CS 4240: Compilers

Lecture 5: Lazy Code Motion, Available Expression Analysis

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

January 23, 2019

Course Announcements

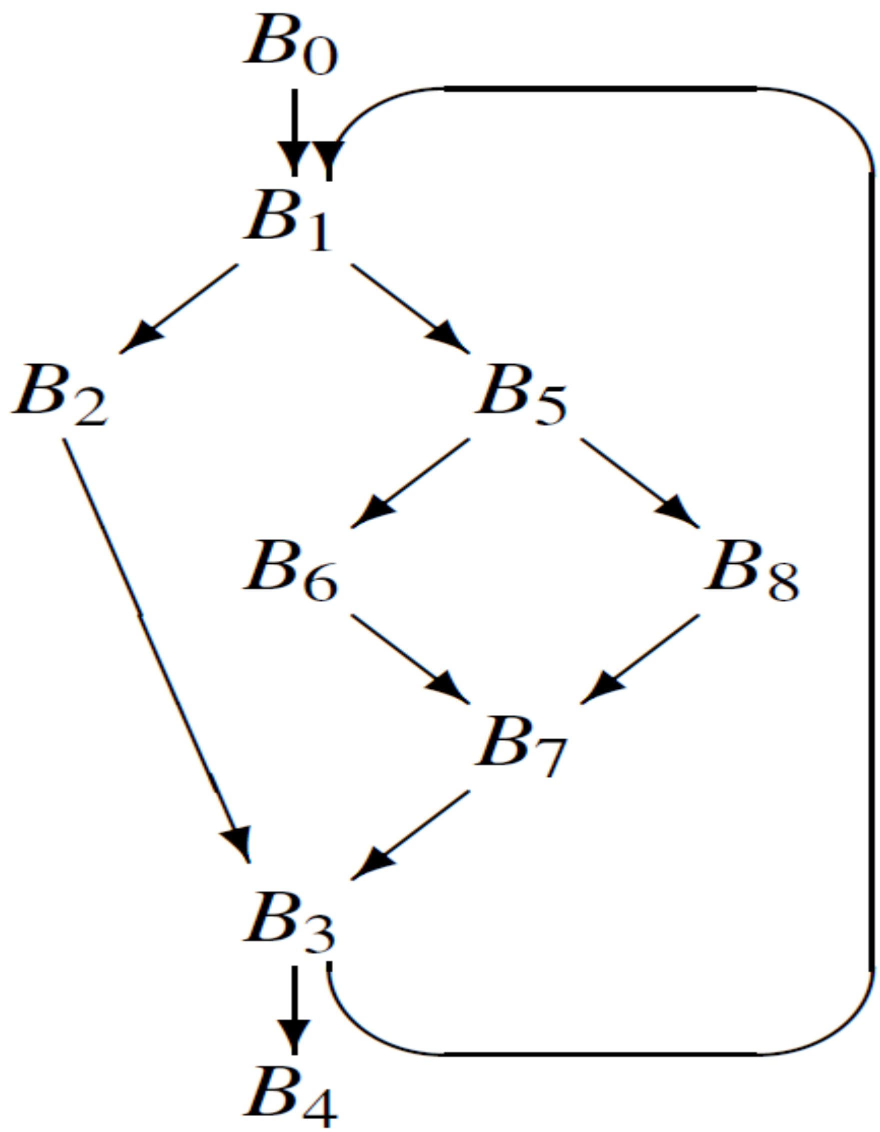
- Homework 1 was released on Monday (1/14/19) on Piazza
  - Due by 11:59pm on Wednesday, 1/30/19 on Canvas
  - Must be submitted as PDF file
  - 5% of course grade

- Project 1 was released on Wednesday (1/16/19) on Piazza
  - Due by 11:59pm on Wednesday, 2/13/19 on Canvas
  - Must be submitted as zip file including instructions on how to build and run your project
  - 5% of course grade
  - Project teams have been announced on Piazza — please inform us ASAP of any inaccuracies

Worksheet-4 Solution

(From Lecture 4 given on 01/16/2019)

- Q1
  Compute **dominator sets** for each node in the CFG.



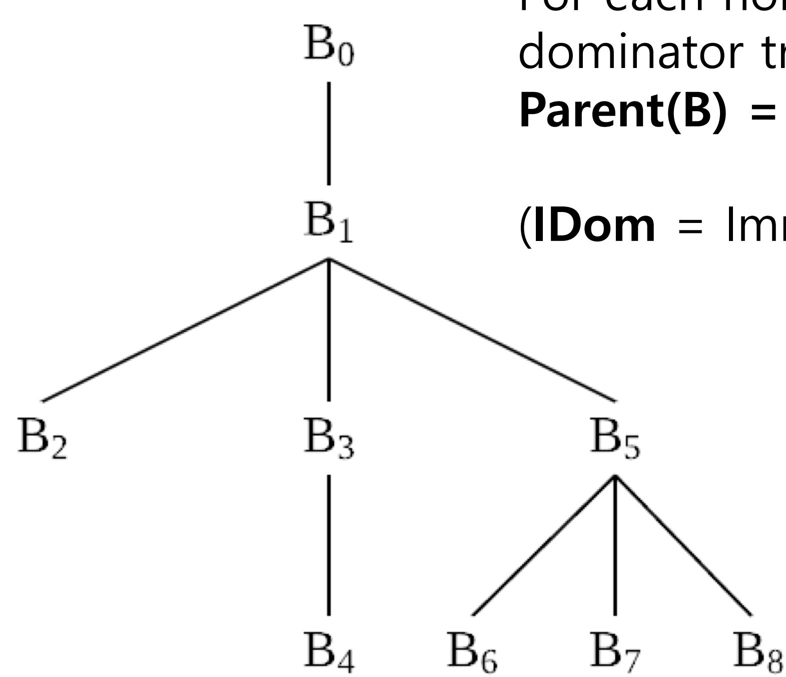
Dom(n)	Set of basic blocks that dominate B_n
Dom(B_0)	{B_0}
Dom(B_1)	{B_0, B_1}
Dom(B_2)	{B_0, B_1, B_2}
Dom(B_3)	{B_0, B_1, B_3}
Dom(B_4)	{B_0, B_1, B_3, B_4}
Dom(B_5)	{B_0, B_1, B_5}
Dom(B_6)	{B_0, B_1, B_5, B_6}
Dom(B_7)	{B_0, B_1, B_5, B_7}
Dom(B_8)	{B_0, B_1, B_5, B_8}

Q2: Draw the dominator tree of the CFG

Observations:

- The immediate dominator idom(n) of a node n is the unique last strict (different from n) dominator on any path from ENTRY to n
- The dominator tree is an efficient encoding of dominator sets; the dominator set of node n can be obtained by enumerating all nodes from n to the root of the tree.

Dom(n)	Set of basic blocks that dominate B_n
Dom(B_0)	{B_0}
Dom(B_1)	{B_0, B_1}
Dom(B_2)	{B_0, B_1, B_2}
Dom(B_3)	{B_0, B_1, B_3}
Dom(B_4)	{B_0, B_1, B_3, B_4}
Dom(B_5)	{B_0, B_1, B_5}
Dom(B_6)	{B_0, B_1, B_5, B_6}
Dom(B_7)	{B_0, B_1, B_5, B_7}
Dom(B_8)	{B_0, B_1, B_5, B_8}



For each non-entry node B in the dominator tree,
Parent(B) = IDom(B)

(**IDom** = Immediate Dominator)

Comments on students' answers

- Most of the students got the answers right.
- Common mistakes
  1. Some students didn't add **B** to **Dom(B)**.
     (for each basic block B in the CFG)
  2. Some students computed dominator sets correctly, but failed to draw the correct dominator tree.

Computing Dominators

- Approach 1:
  - Formulate problem as a system of data flow constraints:
    - Define dom(n) = set of nodes that dominate n
    - $dom(n_0) = \{n_0\}$
    - $dom(n) = \cap \{ dom(m) \mid m \in pred(n) \} \cup \{n\}$
    - i.e, the dominators of n include the dominators of all of n's predecessors and n itself
  - Can be solved using iterative algorithm by initializing all dom sets except $dom(n_0)$ to the universal set (set of all CFG nodes)

- Approach 2:
  - More efficient graph algorithm based on depth-first search
  - See Lengauer, Thomas; and Tarjan; Robert Endre (July 1979). "A fast algorithm for finding dominators in a flowgraph". ACM Transactions on Programming Languages and Systems. 1 (1): 121–141. https://doi.org/10.1145%2F357062.357071

Computing Dominators (Approach 1)

- A node n dominates m iff n is on every path from n_0 to m
  - Every node dominates itself
  - n's <u>immediate dominator</u> is its closest dominator, $\text{I}\textsc{Dom}(n)$

— Initialize $\textsc{Dom}(n_0) = \{ n_0 \}$, where n_0 is the ENTRY node, and DOM(n) = N, set of all CFFG vertices,

— Iterate on the following recursive equations for all nodes n, until a fixpoint is reached.

$$\textsc{Dom}(n) = \{ n \} \cup \left(\bigcap_{p \in preds(n)} \textsc{Dom}(p)\right)$$

Computing DOM

- These simultaneous set equations define a simple problem in data-flow analysis
- Equations have a unique fixed point solution
- An iterative fixed-point algorithm will solve them quickly

Redundancy Elimination (Recap)

An expression x+y is redundant if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

If the compiler can prove that an expression is redundant

- It can preserve the results of earlier evaluations
- It can replace the current evaluation with a reference

Two pieces to the problem

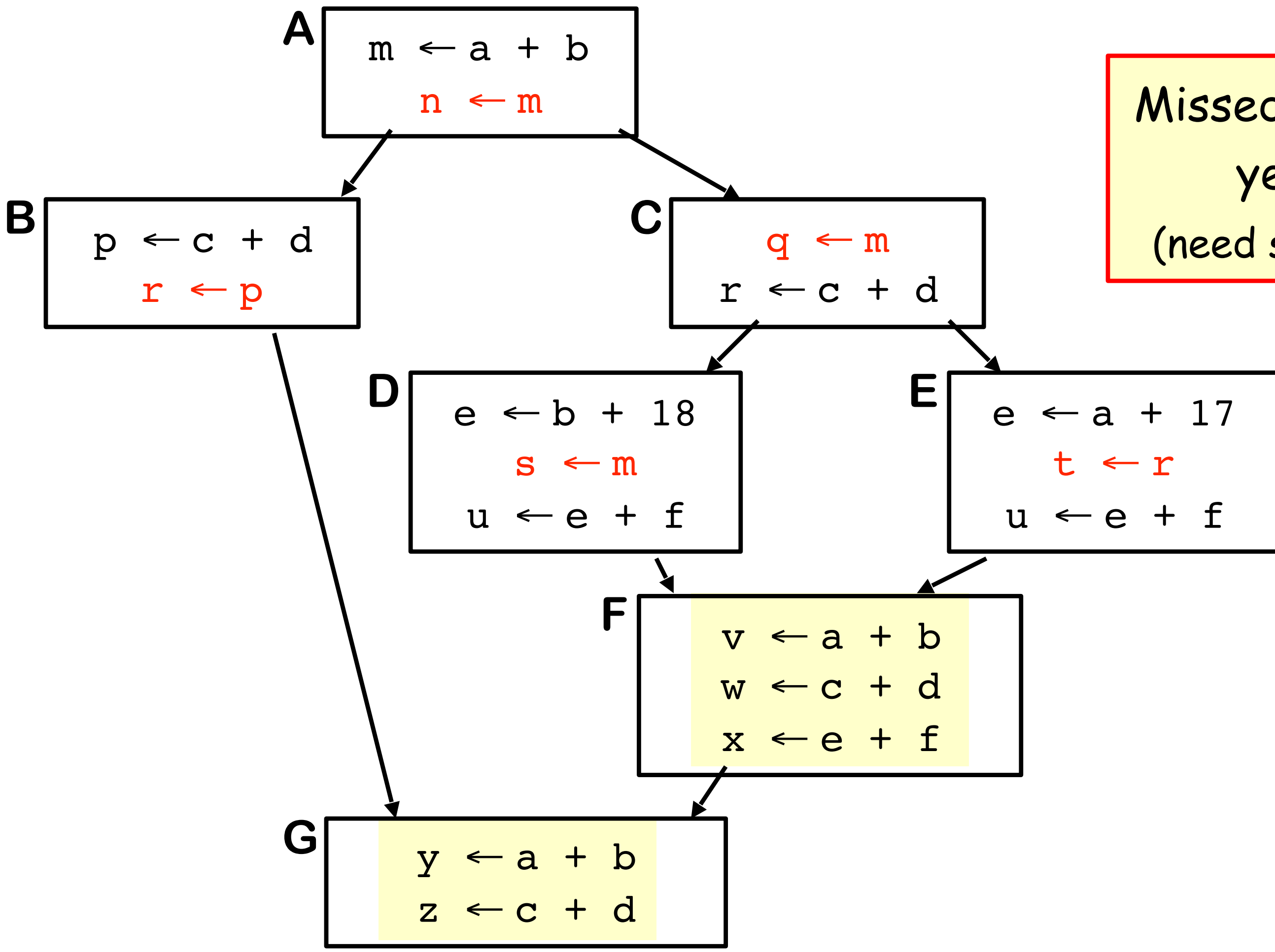
- Proving that x+y is redundant
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called value numbering

We learned how to perform value numbering within a basic block (local) and an extended basic block (superlocal)

Limitations of Superlocal Value Numbering (Recap)

SVN finds redundant ops in red



Missed opportunities in yellow remain!
(need stronger methods)

Motivation for Dominators (Recap)

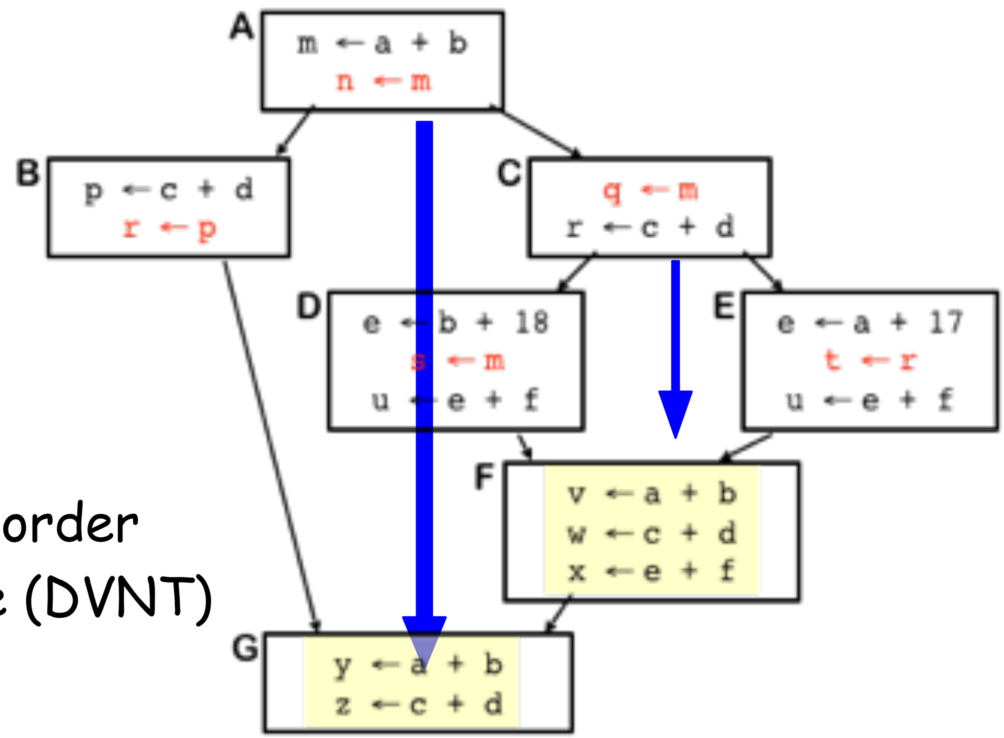
We have not helped with F or G

» Multiple predecessors

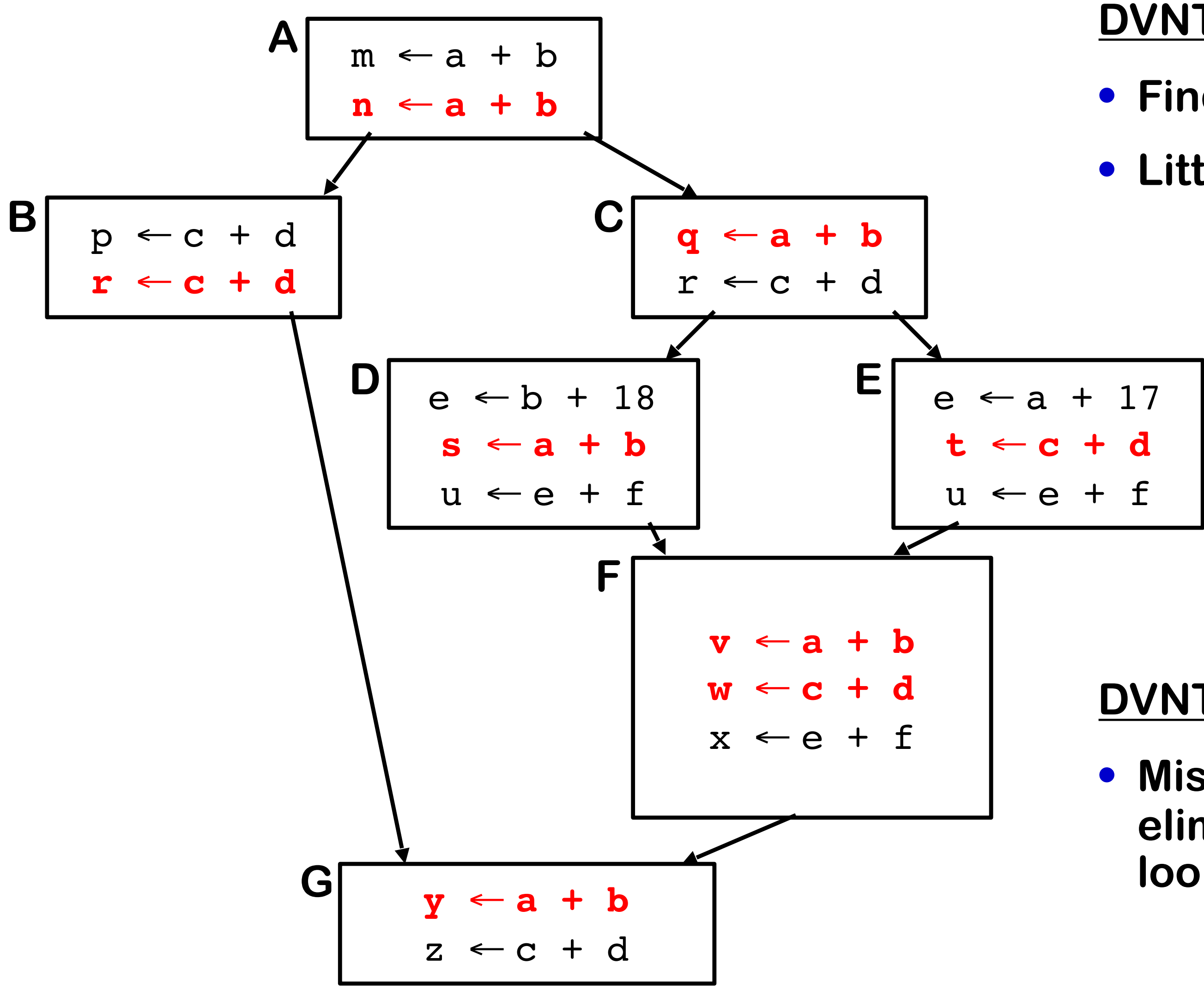
» Must decide what facts hold in F and in G

  - For G, combine B & F?
  - Merging state is expensive
  - Fall back on what's known

- Can use value numbers from block IDom(x) when processing x, e.g.,
  - Use C for F and A for G
- Imposes a Dom-based application order
- Leads to Dominator VN Technique (DVNT)



Dominator Value Numbering: Example



DVNT advantages

- Find more redundancy
- Little additional cost

DVNT shortcomings

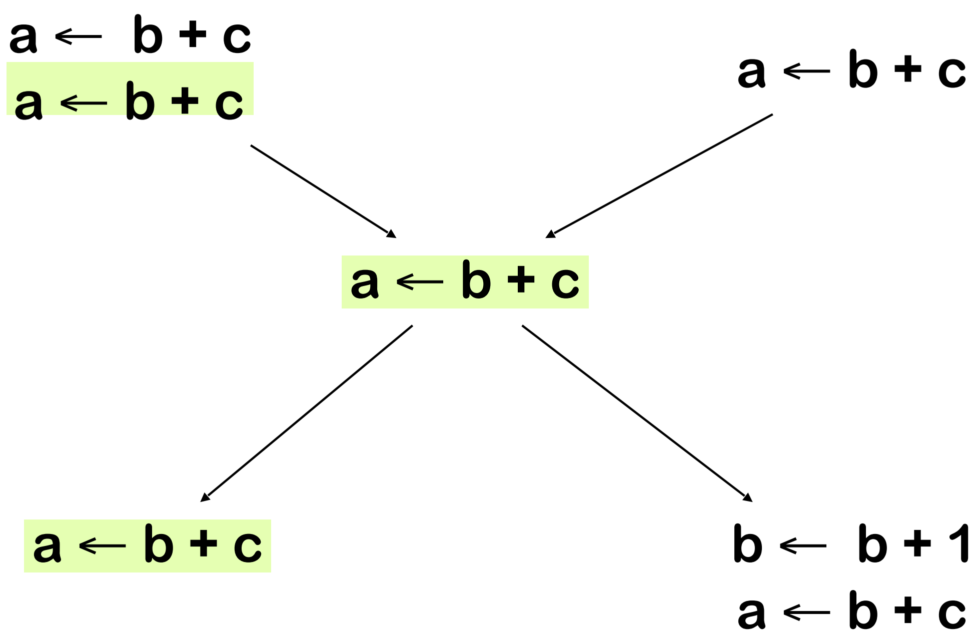
- Misses redundancy elimination opportunities in loops

Redundant Expression: General Definition

An expression is redundant at point p if, on every path to p

1. It is evaluated before reaching p, and
2. None of its constitutent values is redefined before p

Example

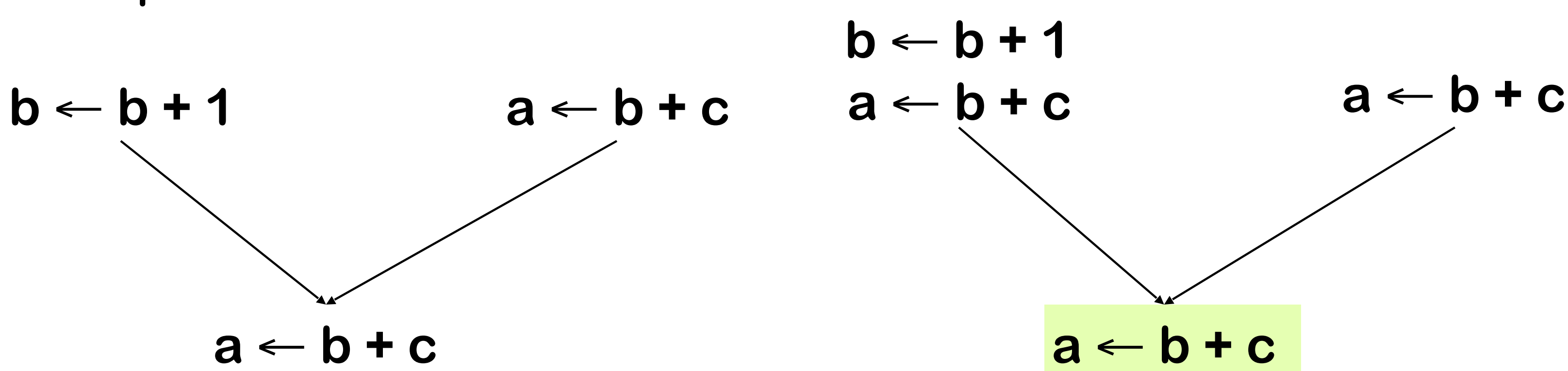


Some occurrences of b+c are redundant

Partially Redundant Expression

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

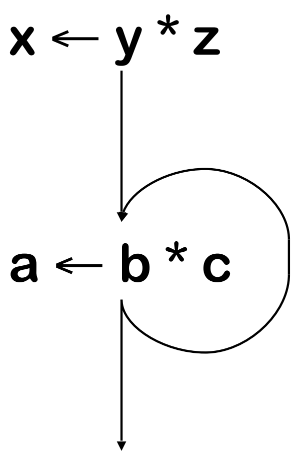
Example

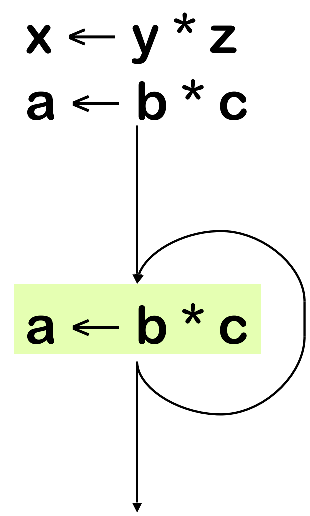


Inserting a copy of "a ← b + c" after the definition of b can make it (fully) redundant

Loop Invariant Expression

Another example





Loop invariant expressions are partially redundant

- Partial redundancy elimination performs code motion
- Major part of the work is figuring out where to insert operations
- Question: what if we had a while loop, instead of a do-while loop as in the above example?

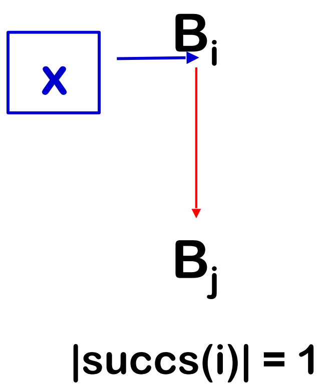
Lazy Code Motion

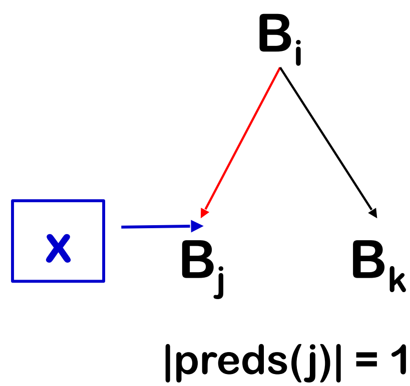
The concept

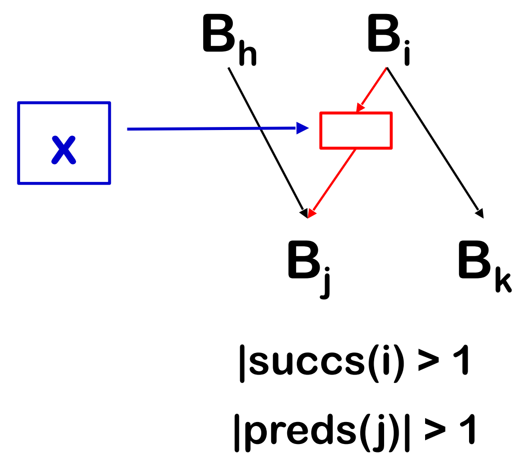
- Compute INSERT & DELETE sets by solving data flow subproblems
  - Compute <u>available expressions (AVAIL)</u>
    - Can be extended with value numbering
  - Compute <u>anticipable expressions (ANT)</u>
  - AVAIL and ANT are advanced concepts that are not required for this class
- Linear pass to rewrite code using INSERT & DELETE sets
  - $x \in$ INSERT$(i,j) \Rightarrow$ insert x at start of j, end of i, or new block
  - $x \in$ DELETE$(k) \Rightarrow$ delete first evaluation of x in k
- Lazy Code Motion (LCM) extends earlier work on Partial Redundancy Elimination (PRE)
- See textbook for details

Lazy Code Motion: Placement Rules

- x ∈ INSERT(i,j)





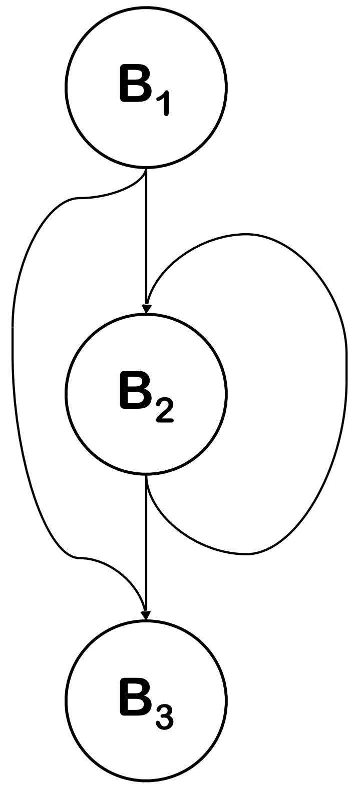


Three cases

- |succs(i)| = 1 ⇒ insert at end of i
- | succs(i)| > 1, but |preds(j)| = 1⇒ insert at start of j
- | succs(i)| > 1, & |preds(j)| > 1 ⇒ create new block in ‹i,j› for x
  - Modify CFG by adding a new basic block in this case

Lazy Code Motion: Example

B_1: $r_1 \leftarrow 1$
$r_2 \leftarrow r_0 + @m$
if $r_1 < r_2 \rightarrow B_2, B_3$
B_2: …
$r_{20} \leftarrow r_{17} * r_{18}$
…
$r_4 \leftarrow r_1 + 1$
$r_1 \leftarrow r_4$
if $r_1 < r_2 \rightarrow B_2, B_3$
B_3: …



Insert(1,2) = { r20 ← r17 * r18 }
Delete(2) = { r20 ← r17 * r18 }
==>
Move r20 ← r17 * r18 from start of B_2 to B_1—> B_2 edge

Available Expression Analysis

An expression x+y is available if and only if, along every path from the procedure's entry, it has been evaluated, and its constituent subexpressions (x & y) have not been re-defined.

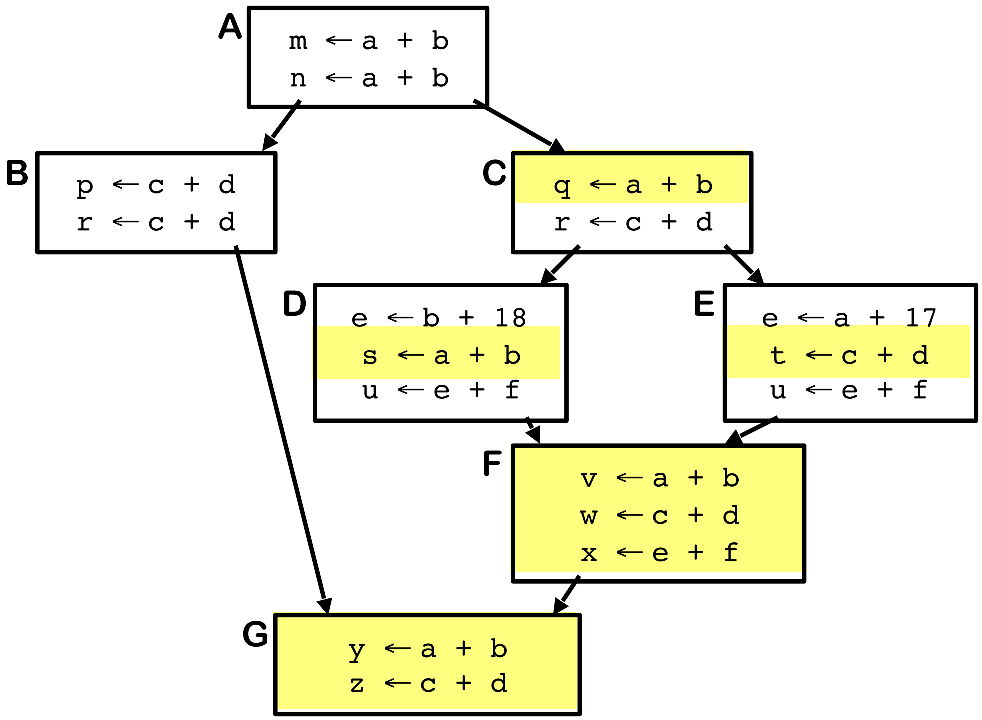
If the compiler can prove that an expression is available

- » It can preserve the results of earlier evaluations
- » It can replace the current evaluation with a reference

Two pieces to redundancy elimination

- » Proving that x+y is available
- » Rewriting the code to eliminate the redundant evaluation

Example



AVAIL(A) = Ø

AVAIL(B) = {a+b}

AVAIL(C) = {a+b}

AVAIL(D) = {a+b, c+d}

AVAIL(E) = {a+b, c+d}

AVAIL(F) = {a+b, c+d, e+f}

AVAIL(G) = {a+b, c+d}

Formal Definition of Available Expressions

For each block b, let

- » Avail(b) be the set of expressions available on entry to b
- » DEExpr(b) be the set of expressions computed in b and available on exit (Downward Exposed Expressions)
- » ExprKill(b) be these set of expressions that are killed in b
  - » An expression is killed one of its inputs is assigned a value

Now, Avail(b) can be defined as:

$$\text{Avail}(b) = \bigcap_{x \in \text{pred}(b)} (\text{DEExpr}(x) \cup (\text{Avail}(x) - \text{ExprKill}(x)))$$

preds(b) is the set of b's predecessors in the control-flow graph

- This system of simultaneous equations forms a data-flow problem, and can be solved as past data-flow problems that we've seen (reaching definitions, dominators)

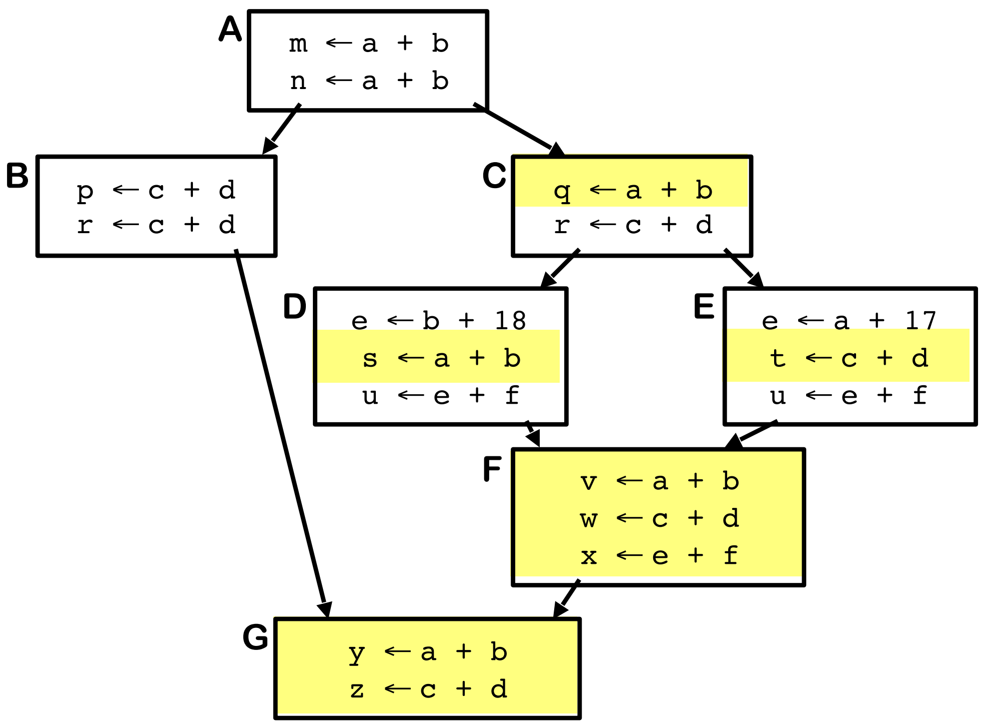
Computing Available Expressions

The Big Picture

1. Build a control-flow graph
2. Gather the initial (local) data — DEExpr(b) & ExprKill(b)
3. Propagate information around the graph, evaluating the equation
4. Use output of Available Expression analysis as needed, e.g., for lazy code motion or redundancy elimination

Computing Avail for the Example

	A	B	C	D	E	F	G
DEExpr	a+b	c+d	a+b, c+d	b+18, a+b, e+f	a+17, c+d, e+f	a+b, c+d, e+f	a+b, c+d
ExprKill	∅	∅	∅	e+f	e+f	∅	∅



AVAIL(A) = Ø

AVAIL(B) = {a+b} ∪ (ø ∩ all)
= {a+b}

AVAIL(C) = {a+b}

AVAIL(D) = {a+b, c+d} ∪ ({a+b} ∩ all)
= {a+b, c+d}

AVAIL(E) = {a+b, c+d}

AVAIL(F) = [{b+18, a+b, e+f} ∪
({a+b, c+d} ∩ {all - e+f})]
∩ [{a+17, c+d, e+f} ∪
({a+b, c+d} ∩ {all - e+f})]
= {a+b, c+d, e+f}

AVAIL(G) = [{c+d} ∪ ({a+b} ∩ all)]
∩ [{a+b, c+d, e+f} ∪
({a+b, c+d, e+f} ∩ all)]
= {a+b, c+d}

$$\text{Avail}(b) = \bigcap_{x \in pred(b)} (\text{DEExpr}(x) \cup (\text{Avail}(x) - \text{ExprKill}(x)))$$

Algorithm halts in one pass for this example, because graph is acyclic