CS 4240: Compilers

Lecture 4: Value Numbering, Dominators

Instructor: Vivek Sarkar (vsarkar@gatech.edu)
January 16, 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
- Forming project teams
  - We will create a 0-point (pseudo) assignment in Canvas for you to report your team members and implementation language. Deadline: TODAY
  - You can use "Search for Teammates!" in Piazza if needed.

Formalizing a Solution to the Reaching Definitions Problem (Recap)

- Given a CFG vertex S, define
  - Local sets that can be extracted from S
    - GEN[S] = set of definitions in S ("generated" by S)
    - KILL[S] = set of definitions that may be overwritten by S (e.g., all definitions in program that write to S's lval, whether or not they reach S)
  - Global sets to be computed using CFG
    - IN[S] = set of definitions that reach the entry point of S
    - OUT[S] = set of definitions in S as well as definitions from IN[S] that go beyond S (are not "killed" by S)
- Data flow equations (invariants) for these sets

$$\text{OUT[S]} = \text{GEN[S]} \cup (\text{IN[S]} - \text{KILL[S]})$$

$$\text{IN[S]} = \cup_{p \in predecessors} \text{OUT[p]}$$

Algorithm Summary: Inputs and Outputs (Recap)

- *Input:* A flow graph for which $kill[B]$ and $gen[B]$ have been computed for each basic block B
- *Output:* $in[B]$ and $out[B]$ for each block B
- The Idea: Use an iterative approach where the "initial" in and out information is *propagated* across edges and along the paths of the graph *until* none of the outs change

  All computation is at the granularity of basic-blocks

Algorithm Summary: Overall Steps (Recap)

Reaching Definitions:

- // Initialize *out* under the assumpt1¡ion that $in = \emptyset$ by setting $out[B] := gen[B]$ for all the blocks //
- $change :=$ **true**

  // This initiates the iteration and if there is a change after the iteration in *any* of the *out* sets, then it remains true//
- While *change* remains **true** compute
  - $in[B] = \cup_{p \in P}\, out[p]$ where P is the set of all predecessors of block B
- $tempout := out[B]$
- $out[B] := gen[B] \cup (in[B] - kill[B])$
- if $out[B] \neq \boldsymbol{tempout}\;\; change :=$ **true**

Summary available in Wikipedia page as well!
https://en.wikipedia.org/wiki/Reaching_definition

Using Reaching Definitions to improve Dead-code Elimination algorithm

Mark

```
for each op i
    clear i's mark
    if i is critical then
        // for simplicity, assume all
       // branch instructions are critical
        mark i
        add i to WorkList

while (Worklist ≠ Ø)
    remove i from  WorkList
        (i has form "x ← op y" or
         "x←y op z" )
      for each instruction j that
       contains a def of y or z that
       reaches i
        if j is not marked then
          mark j
          add j to WorkList
```

Sweep

```
for each op i
    if i is not marked then
        delete i
```

NOTES:

- A def reaches instruction i
1) if it is in the IN set for the basic block B(i) containing i, and
2) the def is not killed locally within B(i) before instruction i
- Condition 2) above can be omitted if reaching definitions analysis is performed on an instruction-level CFG
- Additional smarts are needed to also avoid marking branch instructions as critical

Some applications of Reaching Definitions

- Optimization examples
  - Identify dead (useless) code
  - Identify uses that can be replaced by constants (constant propagation)
  - Identify uses that can be replaced by a def's rval (copy propagation)
  - . . .

- Debugging examples
  - Identify uses of unitialized variables (uses that are not reached by any def)
  - . . .

Redundancy Elimination as an Example

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 <u>not</u> 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, or <u>available</u>
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called <u>value numbering</u>

Value Numbering

The key notion

- Assign an identifying "value number", V(i), to each expression (rval) in IR instruction i
- Invariant: if two expressions have the same value number, they will always have the same value
  - If value numbers are different, they may or may not always be the same

Applications of value numbers

- Replace redundant expression by previously computed value, instead of recomputing the expression, e.g.
  1. t1 = a + b
  2. t2 =a + b // can be replaced by "t2 = t1"
- Simplify algebraic identities, e.g., (t1-t2) can be replaced by zero
- Discover when value numbers denote constant-valued operands, in which case the operator can be evaluated at compile-time

Local Value Numbering

The Algorithm

For each operation o = <operator, o_1, o_2> in a basic block, in order

1. Get value numbers for operands from hash lookup
2. Hash <operator,VN(o_1),VN(o_2)> to get a value number for o
3. If o already had a value number, replace o with a reference
4. If o_1 & o_2 are constant, evaluate it & replace with a `loadI`

If hashing behaves, the algorithm runs in linear time

Handling algebraic identities

- Case statement on operator type
- Handle special cases within each operator

Local Value Numbering

An example (superscripts are value numbers, and are not part of the IR)

Original Code

$a \leftarrow x + y$
* $b \leftarrow x + y$
$a \leftarrow 17$
* $c \leftarrow x + y$

With VNs

$a^3 \leftarrow x^1 + y^2$
* $b^3 \leftarrow x^1 + y^2$
$a^4 \leftarrow 17$
* $c^3 \leftarrow x^1 + y^2$

Rewritten

$a^3 \leftarrow x^1 + y^2$
* $b^3 \leftarrow a^3$
$a^4 \leftarrow 17$
* $c^3 \leftarrow a^3$ (oops!)

Two redundancies

- Eliminate stmts with a *
- Coalesce results ?

Corrected

$t \leftarrow x^1 + y^2$
$a \leftarrow t$
* $b^3 \leftarrow a^3$
$a^4 \leftarrow 17$
* $c^3 \leftarrow t$

Options

- Use $c^3 \leftarrow b^3$
- Save a^3 in t^3
- Rename around it (next slide)
- Introduce a temporary (corrected code on left)
- . . .

Local Value Numbering with Renaming

Example (continued — add subscripts to rename variables)

Original Code	**With VNs**	**Rewritten**
$a_0 \leftarrow x_0 + y_0$	$a_0^3 \leftarrow x_0^1 + y_0^2$	$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0 \leftarrow x_0 + y_0$	* $b_0^3 \leftarrow x_0^1 + y_0^2$	* $b_0^3 \leftarrow a_0^3$
$a_1 \leftarrow 17$	$a_1^4 \leftarrow 17$	$a_1^4 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$	* $c_0^3 \leftarrow x_0^1 + y_0^2$	* $c_0^3 \leftarrow a_0^3$

Renaming:

- Give each value a unique name
- Makes it clear

Notation:

- While complex, the meaning is clear

Result:

- a_0^3 is available
- Rewriting now works

Local Value Numbering

The LVN Algorithm, with bells & whistles

Block is a sequence of n operations of the form

$T_i \leftarrow L_i \; Op_i \; R_i$

for i ← 0 to n-1

1. get the value numbers V_1 and V_2 for L_i and R_i
2. if L_i and R_i are both constant then

    evaluate Li Op_i R_i, assign it to T_i, and mark T_i as a constant

    Constant folding

3. if Li Op_i R_i matches an identity then

    replace it with a copy operation or an assignment

    Algebraic identities

4. if Op_i commutes and $V_1 > V_2$ then

    swap V_1 and V_2

    Commutativity

5. construct a hash key $<V_1,Op_i,V_2>$
6. if the hash key is already present in the table then

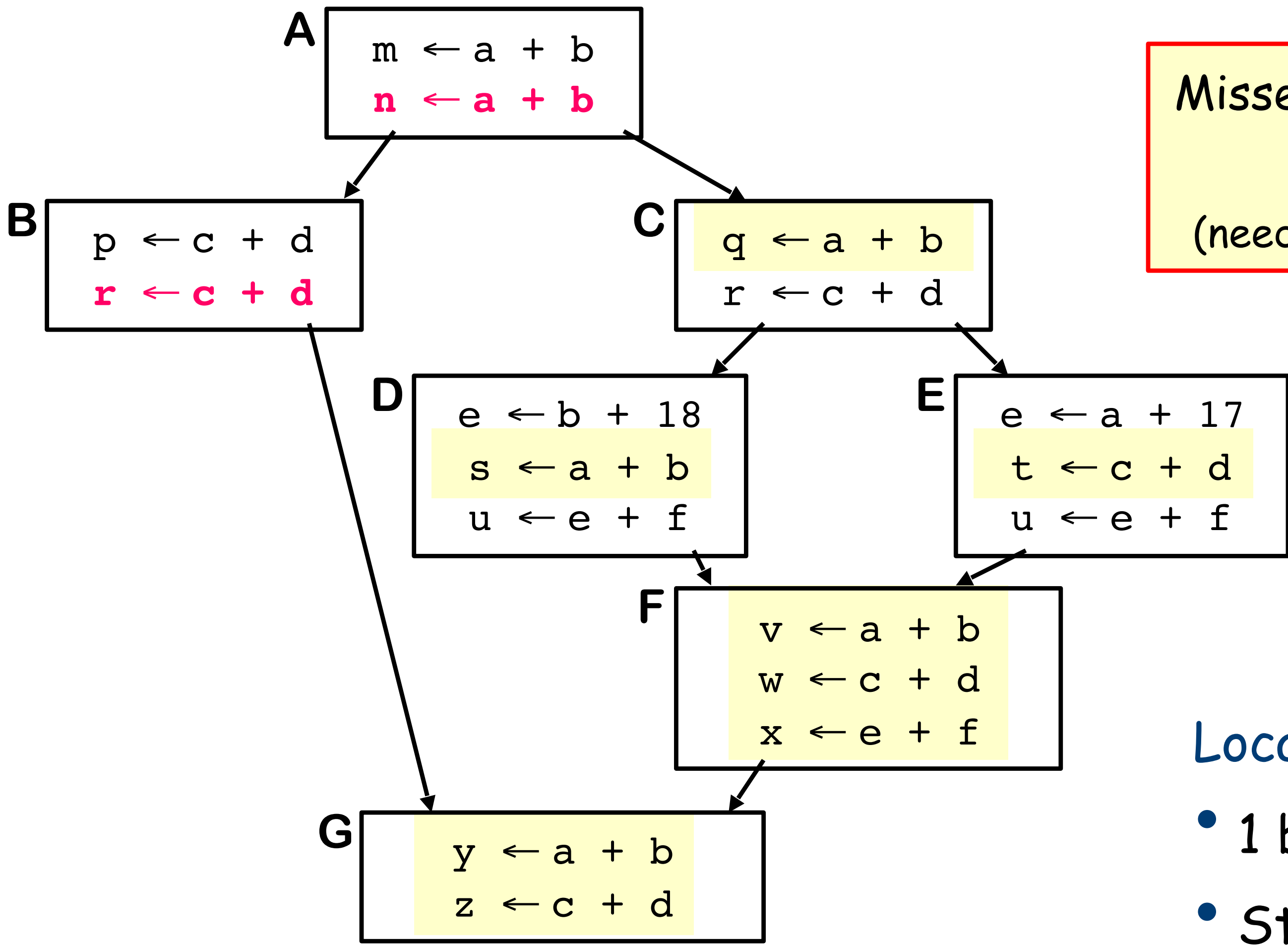
    replace operation I with a copy into T_i and mark T_i with the VN

    else

    insert a new VN into table for hash key & mark T_i with the VN

Limitations of Local Value Numbering

LVN finds redundant ops in red



Missed opportunities in yellow
(need stronger methods)

Local Value Numbering

- 1 block at a time
- Strong local results
- No cross-block effects

Scope of Optimization

A basic block is a sequence of straight-line code.

Local optimization

- Operates entirely within a single basic block
- Properties of block lead to strong optimizations

Regional optimization

- Operate on a region in the CFG that contains multiple blocks
- Loops, trees, paths, extended basic blocks

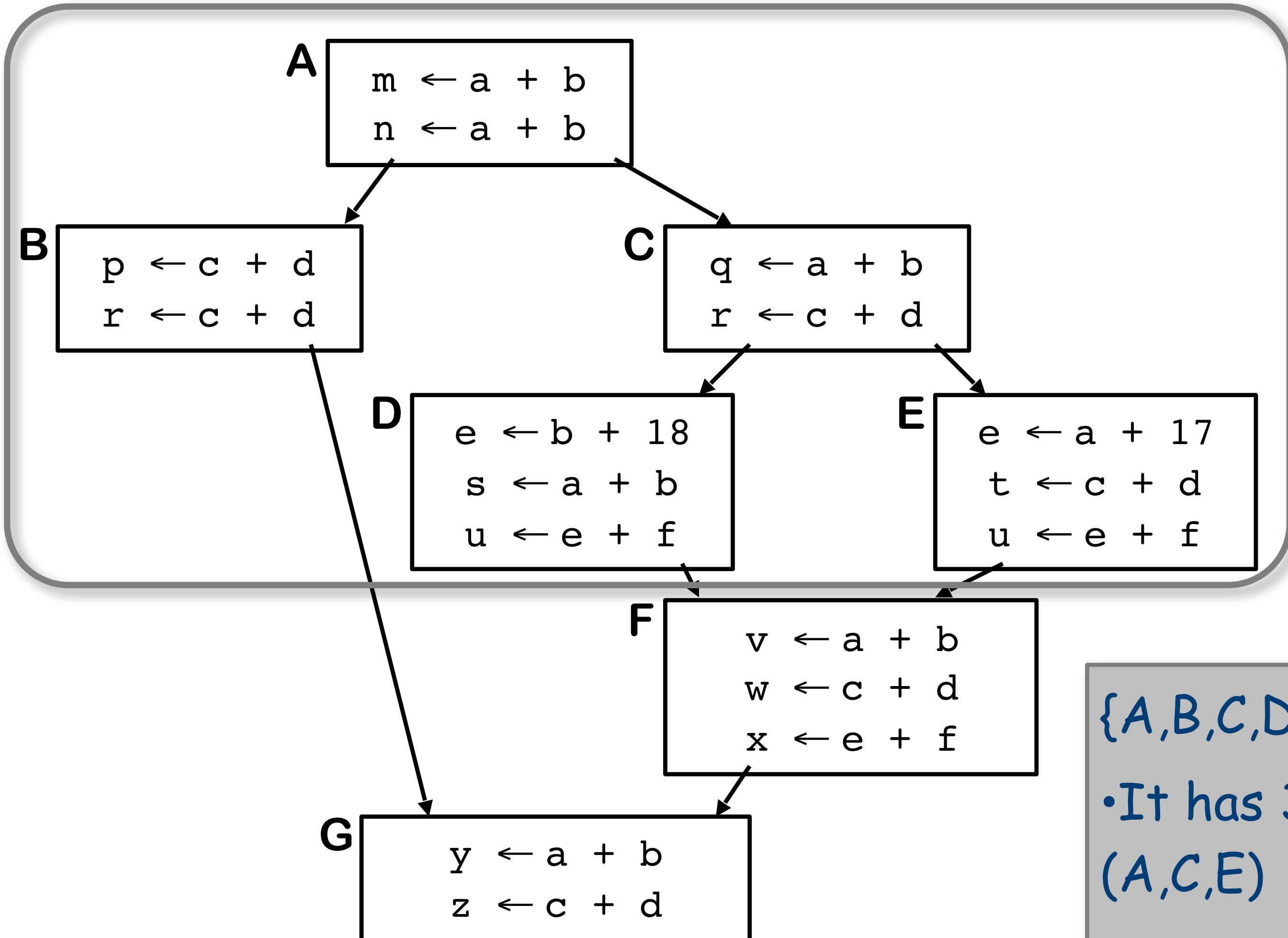
Whole procedure optimization (intraprocedural)

- Operate on entire CFG for a procedure
- Presence of cyclic paths forces analysis then transformation

Whole program optimization (interprocedural)

- Operate on some or all of the call graph (multiple procedures)
- Must contend with call/return & parameter binding

Superlocal Value Numbering (SVN)



EBB: A maximal set of blocks B_1, B_2, …, B_n where each B_i, except B_1, has only exactly one predecessor and that block is in the EBB.

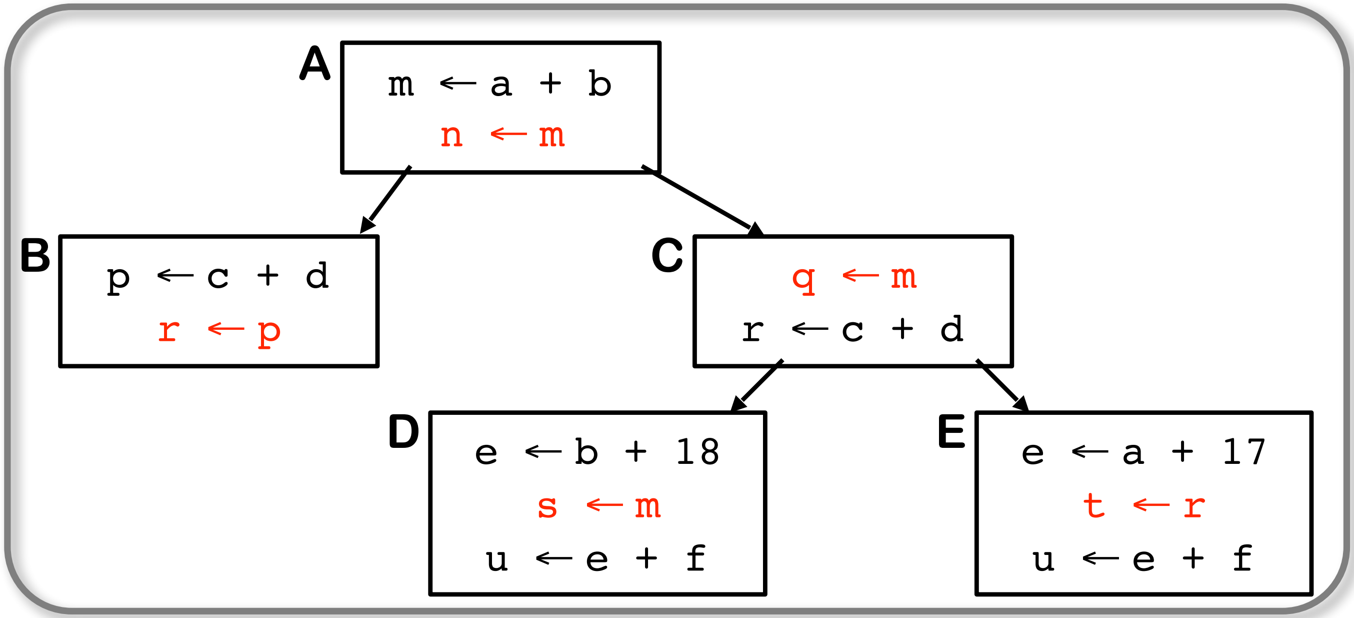
{A,B,C,D,E} is an EBB

- It has 3 paths: (A,B), (A,C,D), & (A,C,E)
- Can sometimes treat each path as if it were a block

{F} & {G} are degenerate EBBs

Superlocal: "applied to an EBB"

After Superlocal Value Numbering (SVN)



EBB: A maximal set of blocks B_1, B_2, …, B_n where each B_i, except B_1, has only exactly one predecessor and that block is in the EBB.

- Capture expression in temporaries to avoid bugs if variable m is rewritten

{A,B,C,D,E} is an EBB

•It has 3 paths: (A,B), (A,C,D), & (A,C,E)

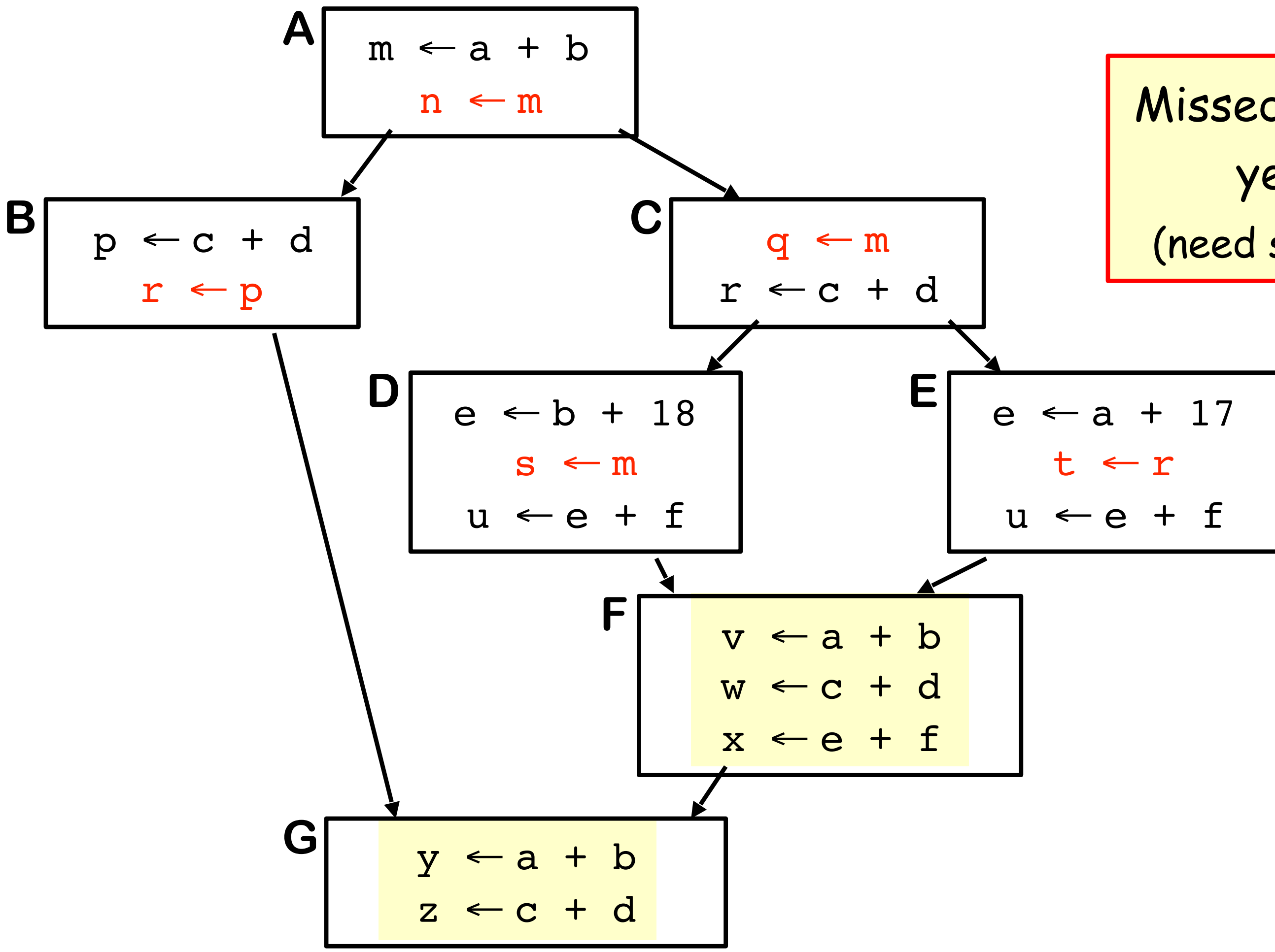
•Can sometimes treat each path as if it were a block

{F} & {G} are degenerate EBBs

Superlocal: "applied to an EBB"

Limitations of Superlocal Value Numbering

SVN finds redundant ops in red



Missed opportunities in yellow remain!

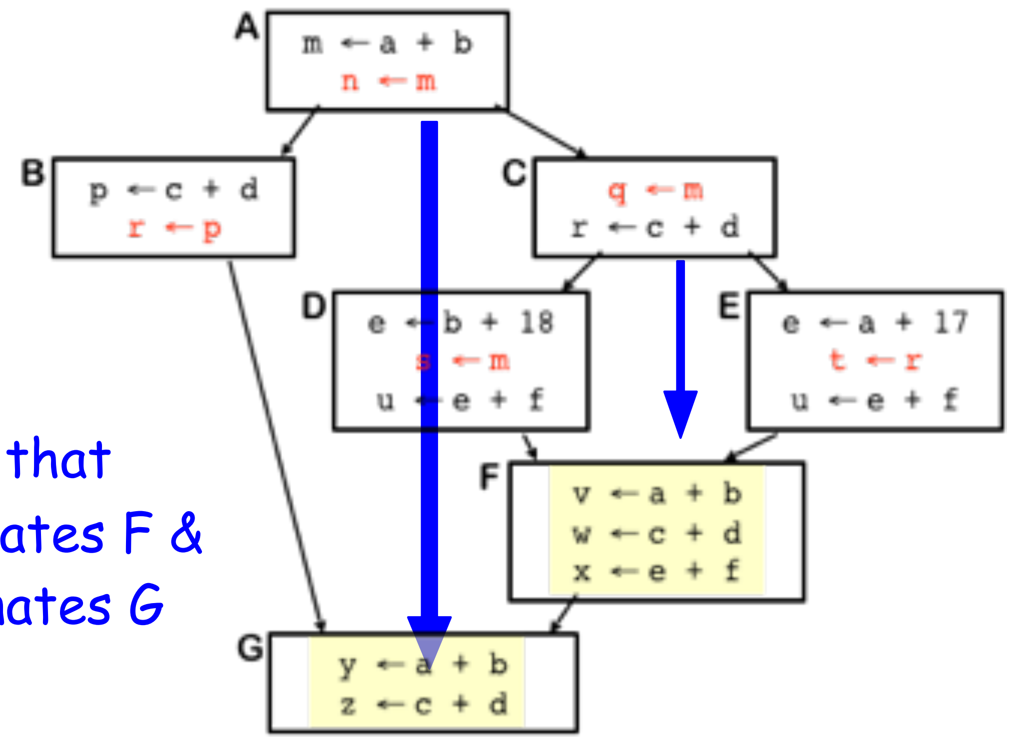
(need stronger methods)

What About Larger Scopes?

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



C lies on every path to F

A lies on every path to G

We say that
C dominates F &
A dominates G

Dominators

Definitions

x <u>dominates</u> y if and only if every acyclic path from the entry of the control-flow graph to the node for y includes x

- By definition, x <u>dominates</u> x
- We associate a Dom set with each node
- |Dom(x)| ≥ 1

Immediate dominators

- For any node x, there must be a y in Dom(x) closest to x
- We call this y the <u>immediate</u> <u>dominator</u> of x
- As a matter of notation, we write this as IDom(x)
- Dominator Tree defined with root = entry, and IDom has parent map

Dominators

Dominators have many uses in analysis & transformation

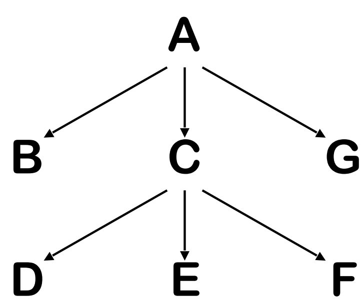
» More general value numbering

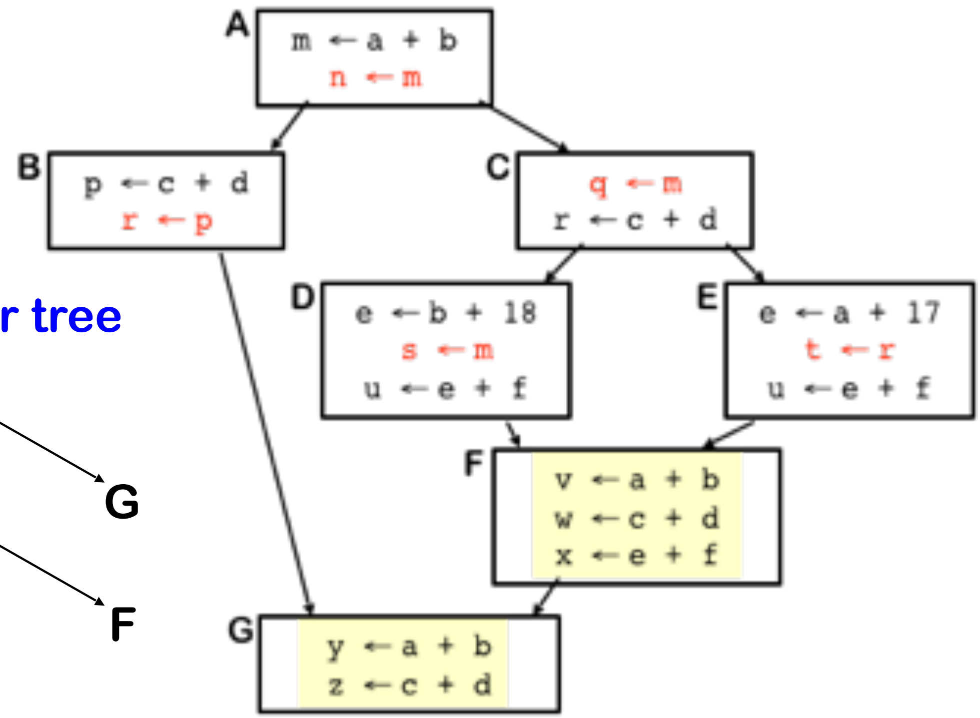
» Finding loops

Dominator sets

Block	Dom	IDom
A	A	–
B	A,B	A
C	A,C	A
D	A,C,D	C
E	A,C,E	C
F	A,C,F	C
G	A,G	A

Dominator tree





Let's now look at how to compute dominators

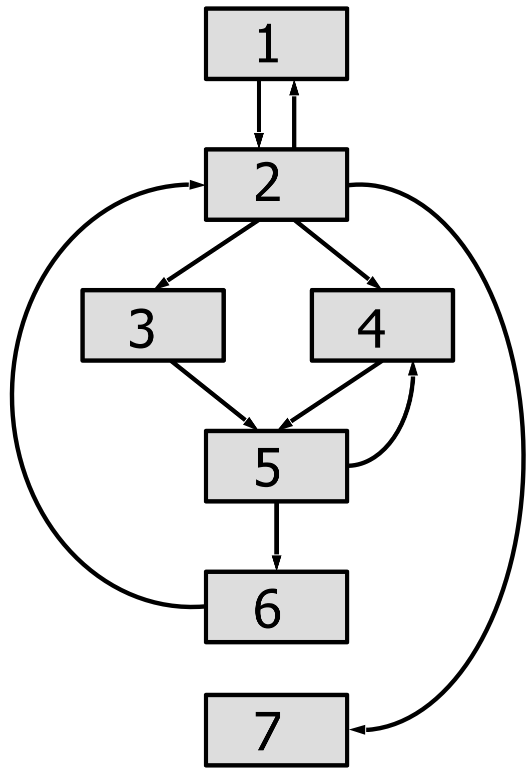
Immediate Dominators

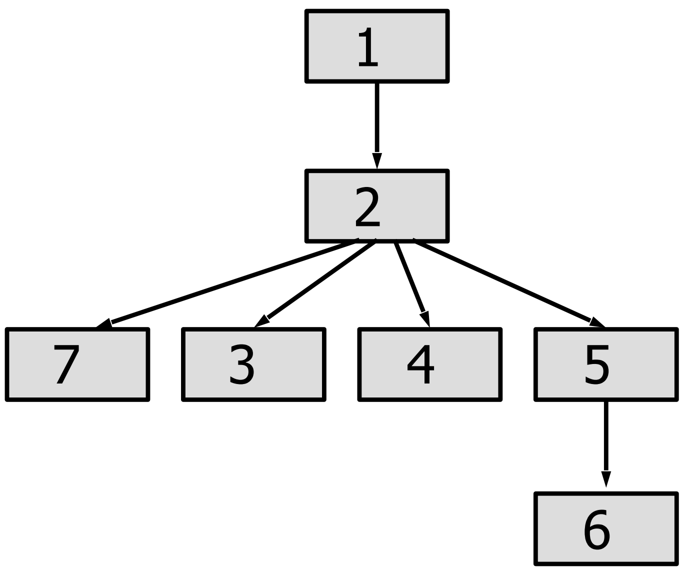
- Properties:
  1. CFG entry node n_0 in dominates all CFG nodes
  2. If d1 and d2 dominate n, then either
     - d1 dominates d2, or
     - d2 dominates d1
- The immediate dominator idom(n) of a node n is the unique last strict dominator on any path from n_0 to n

Dominator Tree

- Build a dominator tree as follows:
  - Root is CFG entry node n_0
  - m is child of node n iff n=idom(m)

- Example:





Today’s in-class Worksheet

- Worksheets can be solved collaboratively
  - All other course work must be done individually or in project groups (see syllabus for details)
- Each student should turn in their own solution, based on collaborative discussions
- Worksheets will not be graded or returned, but solutions will be provided
- Worksheets will contribute to class participation grade
- Worksheets will inform teaching staff of concepts that need to be reviewed/reinforced in future lectures