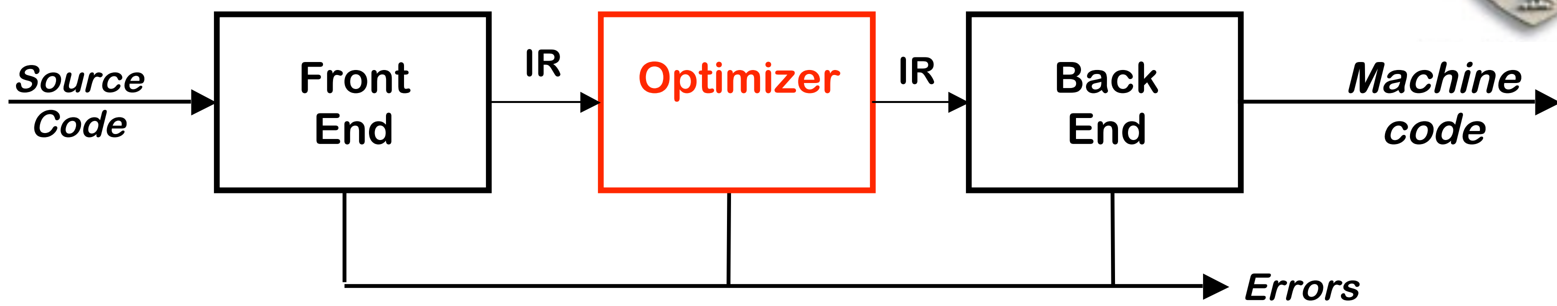
Introduction to Optimization

COMP 412

Fall 2005

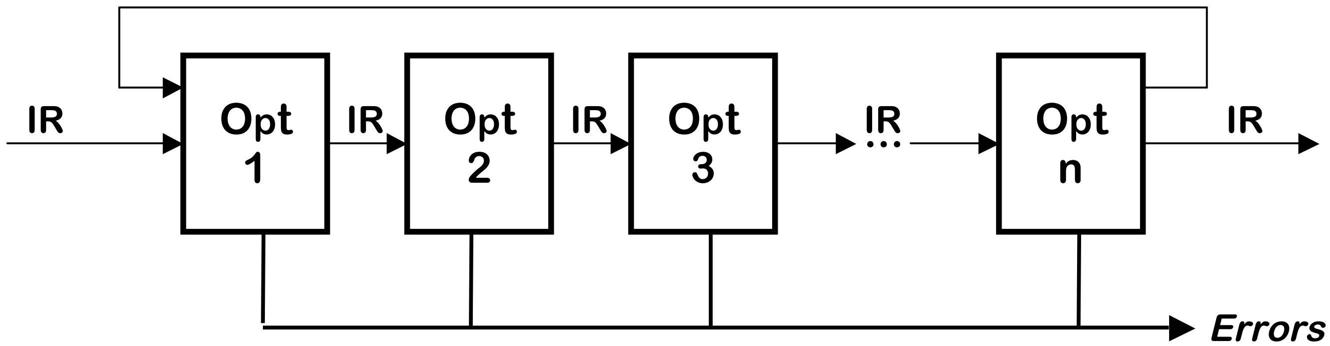
Traditional Three-pass Compiler



Code Improvement (or Optimization)

- Analyzes IR and rewrites (or *transforms*) IR
- Primary goal is to reduce running time of the compiled code
  - May also improve space, power consumption, …
- Must preserve "meaning" of the code
  - Measured by values of named variables
  - A course (or two) unto itself

The Optimizer



Modern optimizers are structured as a series of passes

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

The Role of the Optimizer

- The compiler can implement a procedure in many ways
- The optimizer tries to find an implementation that is "better"
  - Speed, code size, data space, …

To accomplish this, it

- Analyzes the code to derive knowledge about run-time behavior
  - Data-flow analysis, pointer disambiguation, …
  - General term is "static analysis"
- Uses that knowledge in an attempt to improve the code
  - Literally hundreds of transformations have been proposed
  - Large amount of overlap between them

Nothing "optimal" about optimization

- Proofs of optimality assume restrictive & unrealistic conditions

Scalar Optimization

- Uniprocessor optimization
  - Applied at a low level of abstraction (near assembly)
  - Targets performance on a single processor
  - Usually excludes issues that require near-source analysis
    - → Memory hierarchy, loop-level parallelism
- Transformations a sophisticated user would expect
  - Constant folding, redundancy elimination, dead code elimination
  - Code motion, operator strength reduction, …

Among the most effective scalar optimizations are

- Register allocation, constant folding, redundancy elimination

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 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, or *available*
- Rewriting the code to eliminate the redundant evaluation

One technique for accomplishing both is called *value numbering*

Value Numbering

Local algorithm due to Balke (1968) or Ershov (1954)

The key notion

- Assign an identifying number, V(n), to each expression
  - V(x+y) = V(j) iff x+y and j always have the same value
  - Use hashing over the value numbers to make it efficient
- Use these numbers to improve the code

Within a basic block; definition becomes more complex across blocks

Improving the code

- Replace redundant expressions
  - Same VN ⇒ refer rather than recompute
- Simplify algebraic identities
- Discover constant-valued expressions, fold & propagate them
- Technique designed for low-level, linear IRs, similar methods exist for trees (e.g., build a DAG)

Local Value Numbering

The Algorithm

For each operation *o* = ‹operator, o_1, o_2› in the 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

- If not, use multi-set discrimination (*see p. 251 in EaC*)

Handling algebraic identities

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

Local Value Numbering

An example

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

Two redundancies:

- Eliminate stmts with a *
- Coalesce results ?

Options:

- Use $c^3 \leftarrow b^3$
- Save a^3 in t^3
- Rename around it

Local Value Numbering

Example (continued):

Original Code

$a_0 \leftarrow x_0 + y_0$
* $b_0 \leftarrow x_0 + y_0$
$a_1 \leftarrow 17$
* $c_0 \leftarrow x_0 + y_0$

With VNs

$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0^3 \leftarrow x_0^1 + y_0^2$
$a_1^4 \leftarrow 17$
* $c_0^3 \leftarrow x_0^1 + y_0^2$

Rewritten

$a_0^3 \leftarrow x_0^1 + y_0^2$
* $b_0^3 \leftarrow a_0^3$
$a_1^4 \leftarrow 17$
* $c_0^3 \leftarrow a_0^3$

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

Example (continued):

<u>Original Code</u>

$a_0 \leftarrow x_0 + y_0$

* $b_0 \leftarrow x_0 + y_0$

$a_1 \leftarrow 17$

* $c_0 \leftarrow x_0 + y_0$

Renaming to provide a unique name for each *definition* is the *key idea* underlying <u>S</u>tatic <u>S</u>ingle <u>A</u>ssignment form (SSA form)

Renaming:

- Give each value a unique name
- Makes it clear

Simple Extensions to Value Numbering

Constant folding

- Add a bit that records when a value is constant
- Evaluate constant values at compile-time
- Replace with load immediate or immediate operand
- No stronger local algorithm

Identities (on VNs) :

x←y, x+0, x-0, x∗1, x÷1, x-x, x∗0, x÷x, x∨0, x ∧ 0xFF…FF, max(x,MAXINT), min(x,MININT), max(x,x), min(y,y), and so on ...

Algebraic identities

- Must check (many) special cases
- Replace result with input VN
- Build a decision tree on operation

Safety & Value Numbering

Why is local value numbering safe?

- Hash table starts empty
- Expressions placed in table as processed
- If $\langle operator, VN(o_1), VN(o_2)\rangle$ is in the table, then
  - It has already occurred at least once in the block
  - Neither o_1 nor o_2 have been subsequently redefined
    - → The mapping uses $VN(o_1)$ and $VN(o_2)$, not o_1 and o_2

If $\langle operator, VN(o_1), VN(o_2)\rangle$ has a VN, the compiler can safely use it

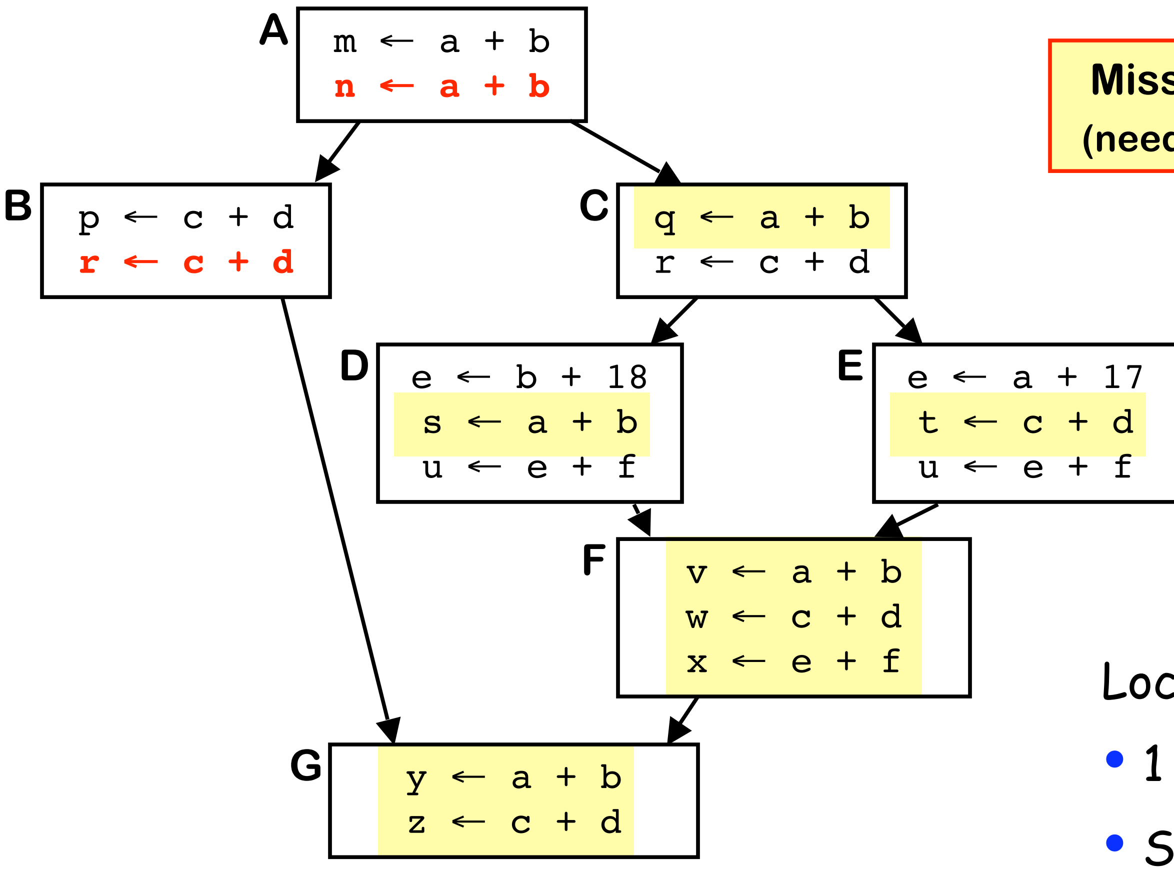
- Algorithm incrementally constructs a proof that $\langle operator, VN(o_1), VN(o_2)\rangle$ is redundant
- Algorithm modifies the code, but does not invalidate the table

Profitability & Value Numbering

When is local value numbering profitable?

- If reuse is cheaper than re-computation
  - Does not cause a spill or a copy (*hard to determine*)
  - In practice, assumed to be true
- Local constant folding is always profitable
  - Re-computing uses a register, as does load immediate
  - Immediate form of operation avoids even that cost
- Algebraic identities
  - If it eliminates an operation, it is profitable (x + 0)
  - Profitability of simplification depends on target (2x ⇒ x+x)
  - Easy to factor into design (*don't do the unprofitable ones!*)

Local Value Numbering

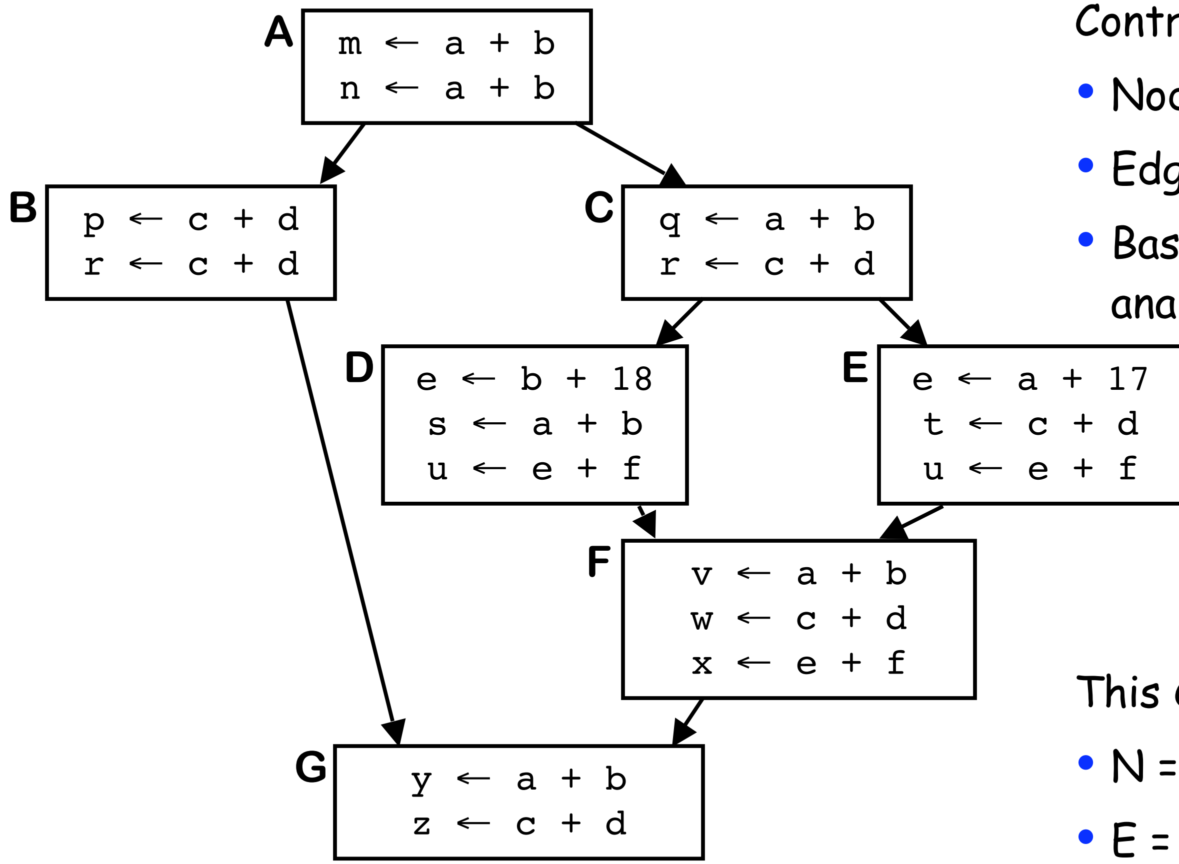


Missed opportunities
(need stronger methods)

Local Value Numbering

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

An Aside on Terminology



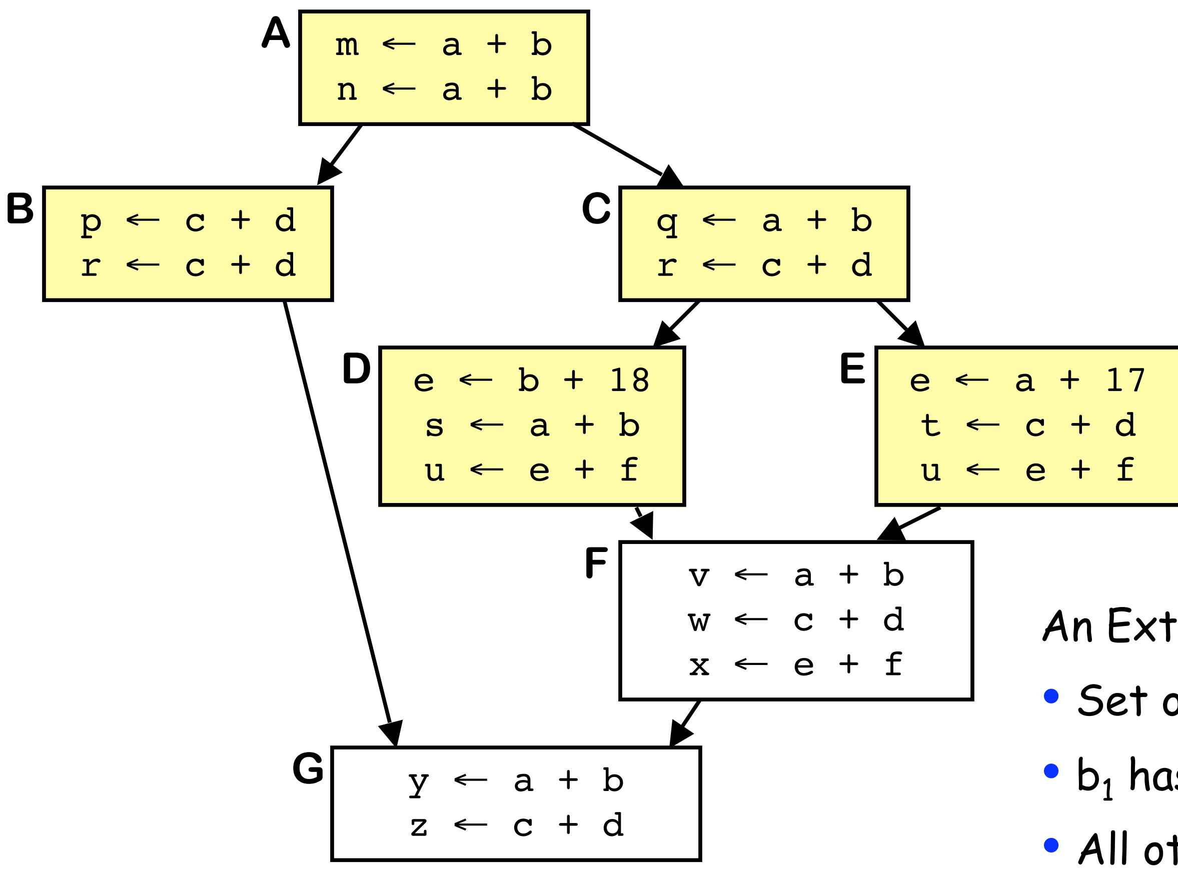
Control-flow graph (CFG)

- 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

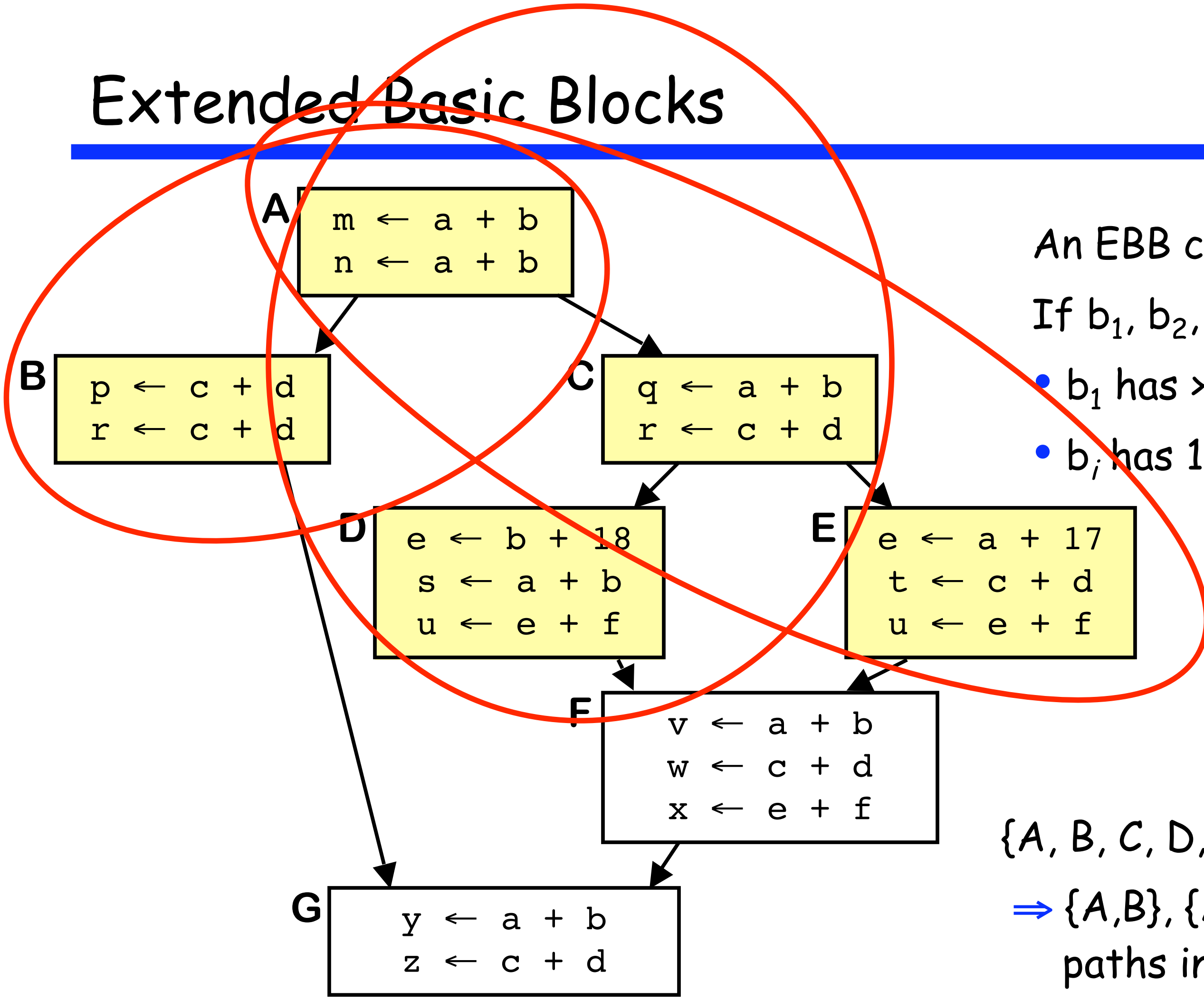
Extended Basic Blocks



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 predecessor
- EBBs provide more context for optimization than single blocks

Extended Basic Blocks



An EBB contains 1 or more path

If $b_1, b_2, \ldots, b_n$ is a path

- b_1 has > 1 predecessor
- b_i has 1 predecessor, b_{i-1}

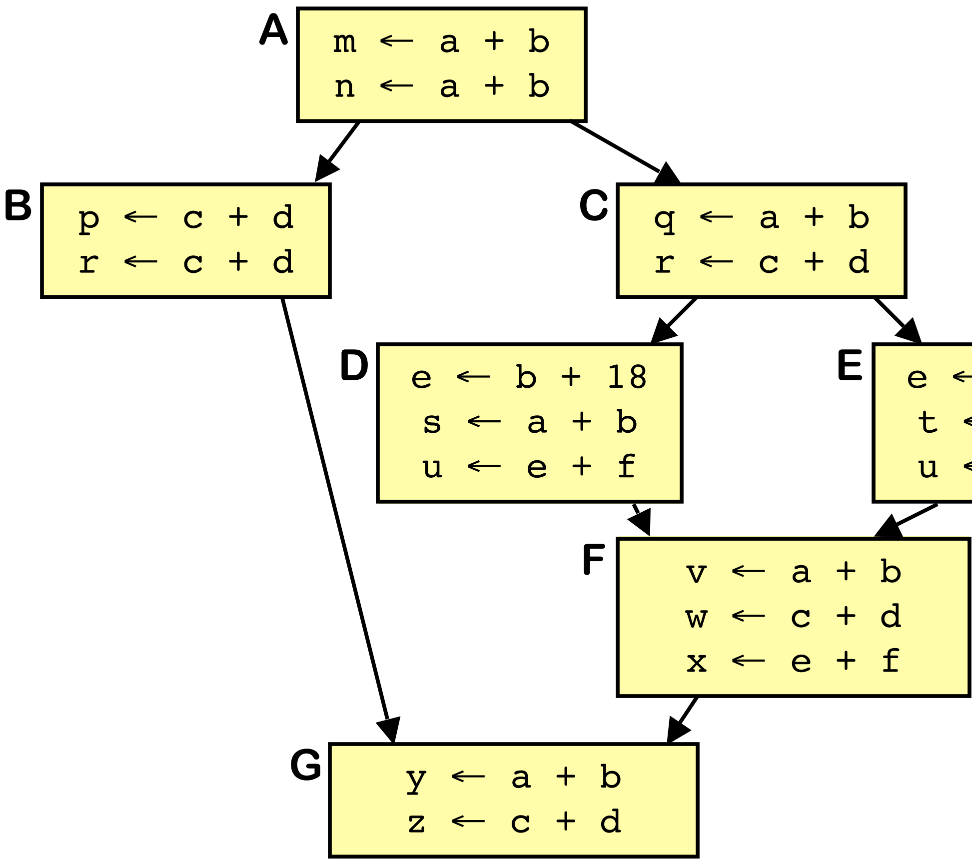
{A, B, C, D, E} is an EBB

⇒ {A,B}, {A,C,D}, and {A,C,E} are paths in {A, B, C, D, E}

{F} and {G} are also EBBs

⇒ They have only trivial paths

Superlocal Value Numbering



The Concept

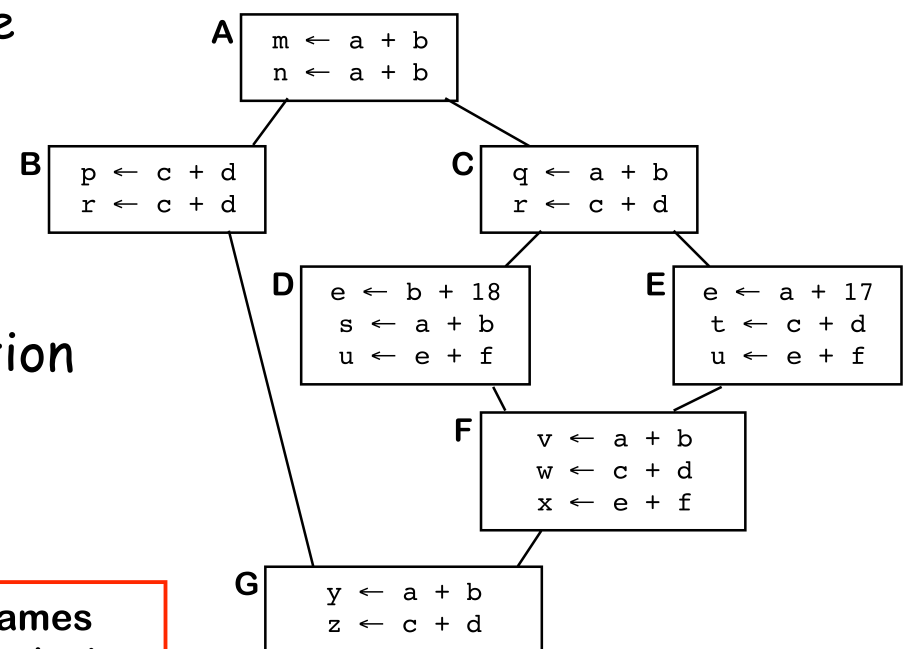
- Apply local method to each path in the EBB
- Do {A,B}, {A,C,D}, & {A,C,E}
- Obtain reuse from ancestors
- Does not help with F or G
- Key: avoid re-analyzing A & C

Superlocal Value Numbering

EaC: § 5.7.3 & App. B

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



To simplify matters

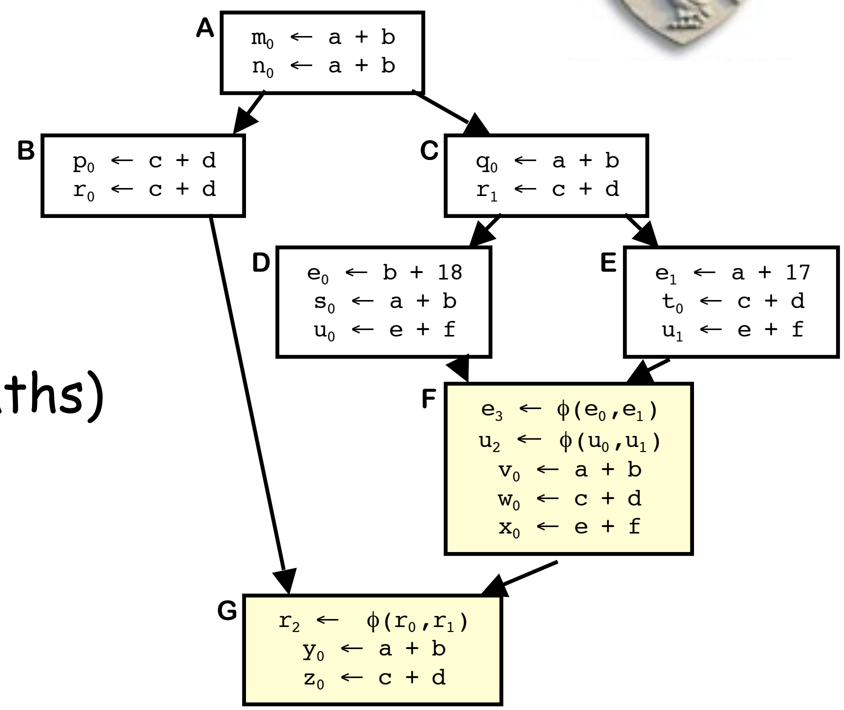
- Unique name for each definition
- Makes name ⇔ VN
- Use the SSA name space

Subscripted names from example in last lecture

What About Larger Scopes?

We have not helped with F or G

- Multiple predecessors
- Not part of an EBB
- "Traces" do not capture safety conditions (value known on all paths)

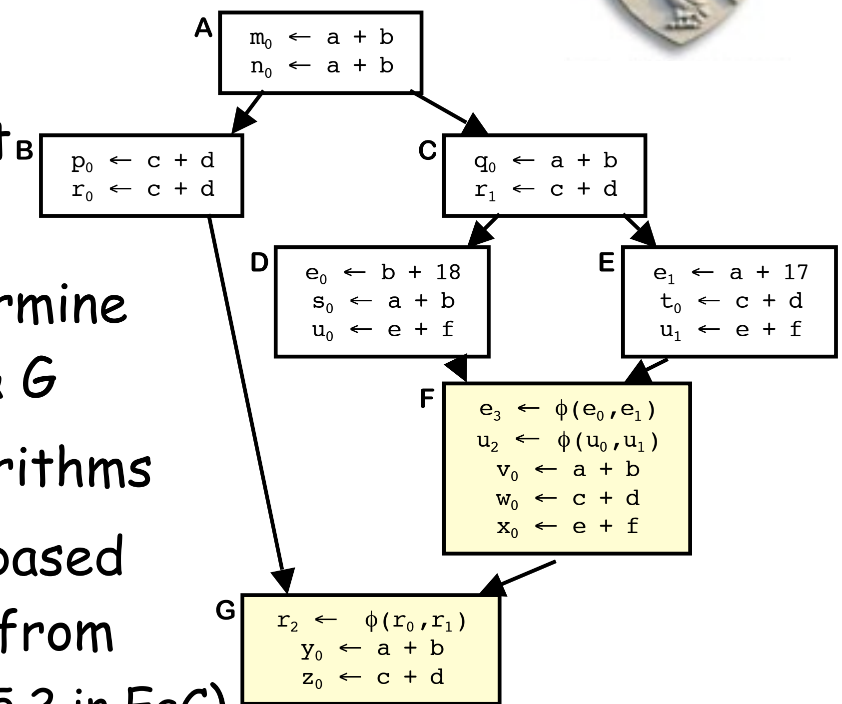


- Must decide what facts hold in F and in G
  - For G, combine B & F?
  - Merging state is expensive
  - Fall back on what's known

What About Larger Scopes?

Two interesting approaches

- Change IR to represent context in an explicit way (SSA form)
- Perform global analysis to determine what facts hold on entry to F & G



Approaches lead to different algorithms

- SSA form leads to fast, value-based technique using strong notions from control-flow analysis (DVNT, §8.5.2 in EaC)
- Global analysis leads to classic formulation of redunancy analysis as a problem in *global data-flow analysis*
  - Syntactic equivalence rather than value equivalence