Summary of Optimization Material

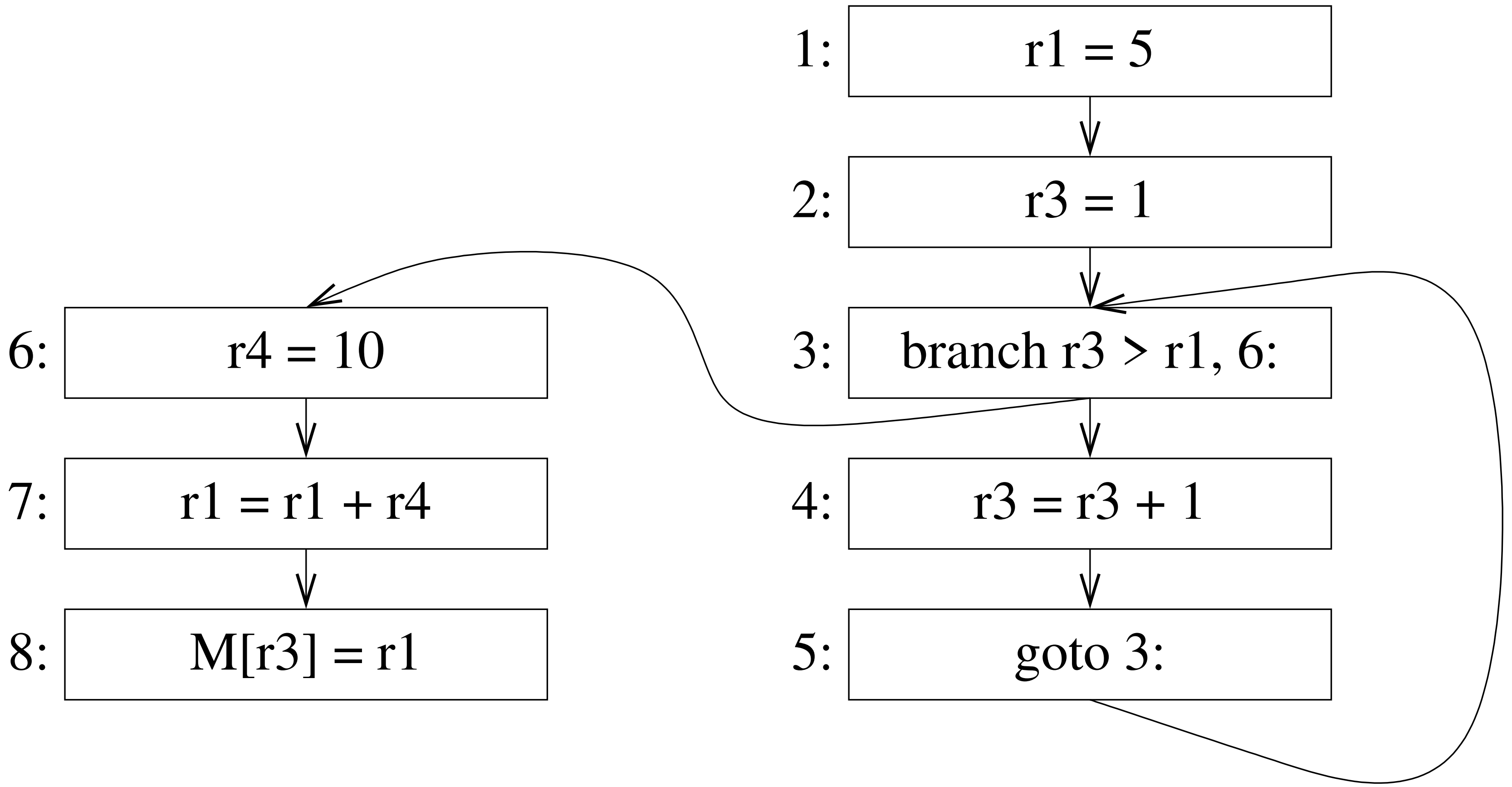
We've looking at a variety of different analysis techniques and optimization techniques over the last couple of weeks:

- Chapter 17.1-17.3: Data-flow analysis and optimizations
  - Liveness analysis, reaching definition analysis, available expression analysis, ...
  - Constant propagation, copy propagation, common sub expression elimination, constant folding, ...
- Chapter 18.1-18.3: Dominators, loops, analysis and optimizations
  - Loop invariant analysis and statement hoisting
  - Induction variable analysis, strength reduction and elimination.
- Chapter 19.1, 19.3 (not conditional constant propagation): Static Single Assignment (SSA), a pervasive intermediate representation for advanced optimization

Motivating SSA

- Many optimizations need to find all use-sites for each definition, and all definition-sites for each use.
  - Constant propagation must refer to the definition-site of the unique reaching definition.
  - Copy propagation, common sub-expression elimination...
- Information connecting all use-sites to corresponding definition-sites can be stored as *def-use chains* and/or *use-def chains*.
- *def-use chains*: for each definition d of r, list of pointers to all uses of r that d reaches.
- *use-def chains*: for each use u of r, list of pointers to all definitions of r that reach u.

Use-Def Chains, Def-Use Chains Example



Static Single Assignment

Static Single Assignment (SSA):

- improvement on def-use chains
- each temporary has only one definition in program
- for each use u of r, only one definition of r reaches u

Static Single Assignment

Static Single Assignment Advantages:

- Dataflow analysis and code optimization is simplified and made more efficient.
- Less space required to represent def-use chains. Def-use chains require space proportional to uses * defs for each variable.
- Eliminates unnecessary relationships:
  - aids register allocation:

    ```
    for i = 1 to N do A[i] = 0
    for i = 1 to M do B[i] = 1
    ```

  - No reason why both loops should be forced to use same register to hold index register.
  - SSA renames second `i` to a new temporary which may lead to better register allocation/optimization.
  - also aids hoisting invariant code out of loops

Static Single Assignment

```
int f(int i, int j) {
  int x,y;
  switch (i) {
   case 0: x = 3; break;
   case 1: x = 7; break;
   case 2: x = 4; break;
   default: x = 17; break;
  }
  switch (j) {
   case 0: y = x+1; break;
   case 1: y = x+7; break;
   case 2: y = x+3; break;
   default: y = x+33; break;
  }
  return y;
}
```

Building def-use chains costs quadratic space whereas SSA encodes def-use information in linear space.

Conversion to SSA Form

Easy to convert basic blocks into SSA form:

- Each definition modified to define brand-new register, instead of redefining old one.
- Each use of register modified to use most recently defined version.

```
r1 = r3 + r4

r2 = r1 - 1

r1 = r4 + r2

r2 = r5 * 4

r1 = r1 + r2
```

This is easy for straight-line programs but complex control flow introduces problems. How do we ensure each temporary has exactly 1 definition and each use is reached by exactly 1 definition?

Conversion to SSA Form

Easy to convert basic blocks into SSA form:

- Each definition modified to define brand-new register, instead of redefining old one.
- Each use of register modified to use most recently defined version.

```
r1a = r3 + r4

r2a = r1a - 1

r1b = r4 + r2a

r2b = r5 * 4

r1c = r1b + r2b
```

This is easy for straight-line programs but complex control flow introduces problems. How do we ensure each temporary has exactly 1 definition and each use is reached by exactly 1 definition?

Conversion to SSA Form

```
r1 = r3 * 5;

if (r2 > 0)
   r1 = r1 + 1;
else
   r2 = r2 + 1;

r4 = r1 + r2
```

convert to SSA???

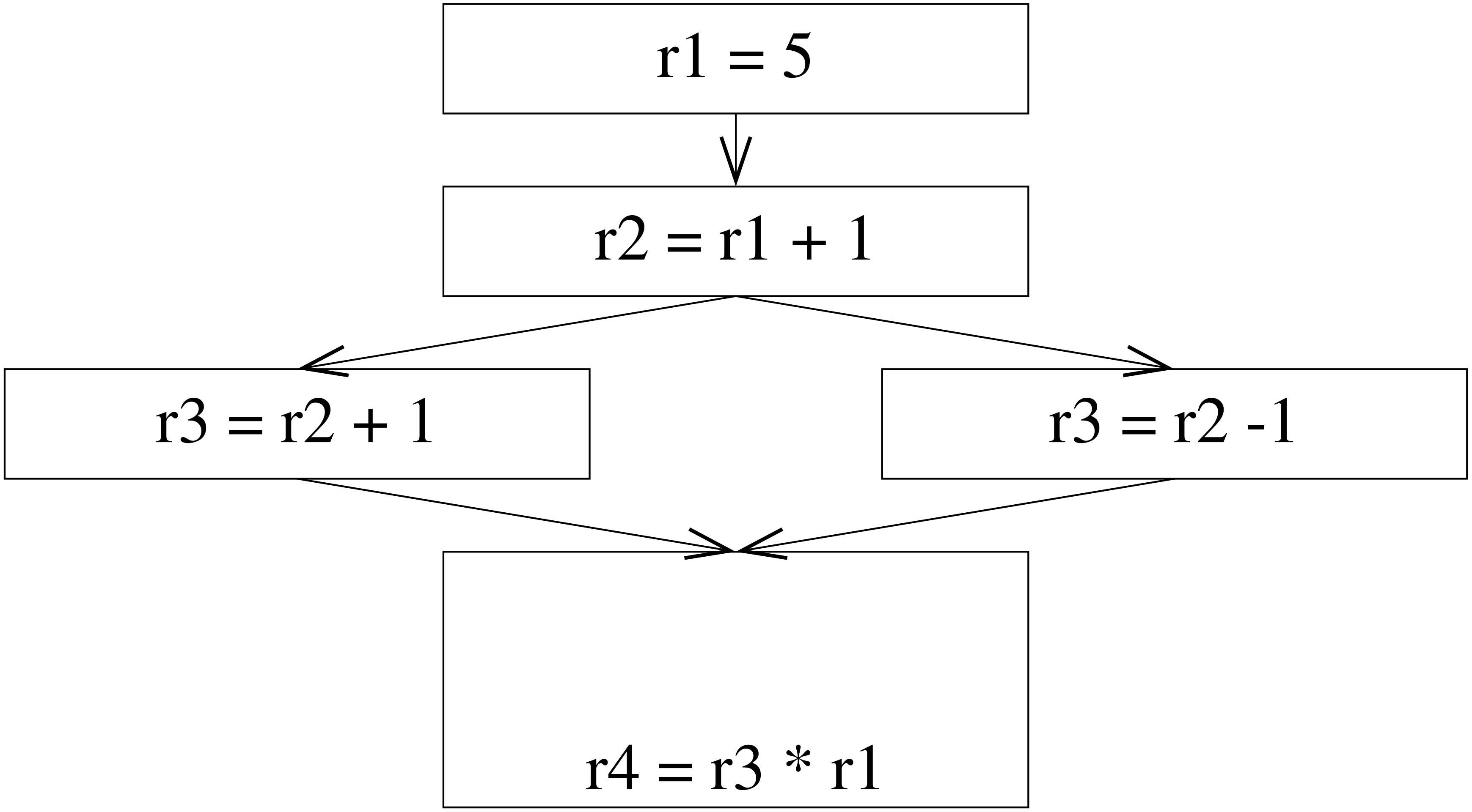
Use ϕ functions.

```
r1 = r3 * 5;

if (r2 > 0)
   r1 = r1 + 1;
else
   r2 = r2 + 1;

r4 = r1 + r2
```

- ϕ-functions enable the use of r1 to be reached by exactly one definition of r1.
- $r1c = \phi(r1a, r1b)$:
  - $r1c = r1a$ if control enters from left
  - $r1c = r1b$ if control enters from right
- Can implement ϕ-functions as set of move operations on each incoming edge.
- In practice, ϕ-functions are just used as notation.

Conversion to SSA Form - Simple Approach

Can insert ϕ-functions for each register at each node with more than two predecessors.



We can do better...

Conversion to SSA Form

Path-Convergence Criterion: Insert a ϕ-function for a register r at node z of the flow graph if ALL of the following are true:

1. There is a block x containing a definition of r.
2. There is a block $y \neq x$ containing a definition of r.
3. There is a non-empty path P_{xz} of edges from x to z.
4. There is a non-empty path P_{yz} of edges from y to z.
5. Paths P_{xz} and P_{yz} do not have any node in common other than z.
6. The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.

Assume CFG entry node contains implicit definition of each register:

- r = actual parameter value
- r = undefined

ϕ-functions are counted as definitions.

Conversion to SSA Form

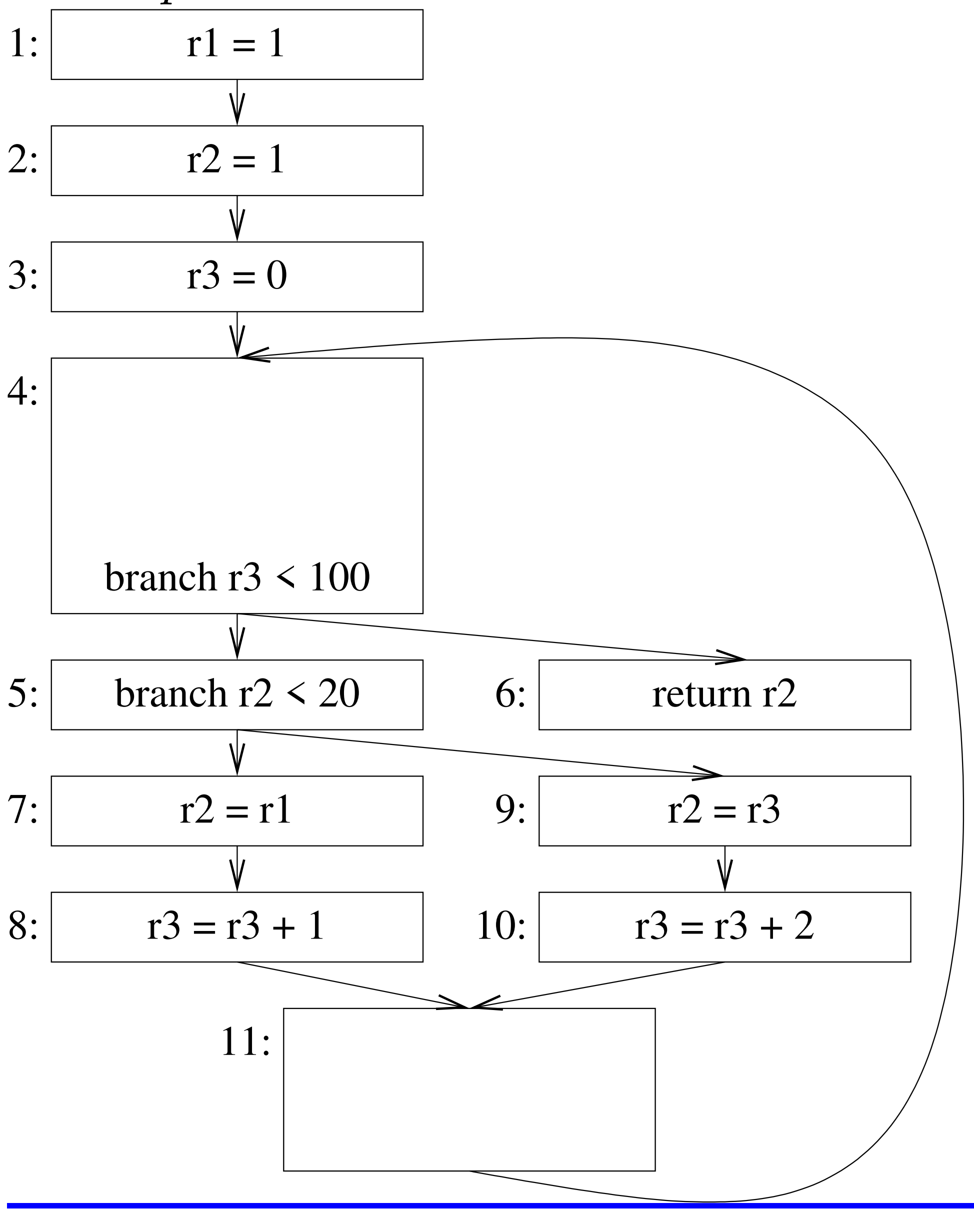
Solve path-convergence iteratively:

WHILE (there are nodes x, y, z satisfying conditions 1-6) &&
(
z does not contain a phi-function for r) DO:
insert $r = \phi(r, r, ..., r)$ (one per predecessor) at node z.

- Costly to compute.
- Since definitions dominate uses, use domination to simplify computation.
- Use *Dominance Frontier* algorithm (see section 19.1)

Static Single Assignment Example

Insert *phi*-functions:



Static Single Assignment Example

To rename variables:

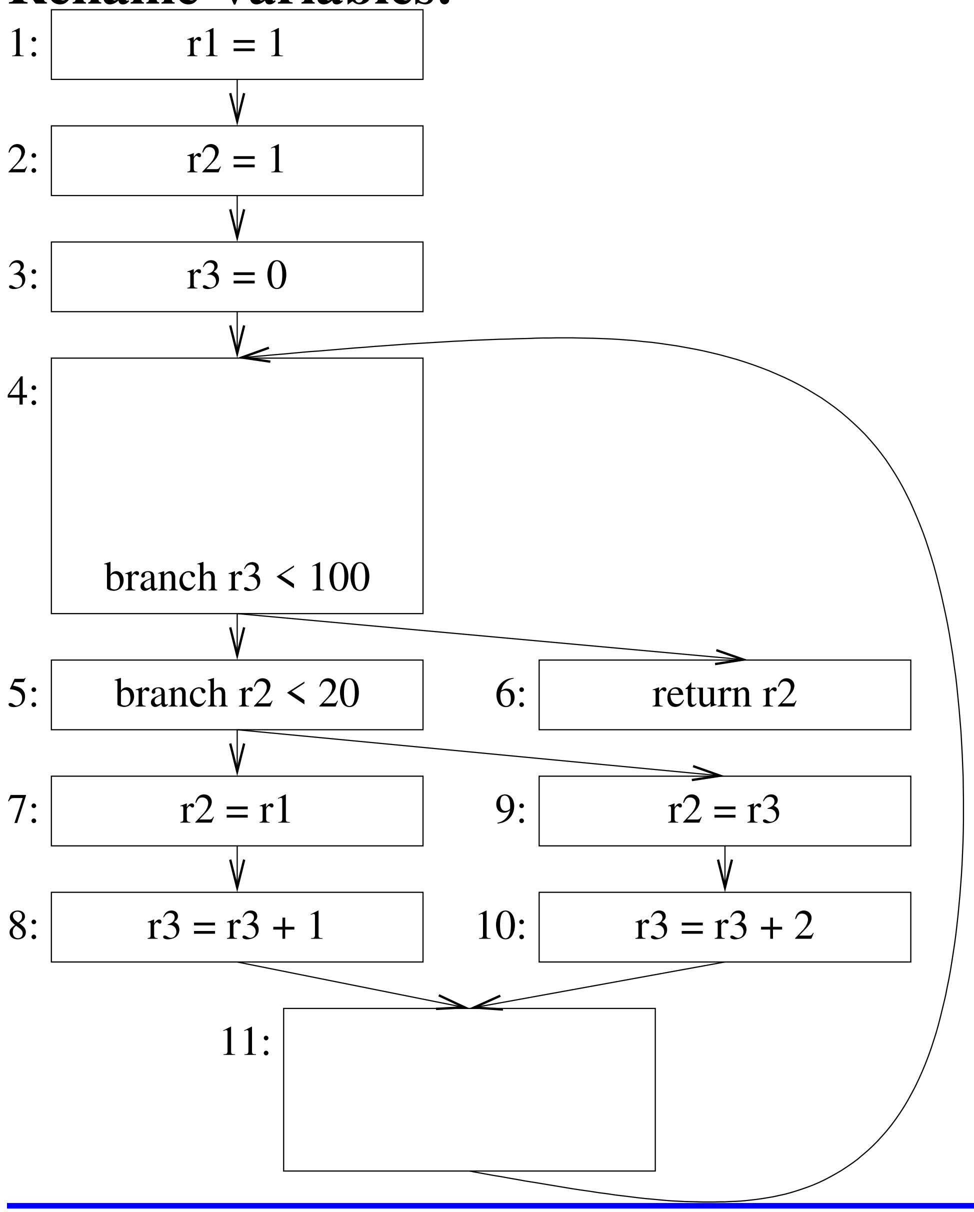
1. traverse dominator tree, renaming different definitions of r to r_1, r_2, r_3...
2. rename each regular use of r to most recent definition of r
3. rename ϕ-function arguments with each incoming edge's unique definition

Dominance property of SSA form:

- If x is i^{th} argument of ϕ-function in node n, then definition of x dominates i^{th} predecessor of n.
- If x is used in non-ϕ statement in node n, then definition of x dominates n.

Static Single Assignment Example

Rename Variables:



Concrete Representations of SSA

Data structures for representing SSA involve:

- *statement* data structure is a record with fields for
  - sort (assignment, ϕ, branch,...), variables used and defined
  - containing block, next and previous statements in block
- *variable* data structure is a record with fields for
  - definition statement, list of use statements
- *basic block* data structure is a record with fields for
  - statements, (ordered) list of predecessors, list of successors

Dead Code Elimination

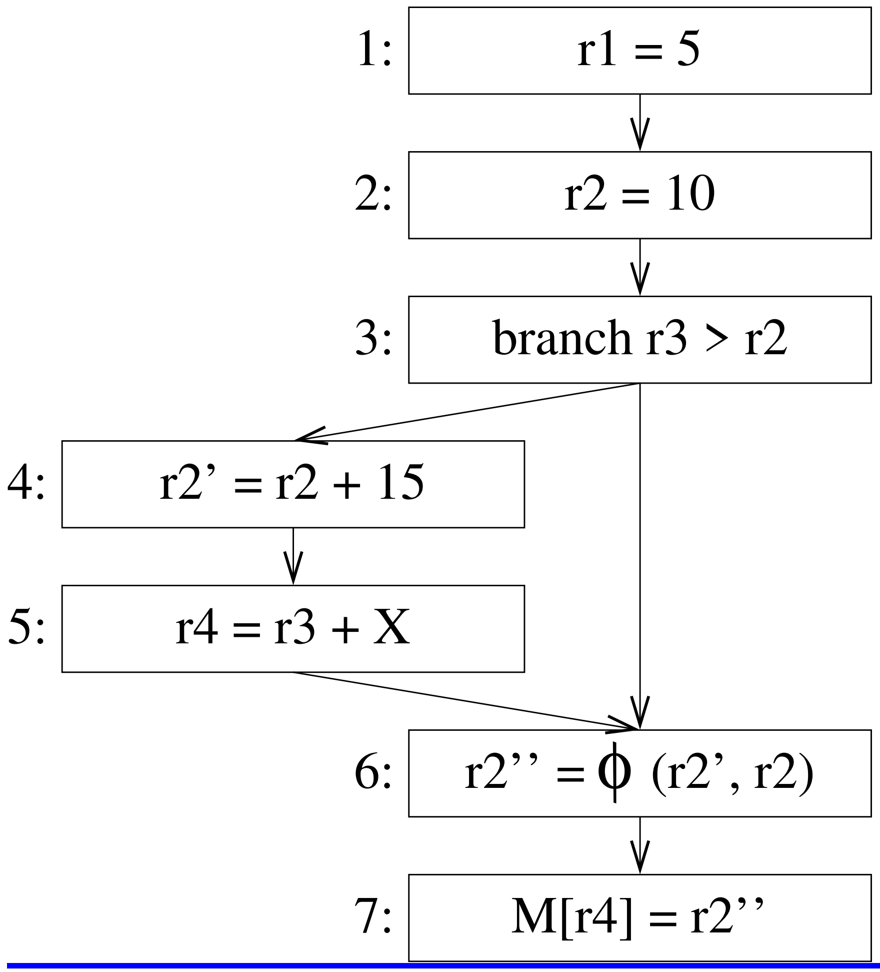
Given d: `t = x op y`

- `t` is live at end of node d if there exists path from end of d to use of `t` that does not go through definition of `t`.
- if program not in SSA form, need to perform liveness analysis to determine if `t` live at end of d.
- if program is in SSA form:
  - cannot be another definition of `t`
  - if there exists use of `t`, then path from end of d to use exists, since definitions dominate uses.
  - otherwise (provided the statement in question has no side effects), it will be dead.
  - to check for dead statements, we check for side effects and to see if the statement use list is empty

Dead Code Elimination

Algorithm:

WHILE (for each temporary `t` with no uses &&
statement defining `t` has no other side-effects) DO
delete statement definition `t`



Simple Constant Propagation

Given d: `t = c`, c is constant Given u: `x = t op b`

- if program not in SSA form:
  - need to perform reaching definition analysis
  - use of `t` in u may be replaced by `c` if d reaches u and no other definition of `t` reaches u
- if program is in SSA form:
  - d reaches u, since definitions dominate uses, and no other definition of `t` exists on path from d to u
  - d is only definition of `t` that reaches u, since it is the only definition of `t`.
    * any use of `t` can be replaced by `c`
    * any ϕ-function of form `v` $=$ $\phi(c_1, c_2, ..., c_n)$, where $c_i = c$, can be replaced by `v = c`

Simple Constant Propagation