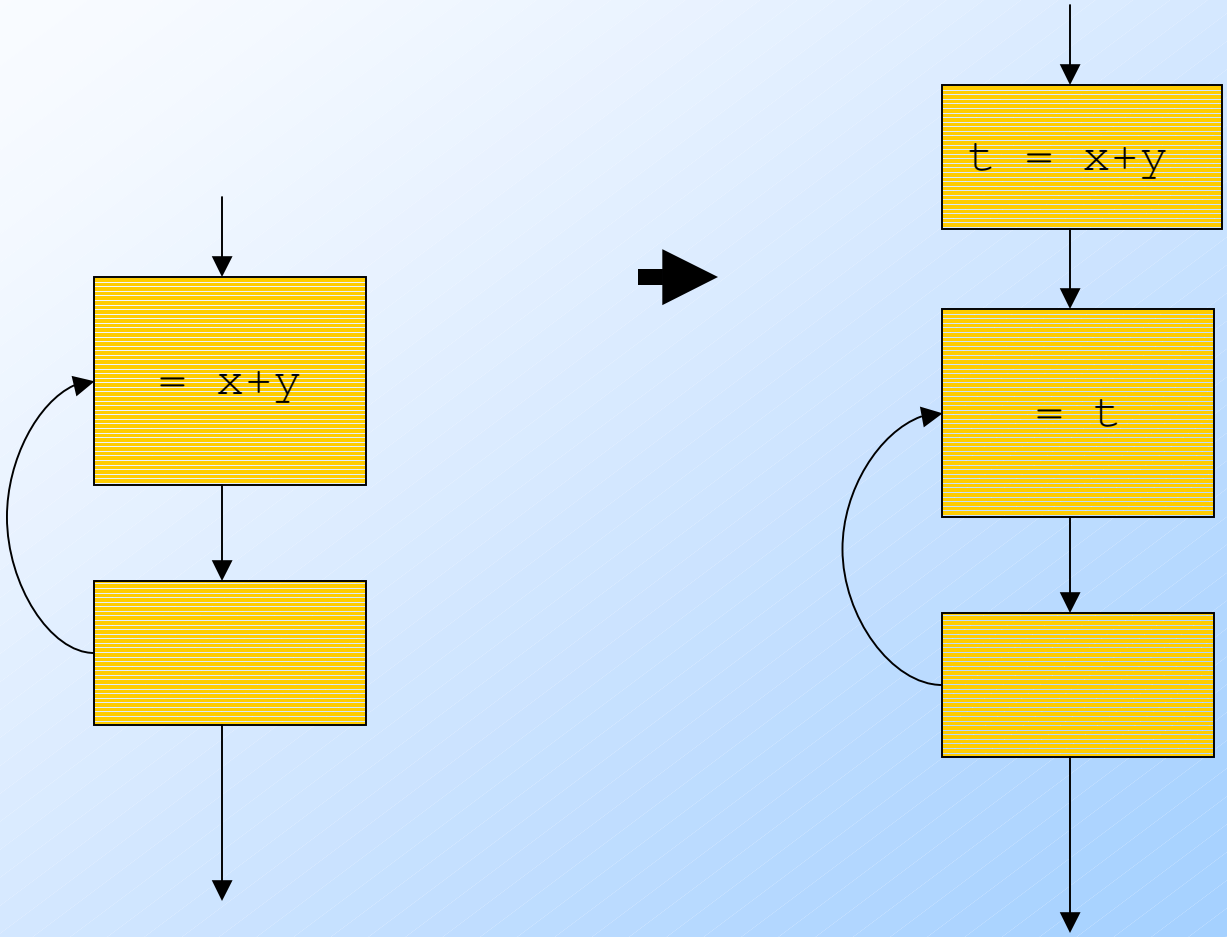
Role of PRE

- Generalizes:
  1. Moving loop-invariant computations outside the loop.
  2. Eliminating common subexpressions.
  3. True partial redundancy: an expression is sometimes available, sometimes not.

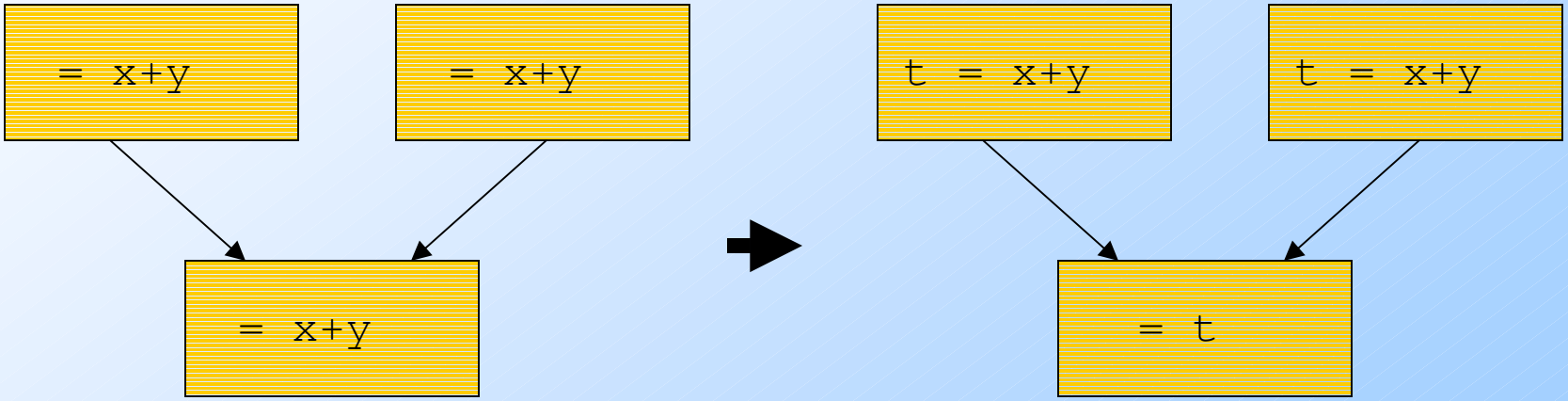
Convention

- Throughout, assume that neither argument of an expression x+y is modified unless we explicitly assign to x or y.
- And of course, we assume x+y is the only expression anyone would ever want to compute. ☺

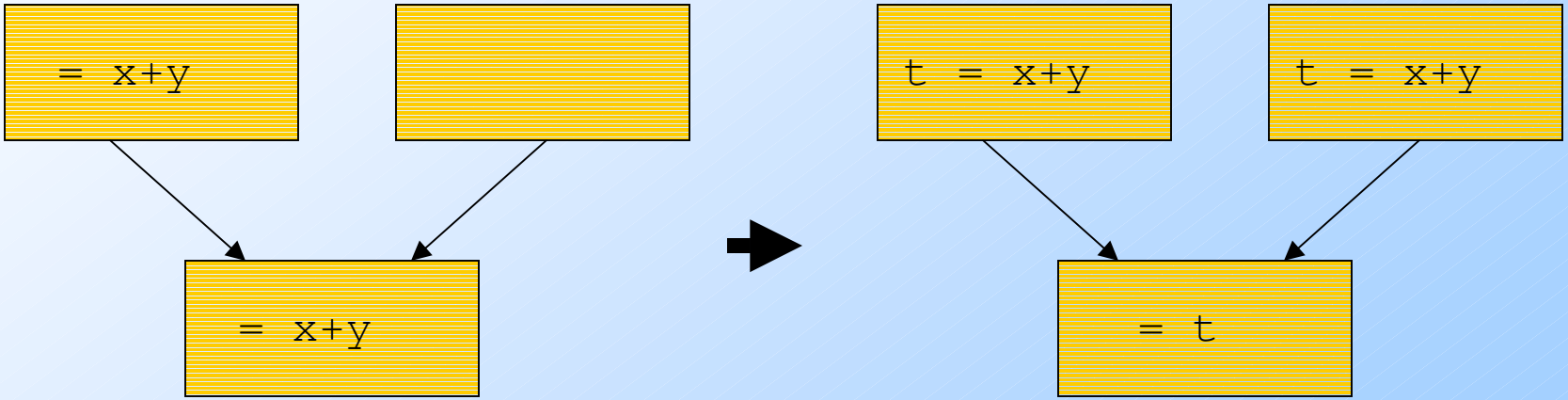
Example: Loop-Invariant Code Motion



Example: Common Subexpression Elimination



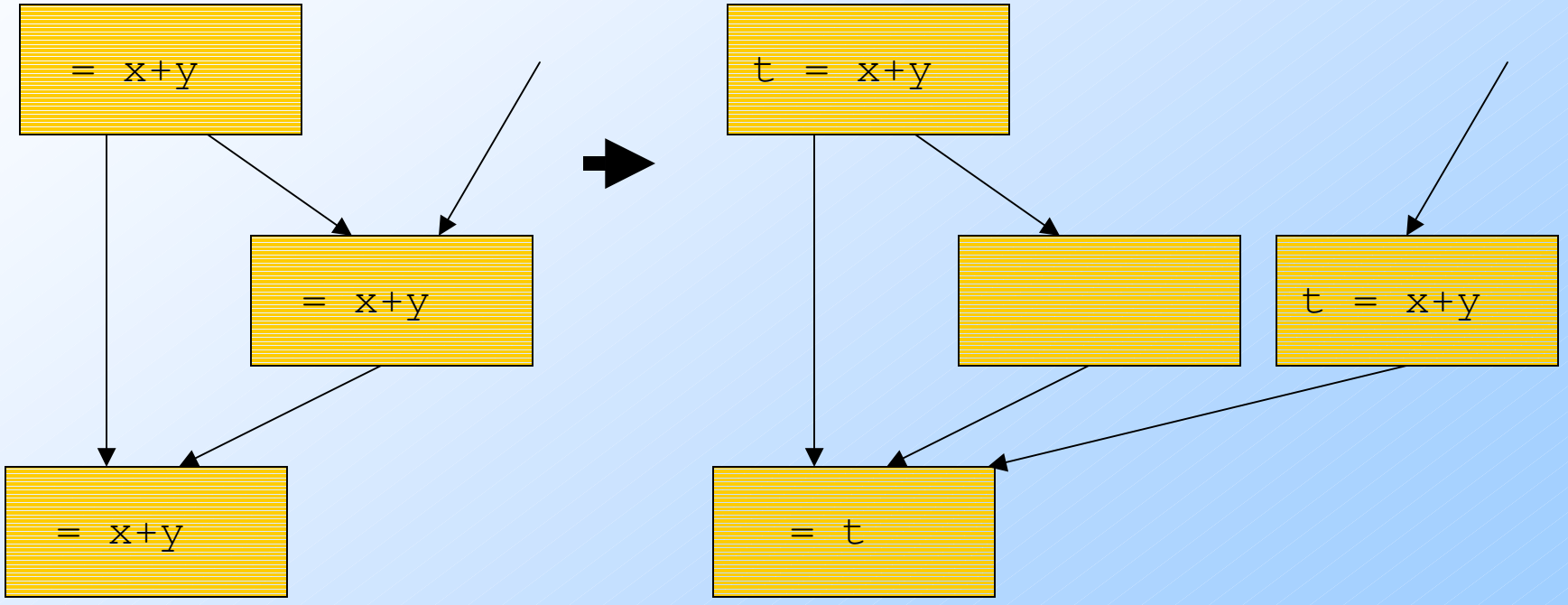
Example: True Partial Redundancy



Modifying the Flow Graph

1. Add a new block along an edge.
   - Only necessary if the edge enters a block with several predecessors.
2. Duplicate blocks so an expression x+y is evaluated only along paths where it is needed.

Example: Node Splitting



Problem With Node-Splitting

- Can exponentiate the number of nodes.
- Our PRE algorithm needs to move code to new blocks along edges, but will not split blocks.
- Convention: All new instructions are either inserted at the beginning of a block or placed in a new block.

Da Plan Boss --- Da Plan

1. Determine for each expression the earliest place(s) it can be computed while still being sure that it will be used.
2. Postpone the expressions as long as possible without introducing redundancy.
   - We trade space for time --- an expression can be computed in many places, but never if it is already computed.

The Guarantee

- No expression is computed at a place where it its value might have been computed previously, and preserved instead.
  - Even along a subset of the possible paths.

More About the Plan

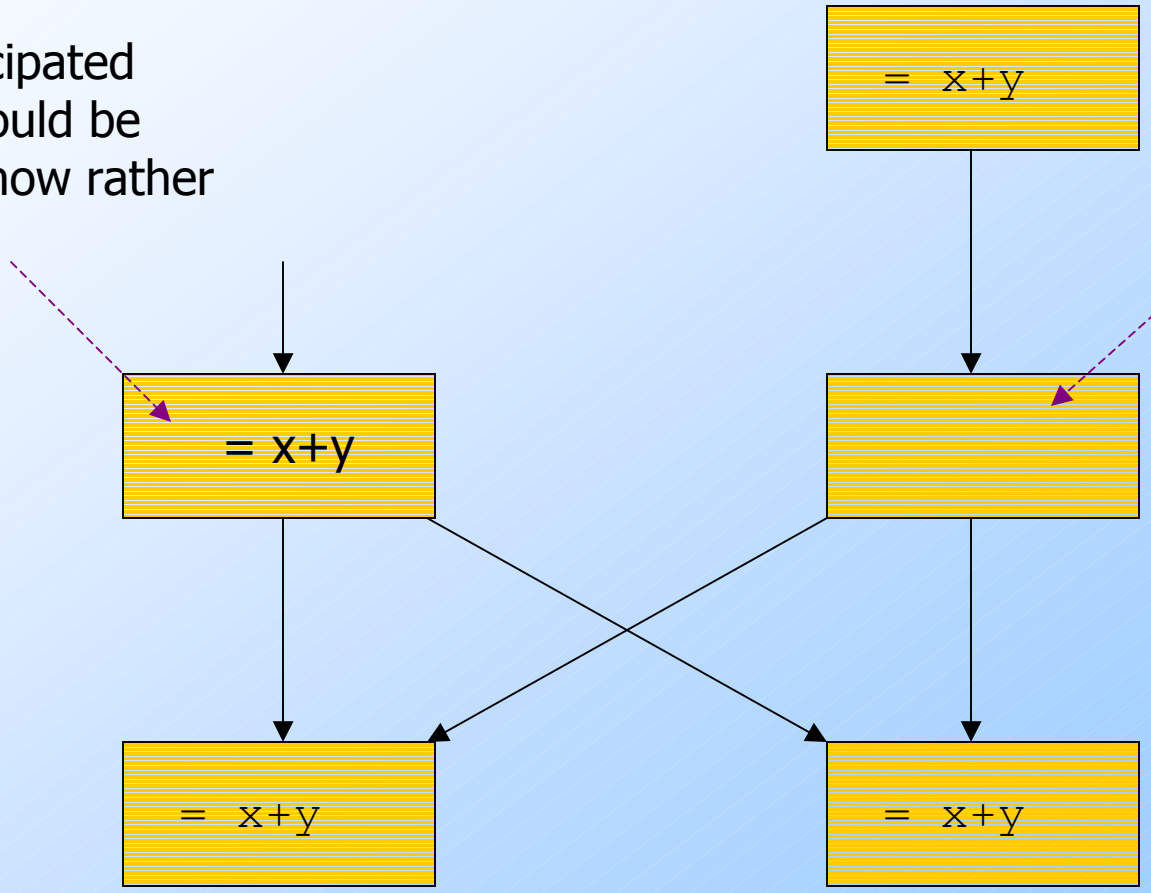
- We use four data-flow analyses, in succession, plus some set operations on the results of these analyses.
- After the first, each analysis uses the results of the previous ones in a role similar to that of Gen (for RD's) or Use (for LV's).

Anticipated Expressions

- Expression x+y is *anticipated* at a point if x+y is certain to be evaluated along any computation path, before any recomputation of x or y.
- An example of the fourth kind of DF schema: backwards-intersection.

Example: Anticipated Expressions

x+y is anticipated here and could be computed now rather than later.



x+y is anticipated here, but is also available. No computa-tion is needed.

Computing Anticipated Expressions

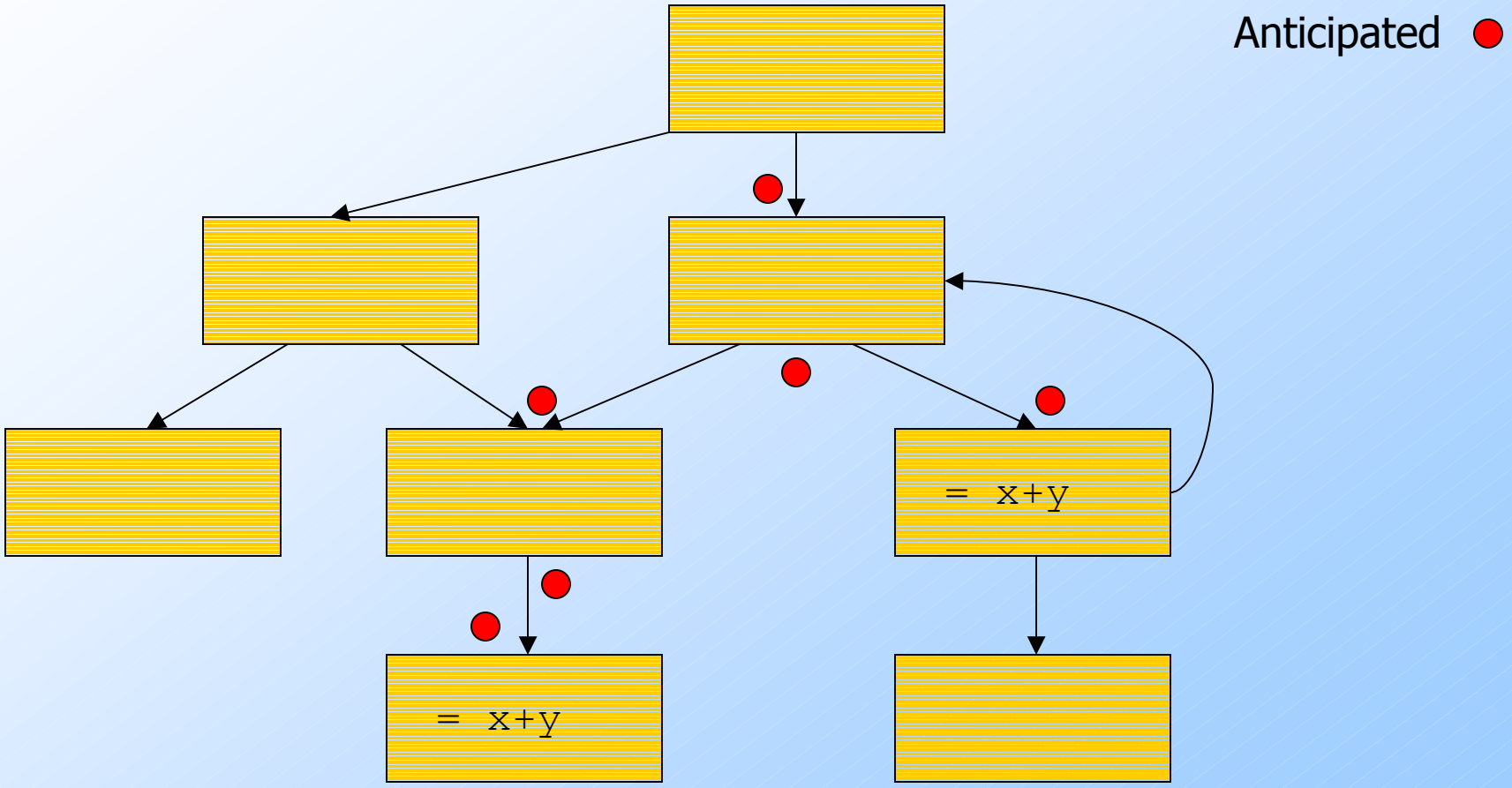
- Use(B) = set of expressions x+y evaluated in B before any assignment to x or y.
- Def(B) = set of expressions one of whose arguments is assigned in B.

Computing Anticipated Expressions --- (2)

- Direction = backwards.
- Meet = intersection.
- Boundary condition: IN[exit] = $\emptyset$.
- Transfer function:

  $IN[B] = (OUT[B] - Def(B)) \cup Use(B)$

Example: Anticipated Expressions



Backwards; Intersection; IN[B] = (OUT[B] – Def(B)) ∪ Use(B)

"Available" Expressions

- Modification of the usual AE.
- x+y is "available" at a point if either:
  1. It is available in the usual sense; i.e., it has been computed and not killed, or
  2. It is anticipated; i.e., it could be available if we chose to precompute it there.

"Available" Expressions

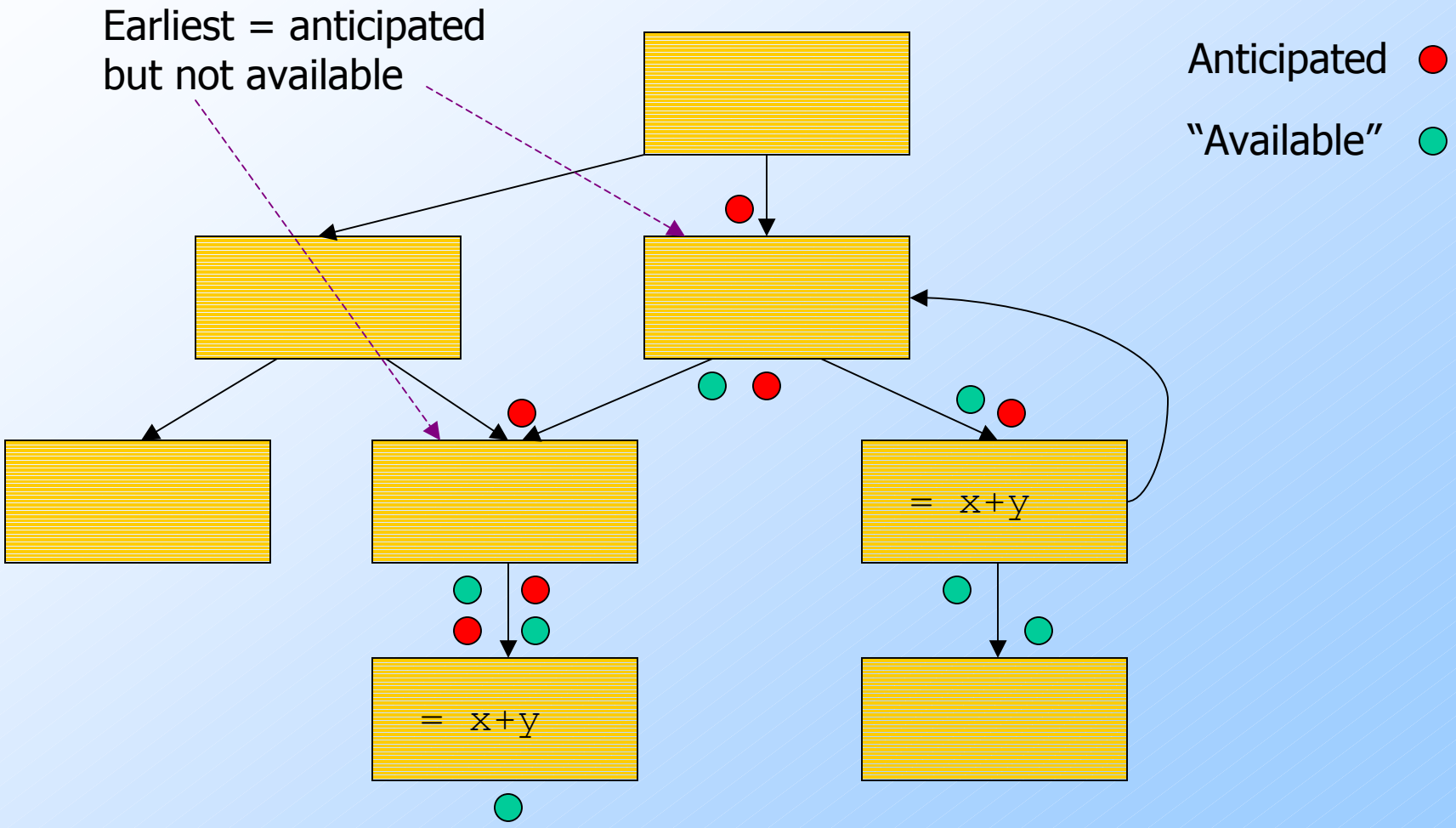
- x+y is in Kill(B) if x or y is defined, and x+y is not later recomputed (same as previously).
- Confluence = intersection.
- Transfer function:

$$OUT[B] = (IN[B] \cup IN_{ANTICIPATED}[B]) - Kill(B)$$

Earliest Placement

- x+y is in Earliest[B] if it is anticipated at the beginning of B but not "available" there.
  - That is: when we compute anticipated expressions, x+y is in IN[B], but
  - When we compute "available" expressions, x+y is not in IN[B].
- I.e., x+y is anticipated at B, but not anticipated at OUT of some predecessor.

Example: Available/Earliest



Forward; Intersection; OUT[B] = (IN[B] ∪ $IN_{ANTICIPATED}$[B]) – Kill(B)

Postponable Expressions

- Now, we need to delay the evaluation of expressions as long as possible.
  1. Not past the use of the expression.
  2. Not so far that we wind up computing an expression that is already evaluated.
- Note viewpoint: It is OK to use code space if we save register use.

Postponable Expressions --- (2)

- x+y is *postponable* to a point p if on every path from the entry to p:
  1. There is a block B for which x+y is in earliest[B], and
  2. After that block, there is no use of x+y.

Postponable Expressions --- (3)

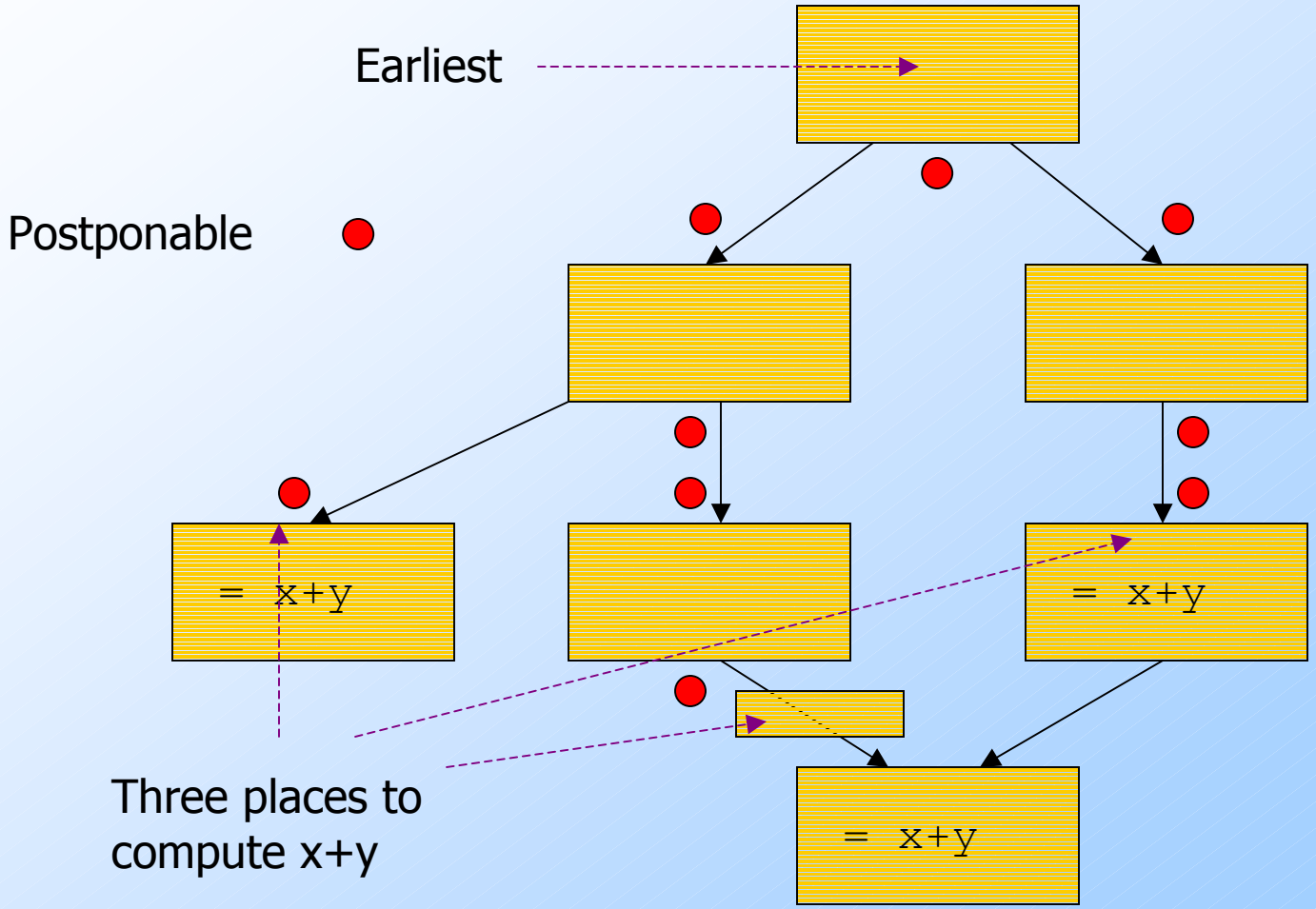
- Computed like "available" expressions, with two differences:
  1. In place of killing an expression (assigning to one of its arguments): Use(B), the set of expressions used in block B.
  2. In place of $IN_{ANTICIPATED}[B]$: earliest[B].

Postponable Expressions --- (4)

- Confluence = intersection.
- Transfer function:

$$OUT[B] = (IN[B] \cup earliest[B]) - Use(B)$$

Example: Postponable Expressions



Forward; Intersection; OUT[B] = (IN[B] ∪ earliest[B]) – Use(B)

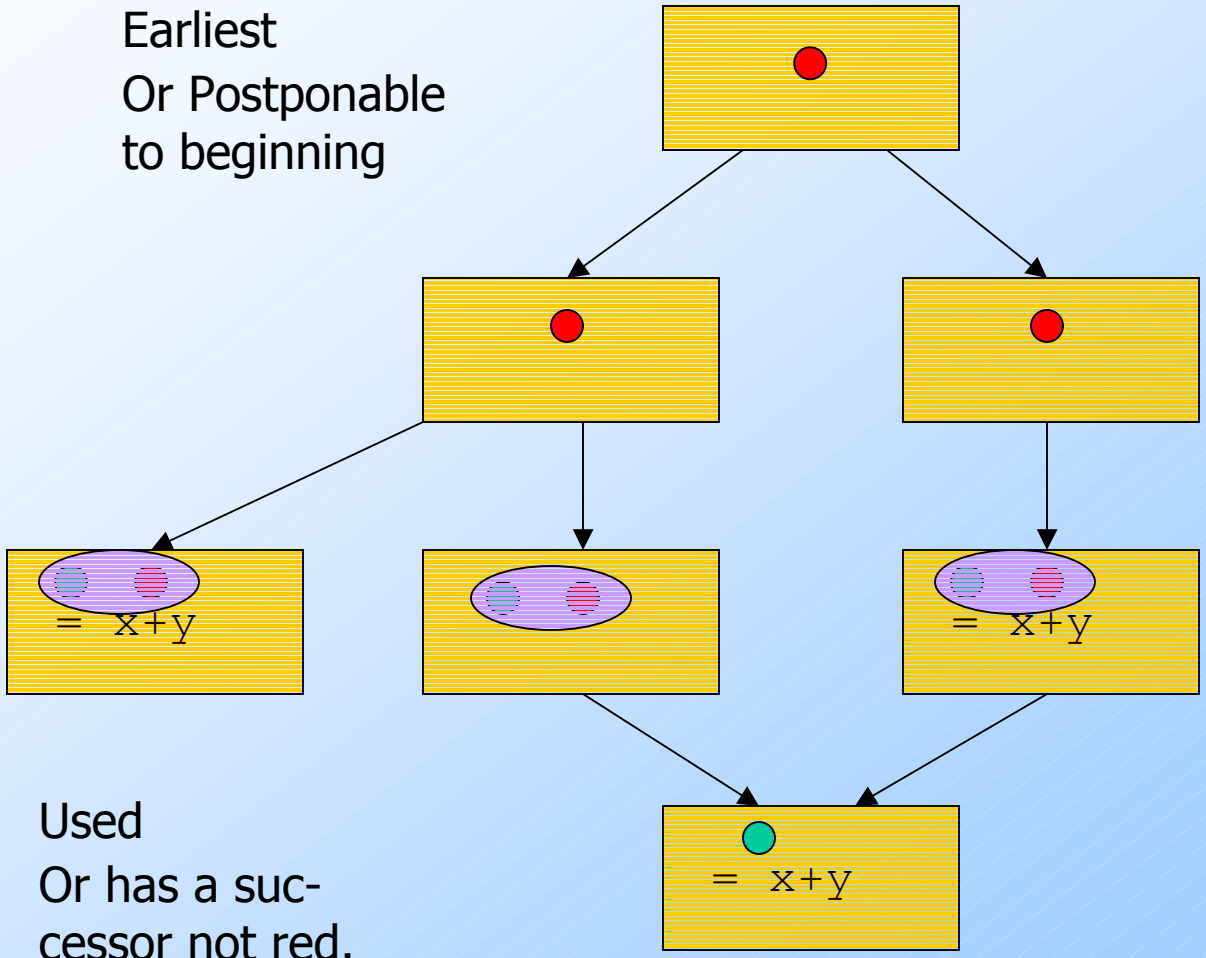
Latest Placement

- We want to postpone as far as possible.
- How do we compute the "winners" --- the blocks such that we can postpone no further?
- Remember --- postponing stops at a use or at a block with another predecessor where x+y is not postponable.

Latest[B]

- For x+y to be in latest[B]:
  1. x+y is either in earliest[B] or in $IN_{POSTPONABLE}[B]$.
     - I.e., we can place the computation at B.
  2. x+y is either used in B or there is some successor of B for which (1) does not hold.
     - I.e., we cannot postpone further along all branches.

Example: Latest



Final Touch --- Used Expressions

- We're now ready to introduce a temporary t to hold the value of expression x+y everywhere.
- But there is a small glitch: t may be totally unnecessary.
  - E.g., x+y is computed in exactly one place.

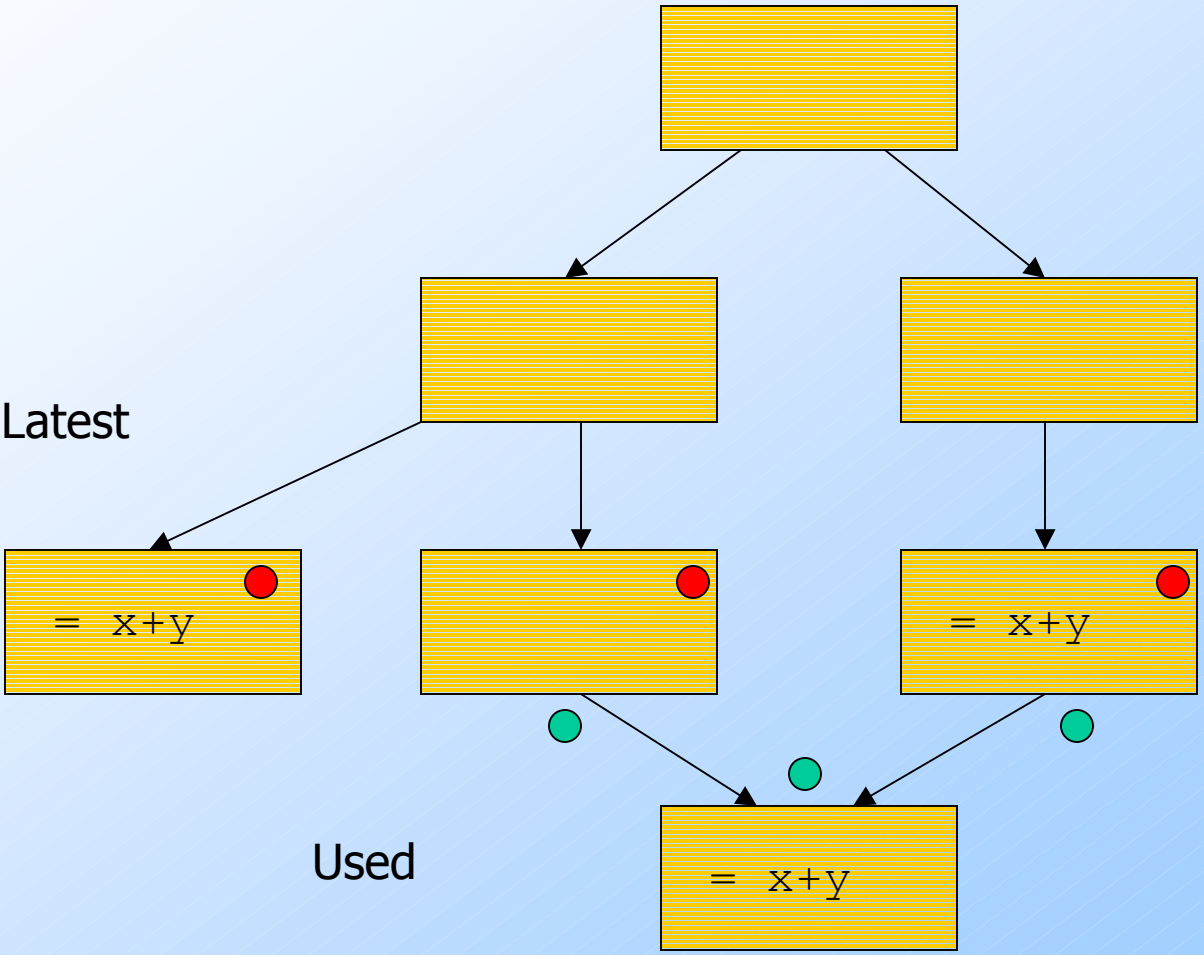
Used Expressions --- (2)

- used[B] = expressions used along some path from the exit of B.
- Backward flow analysis.
- Confluence = union.
- Transfer function:

IN[B] = (OUT[B] ∪ e-used[B]) – Latest(B)

  - e-used = "expression is used in B."

Example: Used



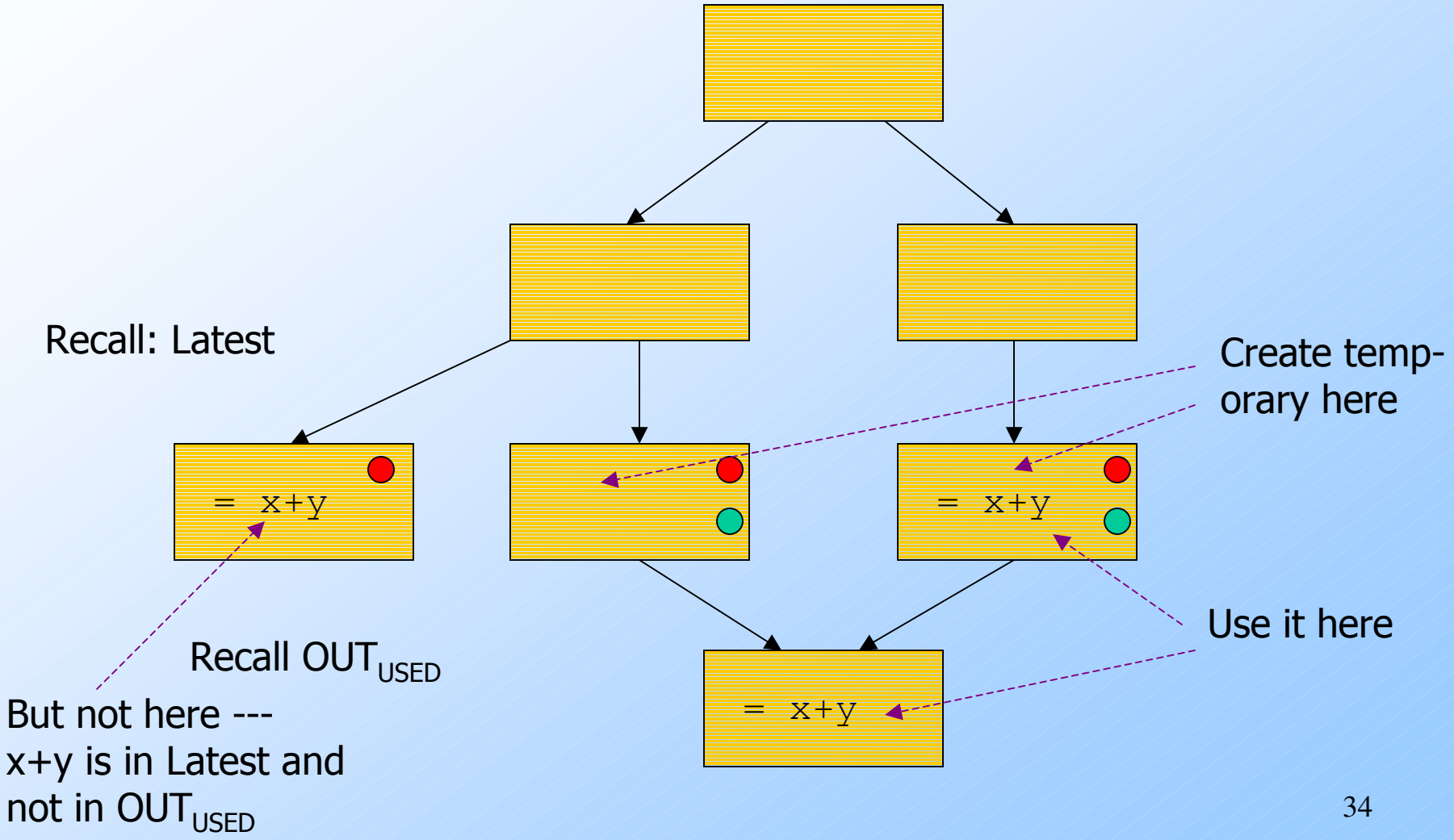
Backwards; Union; IN[B] = (OUT[B] ∪ e-used[B]) – Latest(B)

Rules for Introducing Temporaries

1. If x+y is in both Latest[B] and OUT_{USED}[B], introduce `t = x+y` at the beginning of B.
2. If x+y is used in B, but either
   1. Is not in Latest[B] or
   2. Is in OUT_{USED}[B],

   replace the use(s) of x+y by uses of t.

Example: Where is a Temporary Used?



Example: Here's Where It's Used