Compilers

Topic: Local Optimizations

Monsoon 2011, IIIT-H, Suresh Purini

Optimization

Compilers operate at many granularities or scopes

- Local techniques
  - Work on a single basic block
  - Maximal length sequence of straight-line code
- Regional techniques
  - Consider multiple blocks, but less than whole procedure
  - Single loop, loop nest, dominator region, …
- Intraprocedural (or global) techniques
  - Operate on an entire procedure (but just one)
  - Common unit of compilation
- Interprocedural (or whole-program) techniques
  - Operate on > 1 procedure, up to whole program
  - Logisitical issues related to accessing the code (link time?)
- Link Time Optimizations

Optimization

At each of these scopes, the compiler uses different graphs

- Local techniques
  - Dependence graph (instruction scheduling)
- Regional Techniques
  - Control-flow graph (natural loops)
  - Dominator tree
- Intra-procedural (or global) techniques
  - Control-flow graph
  - Def-use chains, sparse evaluation graphs, SSA as graph
- Inter-procedural (or whole-program) techniques
  - Call (multi) graph

Analysis versus Transformation

We want to differentiate between analysis and transformation

- Analysis reasons about the code's behavior
- Transformation rewrites the code to change its behavior

- Local techniques can interleave analysis and transformation
  - Property of basic block: operations execute in defined order
- Over larger regions, the compiler typically must complete its analysis before it transforms the code
- Leads to confusion in terminology
  - Don't use "optimization" for both analysis & transformation

Limitations of GCSE

$$\begin{aligned} a &\leftarrow b + c \\ b &\leftarrow a - d \\ c &\leftarrow b + c \\ d &\leftarrow a - d \end{aligned}$$

Original Block

$$\begin{aligned} a &\leftarrow b + c \\ b &\leftarrow a - d \\ c &\leftarrow b + c \\ d &\leftarrow b \end{aligned}$$

Rewritten Block

$$\begin{aligned} a &\leftarrow b \times c \\ d &\leftarrow b \\ e &\leftarrow d \times c \end{aligned}$$

Effect of Assignment

Value Numbering

$$
\begin{array}{llll}
a^2 & \leftarrow & b^0 & + & c^1 \\
b^4 & \leftarrow & a^2 & - & d^3 \\
c^5 & \leftarrow & b^4 & + & c^1 \\
d^4 & \leftarrow & a^2 & - & d^3
\end{array}
$$

Along with the Value Number Hash Table, maintain two other data structures.

1. Variable to Value Number Map
2. Value Number to Variables Map

for $i \leftarrow 0$ to $n-1$, where the block has n operations "$T_i \leftarrow L_i\ Op_i\ R_i$"

1. *get the value numbers for L_i and R_i*
2. *construct a hash key from Op_i and the value numbers for L_i and R_i*
3. *if the hash key is already present in the table then*
   replace operation i with a copy of the value into T_i and
   associate the value number with T_i

   else
   insert a new value number into the table at the hash key location
   record that new value number for T_i

Value Numbering

Original Code	With VNs	
$a = x + y$	$a^3 \leftarrow x^1 + y^2$	$a^3 = x^1 + y^2$
$b = x + y$	$b^3 \leftarrow x^1 + y^2$	$b^3 = x^1 + y^2$
$a = 17$	$a^4 \leftarrow 17^4$	$a^4 = 17^4$
$c = x + y$	$c^3 \leftarrow x^1 + y^2$	$c^3 = b^3$

Value Numbering

Original Code

$$
\begin{aligned}
a &= x + y \\
a &= 17 \\
c &= x + y
\end{aligned}
$$

With VNs

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

- Issue: Although the computation in c=x+y is redundant, its value (VN-3) is not available in any variable
- Possible Solution: Introduce temporary variables

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

Static Single Assignment Form(SSA)

Original Code	SSA Form
$a = x + y$	$a_0 = x + y$
$b = x + y$	$b_0 = x + y$
$a = 17$	$a_1 = 17$
$c = x + y$	$c_0 = x + y$

- Idea: Each definition (or assignment) to a variable creates a new version of variable. A new name space would be created using this approach.

Static Single Assignment (SSA)

Original Code

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

SSA Form

$$
\begin{aligned}
a_0 &= x + y \\
b_0 &= x + y \\
a_1 &= 17 \\
c_0 &= x + y
\end{aligned}
$$

- Issue: How to reconcile with the rest of the name space in other BBs?

$$
\begin{aligned}
a_0 &= x + y \\
b_0 &= x + y \\
a_1 &= 17 \\
c_0 &= x + y \\
a &= a_1 \\
b &= b_0 \\
c &= c_0
\end{aligned}
$$

Value Numbering – Static Single Assignment (SSA)

Original Code

$a = x + y$

$a = 17$

$c = x + y$

SSA Form

$a_0 = x + y$

$a_1 = 17$

$c_0 = x + y$

SSA Form with VNs

${a_0}^3 = x^1 + y^2$

${a_1}^4 = 17^4$

${c_0}^3 = x^1 + y^2$

Optimized Code

${a_0}^3 = x^1 + y^2$

${a_1}^4 = 17^4$

${c_0}^3 = {a_0}^3$

SSA Form

A program is in SSA form when it meets two constraints

- Each definition has a distinct name
  - Implication: Names correspond to specific definition points in code
- Each use refers to a single definition

Translating to SSA Form

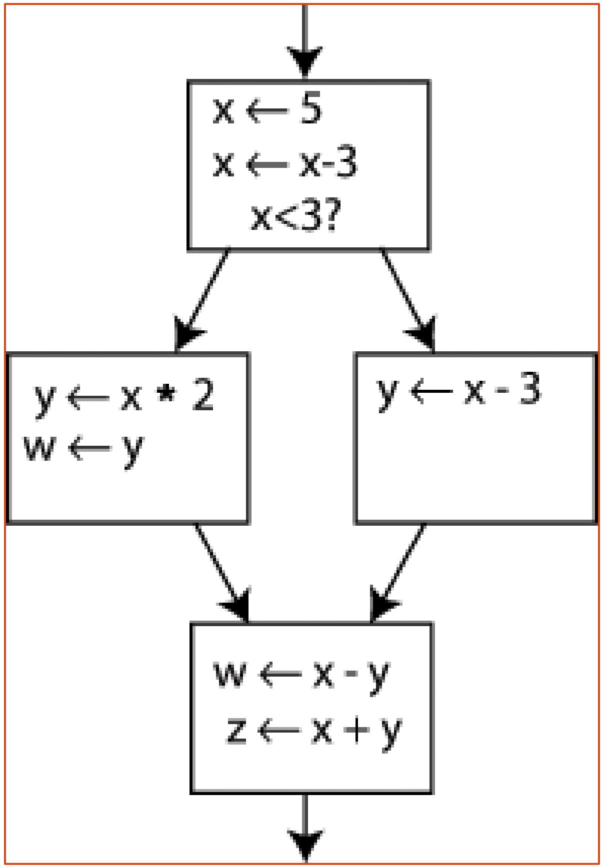
Question: How to translate IR into SSA form?

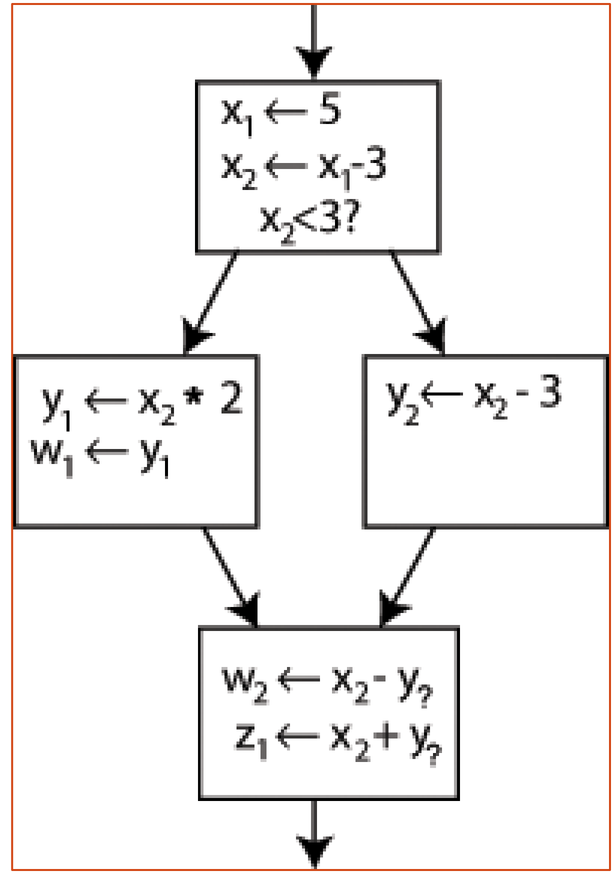
- Easy for straight line sequence of code
  - Each assignment to a variable is given a unique name
  - All of the uses reached by that assignment are renamed

```
1: a = x + y
2: a = a + 3
3: b = x + y
```

$$
\begin{aligned}
&1:\ a_1 = x_0 + y_0\\
&2:\ a_2 = a_1 + 3\\
&3:\ b_1 = x_0 + y_0
\end{aligned}
$$

Translating to SSA Form

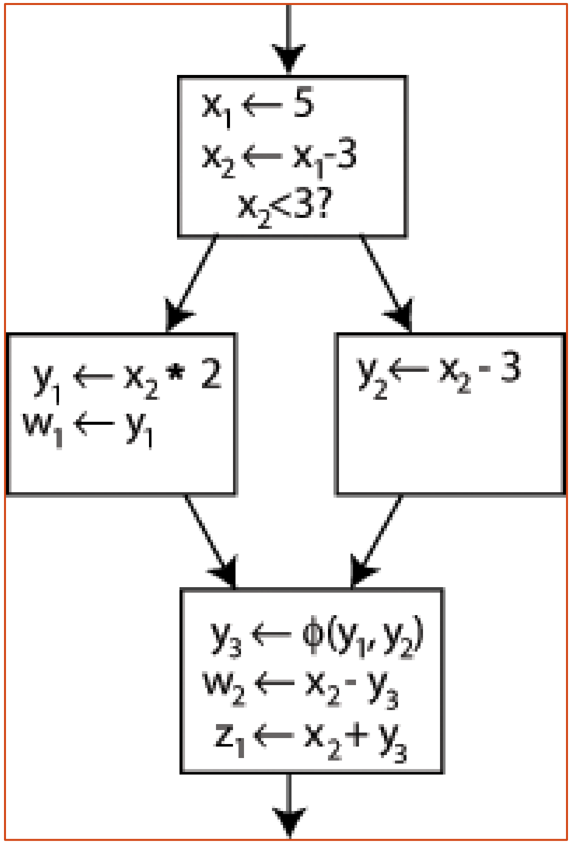




- What's easy: Straight line sequence of codes, splits in CFGs
- What's hard: Joins in CFGs

Translating to SSA Form

- Introduce Phi Functions to handle joins wherever necessary
- Question: Isn't a phi function necessary for the variable x?



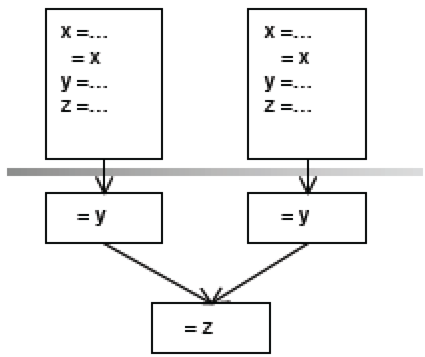
SSA Variations

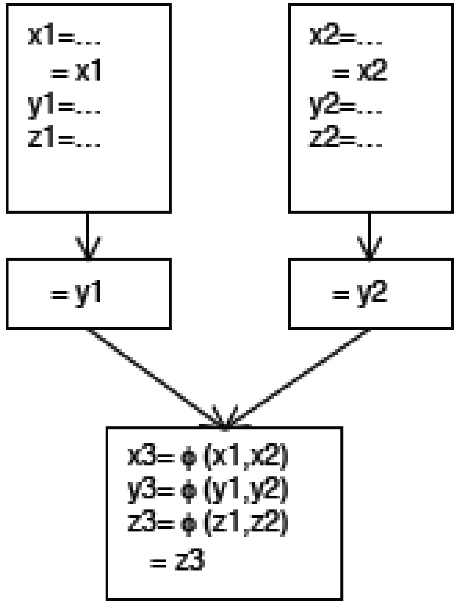
Many variations of SSA exist.

- Minimal SSA
- Pruned SSA
- Semi-pruned SSA
- ………

Minimal SSA

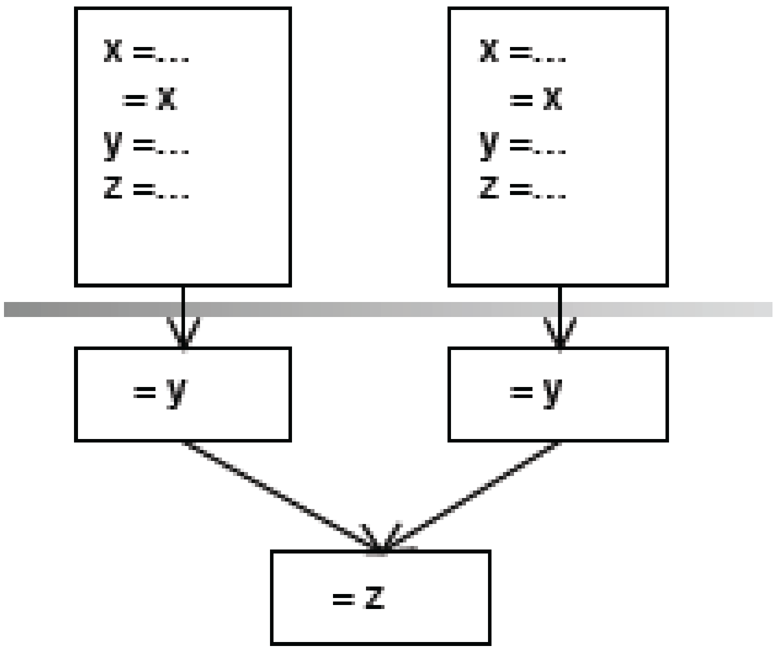
- Insert a phi function at any join point where two distinct definitions for the same original name meet.
- The minimal number consistent with the definition of SSA
- Some of those phi functions may be dead

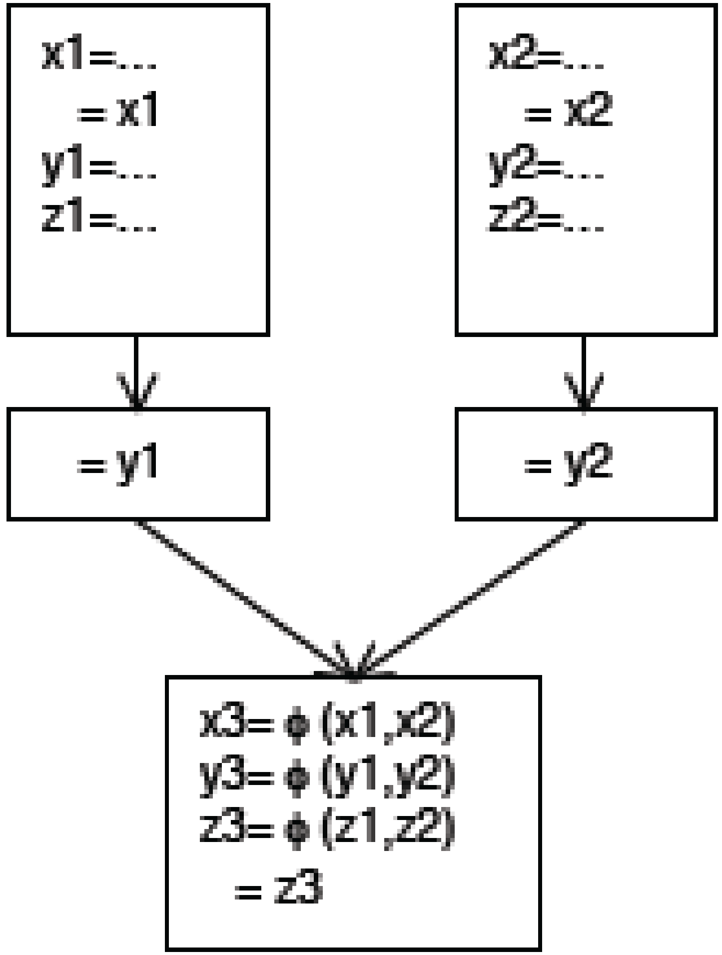




Minimal SSA form

Minimal SSA

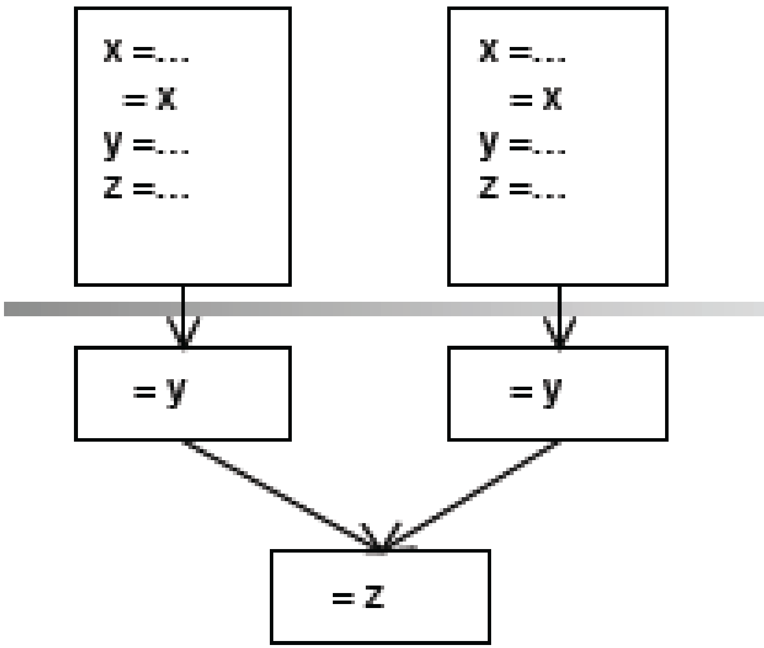


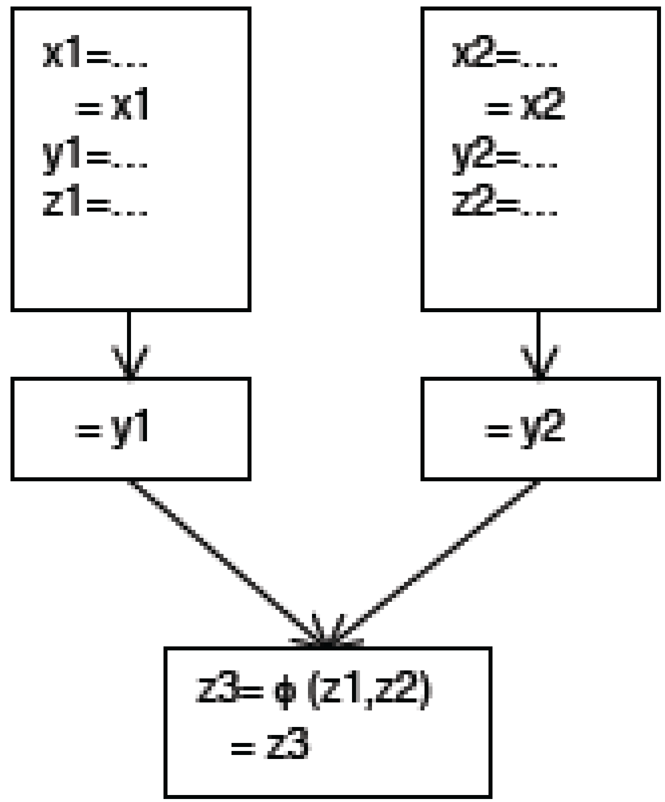


Minimal SSA form

Pruned SSA

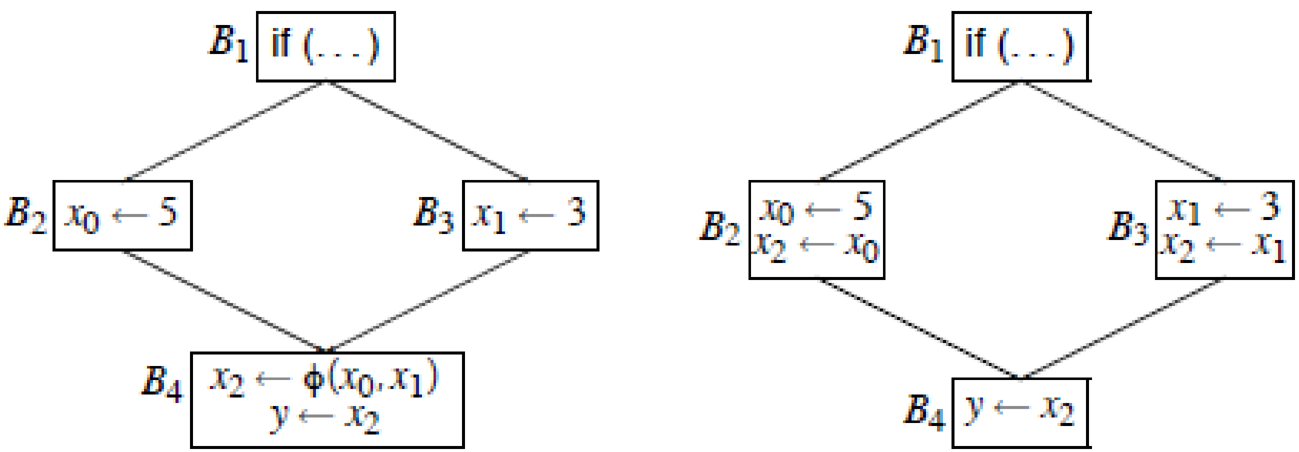
- Same as minimal SSA
- Perform liveness analysis to drop dead phi functions



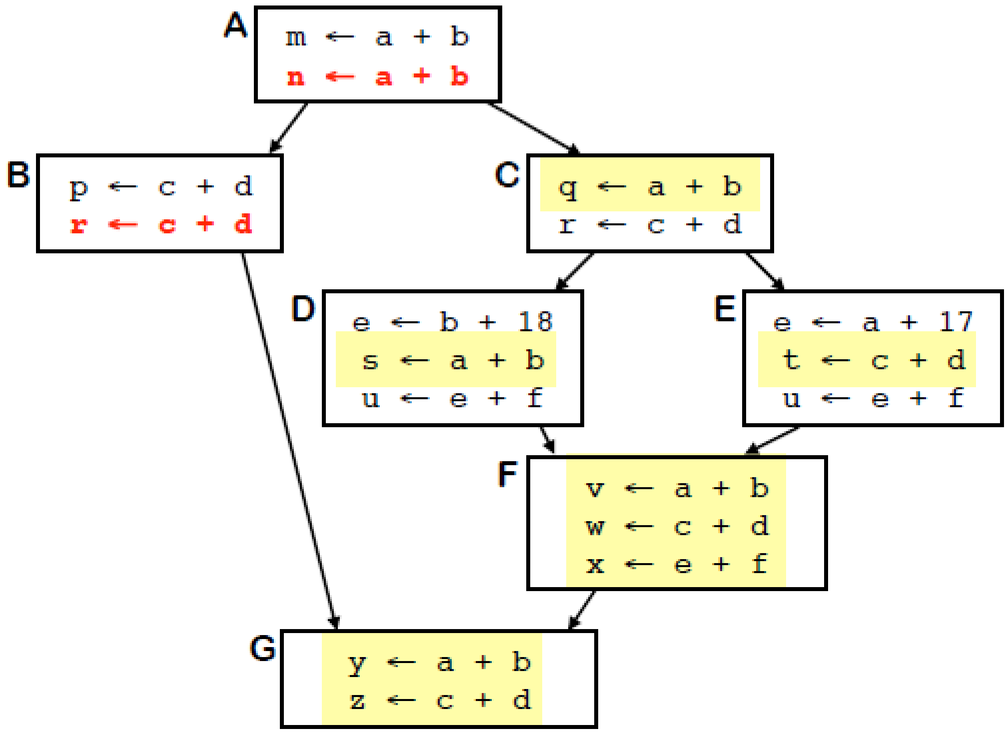


Translating out of SSA Form

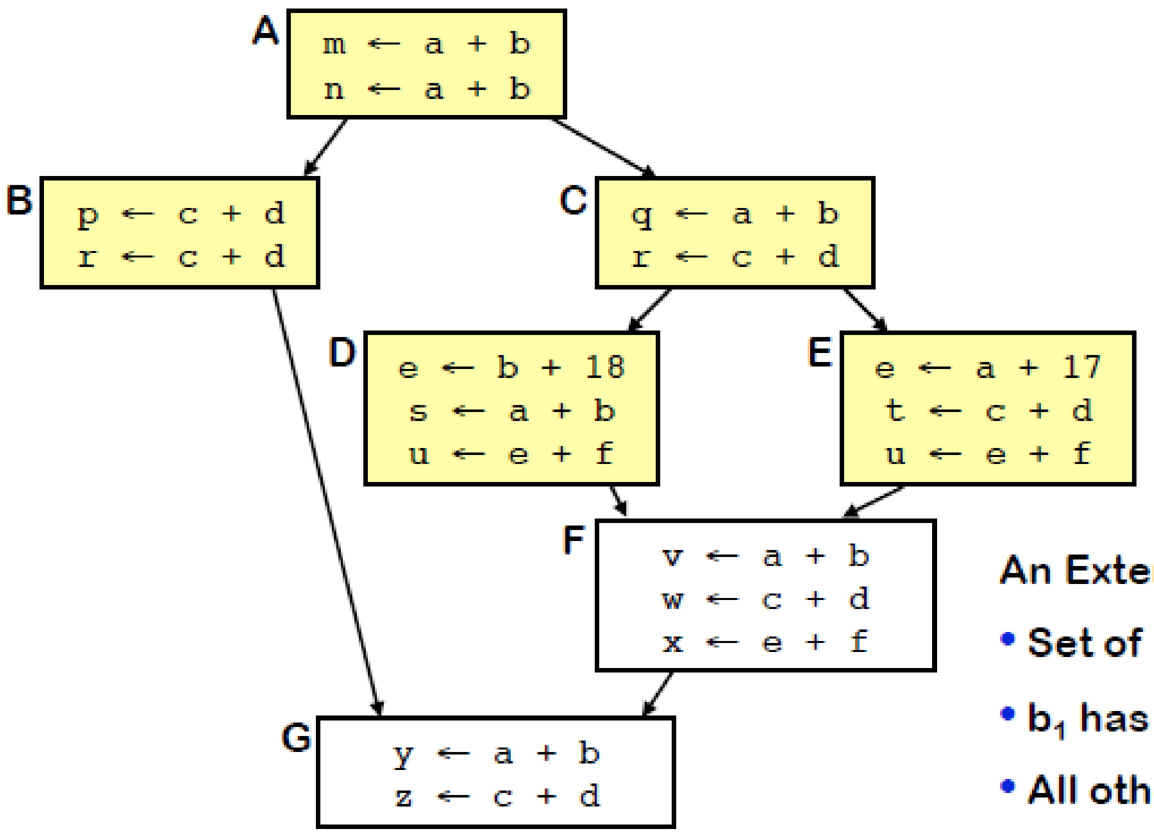
- Replace phi nodes with copy statements in the predecessors



Superlocal Value Numbering



Regional Optimization: Extended Basic Blocks



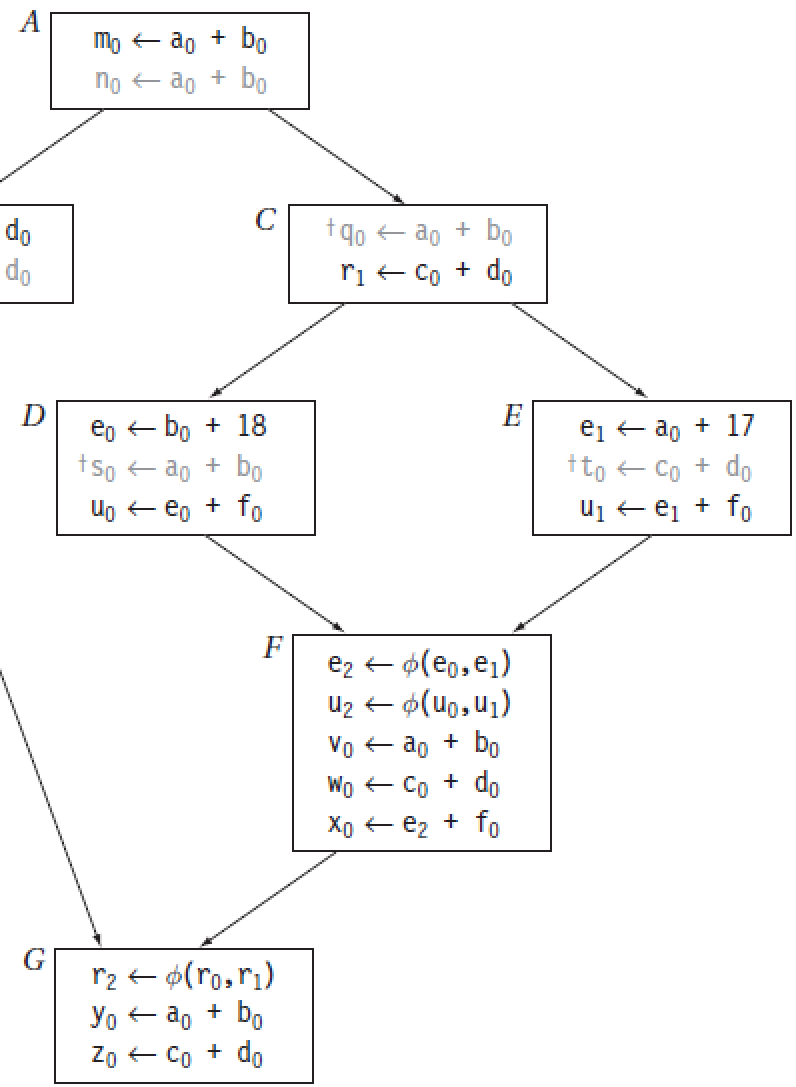
An Extended Basic Block (EBB)

- Set of blocks $b_1, b_2, \ldots, b_n$
- b_1 has > 1 predecessor
- All other b_i have 1 predecessor
- EBBs provide more context for optimization

Extended Basic Blocks

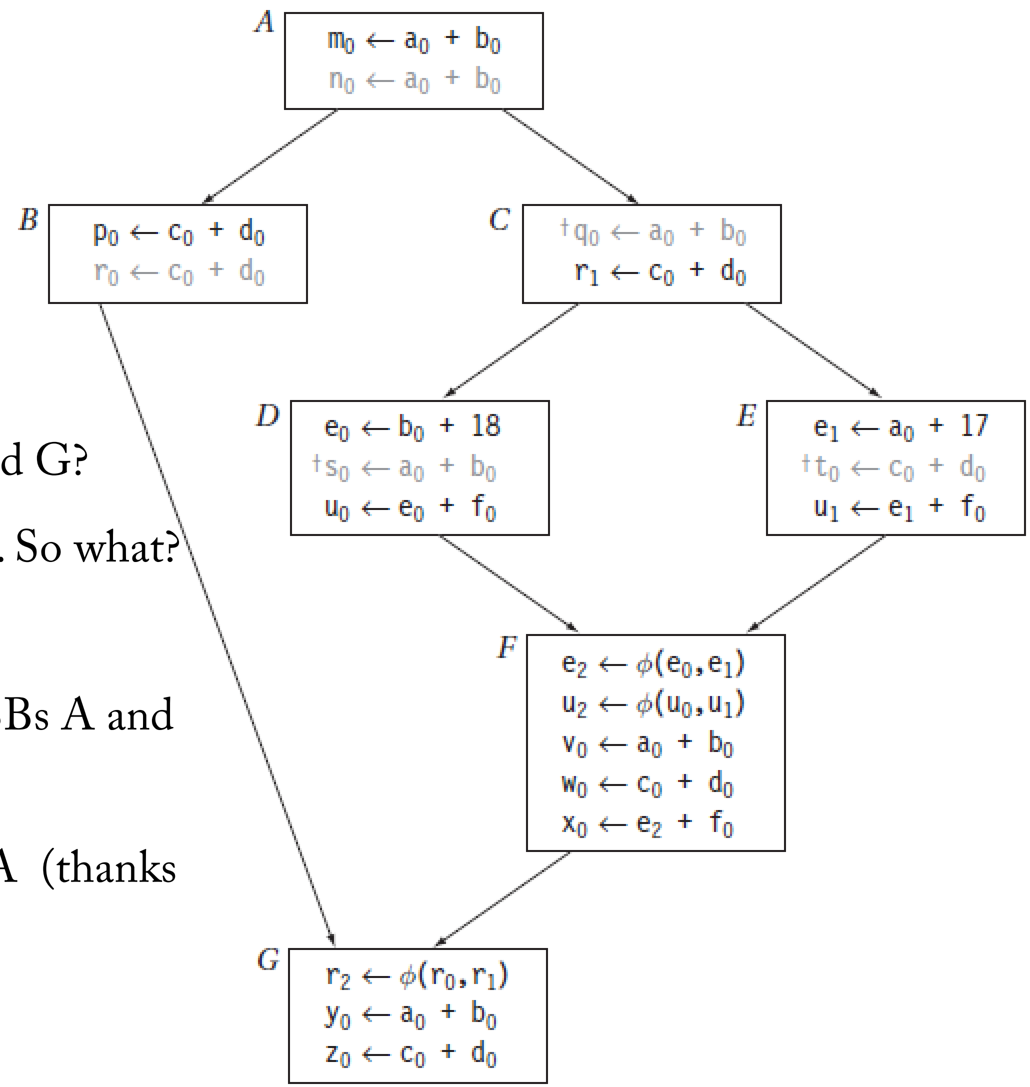
- An EBB can be seen as a tree with nodes having in-degree 1 (except for the root node) and arbitrary out degree
- Value Numbering Algorithm on an EBB
- Key Idea: While optimizing a BB, we can use the values generated in any of the ancestor Basic Blocks

Supervalue Local Numbering



Dominator-Based Value Numbering

- What about the redundancies in BBs F and G?
- Problem: They have multiple predecessors. So what?
- Observation
  - BB F can use the values computed in BBs A and C (thanks to SSA)
  - BB G can use the values computed in A (thanks to SSA)



Dominators

Def: x dominates y if and only if every path from the entry of the control-flow graph to the node for y includes x.

- By definition, x dominates x
- We associate a Dom set with each node
- $|\text{Dom}(x)| \geq 1$

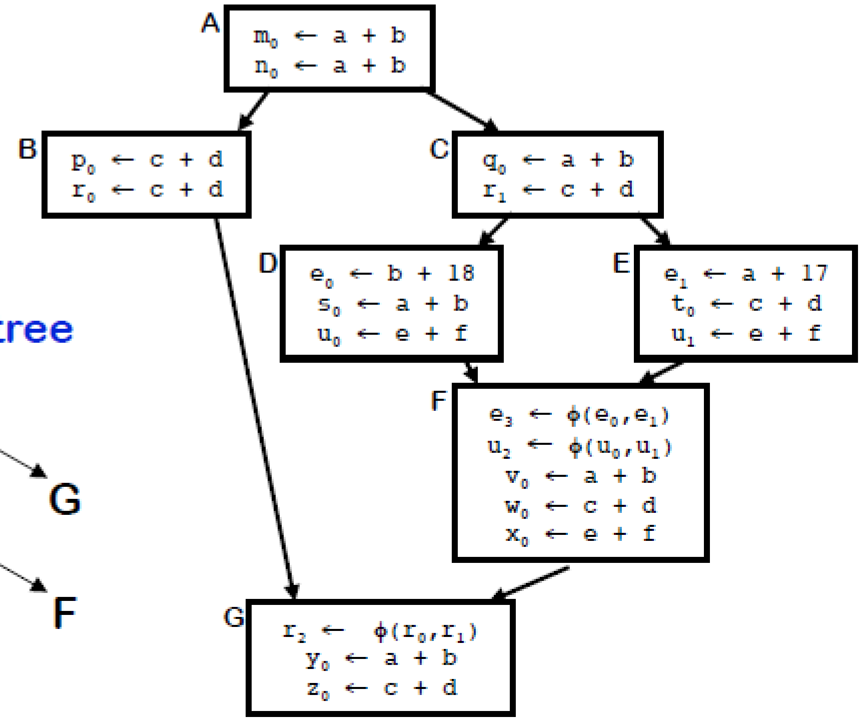
Immediate dominators

- For any node x, there must be a y in Dom(x) closest to x
- We call this y the immediate dominator of x
- As a matter of notation, we write this as IDom(x)

Dominators

Dominators have many uses in analysis & transformation

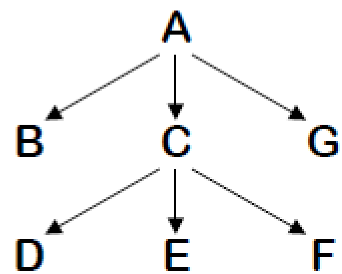
- **Finding loops**
- **Building SSA form**
- **Making code motion decisions**



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



Dominator Based Value Numbering

- Key Idea: While Value Numbering a BB B, start with IDom(B)'s Value Number table
- While processing BB F, start with BB C's Value Number table (which includes BB A's also)
- Similarly for BB G, start with the Value Number table of BB A