CMPT 755
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)
- Full Employment Theorem for Compiler Writers, see Rice(1953)

Optimizations

- Non-Optimizations
- Types of optimizations
- Correctness of optimizations
  - Optimizations must not change the meaning of the program
- Amdahl’s Law
- Moore’s Law

Non-Optimizations

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

void check() {
  int rc;

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

```c
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
- 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

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

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

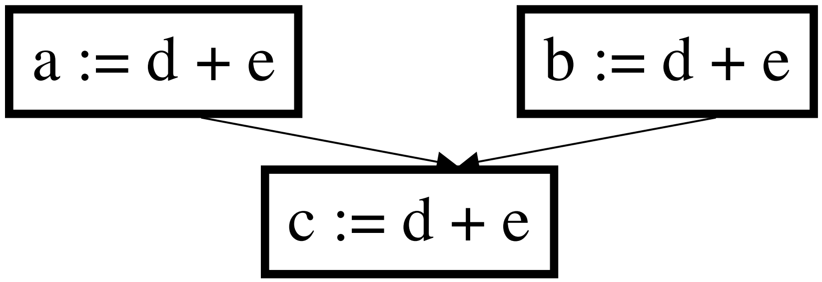
  L1: goto L2
- Algebraic simplification
- Reduction in strength
  - Use faster instructions whenever possible
- Use of Machine Idioms
- Filling delay slots

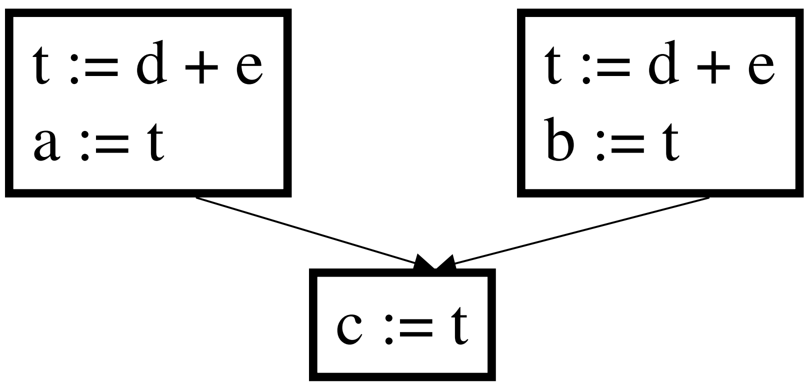
Constant folding & propagation

- 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

  $a := b + c$

  $b := a - d$

  $c := b + c$

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

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 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 ($\Rightarrow$ a)

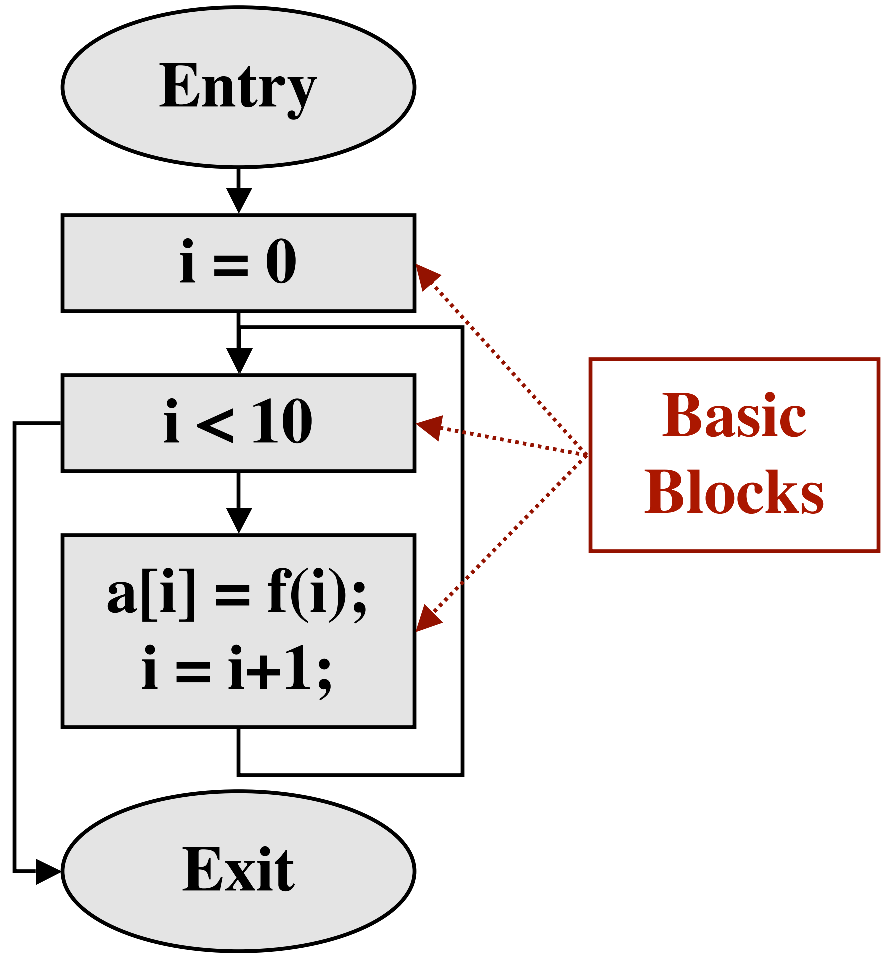
  d := d * 1 ($\Rightarrow$ *eliminate*)

- Reduction of strength

  d := a ** 2 ($\Rightarrow$ a * a)

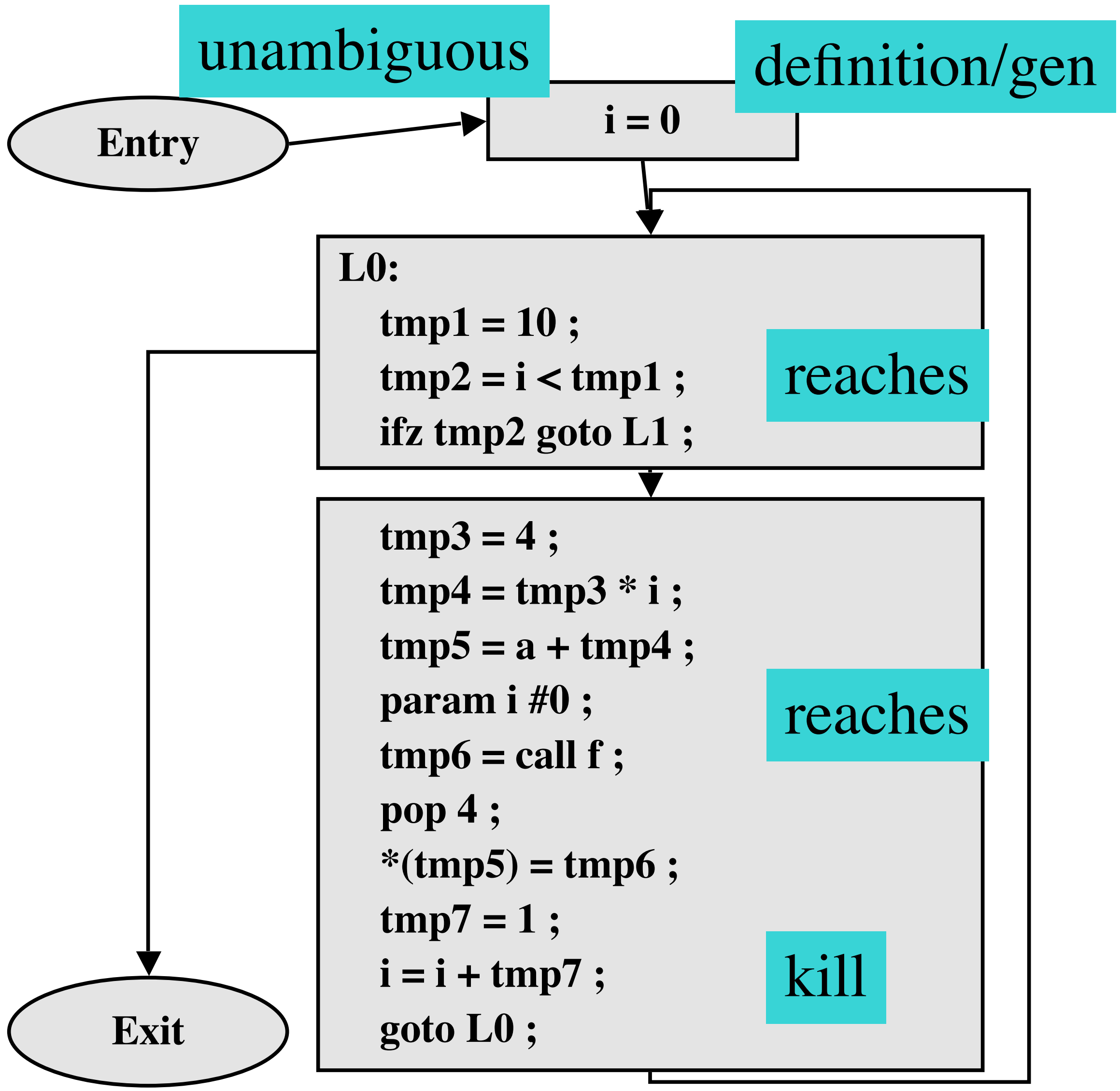
Control Flow Graph (CFG)

```
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

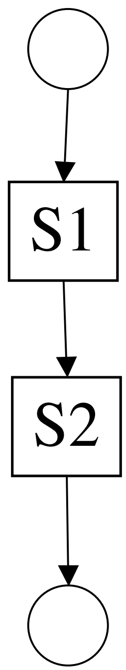
```
main:
   BeginFunc 72 ;
   i = 0 ;
L0:
   tmp1 = 10 ;
   tmp2 = i < tmp1 ;
   IfZ tmp2 Goto L1 ;
   tmp3 = 4 ;
   tmp4 = tmp3 * i ;
   tmp5 = a + tmp4 ;
   param i #0 ;
   tmp6 = call f ;
   pop 4 ;
   *(tmp5) = tmp6 ;
   tmp7 = 1 ;
   i = i + tmp7 ;
   goto L0 ;
L1:
   EndFunc ;
```



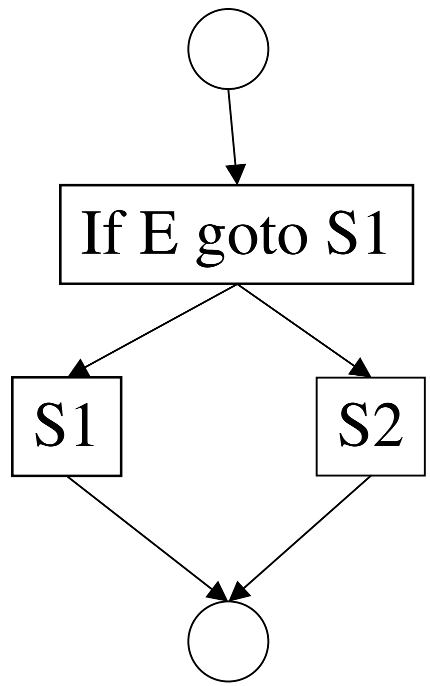
Dataflow Analysis

- $S \rightarrow id := E$
- $S \rightarrow S ; S$
- $S \rightarrow$ if E then S else S
- $S \rightarrow$ do S while E
- $E \rightarrow id + id$
- $E \rightarrow id$

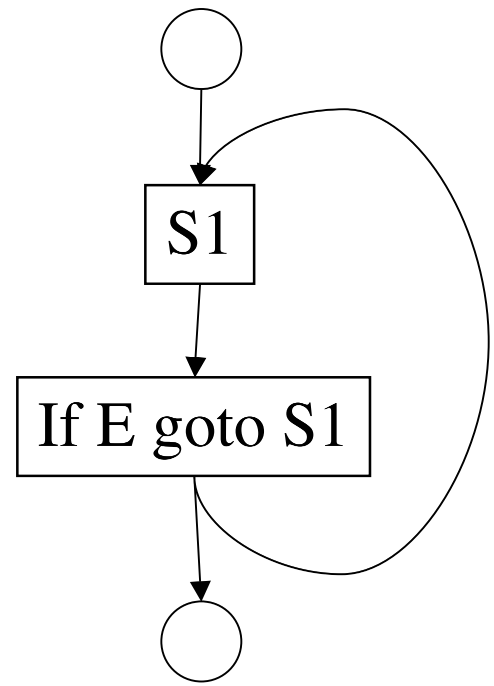
Dataflow Analysis



S ; S

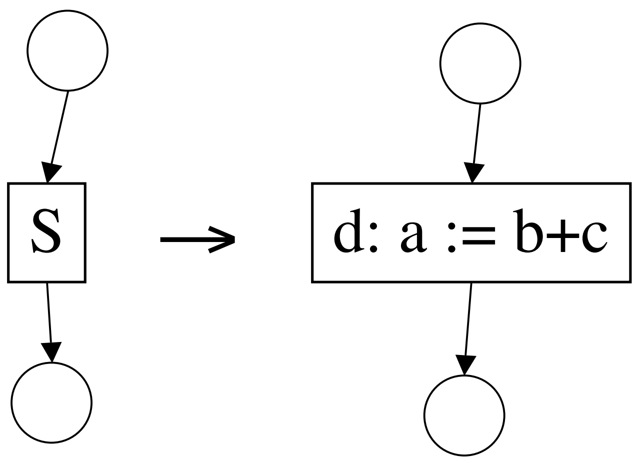


if E then S else S



do S while E

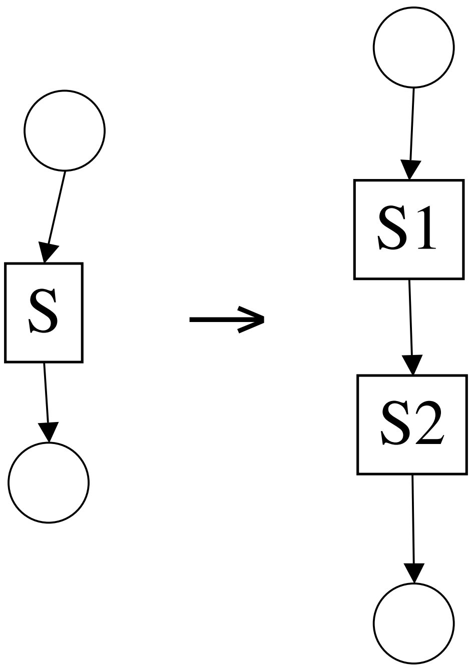
Reaching definitions



gen[S] = { d }
kill[S] = def(a) - { d }

out[S] = gen[S] ∪ (in[S] - kill[S])

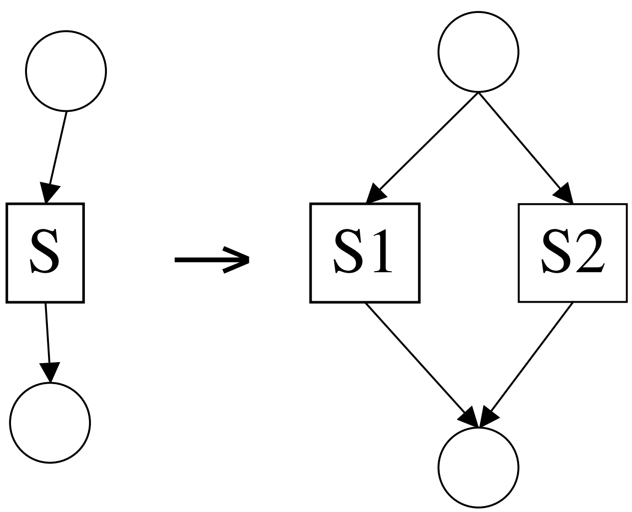
Reaching definitions



$$gen[S] = gen[S2] \cup (gen[S1] - kill[S2])$$
$$kill[S] = kill[S2] \cup (kill[S1] - gen[S2])$$

$$in[S1] = in[S]$$
$$in[S2] = out[S1]$$
$$out[S] = out[S2]$$

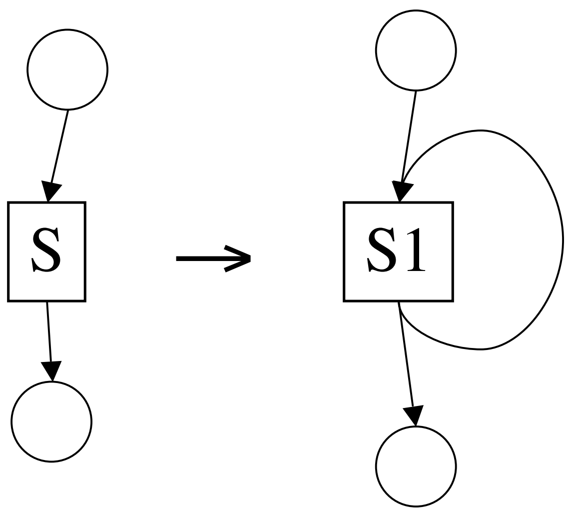
Reaching definitions



gen[S] = gen[S1] $\cup$ gen[S2]
kill[S] = kill[S1] $\cap$ (kill[S1] - gen[S2])

in[S1] = in[S]
in[S2] = in[S]
out[S] = out[S1] $\cup$ out[S2]

Reaching definitions



gen[S] = gen[S1]
kill[S] = kill[S1]

in[S1] = in[S] ∪ gen[S1]
out[S] = out[S1]

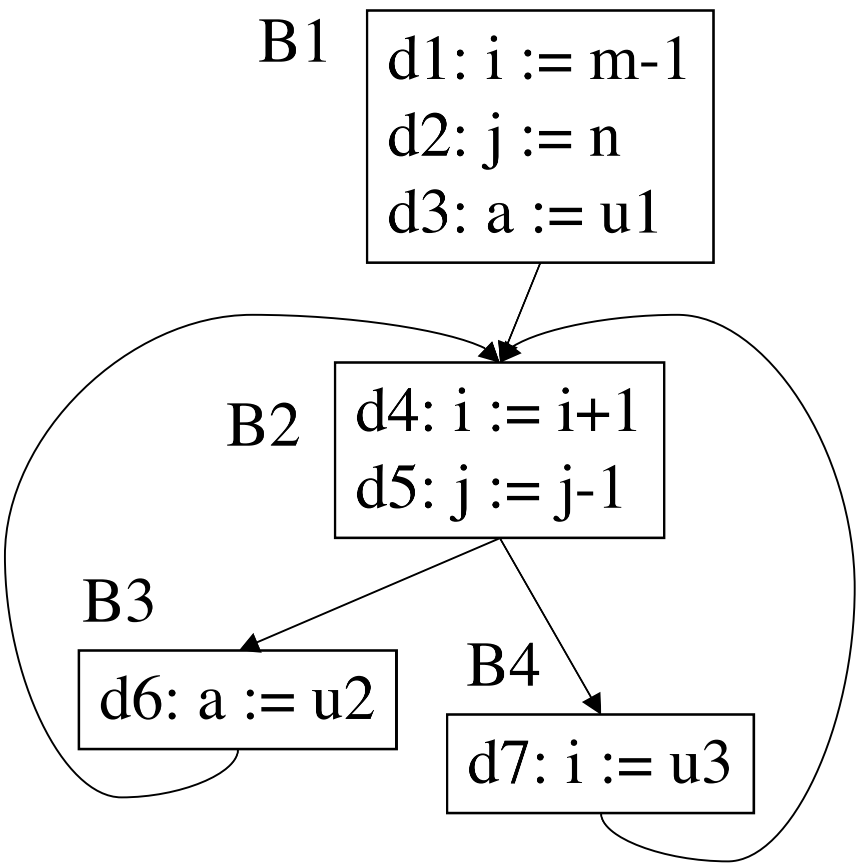
Iteratively find out[S] (fixed point)

out = synthesized attribute

in = inherited attribute

out[S1] = gen[S1] ∪ (in[S1] - kill[S1])

Reaching definitions



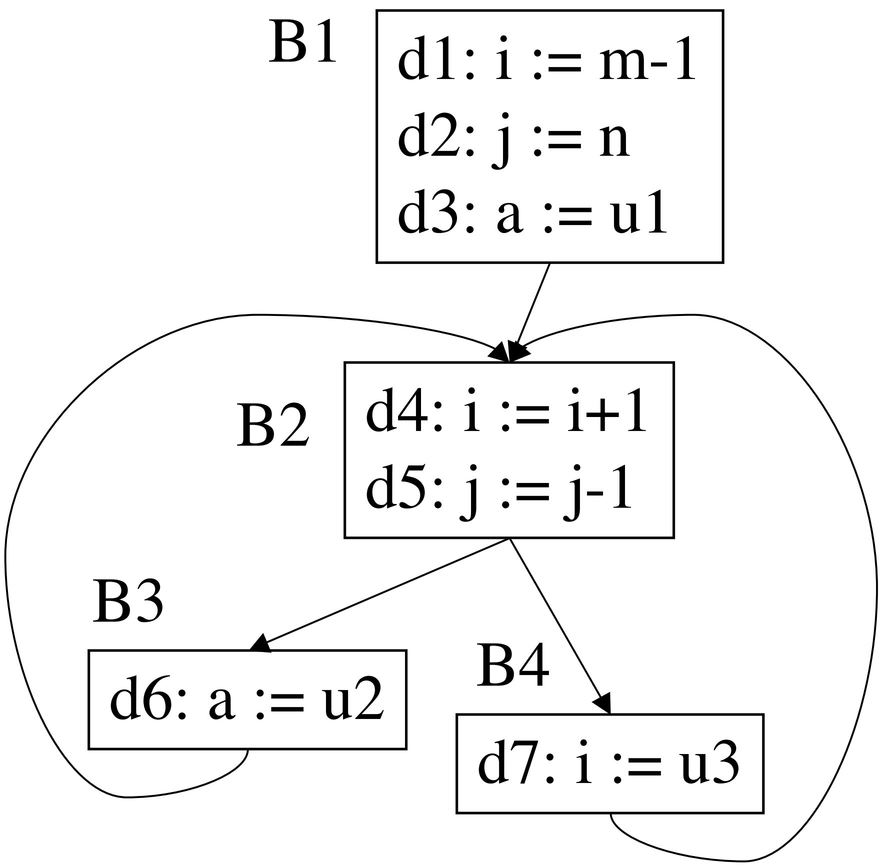
gen[B1] = { d1, d2, d3 }
kill[B1] = { d4, d5, d6, d7 }

gen[B2] = { d4, d5 }
kill[B2] = { d1, d2, d7 }

gen[B3] = { d6 }
kill[B3] = { d3 }

gen[B4] = { d7 }
kill[B4] = { d1, d4 }

Reaching definitions



gen[B1] = { d1, d2, d3 }
kill[B1] = { d4, d5, d6, d7 }

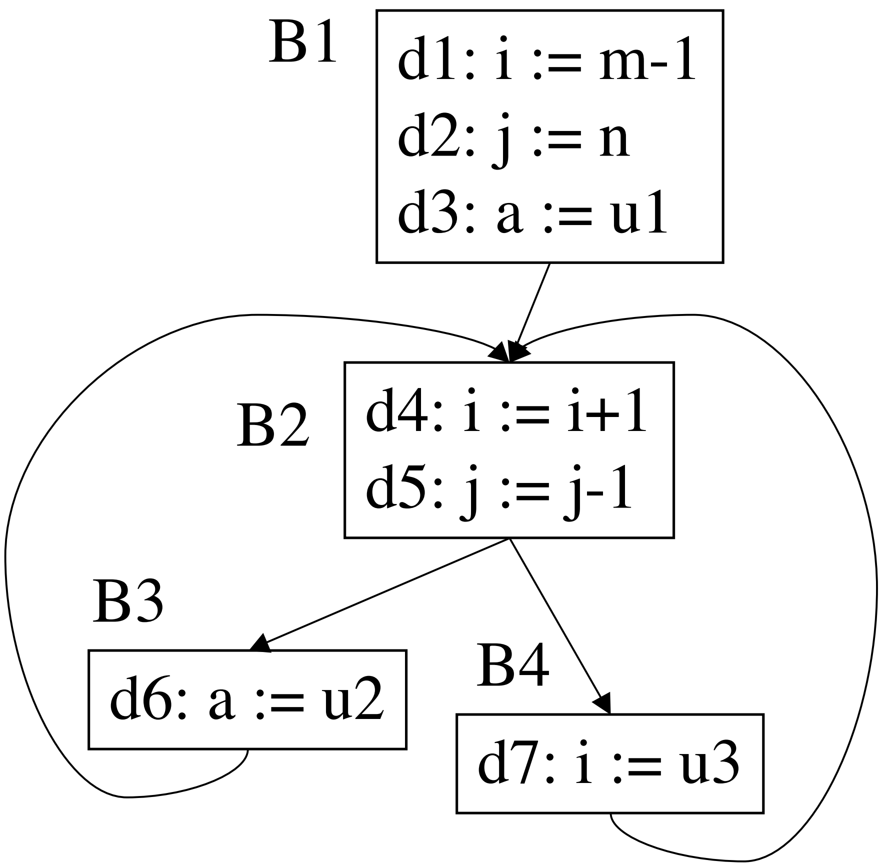
gen[B2] = { d4, d5 }
kill[B2] = { d1, d2, d7 }

gen[B3] = { d6 }
kill[B3] = { d3 }

gen[B4] = { d7 }
kill[B4] = { d1, d4 }

in[B2] = out[B1] ∪ out[B3] ∪ out[B4]

Reaching definitions



gen[B1] = { d1, d2, d3 }
kill[B1] = { d4, d5, d6, d7 }

gen[B2] = { d4, d5 }
kill[B2] = { d1, d2, d7 }

gen[B3] = { d6 }
kill[B3] = { d3 }

gen[B4] = { d7 }
kill[B4] = { d1, d4 }

√ out[B2] = gen[B2] ∪ (in[B3] - kill[B2])
out[B2] = gen[B2] ∪ (in[B4] - kill[B2])

Dataflow Analysis

- Compute Dataflow Equations over Control Flow Graph
  - Reaching Definitions (Forward)

    out[BB] := gen[BB] ∪ (in[BB] – kill[BB])

    in[BB] := ∪ out[s] : forall s ∈ pred[BB]
  - Liveness Analysis (Backward)

    in[BB] := use[BB] ∪ (out[BB] – def[BB])

    out[BB] := ∪ in[s] : forall s ∈ succ[BB]
- Computation by fixed-point analysis

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

a := x + y

b := a - 1

a := y + b

b := x * 4

a := a + b

- SSA Form

a1 := x + y

b1 := a1 - 1

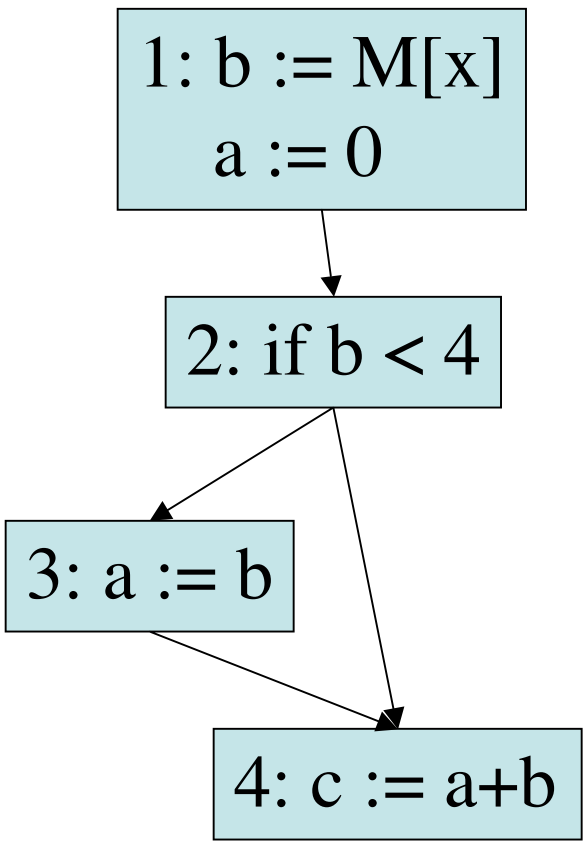
a2 := y + b1

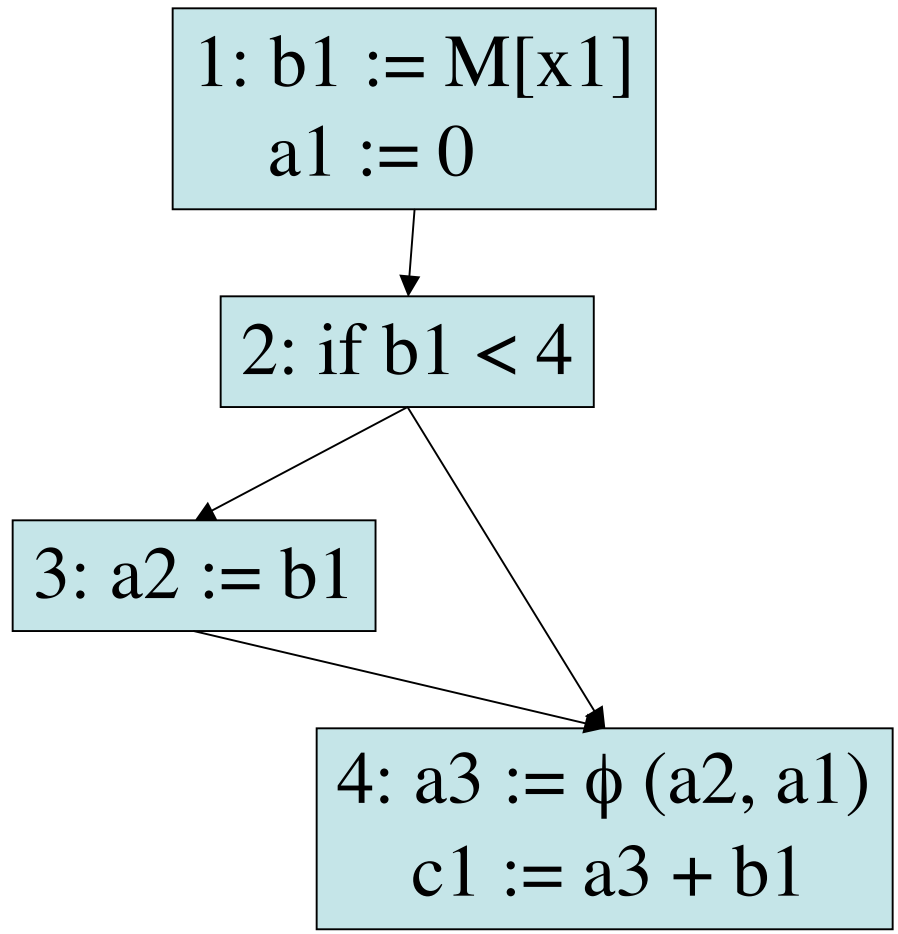
b2 := x * 4

a3 := a2 + b2

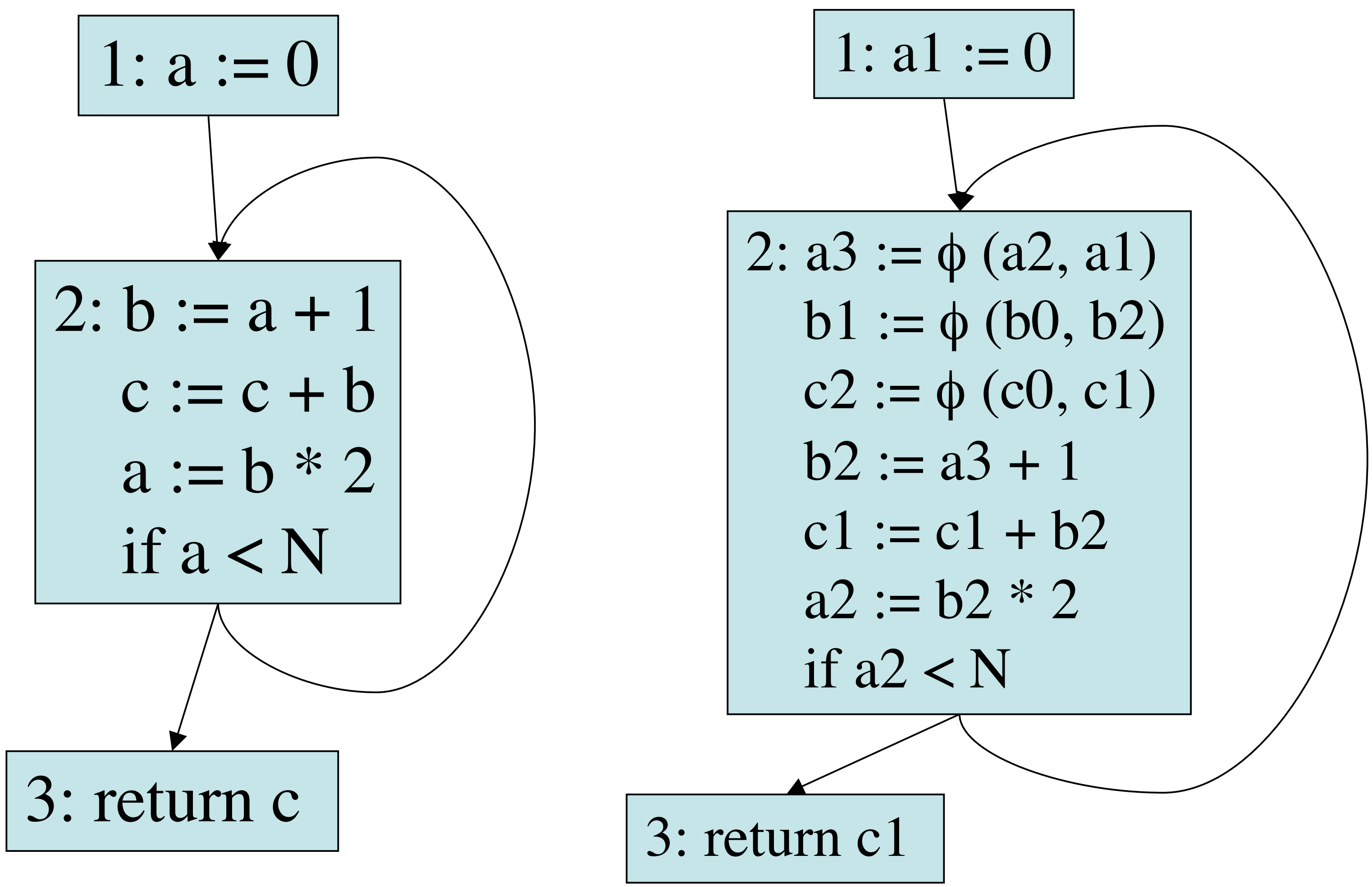
what about conditional branches?

SSA Form


1: b := M[x]
a := 0
2: if b < 4
3: a := b
4: c := a+b


1: b1 := M[x1]
a1 := 0
2: if b1 < 4
3: a2 := b1
4: a3 := ϕ (a2, a1)
c1 := a3 + b1

SSA Form


1: a := 0
2: b := a + 1
c := c + b
a := b * 2
if a < N
3: return c
1: a1 := 0
2: a3 := ϕ (a2, a1)
b1 := ϕ (b0, b2)
c2 := ϕ (c0, c1)
b2 := a3 + 1
c1 := c1 + b2
a2 := b2 * 2
if a2 < N
3: return c1

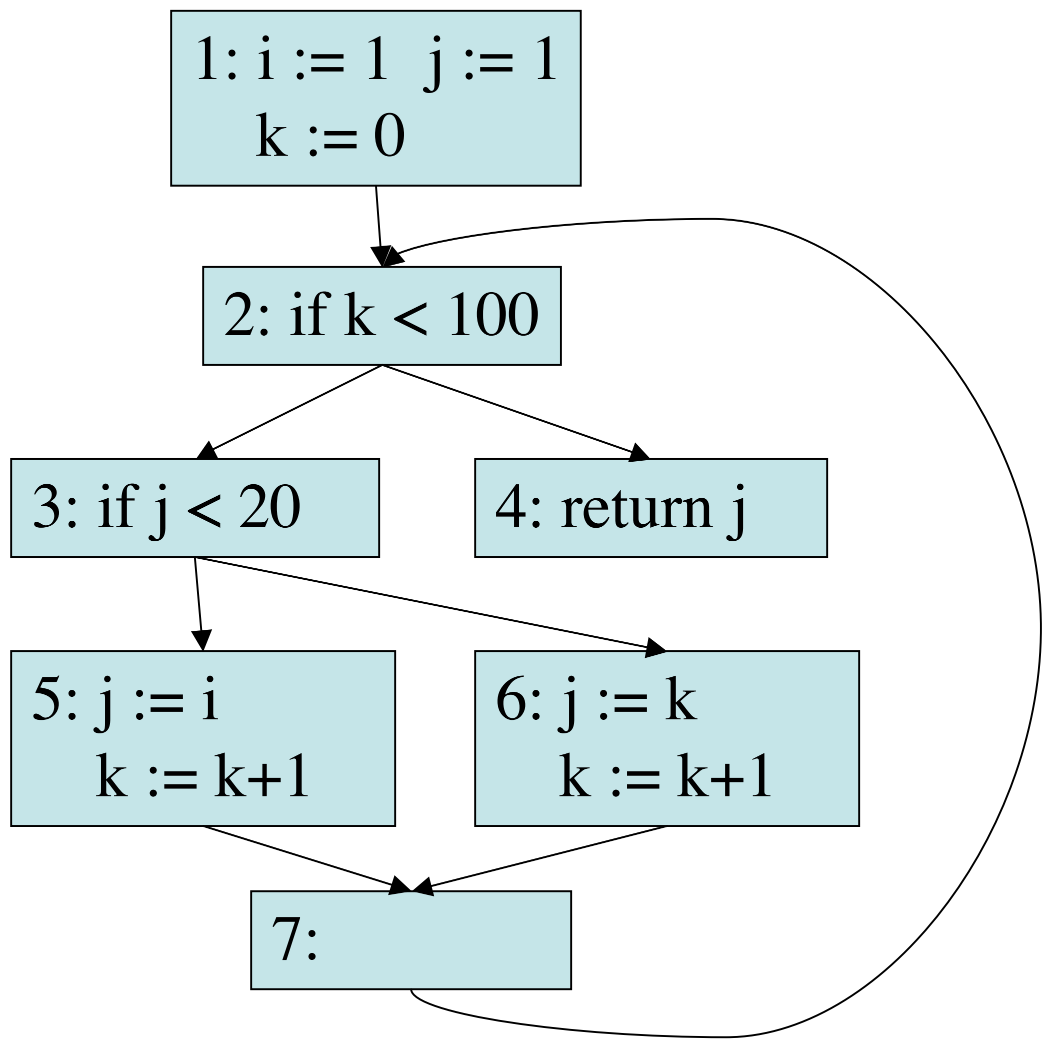
Optimizations using SSA

- 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 *uses* 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
  - $v := \phi\ (c1, c2, ..., cn)$ where all c_i are equal to c can be replaced by $v := c$

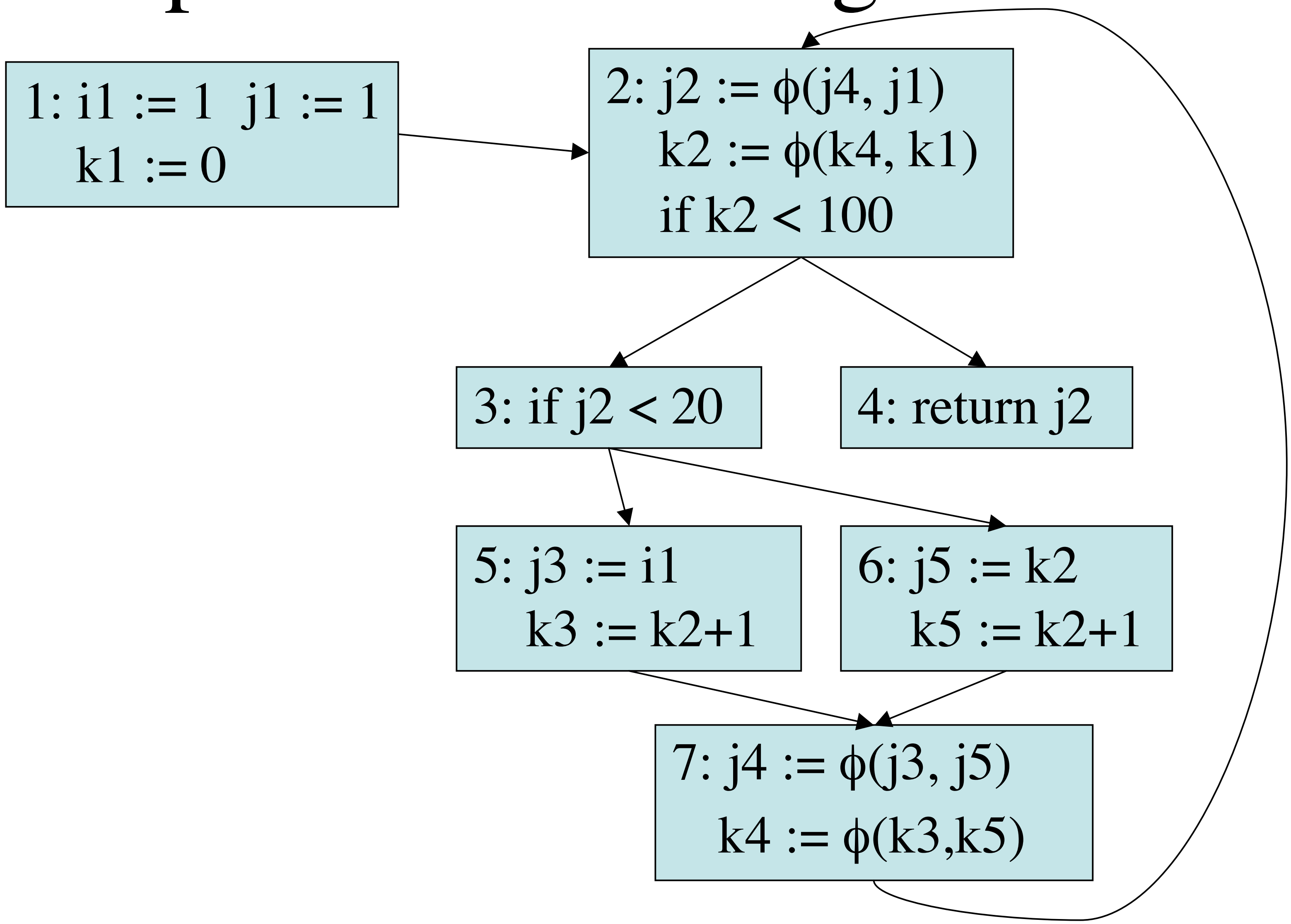
Optimizations using SSA



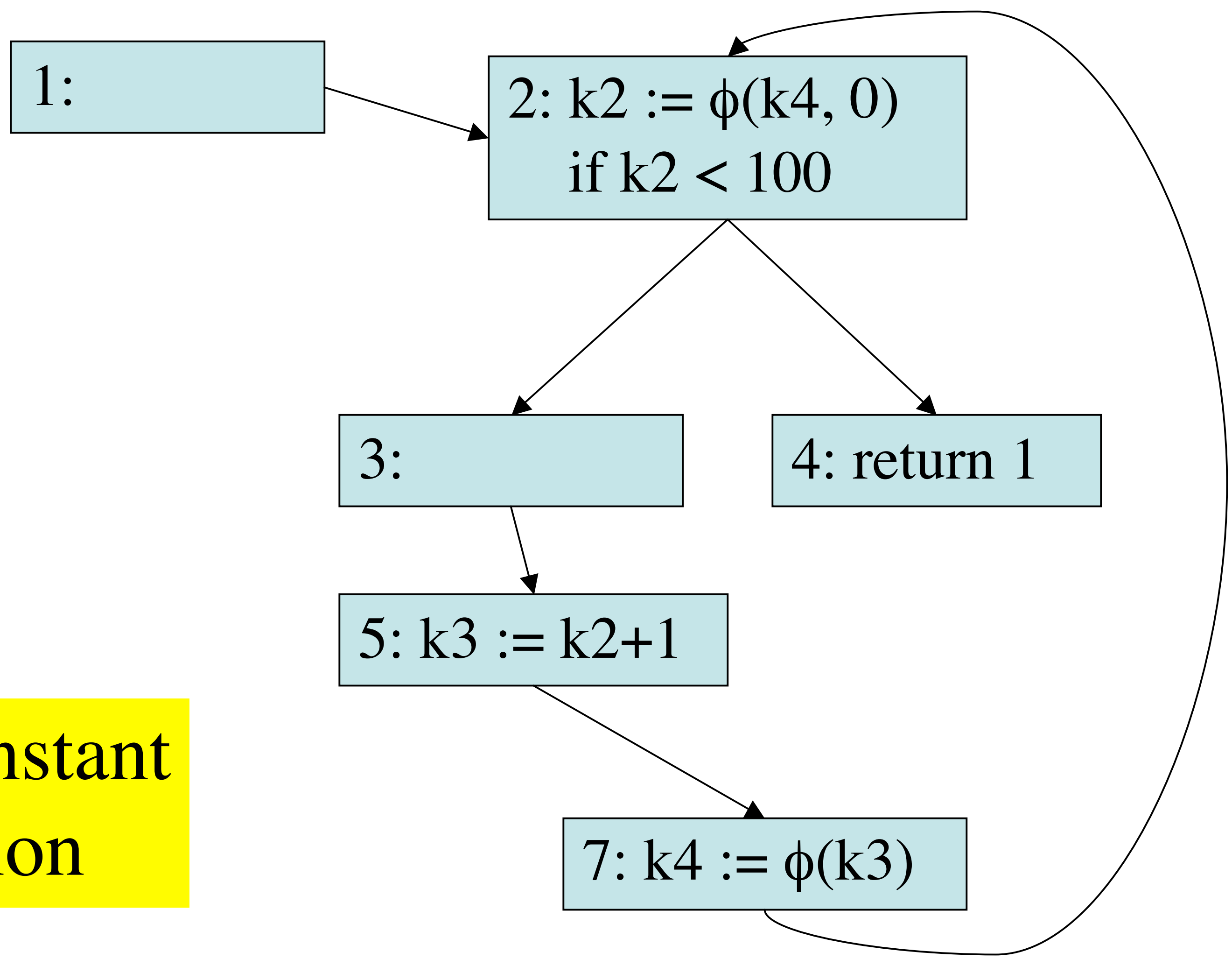
Optimizations using SSA

- 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

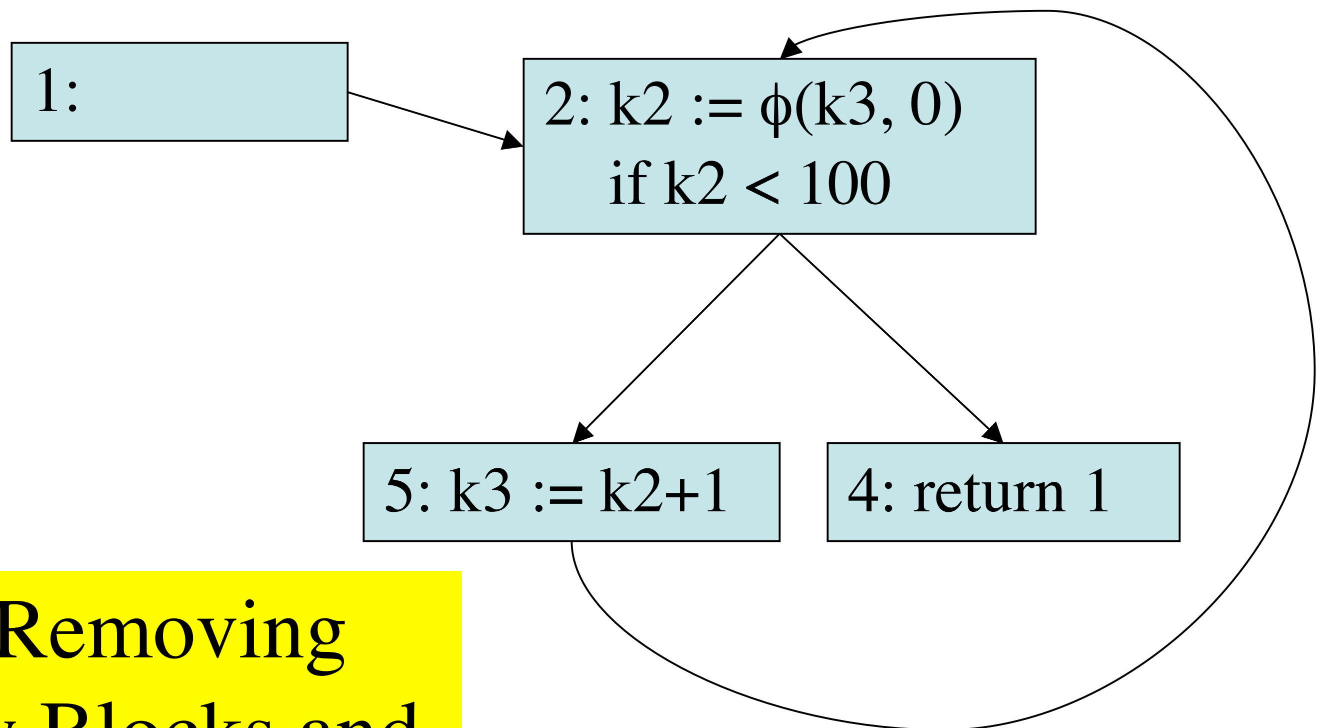


Optimizations using SSA



After Constant Propagation

Optimizations using SSA



After Removing Empty Blocks and 1-arg ϕ functions

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

SSA Form

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

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

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

Moore’s Law

- Speed per $ doubles every 18 months
- How long do you have to wait until a new processor obsoletes your +5% performance improvement?
- And how does that feel if the optimization was machine-specific?)

Summary

- Optimizations can improve speed, while maintaining correctness
- Various early optimization steps
- Global optimizations = dataflow analysis
- Reachability and Liveness analysis provides dataflow analysis
- Static Single-Assignment Form (SSA)