Partial Redundancy Elimination

15-411/15-611 Compiler Design

Seth Copen Goldstein

November 3, 2020

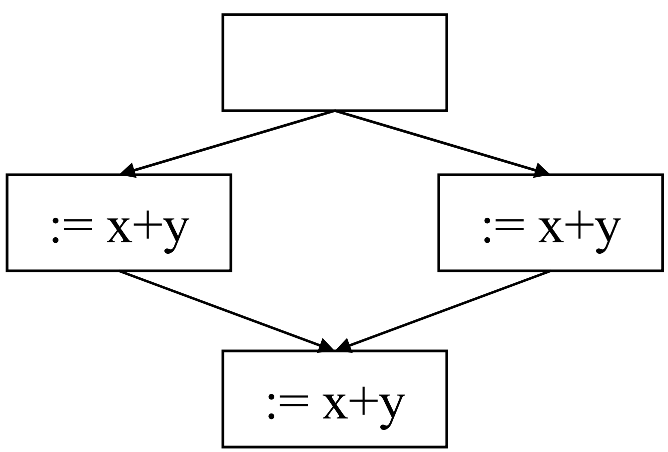
(the day has finally arrived)

Common Subexpression Elimination

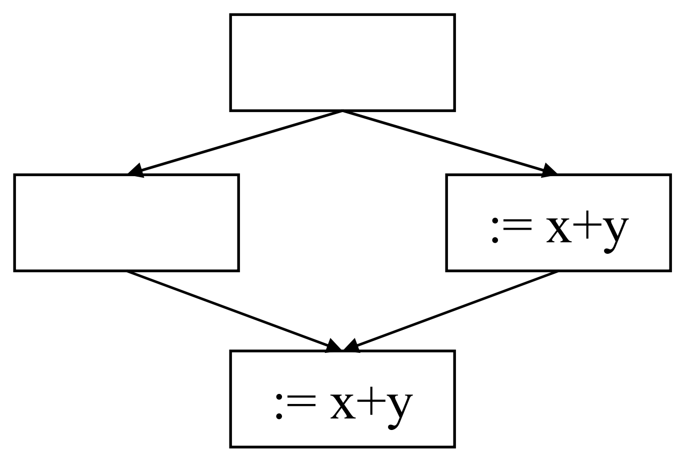
- Find computations that are always performed at least twice on an execution path and eliminate all but the first
- Usually limited to algebraic expressions
  - put in some cannonical form
- Almost always improves performance
  - except when?

CSE Limitation

- Searches for "totally" redundant expressions
  - An expression is totally redundant if it is recomputed along all paths leading to the redundant expression
  - An expression is partially redundant if it is recomputed along some but not all paths



fully redundant



partially redundant

Loop-Invariant Code Motion

- Moves computations that produce the same value on every iteration of a loop outside of the loop
- When is a statement loop invariant?
  - when all its operands are loop invariant...

Loop Invariance

- An operand is loop-invariant if
  1. it is a constant,
  2. all defs (use ud-chain) are located outside the loop, or
  3. has a single def (ud-chain again) which is inside the loop and that def is itself loop invariant
- Can use iterative algorithm to compute loop invariant statements

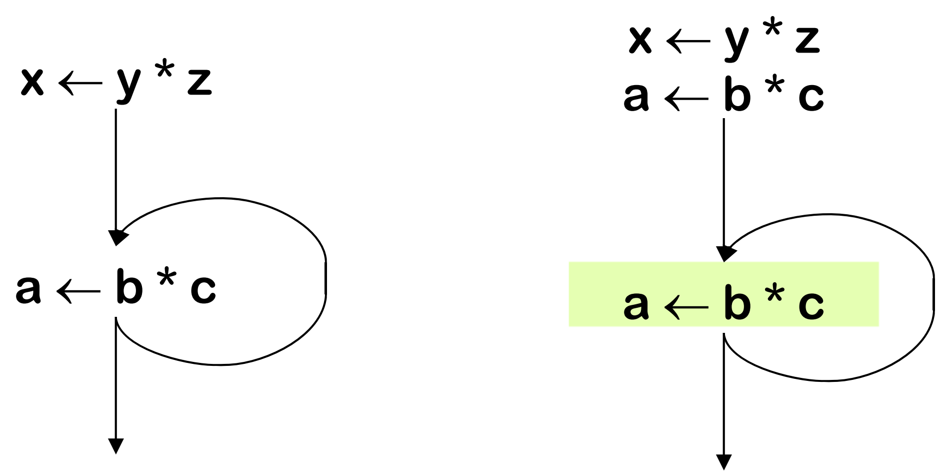
Loop Invariant Code Motion

- Naïve approach: move all loop-invariant statements to the preheader
- Not always valid for statements which define variables
- If statement s defines *v*, can only move s if
  - s dominates **all** uses of *v* in the loop
  - s dominates all loop exits

Why?

Loop Invariant Code Motion

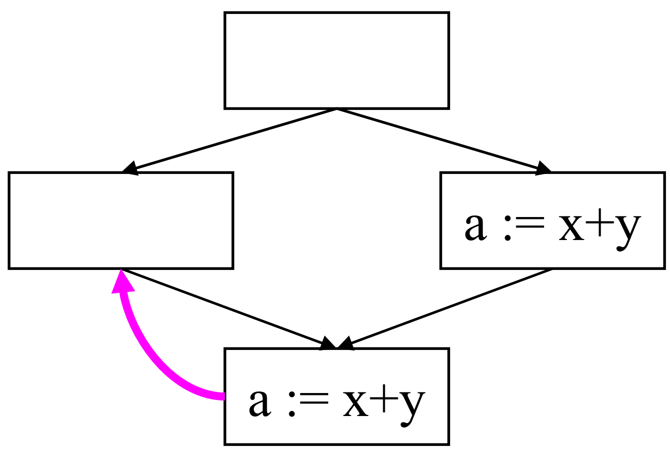
- Loop invariant expressions are a form of partially redundant expressions. Why?



*

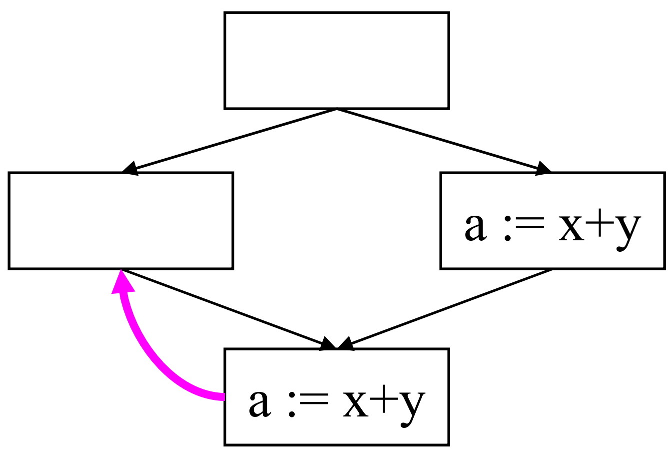
Partial Redundancy Elimination

- Moves computations that are at least partially redundant to their optimal computation points and eliminates totally redundant ones
- Encompasses CSE and loop-invariant code motion

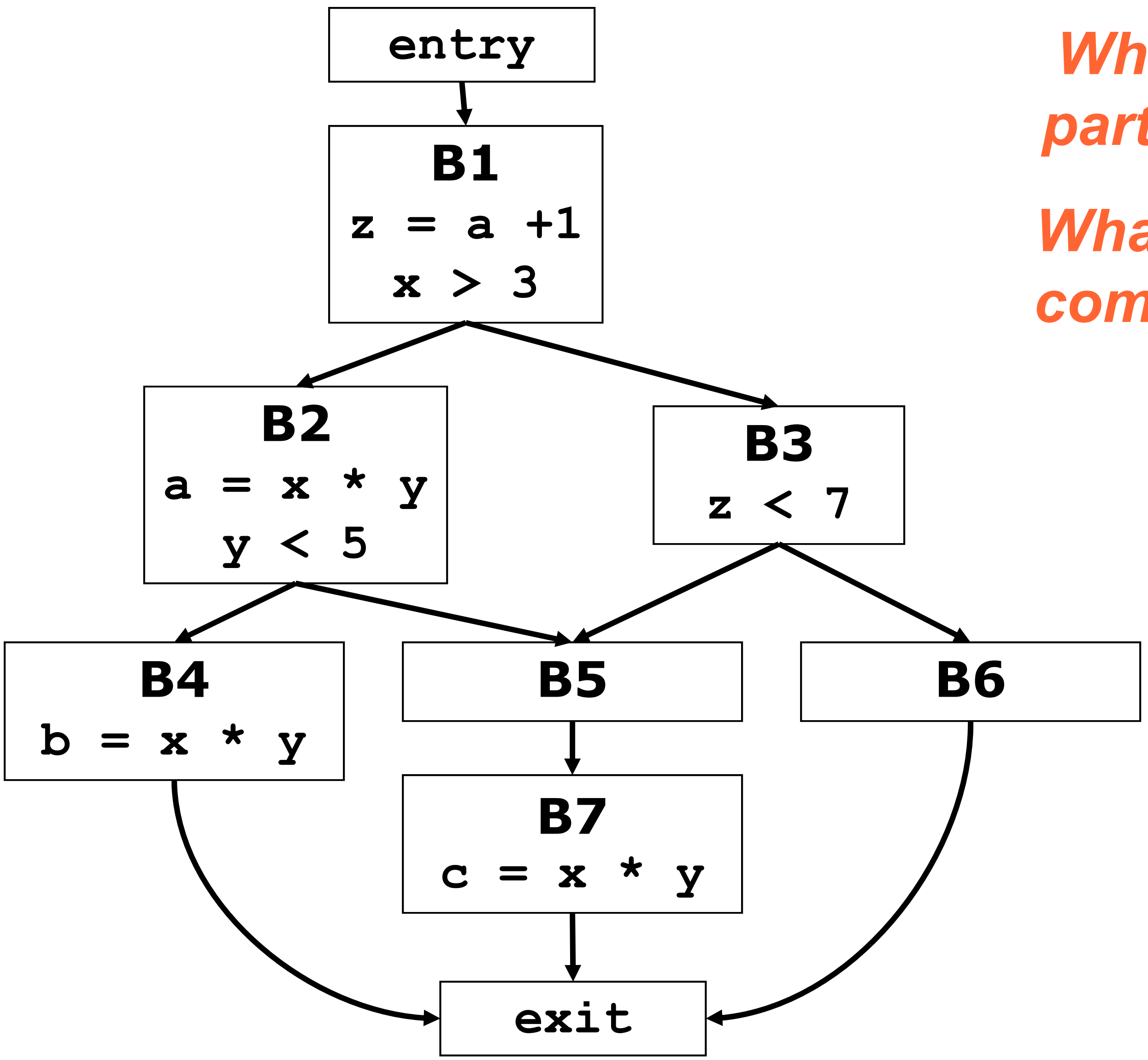


Optimal Computation Point

- Optimal?
  - Result used and never recalculated
  - Expression placed late as possible *Why?*



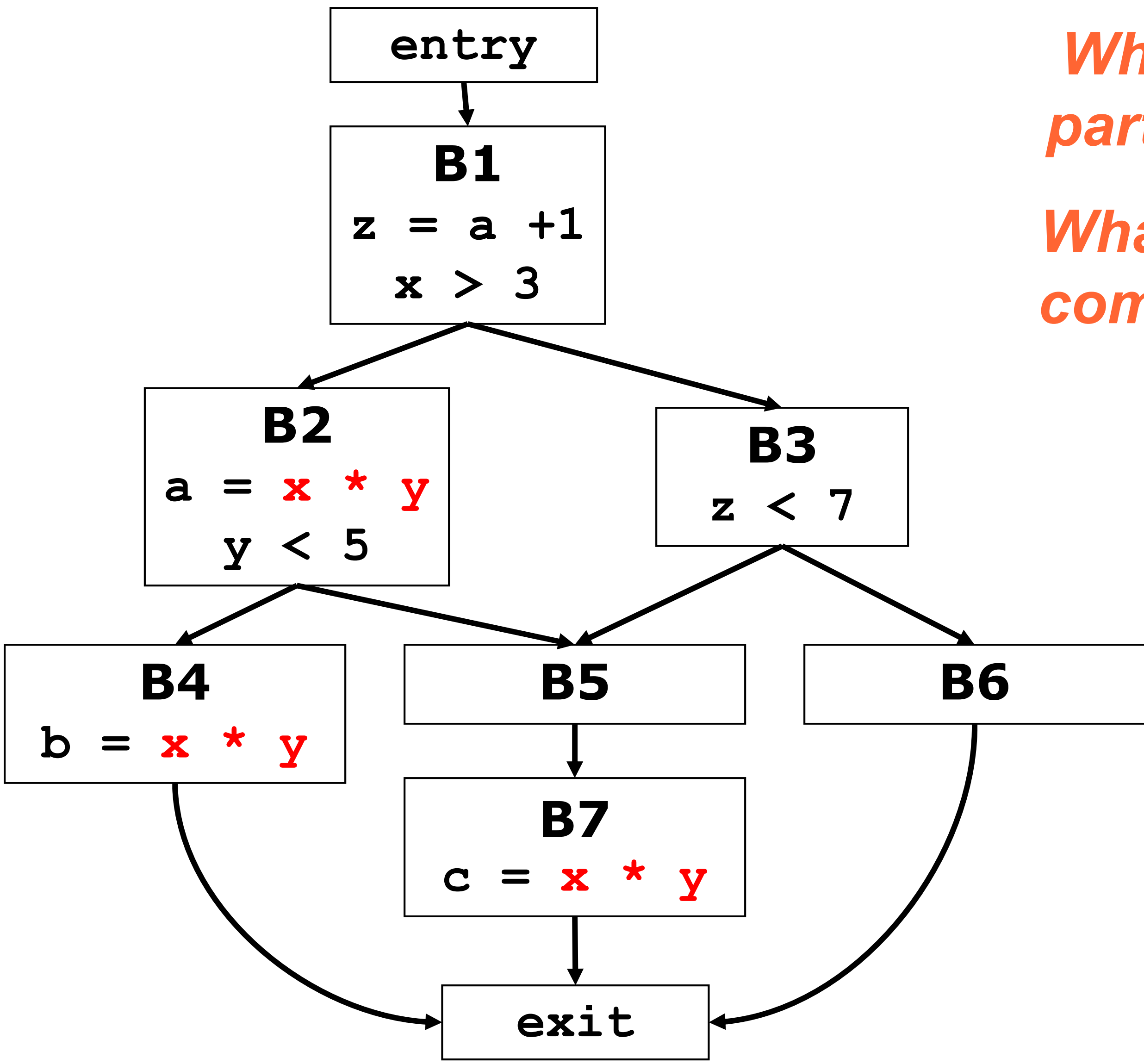
PRE Example



What expression is partially redundant?

What are the optimal computation points?

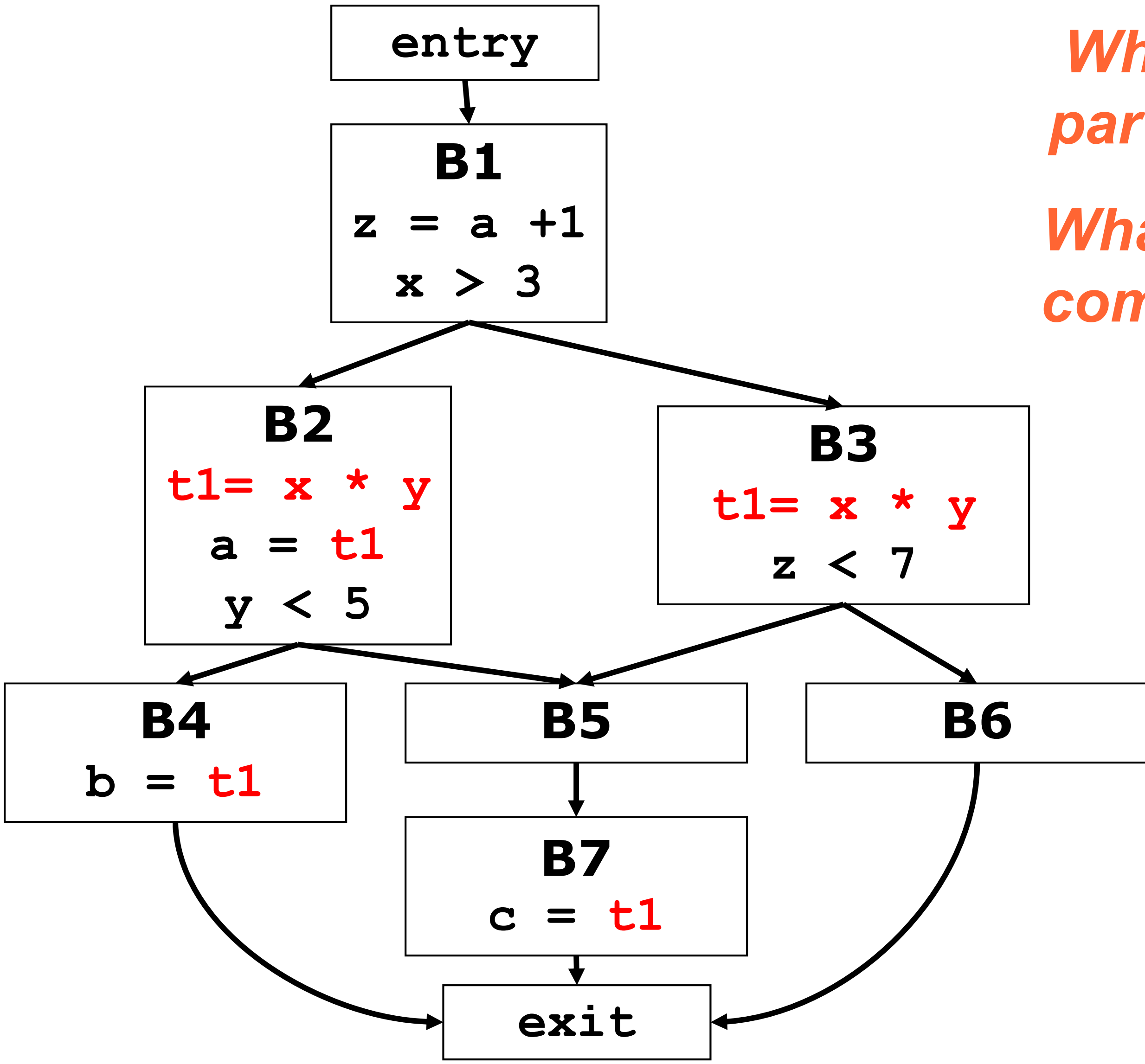
PRE Example



What expression is partially redundant?

What are the optimal computation points?

PRE Example

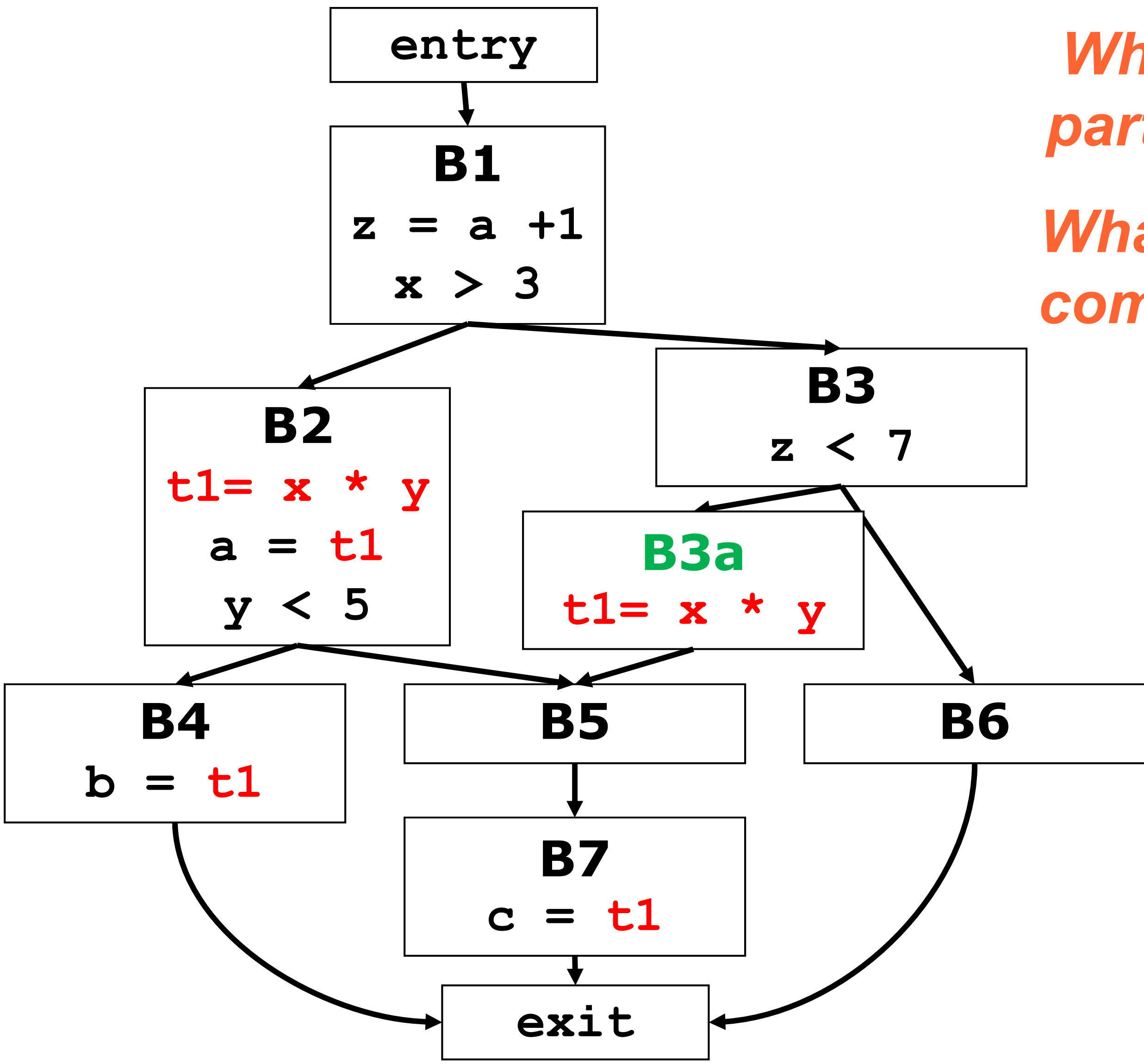


What expression is partially redundant?

What are the optimal computation points?

Not quite perfect

PRE Example

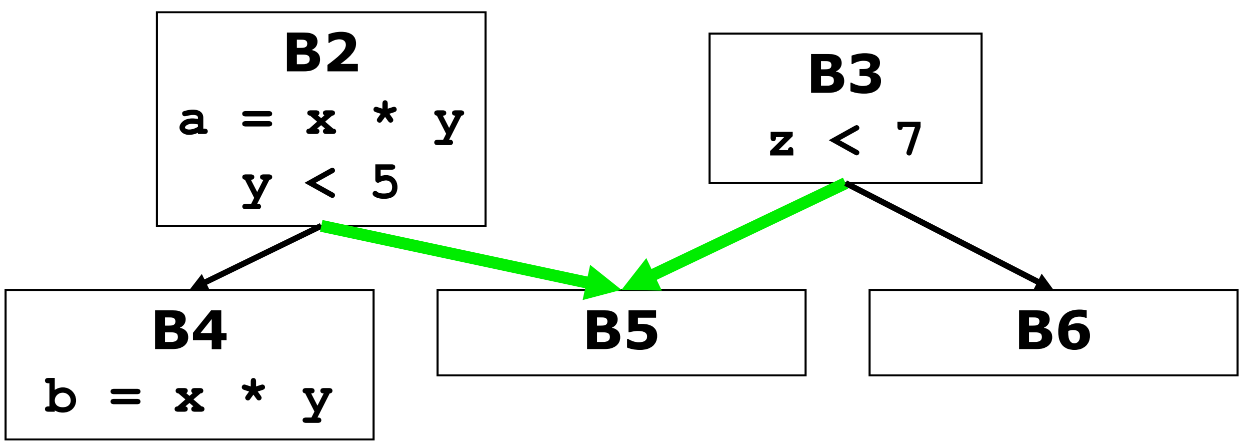


What expression is partially redundant?

What are the optimal computation points?

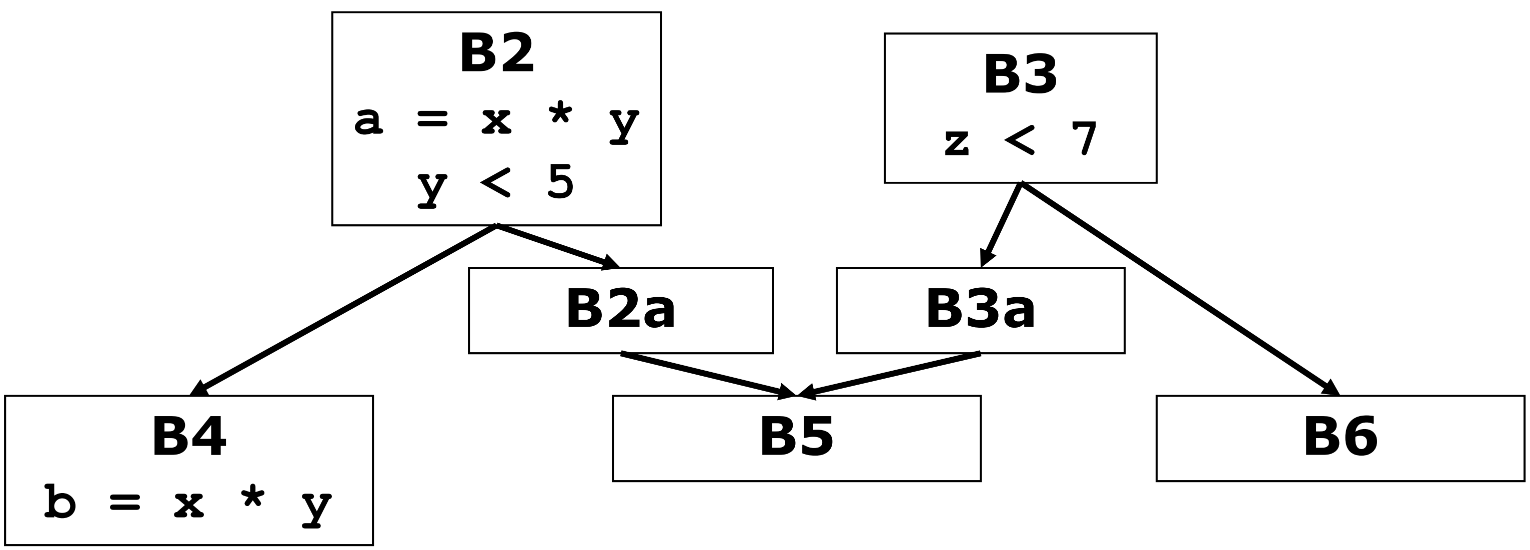
Critical Edge Splitting

- In order for PRE to work well, we must split critical edges
- A *critical edge* is an edge that connects a block with multiple successors to a block with multiple predecessors



Critical Edge Splitting

- In order for PRE to work well, we must split critical edges
- A *critical edge* is an edge that connects a block with multiple successors to a block with multiple predecessors



PRE History

- PRE was first formulated as a *bidirectional* data flow analysis by Morel and Renvoise in 1979
- Knoop, Rüthing, and Steffen came up with a way to do it using several unidirectional analysis in 1992 (called their approach *lazy code motion*)
  - this is a much simpler method
  - but it is still very complicated

The 60K Plan

- **Find Earliest but useful**
  Determine for each expression the earliest place(s) it can be computed while still guaranteeing that it will be used
- **Find latest but needed**
  Postpone the expression as long as possible without introducing redundancy

- Trading size for speed
  An expression may be computed in many places, but never if already computed

General Approach to analysis

- Computationally Optimal Placement
  - Anticipatable
    computing exp is useful along any path to exit
  - Earliest
    p is the earliest point to compute exp
  - Compute exp Ant $\cap$ Earliest
  - Increases register Pressure
- Lazy Code Motion

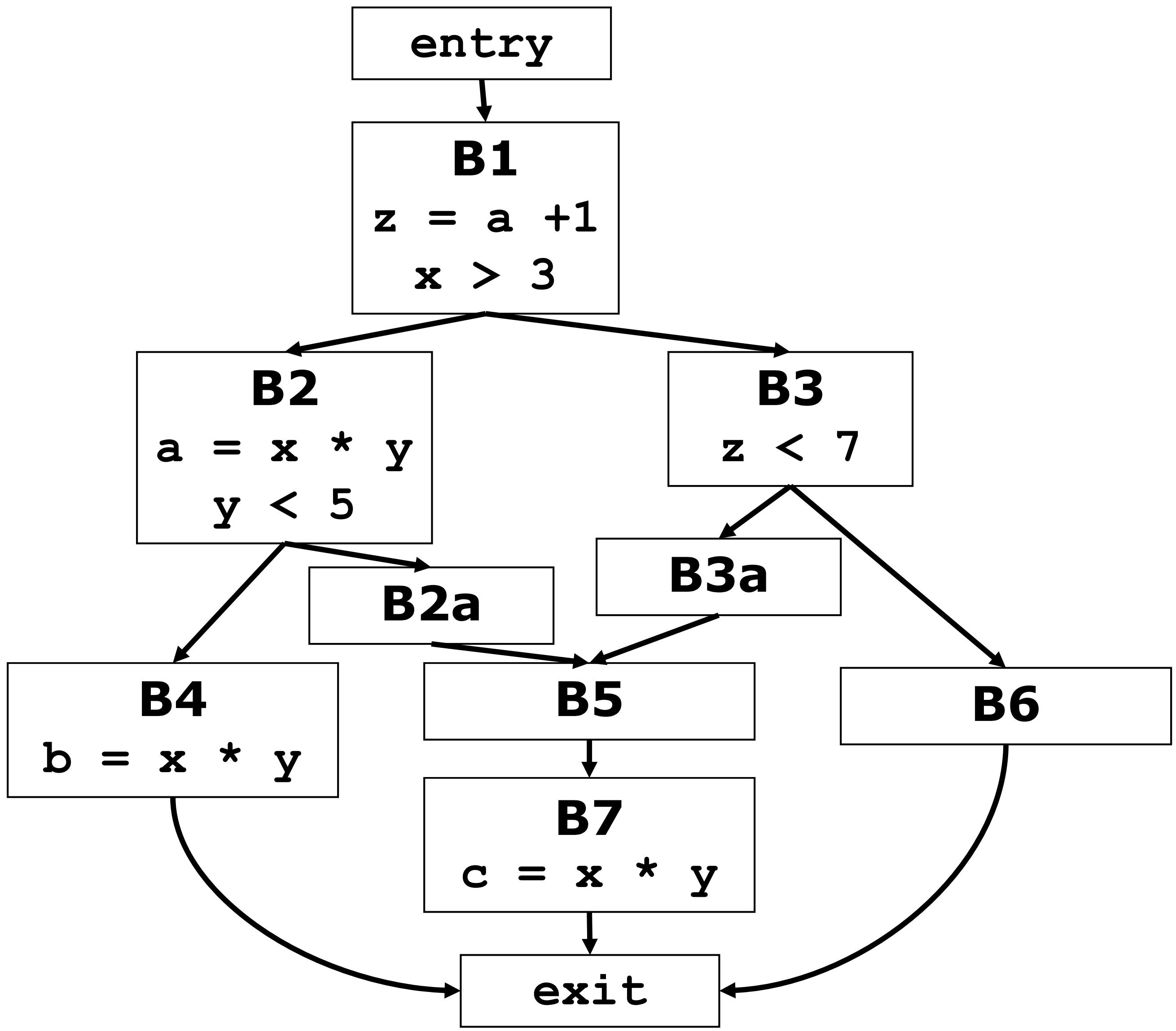
General Approach to analysis

- Computationally Optimal Placement
- Lazy Code Motion
  - Latest
    Cannot move exp past p on any path
  - Isolated
    all uses of exp follow immediately after p
  - Compute exp at Latest $\cap$ ~Isolated

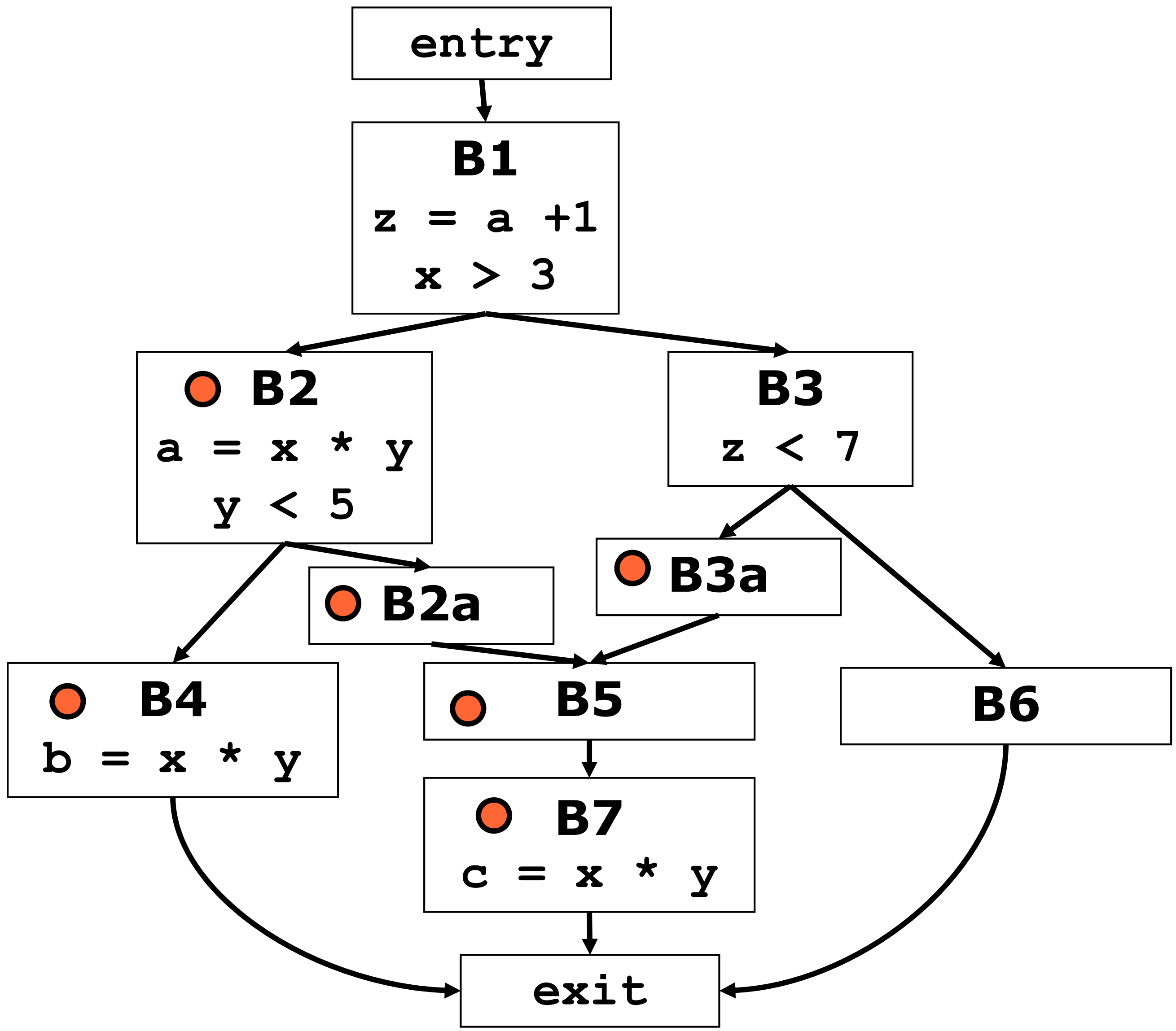
Anticipatable Expressions

- Expression **`x+y`** is anticipated at a point p if **`x+y`** is guaranteed to be computed along any path from p->exit before any recomputation of **`x`** or **`y`**
- What kind of data flow is this?
  - backwards
  - intersection

Anticipated for 'x*y'?

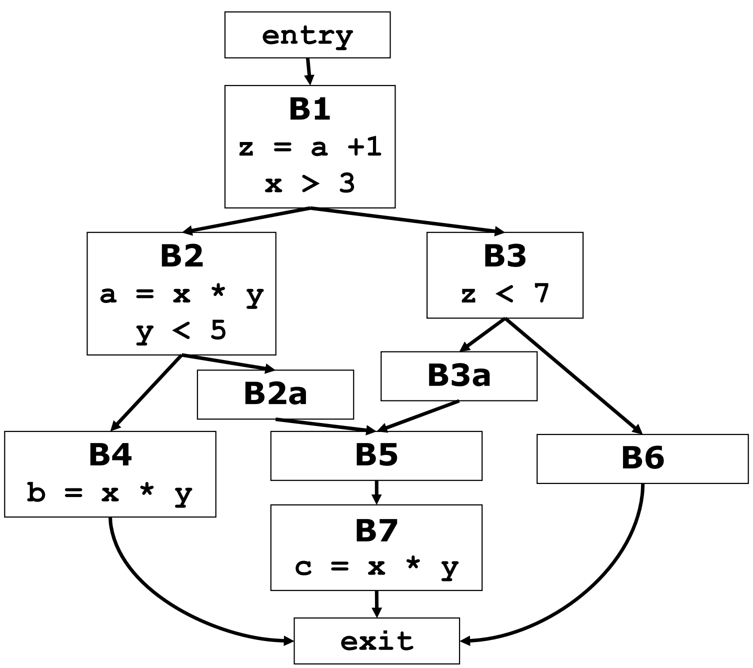


Anticipated for ‘x*y’?



Local Transparency (TRANSloc)

- An expression's value is *locally transparent* in a block if there are no assignments in the block to variables within the expression
  - ie, expression not killed

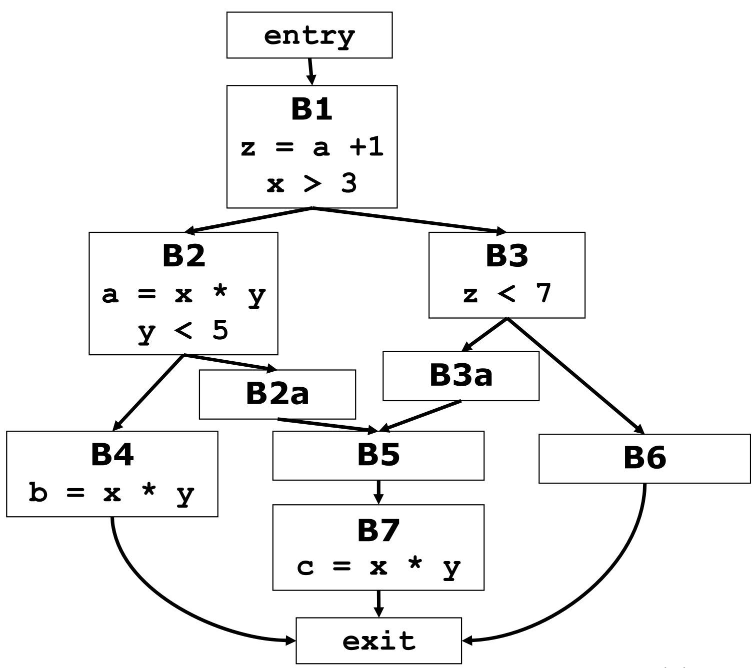


Block	TRANSloc
entry	{a+1,x*y}
B1	{a+1,x*y}
B2	{x*y}
B2a	{a+1,x*y}
B3	{a+1,x*y}
B3a	{a+1,x*y}
B4	{a+1,x*y}
B5	{a+1,x*y}
B6	{a+1,x*y}
B7	{a+1,x*y}
exit	{a+1,x*y}

Local Anticipatable (ANTloc)

- An expression's value is *locally anticipatable* in a block if
  - there is a computation of the expression in the block
  - the computation can be safely moved to the beginning of the block

Block	ANTloc
entry	{}
B1	{a+1}
B2	{x*y}
B2a	{}
B3	{}
B3a	{}
B4	{x*y}
B5	{}
B6	{}
B7	{x*y}
exit	{}



Globally Anticipatable (ANT)

- An expression's value is *globally anticipatable* on entry to a block if
  - every path from this point includes a computation of the expression
  - it would be valid to place a computation of an expression anywhere along these paths

This is like liveness, only for expressions

Globally Anticipatable (ANT)

$$ANTin(i) = ANTloc(i) \cup \left(TRANSloc(i) \cap ANTout(i)\right)$$

$$ANTout(i) = \bigcap_{j \in succ(i)} ANTin(j)$$

$$ANTout(exit) = \{\}$$

Block	ANTin	ANTout
entry	{a+1}	{a+1}
B1	{a+1}	{}
B2	{x*y}	{x*y}
B2a	{x*y}	{x*y}
B3	{}	{}
B3a	{x*y}	{x*y}
B4	{x*y}	{}
B5	{x*y}	{x*y}
B6	{}	{}
B7	{x*y}	{}
exit	{}	{}

Earliest (EARL)

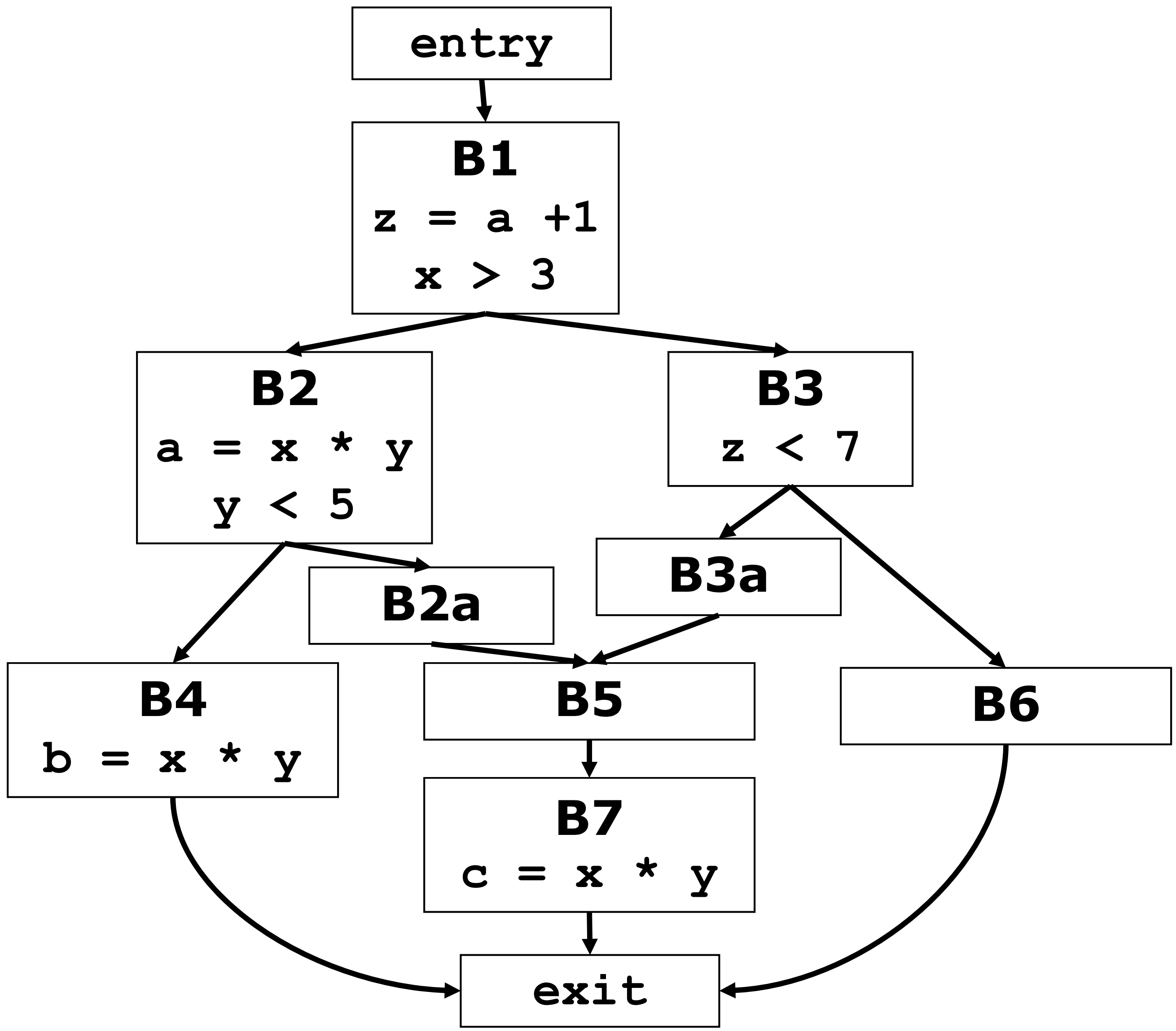
- An expression's value is *earliest* on entry to a block if
  - **no** path from entry to the block evaluates the expression to produce the same value as evaluating it at the block's entry would

Intuition:

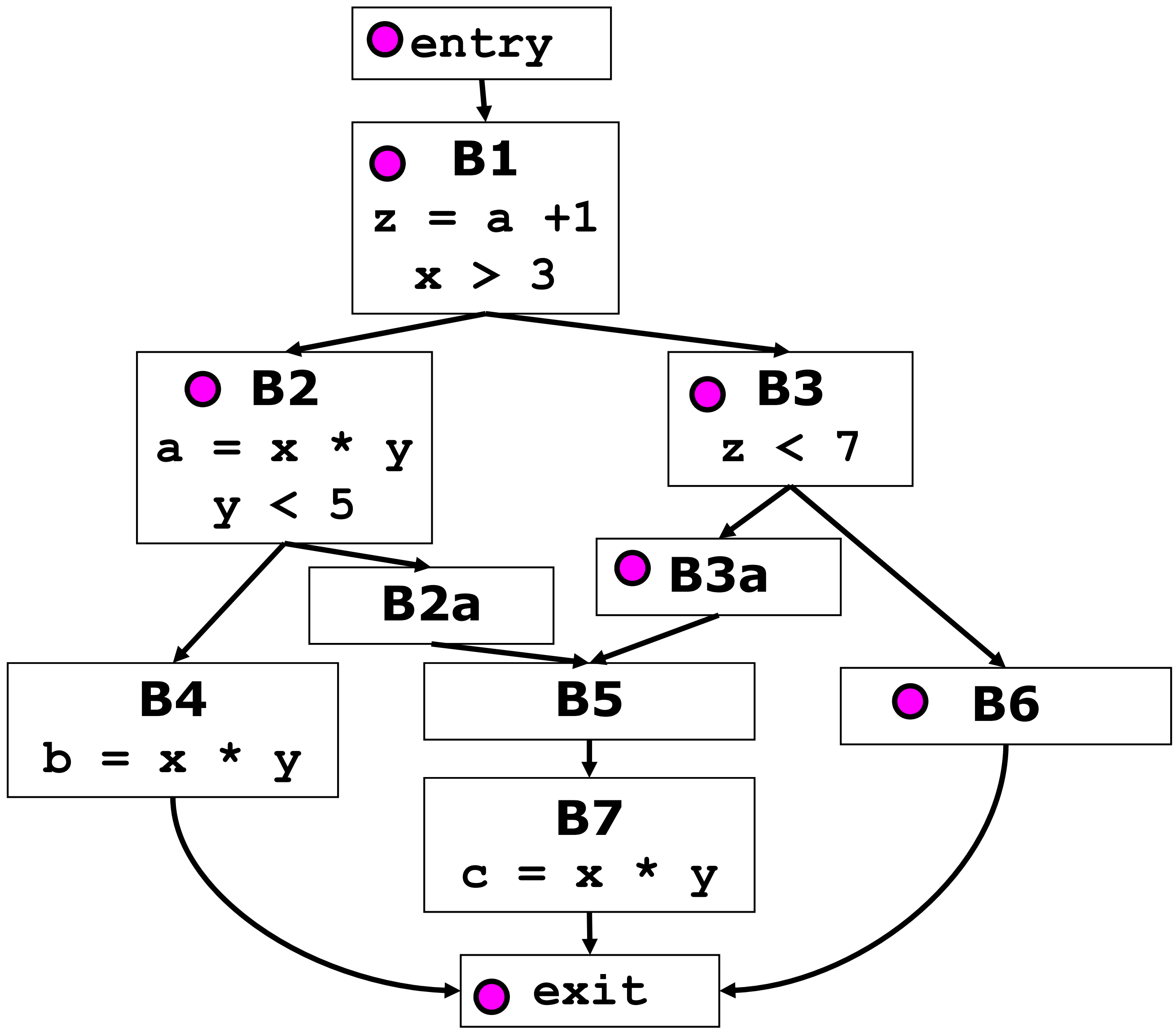
at this point if we compute the expression we are computing something completely new

says nothing about usefulness of computing expression

Earliest for ‘x*y’?



Earliest for ‘x*y’?



Earliest (EARL)

$$EARLin(i) = \bigcup_{j \in pred(i)} EARLout(j)$$

$$EARLout(i) = \overline{TRANSloc(i)} \cup \left(\overline{ANTin(i)} \cap EARLin(i) \right)$$

$$EARLin(entry) = \mathrm{U}$$

Block	EARLin	EARLout
entry	{a+1,x*y}	{x*y}
B1	{x*y}	{x*y}
B2	{x*y}	{a+1}
B2a	{a+1}	{a+1}
B3	{x*y}	{x*y}
B3a	{x*y}	{}
B4	{a+1}	{a+1}
B5	{a+1}	{a+1}
B6	{x*y}	{x*y}
B7	{a+1}	{a+1}
exit	{a+1,x*y}	{a+1,x*y}

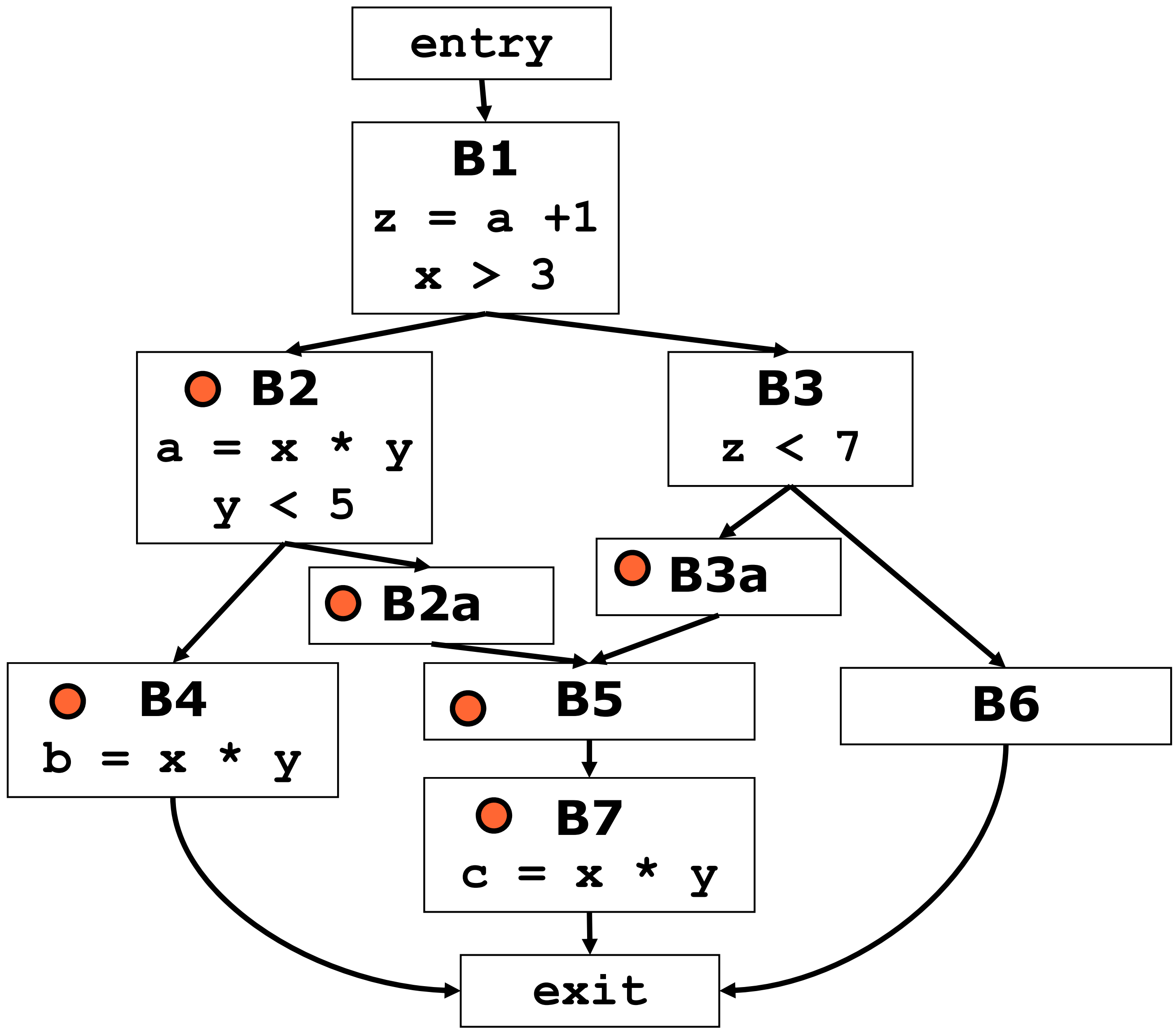
Computationally Optimal

- It is computationally optimal to compute exp at entry to block if

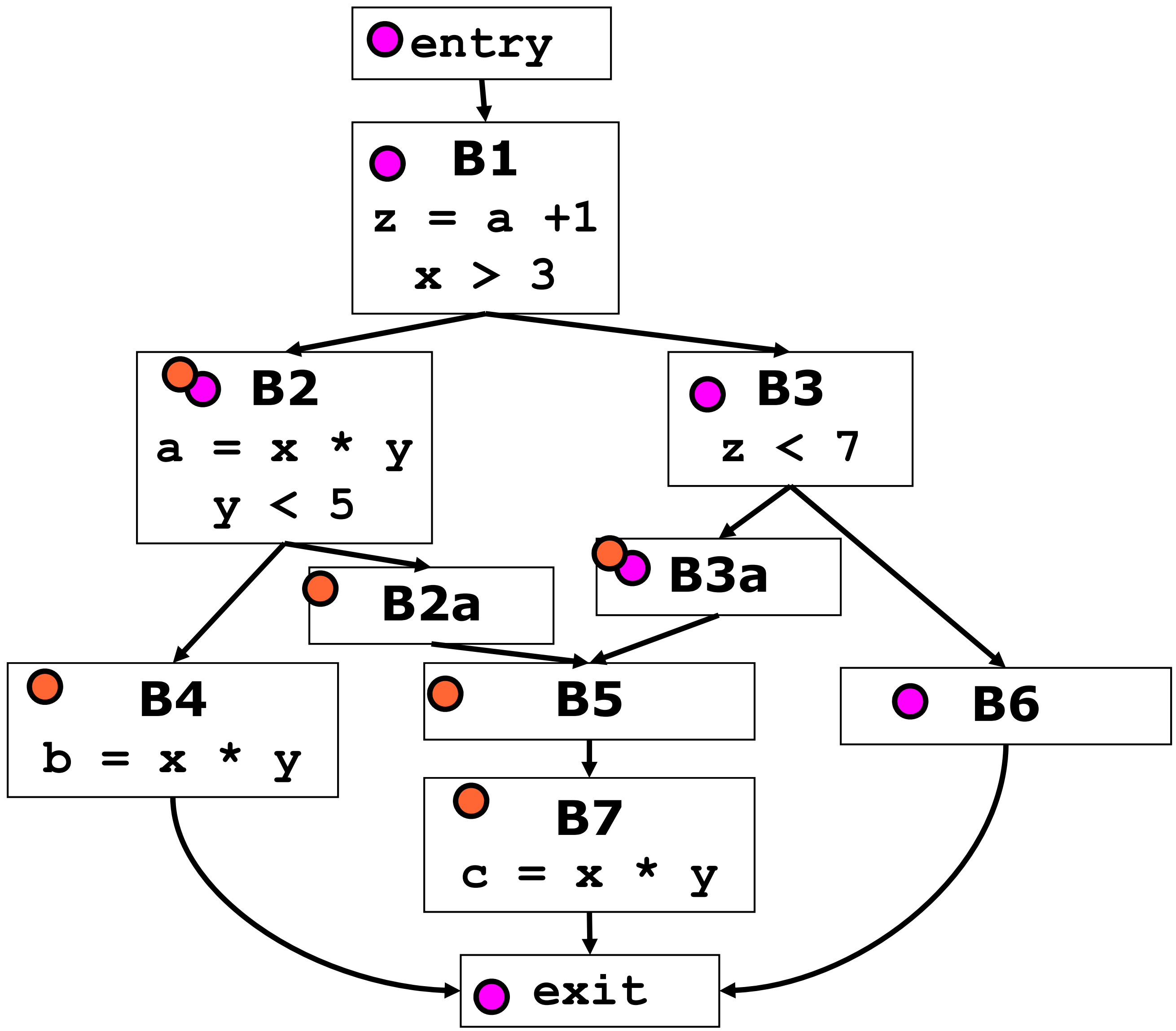
$$\exp \in ANTin(block) \bigcap EARLin(block)$$

- But, it may increase register pressure.

Anticipated for 'x*y'?



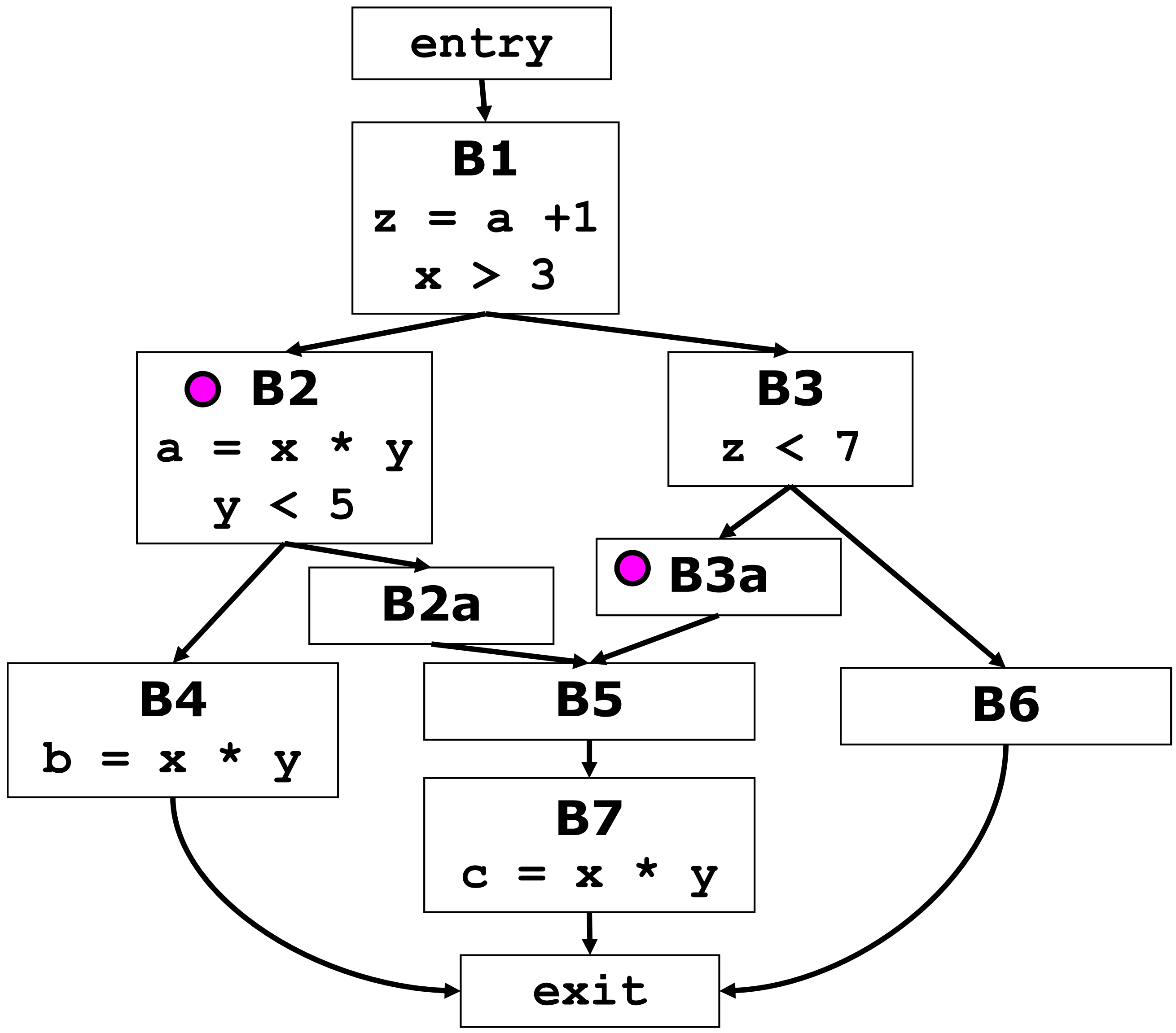
Anticipated & Early



Delayedness (DELAY)

- An expression is *delayed* on entry to a block if
  - All paths from entry to block contain an anticipatable and early computation of exp (could be this block) AND all uses of exp follow this block.
  - I.e., exp can be delayed to at least this block.

Delayed for ‘x*y’



Delayedness (DELAY)

$$DELAYin(i) = \left(ANTin(i) \cap EARLin(i) \right) \cup \bigcap_{j \in pred(i)} DELAYout(j)$$

$$DELAYout(i) = \overline{ANTloc(i)} \cap DELAYin(i)$$

$$DELAYin(entry) = ANTin(entry) \cap EARLin(entry)$$

Block	ANTin(i) ∩ EARLin(i)
entry	{a+1}
B2	{x*y}
B3a	{x*y}

Block	DELAYin	DELAYout
entry	{a+1}	{a+1}
B1	{a+1}	{}
B2	{x*y}	{}
B2a	{}	{}
B3	{}	{}
B3a	{x*y}	{x*y}
B4	{}	{}
B5	{}	{}
B6	{}	{}

Lateness (LATE)

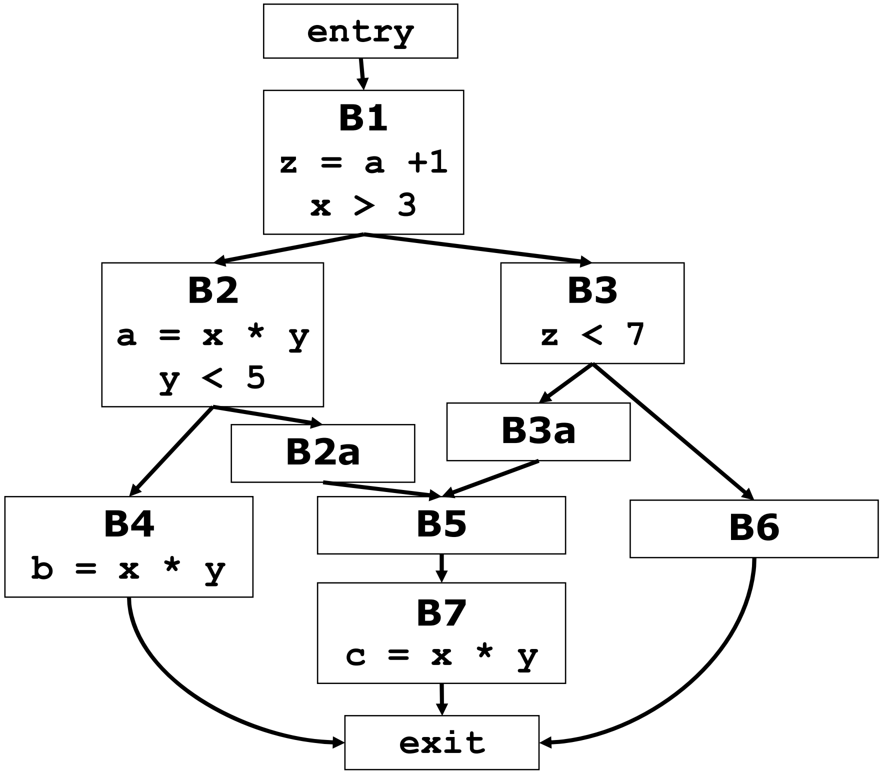
- An expression is *latest* on entry to a block if
  - it is the optimal point for computing the expression and
  - on every path from the block entry to exit, any other optimal computation point occurs after an expression computation in the original flowgraph

i.e., there is no "later" placement for this expression

Latestness (LATE)

$$LATEin(i) = DELAYin(i) \cap \left(ANTloc(i) \cup \overline{\bigcap_{j \in succ(i)} DELAYin(j)} \right)$$

Block	LATEin
entry	{}
B1	{a+1}
B2	{x*y}
B2a	{}
B3	{}
B3a	{x*y}
B4	{}
B5	{}
B6	{}
B7	{}
exit	{}



Isolatedness (ISOL)

- An optimal placement in a block for the computation of an expression is *isolated* iff
  - on every path from a successor of the block to the exit block, every original computation is preceded by the optimal placement point

Isolatedness (ISOL)

$$ISOLin(i) = LATEin(i) \cup \left(\overline{ANTloc(i)} \cap ISOLout(i)\right)$$

$$ISOLout(i) = \bigcap_{j \in succ(i)} ISOLin(j)$$

$$ISOLout(exit) = \{\}$$

Block	ISOLin	ISOLout
entry	{}	{}
B1	{a+1}	{}
B2	{x*y}	{}
B2a	{}	{}
B3	{}	{}
B3a	{x*y}	{}
B4	{}	{}
B5	{}	{}
B6	{}	{}
B7	{}	{}
exit	{}	{}

Optimal Placement

- The set of expression for which a given block is the optimal computation point is the set of expressions that are latest and not isolated

$$OPT(i) = LATEin(i) \cap \overline{ISOLout(i)}$$

Redundant Computations

- The set of redundant expressions in a block consist of those used in the block that are neither isolated nor latest

$$REDN(i) = ANTloc(i) \cap \overline{LATEin(i) \cup ISOLout(i)}$$

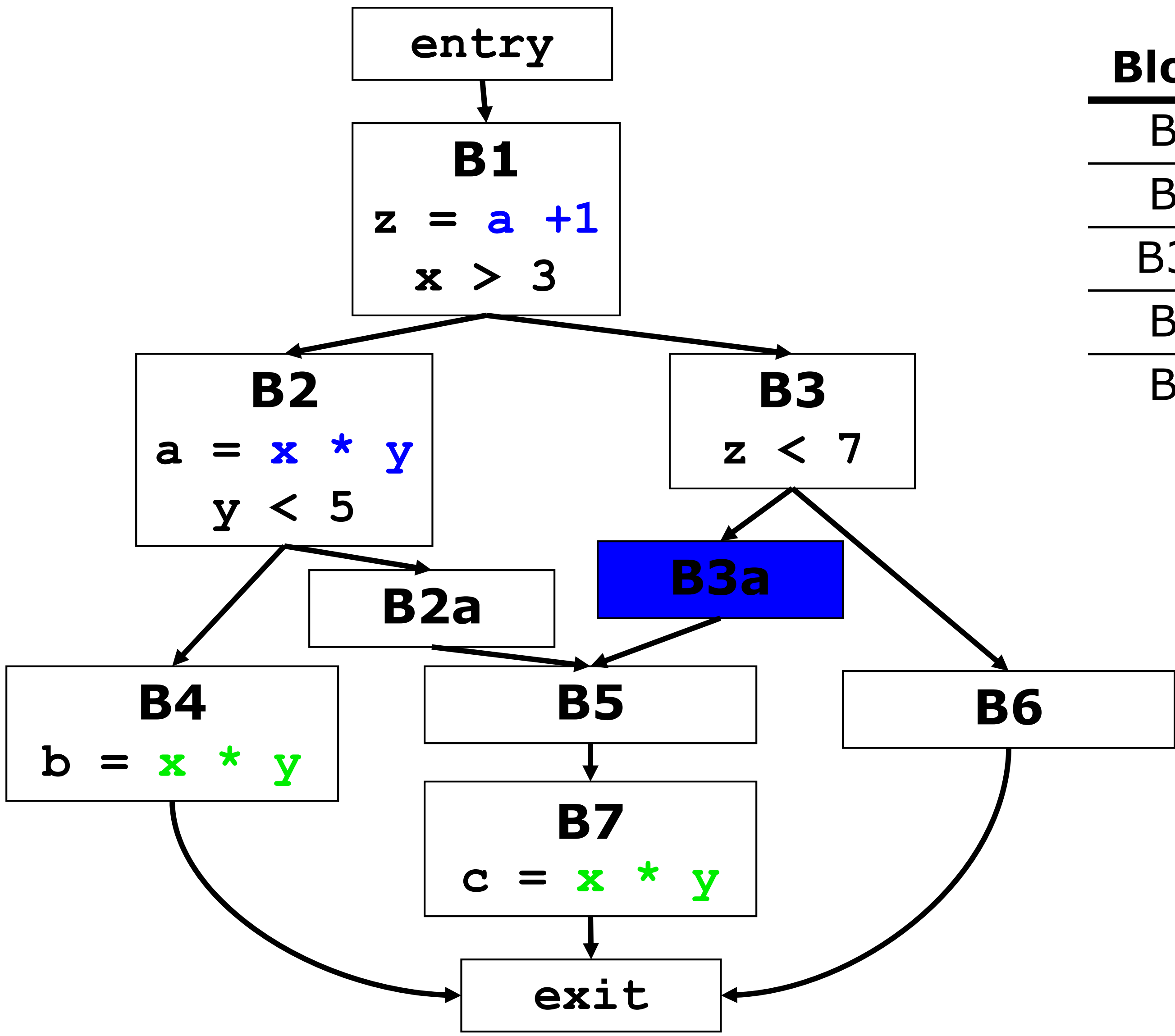
OPT and REDN

insert these (if necessary)

Block	OPT	REDN
entry	{}	{}
B1	{a+1}	{}
B2	{x*y}	{}
B2a	{}	{}
B3	{}	{}
B3a	{x*y}	{}
B4	{}	{x*y}
B5	{}	{}
B6	{}	{}
B7	{}	{x*y}
exit	{}	{}

remove these

PRE Example



Block	OPT	REDN
B1	{a+1}	{}
B2	{x*y}	{}
B3a	{x*y}	{}
B4	{}	{x*y}
B7	{}	{x*y}

PRE Example

4 data flow analyses later…