CSC2/455 Software Analysis and Improvement
Partial Redundancy Elimination

Sreepathi Pai

February 23, 2025

URCS

Outline

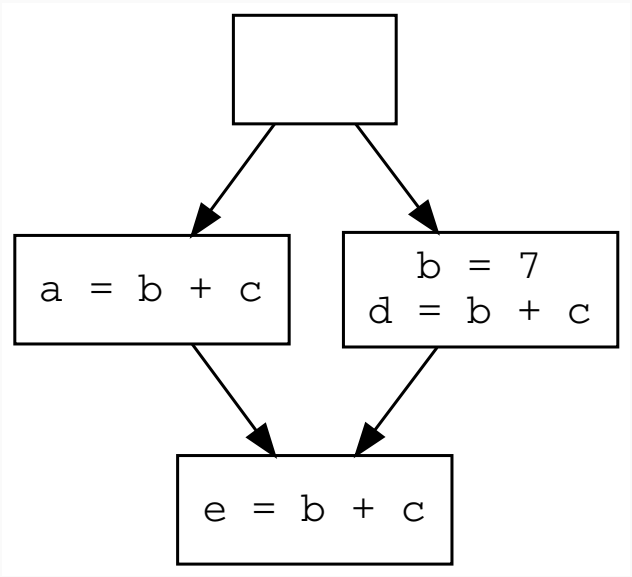
Outline

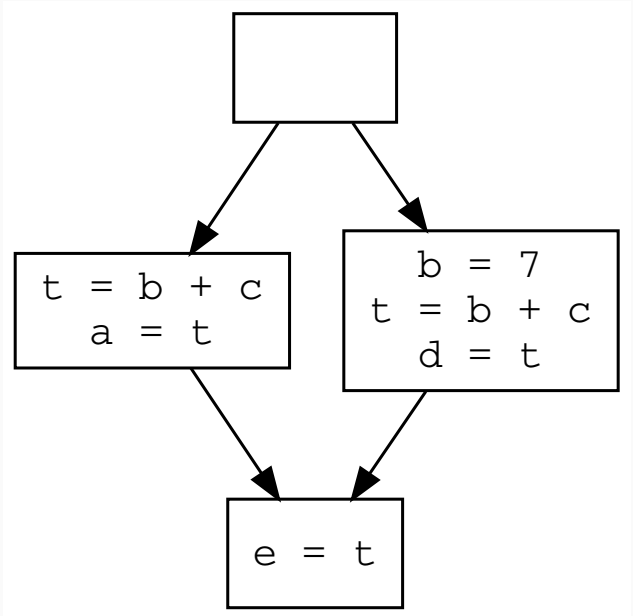
Optimizations: Dead Code Elimination

- Find useful operations (backward analysis)
- Find useful conditional branches
    - Reverse Dominance Frontier
- Remove code, and "touch up CFG"

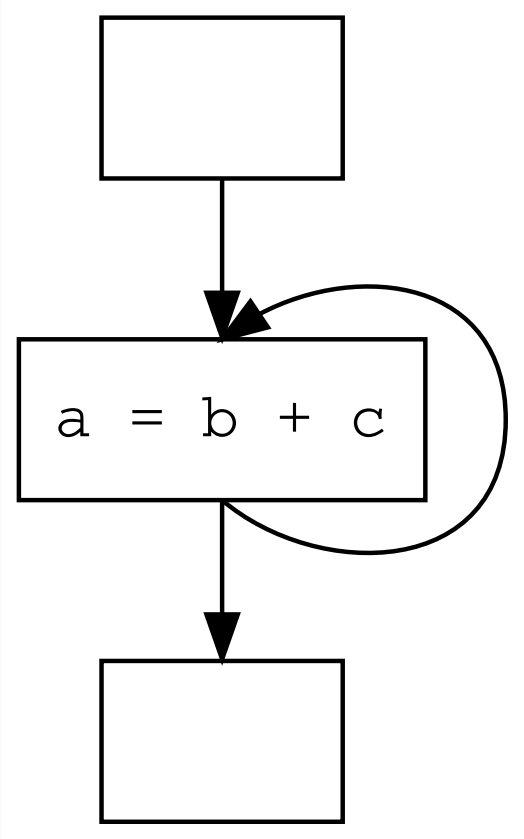
Outline

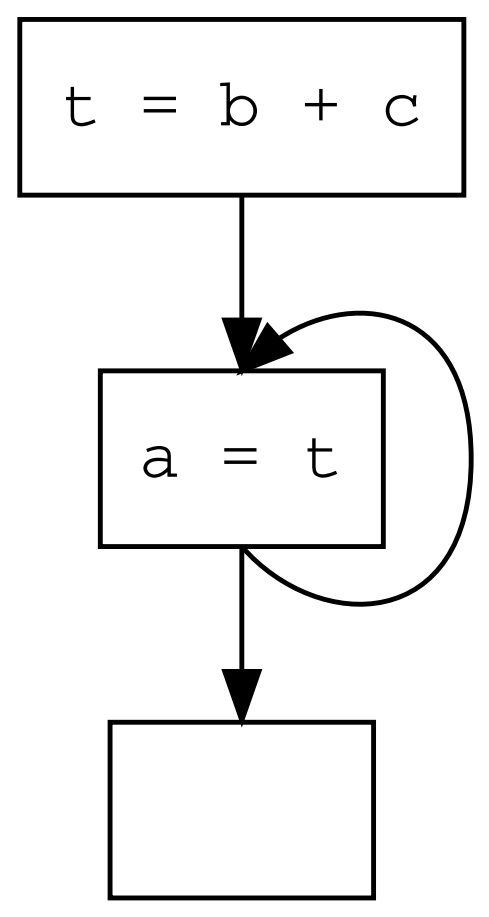
Redundancy: Fully Redundant



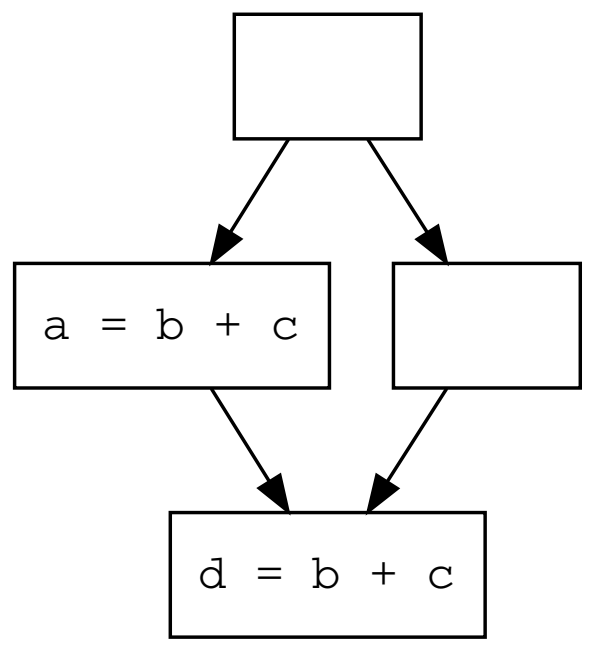


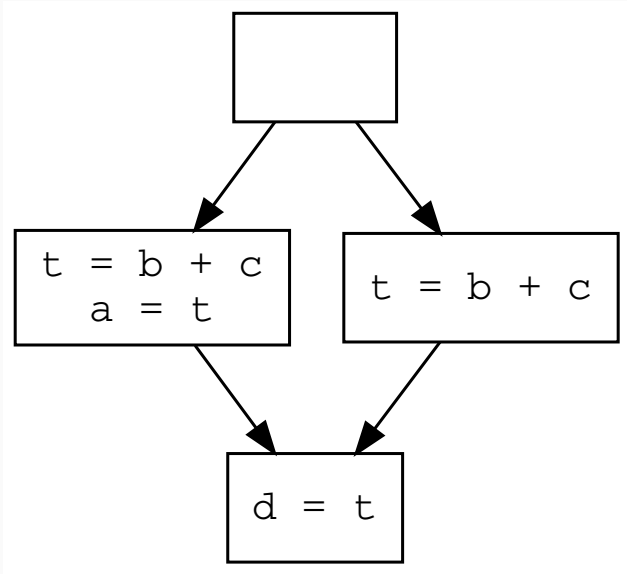
Redundancy: Loop Invariant



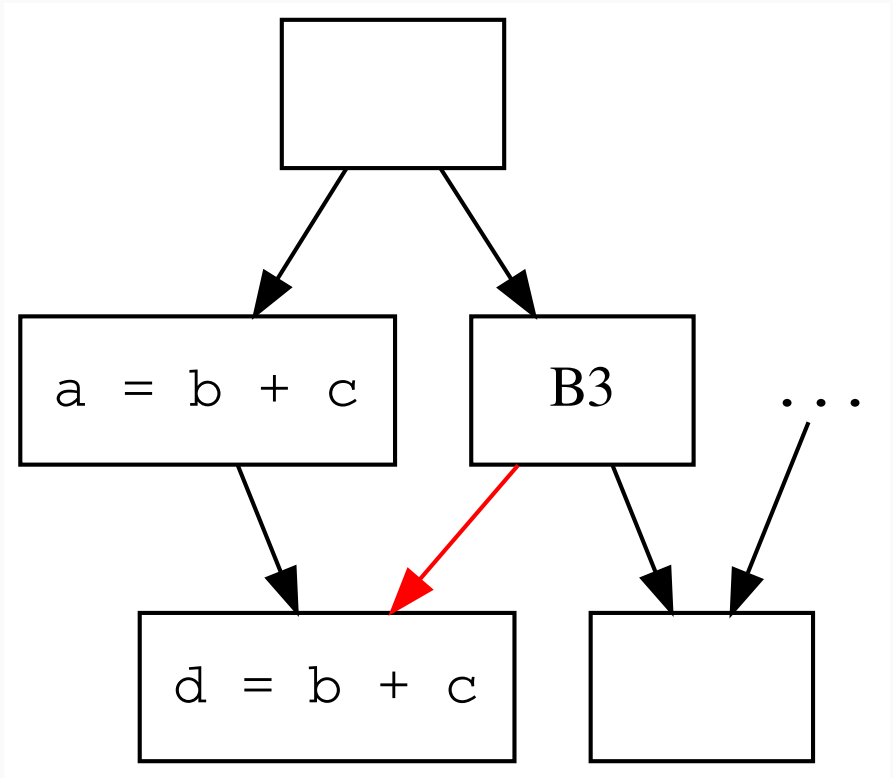


Redundancy: Partial Redundancy



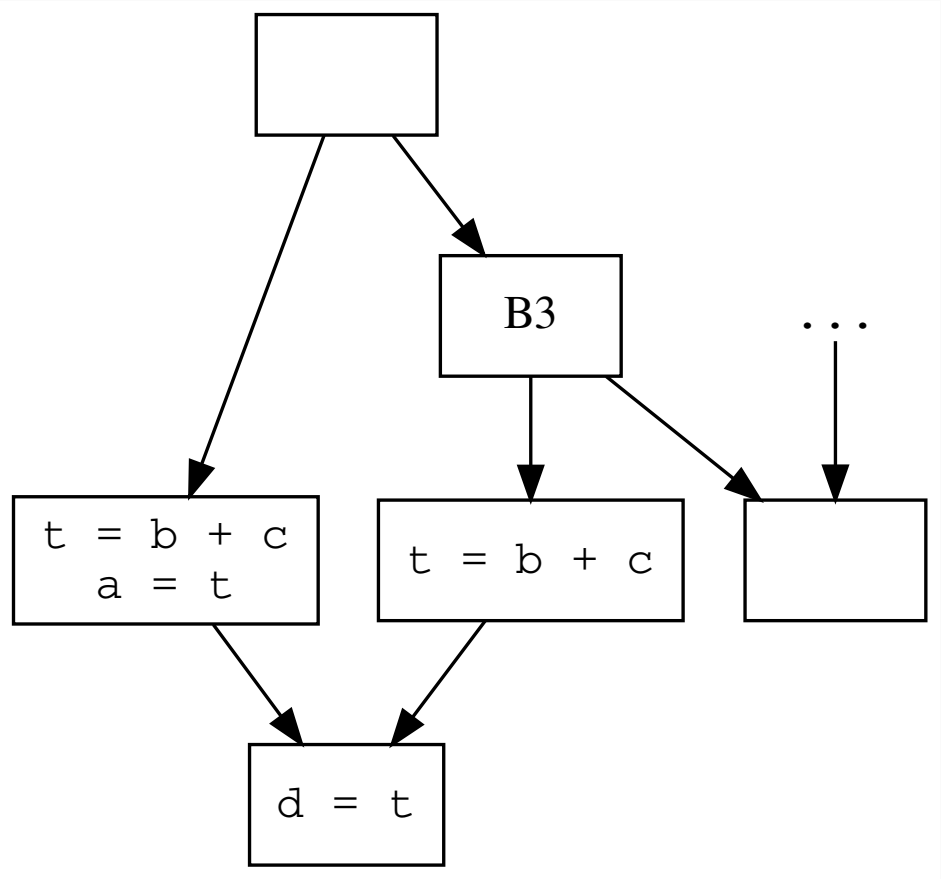


Eliminating Redundancy: Complication 1



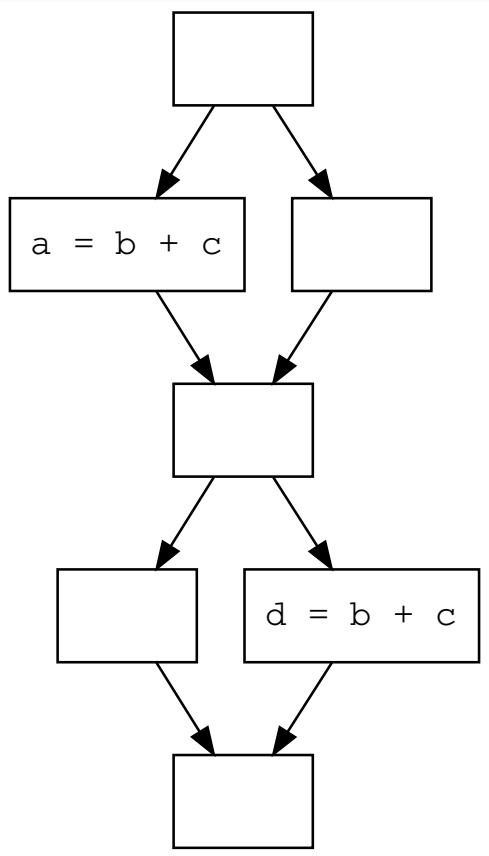
Can we insert `t = b + c` in B_3?

Splitting Critical Edges



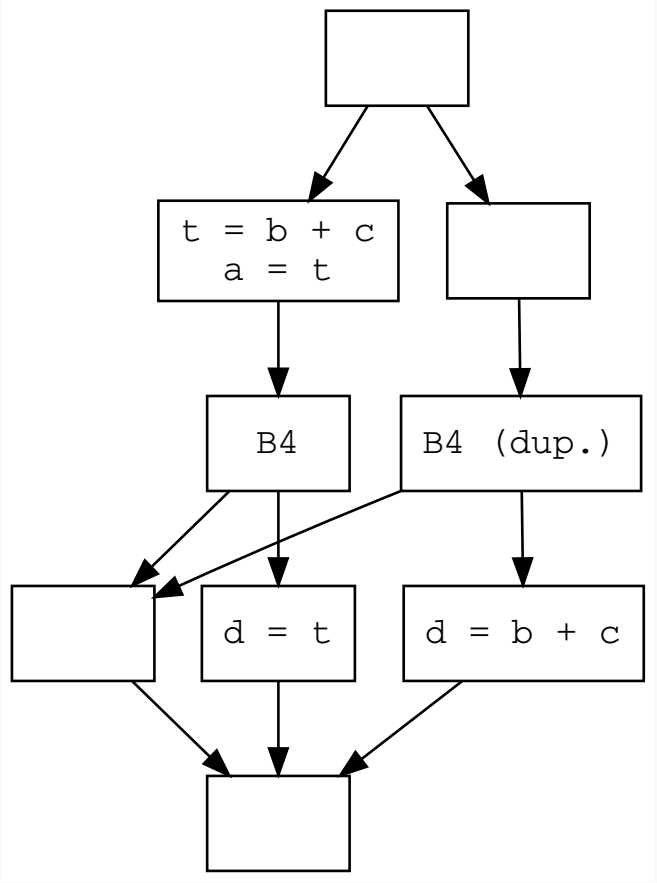
(Similar to when we were inserting minimal ϕ-functions.)

Eliminating Redundancy: Complication 2



Note that there is no block where `t = b + c` can be introduced without introducing computations not in the original program.

CFG duplication



(Possibility of exponential blowup.)

The Lazy Code Motion Algorithm

- Eliminate all expressions when it will not duplicate code
- Do not perform computations not in original program
  - Although where the computation is performed can change
- Delay computation for as long as possible
  - "Lazy"
  - Helps lower resource (esp. register) usage

Setup

For all blocks B in CFG, compute:

- e_use_B: set of expressions used in a block
- e_kill_B: set of expressions killed in block
    - usually by redefining subcomponents

Also, split all critical edges, inserting empty blocks.

Anticipable Expressions

Recall *very busy expressions*. An expression e is anticipable at block p if:

- ?

Anticipable Expressions

Recall *very busy expressions*. An expression e is anticipable at block p if:

- e is used/computed on all paths leading out of p
- And it is not killed before the use
- Implies that p can compute e and all paths could use this result

Anticipable Expressions Analysis

- Direction: Backwards
- Values: Expressions in programs
- Meet: $\cap$
- Transfer Function
  - $f_B(x) = e\_use_B \cup (x - e\_kill_B)$
- Equations:
  - $\mathrm{OUT}[B] = \wedge_{S \in succ(B)} \mathrm{IN}[S]$
  - $\mathrm{IN}[B] = f_B(\mathrm{OUT}[B])$
- $\top = U$
- $\mathrm{IN}[EXIT] = \emptyset$

Available Expressions

An expression is available at a program point p if:

- it has been computed along all paths leading into p
- it has not been killed since being computed until p
- (NEW) it is anticipated at p
    - we could make it available if it is anticipated

Available Expressions Analysis

- Direction: Forwards
- Meet: $\cap$
- Transfer function
  - $f_B(x) = (e\_use_B \cup anticipable[B].in) \cup (x - e\_kill_B)$
- Equations
  - $\mathrm{IN}[B] = \wedge_{P \in pred(B)} \mathrm{OUT}[P]$
  - $OUT[B] = f_B(\mathrm{IN}[B])$
- $\top = U$
- $\mathrm{OUT}[ENTRY] = \emptyset$

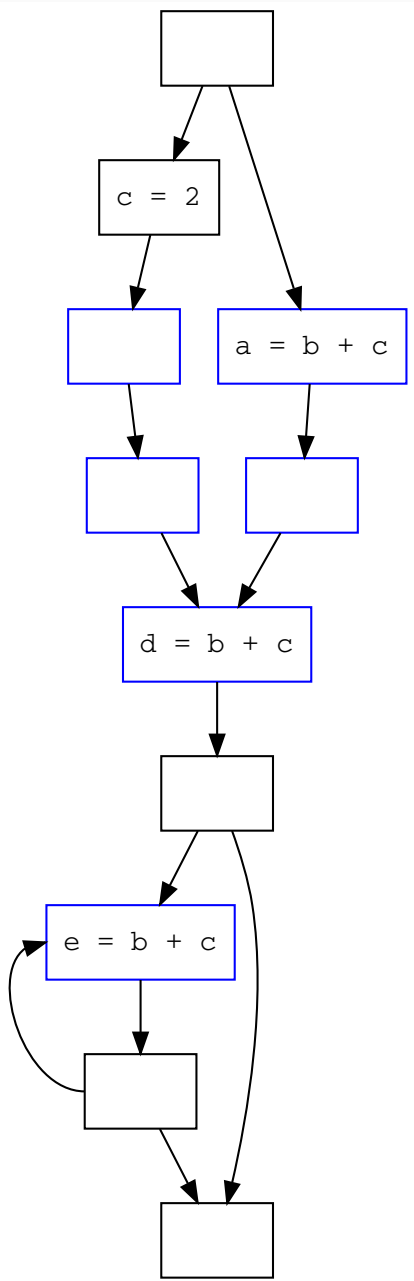
Positioning Expressions

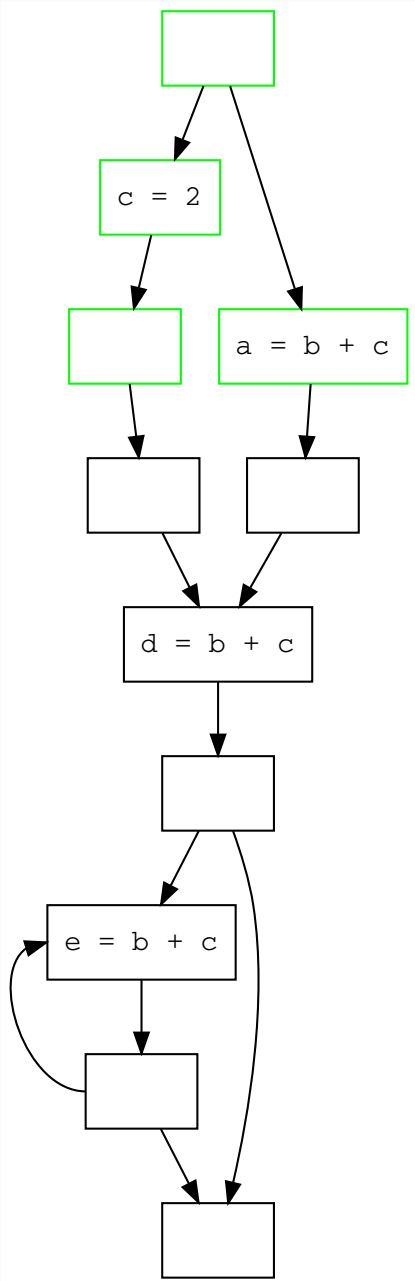
- When is the earliest an expression can be evaluated?
- When is the latest an expression can be evaluated?

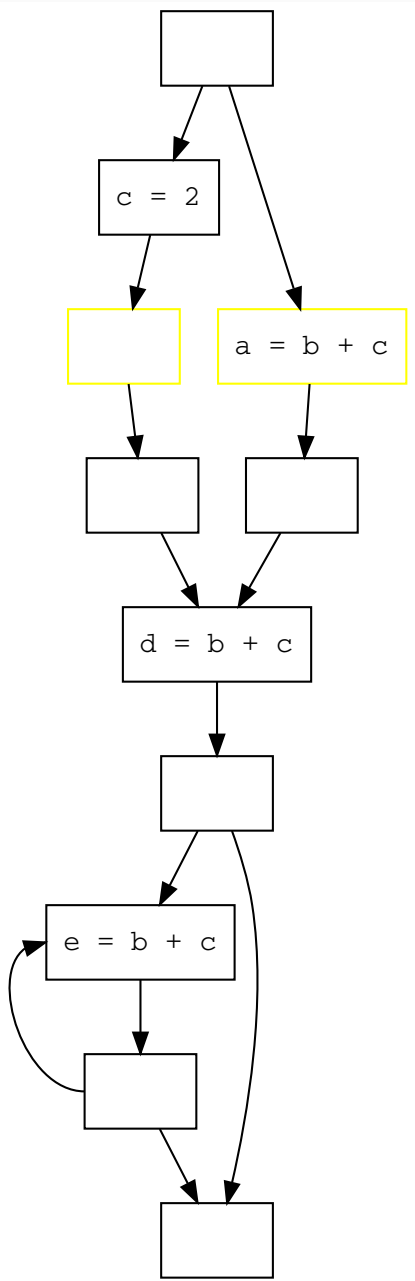
Positioning Expressions: Earliest

- When is the earliest an expression can be evaluated?
  - When it anticipated, but not available
- $earliest[B] = anticipable[B].in - available[B].in$
  - Observe notation for results of different analyses

Anticipable + (Not) Available = Earliest







Positioning Expressions: Latest

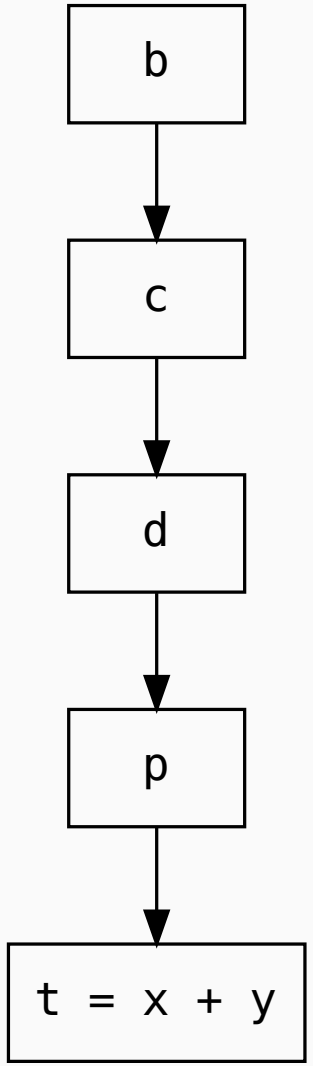
- When is the latest an expression can be evaluated?
  - When it can no longer be postponed
- "Postponed": expression pushed down from earliest placement

  - When can we push down an expression into the next block?

Postponable Expressions

An expression e is postponable to a block p if:

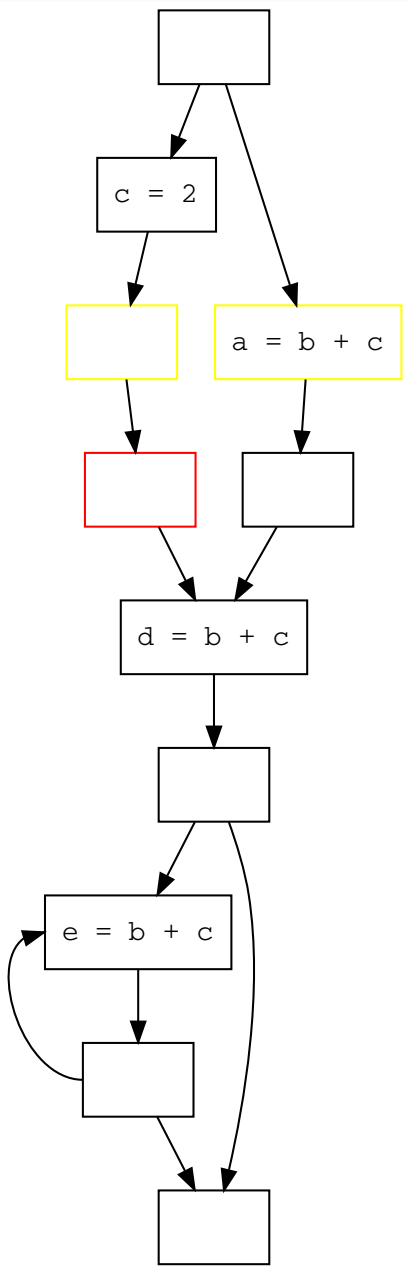
- e could be placed in block b before p (earliest is before p)
- Such that it is available on all paths leading to p from ENTRY
- And e is not used after block b (i.e., before p)



Postponable Expressions Analysis

- Direction: Forwards
- Values: Expressions
- Meet: $\cap$
- Transfer functions
  - $f_B(x) = (earliest[B] \cup x) - e\_use_B$
- Equations
  - $\mathrm{OUT}[B] = f_B(\mathrm{IN}[B])$
  - $\mathrm{IN}[B] = \wedge_{P \in pred(B)} OUT[P]$
- $\top = U$
- $\mathrm{OUT}[ENTRY] = \emptyset$

Postponable



Postponement Frontier

A block p is on the *postponement frontier* for an expression e if

- e can be postponed to p
- e cannot be placed at entry to a successor s of p
  - e is used in p
  - e is not postponable from some predecessor of s
  - e is not in $earliest[S]$

$$latest[B] = (earliest[B] \cup postponable[B].in) \cap (e\_use_B \cup (\cap_{S \in succ(B)}(earliest[S] \cup postponable[S].in))^{\complement})$$

(Note: $A^{\complement}$ means the complement of set A)

Used Expressions

An expression e in block p is used if:

- Some block q uses e
- There exists a path from p to q that does not invalidate e
    - I.e. recompute e or invalidate its operands

Used Expressions Analysis

- Direction: Backwards
- Values: Expressions
- Meet: $\cup$
- Transfer function
  - $f_B(x) = (x \cup e\_use_B) - latest[B]$
- Equations
  - $\mathrm{IN}[B] = f_B(\mathrm{OUT}[B])$
  - $\mathrm{OUT}[B] = \wedge_{S \in succ(B)} \mathrm{IN}[s]$
- $\top = \emptyset$
- $\mathrm{IN}[EXIT] = \emptyset$

Putting it all together - I

- Compute $anticipable[B].in$, $available[B].in$
- Compute $earliest[B]$
- Compute $postponable[B].in$
- Compute $latest[B]$
- Compute $used[B].out$

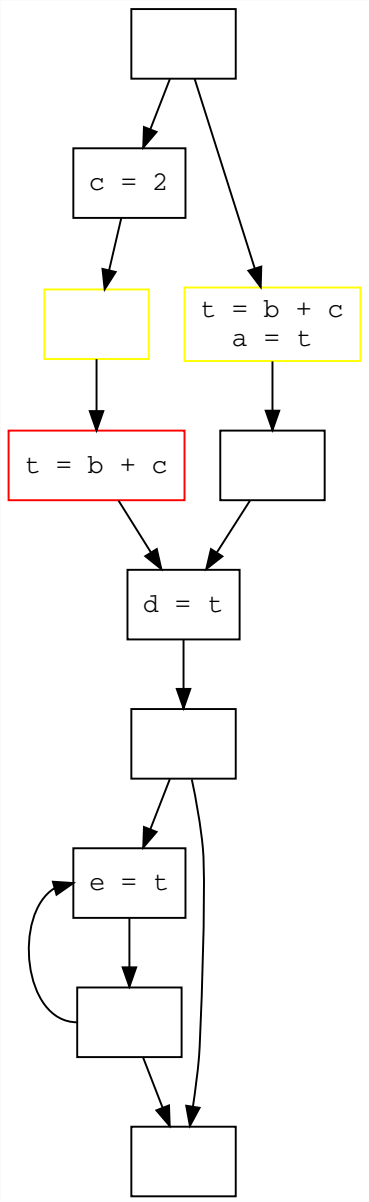
Putting it all together - II

For each expression `x + y` in program:

- Create `t = x + y` (where `t` is a unique temporary)
- Place `t = x + y` at the beginning of all blocks B such that
  - `x + y` is in $\mathit{latest}[B] \cap \mathit{used}[B].\mathit{out}$
  - i.e. B is the last block where `x + y` can be placed, and `x + y` is used after B
- Replace all `x + y` with `t` in all block B where:
  - `x + y` $\in (\mathit{e\_use}_B \cap (\mathit{latest}[B]^{\complement} \cup \mathit{used}[B].\mathit{out})$
  - I.e., `x + y` is in $\mathit{e\_use}_B$, and
  - `x + y` is NOT in $\mathit{latest}[B]$, or
  - `x + y` is in `used[B].out`

Algorithm 9.36 in the Dragon Book.

Final result



Outline

References

- Chapter 9 of the Dragon Book
  - Section 9.5