

# CSE P 501 – Compilers

Value Numbering & Optimizations

Hal Perkins

Winter 2016

# Agenda

- Optimization (Review)
  - Goals
  - Scope: local, superlocal, regional, global (intraprocedural), interprocedural
- Control flow graphs (reminder)
- Value numbering
- Dominators
- Ref.: Cooper/Torczon ch. 8

# Code Improvement (1)

- Pick a better algorithm(!)
- Use machine resources efficiently
  - Instructions, registers
  - More later...

## Code Improvement (2)

- Local optimizations – basic blocks
  - Algebraic simplifications
  - Constant folding
  - Common subexpression elimination (i.e., redundancy elimination)
  - Dead code elimination
  - Specialize computation based on context
  - etc., etc., ...

## Code Improvement (3)

- Global optimizations
  - Code motion
  - Moving invariant computations out of loops
  - Strength reduction (replace multiplications by repeated additions, for example)
  - Global common subexpression elimination
  - Global register allocation
  - Many others...

# “Optimization”

- None of these improvements are truly “optimal”
  - Hard problems (in theory-of-computation sense)
  - Proofs of optimality assume artificial restrictions
- Best we can do is to improve things
  - Most (much?) (some?) of the time
  - Realistically: try to do better for common idioms both in the code and on the machine

# Optimization Phase

- Goal
  - Discover, at compile time, information about the runtime behavior of the program, and use that information to improve the generated code

## A First Running Example: Redundancy Elimination

$\neq$   
 $P: \neq$

- An expression  $x+y$  is *redundant* at a program point iff, along every path from the procedure's entry, it has been evaluated and its constituent subexpressions ( $x$  and  $y$ ) have not been redefined
- If the compiler can prove the expression is redundant:
  - Can store the result of the earlier evaluation
  - Can replace the redundant computation with a reference to the earlier (stored) result



## Common Pattern for Code Improvement

- Typical for most compiler optimizations
- First, discover opportunities through program analysis
- Then, modify the IR to take advantage of the opportunities
  - Historically, goal usually was to decrease execution time
  - Other possibilities: reduce space, power, ...

## Issues (1)

- Safety – transformation must not change program meaning
  - Must generate correct results
  - Can't generate spurious errors
  - Optimizations must be conservative
  - Large part of analysis goes towards proving safety
  - Can pay off to speculate (be optimistic) but then need to recover if reality is different

## Issues (2)

- Profitability
  - If a transformation is possible, is it profitable?
  - Example: loop unrolling
    - Can increase amount of work done on each iteration, i.e., reduce loop overhead
    - Can eliminate duplicate operations done on separate iterations

## Issues (3)

- Downside risks
  - Even if a transformation is generally worthwhile, need to think about potential problems
  - For example:
    - Transformation might need more temporaries, putting additional pressure on registers
    - Increased code size could cause cache misses, or, in bad cases, increase page working set

## Example: Value Numbering

- Technique for eliminating redundant expressions: assign an identifying number  $VN(n)$  to each expression
  - $VN(x+y) = VN(j)$  if  $x+y$  and  $j$  have the same value
  - Use hashing over value numbers for efficiency
- Old idea (Balke 1968, Ershov 1954)
  - Invented for low-level, linear IRs
  - Equivalent methods exist for tree IRs, e.g., build a DAG

# Uses of Value Numbers

- Improve the code
  - Replace redundant expressions
  - Simplify algebraic identities
  - Discover, fold, and propagate constant valued expressions

# Local Value Numbering

A handwritten diagram showing an operation  $o = o1 \text{ op } o2$ . Above the expression, there are four horizontal lines representing a stack. Below the expression, a green line is drawn under the entire expression, with the label 'VN' written below it. Below the 'VN' label, the number '0' is written.

- Algorithm
  - For each operation  $o = \langle op, o1, o2 \rangle$  in a block
    - 1. Get value numbers for operands from hash lookup
    - 2. Hash  $\langle op, VN(o1), VN(o2) \rangle$  to get a value number for  $o$  (If  $op$  is commutative, sort  $VN(o1), VN(o2)$  first)
    - 3. If  $o$  already has a value number, replace  $o$  with a reference to the value
    - 4. If  $o1$  and  $o2$  are constant, evaluate  $o$  at compile time and replace with an immediate load
- If hashing behaves well, this runs in linear time

## Example

Code

$$\begin{aligned}a^3 &= x^1 + y^2 \\ b^3 &= x^1 + y^2 \\ a^4 &= 17^4 \\ c^3 &= x^1 + y^2\end{aligned}$$

Rewritten

$$\begin{aligned}a^3 &= x^1 + y^1 \\ b^3 &= a^3 \\ a^4 &= 17^4 \\ c^3 &= a^3\end{aligned}$$

vn table

expr	vn
x	1
y	2
<1,+>	3
17	4
a	3
b	3
a	4



## Bug in Simple Example

- If we use the original names, we get in trouble when a name is reused
- Solutions
  - Be clever about which copy of the value to use (e.g., use `c=b` in last statement)
  - Create an extra temporary
  - Rename around it (best!)

# Renaming



- Idea: give each value a unique name
  - $a_i^j$  means  $i^{\text{th}}$  definition of  $a$  with  $\text{VN} = j$
- Somewhat complex notation, but meaning is clear
- This is the idea behind SSA (Static Single Assignment)
  - Popular modern IR – exposes many opportunities for optimizations

## Example Revisited

Code

$$a_0^3 = x_0^1 + y_0^2$$

$$b_0^3 = x_0^1 + y_0^2$$

$$a_1^4 = 17^4$$

$$c_0^3 = x_0^1 + y_0^2$$

Rewritten

$$a_0^3 = x_0^1 + y_0^2$$

$$b_0^3 = a_0^3$$

$$a_1^4 = 17^4$$

$$c_0^3 = a_0^3$$

## Simple Extensions to Value Numbering

- Constant folding
  - Add a bit that records when a value is constant
  - Evaluate constant values at compile time
  - Replace op with load immediate
- Algebraic identities:  $x+0$ ,  $x*1$ ,  $x-x$ , ...
  - Many special cases
    - Switch on op to narrow down checks needed
    - Replace result with input VN

## Larger Scopes

- This algorithm works on straight-line blocks of code (basic blocks)
  - Best possible results for single basic blocks
  - Loses all information when control flows to another block
- To go further we need to represent multiple blocks of code and the control flow between them

## Control Flow Graph (CFG) reminder

- Nodes: basic blocks
  - Key property: all statements executed sequentially if any are
- Edges: include a directed edge from n1 to n2 if there is *any* possible way for control to transfer from block n1 to n2 during execution

# Optimization Categories (1)

- *Local methods*
  - Usually confined to basic blocks
  - Simplest to analyze and understand
  - Most precise information

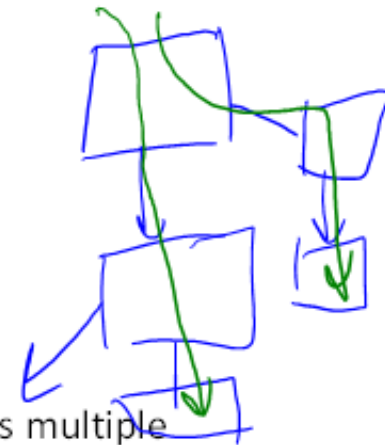
## Optimization Categories (2)

- *Superlocal methods*

- Operate over *Extended Basic Blocks* (EBBs)

- An EBB is a set of blocks  $b_1, b_2, \dots, b_n$  where  $b_1$  has multiple predecessors and each of the remaining blocks  $b_i$  ( $2 \leq i \leq n$ ) have only  $b_{i-1}$  as its unique predecessor
- The EBB is entered only at  $b_1$ , but may have multiple exits
- A single block  $b_i$  can be the head of multiple EBBs (these EBBs form a tree rooted at  $b_i$ )

- Use information discovered in earlier blocks to improve code in successors

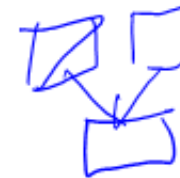




## Optimization Categories (3)

- *Regional methods*

- Operate over scopes larger than an EBB but smaller than an entire procedure/function/method
- Typical example: loop body
- Difference from superlocal methods is that there may be merge points in the graph (i.e., a block with two or more predecessors)
  - Facts true at merge point are facts known to be true on all possible paths to that point



## Optimization Categories (4)

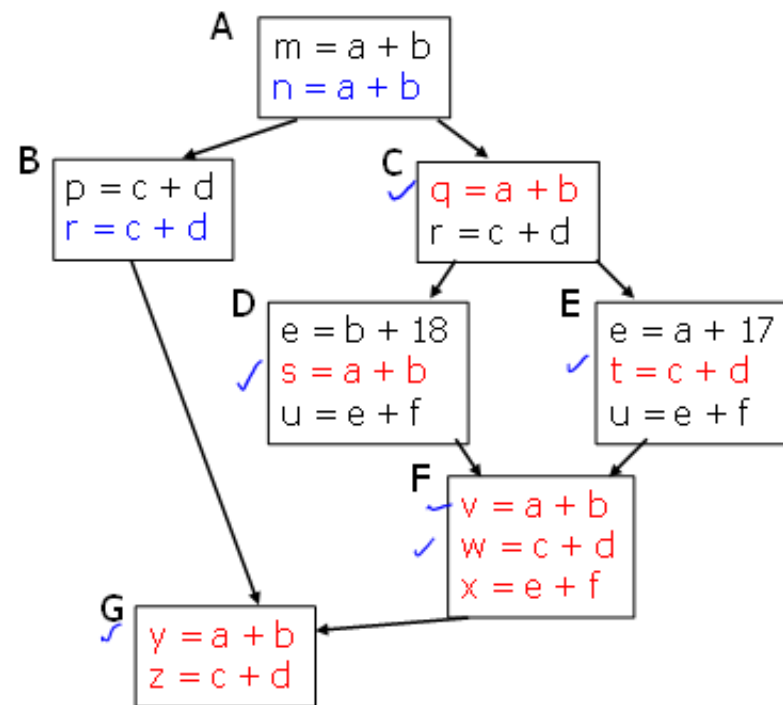
- *Global methods*
  - Operate over entire procedures
  - Sometimes called *intraprocedural* methods
  - Motivation is that local optimizations sometimes have bad consequences in larger context
  - Procedure/method/function is a natural unit for analysis, separate compilation, etc.
  - Almost always need global *data-flow* analysis information for these

## Optimization Categories (5)

- *Whole-program methods*
  - Operate over more than one procedure
  - Sometimes called *interprocedural* methods
  - Challenges: name scoping and parameter binding issues at procedure boundaries
  - Classic examples: inline method substitution, interprocedural constant propagation
  - Common in aggressive JIT compilers and optimizing compilers for object-oriented languages

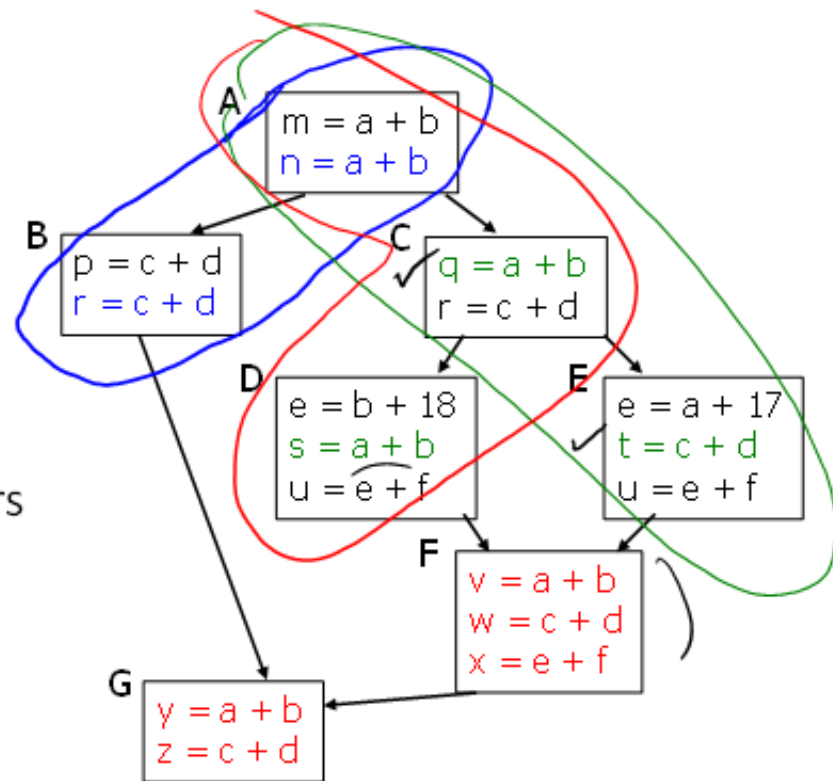
# Value Numbering Revisited

- Local Value Numbering
  - 1 block at a time
  - Strong local results
  - No cross-block effects
- Missed opportunities



# Superlocal Value Numbering

- Idea: apply local method to EBBs
  - {A,B}, {A,C,D}, {A,C,E}
- Final info from A is initial info for B, C; final info from C is initial for D, E
- Gets reuse from ancestors
- Avoid reanalyzing A, C
- Doesn't help with F, G



## SSA Name Space (from before)

Code

$$a_0^3 = x_0^1 + y_0^2$$

$$b_0^3 = x_0^1 + y_0^2$$

$$a_1^4 = 17$$

$$c_0^3 = x_0^1 + y_0^2$$

Rewritten

$$a_0^3 = x_0^1 + y_0^2$$

$$b_0^3 = a_0^3$$

$$a_1^4 = 17$$

$$c_0^3 = a_0^3$$

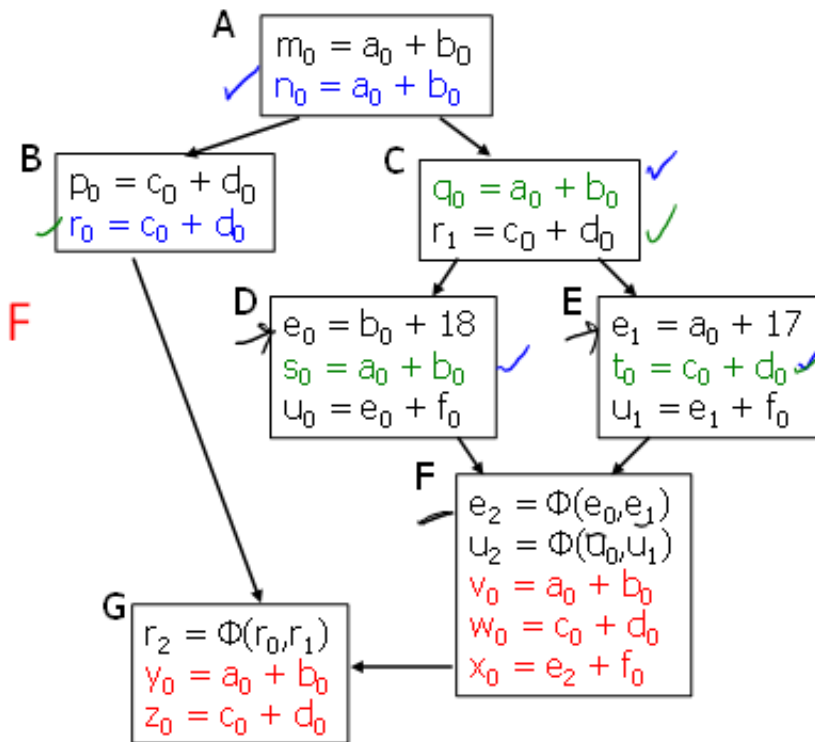
- Unique name for each definition
- Name  $\Leftrightarrow$  VN
- $a_0^3$  is available to assign to  $c_0^3$

# SSA Name Space

- Two Principles
  - Each name is defined by exactly one operation
  - Each operand refers to exactly one definition
- Need to deal with merge points
  - Add  $\Phi$  functions at merge points to reconcile names
  - Use subscripts on variable names for uniqueness

# Superlocal Value Numbering with All Bells & Whistles

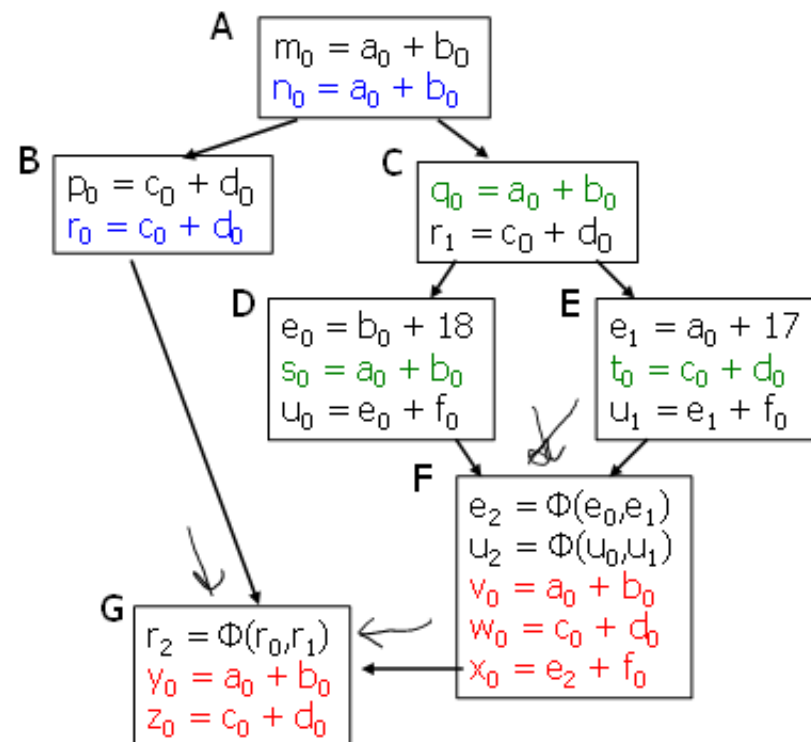
- Finds more redundancies
- Little extra cost
- Still does nothing for F and G





# Larger Scopes

- Still have not helped F and G
- Problem: multiple predecessors
- Must decide what facts hold in F and in G
  - For G, combine B & F?
  - Merging states is expensive
  - Fall back on what we know



# Dominators



- Definition
  - $x$  *dominates*  $y$  iff every path from the entry of the control-flow graph to  $y$  includes  $x$
- By definition,  $x$  dominates  $x$
- Associate a Dom set with each node
  - $| \text{Dom}(x) | \geq 1$
- Many uses in analysis and transformation
  - Finding loops, building SSA form, code motion

## Immediate Dominators

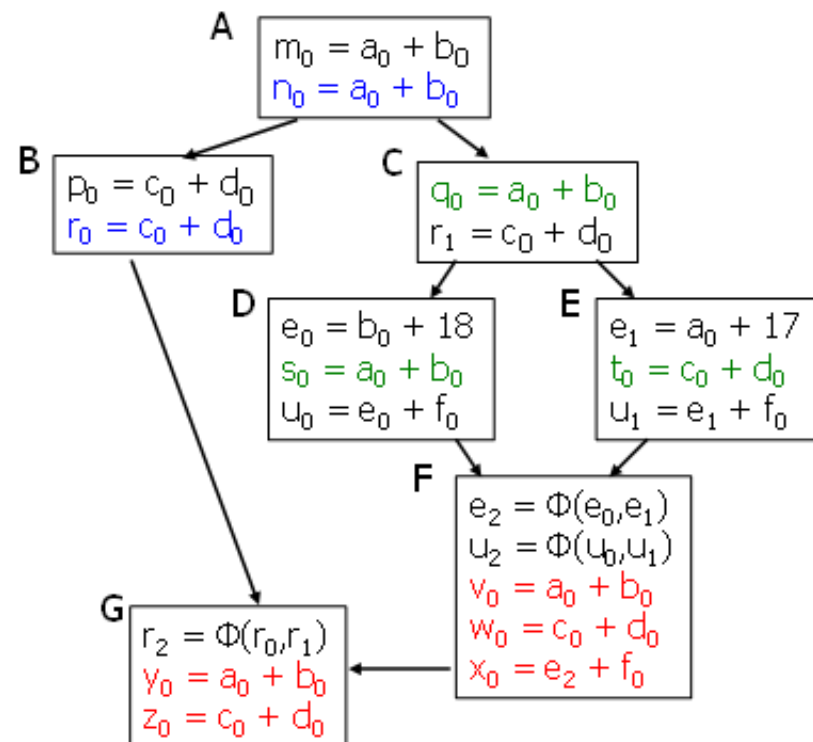
- For any node  $x$ , there is a  $y$  in  $\text{Dom}(x)$  closest to  $x$
- This is the *immediate dominator* of  $x$ 
  - Notation:  $\text{IDom}(x)$



# Dominator Sets

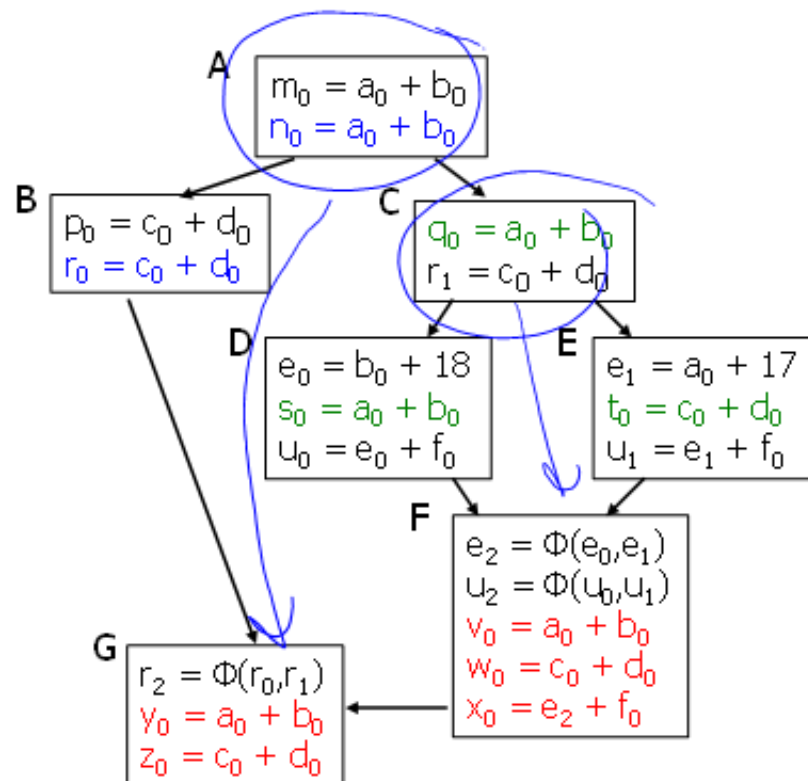
Block	Dom	IDom
A	A	-
B	A, B	A
C	A, C	A
D	A, C, D	C
E	A, C, E	C
F	A, C, F	C
G	A, G	A

Note that the IDOM relation defines a tree!



# Dominator Value Numbering

- Still looking for a way to handle F and G
- Idea: Use info from  $IDom(x)$  to start analysis of  $x$ 
  - Use C for F and A for G
- Dominator VN Technique (DVNT)

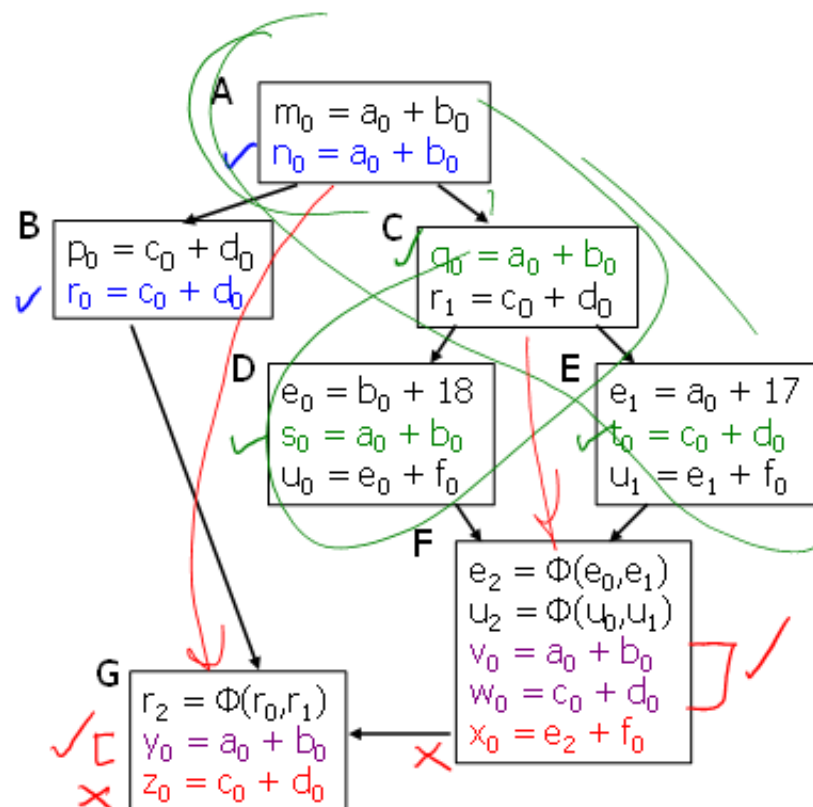


## DVNT algorithm

- Use superlocal algorithm on extended basic blocks
  - Use scoped hash tables & SSA name space as before
- Start each node with table from its IDOM
- No values flow along back edges (i.e., loops)
- Constant folding, algebraic identities as before

# Dominator Value Numbering

- Advantages
  - Finds more redundancy
  - Little extra cost
- Shortcomings
  - Misses some opportunities (common calculations in ancestors that are not IDOMs)
  - Doesn't handle loops or other back edges



## The Story So Far...

- Local algorithm
- Superlocal extension
  - Some local methods extend cleanly to superlocal scopes
- Dominator VN Technique (DVNT)
- All of these propagate along forward edges
- None are global



# Coming Attractions

- Data-flow analysis
  - Provides global solution to redundant expression analysis
    - Catches some things missed by DVNT, but misses some others
  - Generalizes to many other analysis problems, both forward and backward
- Loops
- SSA for general transformations