SSA: Single Static Assignment

15-745: Optimizing Compilers

February 16, 2006

Previously… Def-Use Chains

```
…
for (i=0; i++; i<10) {
      … = … i …;
      …
}
for (i=j; i++; i<20) {
      … = … i …
}
```


Def-Use chains are expensive

```
…
switch (i) {
case 0: x=3;
case 1: x=1;
case 2: x=6;
case 3: x=7;
default: x = 11;
}
switch (j) {
case 0: y=x+7;
case 1: y=x+4;
case 2: y=x-2;
case 3: y=x+1;
default: y=x+9;
}
…
```


Def-Use chains are expensive

```
…
switch (i) {
case 0: x=3; break;
case 1: x=1; break;
case 2: x=6; break;
case 3: x=7; break;
default: x = 11;
}
switch (j) {
case 0: y=x+7;
case 1: y=x+4;
case 2: y=x-2;
case 3: y=x+1;
default: y=x+9;
}
…
```

In general,
- N defs
- M uses
- ⇒ O(NM) space and time

A solution is to limit each var to ONE def site

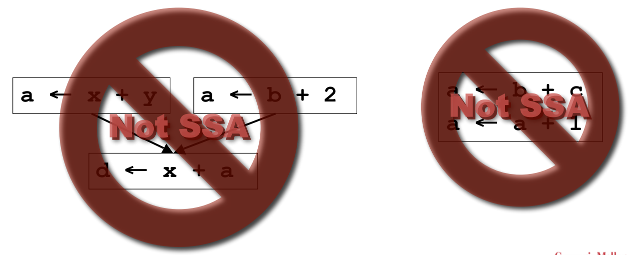
Def-Use chains are expensive

```
…
switch (i) {
case 0: x=3; break;
case 1: x=1; break;
case 2: x=6; break;
case 3: x=7; break;
default: x = 11;
}
x1 is one of the above x's
switch (j) {
case 0: y=x1+7;
case 1: y=x1+4;
case 2: y=x1-2;
case 3: y=x1+1;
default: y=x1+9;
}
…
```

A solution is to limit each var to ONE def site

SSA

- Static single assignment is an **IR** where every variable is assigned a value *at most once*



Advantages of SSA

- Makes du-chains explicit
  – every definition knows its uses
  – every use knows its single definition
- Makes dataflow optimizations
  – easier
  – faster
- For most optimizations reduces space/time requirements

Simple SSA Optimizations

- Dead Code Elimination
  – a definition with no uses (and no ????)

```
a ← x / y
```

- Constant Propagation
  – a use with a constant definition

```
a ← 1
c ← a + b
```

SSA History

- Developed by Wegman, Zadeck, Alpern, and Rosen in 1988
  - and improved by Cytron, Ferrante, Wegman, and Zadeck in 1989
- New to gcc 4.0, used in ORC, used in both IBM and Sun Java JIT compilers

Converting to SSA

- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.

```
a ← x + y
b ← a + x
a ← b + 2
c ← y + 1
a ← c + a
```


Converting to SSA

- Easy for a basic block:
  - assign to a fresh variable at each stmt.
  - Each use uses the most recently defined var.

```
a ← x + y
b ← a + x
a ← b + 2
c ← y + 1
a ← c + a
```

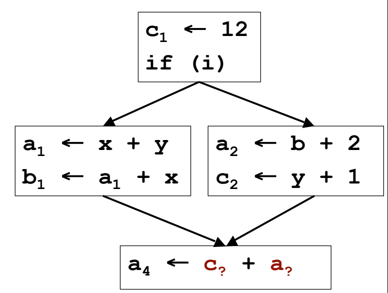
⇒

$a_1 \leftarrow x + y$
$b_1 \leftarrow a_1 + x$
$a_2 \leftarrow b_1 + 2$
$c_1 \leftarrow y + 1$
$a_3 \leftarrow c_1 + a_2$

What about at joins in the CFG?

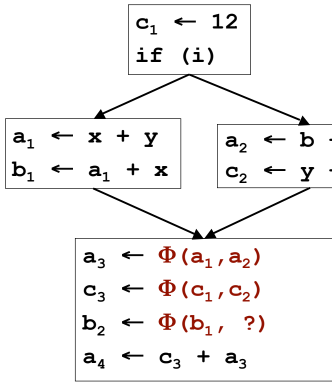
Merging at Joins

```
c ← 12
if (i) {
  a ← x + y
  b ← a + x
} else {
  a ← b + 2
  c ← y + 1
}
a ← c + a
```



Use a notional fiction: A Φ function

Merging at Joins



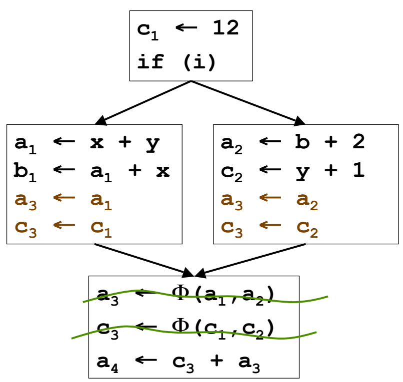
The Φ function

- Φ merges multiple definitions along multiple control paths into a single definition
- At a BB with p predecessors, there are p arguments to the Φ function

  $x_{new} \leftarrow \Phi(x_1, x_2, x_3, \ldots, x_p)$

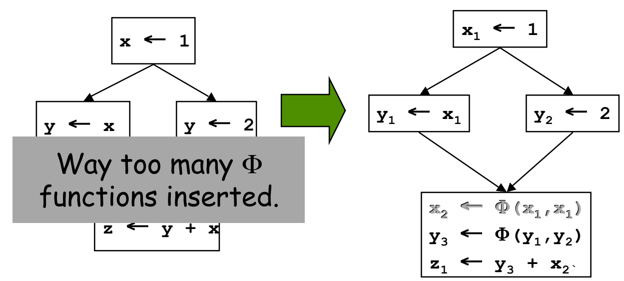
- How does phi choose which x_i to use?
  - We don't really care!
  - If we care, use moves on each incoming edge

"Implementing" Φ



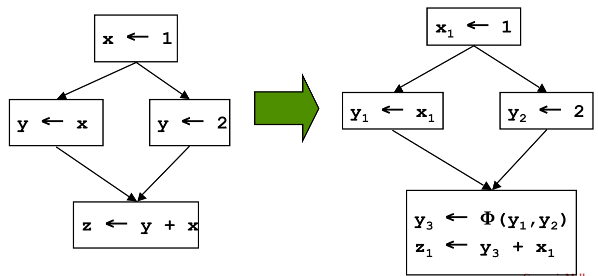
Trivial SSA

- Each assignment generates a fresh variable
- At each join point insert Φ functions for all live variables

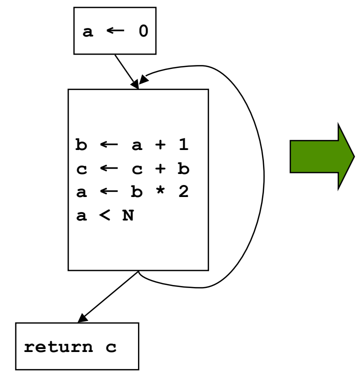


Minimal SSA

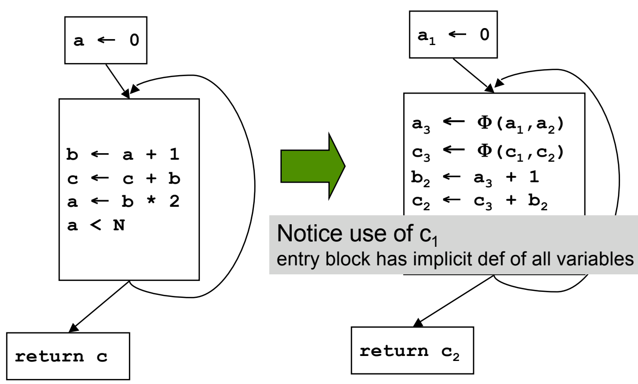
- Each assignment generates a fresh variable.
- At each join point insert Φ functions for all variables with multiple outstanding defs.



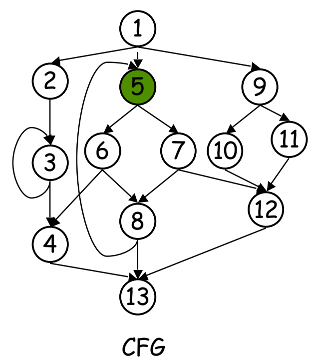
Another Example



Another Example



When do we insert Φ?



If there is a def of **a** in block 5, which nodes need a Φ()?

Alternatively, which nodes don't need a Φ()?

When do we insert Φ?

- We insert a Φ function for variable **a** in block Z iff:
  - There exist blocks X and Y, X ≠ Y, such that **a** is defined in X and Y
    - **a** is defined in more than one block
  - There exists a non-empty path from X to Z, P_{XZ}, and a non-empty from Y to Z, P_{YZ} s.t.
    - $P_{XZ} \cap P_{YZ} = \{ Z \}$
      - paths P_{XZ} and P_{YZ} have no nodes in common except for Z
    - $Z \notin P_{XQ}$ or $Z \notin P_{XR}$ where $P_{XZ} = P_{XQ} \rightarrow Z$ and $P_{YZ} = P_{XR} \rightarrow Z$
      - if Z is in contained elsewhere in P_{XZ} before the end, it is only found at the end of P_{YZ} and vice versa

This is the path-convergence criterion

Dominance Property of SSA

- In SSA definitions dominate uses.
  - If x_i is used in x ← Φ(…, x_i, …), then BB(x_i) dominates ith pred of BB(PHI)
  - If x is used in y ← … x …, then BB(x) dominates BB(y)
- We can use dominance information to get an efficient algorithm for converting to SSA

Dominators

a dom b

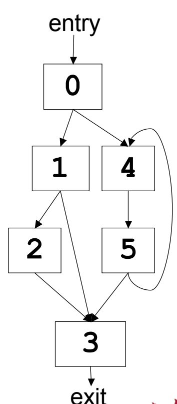
block *a* *dominates* block *b* if every possible execution path from *entry* to *b* includes *a*

entry dominates everything

0 dominates everything but entry

1 dominates and

Dominators are useful in identifying "natural" loops



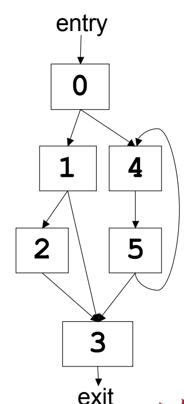
Definitions

a sdom b

If *a* and *b* are different blocks and *a dom b*, we say that *a strictly dominates b*

a idom b

If *a sdom b*, and there is no *c* such that *a sdom c* and *c sdom b*, we say that *a* is the *immediate dominator* of *b*



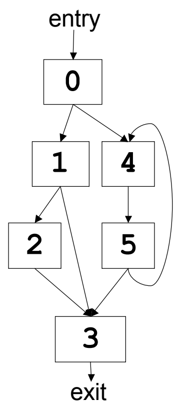
Properties of Dom

Dominance is a partial order on the blocks of the flow graph, i.e.,

1. Reflexivity: a dom a for all a
2. Anti-symmetry: a dom b and b dom a implies a = b
3. Transitivity: a dom b and b dom c implies a dom c

NOTE: there may be blocks a and b such that neither a dom b or b dom a holds.

The dominators of each node n are linearly ordered by the dom relation. The dominators of n appear in this linear order on any path from the initial node to n.



Computing dominators

We want to compute *D*[*n*], the set of blocks that dominate *n*

Initialize each *D*[*n*] (except *D*[entry]) to be the set of all blocks, and then iterate until no *D*[*n*] changes:

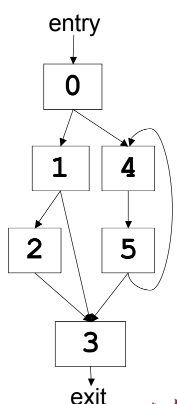
$$D[\text{entry}] = \{\text{entry}\}$$

$$D[n] = \{n\} \cup \left(\bigcap_{p \in \text{pred}(n)} D[p] \right), \quad \text{for } n \neq \text{entry}$$

Carnegie Mellon School of Computer Science

Example

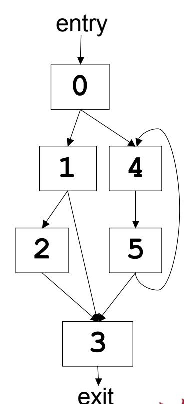
block	Initialization D[n]
entry	{entry}
0	{entry,0,1,2,3,4,5,exit}
1	{entry,0,1,2,3,4,5,exit}
2	{entry,0,1,2,3,4,5,exit}
3	{entry,0,1,2,3,4,5,exit}
4	{entry,0,1,2,3,4,5,exit}
5	{entry,0,1,2,3,4,5,exit}
exit	{entry,0,1,2,3,4,5,exit}



Carnegie Mellon School of Computer Science

Example

block	Initialization D[n]	First Pass D[n]
entry	{entry}	{entry}
0	{entry,0,1,2,3,4,5,exit}	{0,entry}
1	{entry,0,1,2,3,4,5,exit}	{1,0,entry}
2	{entry,0,1,2,3,4,5,exit}	{2,1,0,entry}
3	{entry,0,1,2,3,4,5,exit}	{3,1,0,entry}
4	{entry,0,1,2,3,4,5,exit}	{4,0,entry}
5	{entry,0,1,2,3,4,5,exit}	{5,4,0,entry}
exit	{entry,0,1,2,3,4,5,exit}	{exit,3,1,0,entry}

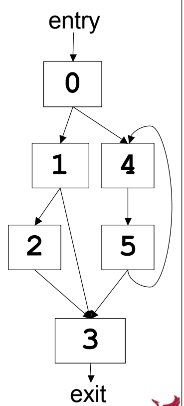


Update rule: $D[n] = \{n\} \cup \left(\bigcap_{p \in pred(n)} D[p] \right)$

Carnegie Mellon School of Computer Science

Example

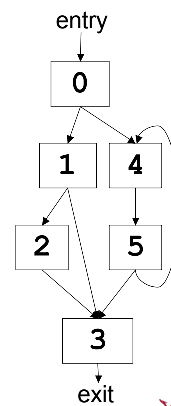
block	First Pass D[n]	Second Pass D[n]
entry	{entry}	{entry}
0	{0,entry}	{0,entry}
1	{1,0,entry}	{1,0,entry}
2	{2,1,0,entry}	{2,1,0,entry}
3	{3,1,0,entry}	{3,0,entry}
4	{4,0,entry}	{4,0,entry}
5	{5,4,0,entry}	{5,4,0,entry}
exit	{exit,3,1,0,entry}	{exit,3,0,entry}



Update rule: $D[n] = \{n\} \cup \left(\bigcap_{p \in pred(n)} D[p] \right)$

Example

block	Second Pass D[n]	Third Pass D[n]
entry	{entry}	{entry}
0	{0,entry}	{0,entry}
1	{1,0,entry}	{1,0,entry}
2	{2,1,0,entry}	{2,1,0,entry}
3	{3,0,entry}	{3,0,entry}
4	{4,0,entry}	{4,0,entry}
5	{5,4,0,entry}	{5,4,0,entry}
exit	{exit,3,0,entry}	{exit,3,0,entry}



Update rule: $D[n] = \{n\} \cup \left(\bigcap_{p \in pred(n)} D[p] \right)$

Complexity

Iterative algorithm (assume bit vector sets)

- cost of set intersection:
- number of intersections in single iteration:
- cost of single iteration:
- number of iterations:
- total complexity:

Complexity: No. of Iterations

The data flow equations for dominator computation form a *rapid** framework

- if nodes are visited in reverse post order the number of iterations is bounded by the *loop-connectedness* of CFG
  - number of back edges that can occur on any acyclic path through G
  - in a *reducible* graph, this is exactly the loop nesting level (typically a small value)
  - if the graph is *irreducible* this is worst case $O(n)$

***Kam, J. B. and Ullman, J. D. 1976. Global Data Flow Analysis and Iterative Algorithms. J. ACM 23, 1 (Jan. 1976), 158-171.**

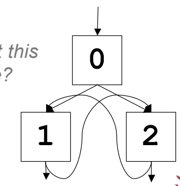
Aside: Reducible flow graphs

Definition: A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, forward edges and back edges, with the following two properties.

1. The forward edges form an acyclic graph in which every node can be reached from the initial node of G.

2. The back edges consist only of edges whose heads dominate their tails.

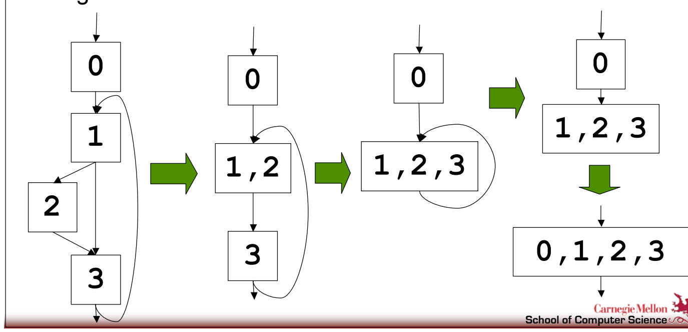
Why isn't this reducible?



This flow graph has no back edges. Thus, it would be reducible if the entire graph were acyclic, which is not the case.

Alternative definition

Definition: A flow graph G is reducible if we can repeatedly collapse (reduce) together blocks (x,y) where x is the only predecessor of y (ignoring self loops) until we are left with a single node



Carnegie Mellon School of Computer Science

Complexity

Iterative algorithm

- $O(n^2e)$ worse case
- O(<loop nest depth>ne) with reducible CFG

More efficient algorithm due to Lengauer and Tarjan

- $O(e{\cdot}\alpha(e,n))$ *inverse Ackermann*
  - more complex, but up to 900x faster than bitset iterative algorithm
  - used in gcc
  - this algorithm has been improved to get better asymptotic behavior*

Improved (very clever) iterative algorithm*

- $O(n+e)$ per an iteration
  - relatively simple to implement
  - on real programs up to 2.5x faster than Lengauer and Tarjan

Adam L. Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R. Westbrook. A new, simpler linear-time dominators algorithm. ACM Transactions on Programming Languages and Systems, 20(6):1265–1296, November 1998.

Timothy J. Harvey, Ph.D. Thesis, Rice University, "An Experimental Analysis of a Set of Compiler Algorithms." (2003)

Carnegie Mellon School of Computer Science

Computing IDOM

Let *sD*[*n*] be the set of blocks that strictly dominate *n*, then

$$sD[n] = D[n] - \{n\}$$

To compute *iD*[*n*], the set of blocks (size <= 1) that immediately dominate *n*

Set

$$iD[n] = sD[n]$$

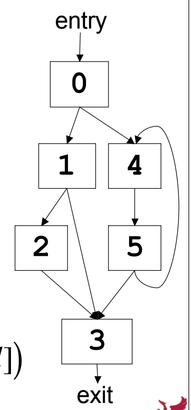
Repeat until no iD[n] changes:

$$iD[n] = iD[n] - \bigcup_{d \in iD[n]} \left(sD[d] \right)$$

Carnegie Mellon School of Computer Science

Example

block	Initialization iD[n]=sD[n]	First Pass iD[n]
entry	{}	{}
0	{entry}	
1	{0,entry}	
2	{1,0,entry}	
3	{0,entry}	
4	{0,entry}	
5	{4,0,entry}	
exit	{3,0,entry}	

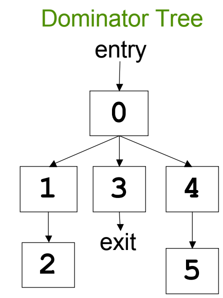


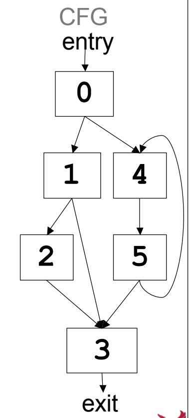
Update rule: $iD[n] = iD[n] - \bigcup_{d \in iD[n]} (sD[d])$

Dominator Tree

In the dominator tree the initial node is the entry block, and the parent of each other node is its immediate dominator.

block	iD[n]
entry	{}
0	{entry}
1	{0}
2	{1}
3	{0}
4	{0}
5	{4}
exit	{3}





Dominance Frontier

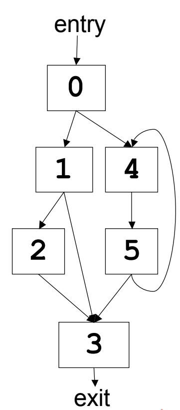
If z is the first node we encounter on the path from x which x does not *strictly* dominate, z is in the dominance frontier of x

For some path from node x to z,
$x \rightarrow \ldots \rightarrow y \rightarrow z$
where x dom y but not x sdom z.

Dominance frontier of **1**?

Dominance frontier of **2**?

Dominance frontier of **4**?



Calculating the Dominance Frontier

- Let *dominates*[n] be the set of all blocks which block n dominates
  - subtree of dominator tree with n as the root
- The dominance frontier of n, *DF*[*n*] is

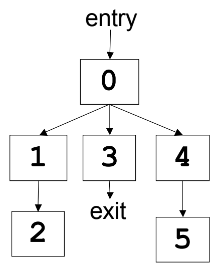
$$DF[n] = \left(\bigcup_{s \in dominates[n]} succs(s) \right) - \left(dominates[n] - \{n\} \right)$$

Example

First calculate dominates[n] from the dominator tree

block	dominates[n]	
entry	{entry,0,1,2,3,4,5,exit}	
0	{0,1,2,3,4,5,exit}	
1	{1,2}	
2	{2}	
3	{3,exit}	
4	{4,5}	
5	{5}	
exit	{exit}	

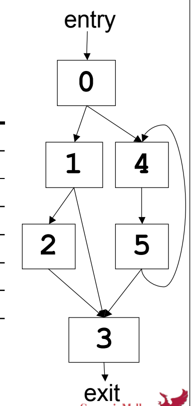
Dominator Tree



Example

Then compute the successor set of dominates[n]

block	dominates[n]	*succ*(dominates[n])
entry	{entry,0,1,2,3,4,5,exit}	
0	{0,1,2,3,4,5,exit}	
1	{1,2}	
2	{2}	
3	{3,exit}	
4	{4,5}	
5	{5}	
exit	{exit}	{}



Carnegie Mellon School of Computer Science

Example

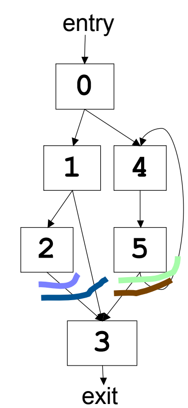
Finally, remove all the blocks from the successor set that are strictly dominated by n to get DF[n]

block	sdominates[n]	*succ*(dominates[n])	DF[n]
entry	{entry,0,1,2,3,4,5,exit}	{0,1,2,3,4,5,exit}	
0	{0,1,2,3,4,5,exit}	{1,2,3,4,5,exit}	
1	{1,2}	{2,3}	
2	{2}	{3}	
3	{3,exit}	{exit}	
4	{4,5}	{3,4,5}	
5	{5}	{3,4}	
exit	{exit}	{}	{}

Carnegie Mellon School of Computer Science

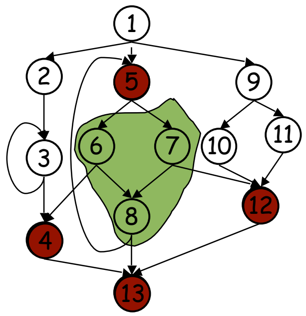
Example

block	DF[n]
entry	{}
0	{}
1	**{3}**
2	**{3}**
3	{}
4	**{3,4}**
5	**{3,4}**
exit	{}

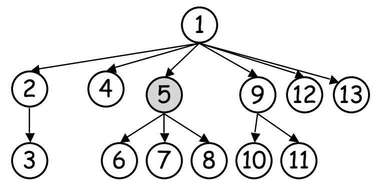


Carnegie Mellon School of Computer Science

Recall: SSA



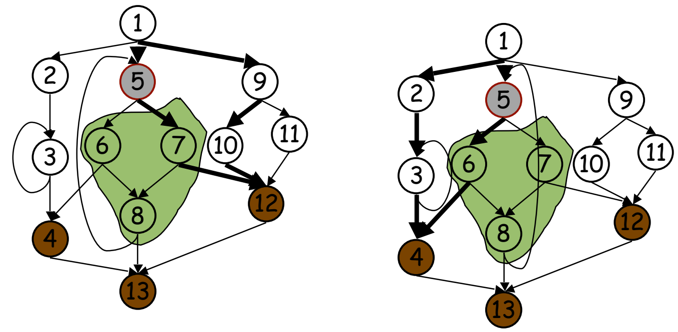
CFG



D-Tree

If there is a def of **a** in block 5, which nodes need a Φ()?

Dominance Frontier & Path-Convergence



Using DF to compute SSA

- Place all Φ()
  - use dominance frontier
  - the arguments to Φ initially unnamed
- Rename all variables
  - a unique def for each use

Using DF to Place Φ()

Gather all the defsites of every variable
Then, for every variable
    foreach defsite
        foreach node in DF(defsite)
            if we haven't put Φ() in node put one in
            If this node didn't define the variable before: add this node to the defsites

This essentially computes the Iterated Dominance Frontier on the fly, inserting the minimal number of phi functions neccesary

Using DF to Place Φ()

```
foreach node n {
  foreach variable v defined in n {
    defsites[v] = defsites[v] ∪ {n}
  }
}
foreach variable v {
    W = defsites[v]
    while W not empty {
      remove n from W
      foreach y in DF[n]
        if y ∉ PHI[v] {
          insert "v ← Φ(v,v,…)" at top of y
          PHI[v] = PHI[v] ∪ {y}
          if v not originally defined in y
            W = W ∪ {y}
      }
    }
  }
```

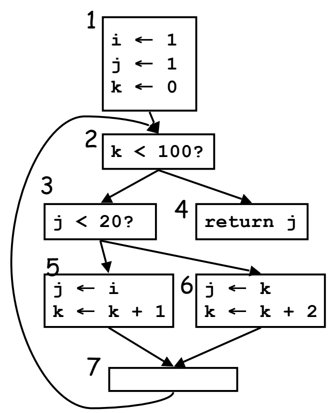

Renaming Variables

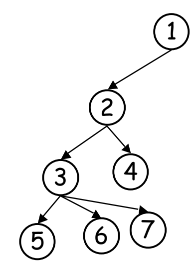
- Walk the D-tree, renaming variables as you go
- Replace uses with more recent renamed def
  - For straight-line code this is easy
  - If there are branches and joins?

Renaming Variables

- Walk the D-tree, renaming variables as you go
- Replace uses with more recent renamed def
  - For straight-line code this is easy
  - If there are branches and joins use the closest def such that the def is above the use in the D-tree
- Easy implementation:
  - for each var: rename (v)
  - rename(v): replace uses with top of stack
    at def: push onto stack
    examine successors for Φ functions
    mark Φ arguments appropriately
    call rename(v) on all children in D-tree
    for each def in this block pop from stack

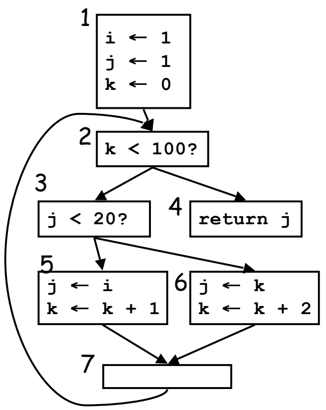
Compute D-tree

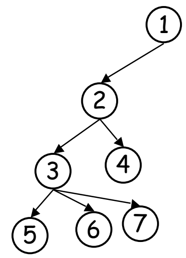




D-tree

Compute Dominance Frontier

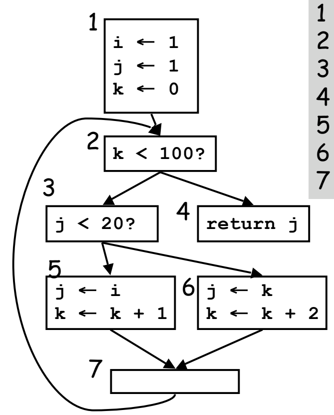




1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

DFs

Insert Φ()



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

defined[n]

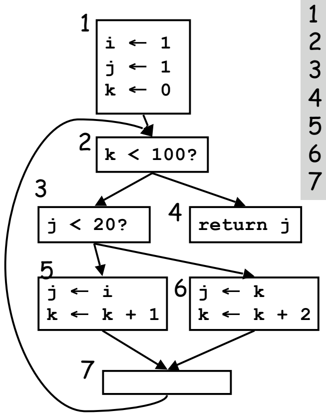
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

var i: W={1}

Insert Φ()



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

defined[n]

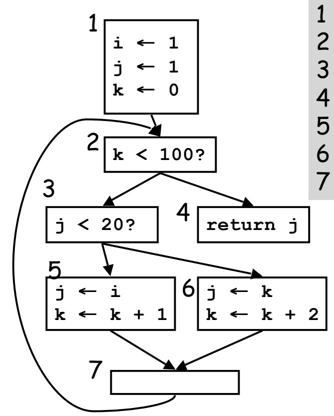
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={1,5,6}

Insert Φ()



DFs

1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

defined[n]

1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

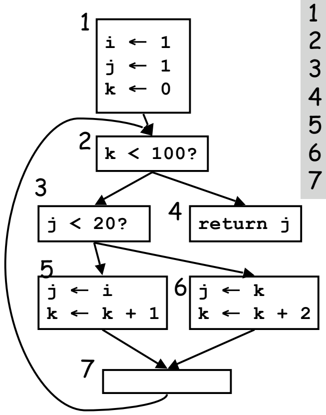
defsites[v]

i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={5,6}

DF{1}

Insert Φ()



DFs	
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

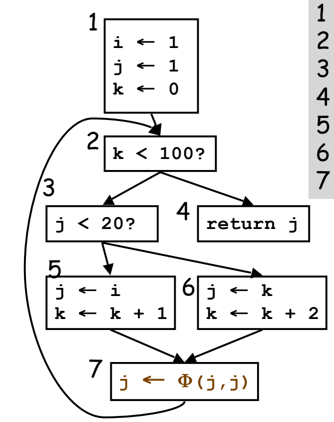
defined[n]	
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={6}

DF{5}

Insert Φ()



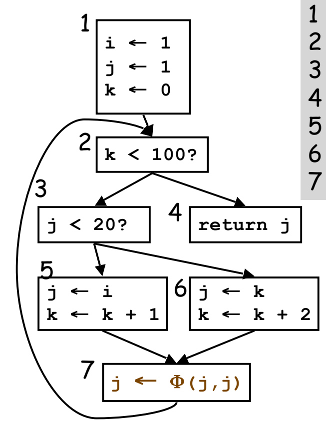
DFs	
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

defined[n]	
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={6,7}

Insert Φ()



DFs	
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

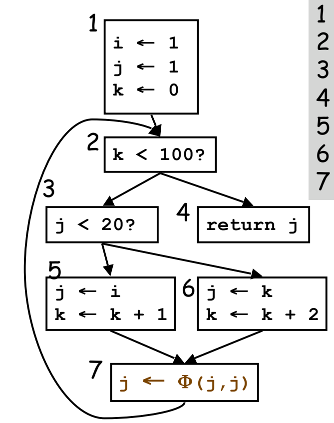
defined[n]	
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={7}

DF{6}

Insert Φ()



DFs	
1	{}
2	{2}
3	{2}
4	{}
5	{7}
6	{7}
7	{2}

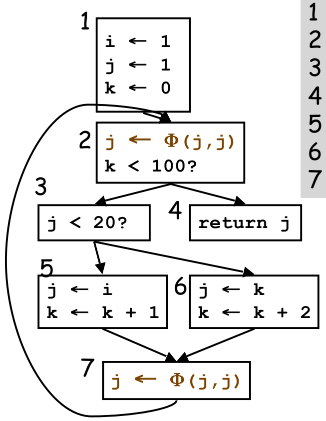
defined[n]	
1	{ i,j,k}
2	{}
3	{}
4	{}
5	{j,k}
6	{j,k}
7	{}

defsites[v]	
i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={}

DF{7}

Insert Φ()



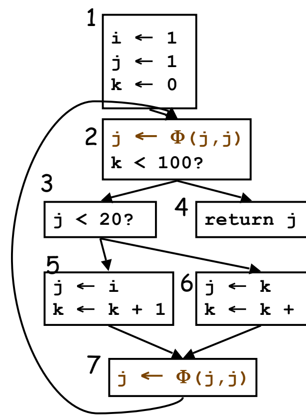
n	DFs	defined[n]
1	{}	{ i,j,k}
2	{2}	{}
3	{2}	{}
4	{}	{}
5	{7}	{j,k}
6	{7}	{j,k}
7	{2}	{}

v	defsites[v]
i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={2}

DF{7}

Insert Φ()



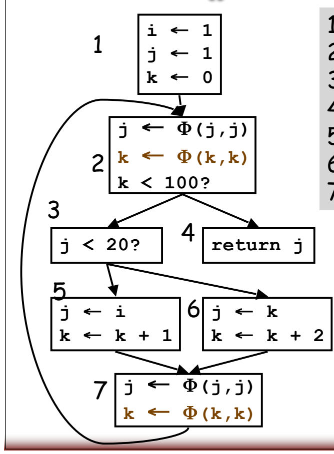
n	DFs	defined[n]
1	{}	{ i,j,k}
2	{2}	{}
3	{2}	{}
4	{}	{}
5	{7}	{j,k}
6	{7}	{j,k}
7	{2}	{}

v	defsites[v]
i	{1}
j	{1,5,6}
k	{1,5,6}

var j: W={}

DF{2}

Insert Φ()

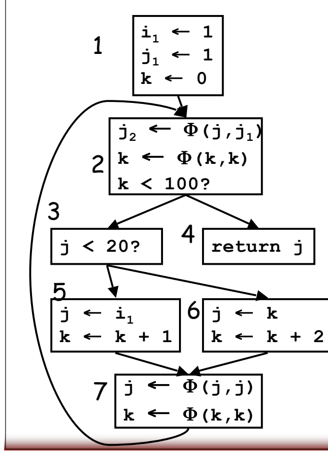


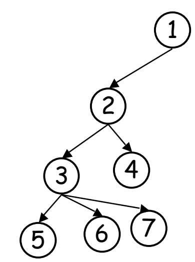
n	DFs	defined[n]
1	{}	{ i,j,k}
2	{2}	{}
3	{2}	{}
4	{}	{}
5	{7}	{j,k}
6	{7}	{j,k}
7	{2}	{}

v	defsites[v]
i	{1}
j	{1,5,6}
k	{1,5,6}

var k: W={1,5,6}

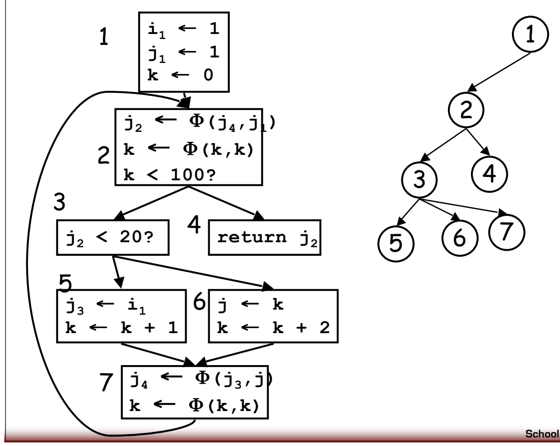
Rename Vars





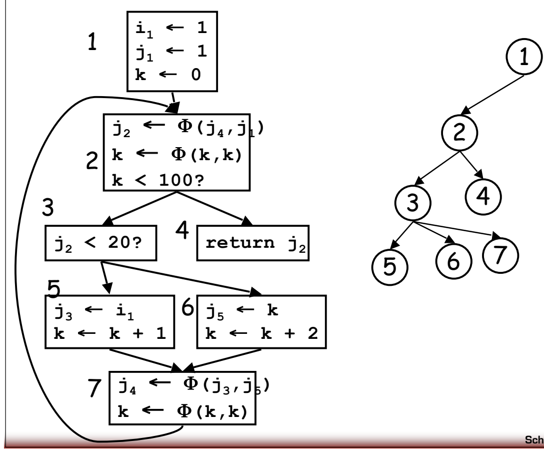
walk the dominator tree

Rename Vars



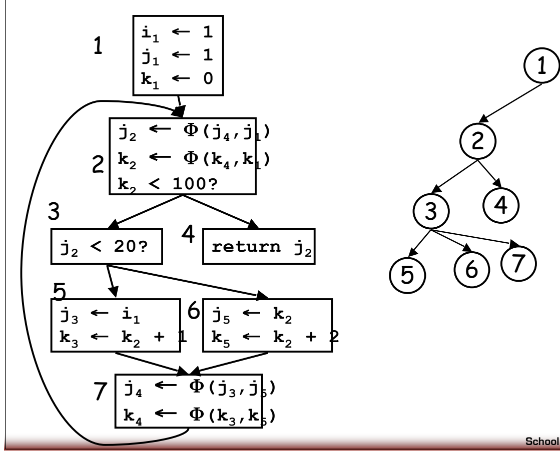
Carnegie Mellon
School of Computer Science

Rename Vars



Carnegie Mellon
School of Computer Science

Rename Vars

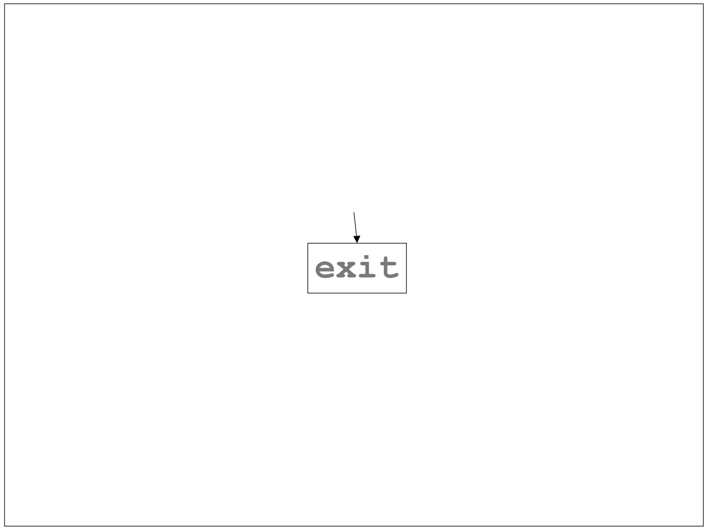


Carnegie Mellon
School of Computer Science

SSA Recap

- SSA is an intermediate representation
- There is a single definition for every use
- Invariant maintained using fake Φ functions
  - an argument for every predecessor at a join
- Conversion to SSA can be very fast
- SSA form makes optimizing easier
  - often linear time
  - no need to analysis

Carnegie Mellon
School of Computer Science


exit