CMPT 379
Compilers

Anoop Sarkar

`http://www.cs.sfu.ca/~anoop`

Code Optimization

- There is no fully optimizing compiler *O*
- Let's assume *O* exists: it takes a program P and produces output **Opt**(P) which is the *smallest* possible
- Imagine a program Q that produces no output and never terminates, then **Opt**(Q) could be:
  `L1: goto L1`
- Then to check if a program P never terminates on some inputs, check if **Opt**(P(i)) is equal to **Opt**(Q) = Solves the Halting Problem
- Full Employment Theorem for Compiler Writers, see Rice(1953)

Optimizations

- Non-Optimizations
- Correctness of optimizations
  - Optimizations must not change the meaning of the program
- Types of optimizations
  - Local optimizations
  - Global dataflow analysis for optimization
  - Static Single Assignment (SSA) Form
- Amdahl's Law

Non-Optimizations

```
enum { GOOD, BAD };
extern int test_condition();

void check() {
 int rc;

 rc = test_condition();
 if (rc != GOOD) {
  exit(rc);
 }
}
```

```
enum { GOOD, BAD };
extern int test_condition();

void check() {
 int rc;

 if ((rc = test_condition())) {
  exit(rc);
 }
}
```

Which version of check runs faster?

Types of Optimizations

- High-level optimizations
  - function inlining
- Machine-dependent optimizations
  - e.g., peephole optimizations, instruction scheduling
- Local optimizations or Transformations
  - within basic block

Types of Optimizations

- Global optimizations or Data flow Analysis
  - across basic blocks
  - within one procedure (*intraprocedural*)
  - whole program (*interprocedural*)
  - pointers (*alias analysis*)

Maintaining Correctness

- What does this program output?

3

Not:

$ decafcc byzero.decaf

Floating exception

```
int main() {
  int x;
  if (false) {
    x = 3/(3-3);
  } else {
    x = 3;
  }
  print_int( x);
}
```

branch delay slot (cf. **load delay** slot)

Peephole Optimization

- Redundant instruction elimination
  - If two instructions perform that same function ***and*** are in the same basic block, remove one
  - Redundant loads and stores

    li $t0, 3

    li $t0, 4
  - Remove unreachable code

    li $t0, 3

    goto L2

    ... (all of this code until next label can be removed)

Peephole Optimization

- Flow control optimization

  goto L1

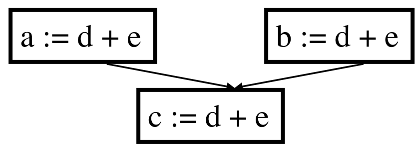
  L1: goto L2
- Algebraic simplification
- Reduction in strength
  - Use faster instructions whenever possible
- Use of Machine Idioms
- Filling delay slots

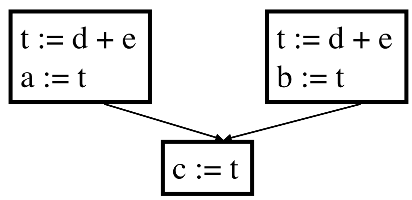
Constant folding & propagation

- Constant folding
  - compute expressions with known values at compile time
- Constant propagation
  - if constant assigned to variable, replace uses of variable with constant unless variable is reassigned

Constant folding & propagation

- Copy Propagation





Transformations

- Structure preserving transformations
- Common subexpression elimination

  a := b + c

  b := a - d

  c := b + c

  d := a - d ($\Rightarrow$ b)

Transformations

- Dead-code elimination (combines copy propogation with removal of unreachable code)

  if (debug) { f(); } /* debug := false (as a constant) */

  if (false) { f(); } /* constant folding */

  using deadcode elimination, code for f() is removed

  x := t3 x := t3

  t4 := x becomes t4 := t3

Transformations

- Renaming temporary variables

  t1 := b+c can be changed to t2 := b+c

  replace all instances of t1 with t2

- Interchange of statements

  t1 := b+c t2 := x+y

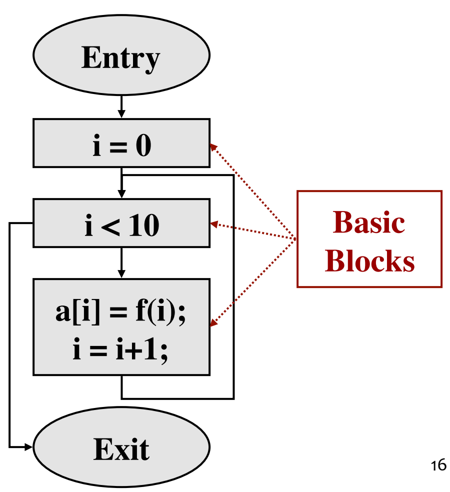
  t2 := x+y can be converted to t1 := b+c

Transformations

- Algebraic transformations
  d := a + 0 (⇒ a)
  d := d * 1 (⇒ *eliminate*)
- Reduction of strength
  d := a ** 2 (⇒ a * a)

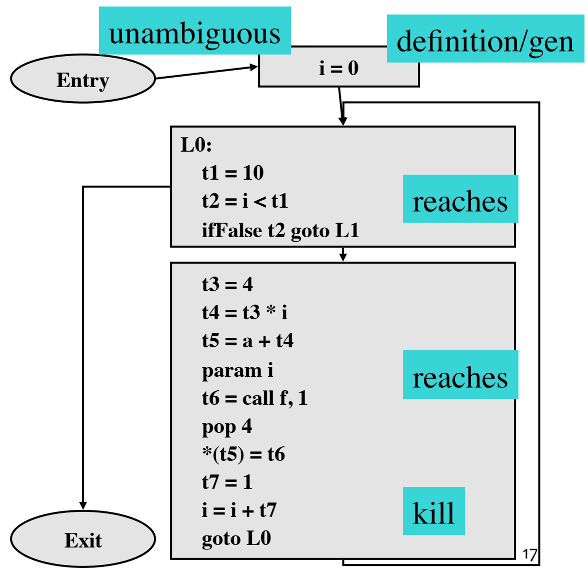
Control Flow Graph (CFG)

```
int main() {
 extern int f(int);
 int i;
 int *a;
 for (i = 0;
        i < 10;
             i = i + 1)
     { a[i] = f(i); }
}
```



Control Flow Graph in TAC

```
main:
  i = 0
L0:
  t1 = 10
  t2 = i < t1
  ifFalse t2 Goto L1
  t3 = 4
  t4 = t3 * i
  t5 = a + t4
  param i
  t6 = call f, 1
  pop 4
  *(t5) = t6
  t7 = 1
  i = i + t7
  goto L0
L1:
  return
```



SSA Form

- *def-use* chains keep track of where variables were defined and where they were used
- Consider the case where each variable has only one definition in the intermediate representation
- One static definition, accessed many times
- Static Single Assignment Form (SSA)

SSA Form

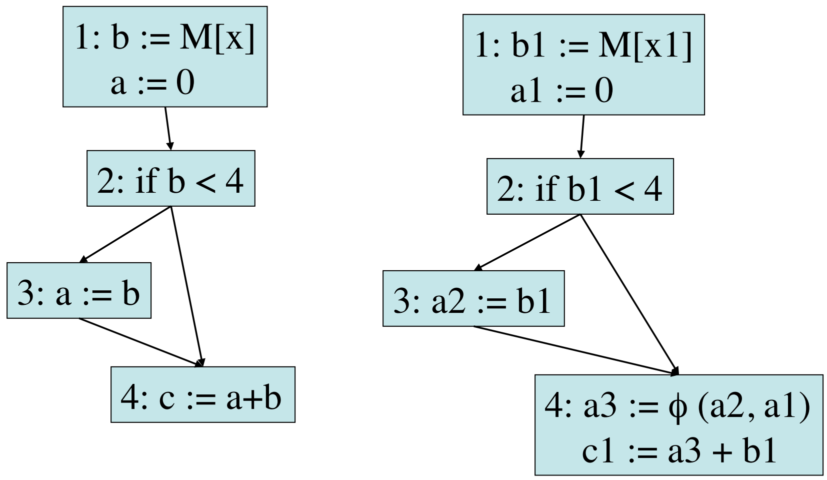
- SSA is useful because
  - Dataflow analysis and optimization is simpler when each variable has only one definition
  - If a variable has N uses and M definitions (which use N+M instructions) it takes N*M to represent def-use chains
  - Complexity is the same for SSA but in practice it is usually linear in number of definitions
  - SSA simplifies the register interference graph

SSA Form

• Original Program	• SSA Form
a := x + y	a1 := x + y
b := a - 1	b1 := a1 - 1
a := y + b	a2 := y + b1
b := x * 4	b2 := x * 4
a := a + b	a3 := a2 + b2

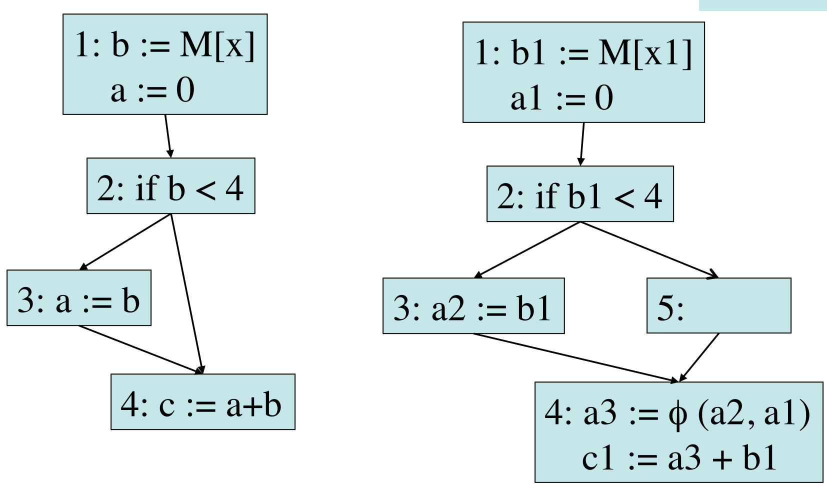
what about conditional branches?

SSA Form



Edge-split SSA Form

Unique Successor & Unique Predecessor



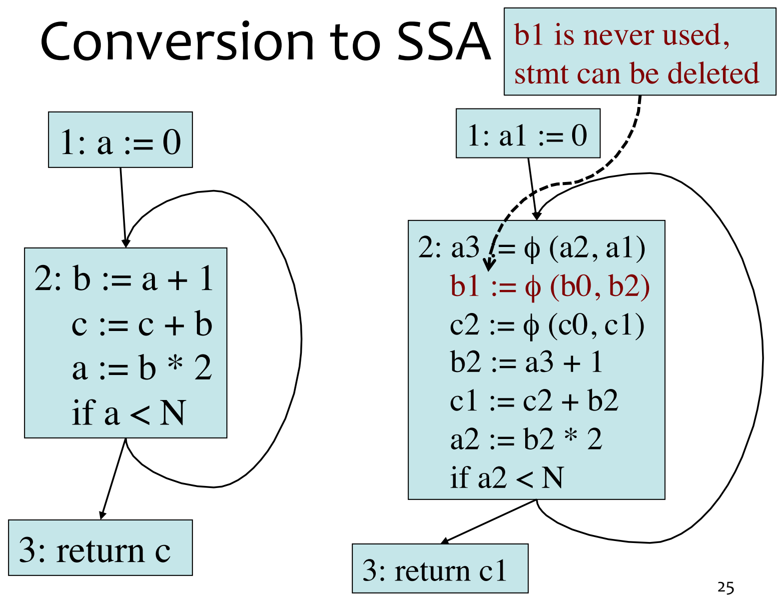
SSA Form

- Conversion from a Control Flow Graph (created from TAC) into SSA Form is not trivial
- SSA creation algorithms:
  - Original algorithm by Cytron et al. 1986
  - Lengauer-Tarjan algorithm (see the Tiger book by Andrew W. Appel for more details)
  - Harel algorithm

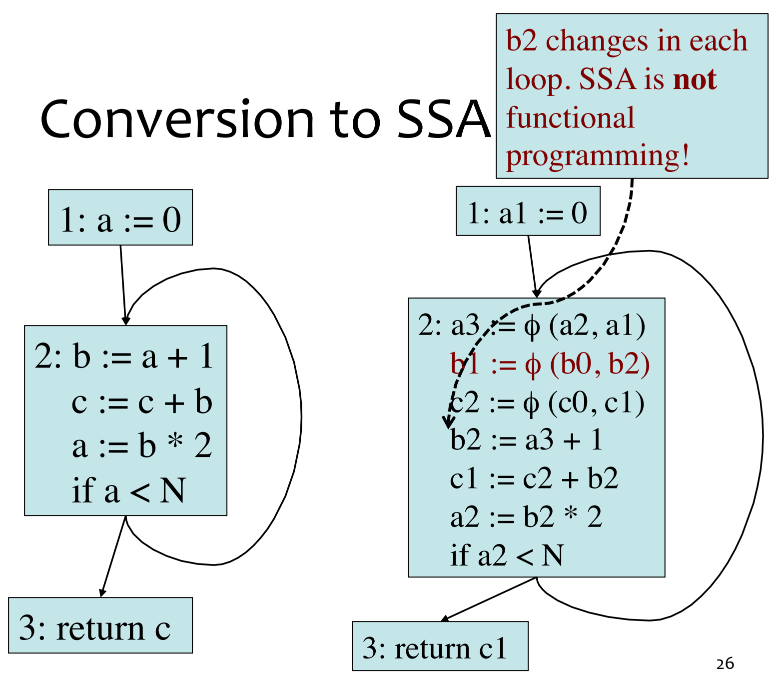
Conversion to SSA Form

- Simple idea: add a ϕ function for every variable at a join point
- A join point is any node in the control-flow graph with more than one predecessor
- But: this is wasteful and unnecessary.

Conversion to SSA



Conversion to SSA

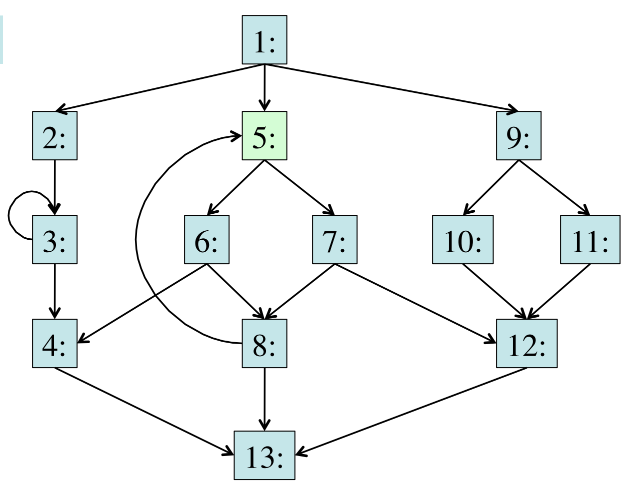


Dominance Relation

- X *dominates* Y if every path from the start node to Y goes through X
- D(X) is the set of nodes that X dominates
- X *strictly dominates* Y if X dominates Y and X ≠ Y

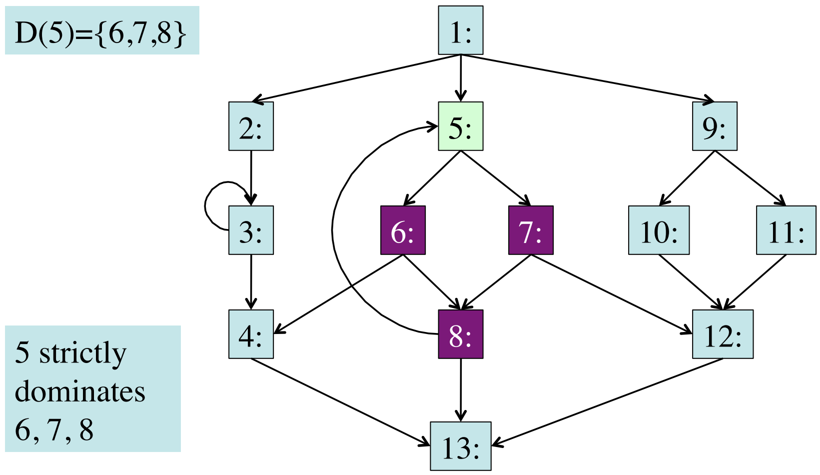
Dominance Relation

D(5)={6,7,8}



5 strictly dominates 6, 7, 8

Dominance Relation



Dominance Property of SSA

- Essential property of SSA form is the definition of a variable must *dominate* use of the variable:
  - If X is used in a ϕ function in block n, then definition of X dominates every predecessor of n
  - If X is used in a non-ϕ statement in block n, then the definition of X dominates n.

Dominance Frontier

- X *strictly dominates* Y if X dominates Y and X ≠ Y
- *Dominance Frontier* (DF) of node X is the set of all nodes Y such that:
  - X dominates a predecessor of Y, AND
  - X does not strictly dominate Y

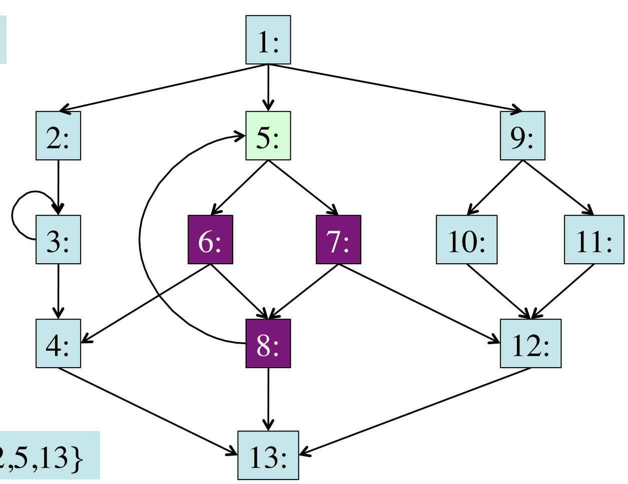
Dominance Frontier

D(5)={6,7,8}

S(6)={4,8}

S(7)={8,12}

S(8)={5,13}



DF(5) = {4,12,5,13}

Dominance Frontier

- Algorithm to compute DF(X):
  - Local(X) := set of successors of X who do not immediately dominate X
  - Up(X) := set of nodes in DF(X) that are not dominated by X's immediate dominator.
  - DF(X) := Union of Local(X) & (Union of Up(K) for all K that are children of X)

Dominance Frontier

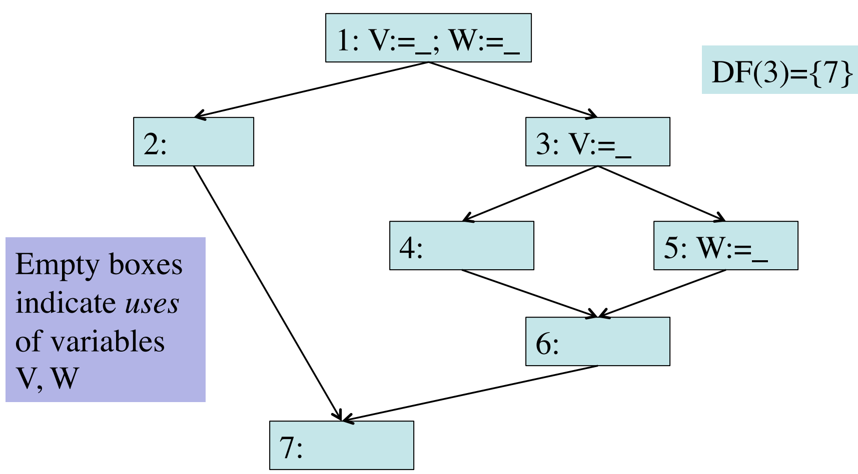
- ComputeDF(X):

```
S := {} // empty set
For each node Y in Successor(X):
  If Y is not immediately dominating X:
    S := S + {Y} // this is Local(X), + means union
For each child K of X in D(X): // X dominates K
  For each element Y in ComputeDF(K):
    If X does not dominate Y,
          S := S + {Y} // this is Up(X)
DF(X) = S
```

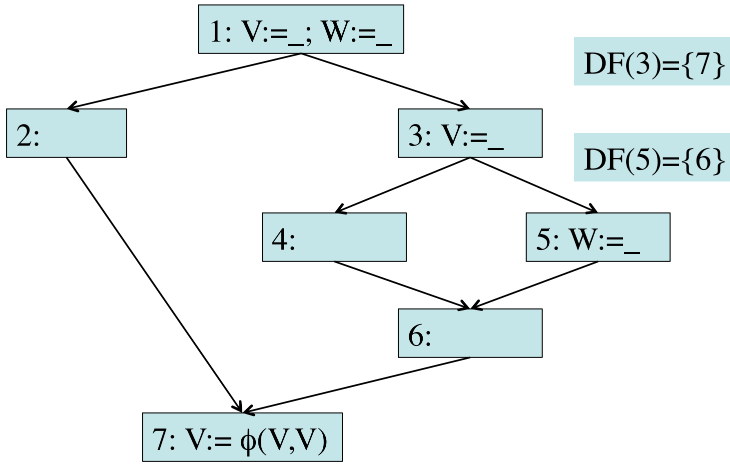
Dominance Frontier

- Dominance Frontier Criterion
  - If node X contains definition of some variable *a*, then any node Y in the DF(X) needs a ϕ function for *a*.
- Iterated Dominance Frontier
  - Since a ϕ function is itself a definition of a new variable, we must iterate the DF criterion until no nodes in the CFG need a ϕ function.

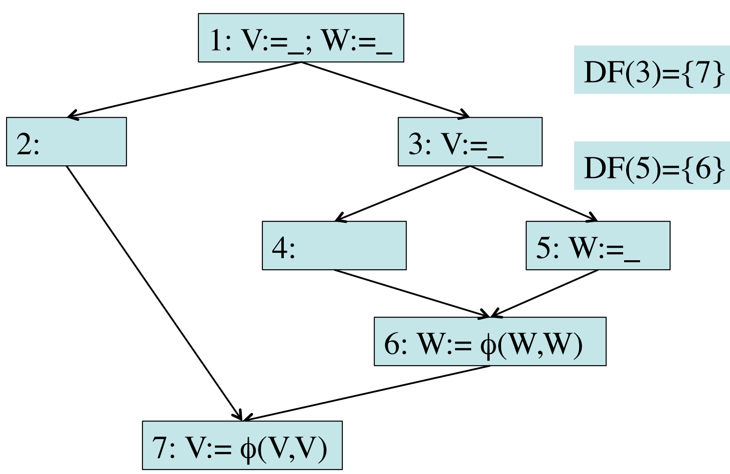
Placing ϕ Functions



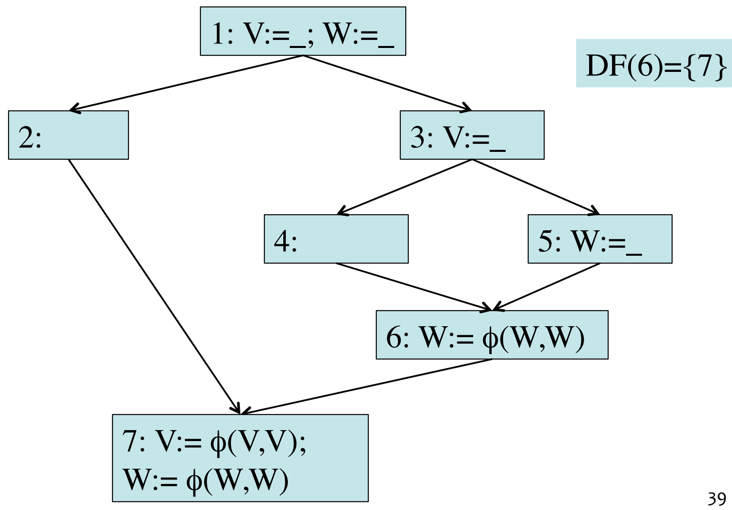
Placing ϕ Functions



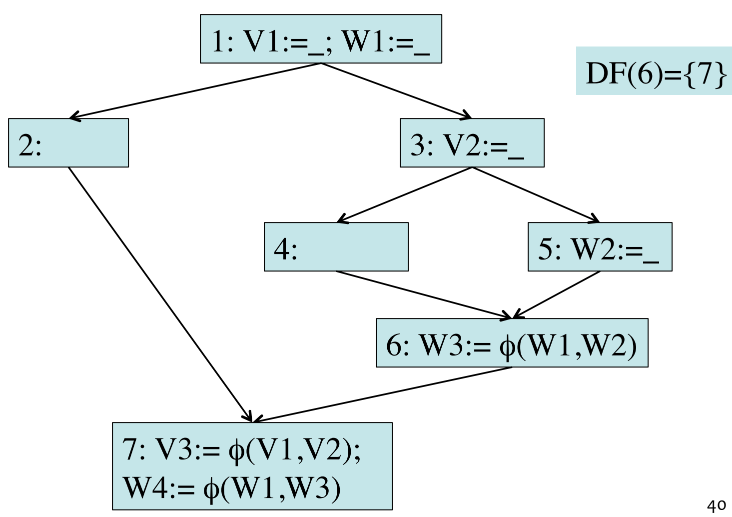
Placing ϕ Functions



Placing ϕ Functions



Rename Variables

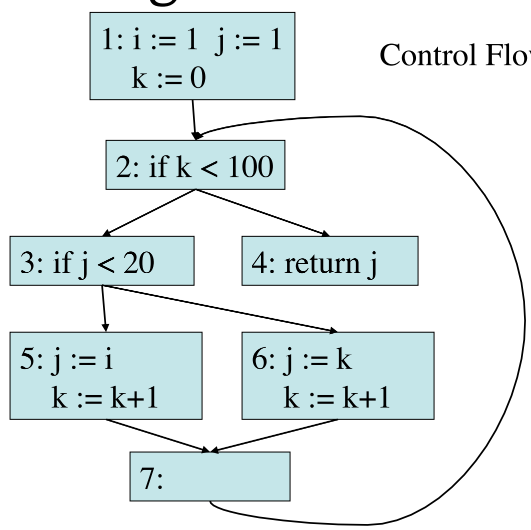


Converting to SSA Form

Program

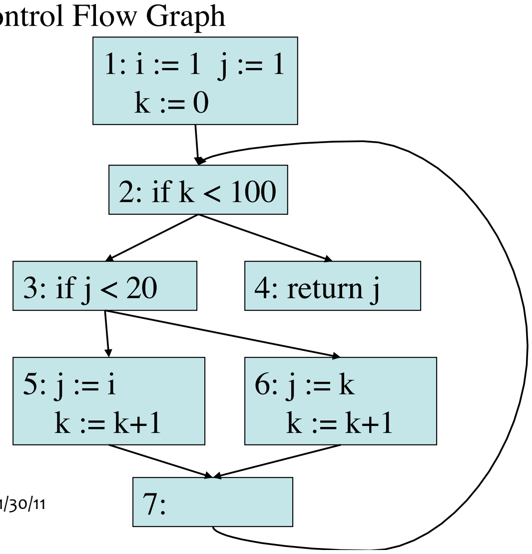
```
i:=1
j:=1
k:=0
while k<100:
    if j < 20:
        j:=i
        k:=k+1
    else:
        j:=k
        k:=k+1
return j
```

Control Flow Graph



Converting to SSA Form

Control Flow Graph



Dominance Relations

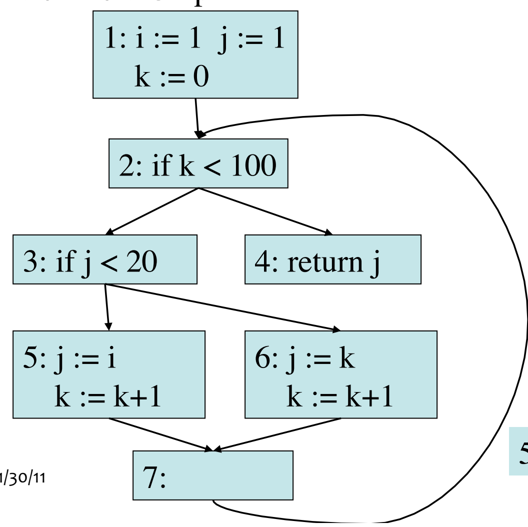
- D(1) = {2,3,4,5,6,7}
- D(2) = {3,4,5,6,7}
- D(3) = {5,6,7}
- D(4) = {}
- D(5) = {}
- D(6) = {}
- D(7) = {}

Converting to SSA

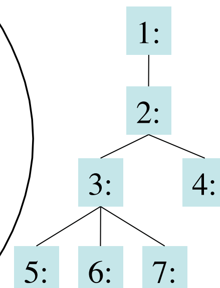
Dominance Relations

- D(1) = {2,3,4,5,6,7}
- D(2) = {3,4,5,6,7}
- D(3) = {5,6,7}
- D(4) = {}
- D(5) = {}
- D(6) = {}
- D(7) = {}

Control Flow Graph



Dominator Tree



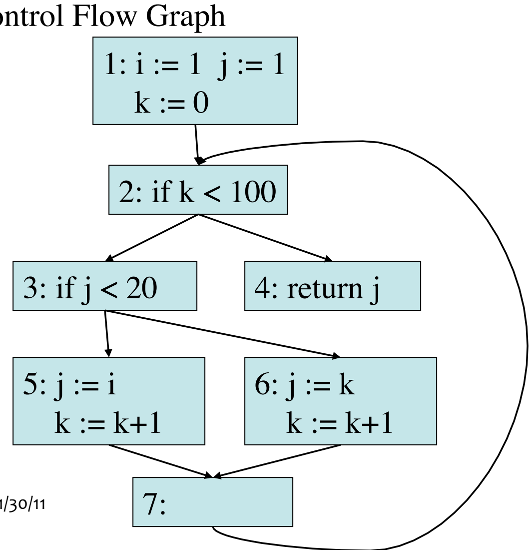
11/30/11 43

Converting to SSA

Dominance Relations

- D(1) = {2,3,4,5,6,7}
- D(2) = {3,4,5,6,7}
- D(3) = {5,6,7}
- D(4) = {}
- D(5) = {}
- D(6) = {}
- D(7) = {}

Control Flow Graph

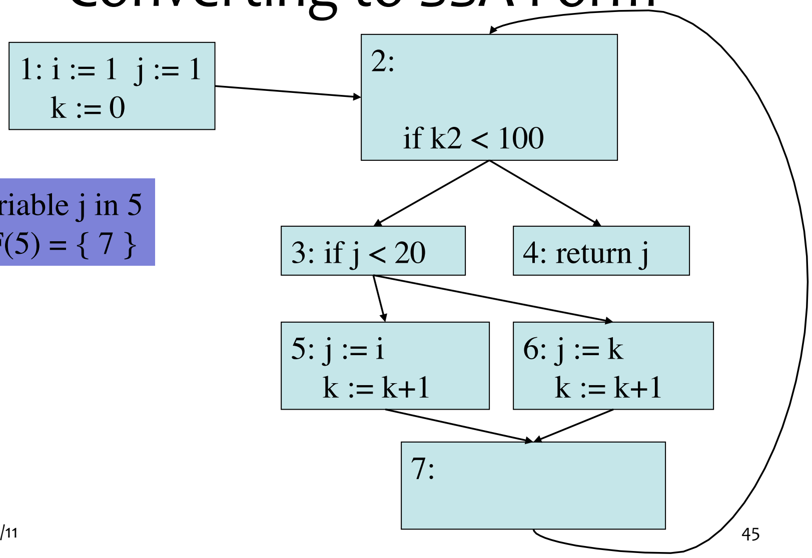


Dominance Frontier

- DF(1) = {}
- DF(2) = {2}
- DF(3) = {2}
- DF(4) = {}
- DF(5) = {7}
- DF(6) = {7}
- DF(7) = {2}

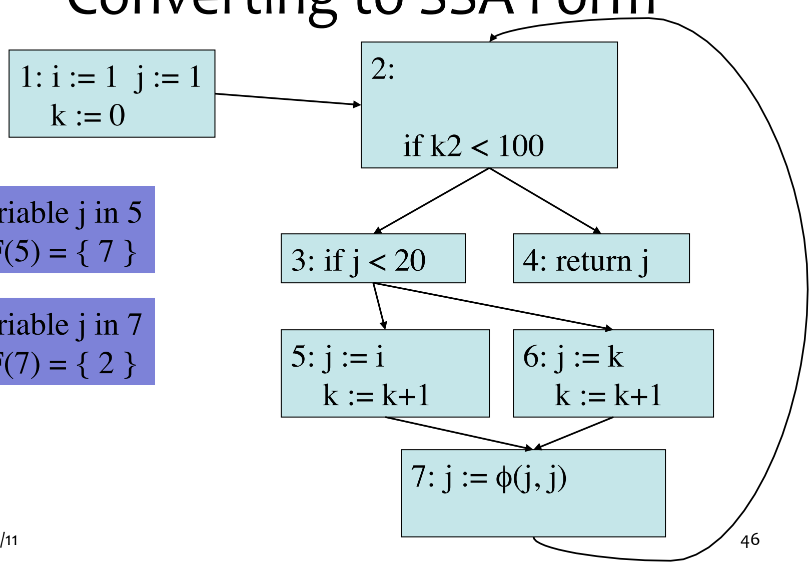
11/30/11 44

Converting to SSA Form



Variable j in 5
DF(5) = { 7 }

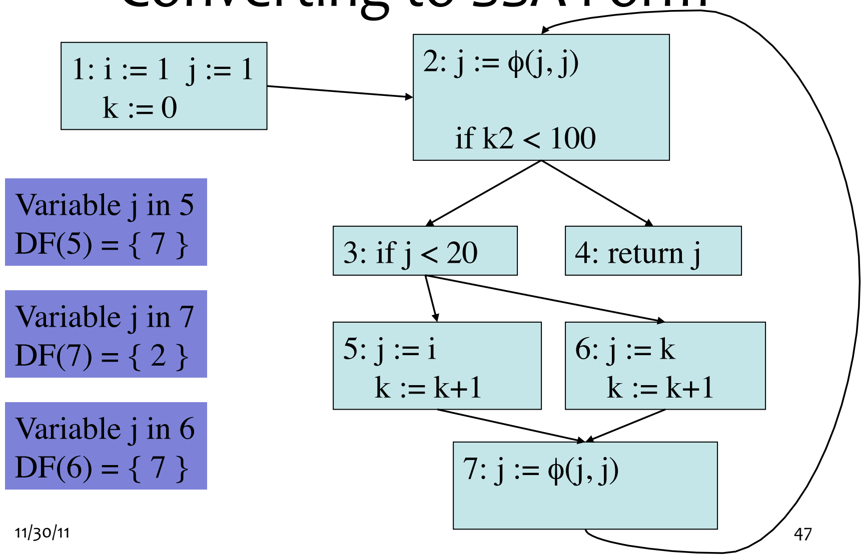
Converting to SSA Form



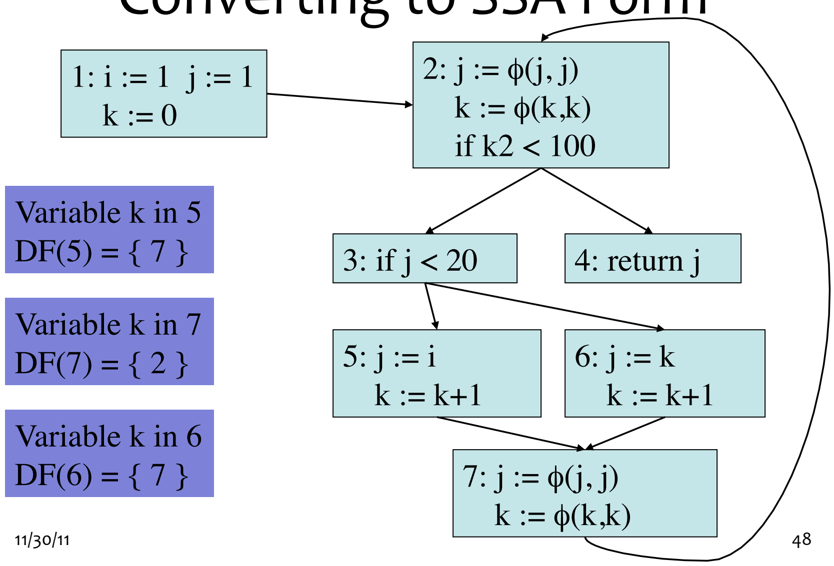
Variable j in 5
DF(5) = { 7 }

Variable j in 7
DF(7) = { 2 }

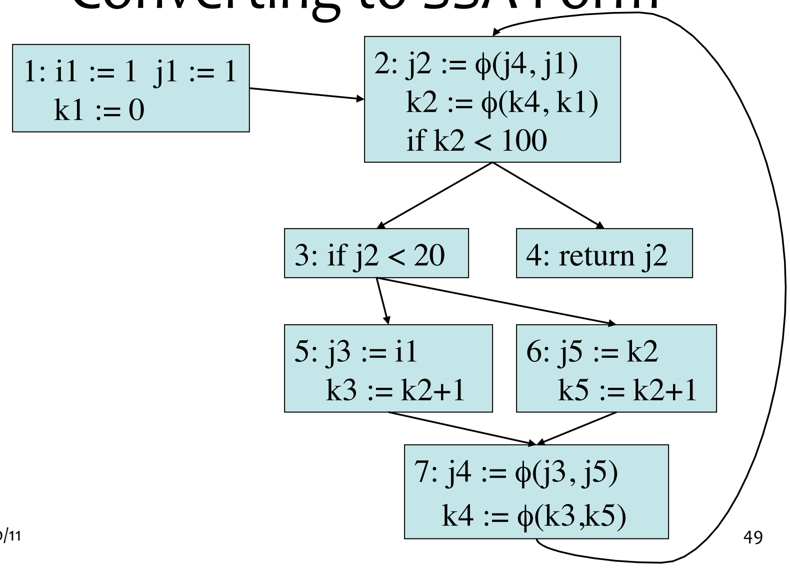
Converting to SSA Form



Converting to SSA Form



Converting to SSA Form



Optimizations using SSA

- SSA form contains *statements*, *basic blocks* and *variables*
- Dead-code elimination
  - if there is a variable *v* with no *uses* and *def* of *v* has no side-effects, delete statement defining *v*
  - if $z := \phi(x, y)$ then eliminate this stmt if no *defs* for *x,y*

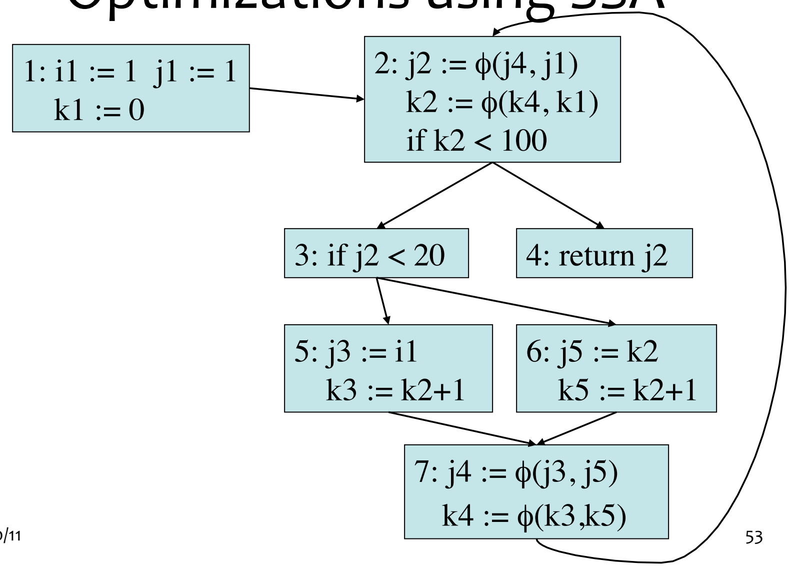
Optimizations using SSA

- Constant Propagation
  - if *v* := *c* for some constant *c* then replace *v* with *c* for all uses of *v*
  - *v* := ϕ (*c1, c2, …, cn*) where all c_i are equal to *c* can be replaced by *v* := *c*

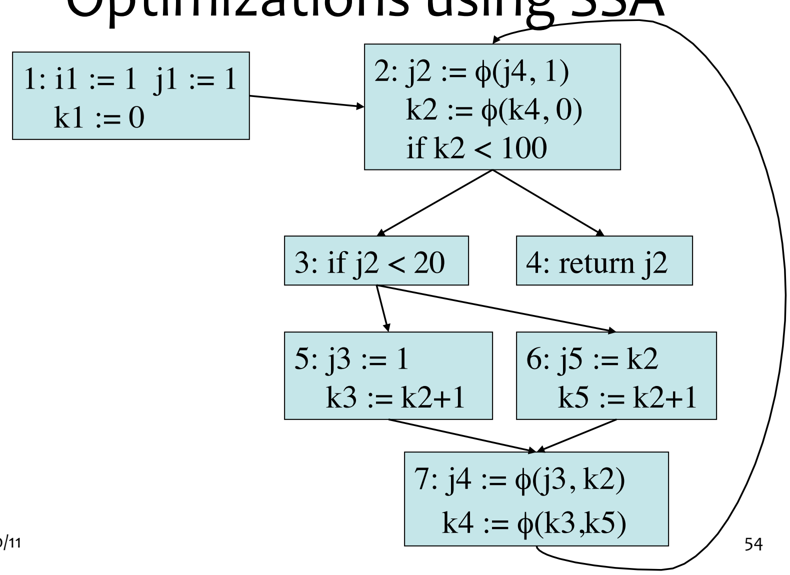
Optimizations using SSA

- Conditional Constant Propagation
  - In previous flow graph, is j always equal to 1?
  - If j = 1 always, then block 6 will never execute and so j := i and j := 1 always
  - If j > 20 then block 6 will execute, and j := k will be executed so that eventually j > 20
  - Which will happen? Using SSA we can find the answer.

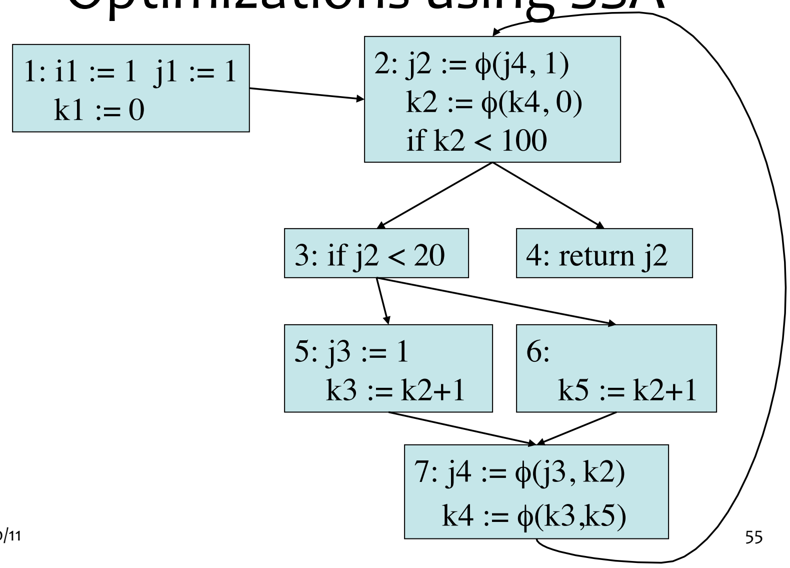
Optimizations using SSA



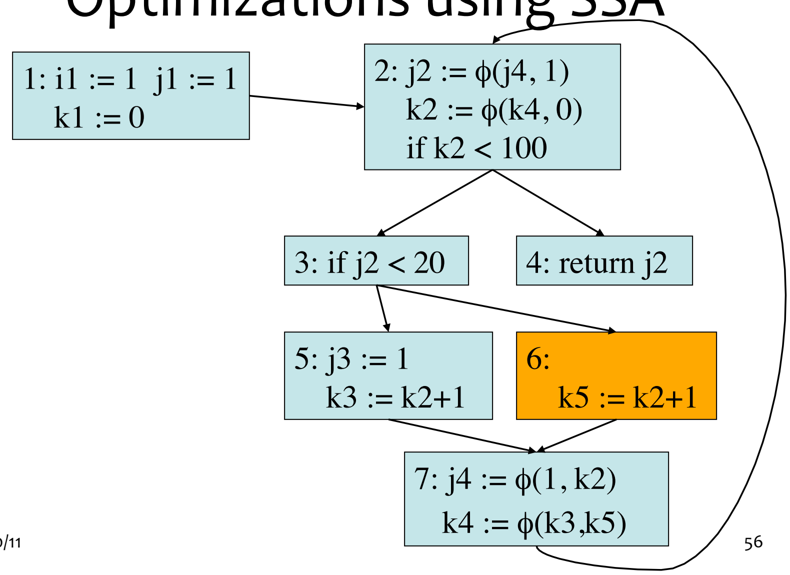
Optimizations using SSA



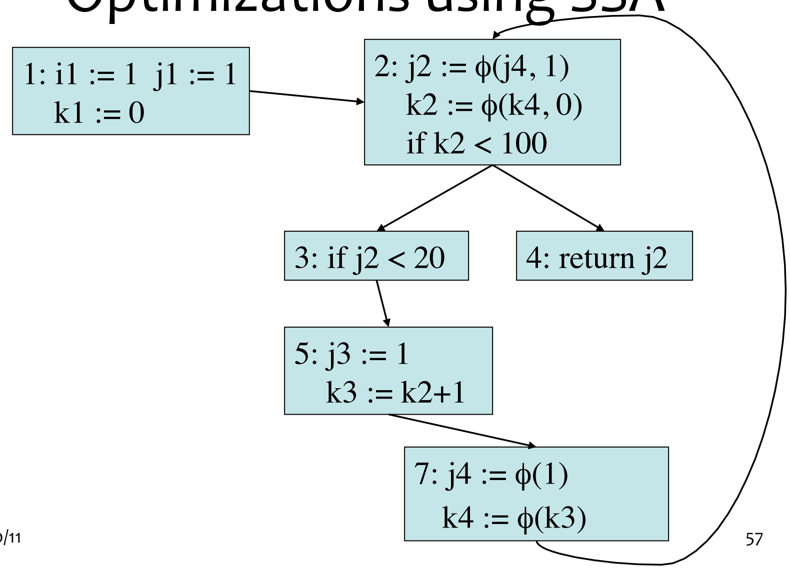
Optimizations using SSA



Optimizations using SSA

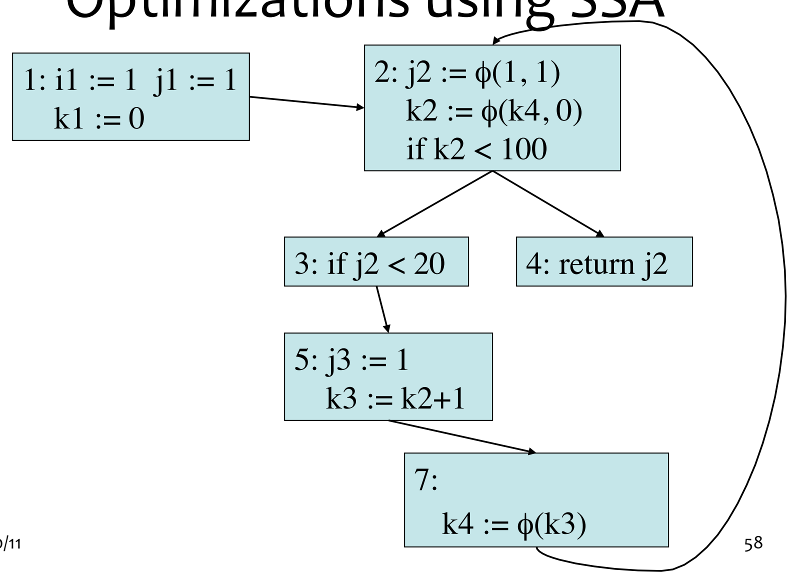


Optimizations using SSA



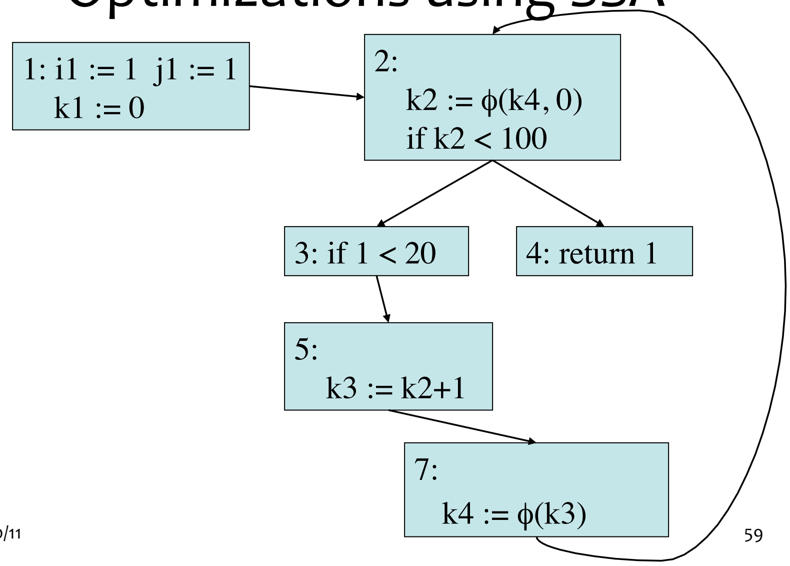
11/30/11 57

Optimizations using SSA



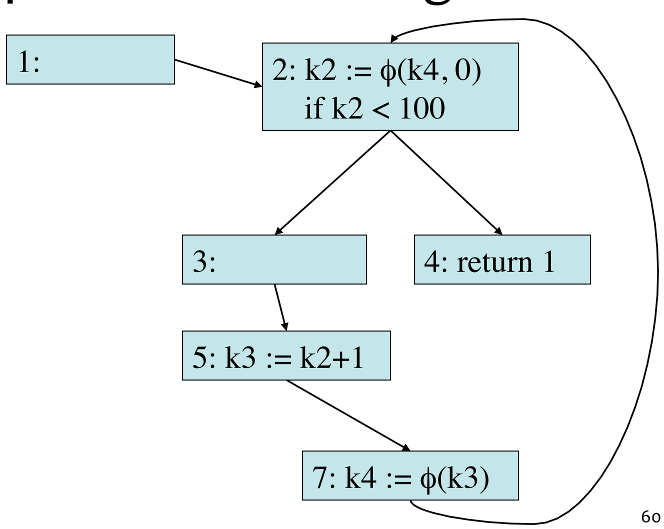
11/30/11 58

Optimizations using SSA



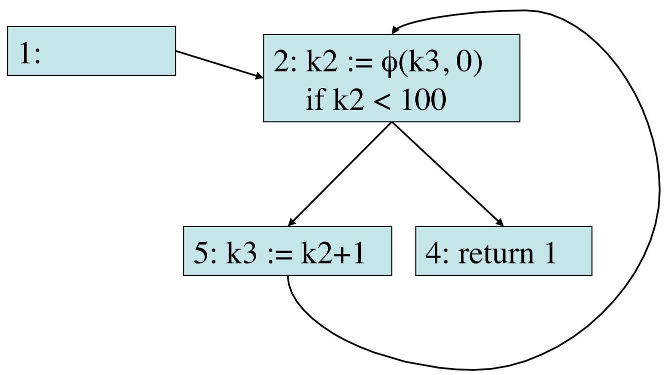
11/30/11 59

Optimizations using SSA



11/30/11 60

Optimizations using SSA



Optimizations using SSA

- Arrays, Pointers and Memory
  - For more complex programs, we need *dependencies*: how does statement B depend on statement A?
  - **Read after write**: A defines variable *v*, then B uses *v*
  - **Write after write**: A defines *v*, then B defines *v*
  - **Write after read**: A uses *v*, then B defines *v*
  - **Control**: A controls whether B executes

Optimizations using SSA

- Memory dependence
  M[i] := 4
  x := M[j]
  M[k] := j
- We cannot tell if *i, j, k* are all the same value which makes any optimization difficult
- Similar problems with Control dependence
- SSA does not offer an easy solution to these problems

More on Optimization

- *Advanced Compiler Design and Implementation* by Steven S. Muchnick

- Control Flow Analysis
- Data Flow Analysis
- Dependence Analysis
- Alias Analysis
- Early Optimizations
- Redundancy Elimination
- Loop Optimizations
- Procedure Optimizations
- Code Scheduling (pipelining)
- Low-level Optimizations
- Interprocedural Analysis
- Memory Hierarchy

Amdahl's Law

- $\text{Speedup}_{\text{total}} = ((1 - \text{Time}_{\text{Fractionoptimized}}) + \text{Time}_{\text{Fractionoptimized}} / \text{Speedup}_{\text{optimized}})^{-1}$
- Optimize the common case, 90/10 rule
- Requires quantitative approach
  - Profiling + Benchmarking
- Problem: Compiler writer doesn't know the application beforehand

Summary

- Optimizations can improve speed, while maintaining correctness
- Various early optimization steps
- Static Single-Assignment Form (SSA)
- Optimization using SSA Form