Announcements

P6, H10 assigned

Final exam

Final exam will cover *all* material covered in course – i.e., all compiler phases

– Focus will be on after-midterm material

There will be some questions on special topics

100 minutes exam, similar in format to midterm

If we add a new feature to the compiler, do we need to change certain part of the compiler.
Would not be asked to run a parser.

Optimization Frameworks

Roadmap

Last time:

– Optimization overview
  - Soundness and completeness
– Simple optimizations
  - Peephole
  - LICM

This time:

– More Optimization
– Analysis frameworks

Outline

Review Dominators

Introduce some more advanced concepts

– Static single assignment (SSA)

– Dataflow propagation

DOMINATOR REVIEW

Dominator terms

Domination (A dominates B):

– to reach block B, you must have gone through block A

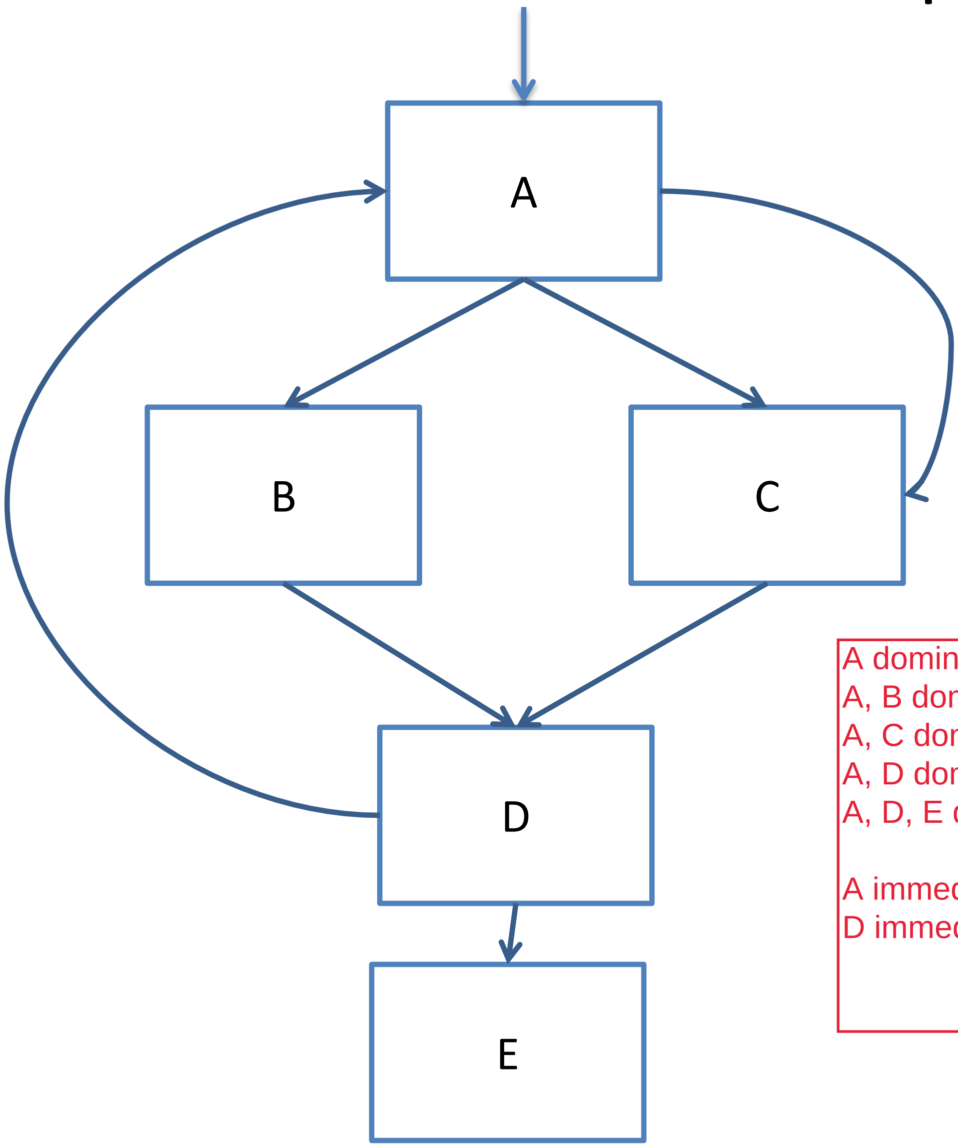
Strict Domination (A strictly dominates B)

– A dominates B and A is not B

Immediate Domination (A immediately dominates B)

– A immediately dominates B if A dominates B and has no intervening dominators

Dominator example



A dominates A
A, B dominate B
A, C dominate C
A, D dominate D
A, D, E dominate E

A immediately dominates B, C, D
D immediately dominates E.

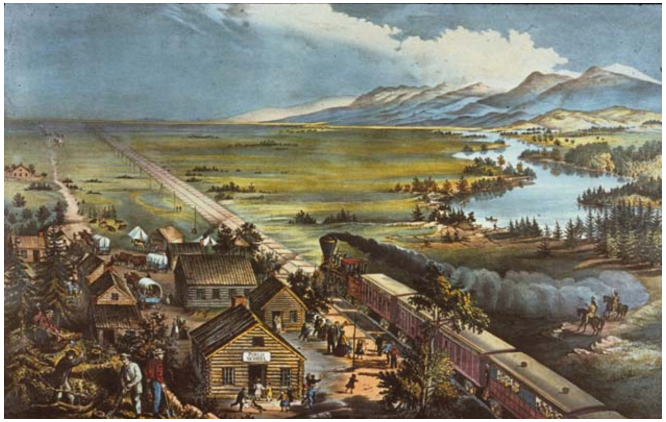
Dominance frontier

Definition:

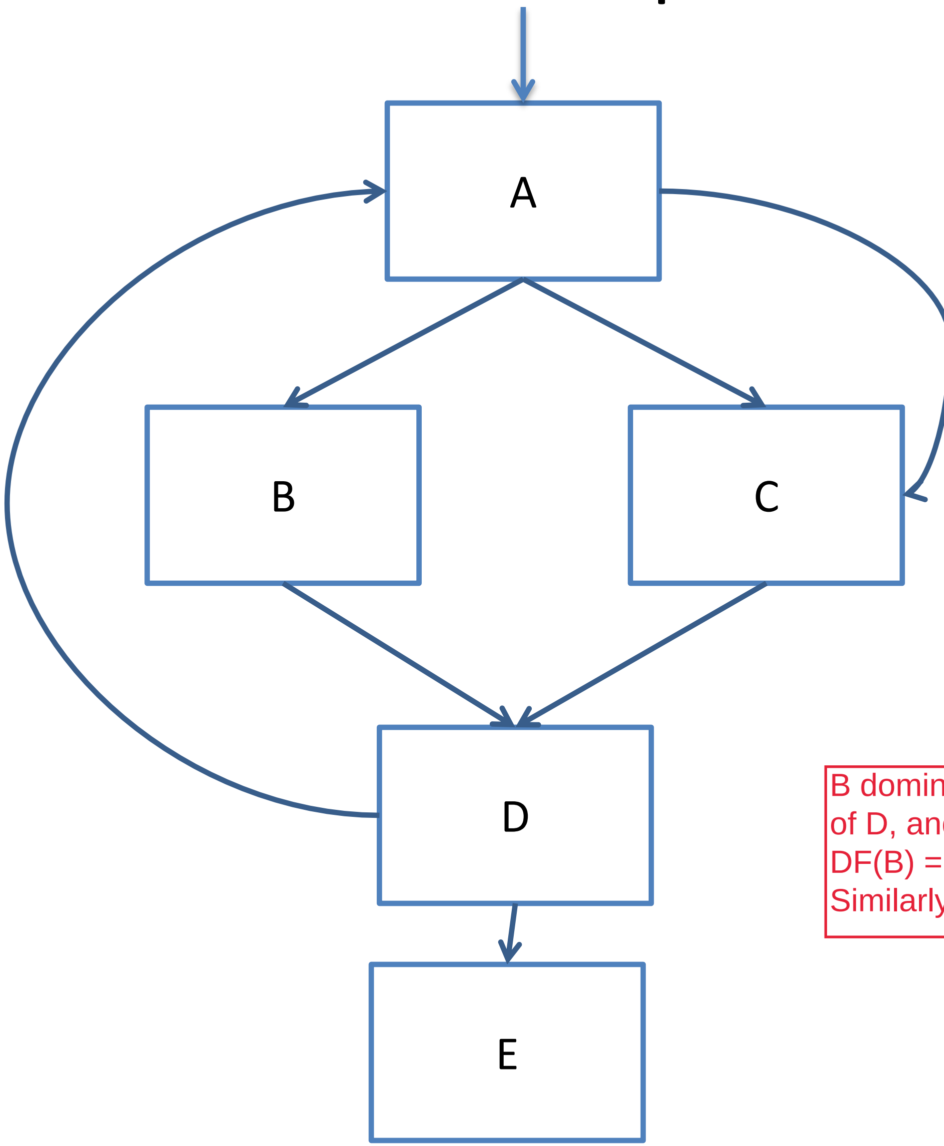
For a block **x**, the dominance frontier of x is the set of nodes **Y** such that

- **x** dominates an immediate predecessor of **Y**
- **x** does not strictly dominate **Y**

Think of the dominance frontier of x as the set of nodes where x's dominance stops.



Example



B dominates an immediate predecessor of D, and doesn't strictly dominates D, so DF(B) = {D}.
Similarly, DF(C) = {D}.

STATIC SINGLE ASSIGNMENT

Goal of SSA Form

Build an intermediate representation of the program in which each variable is assigned a value in **at most 1 program point**:

✓

```
x = 1
z = 2
y = 3
```

✓

```
x = y
z = y
w = z
```

✗

```
x = 1
x = 2
y = 3
```

✓

```
i = 0;
while( i < 10){
  k = i + 1;
}
```

Statically: There is at most *one* assignment statement that assigns to k

Dynamically: k can be assigned to *multiple* times

Conversion

We'll make new variables to carry over the effect of the original program

$x = 1$
$x = x$
$y = x$

$x_1 = 1$
$x_2 = x_1$
$y_1 = x_2$

Use the latest one.

Benefits of SSA Form

There are some obvious advantages to this format for program analysis

– Easy to see the *live range* of a given variable x assigned to in statement s
  - The region from "x = …;" until the last use(s) of x before x is redefined
  - In SSA form, from "x_i = …;" to all uses of x_i, e.g., "… = f(…, x_i, …);"
– Easy to see when an assignment is *useless*
  - We have "x_i = …;" and there are *no uses of* x_i in any expression or assignment RHS
  - "'x_i = …;' is a useless assignment"
  - "'x_i = …;' is dead code"

In other words, some u … l information is pre-computed, or at least easily reco…

Warning 1: Dead code = useless assignments + unreachable code

+ means OR here, or UNION.

Optim… Helps

At “if (b < 4)”, b is only reached by “b = 2;”
Therefore, the else branch is unreachable (dead), and can be removed

Dead-Code Eli…tion

```
int a = 9;
int b = 2;

if (g < 12){
  a = 1;
} else {
  if (b < 4){
    a = 2;
  } else {
    a = 3;
  }
}
b = a;
return 2;
```

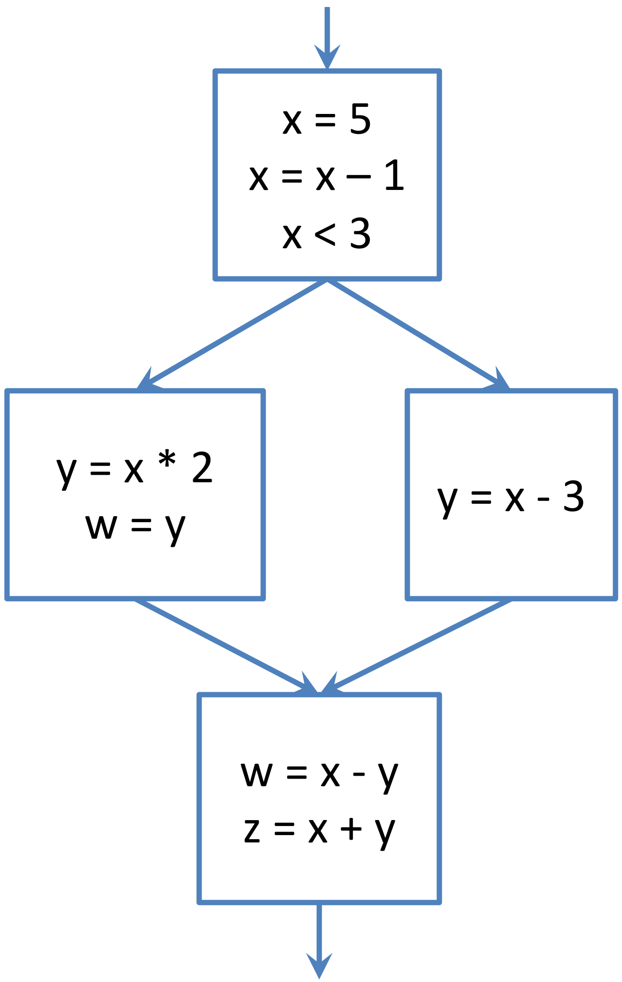
```
int a_1 = 9;
int b_1 = 2;

if (g_1 < 12){
  a_2 = 1;
} else {
  if (b_1 < 4){
    a_3 = 2;
  } else {
    a_4 = 3;
  }
  a_5 = φ(a_3, a_4);
}
a_6 = φ(a_2, a_5);
b_2 = a_6;
return 2;
```

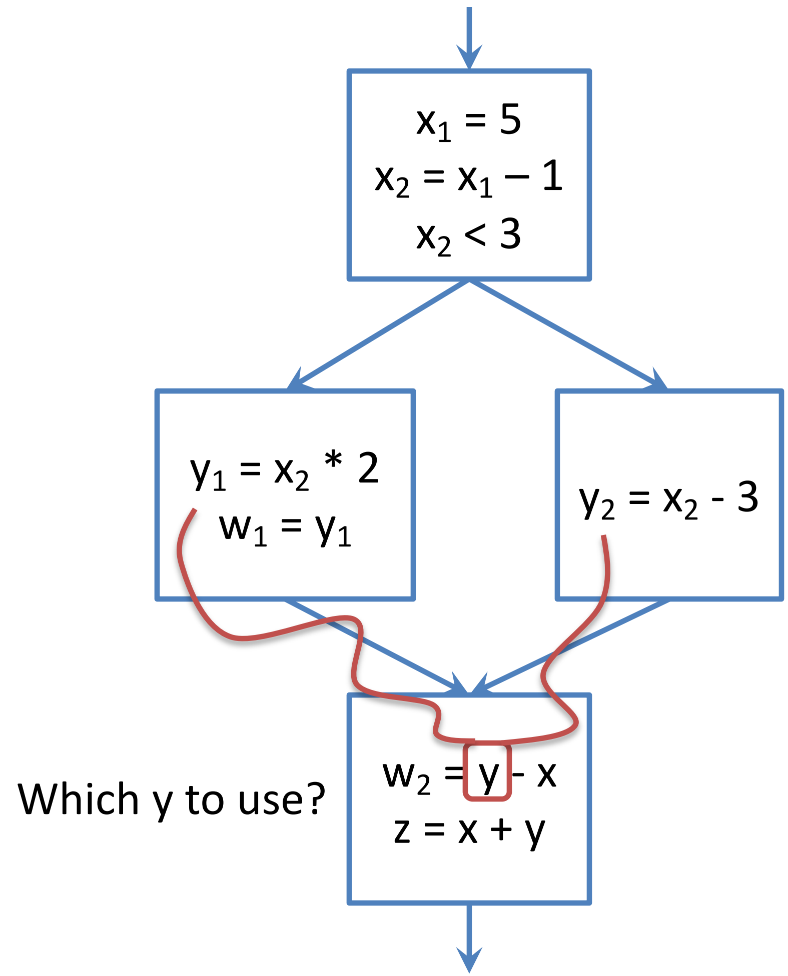
Optimizations Where SSA Helps

Constant-propagation/constant-folding

```
int a = 30;
int b = 9 - (a / 5);
int c;
c = b * 4;
if (c > 10) {
   c = c - 10;
}
return c * (60 / a);
```

What About Conditionals?





Phi Functions (ϕ)

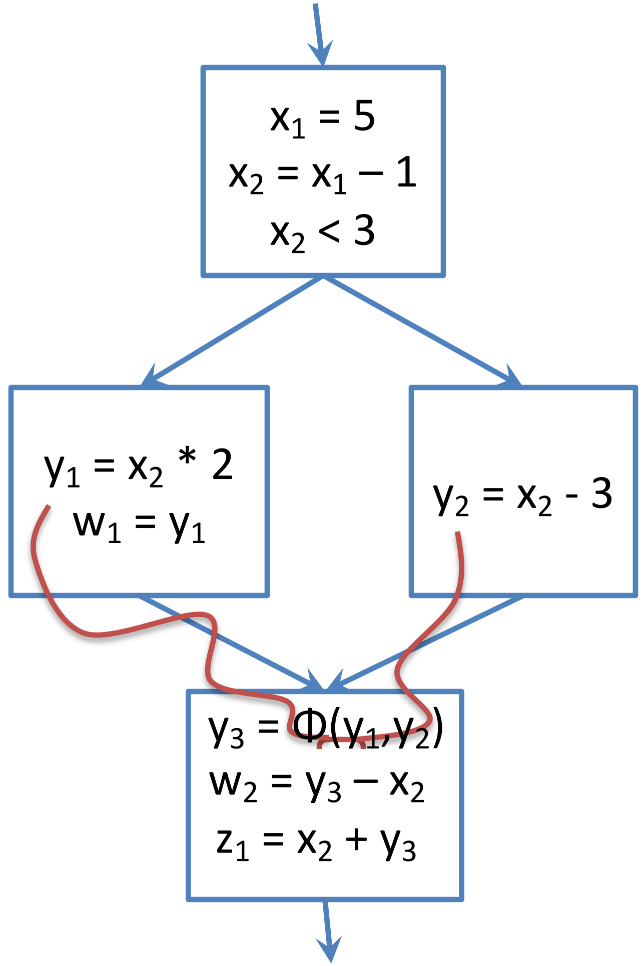
We introduce a special symbol Φ at such points of confluence

Φ's arguments are all the instances of variable y that might be the most recently assigned variant of y

Returns the "correct" one

Do we need a Φ for x?

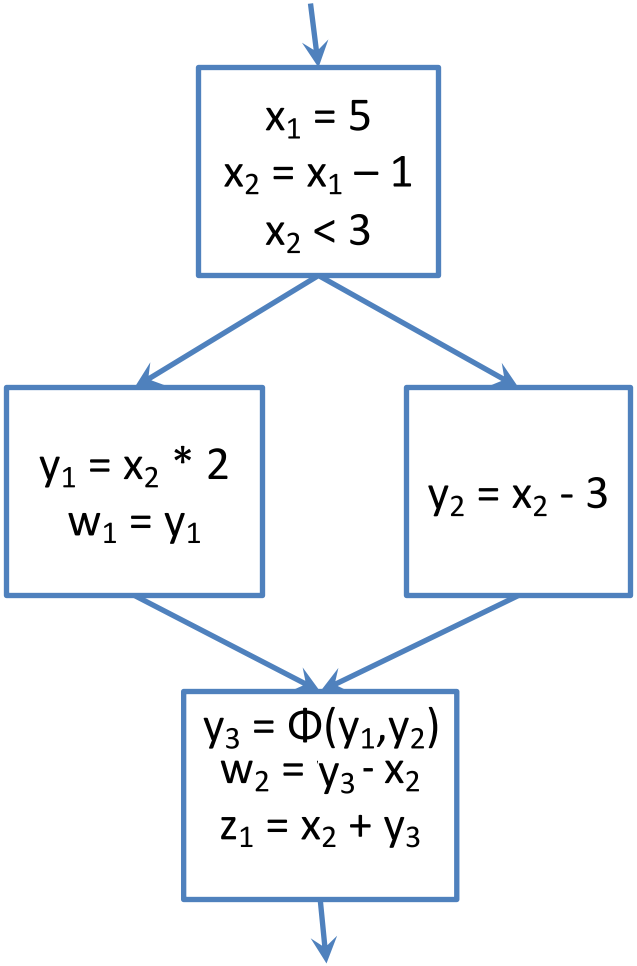
- No!



Computing Phi-Function Placement

Intuitively, we want to figure out cases where there are multiple assignments that can reach a node

To be safe, we can place a Φ function for each assignment at every node in the *dominance frontier*



Pruned Phi Functions

This criterion causes a bunch of useless Φ functions to be inserted

– Cases where the result is never used "downstream" (useless)

Pruned SSA is a version where useless Φ nodes are suppressed

DATAFLOW ANALYSIS

Dataflow framework idea

Many analyses can be formulated as how data is transformed over the control flow graph

Propagate static information from:

– the beginning of a single basic block

– the end of a single basic block

– The join points of multiple basic blocks

Dataflow framework idea

Meet Lattice

Transfer function

– How data is propagated from one end of a basic block to the other

Meet operation

– Means of combining lattice between blocks

Dataflow analysis direction

Forward analysis

– Start at the beginning of a function's CFG, work along the control edges

Backwards analysis

– Start at the end of a function's CFG, work against the control edges

Continuously propagate values until there is no change

Dataflow-Analysis Example 1

Reaching definitions

Transfer function:
$\lambda S.\,(S - \{< p_i, x >\}) \cup \{< p2, x >\}$

Before p1: ∅
After p1: {<p1, x>} p1: x = 1;

. . .

Before p2: {<p1, x>, …}
After p2: {<p2, x>, …} p2: x = 2;

. . .

Before p3: {<p2, x>, …}
After p3: {<p2, x>, <p3, y>, …} p3: y = x;

Data: sets of <program-point, variable> pairs

Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.

Dataflow-Analysis Example 1

Reaching definitions

Meet operation: Union of sets (of <program-point, variable> pairs)

Before p1: Ø
After p1: {<p1, x>}

p1: x = 1;

. . .

Before p2: {<p1, x>, …}
After p2: {<p2, x>, …}

p2: x = 2;

. . .

Before p4: Ø
After p4: {<p4, x>}

p4: x = 7;

You need a phi-function later.

Before p3: {<p2, x>, <p4,x>, …}
After p3: {<p2, x>, <p3, y>, <p4,x>,…}

p3: y = x;

p5: x = 5;

After p5: {<p5, x>, <p3, y>, ... }

Note: for expository purposes, it is convenient to assume we have a statement-level CFG rather than a basic-block-level CFG.

Dataflow-Analysis Example 1

Reaching definitions: Why is it useful?

Answers the question “Where could this variable have been defined?”

Connect uses to definitions.

Before p1: Ø
After p1: {<p1, x>}

p1: x = 1;

. . .

Before p2: {<p1, x>, …}
After p2: {<p2, x>, …}

p2: x = 2;

p4: x = 7;

Before p4: Ø
After p4: {<p4, x>}

. . .

Before p3: {<p2, x>, <p4,x>, …}
After p3: {<p2, x>, <p3, y>, <p4,x>,…}

p3: y = x;

Dataflow-Analysis Example 2

Live Variables

Transfer function:
$\lambda S. (S - \{z\}) \cup \{x, y\}$

Before p1: ∅		
After p1: {x}	p1: x = 1;	
	if (…) {	
Before p2: {x}		
After p2: {x,y}	p2: y = 0;	
Before p3: {x,y}		
After p3: ∅	p3: z = x + y;	
	}	
Before p4: ∅		
After p4: {x}	p4: x = 2;	
Before p5: {x}		
After p5: {x}	p5: z = 3;	
Before p6: {x}		
After p6: ∅	p6: cout << x;	

The union of {x} and {}.

Previous definitions of x is not important, at least from this point of view.

Data: sets of variables

z is not live after p5, and thus p5 is a useless assignment (= dead code)

The end: or is it?

Covered a broad range of topics

– Some formal concepts

– Some practical concepts

What we skipped

– Linking and loading

– Interpreters

– Register allocation

– Performance analysis / Proofs