COMP 512
Rice University
Spring 2015

Overview Of Optimization, 2

Superlocal Value Numbering, GCSE

Citation numbers, when given, refer to entries in the EaC2e bibliography.

Local Value Numbering

Review

The algorithm

For each operation *o* in the block

1 Get value numbers for the operands from a 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 & use a "load immediate"

If hashing behaves, the algorithm runs in linear time

- ♦ If you don't believe in hashing, try multi-set discrimination

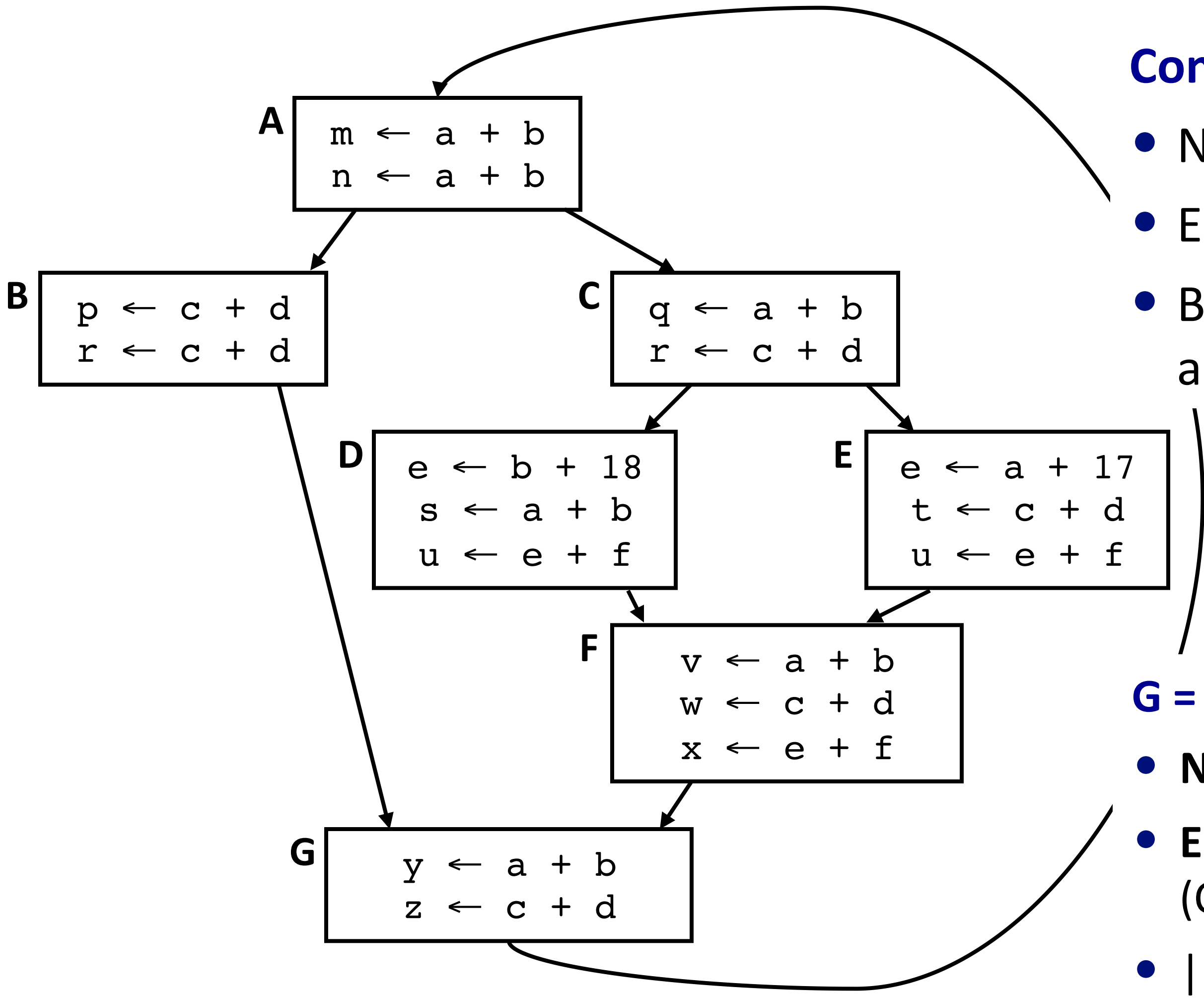
Minor issues

- Commutative operator ⇒ hash operands in each order *or* sort the operands by VN before hashing (*either works, sorting is cheaper*)
- Looks at operand's value number, not its name

EaC2e: digression on page 256 or reference [65]

A Multi-Block Example

Review



Control-flow graph (CFG)

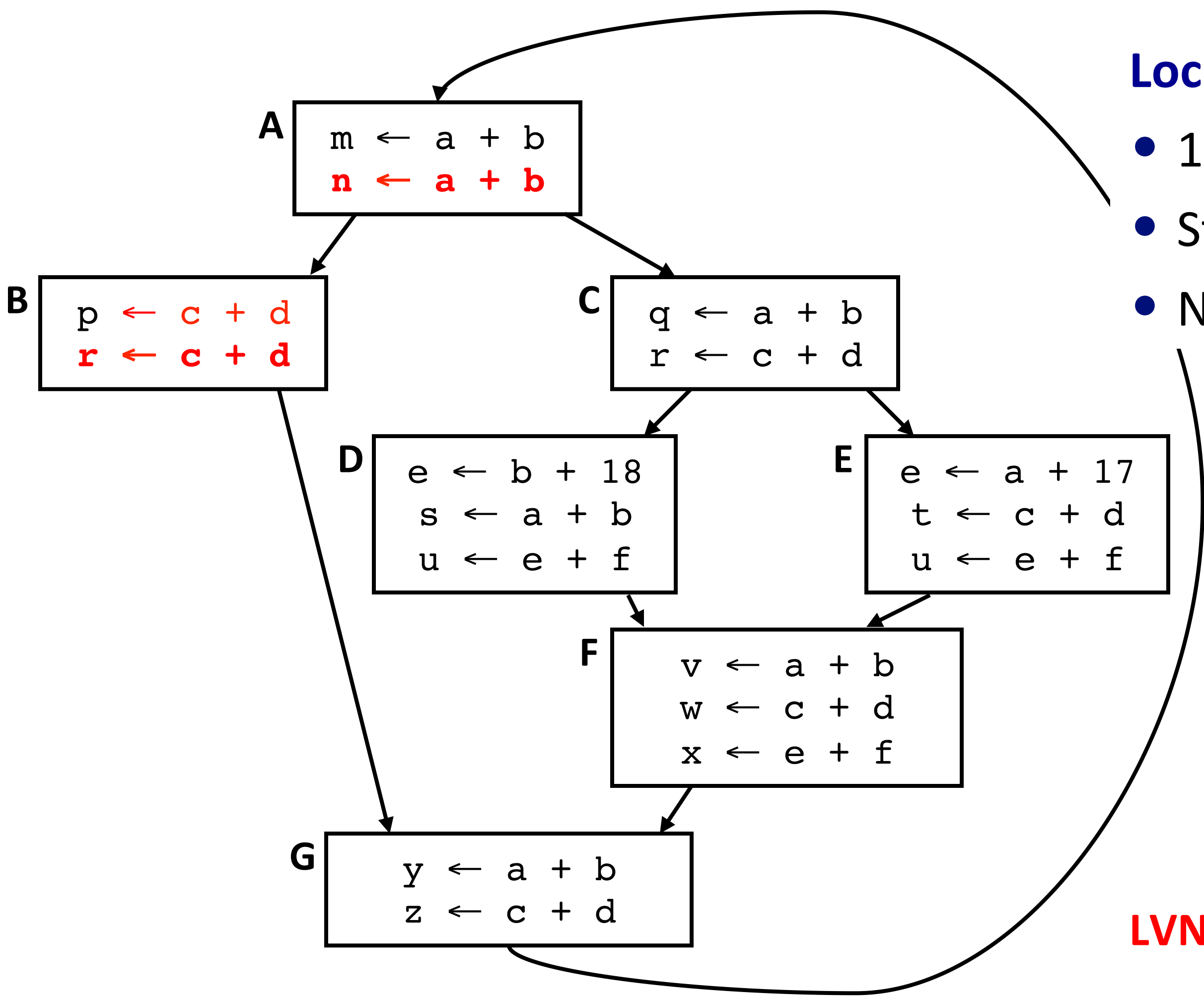
- Nodes for basic blocks
- Edges for branches
- Basis for much of program analysis & transformation

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

A Multi-Block Example

Review



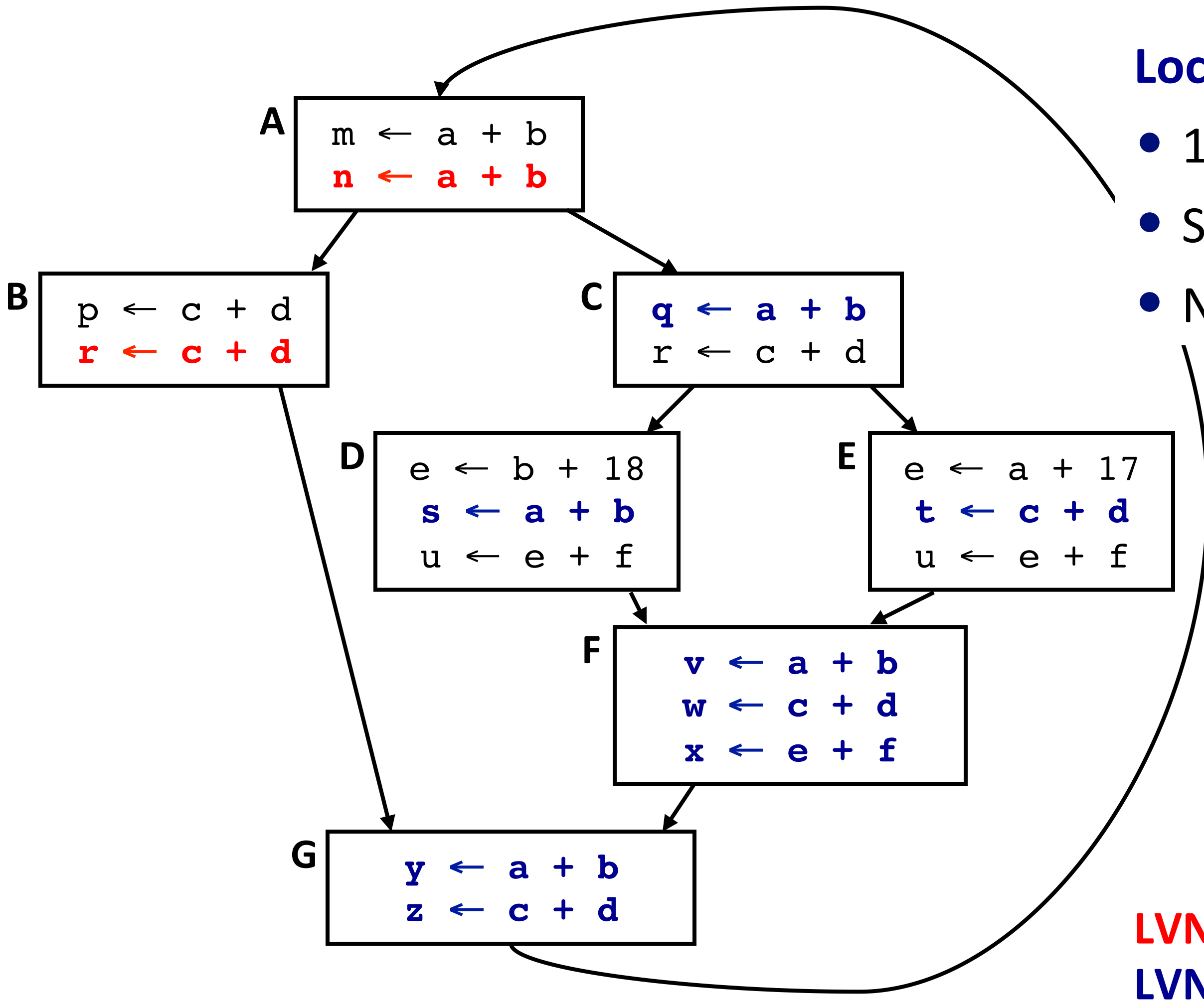
Local Value Numbering (LVN)

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

LVN finds redundant ops in red

A Multi-Block Example

Review



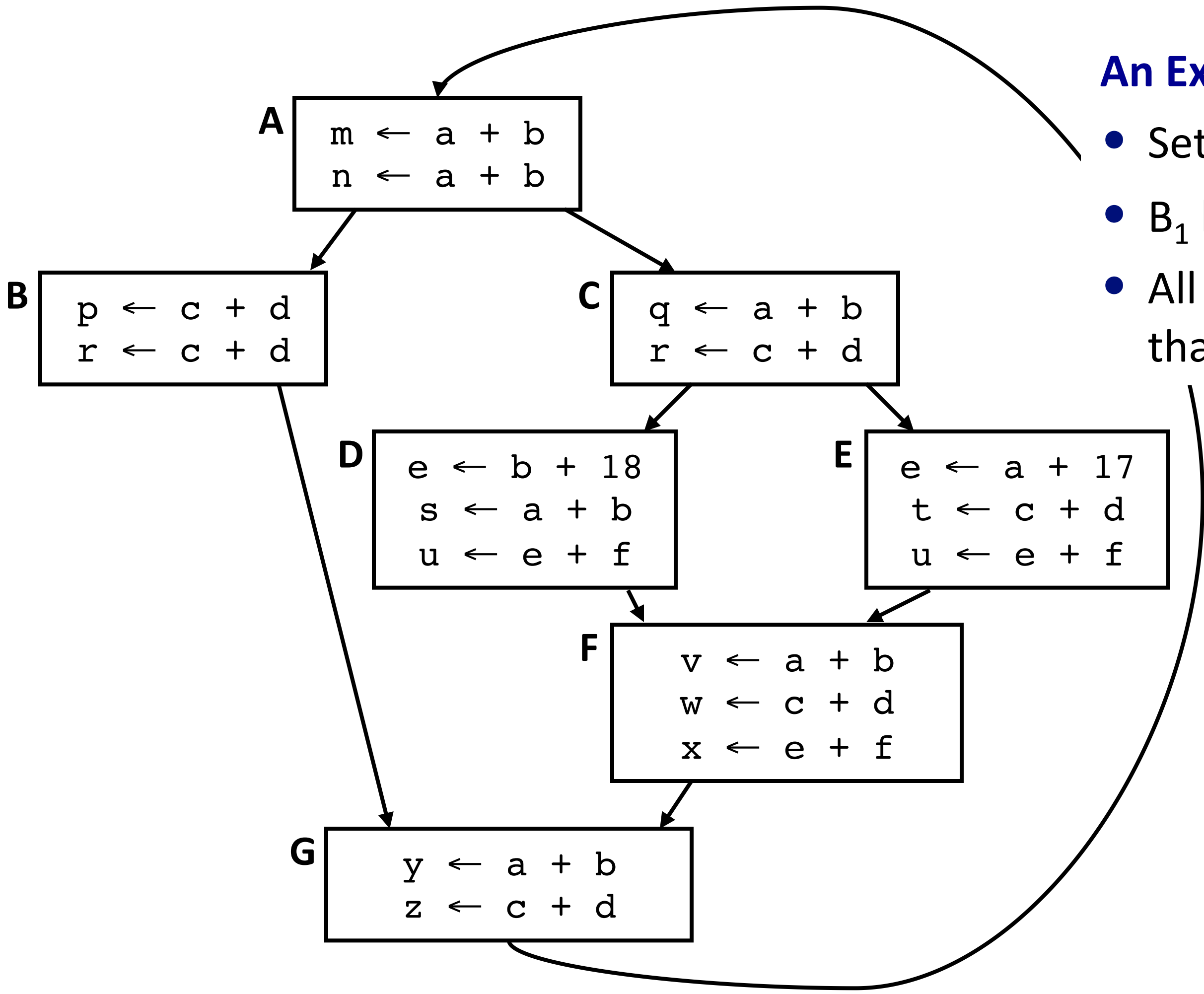
Local Value Numbering (LVN)

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

LVN finds redundant ops in red
LVN misses redundant ops in blue

Beyond Basic Blocks: Extended Basic Blocks

Review

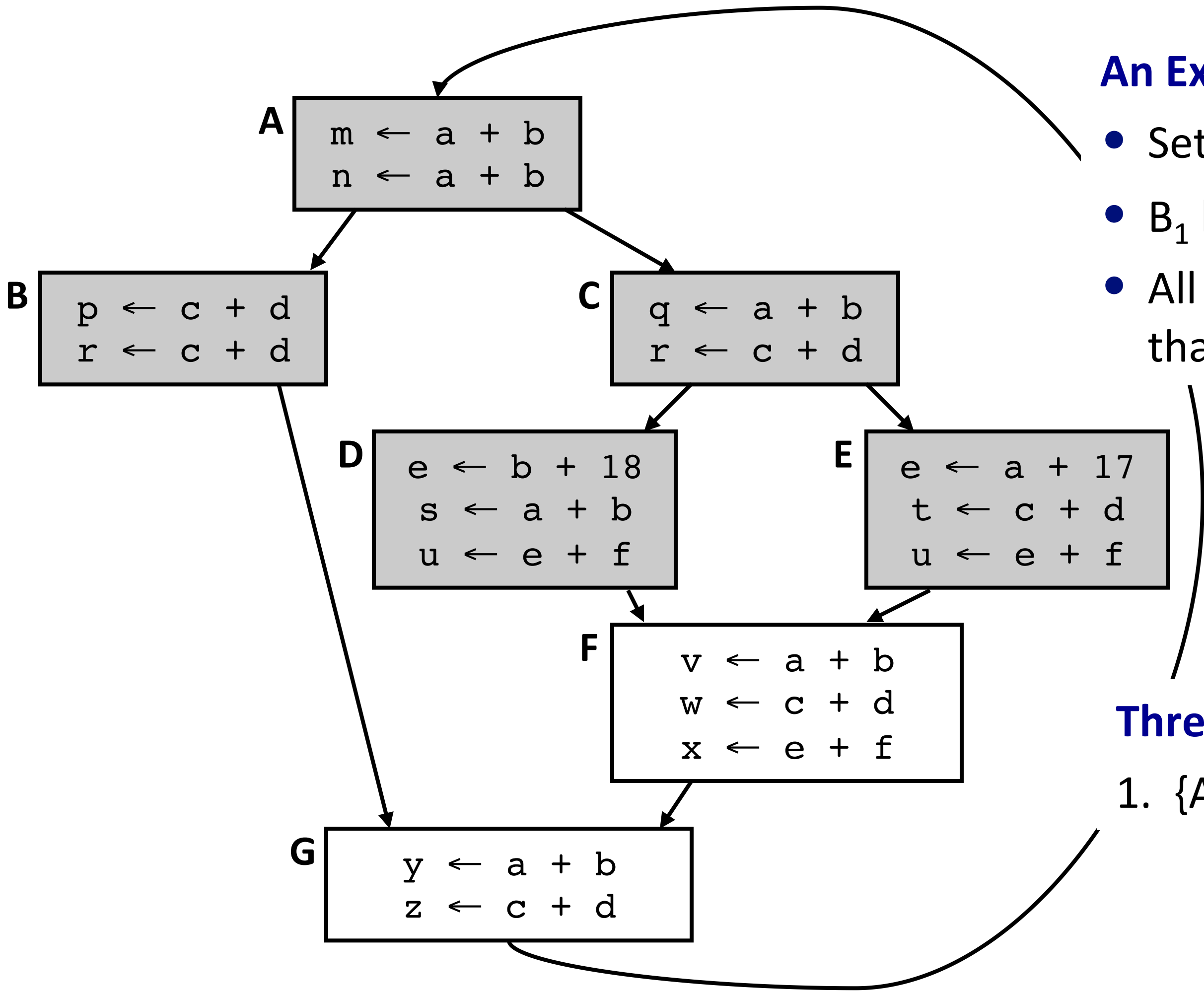


An Extended Basic Block (EBB)

- Set of blocks $B_1, B_2, \ldots, B_n$
- B_1 has > 1 predecessor
- All other B_i have 1 pred. & that pred. is in the **EBB**

Extended Basic Blocks

Review



An Extended Basic Block (EBB)

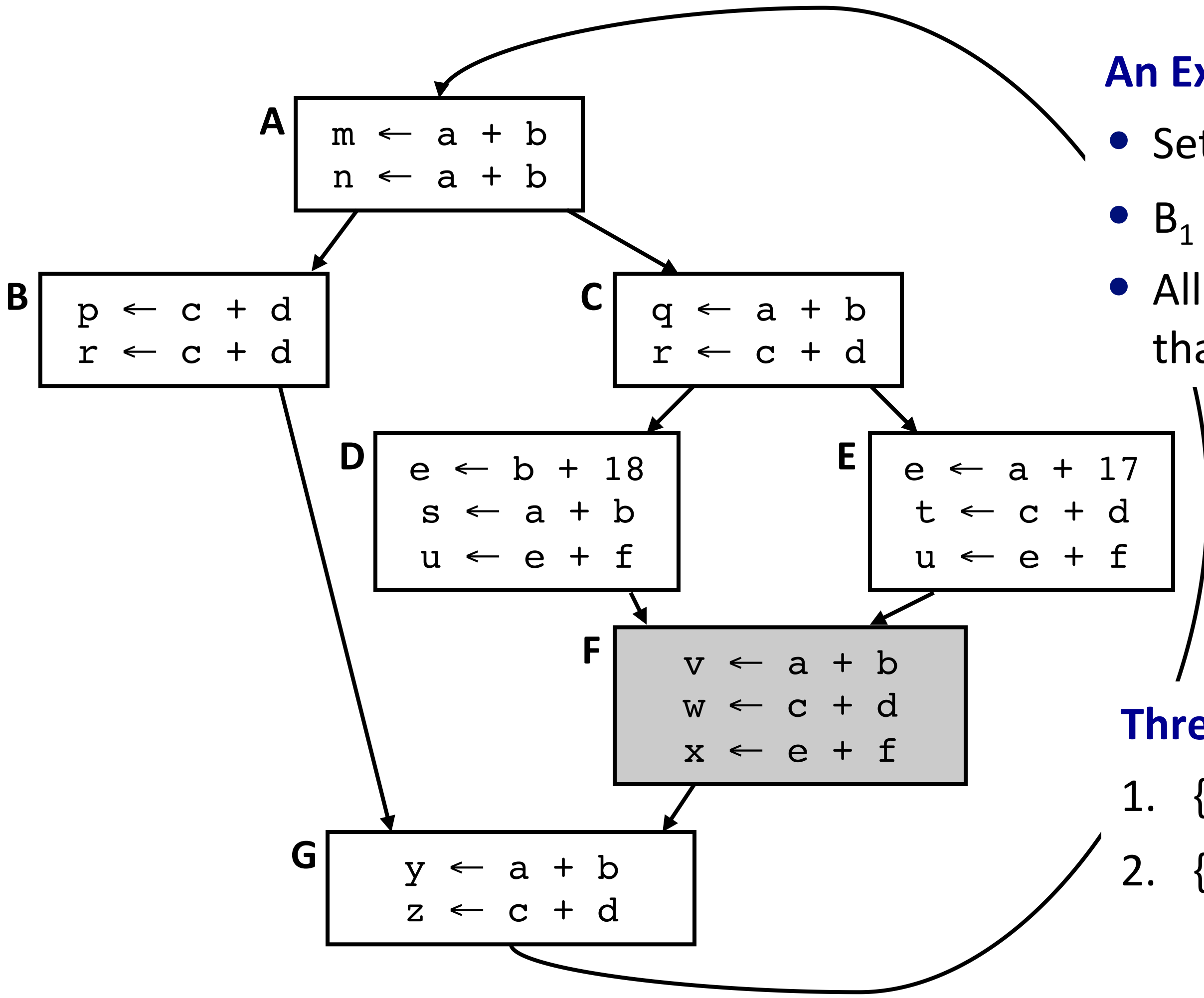
- Set of blocks B_1, B_2, …, B_n
- B_1 has > 1 predecessor
- All other B_i have 1 pred. & that pred. is in the **EBB**

Three EBBs in this CFG

1. {A, B, C, D, E}

Extended Basic Blocks

Review



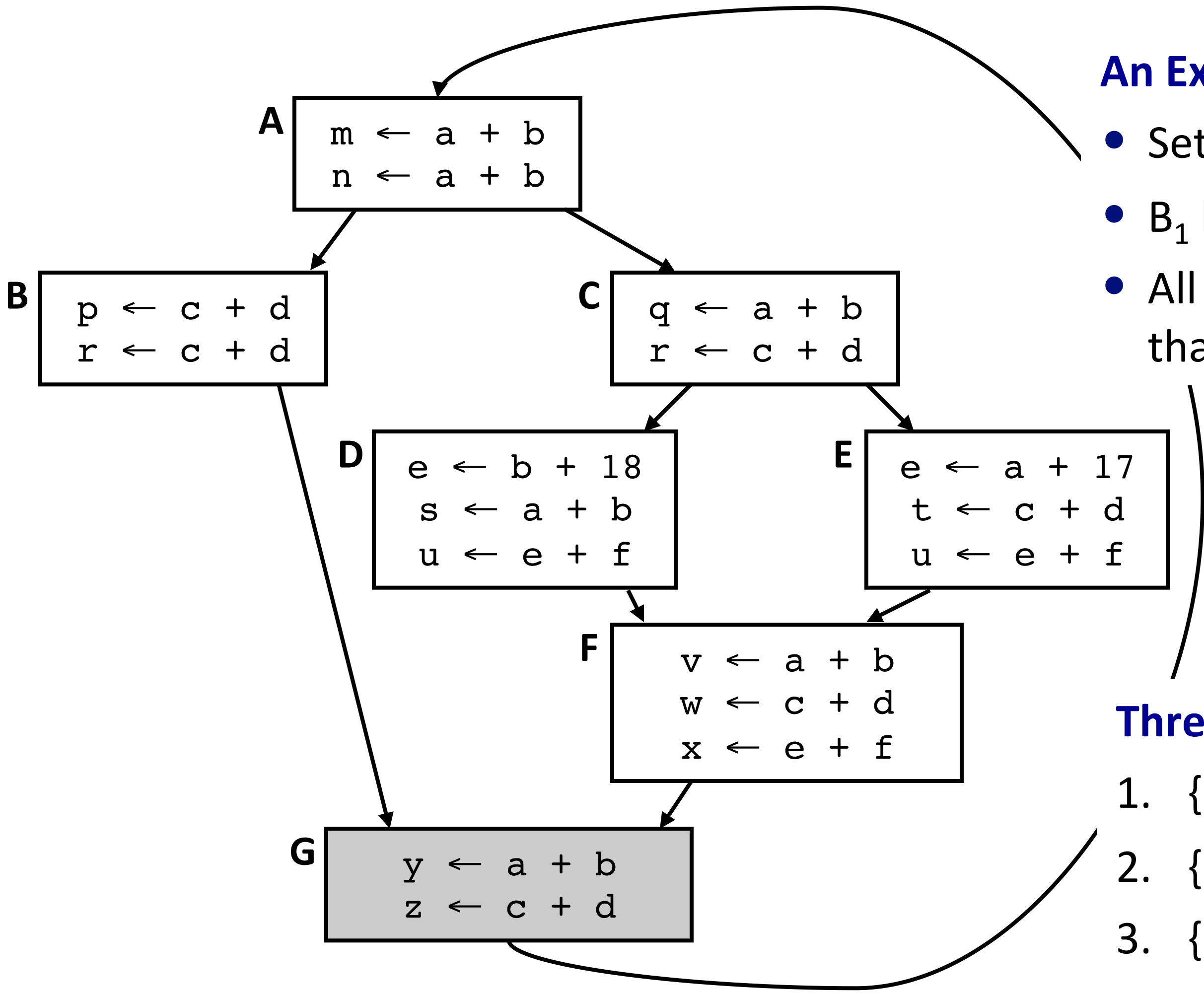
An Extended Basic Block (EBB)

- Set of blocks $B_1, B_2, \ldots, B_n$
- B_1 has > 1 predecessor
- All other B_i have 1 pred. & that pred. is in the **EBB**

Three EBBs in this CFG

1. { A, B, C, D, E }
2. { F }

Extended Basic Blocks



An Extended Basic Block (EBB)

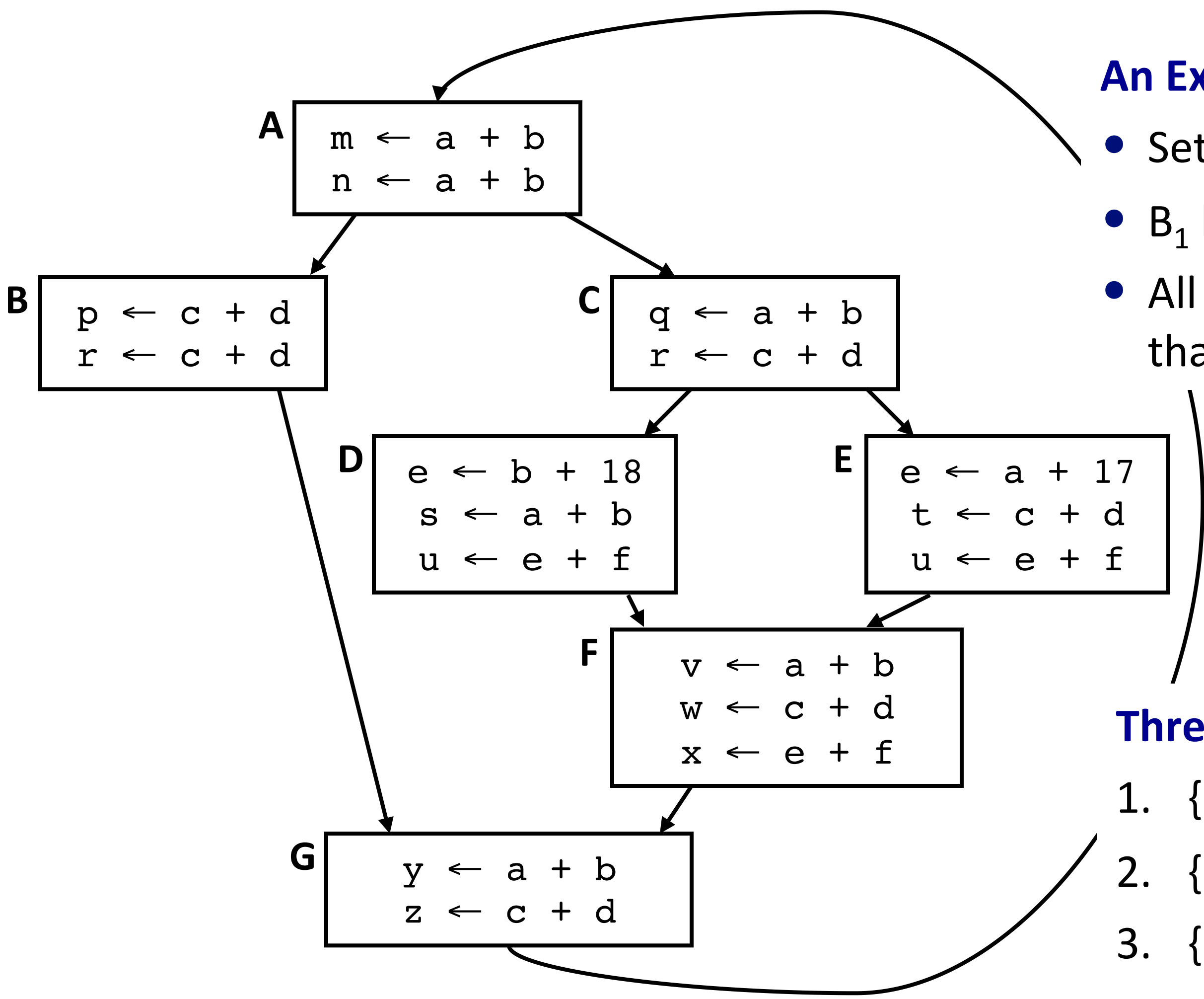
- Set of blocks $B_1, B_2, \ldots, B_n$
- B_1 has > 1 predecessor
- All other B_i have 1 pred. & that pred. is in the **EBB**

Three EBBs in this CFG

1. { A, B, C, D, E }
2. { F }
3. { G }

Extended Basic Blocks

Review



An Extended Basic Block (EBB)

- Set of blocks $B_1, B_2, \ldots, B_n$
- B_1 has > 1 predecessor
- All other B_i have 1 pred. & that pred. is in the **EBB**

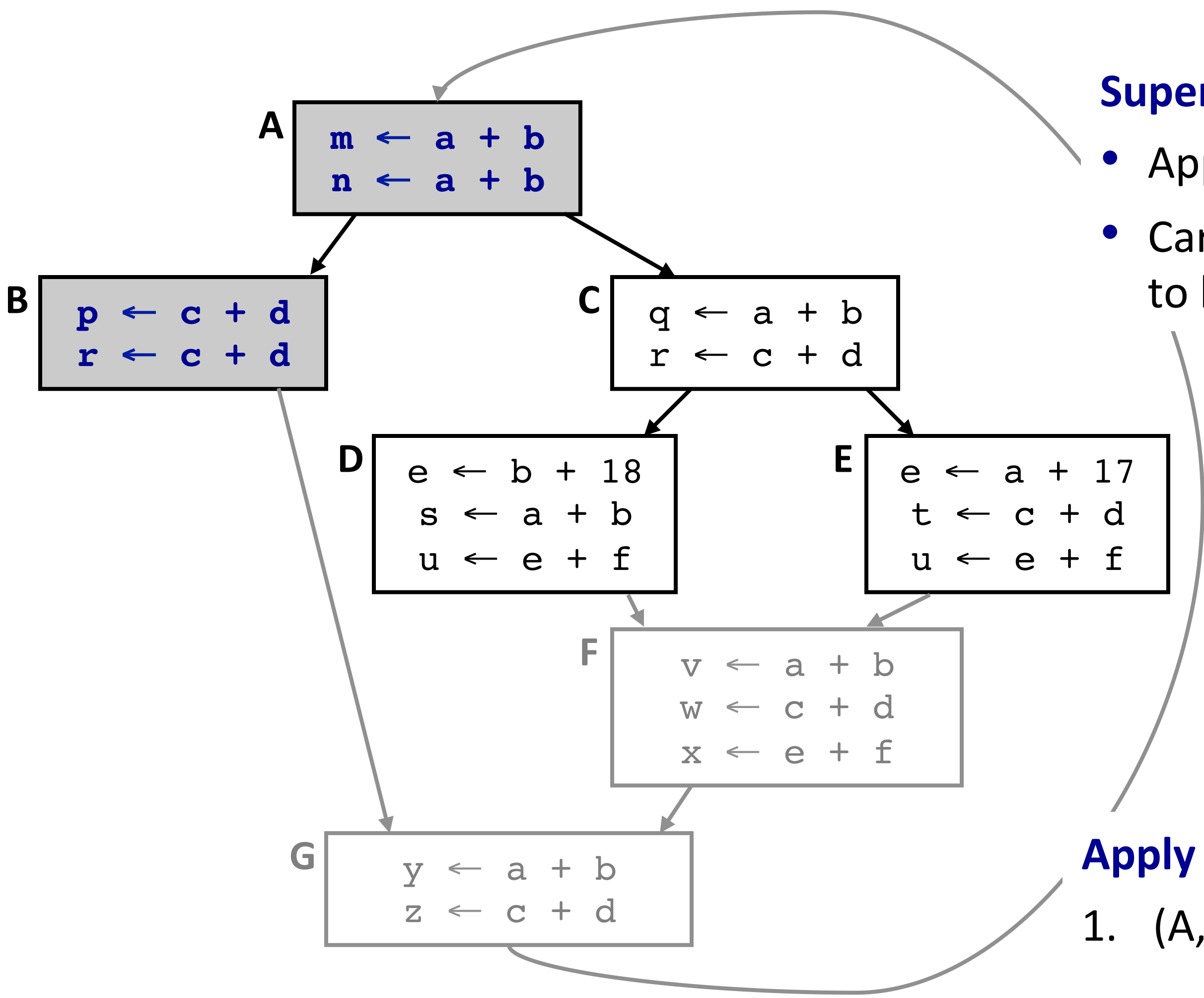
Three EBBs in this CFG

1. { A, B, C, D, E }
2. { F }
3. { G }

Degenerate or trivial EBBs (applies to 2 and 3)

Value Numbering Over Extended Basic Blocks

Review



Superlocal VN (SVN)

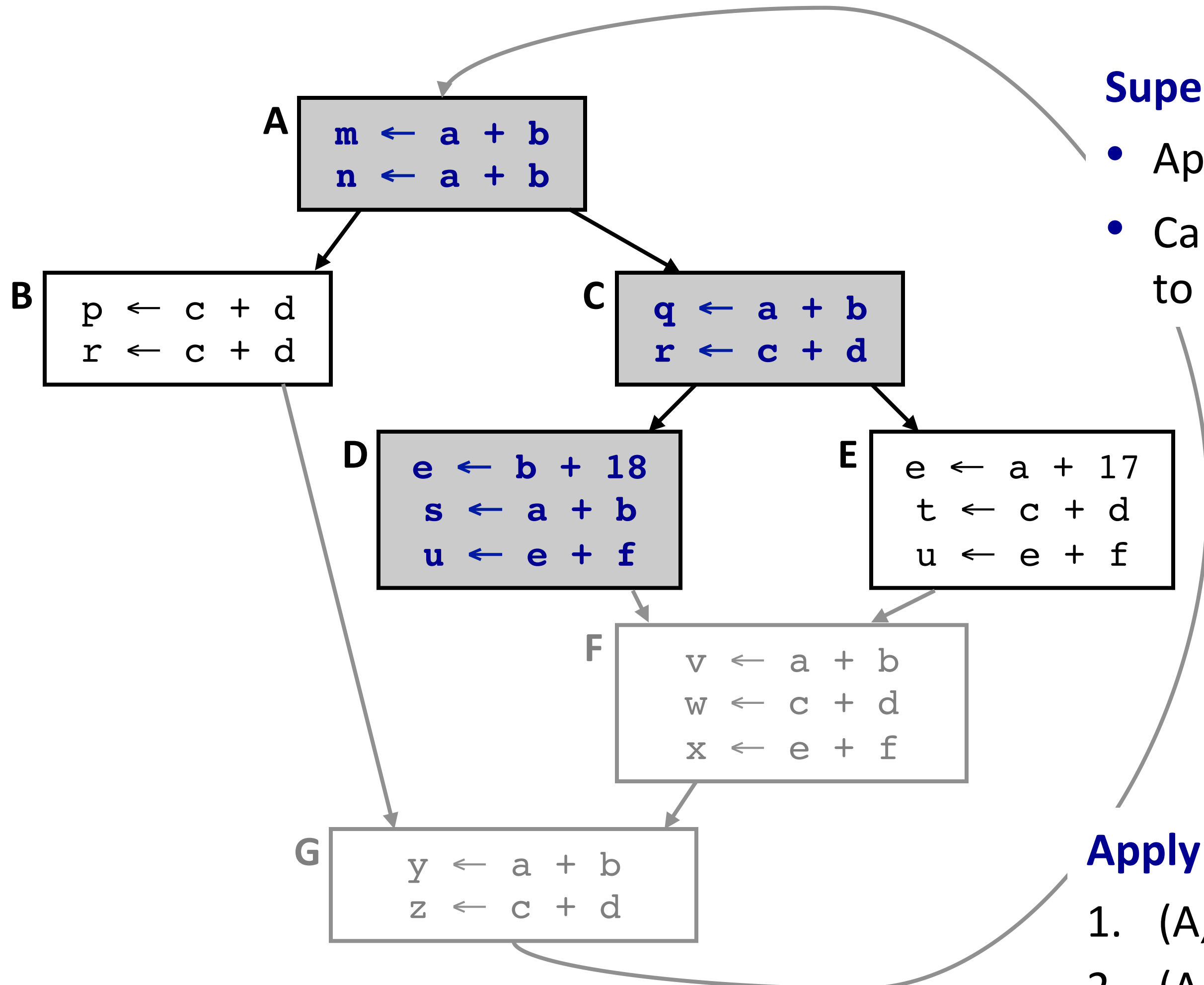
- Apply **LVN** to each path in EBB
- Carry hash table forward, block to block

Apply LVN to each path in EBB

1. (A, B)

Value Numbering Over Extended Basic Blocks

Review



Superlocal VN

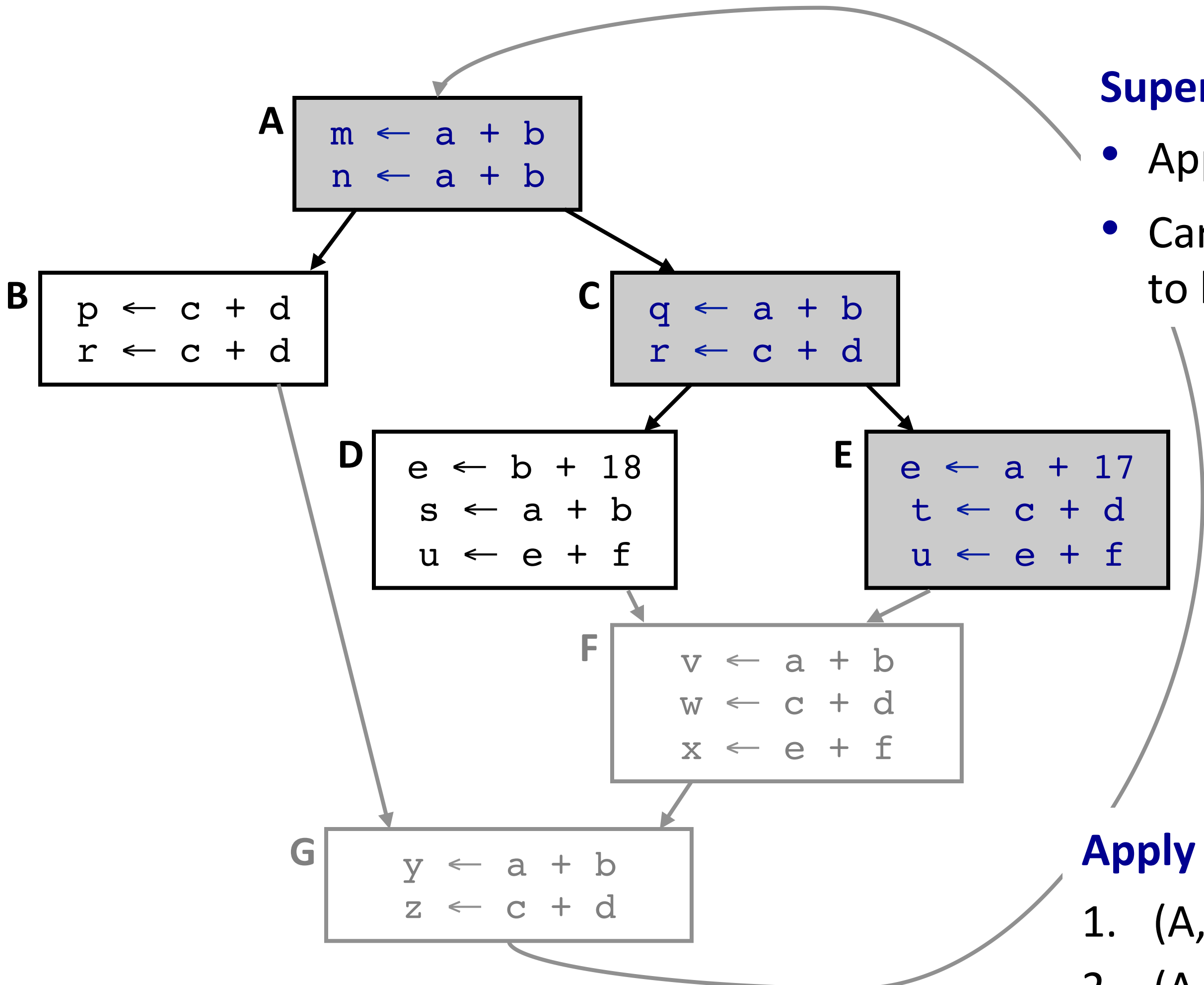
- Apply **LVN** to each path in EBB
- Carry hash table forward, block to block

Apply LVN to each path in EBB

1. (A, B)
2. (A, C, D)

Value Numbering Over Extended Basic Blocks

Review



Superlocal VN

- Apply **LVN** to each path in EBB
- Carry hash table forward, block to block

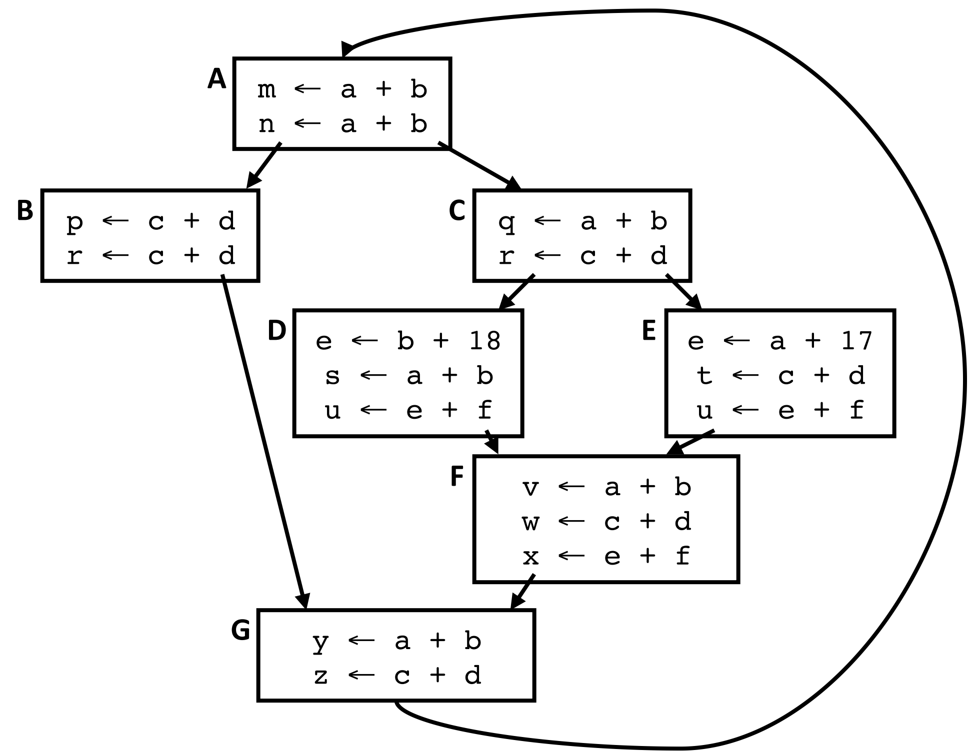
Apply LVN to each path in EBB

1. (A, B)
2. (A, C, D)
3. (A, C, E)

Superlocal Value Numbering

Efficiency

- Easy to implement if we are willing to process A three times & C twice
  - ♦ **A, AB, A, AC, ACD, A, AC, ACE, F, G**
- Could be faster if we reused the results from A & C
  - ♦ **A, AB, AC, ACD, ACE, F, G**



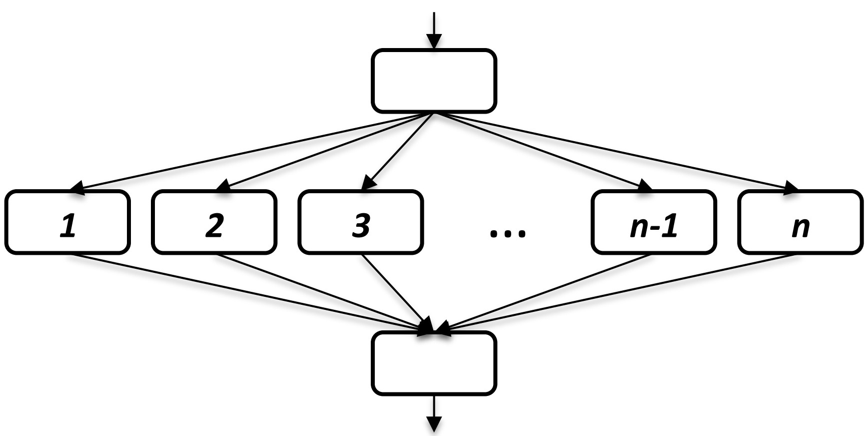
Superlocal Value Numbering

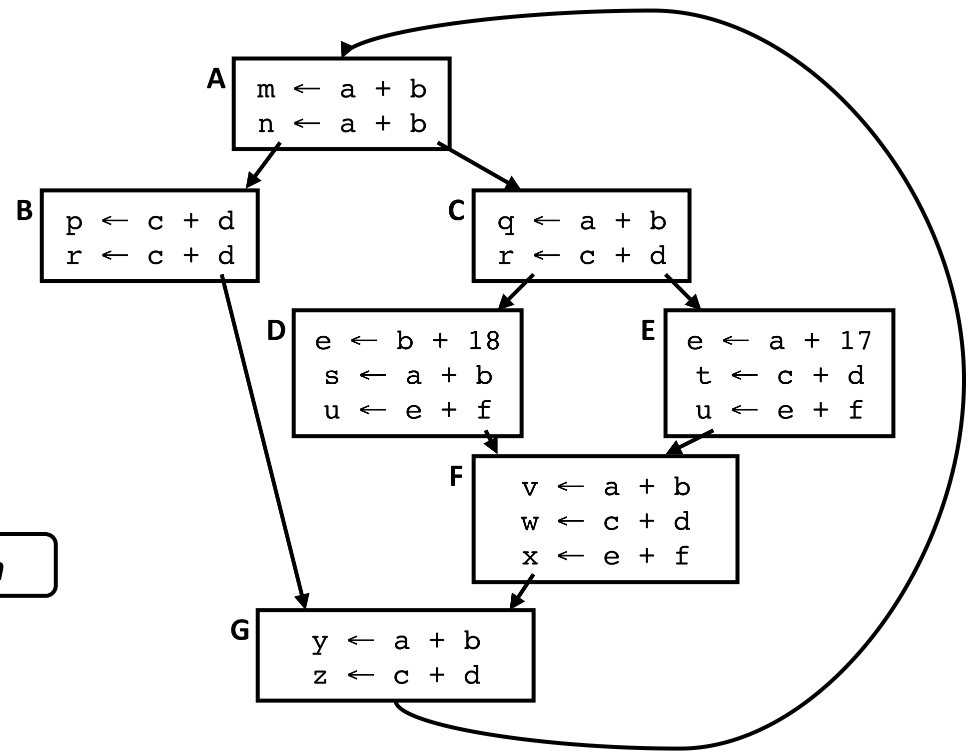
Efficiency

- Easy to implement if we are willing to process A three times & C twice
  - ♦ **A, AB, A, AC, ACD, A, AC, ACE, F, G**
- Could be faster if we reused the results from A & C
  - ♦ **A, AB, AC, ACD, ACE, F, G**

Worst Case

- Imagine **SVN** on a case statement

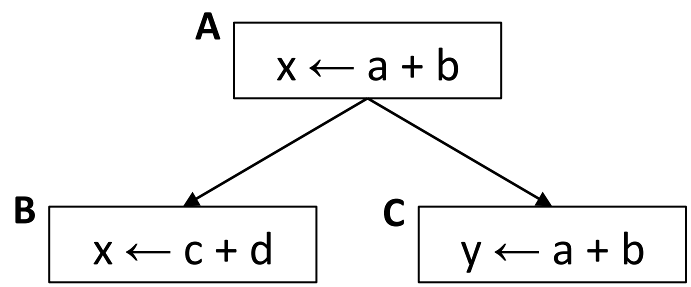




The Role of Names in Superlocal Value Numbering

What work must be repeated in a predecessor block?

- Value numbers are stored in a hash table
  - ♦ Keyed by name or <op,vn,vn> construct
- To avoid repeated work, SVN should roll back changes to the hash table
  - ♦ Rather than A, AB, A, AC we want to go from AB to AC without revisiting A



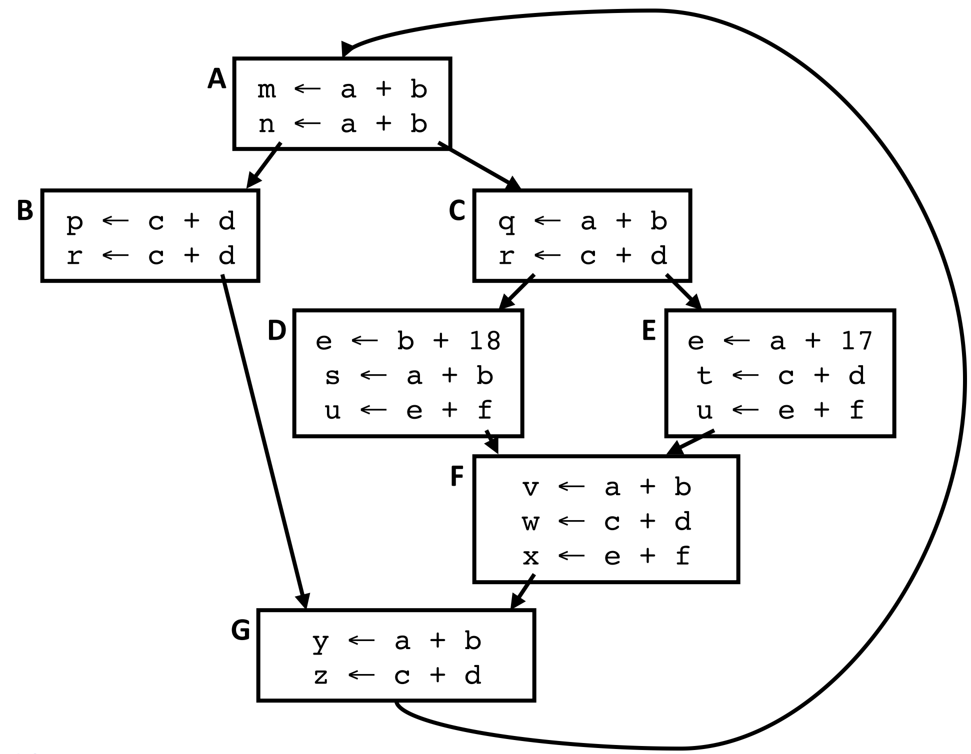
In the example, the definition of x in B changes the hash table entry for x

- After AB, SVN needs to roll x's value number back to the value from A
  - ♦ Could run backward through B and "undo" each definition (*with bookkeeping*)
  - ♦ Could reprocess A
- Better way is to rename so that each definition has a unique name
  - → ***We saw the same issue in LVN, in local register allocation, & in local scheduling.***
- We need a global name space with the right set of properties

Superlocal Value Numbering

Efficiency

- Easy to implement if we are willing to process A three times & C twice
  - **A, AB, A, AC, ACD, A, AC, ACE, F, G**
- Could be faster if we reused the results from A & C
  - **A, AB, AC, ACD, ACE, F, G**
  - Need an appropriate name space & a scoped hash table (*parsing?*)
    - → ***Alternative is to add lots of complex mechanism for kills & table management***



Desired Name Space

- Unique name for each definition
  - Name ⇔ VN
- SSA name space is ideal

Aside: SSA Name Space (In General)

Two principles

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

A ϕ-function selects one of its operands, based on the control-flow path used to reach the block.

To reconcile these principles with real code

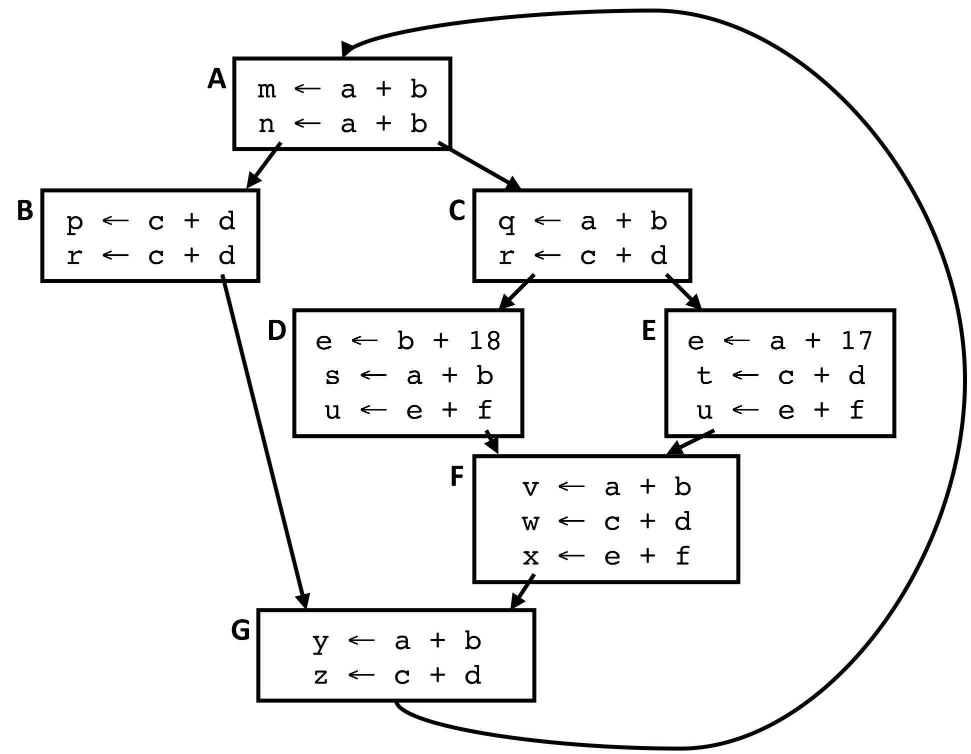
- Insert ϕ-functions at merge points to reconcile name space
- Add subscripts to variable names for uniqueness

We'll look at how to construct SSA form in a week or two

Superlocal Value Numbering

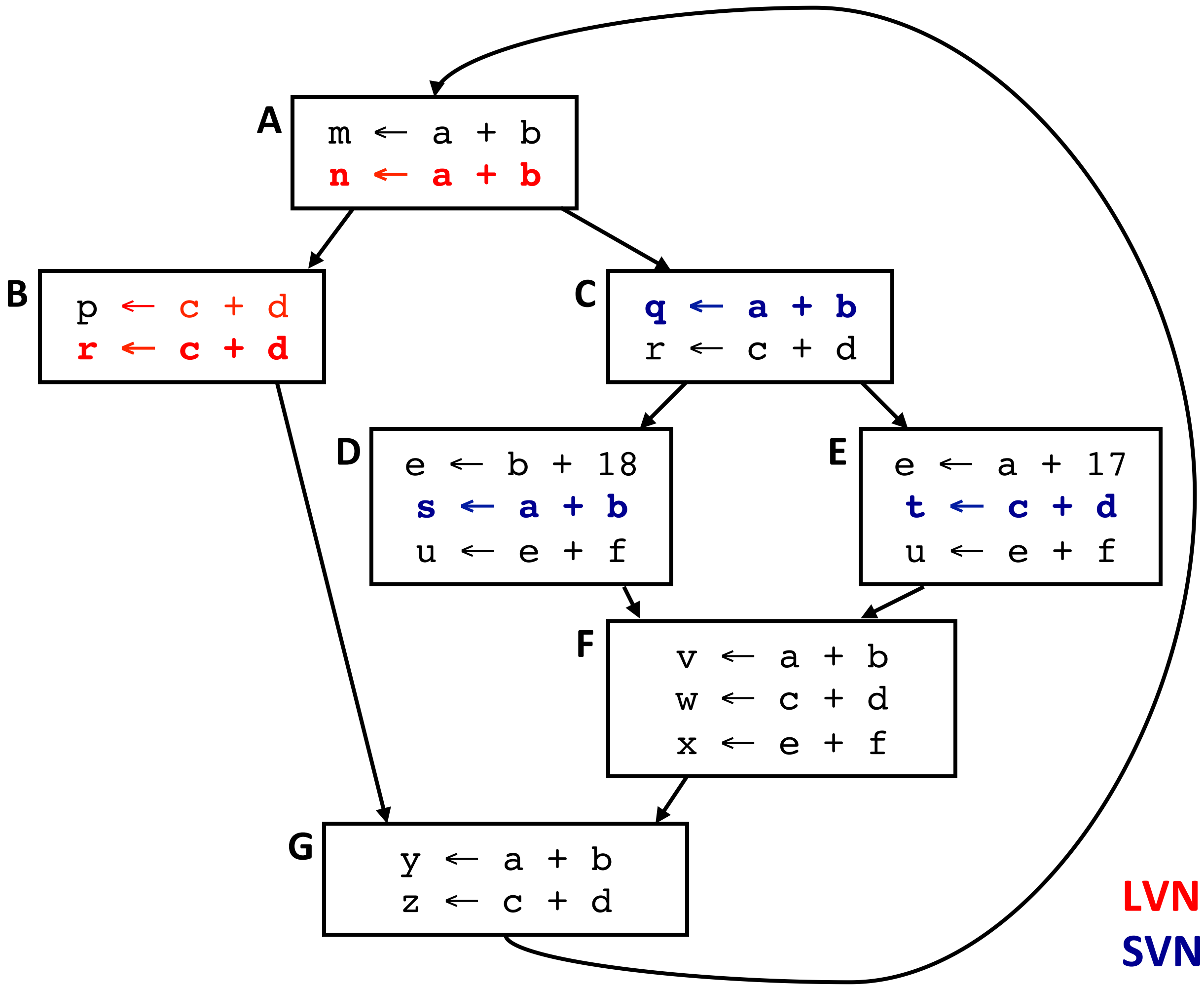
Now, SVN becomes

1. Identify **EBB**s
2. In depth-first order over an **EBB**, starting with the head of the **EBB**, b_0
   a. Apply **LVN** to b_i
   b. Invoke **SVN on** each of b_i's **EBB** successors
      → When going from b_i to its **EBB** successor b_j, extend the symbol table with a new scope for b_j, apply **LVN** to b_j, & process b_j's **EBB** successors
      → When going from b_j to its **EBB** predecessor b_i, discard the scope for b_j



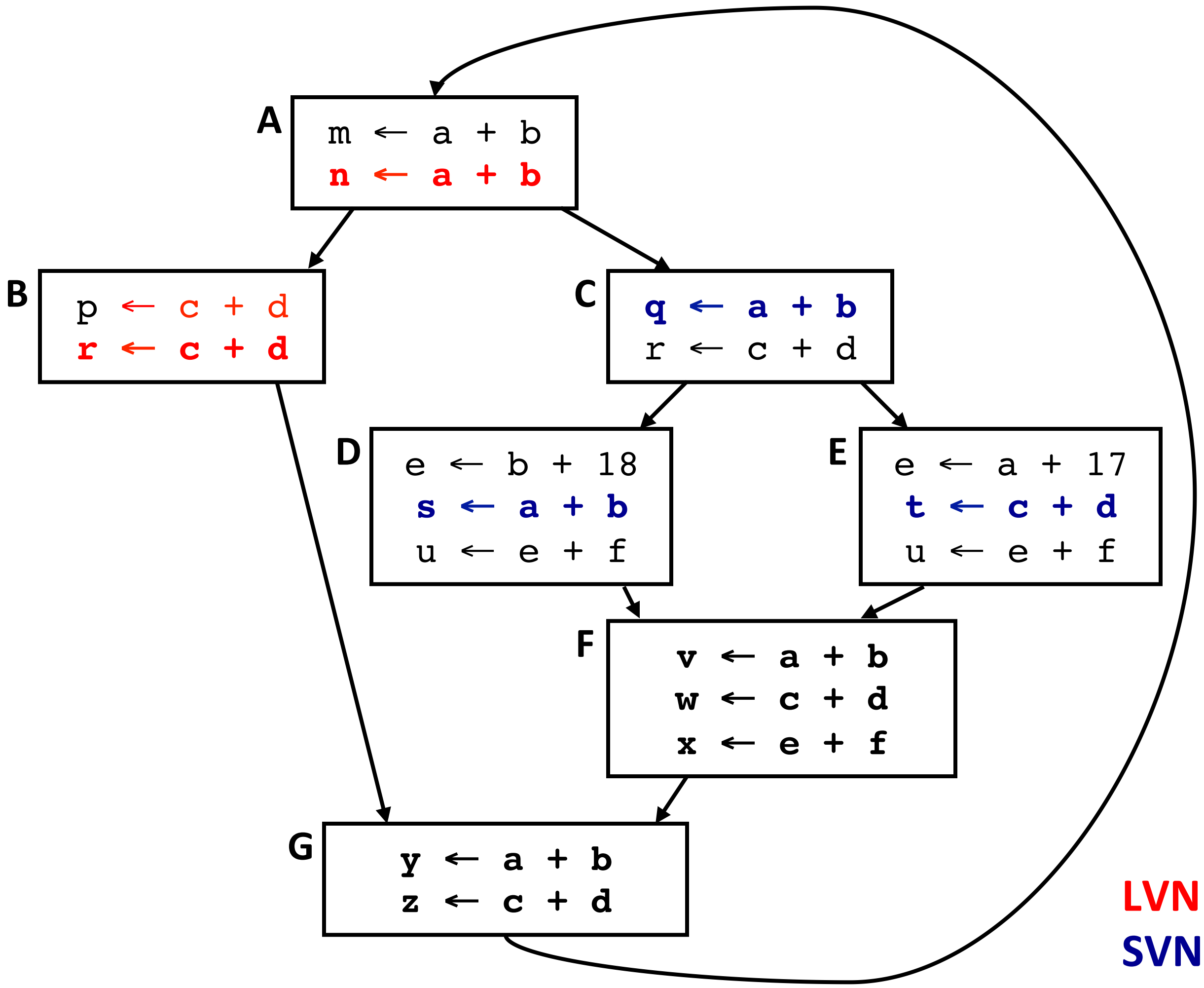
It *is* that easy, with a scoped table & the right name space

SVN on the Example



LVN finds redundant ops in red
SVN finds redundant ops in blue

SVN on the Example



LVN finds redundant ops in red

SVN finds redundant ops in blue

Both miss redundancies in F & G

Perspective

SVN sidesteps the need for separate analysis & transformation

- Applies **LVN** over a larger acyclic context
- Along a path in an **EBB**, order is fully specified
  - ♦ Direct contrast with scheduling in an **EBB** or a trace, because scheduling moves around operations and changes the order
  - ♦ Result, in scheduling, is *compensation code*
  - ♦ Redundancy elimination preserves the order, so we can stretch **LVN** to **EBB**s

To go (much) beyond EBBs, we need separate transformation & analysis

Later in the semester, we will look at methods that combine code motion & redundancy elimination, such as lazy code motion [225,133], and at a technique that applies Hopcroft's partitioning algorithm to expressions over ***SSA*** *names [22].*

⇒ But first, we will look at the classical formulation of *global common subexpression elimination* based on the global data-flow problem: ***available expressions*** [218]

Global Common Subexpression Elimination (GCSE)

The Goal

Find redundant expressions (“*common subexpressions*”) whose range spans multiple basic blocks, ***and*** eliminate any unnecessary re-evaluations

Safety

- Formulate availability of a redundant expression at point *p* as a data-flow problem: ***available expressions*** (annotate each block *b* with a set **AVAIL**(*b*))
  - ♦ If $x \in$ **AVAIL**(*b*), then, along each path from the entry to block *b*, *x* is evaluated and its constituent subexpressions (*i.e., operands*) are not redefined
  - ♦ Evaluating *x* at the start of *b* would produce the same answer as at its most recent evaluation, along any path leading from the entry to *b*
- Transformation preserves the result of prior computations and uses them
  - ♦ Only replaces an evaluation that is in the **AVAIL** set of its block & still available at the point of evaluation
  - ♦ **GCSE** does not move evaluations, it eliminates them

*Safety of **GCSE** hinges on the correctness of the **AVAIL** sets*

This treatment follows Cocke’s classic paper [87].

Global Common Subexpression Elimination

The Goal

Find redundant expressions (“*common subexpressions*”) whose range spans multiple basic blocks, ***and*** eliminate any unnecessary re-evaluations

Profitability

- The transformation does not add any new evaluations to the code
- The transformation replaces the evaluation of the redundant expression with a register-to-register copy from a preserved value
    - ♦ Copy operations are inexpensive
    - ♦ Many copies will coalesce away
- The transformation can increase or decrease demand for registers
    - ♦ If the redundant expression is the last use of one of its operands, it may reduce register pressure
    - ♦ Difficult to understand the impact of any given replacement on register pressure

Available Expressions

For each block *b*

- Let **AVAIL**(*b*) be the set of expressions available on entry to *b*
  - ♦ Initially, **AVAIL**(*n*) = { *all expressions* }, $\forall n \in N$, except n_0
  - ♦ Initially, **AVAIL**(n_0) = Ø
- Let **EXPRKILL**(*b*) be the set of expressions *killed* in *b*
- Let **DEEXPR**(*b*) be the set of expressions defined in *b* and not subsequently killed in *b* (*downward-exposed expressions*)

Now, AVAIL(*b*) can be defined as:

complement operator

$$\textbf{AVAIL}(b) \;=\; \bigcap_{x \in preds(b)} \left(\textbf{DEEXPR}(x) \cup \left(\textbf{AVAIL}(x) \cap \overline{\textbf{EXPRKILL}(x)}\right)\right)$$

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

This system of simultaneous equations forms a data-flow problem

⇒ Solve it with a data-flow algorithm (*e.g., iterative fixed-point scheme*)

Using Available Expressions for GCSE

The Method

1. Build a control-flow graph (**CFG**)
2. ∀ block *b*, compute **DEEXPR**(*b*) and **EXPRKILL**(*b*) & initialize **AVAIL**(*b*)
3. ∀ block *b*, compute **AVAIL**(*b*)
4. ∀ block *b*, replace expressions that are available with references

Expressions killed in *b*

Downward-exposed expressions

Two key issues

- Computing **AVAIL**(*b*) †
- Managing the replacement process

We'll look at the replacement issue first

† Assume, without loss of generality (**wlog**), that we can compute **AVAIL**(*b*) correctly and efficiently for each block *b.*

Replacement in GCSE

The key lies in managing the name space

Need a unique name ∀ *e* ∈ AVAIL(*b*)

1. Can generate them as replacements are done (Fortran H)
2. Can pre-compute a static mapping (Classic answer)
3. Can encode value numbers into names (Simpson)

Strategy

1. This works; it is the classic method
2. Fast; allows single pass to insert code to preserve values of non-redundant evaluations & to replace the redundant evaluations
3. Requires more analysis (VN), but yields more CSEs

Assume solution 2

Global CSE (*replacement step*)

Compute a static mapping from expressions to names

- After analysis & before transformation
  - ♦ ∀ block *b*, ∀ $e \in$ **AVAIL**(*b*), assign a global name to *e*
  - ♦ Integer can be tied to index of bit-vector set representation
- During transformation step
  - ♦ Evaluation of $e \Rightarrow$ insert copy $name(e) \leftarrow e$
  - ♦ Reference to $e \Rightarrow$ replace *e* with *name(e)*

Common strategy:

- Insert copies that might be useful
- Let dead code elim. sort them out

Simplifies design & implementation

The major problem with this approach

- Inserts extraneous copies to preserve values that are of no later use
  - ♦ At all definitions and uses of any $e \in$ **AVAIL**(*b*), ∀ *b*
    - → $e \in$ **AVAIL**(*b*) says nothing about whether or not *e* is ever computed again
  - ♦ Those extra copies are dead and easy to remove
  - ♦ The useful ones often coalesce away

An Aside on Dead Code Elimination

What does "dead" mean?

- Useless code — result is never used
- Unreachable code — code that <u>cannot</u> execute

Both useless code & unreachable are often lumped together as "dead"

To perform Dead Code Elimination

- Must have a global mechanism to recognize usefulness
- Must have a global mechanism to eliminate unneeded stores
- Must have a global mechanism to simplify control-flow predicates

All of these will come later in the course

Global CSE

So, we have a three step process

1. Compute **AVAIL**(*b*), ∀ block *b*
2. Assign unique global names to expressions in **AVAIL**(*b*)
3. Perform replacement with local value numbering

Earlier in the lecture, the slide said

> *Assume, without loss of generality, that we can compute available expressions for a procedure.*

Next lecture, we will make good on that assumption