SSA form and pointers

What about pointers?

```
x := 5;
y := 7;
p := new int;
q := test1 ? &x : (test2 ? &y : p);
*q := 9;
```

*// what are the unique SSA names for x & y here? *p?*

```
x := x + 1;
```

// what does q point to here?

SSA wishes to assign a unique name for each variable (memory location?) at each point

- dynamic memory allocations introduce many "anonymous variables"
- pointer stores don't definitely update any variable, but may update many
- SSA gives different names to the same variable, but & creates a pointer to all of them

Some solutions

Don't use SSA invariant for heap memory

- maybe even locals that have had their addresses taken

Introduce ι-function at each may-def point of a variable, analogously to φ-functions

- pointers point to original unsubscripted variable

x_1 := 5;
y_1 := 7;
p_1 := new int;
q_1 := *test1* ? &x : (*test2* ? &y : p);
  x := x_1;
  y := y_1;
*q_1 := 9;
  x_2 := $\iota(x_1, x)$;
  y_2 := $\iota(y_1, y)$;
x_3 := x_2 + 1;

Loop-invariant code motion

Two steps: analysis & transformation

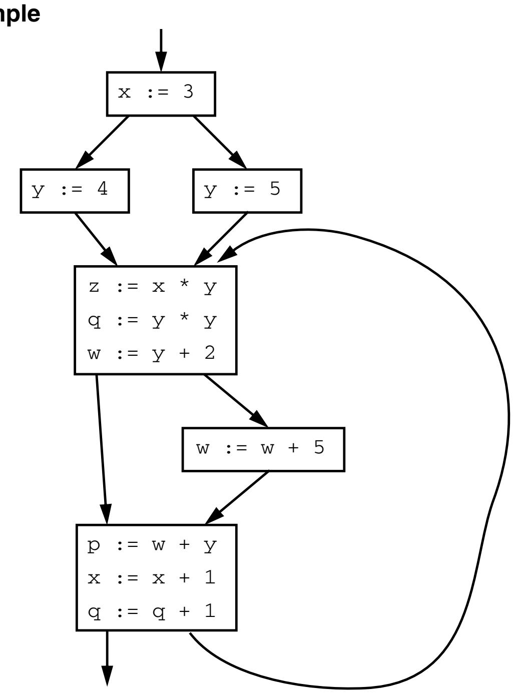
Step 1: find invariant computations in loop

- invariant: computes same result each time evaluated

Step 2: move them outside loop

- to top: **code hoisting**
  - if used within loop
- to bottom: **code sinking**
  - if only used after loop

Example



Detecting loop-invariant expressions

An expression is invariant w.r.t. a loop *L* iff:

base cases:

- it's a constant
- it's a variable use, **all of whose defs are outside *L***

inductive cases:

- it's an idempotent computation
  all of whose args are loop-invariant
- it's a variable use **with only one reaching def**,
  and the rhs of that def is loop-invariant

Computing loop-invariant expressions

Option 1:

- repeat iterative dfa
  until no more invariant expressions found
  - to start, optimistically assume all expressions loop-invariant

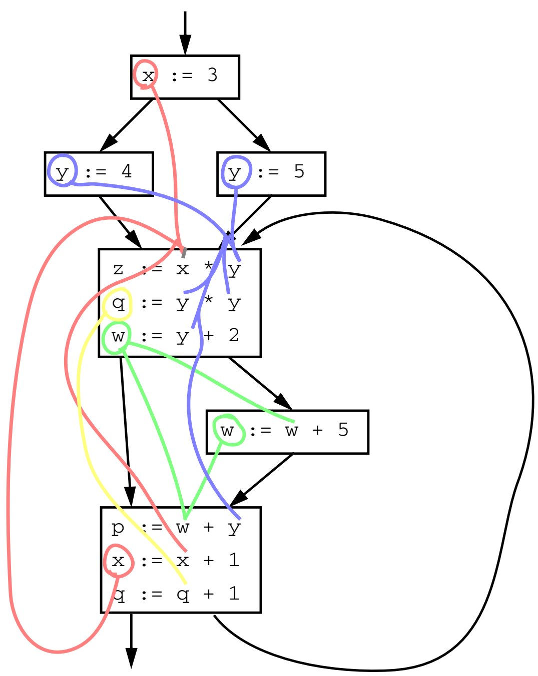
Option 2:

- build def/use chains,
  follow chains to identify & propagate
  invariant expressions

Option 3:

- convert to SSA form,
  then similar to def/use form

Example using def/use chains



Loop-invariant expression detection for SSA form

SSA form simplifies detection of loop invariants,
since each use has only one reaching definition

An expression is invariant w.r.t. a loop *L* iff:

base cases:

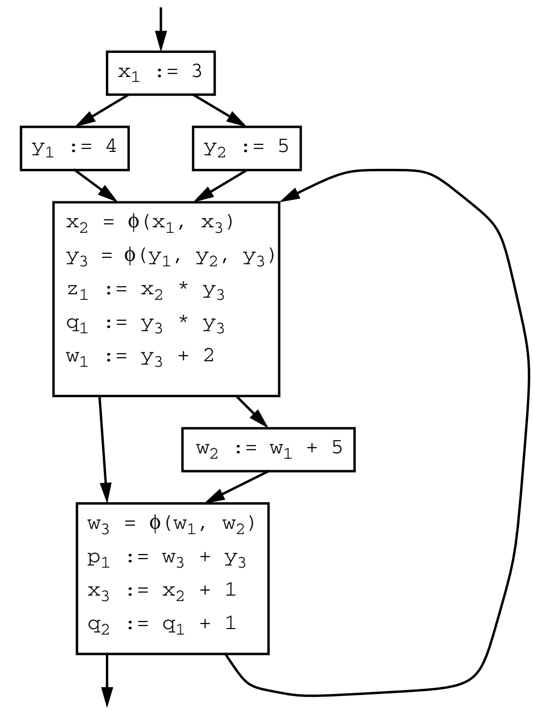
- it's a constant
- it's a variable use **whose <u>single def</u> is outside *L***

inductive cases:

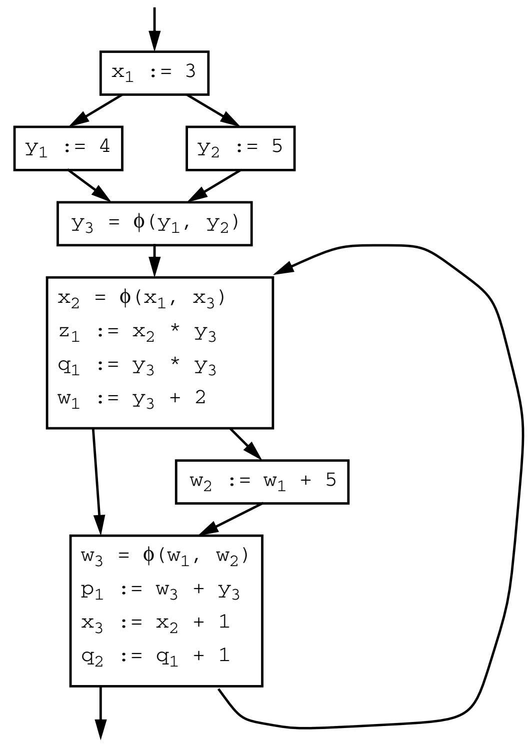
- it's an idempotent computation
  all of whose args are loop-invariant
- it's a variable use
  whose <u>single def</u>'s rhs is loop-invariant

ϕ functions are *not* idempotent

Example using SSA form



Example using SSA form & preheader



Code motion

When find invariant computation *S:* `z := x op y`,
want to move it out of loop (to loop preheader)

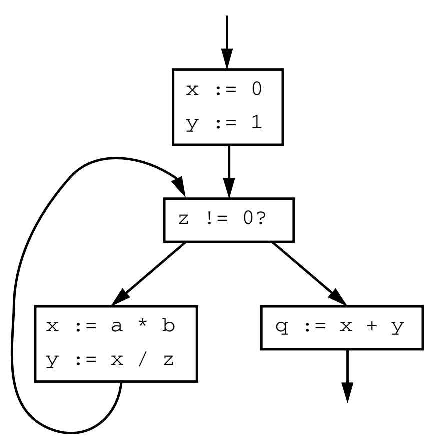
- preserve relative order of invariant computations, to preserve data flow among moved statements

When is this legal?

Condition #1: domination restriction

To move *S:* `z := x op y`,
S must **dominate** all loop exits
[*A* dominates *B* when all paths to *B* first pass through *A*]

- otherwise may execute *S* when never executed otherwise
- can relax this condition, if *S* has no side-effects or traps, at cost of possibly slowing down program



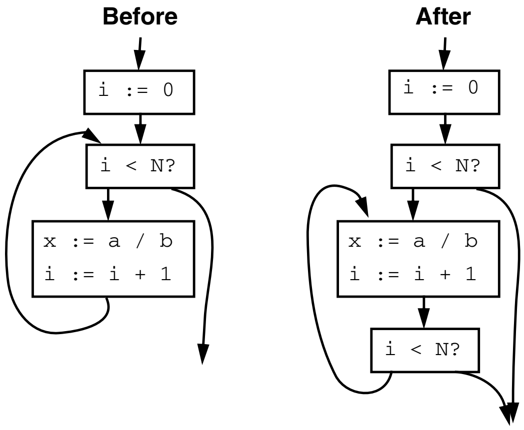
Avoiding domination restriction

Requirement that invariant computation dominates exit is strict

- nothing in conditional branch can be moved
- nothing after loop exit test can be moved

Can be circumvented through other transformations such as **loop normalization**

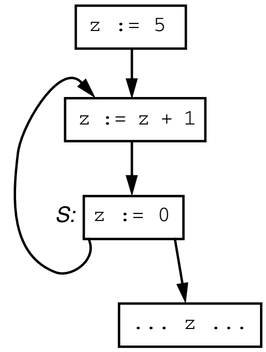
- move loop exit test to bottom of loop (while-do ⇒ if-do-while)



Condition #2: data dependence restriction

To move *S:* `z := x op y`,
S must be the only assignment to `z` in loop, &
no use of `z` in loop is reached by any def other than *S*

- otherwise may reorder defs/uses and change outcome



Avoiding data dependence restriction

Restrictions unnecessary if in SSA form

- implementation of ϕ functions as moves will cope with reordered defs/uses