COMP 412
FALL 2010

# Code Optimization, Part II
# Regional Techniques

## Comp 412

# Last Lecture

## Introduced concept of a redundant expression

*An expression, x+y, is redundant at point p if, along each path from the procedure's entry point to p, x+y has already been evaluated and neither x nor y has been redefined.*

- If *x+y* is redundant at *p*, we can save the results of those earlier evaluations and reuse them at *p*, avoiding evaluation
- In a single block, we need only consider one such path
- We developed an algorithm for redundancy elimination in a single basic block
- Two pieces to the problem
  - Proving that *x+y* is redundant
  - Rewriting the code to eliminate the redundant evaluation
- Value numbering does both for straightline code

# Local Value Numbering (Recap)

## The LVN Algorithm, with bells & whistles

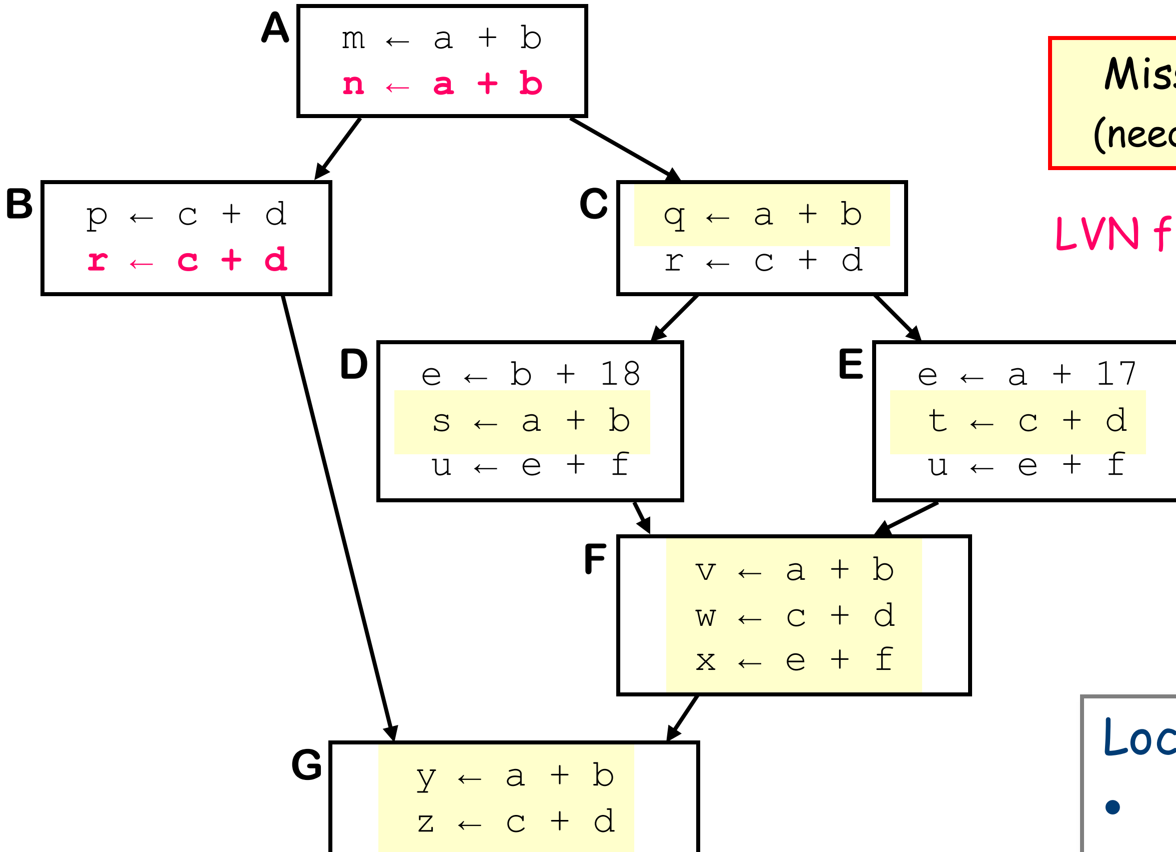
Block is a sequence of n operations of the form

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

*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* — Constant folding
   *evaluate Li* $Op_i$ $R_i$*, assign it to* $T_i$*, and mark* $T_i$ *as a constant*
3. *if Li* $Op_i$ $R_i$ *matches an identity then* — Algebraic identities
   *replace it with a copy operation or an assignment*
4. *if* $Op_i$ *commutes and* $V_1 > V_2$ *then* — Commutativity
   *swap* $V_1$ *and* $V_2$
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*

# Local Value Numbering



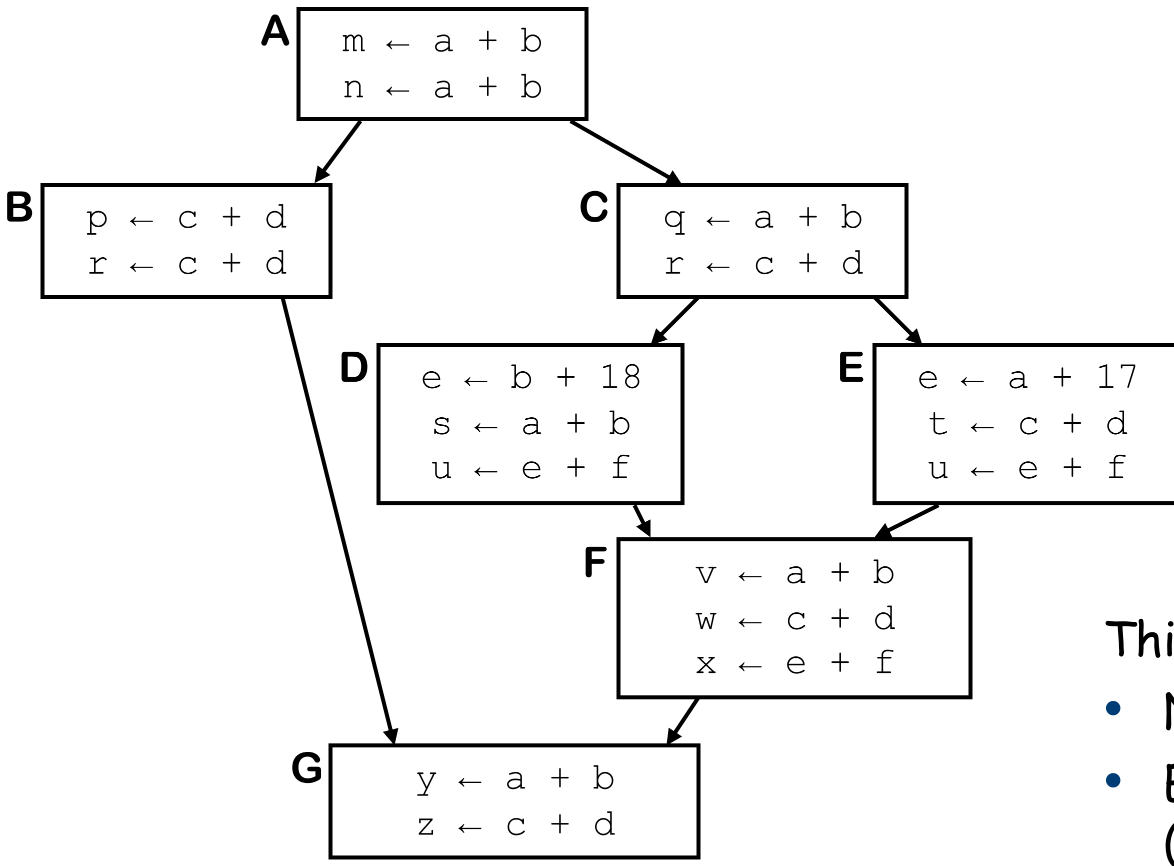

Missed opportunities
(need stronger methods)

LVN finds these redundant ops

**Local Value Numbering**

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

# Terminology




Control-flow graph (CFG)

- Nodes for basic blocks
- Edges for branches
- Basis for much of program analysis & transformation

This CFG, G = (N,E)

- N = {A,B,C,D,E,F,G}
- E = {(A,B),(A,C),(B,G),(C,D),(C,E),(D,F),(E,F),(F,E)}
- |N| = 7, |E| = 8

# Scope of Optimization

In scanning and parsing, "scope" refers to a region of the code that corresponds to a distinct name space.

In optimization "scope" refers to a region of the code that is subject to analysis and transformation.

- Notions are somewhat related
- Connection is not necessarily intuitive

Different scopes introduces different challenges & different opportunities

Historically, optimization has been performed at several distinct scopes.

# Scope of Optimization

A basic block is a maximal length sequence of straightline 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

# A Comp 412 Fairy Tale

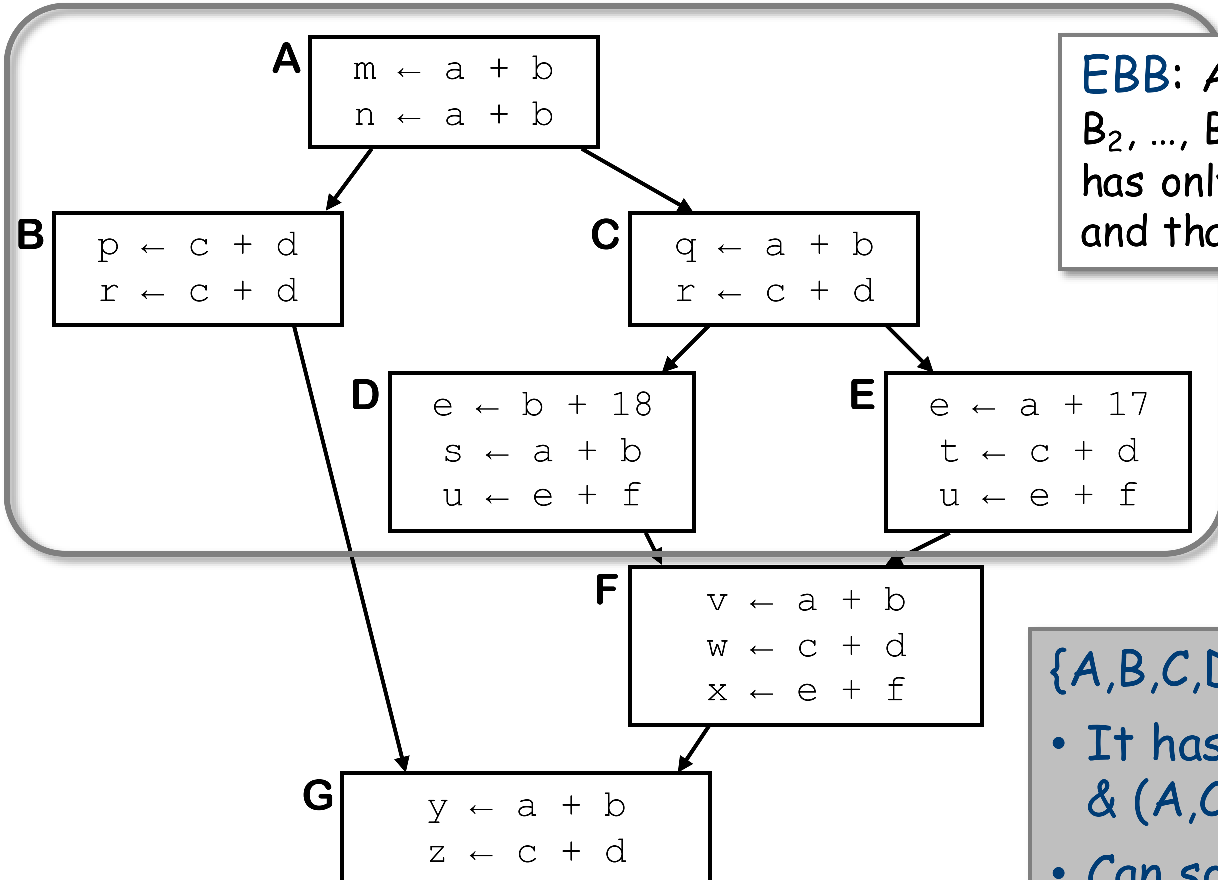
We would like to believe optimization developed in an orderly fashion

- Local methods led to regional methods
- Regional methods led to global methods
- Global methods led to interprocedural methods

It did not happen that way

- First compiler, FORTRAN, used both local & global methods
- Development has been scattershot & concurrent
- Scope appears to relate to the inefficiency being attacked, rather than the refinement of the inventor.

# Superlocal Value Numbering




**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.

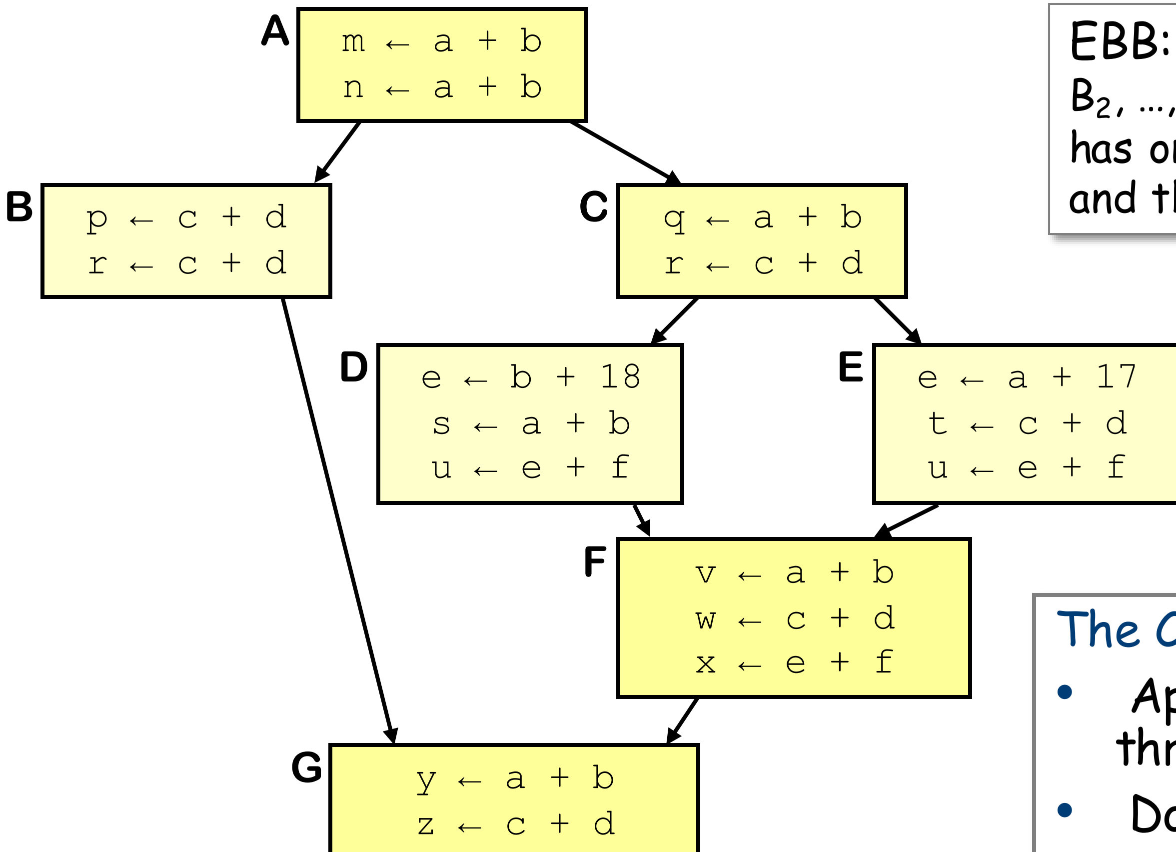
{A,B,C,D,E} is an EBB

- 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"

# Superlocal Value Numbering




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.

**The Concept**

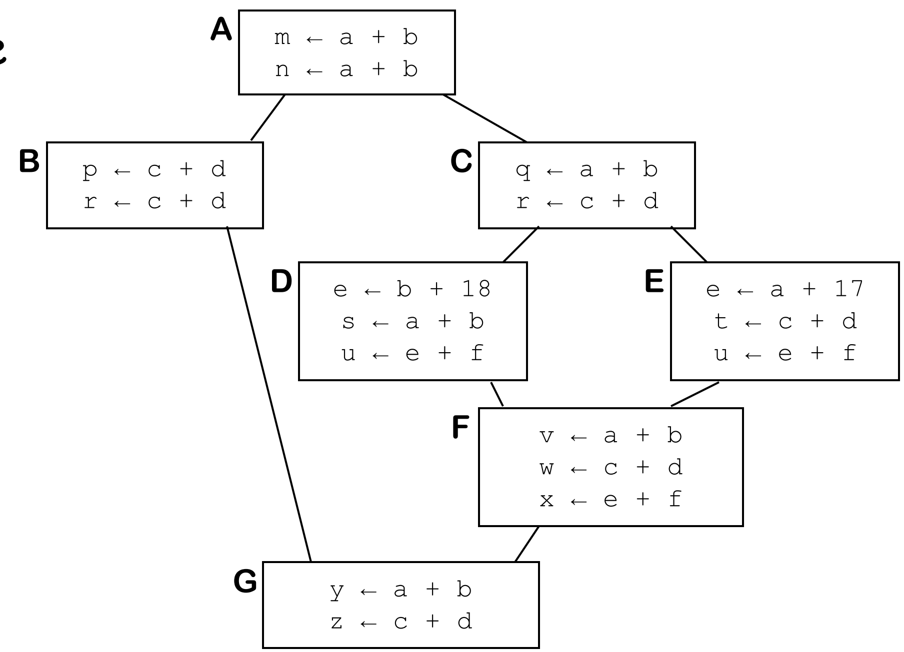
- Apply local method to paths through the EBBs
- Do {A,B}, {A,C,D}, & {A,C,E}
- Obtain reuse from ancestors
- Avoid re-analyzing A & C
- Does not help with F or G

# Superlocal Value Numbering

## Efficiency

- Use A's table to initialize tables for B & C
- To avoid duplication, use a scoped hash table
  - A, AB, A, AC, ACD, AC, ACE, F, G
- Need a VN → name mapping to handle kills
  - Must restore map with scope
  - Adds complication, not cost

"kill" is a re-definition of some name

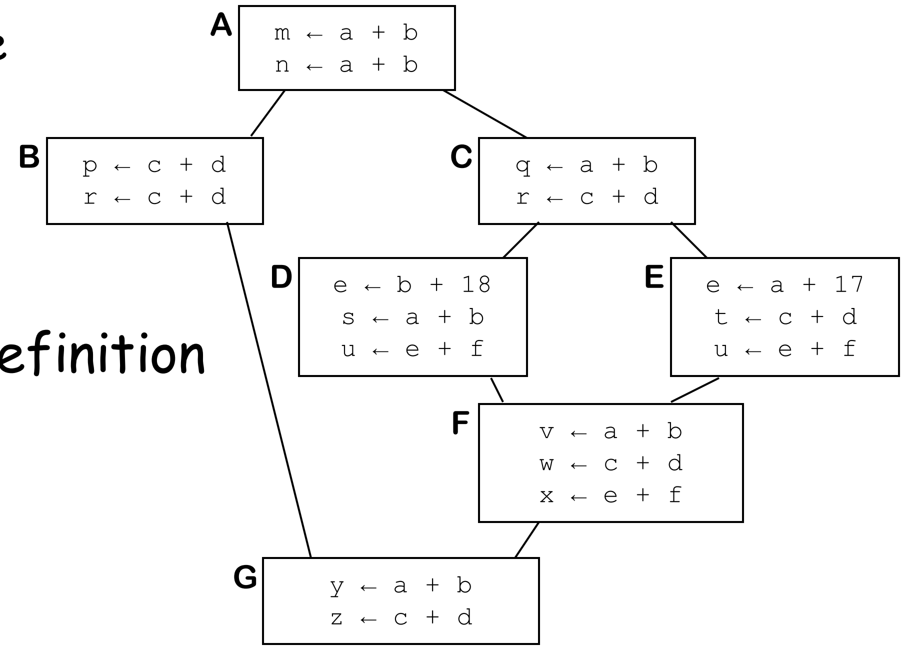


# Superlocal Value Numbering

## Efficiency

- Use A's table to initialize tables for B & C
- To avoid duplication, use a scoped hash table
  - A, AB, A, AC, ACD, AC, ACE, F, G
- Need a VN → name mapping to handle kills
  - Must restore map with scope
  - Adds complication, not cost

"kill" is a re-definition of some name




## To simplify matters

- Need *unique name* for each definition
- Makes name → VN
- Use the SSA name space

The subscripted names from the earlier example are an instance of the SSA name space.

# SSA Name Space (locally)

Example (from earlier):

| 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

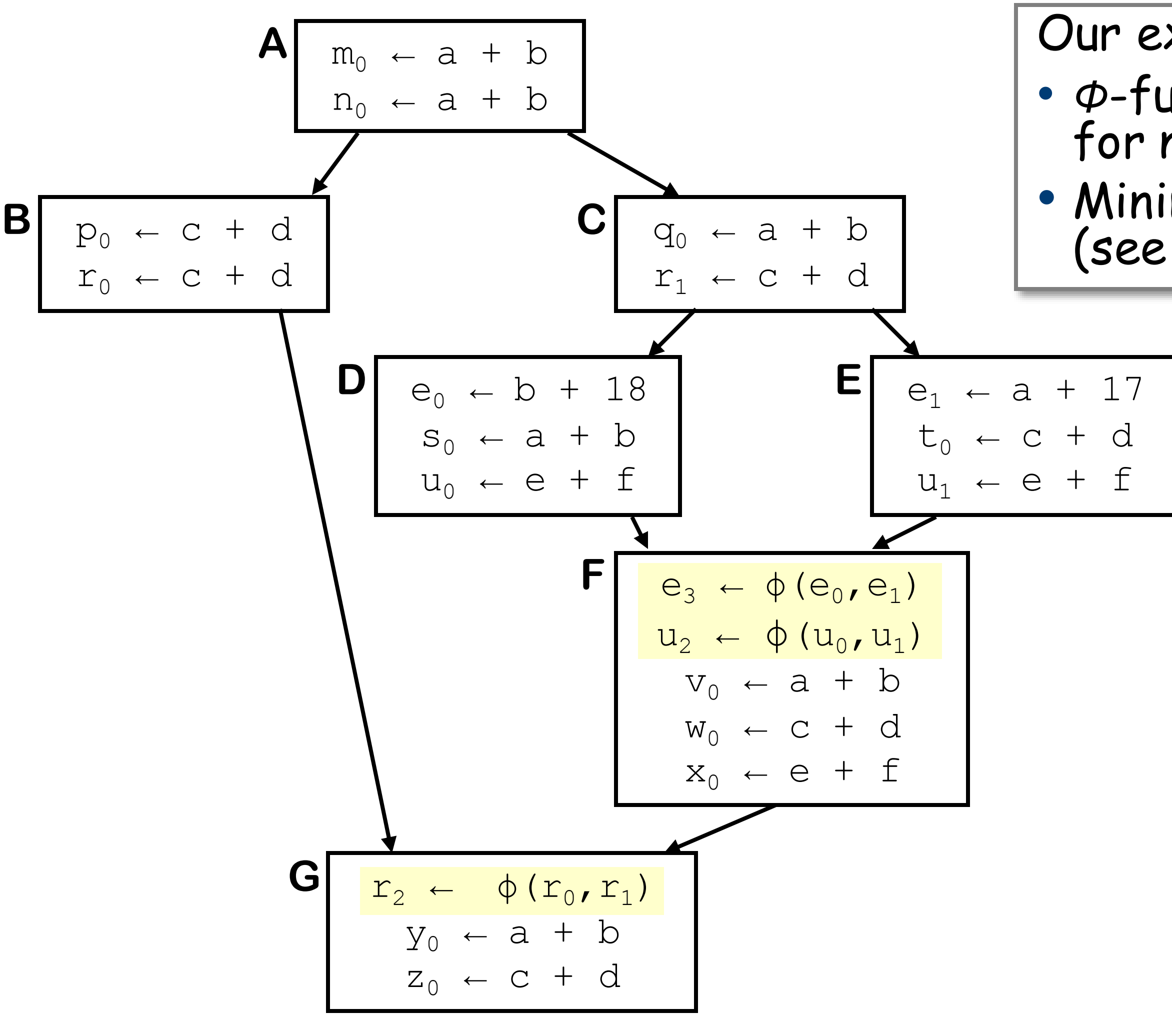
Result:
- $a_0^3$ is available
- Rewriting just works

# SSA Name Space (in general)

Two principles
- Each name is defined by exactly one operation
- Each operand refers to exactly one definition

To reconcile these principles with real code
- Insert ϕ-functions at merge points to reconcile name space
- Add subscripts to variable names for uniqueness

# Superlocal Value Numbering




Our example in SSA form

- $\phi$-functions at join points for names that need them
- Minimal set of $\phi$-functions (see Chapter 9 in EaC)

# Superlocal Value Numbering

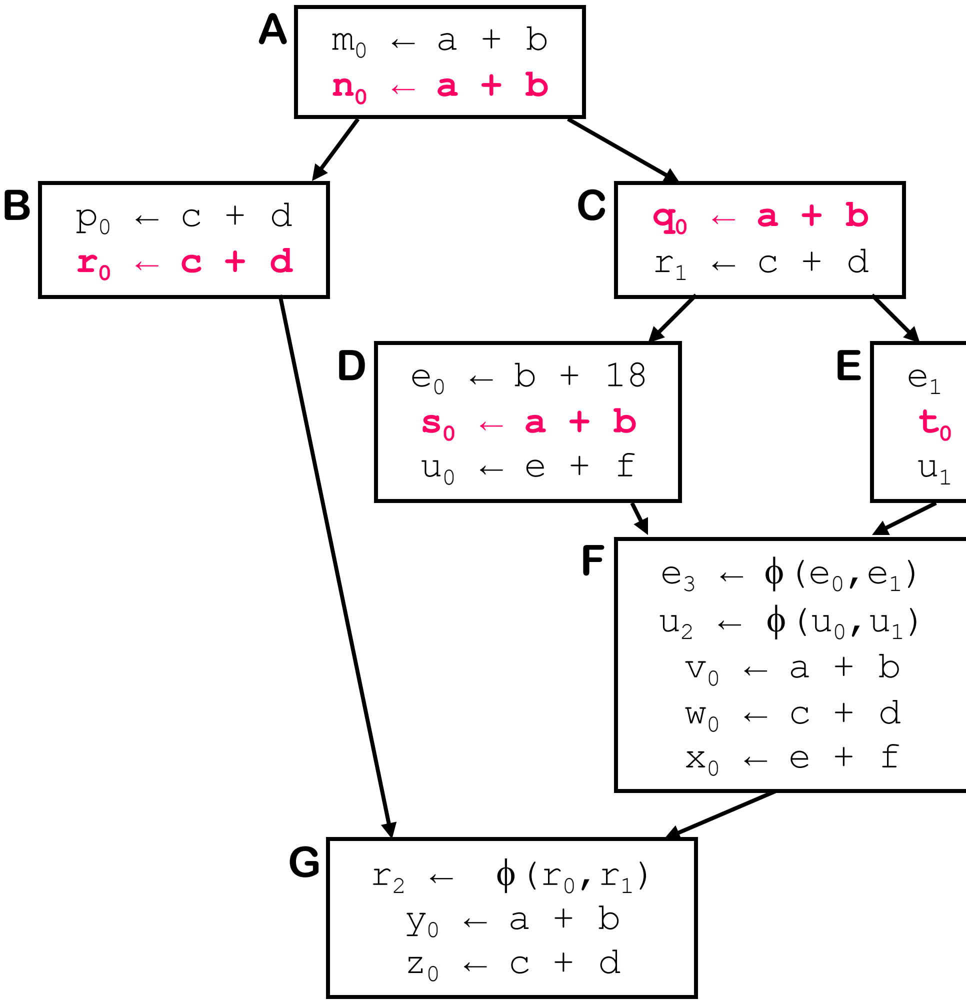
## The SVN Algorithm

```
WorkList ← { entry block }                                        Blocks to process
Empty ← new table                                                 Table for base case
while (WorkList is not empty)
        remove a block b from WorkList
        SVN(b, Empty)

SVN( Block, Table)
        t ← new table for Block, with Table linked as surrounding scope
        LVN( Block, t)                                            Use LVN for the work
        for each successor s of Block
    if s has just 1 predecessor                                   In the same EBB
                then SVN( s, t )
          else if s has not been processed                        Starts a new EBB
                then add s to WorkList
        deallocate t
```

Assumes LVN has been parameterized around block and table

# Superlocal Value Numbering




With all the bells & whistles

- Find more redundancy
- Pay minimal extra cost
- Still does nothing for F & G

Superlocal techniques

- Some local methods extend cleanly to superlocal scopes
- VN does not back up
- Backward motion causes probs

# Loop Unrolling

## Applications spend a lot of time in loops

- We can reduce loop overhead by unrolling the loop

```
do i = 1 to 100 by 1
        a(i) ← b(i) * c(i)
        end
```

Complete unrolling

```
a(1)    ← b(1) * c(1)
a(2)    ← b(2) * c(2)
a(2)    ← b(3) * c(3)
 …
a(100) ← b(100) * c(100)
```

- Eliminated additions, tests, and branches
  - *Can subject resulting code to strong local optimization!*
- Only works with fixed loop bounds & few iterations
- The principle, however, is sound
- Unrolling is always safe, as long as we get the bounds right

# Loop Unrolling

## Unrolling by smaller factors can achieve much of the benefit

Example: unroll by 4

```
do i = 1 to 100 by 1
        a(i) ← b(i) * c(i)
        end
```

Unroll by 4

```
do i = 1 to 100 by 4
    a(i)   ← b(i) * c(i)
    a(i+1) ← b(i+1) * c(i+1)
    a(i+2) ← b(i+2) * c(i+2)
    a(i+3) ← b(i+3) * c(i+3)
    end
```

Achieves much of the savings with lower code growth

- Reduces tests & branches by 25%
- LVN will eliminate duplicate adds and redundant expressions
- Less overhead per useful operation

But, it relied on knowledge of the loop bounds…

# Loop Unrolling

## Unrolling with unknown bounds

Need to generate guard loops

```
do i = 1 to n by 1
        a(i) ← b(i) * c(i)
        end
```

Unroll by 4

```
i ← 1
do while (i+3 < n )
    a(i)   ← b(i) * c(i)
    a(i+1) ← b(i+1) * c(i+1)
    a(i+2) ← b(i+2) * c(i+2)
    a(i+3) ← b(i+3) * c(i+3)
    i ← i + 4
    end
do while (i < n)
    a(i)   ← b(i) * c(i)
    i ← i + 1
    end
```

Achieves most of the savings

- Reduces tests & branches by 25%
- LVN still works on loop body
- Guard loop takes some space

Can generalize to arbitrary upper & lower bounds, unroll factors

# Loop Unrolling

## One other unrolling trick

Eliminate copies at the end of a loop

```
t1 ← b(0)
do i = 1 to 100 by 1
    t2 ← b(i)
    a(i) ← a(i) + t1 + t2
    t1 ← t2
end
```

Unroll and rename

```
t1 ← b(0)
do i = 1 to 100 by 2
    t2     ← b(i)
    a(i)   ← a(i) + t1 + t2
    t1     ← b(i+1)
    a(i+1) ← a(i+1) + t2 + t1
end
```

Unroll by LCM of copy-cycle lengths

- Eliminates the copies, which were a naming artifact
- Achieves some of the benefits of unrolling
  - Lower overhead, longer blocks for local optimization
- Situation occurs in more cases than you might suspect

This result has been rediscovered many times. [Kennedy's thesis]