# Lazy Code Motion

**Bojian Zheng**

CSCD70 Spring 2018

bojian@cs.toronto.edu

# Partial Redundancy Elimination

- Can we replace calculations of $b + c$ such that no path re-executes the same expression?
- subsumes
  - Global Common Subexpression
    - Full Redundancy
  - Loop Invariant Code Motion
    - Partial Redundancy `for`-loops

# Common Subexpression Elimination

- On every path reaching $p$
  - Expression $b + c$ has been computed.
  - Neither $b$ nor $c$ is overwritten after the expression.

# Loop Invariant Code Motion

- 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?

# Lazy Code Motion

# Lazy Code Motion

- The optimization of **eliminating partial redundancy** with the goal of **delaying the computations** as much as possible.
- How are we going to achieve this?
  - **Anticipated** Expressions & **Will-be-Available** Expressions
  - **Postponable** Expressions
  - **Used** Expressions

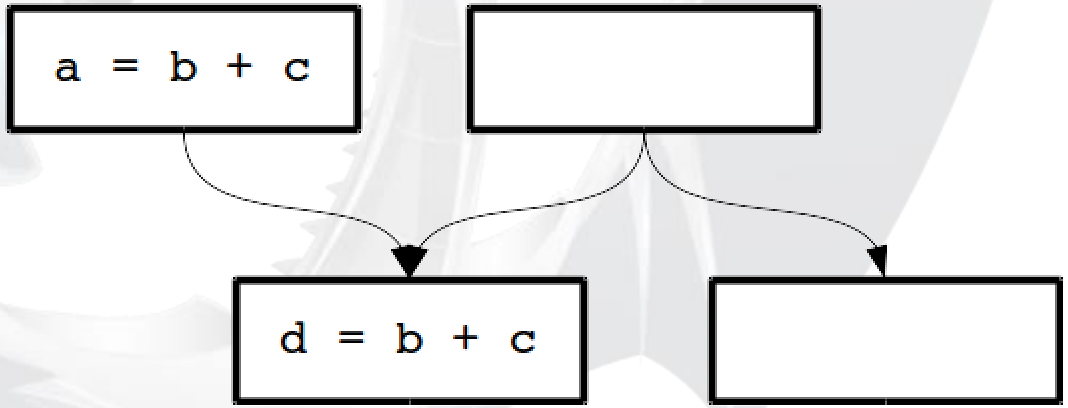
# Our Goal

- **Safety**
- **Maximum Redundancy Elimination**
- **Shortest Register Lifetime**

# Anticipated Expressions

# Safety

- We cannot introduce operations that are not executed originally.
- Given the diagram on the right, can we insert the expression $b + c$ on the right parent?



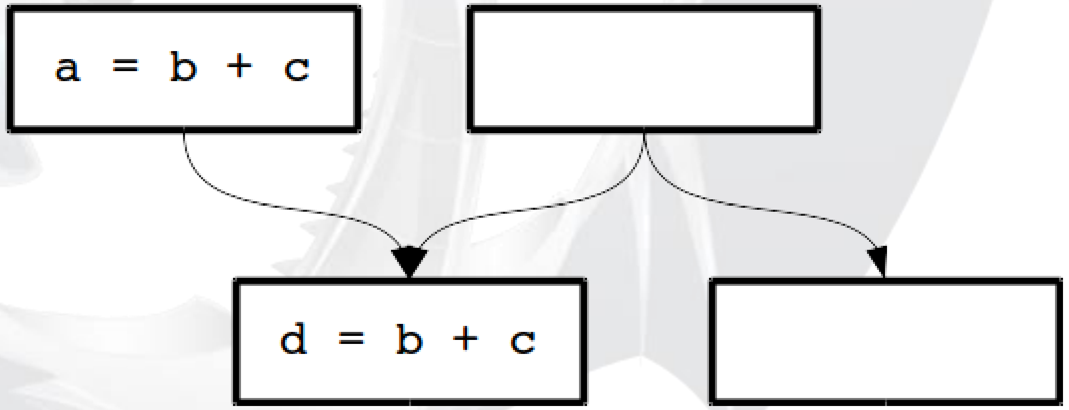
# Anticipated Expressions

- An expression *e* is said to be <u>**anticipated**</u> at program point *p* if <u>**all paths leading from *p* eventually computes *e***</u> (from the values of *e*'s operands that are available at *p*).

| | Anticipated Expressions |
|---|---|
| Domain | Sets of expressions |
| Direction | backward |
| Transfer Function | $f_b(x) = EUse_b \cup (x - EKill_b)$<br>EUse: exp used, EKill: exp killed |
| $\wedge$ | $\cap$ |
| Boundary | in[exit] = $\emptyset$ |
| Initialization | in[b] = {all expressions} |

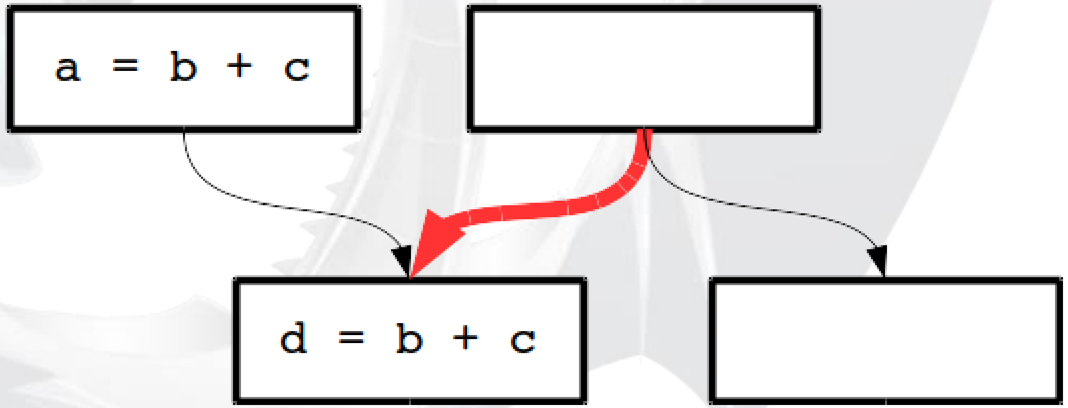
# Safety

- We cannot introduce operations that are not executed originally.
- Given the diagram on the right, can we insert the expression $b + c$ on the right parent?
- NO! The reason is because $b + c$ is not **anticipated** at the right parent.



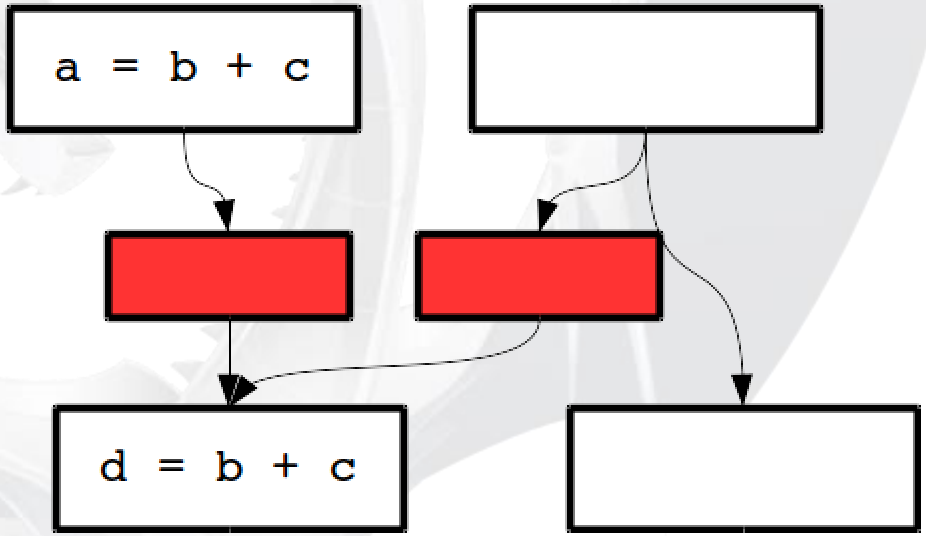
# Critical Edge

- If **the source has multiple successors**, and **the destination has multiple predecessors**, then the path that is connecting them is defined as **Critical Edge**.

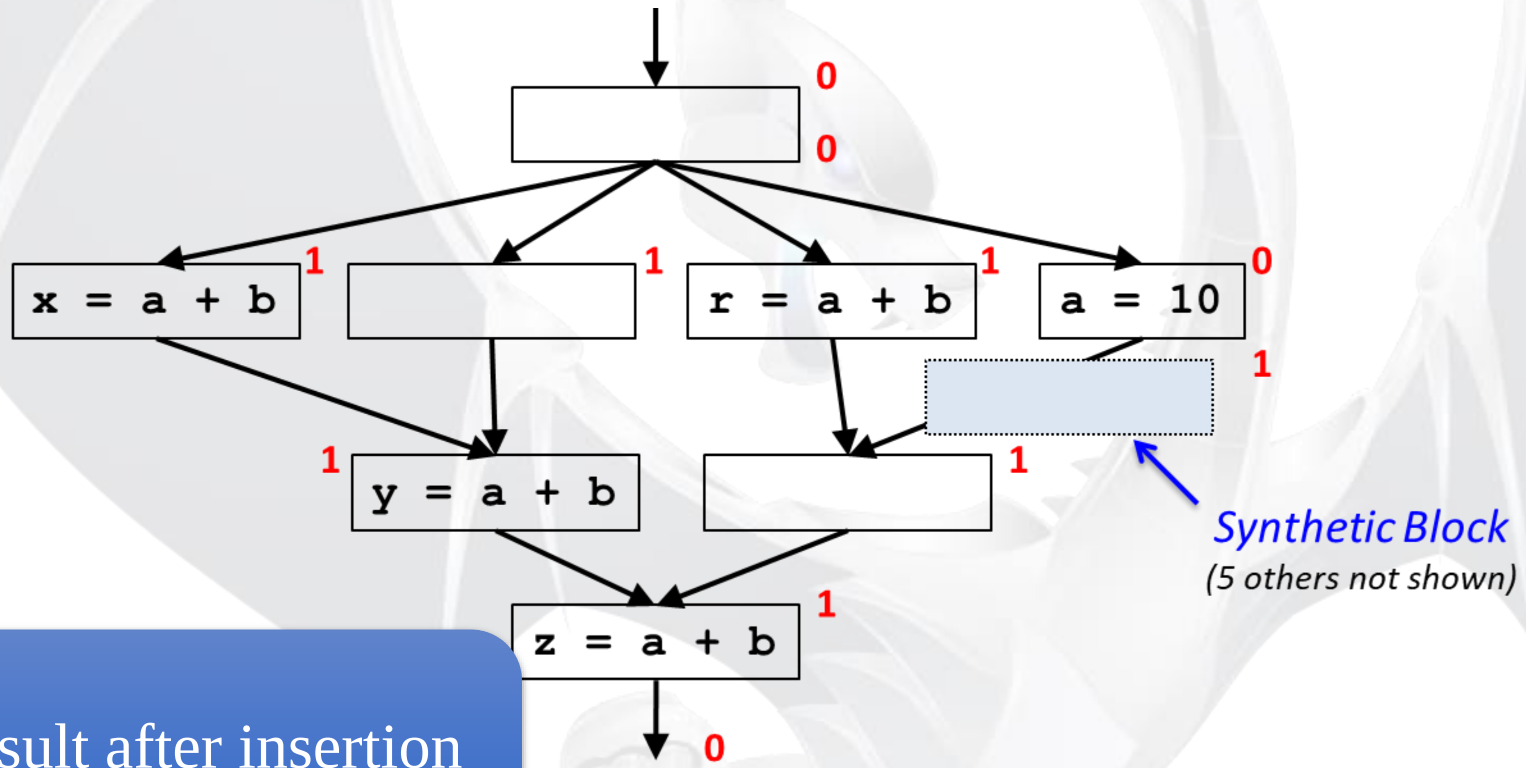


# Solution: Synthetic Block

- Add a basic block for every edge that leads to a basic block with multiple predecessors (not just the back edge).
- This simplifies the algorithm – since we can always place at the beginning of the basic block.

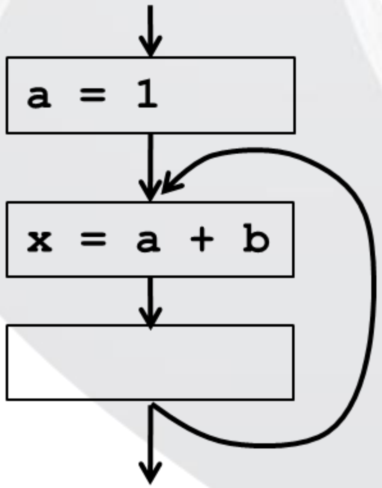


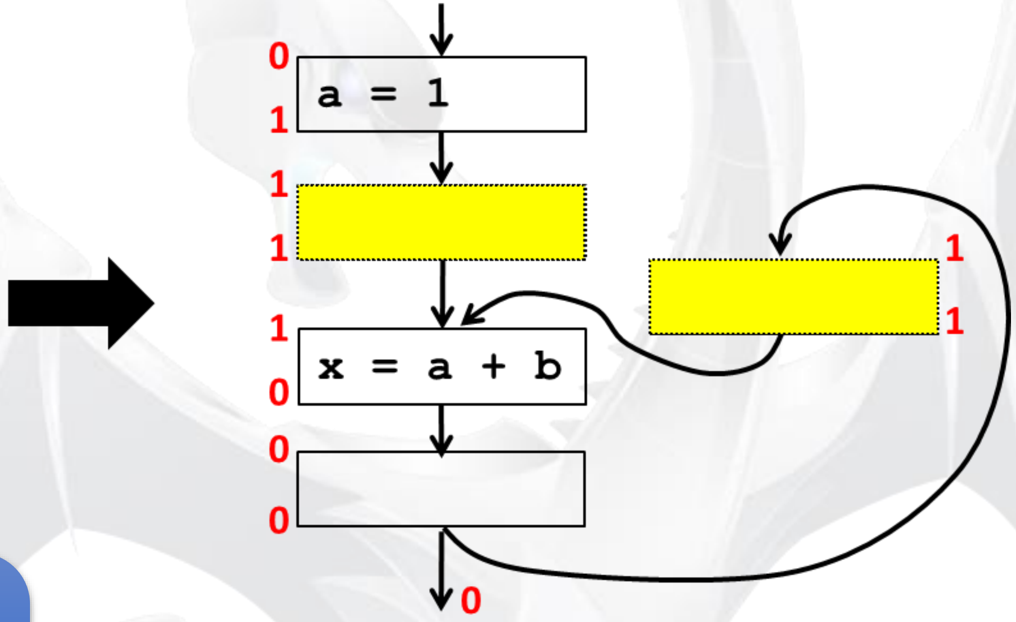
# Example 1




What is the result after insertion at the **anticipation frontier**?

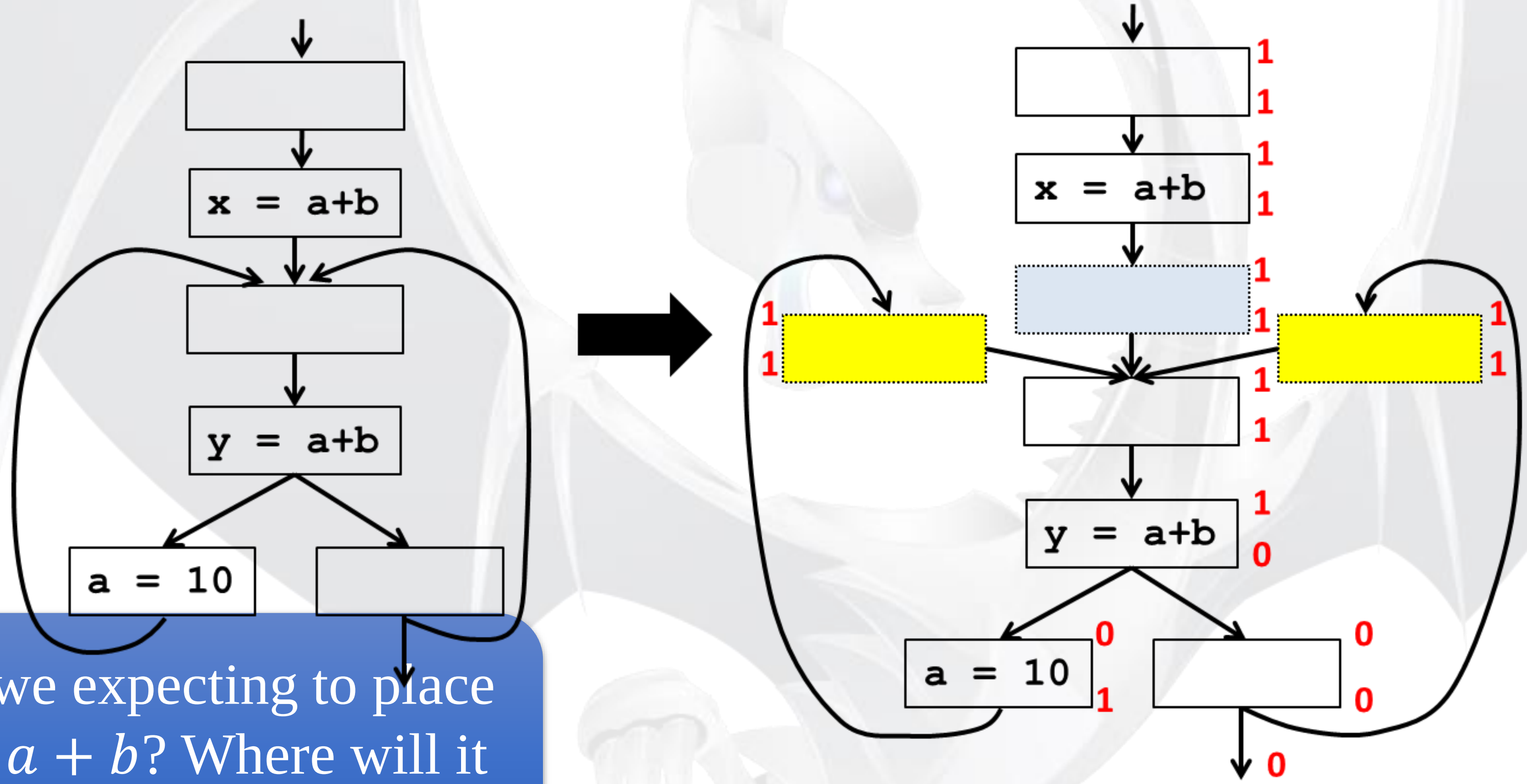
# Example 2: Loop Invariance



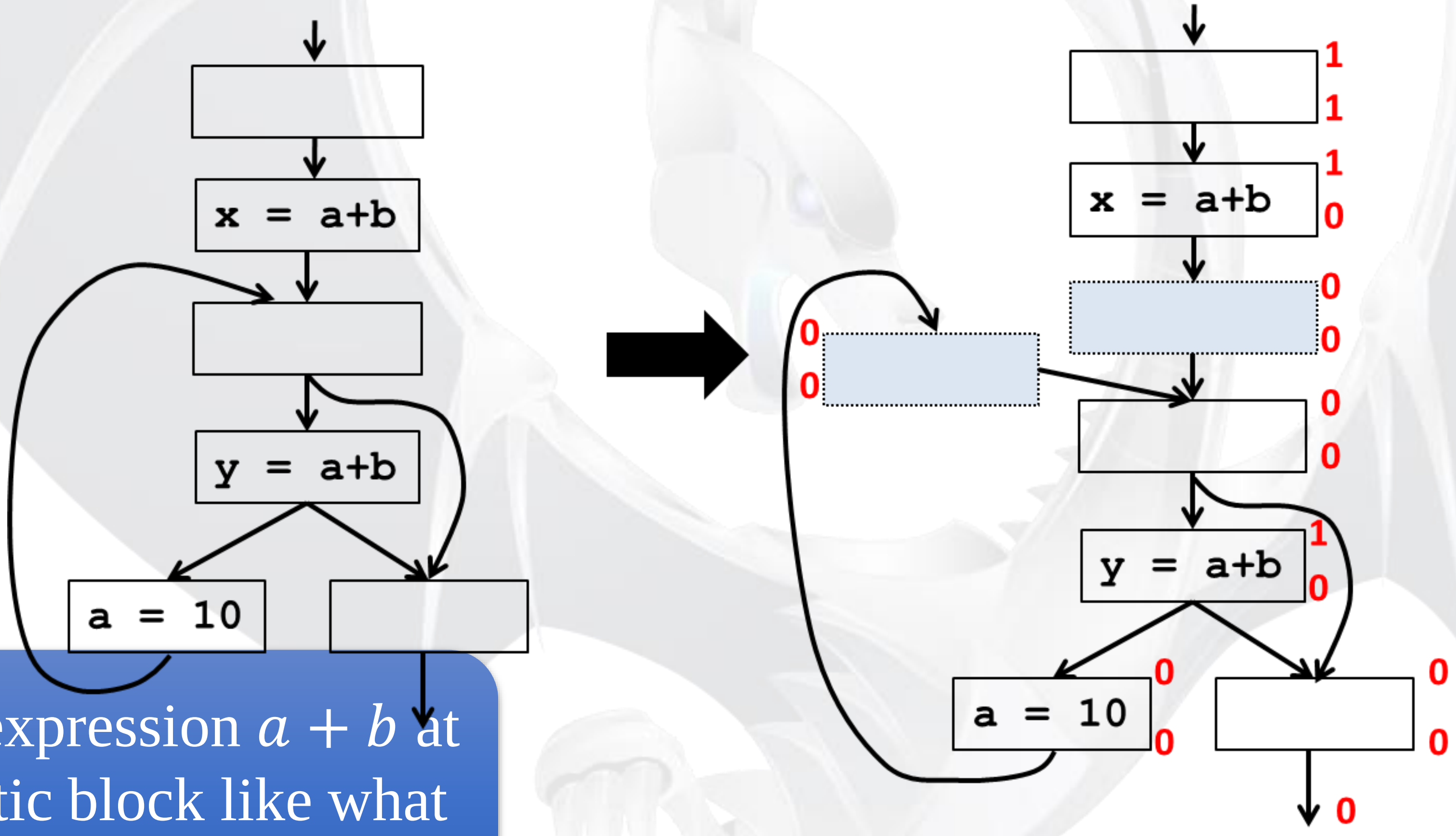

Will insertion at the **anticipation frontier** help in this case?

# Example 3: More Complex Loop




Where are we expecting to place expression $a + b$? Where will it actually be placed?

# Example 4: Complex Loop Variation




Can we place expression $a + b$ at the left synthetic block like what we did previously?

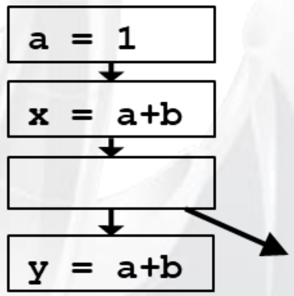
# Questions?

- Keywords:
  - Safety
  - Anticipated Expressions
  - Synthetic Block

# Will-be-Available Expressions

# Complications

- Does the **anticipation frontier** approach always work?
- The reason is because we have not yet considered expression **availability**.
- Want to make the expression *e* available **wherever it is anticipated but unavailable**.



# Will-be-Available Expressions

- An expression *e* is said to be **will-be-available** at program point *p* if **it is anticipated and not subsequently killed along all paths reaching *p***.
- Note how it is different from **Available Expressions**.

| | Available Expressions |
|---|---|
| Domain | Sets of expressions |
| Direction | forward |
| Transfer Function | $f_b(x) = (\text{Anticipated}[b].\text{in} \cup x) - \text{EKill}_b$ |
| $\wedge$ | $\cap$ |
| Boundary | out[entry] = ∅ |
| Initialization | out[b] = {all expressions} |

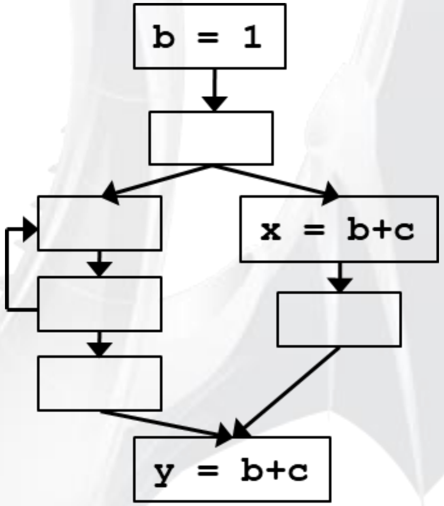
# Early Placement

- earliest$(b)$ is the set of expressions added to block $b$ under **early placement**, and is computed from the results of **anticipated** and **will-be-available**.

$$\text{earliest}(b) = \text{anticipated.in}(b)\,[\text{in}] - \text{will} \cdot \text{be} \cdot \text{available}(b)[\text{in}]$$

# Example

- Where is the **earliest** placement?
- Is it different from the **anticipation frontier**?



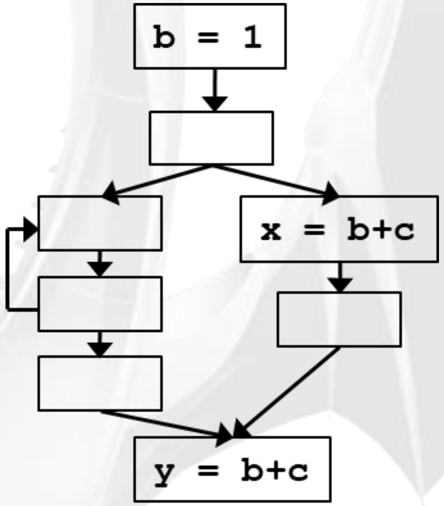
# Questions?

- Keywords:
  - Will-be-Available Expressions
  - Early Placement

# Postponable Expressions

# Shortest Register Lifetime?

- **<u>Early Placement</u>** goes against our goal of **<u>shortest register lifetime</u>**.
- We want to delay creating redundancy to reduce register pressure.

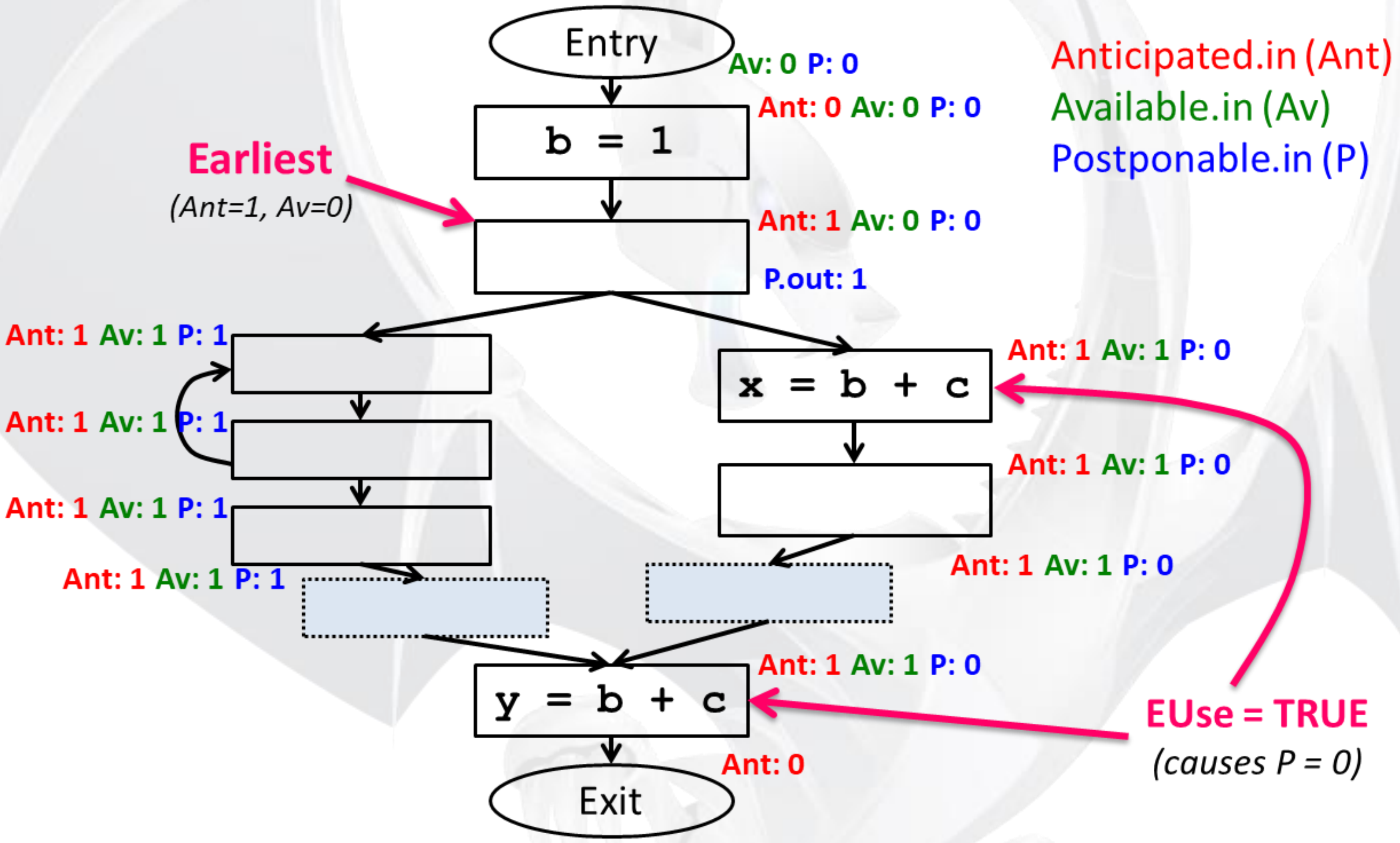


# Postponable Expressions

- An expression *e* is said to be **postponable** at program point *p* if **all paths leading to *p* have seen earliest placement of *e* but not a subsequent use**.

| | Postponable Expressions |
|---|---|
| Domain | Sets of expressions |
| Direction | forward |
| Transfer Function | $f_b(x) = (earliest[b] \cup x) - EUse_b$ |
| $\wedge$ | $\cap$ |
| Boundary | out[entry] = ∅ |
| Initialization | out[b] = {all expressions} |

# Example

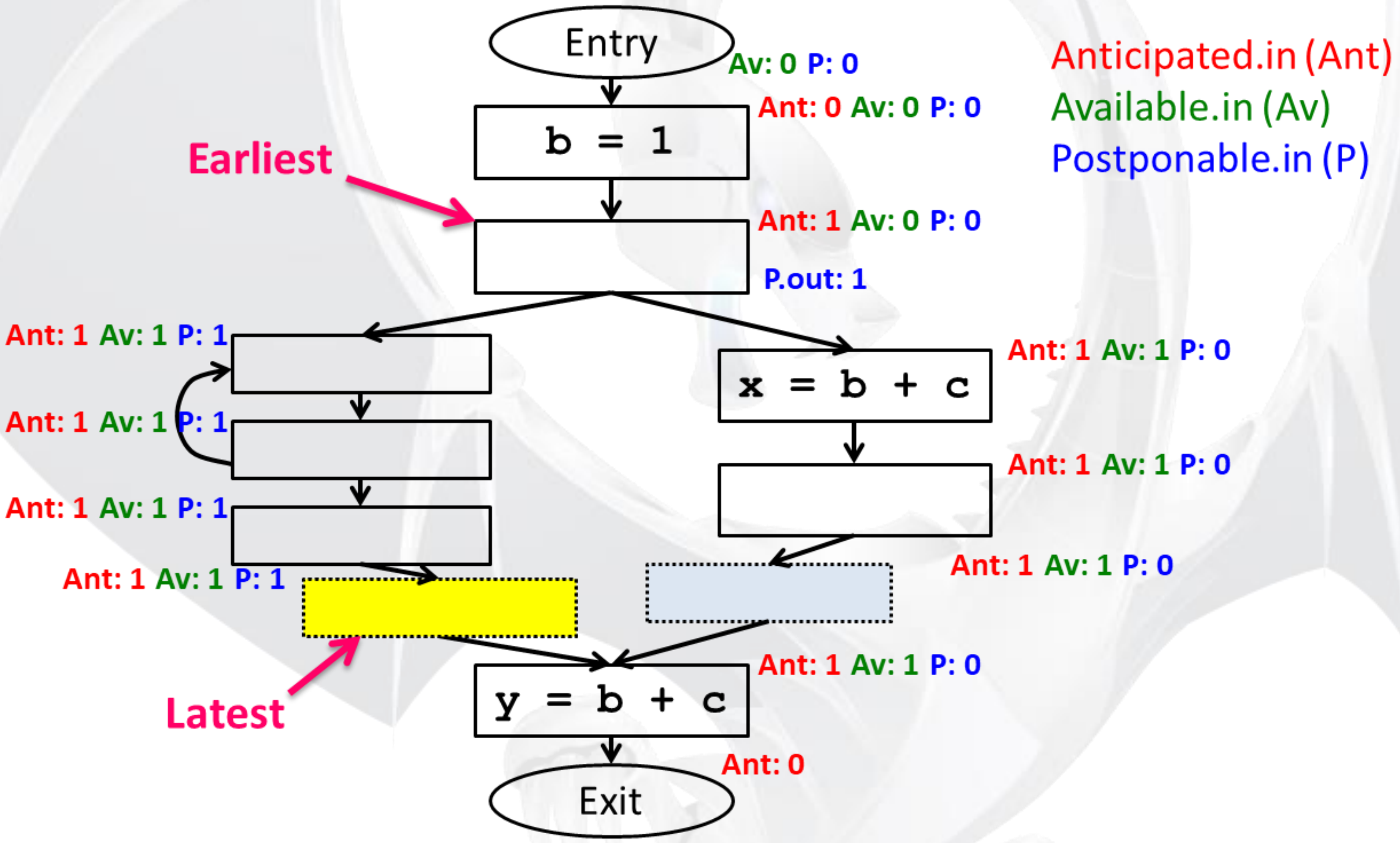

Entry
Av: 0 P: 0
b = 1
Ant: 0 Av: 0 P: 0
Earliest
(Ant=1, Av=0)
Ant: 1 Av: 0 P: 0
P.out: 1
Ant: 1 Av: 1 P: 1
Ant: 1 Av: 1 P: 1
Ant: 1 Av: 1 P: 1
Ant: 1 Av: 1 P: 1
x = b + c
Ant: 1 Av: 1 P: 0
Ant: 1 Av: 1 P: 0
Ant: 1 Av: 1 P: 0
y = b + c
Ant: 1 Av: 1 P: 0
Ant: 0
Exit
Anticipated.in (Ant)
Available.in (Av)
Postponable.in (P)
EUse = TRUE
(causes P = 0)

# Latest Placement

- We define the term **Latest** as follows:
  - It is ok to place the expression $e$: either **Earliest** ① or **Postponable** ②.
  - Need to place at $b$ if either:
    - $e$ is used in $b$ ③.
    - It is NOT ok to place in one of its successors ④.

$$\text{Latest}(b) = \left(\underbrace{\text{earliest}(b)}_{①} \cup \underbrace{\text{postponable}(b)}_{②}\right) \cap \left(\underbrace{\text{EUse}(b)}_{③} \cup \underbrace{\neg\left(\bigcap_{s\in\text{succ}(b)} (\text{postponable}(s))\right)}_{④}\right)$$

# Example


Entry
Av: 0 P: 0
b = 1
Ant: 0 Av: 0 P: 0
Earliest
Ant: 1 Av: 0 P: 0
P.out: 1
Ant: 1 Av: 1 P: 1
Ant: 1 Av: 1 P: 0
x = b + c
Ant: 1 Av: 1 P: 1
Ant: 1 Av: 1 P: 0
Ant: 1 Av: 1 P: 1
Ant: 1 Av: 1 P: 0
Ant: 1 Av: 1 P: 1
Ant: 1 Av: 1 P: 0
Latest
y = b + c
Ant: 0
Exit
Anticipated.in (Ant)
Available.in (Av)
Postponable.in (P)

# Questions?

- Keywords:
  - Postponable Expressions
  - Latest Placement

# Used Expressions

# Used Expressions

- An expression *e* is said to be **<u>used</u>** at program point *p* if **<u>there exists a path leading from *p* that uses the expression before the operands are reevaluated</u>**.

| | Used Expressions |
|---|---|
| Domain | Sets of expressions |
| Direction | backward |
| Transfer Function | $f_b(x) = (EUse[b] \cup x) - latest[b]$ |
| $\wedge$ | $\cup$ |
| Boundary | $in[exit] = \varnothing$ |
| Initialization | $in[b] = \varnothing$ |

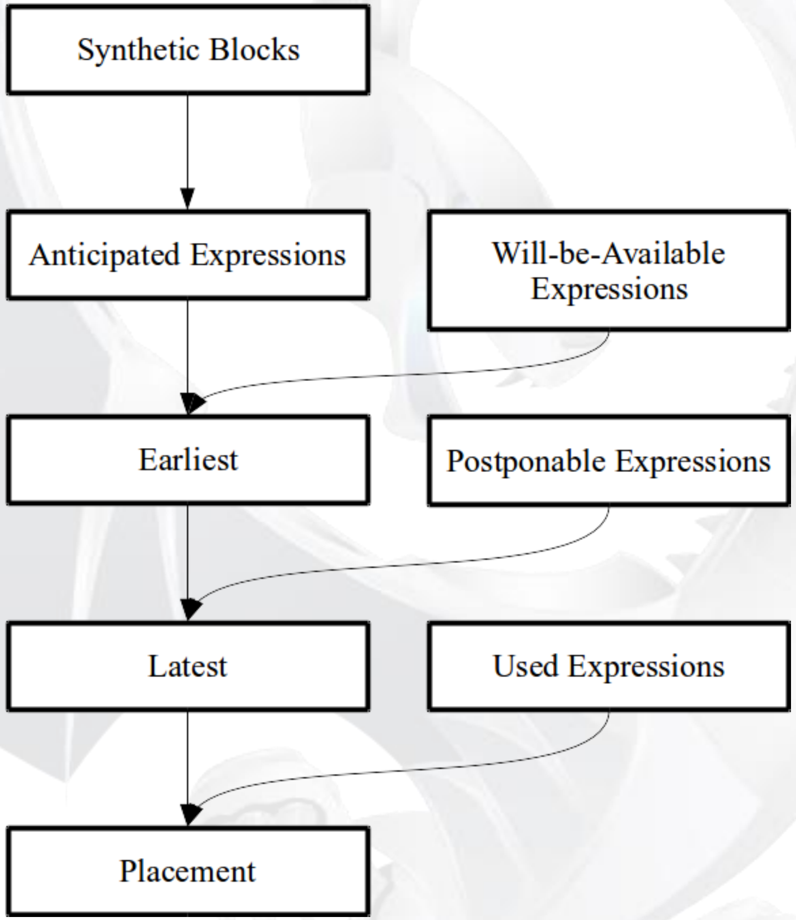
# Final Placement

- Our code transformation goes as follows:

  $\forall b$, if expression $e \in \left(\text{latest}(b) \cap (\text{used}(b))\right)$

  at the beginning of $b$, insert $t = e$, and replace every original $e$ with $t$

# Summary



Synthetic Blocks
Anticipated Expressions
Will-be-Available Expressions
Earliest
Postponable Expressions
Latest
Used Expressions
Placement