Lecture N:

Optimization

CSE401/501m:

Introduction to Compiler Construction

Instructor: Gilbert Bernstein

Administrivia

- Checking due ***tomorrow*** night!
  - ✦ When you think you're "done," make sure to *review the project assignment text!*
  - ✦ And check your work (clean, rebuild from a fresh clone, mark on gradescope, etc.)

Outline

Intro to Optimization

Local Optimization

Intra-Procedural Optimization

Inter-Procedural Optimization

Data-Structures, Analysis, Transformation

Outline

Intro to Optimization

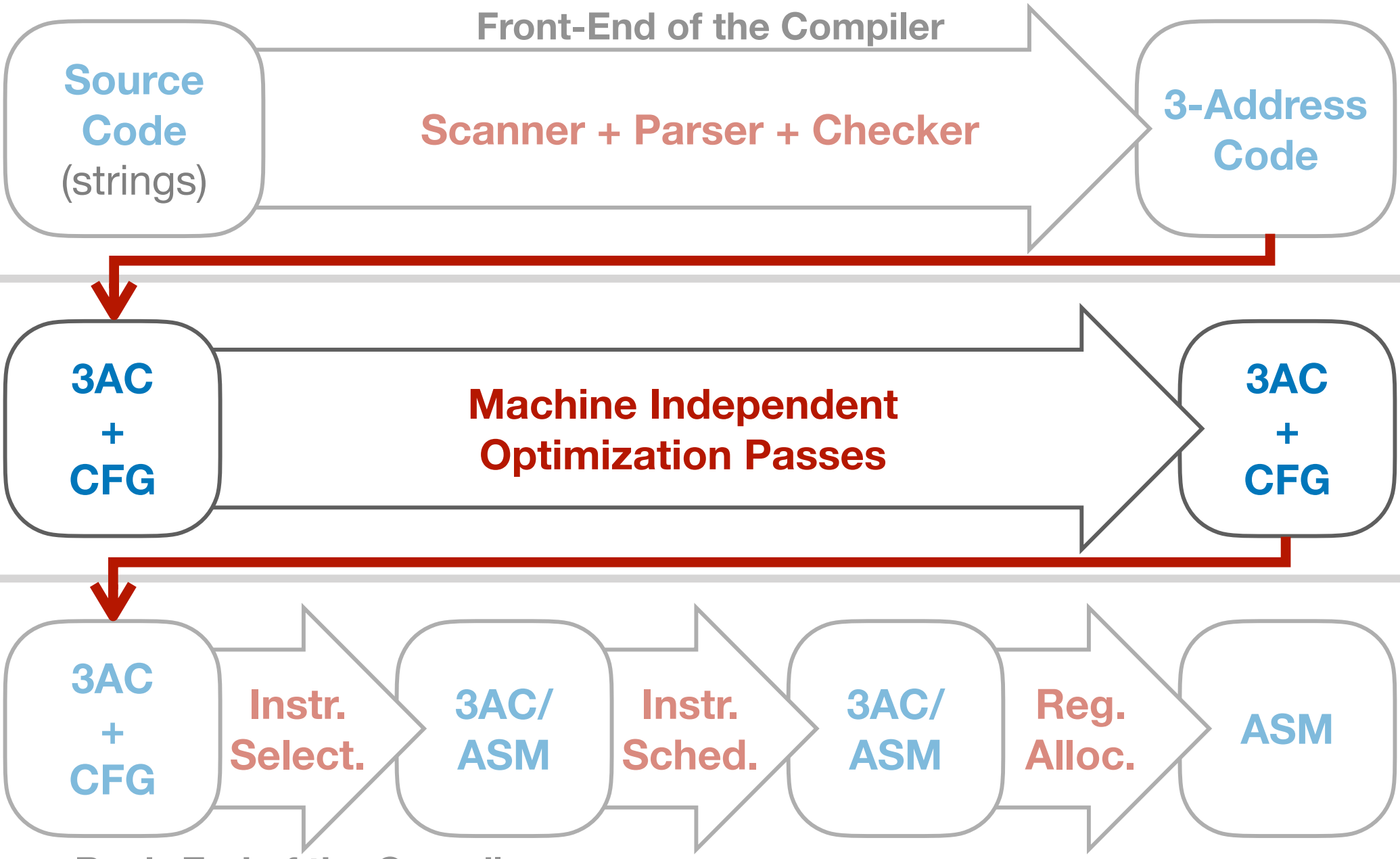
Local Optimization

Intra-Procedural Optimization

Inter-Procedural Optimization

Data-Structures, Analysis, Transformation

Our Compiler Journey



Why Optimize Code?

- Runs faster
- Runs with less power consumption
- Programs use less memory
- Program code takes less space to store (code size)
- buy less machines, power, etc. — saves a lot of $$$
- real-time systems — If you don't have sufficient performance, the system doesn't work!
  - ✦ true in general, but most dramatic for real-time systems
- In compilers — **separation of concerns** — first worry about correctly translating code, then separately worry about making it efficient

A Color Never Seen Before

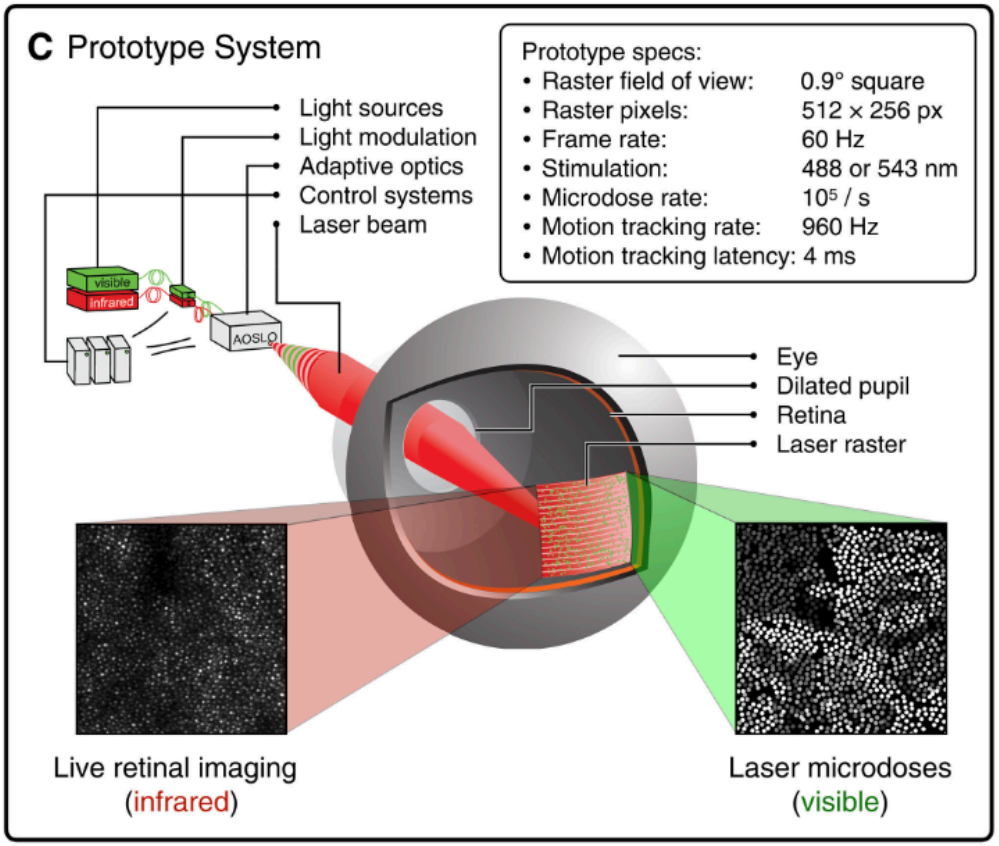
NEWS | 18 April 2025 | Correction 22 April 2025

Brand-new colour created by tricking human eyes with laser

The 'off-the-charts saturated' greenish hue – called olo – has been seen by only five study participants.

By Elizabeth Gibney

Five people have been able to perceive a colour never before seen by human eyes, after researchers used lasers and tracking technology to selectively activate certain cells in their retinas. The blue-greenish hue has an intensity, or 'saturation', outside



1 ms * 4 GHz = 4 million cycles

512 x 256 px = 131,072 px

4M / 0.13M = 30 cycles*

**actually, this is being processed on parallel hardware, so many more cycles are available per pixel.*

Good Optimizations

- What properties do we want to be true about optimizations that our compiler performs?
- It makes our code faster!
  - well, it *usually* makes our code faster…
  - some optimizations are always a good idea, but we will also do optimizations that are *usually* a good idea too
- Optimizations **should not change the behavior** of the program!
  - input/output equivalence? Is memory the same?
  - But, we want it to run faster! (and that's observable behavior… which is sometimes a security issue)

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

note: as mentioned, most compilers will use a 3AC that allows for naming an arbitrary number of "registers." We will cover ***register allocation*** *near the end of the course.*

```
 t1 ← *(fp + ioffset);  // i
 t2 ← t1 * 4;
 t3 ← fp + t2;
 t4 ← *(t3 + aoffset);  // a[i]
 t5 ← 2;
 t6 ← t5 * 4;
 t7 ← fp + t6;
 t8 ← *(t7 + boffset);  // b[2]
 t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← *(fp + xoffset);  // x
t11 ← 5;
t12 ← t10 - t11;
t13 ← *(fp + ioffset);  // i
t14 ← t13 * 4;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t5 ← 2;
t6 ← t5 * 4;
t7 ← fp + t6;
t8 ← *(t7 + boffset);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;  // x = …
t10 ← *(fp + xoffset);  // x
t11 ← 5;
t12 ← t10 - t11;
t13 ← *(fp + ioffset);  // i
t14 ← t13 * 4;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t5 ← 2;
t6 ← t5 << 2;
t7 ← fp + t6;
t8 ← *(t7 + boffset);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;  // x = …
t10 ← *(fp + xoffset);  // x
t11 ← 5;
t12 ← t10 - t11;
t13 ← *(fp + ioffset);  // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Strength Reduction

if shift is cheaper than multiplication

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t5 ← 2;
t6 ← t5 << 2;
t
t
t
*(
t1
t11 ← 5;
t12 ← t10 - t11;
t13 ← *(fp + ioffset);  // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t5 ← 2;
t6 ← 2 << 2;
t7 ← fp + t6;
t8 ← *(t7 + boffset);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;  // x = …
t10 ← *(fp + xoffset);  // x
t11 ← 5;
t12 ← t10 - 5;
t13 ← *(fp + ioffset);  // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Constant Propagation

replace variables with known constant values

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t5 ← 2;
t6 ← 2 << 2;
t7 ← fp + t6;
t8 ← *(t7 + boffset);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;  // x = …
t10 ← *(fp + xoffset); // x
t11 ← 5;
t12 ← t10 - 5;
t13 ← *(fp + ioffset); // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Dead Code Elimination

remove assignments to provably unused variables

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t6 ← 2 << 2;
t7 ← fp + t6;
t8 ← *(t7 + boffset);  // b[2]
t
*(
t1
t1
t1
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t6 ← 8;
t7 ← fp + t6;
t8 ← *(t7 + boffset);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;  // x = …
t10 ← *(fp + xoffset);  // x
t12 ← t10 - 5;
t13 ← *(fp + ioffset);  // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Constant Folding

perform computations whose inputs are all constant

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t6 ← 8;
t7 ← fp + t6;
t8 ← *(t7 + boffset);  // b[2]
t
*(
t1
t1
t1
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=
```

Constant Propagation +

Dead Code Elimination

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t7 ← fp + 8;
t8 ← *(t7 + boffset);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;  // x = …
t10 ← *(fp + xoffset); // x
t12 ← t10 - 5;
t13 ← *(fp + ioffset); // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t7 ← fp + 8;
t8 ← *(t7 + boffset);  // b[2]
t9 ← t4 + t8;
*(
t1
t1
t1
t1
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]
```

Algebraic Simplification

+ is commutative & associative

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t7 ← boffset + 8;
t8 ← *(t7 + fp);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← *(fp + xoffset);  // x
t12 ← t10 - 5;
t13 ← *(fp + ioffset);  // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t7 ← boffset + 8;
t8 ← *(t7 + fp);  // b[2]
t9 ← t4 + t8;
*(
t1
t1
t1
t1
t1
*(
```

assuming boffset = -32

Constant Propagation +
Constant Folding +
Dead Code Elimination

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← *(fp + xoffset);  // x
t12 ← t10 - 5;
t13 ← *(fp + ioffset);  // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← *(fp + xoffset);  // x
t12 ← t10 - 5;
t13 ← *(fp + ioffset);
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← *(fp + xoffset);  // x
t12 ← t10 - 5;
t13 ← t1;  // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Common Subexpresssion Elimination (CSE)

we already computed this!

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x =
t10 ← *(fp + xoffset);  // x
t12 ← t10 - 5;
t13 ← t1;  // i
t14 ← t13 << 2;
t15 ← fp + t14;
*(
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← t9;  // x
t12 ← t10 - 5;
t13 ← t1;  // i
t14 ← t1 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Copy Propagation

replace loads following stores and gratuitous moves with the earlier value

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← t9;  // x
t12 ← t10 - 5;
t13 ← t1;  // i
t14 ← t1 << 2;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← t9;  // x
t12 ← t9 - 5;
t13 ← t1;  // i
t14 ← t2;
t15 ← fp + t2;
*(t15 + coffset) ← t12; // c[i]=…
```

Common Subexpression Elimination + Copy Propagation

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← t9;  // x
t12 ← t9 - 5;
t13 ← t1;  // i
t14 ← t2;
t15 ← t3;
*(t3 + coffset) ← t12; // c[i]=…
```

Common Subexpression Elimination + Copy Propagation

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
~~t10 ← t9;~~  // x
t12 ← t9 - 5;
~~t13 ← t1;~~  // i
~~t14 ← t2;~~
~~t15 ← t3;~~
*(t3 + coffset) ← t12; // c[i]=…
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t12 ← t9 - 5;
*(t3 + coffset) ← t12; // c[i]=…
```

Dead Code Elimination

Opt. Example

```
x = a[i] + b[2];
c[i] = x - 5;
```

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t5 ← 2;
t6 ← t5 * 4;
t7 ← fp + t6;
t8 ← *(t7 + boffset);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t10 ← *(fp + xoffset);  // x
t11 ← 5;
t12 ← t10 - t11;
t13 ← *(fp + ioffset);  // i
t14 ← t13 * 4;
t15 ← fp + t14;
*(t15 + coffset) ← t12; // c[i]=…
```

Original Code

```
t1 ← *(fp + ioffset);  // i
t2 ← t1 << 2;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);  // a[i]
t8 ← *(fp - 24);  // b[2]
t9 ← t4 + t8;
*(fp + xoffset) ← t9;   // x = …
t12 ← t9 - 5;
*(t3 + coffset) ← t12; // c[i]=…
```

Optimized Code

	load	store	mov	+/-	*
orig	5	2	10	12	3
opt	3	2	4	8	1

The Optimizations We Saw

strength reduction
constant propagation
dead code elimination
constant folding
constant propagation
dead code elimination
algebraic simplification
constant propagation
constant folding
dead code elimination
common subexpr. elim.
copy propagation
common subexpr. elim.
copy propagation
common subexpr. elim.
copy propagation
dead code elimination

- Unlike in our example, compilers tend to work on code in ***passes*** that go over a large amount of code, rather than make a bunch of individual changes.
- Some optimizations are needed to expose the opportunity for other optimizations. What order should we choose?
  - ✦ This is known as the **phase ordering problem**

LLVM -O2 optimization passes

```
targetlibinfo
tti
no-aa
tbaa
scoped-noalias
assumption-
  cache-tracker
basicaa
ipsccp
globalopt
deadargelim
domtree
instcombine
simplifycfg
basiccg
prune-eh
inline-cost
inline
functionattrs
domtree
sroa
early-cse
lazy-value-info
jump-threading
correlated-
  propagation
simplifycfg
domtree
instcombine
```

```
tailcallelim
simplifycfg
reassociate
domtree
loops
loop-simplify
lcssa
loop-rotate
licm
loop-unswitch
instcombine
scalar-evolution
loop-simplify
lcssa
indvars
loop-idiom
loop-deletion
loop-unroll
mldst-motion
domtree
memdep
gvn
memdep
memcpyopt
sccp
domtree
bdce
instcombine
lazy-value-info
```

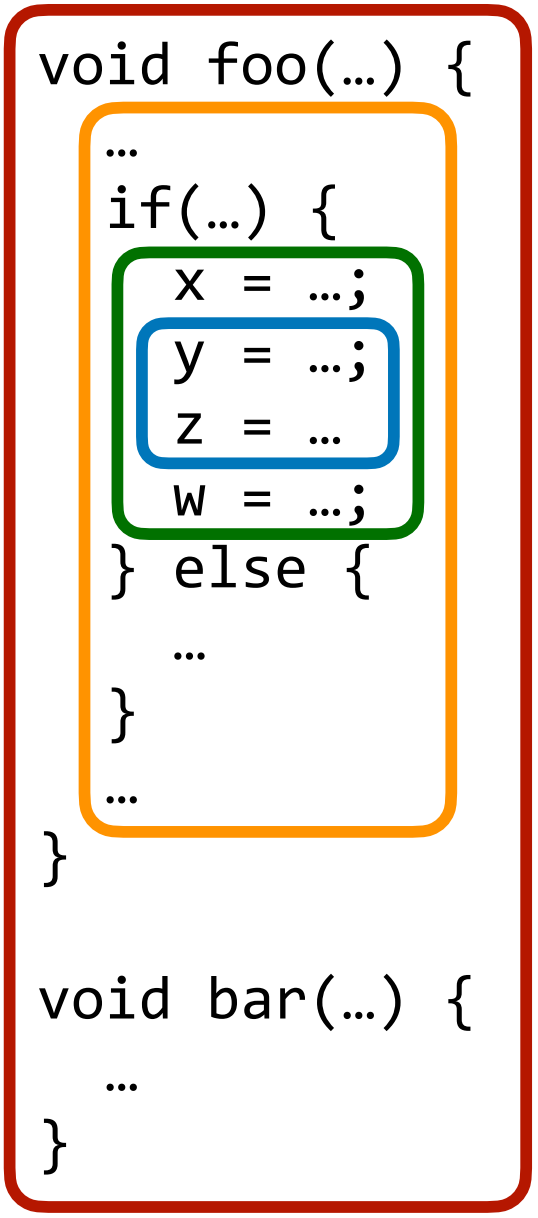
```
jump-threading
correlated-
propagation
domtree
memdep
dse
loops
loop-simplify
lcssa
licm
adce
simplifycfg
domtree
instcombine
barrier
float2int
domtree
loops
loop-simplify
lcssa
loop-rotate
branch-prob
block-freq
scalar-evolution
loop-accesses
loop-vectorize
instcombine
scalar-evolution
slp-vectorizer
```

```
simplifycfg
domtree
instcombine
loops
loop-simplify
lcssa
scalar-evolution
loop-unroll
instcombine
loop-simplify
lcssa
licm
scalar-evolution
alignment-from-
  assumptions
strip-dead-
  prototypes
elim-avail-
  extern
globaldce
constmerge
verify
```

It should start to make more sense now why LLVM has so many optimization passes that it runs.

Scope of Optimization

- **Peephole**
  - look at a **short window** of instructions
- **Local**
  - look at a whole **basic block**
- **Intra-procedural**
  - look at a **whole function**
- **Inter-procedural**
  - look at the **"whole program"**
  - closed-world (nothing new linked) vs. open-world (need to support linking against unknown code)

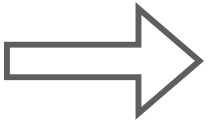


Peephole Optimizations

- After target code generation (so possibly quite late!) look at adjacent instructions (a "peephole") and try to replace a short sequence with a faster sequence
- e.g. (a form of copy propagation/forwarding)

  this eliminates an unnecessary load

```
movq %r9,16(%rsp)
movq 16(%rsp),%r12
```

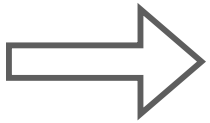


```
movq %r9,16(%rsp)
movq %r9,%r12
```

- Jump chaining can also be considered a form of peephole optimization (a peephole on a possible sequence of instructions)

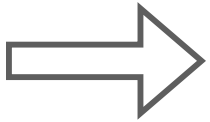
Another peephole example

```
subq $8,%rax
movq %r2,0(%rax)
movq $0,%rax
```



```
movq %r2,-8(%rax)
movq $0,%rax
```

```
movq 16(%rsp),%rax
subq $1,%rax
movq %rax,16(%rsp)
movq $0,%rax
```



```
decq 16(%rsp)
movq $0,%rax
```

- This is one way to do complex instruction selection
  - ✦ We'll see a more systematic method towards the end of the course

Algebraic Simplification

- Using laws of algebra to rewrite code, including “constant folding” and “strength reduction”
  - `z = 3 + 4;` ⇝ `z = 7;`
  - `z = x + 0;` ⇝ `z = x;`
  - `z = x * 1;` ⇝ `z = x;`
  - `z = x * 2;` ⇝ `z = x << 1;` (or `z = x + x`)
  - `z = x * 8;` ⇝ `z = x << 3;`
  - `z = x / 1;` ⇝ `z = x >> 3;` (if `x >= 0`)
  - `z = (x+y)-y;` ⇝ `z = x;` (depends on type/overflow)
- Can be done at almost any level
- Why would anyone write code like this?
  - generated during a lowering; easier to clean up after

Outline

Intro to Optimization

Local Optimization

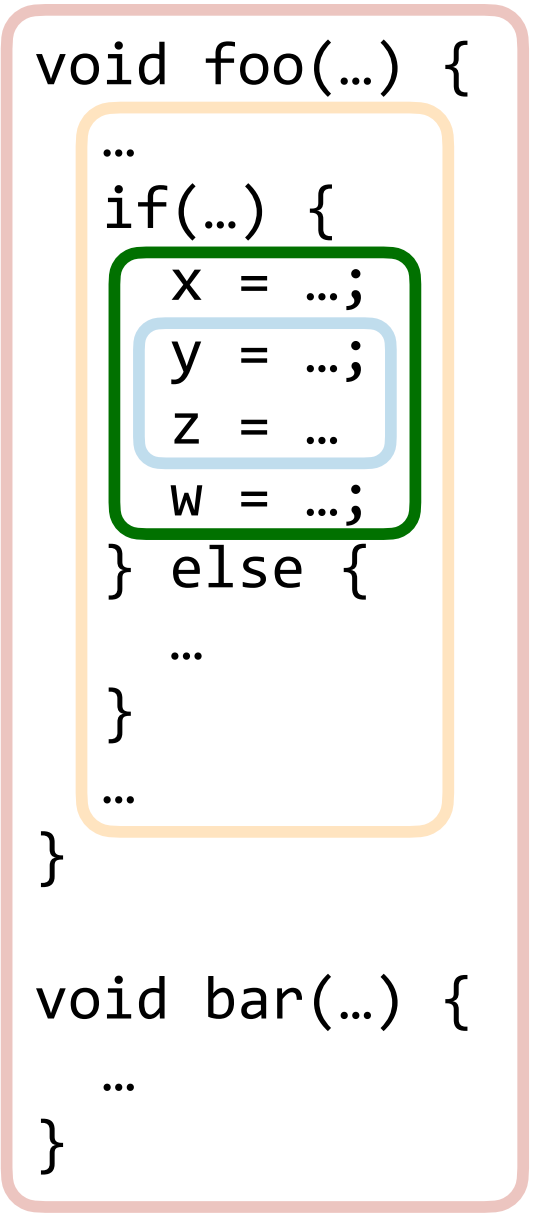
Intra-Procedural Optimization

Inter-Procedural Optimization

Data-Structures, Analysis, Transformation

Local Optimization

- **Peephole**
  - look at a **short window** of instructions
- **Local**
  - look at a whole **basic block**
- **Intra-procedural**
  - look at a **whole function**
- **Inter-procedural**
  - look at the **"whole program"**
  - closed-world (nothing new linked) vs. open-world (need to support linking against unknown code)



Local Const. Propagation

- If x ← const, then we can replace x by const in later instructions
- best done in tandem with constant folding

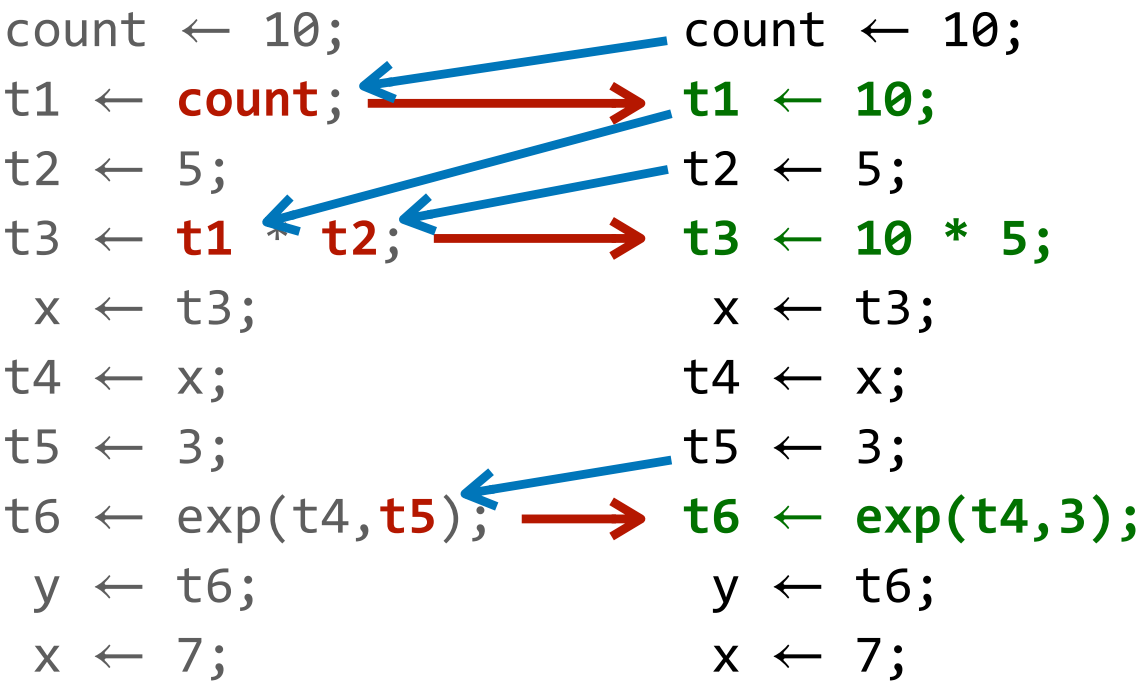
```
count ← 10;
t1 ← count;
t2 ← 5;
t3 ← t1 * t2;
 x ← t3;
t4 ← x;
t5 ← 3;
t6 ← exp(t4,t5);
 y ← t6;
 x ← 7;
```

```
count = 10;
… // count
  // not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

Local Const. Propagation

- If x ← const, then we can replace x by const in later instructions
- best done in tandem with constant folding

Propagate




```
count = 10;
… // count
  // not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

Local Const. Propagation

- If x ← const, then we can replace x by const in later instructions
- best done in tandem with constant folding

Fold

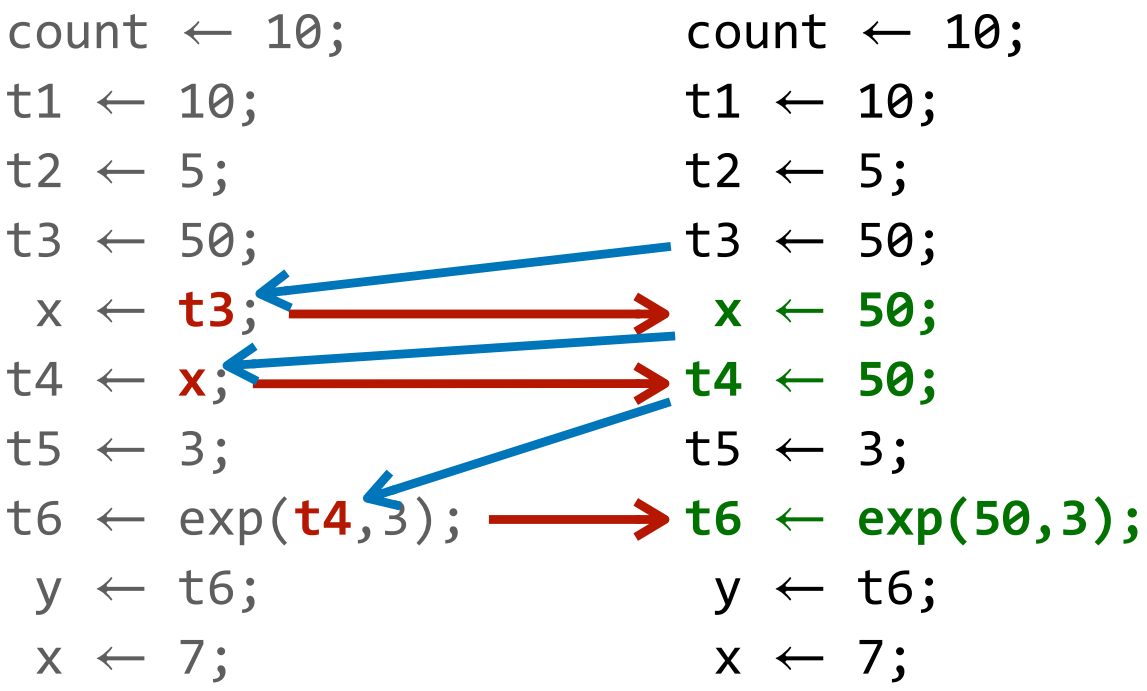
```
count ← 10;          count ← 10;
t1 ← 10;             t1 ← 10;
t2 ← 5;              t2 ← 5;
t3 ← 10 * 5;   ⟶     t3 ← 50;
 x ← t3;              x ← t3;
t4 ← x;              t4 ← x;
t5 ← 3;              t5 ← 3;
t6 ← exp(t4,3);      t6 ← exp(t4,3);
 y ← t6;              y ← t6;
 x ← 7;               x ← 7;
```

```
count = 10;
… // count
  // not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

Local Const. Propagation

- If x ← const, then we can replace x by const in later instructions
- best done in tandem with constant folding

Propagate




```
count = 10;
… // count
  // not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

Local Const. Propagation

- If x ← const, then we can replace x by const in later instructions
- best done in tandem with constant folding

Fold

```
count ← 10;          count ← 10;
t1 ← 10;             t1 ← 10;
t2 ← 5;              t2 ← 5;
t3 ← 50;             t3 ← 50;
 x ← 50;              x ← 50;
t4 ← 50;             t4 ← 50;
t5 ← 3;              t5 ← 3;
t6 ← exp(50,3);  →   t6 ← 125000;
 y ← t6;              y ← t6;
 x ← 7;               x ← 7;
```

```
count = 10;
… // count
  // not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

Local Const. Propagation

- If x ← const, then we can replace x by const in later instructions
- best done in tandem with constant folding

Propagate

```
count ← 10;          count ← 10;
t1 ← 10;             t1 ← 10;
t2 ← 5;              t2 ← 5;
t3 ← 50;             t3 ← 50;
 x ← 50;              x ← 50;
t4 ← 50;             t4 ← 50;
t5 ← 3;              t5 ← 3;
t6 ← 125000;         t6 ← 125000;
 y ← t6;              y ← 125000;
 x ← 7;               x ← 7;
```

```
count = 10;
… // count
  // not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

Local Dead Code Elimination

- If the LHS of an assignment is never used, then the assignment can be deleted
- Careful! "never used" requires us to know that a variable can't ***escape*** the scope we're analyzing it at

```
count ← 10;
t1 ← 10;
t2 ← 5;
t3 ← 50;
 x ← 50;
t4 ← 50;
t5 ← 3;
t6 ← 125000;
 y ← 125000;
 x ← 7;
```

```
count = 10;
… // count
  // not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

Local Dead Code Elimination

- If the LHS of an assignment is never used, then the assignment can be deleted
- Careful! "never used" requires us to know that a variable can't ***escape*** the scope we're analyzing it at

```
count ← 10;
t1 ← 10;
t2 ← 5;
t3 ← 50;
 x ← 50;
t4 ← 50;
t5 ← 3;
t6 ← 125000;
 y ← 125000;
 x ← 7;
```

```
count ← 10;
    y ← 125000;
    x ← 7;
```

```
count = 10;
… // count
  // not changed
x = count * 5;
y = x ^ 3;
x = 7;
```

Local CSE

Common Subexpression Elimination

- Find and eliminate repetitions of the same computation
  - ✦ requires that no side effects happen

```
… a[i] + b[i] …
```

```
t1 ← *(fp + ioffset);
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);
t5 ← *(fp + ioffset);
t6 ← t5 * 4;
t7 ← fp + t6;
t8 ← *(t7 + boffset);
t9 ← t4 + t8;
```

Local CSE

Common Subexpression Elimination

- Find and eliminate repetitions of the same computation
  - ✦ requires that no side effects happen

CSE

`… a[i] + b[i] …`

```
t1 ← *(fp + ioffset);          t1 ← *(fp + ioffset);
t2 ← t1 * 4;                   t2 ← t1 * 4;
t3 ← fp + t2;                  t3 ← fp + t2;
t4 ← *(t3 + aoffset);          t4 ← *(t3 + aoffset);
t5 ← *(fp + ioffset);    →     t5 ← t1;
t6 ← t5 * 4;                   t6 ← t5 * 4;
t7 ← fp + t6;                  t7 ← fp + t6;
t8 ← *(t7 + boffset);          t8 ← *(t7 + boffset);
t9 ← t4 + t8;                  t9 ← t4 + t8;
```

Local CSE

Common Subexpression Elimination

- Find and eliminate repetitions of the same computation
  - ✦ requires that no side effects happen

Copy Prop.

```
… a[i] + b[i] …
```

```
t1 ← *(fp + ioffset);
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);
t5 ← t1;
t6 ← t5 * 4;
t7 ← fp + t6;
t8 ← *(t7 + boffset);
t9 ← t4 + t8;
```

```
t1 ← *(fp + ioffset);
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);
t5 ← t1;
t6 ← t1 * 4;
t7 ← fp + t6;
t8 ← *(t7 + boffset);
t9 ← t4 + t8;
```

Local CSE

Common Subexpression Elimination

- Find and eliminate repetitions of the same computation
  - ✦ requires that no side effects happen

CSE + Copy Prop.

```
… a[i] + b[i] …
```

```
t1 ← *(fp + ioffset);
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);
t5 ← t1;
t6 ← t1 * 4;
t7 ← fp + t6;
t8 ← *(t7 + boffset);
t9 ← t4 + t8;
```

```
t1 ← *(fp + ioffset);
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);
t5 ← t1;
t6 ← t2;
t7 ← fp + t2;
t8 ← *(t7 + boffset);
t9 ← t4 + t8;
```

Local CSE

Common Subexpression Elimination

- Find and eliminate repetitions of the same computation
  - ✦ requires that no side effects happen

CSE + Copy Prop.

```
… a[i] + b[i] …
```

```
t1 ← *(fp + ioffset);
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);
t5 ← t1;
t6 ← t2;
t7 ← fp + t2;
t8 ← *(t7 + boffset);
t9 ← t4 + t8;
```

```
t1 ← *(fp + ioffset);
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);
t5 ← t1;
t6 ← t2;
t7 ← t3;
t8 ← *(t3 + boffset);
t9 ← t4 + t8;
```

Local CSE

Common Subexpression Elimination

- Find and eliminate repetitions of the same computation
  - ✦ requires that no side effects happen

Dead Code Elim.

```
… a[i] + b[i] …
```

```
t1 ← *(fp + ioffset);
t2 ← t1 * 4;
t3 ← fp + t2;
t4 ← *(t3 + aoffset);
t5 ← t1;
t6 ← t2;
t7 ← t3;
t8 ← *(t3 + boffset);
t9 ← t4 + t8;
```

Outline

Intro to Optimization

Local Optimization

Intra-Procedural Optimization

Inter-Procedural Optimization

Data-Structures, Analysis, Transformation

Intra-procedural Optimization

- **Peephole**
  - look at a **short window** of instructions
- **Local**
  - look at a whole **basic block**
- **Intra-procedural**
  - look at a **whole function**
- **Inter-procedural**
  - look at the **"whole program"**
  - closed-world (nothing new linked) vs. open-world (need to support linking against unknown code)

```
void foo(…) {
  …
  if(…) {
    x = …;
    y = …;
    z = …
    w = …;
  } else {
    …
  }
  …
}

void bar(…) {
  …
}
```

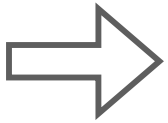
The Intra-procedural level

- Unlike basic-blocks (local), we now have to deal with jumps (loops, branches, etc.)
- Most local optimizations can be extended to work at this level instead
  - ✦ This is the most common/frequent level to perform optimizations at in compilers (e.g. -O2)
- There are some unique opportunities at this new level
  - ✦ Most important — Loop optimizations
  - ✦ the code inside loops makes the biggest impact on performance, & thus is the most important to optimize

Loop Invariant Code Motion

- move computation that doesn't change from loop iteration to loop iteration (i.e. "loop invariant") out of the loops
- Can do this in many different IRs (e.g. on AST vs. middle 3AC vs. backend assembly)

```
for (i=0; i<10; i=i+1) {
  a[i] = a[i] + b[j];
  z = z + 10000;
}
```



```
t1 = b[j];
t2 = 10000;
for (i=0; i<10; i=i+1) {
  a[i] = a[i] + t1;
  z = z + t2;
}
```

Loop Invariant Code Motion

```
for (i=0; i<10; i=i+1) {
  a[i] = b[j];
}
```

```
  *(fp + ioffset) ← 0;
top:
  t0 ← *(fp + ioffset);
  iffalse (t0 < 10) goto done;
  t1 ← *(fp + joffset);
  t2 ← t1 * 4;
  t3 ← fp + boffset;
  t4 ← *(t3 + t2);
  t5 ← *(fp + ioffset);
  t6 ← t5 * 4;
  t7 ← fp + aoffset;
  *(t7 + t6) ← t4;
  t8 ← *(fp + ioffset);
  t9 ← t8 + 1;
  *(fp + ioffset) ← t9;
  goto top;
done:
```

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  *(fp + ioffset) ← 0;
top:
  t0 ← *(fp + ioffset);
  iffalse (t0 < 10) goto done;
  t1 ← *(fp + joffset);
  t2 ← t1 * 4;
  t3 ← fp + boffset;
  t4 ← *(t3 + t2);
  t5 ← *(fp + ioffset);
  t6 ← t5 * 4;
  t7 ← fp + aoffset;
  *(t7 + t6) ← t4;
  t8 ← *(fp + ioffset);
  t9 ← t8 + 1;
  *(fp + ioffset) ← t9;
  goto top;
done:
```

Loop Invariant Code Motion

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  *(fp + ioffset) ← 0;
top:
  t0 ← *(fp + ioffset);
  iffalse (t0 < 10) goto done;
  t1 ← *(fp + joffset);
  t2 ← t1 * 4;
  t3 ← fp + boffset;
  t4 ← *(t3 + t2);
  t5 ← *(fp + ioffset);
  t6 ← t5 * 4;
  t7 ← fp + aoffset;
  *(t7 + t6) ← t4;
  t8 ← *(fp + ioffset);
  t9 ← t8 + 1;
  *(fp + ioffset) ← t9;
  goto top;
done:
```

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  *(t11) ← 0;
top:
  t0 ← *(t11);
  iffalse (t0 < 10) goto done;
  t1 ← *(t12);
  t2 ← t1 * 4;
  t3 ← t13;
  t4 ← *(t3 + t2);
  t5 ← *(t11);
  t6 ← t5 * 4;
  t7 ← t14;
  *(t7 + t6) ← t4;
  t8 ← *(t11);
  t9 ← t8 + 1;
  *(t11) ← t9;
  goto top;
done:
```

Copy Prop.

Loop Invariant Code Motion

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  *(t11) ← 0;
top:
  t0 ← *(t11);
  iffalse (t0 < 10) goto done;
  t1 ← *(t12);
  t2 ← t1 * 4;
  t3 ← t13;
  t4 ← *(t3 + t2);
  t5 ← *(t11);
  t6 ← t5 * 4;
  t7 ← t14;
  *(t7 + t6) ← t4;
  t8 ← *(t11);
  t9 ← t8 + 1;
  *(t11) ← t9;
  goto top;
done:
```

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  *(t11) ← 0;
top:
  t0 ← *(t11);
  iffalse (t0 < 10) goto done;
  t1 ← *(t12);
  t2 ← t1 * 4;
  t3 ← t13;
  t4 ← *(t13 + t2);
  t5 ← *(t11);
  t6 ← t5 * 4;
  t7 ← t14;
  *(t14 + t6) ← t4;
  t8 ← *(t11);
  t9 ← t8 + 1;
  *(t11) ← t9;
  goto top;
done:
```

Copy Prop.

DCE

Loop Invariant Code Motion

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  *(t11) ← 0;
top:
  t0 ← *(t11);
  iffalse (t0 < 10) goto done;
  t1 ← *(t12);
  t2 ← t1 * 4;
  t3 ← t13;
  t4 ← *(t13 + t2);
  t5 ← *(t11);
  t6 ← t5 * 4;
  t7 ← t14;
  *(t14 + t6) ← t4;
  t8 ← *(t11);
  t9 ← t8 + 1;
  *(t11) ← t9;
  goto top;
done:
```

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  t1 ← *(t12);
  t2 ← t1 * 4;
  *(t11) ← 0;
top:
  t0 ← *(t11);
  iffalse (t0 < 10) goto done;
  t4 ← *(t13 + t2);
  t5 ← *(t11);
  t6 ← t5 * 4;
  *(t14 + t6) ← t4;
  t8 ← *(t11);
  t9 ← t8 + 1;
  *(t11) ← t9;
  goto top;
done:
```

Loop-Invariant Code Motion

Loop Invariant Code Motion

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  t1 ← *(t12);
  t2 ← t1 * 4;
  *(t11) ← 0;
top:
  t0 ← *(t11);
  iffalse (t0 < 10) goto done;
  t4 ← *(t13 + t2);
  t5 ← *(t11);
  t6 ← t5 * 4;
  *(t14 + t6) ← t4;
  t8 ← *(t11);
  t9 ← t8 + 1;
  *(t11) ← t9;
  goto top;
done:
```

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  t1 ← *(t12);
  t2 ← t1 * 4;
  t4 ← *(t13 + t2);
  *(t11) ← 0;
top:
  t0 ← *(t11);
  iffalse (t0 < 10) goto done;
  t5 ← *(t11);
  t6 ← t5 * 4;
  *(t14 + t6) ← t4;
  t8 ← *(t11);
  t9 ← t8 + 1;
  *(t11) ← t9;
  goto top;
done:
```

Loop-Invariant Code Motion

Loop Invariant Code Motion

```
for (i=0; i<10; i=i+1) {
  a[i] = b[j];
}
```

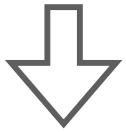
```
  *(fp + ioffset) ← 0;
top:
  t0 ← *(fp + ioffset);
  iffalse (t0 < 10) goto done;
  t1 ← *(fp + joffset);
  t2 ← t1 * 4;
  t3 ← fp + boffset;
  t4 ← *(t3 + t2);
  t5 ← *(fp + ioffset);
  t6 ← t5 * 4;
  t7 ← fp + aoffset;
  *(t7 + t6) ← t4;
  t8 ← *(fp + ioffset);
  t9 ← t8 + 1;
  *(fp + ioffset) ← t9;
  goto top;
done:
```

```
  t11 ← fp + ioffset;
  t12 ← fp + joffset;
  t13 ← fp + boffset;
  t14 ← fp + aoffset;
  t1 ← *(t12);
  t2 ← t1 * 4;
  t4 ← *(t13 + t2);
  *(t11) ← 0;
top:
  t0 ← *(t11);
  iffalse (t0 < 10) goto done;
  t5 ← *(t11);
  t6 ← t5 * 4;
  *(t14 + t6) ← t4;
  t8 ← *(t11);
  t9 ← t8 + 1;
  *(t11) ← t9;
  goto top;
done:
```

Loop Induction Variable Elimination

- A common special case of loop-based strength reduction
- For-loops generally have an ***induction variable***
  - Incremented each time around the loop
  - e.g. `for (int i = 0; i < 10; i = i+1) {…}`
- If only used to index arrays, we can rewrite to instead increment those pointers directly
  - compute initial pointers before loop, then increment each time around
  - reduces amount and cost of index math in loop

```
for (i=0; i<10; i=i+1) {
  a[i] = a[i] + x;
}
```



```
for(p = &a[0];
    p < &a[10]; p=p+4) {
  *p = *p + x;
}
```

Outline

Intro to Optimization

Local Optimization

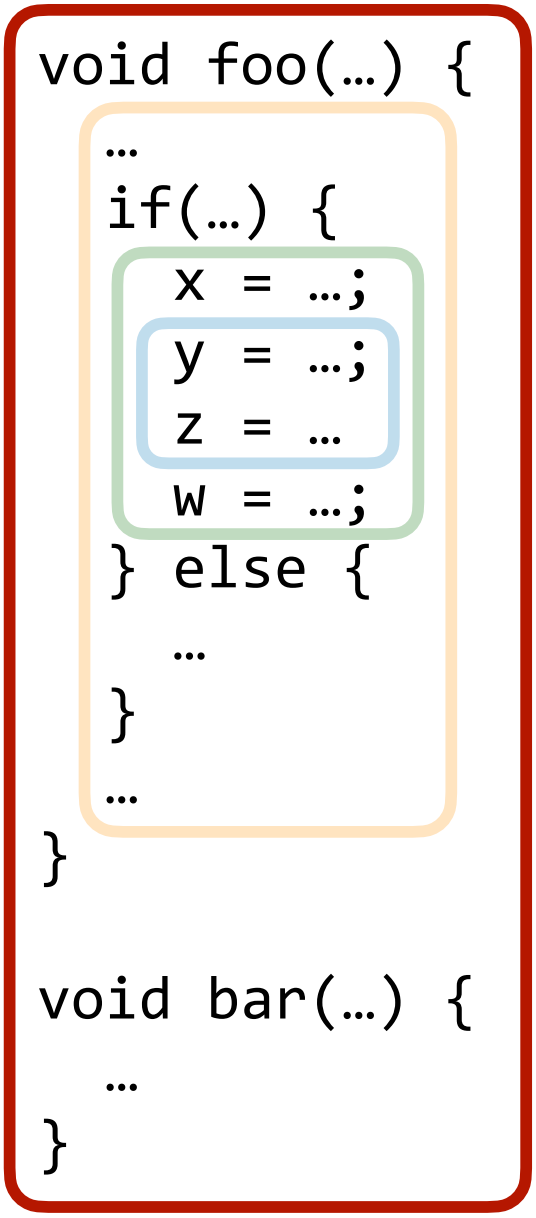
Intra-Procedural Optimization

Inter-Procedural Optimization

Data-Structures, Analysis, Transformation

Intra-procedural Optimization

- **Peephole**
  - look at a **short window** of instructions
- **Local**
  - look at a whole **basic block**
- **Intra-procedural**
  - look at a **whole function**
- **Inter-procedural**
  - look at the **"whole program"**
  - closed-world (nothing new linked) vs. open-world (need to support linking against unknown code)



Inlining: Replace Call with Body

- Most common way to achieve inter-procedural optimizations (basically, reduce to intraprocedural)
- Source

```
final double pi = 3.1415927;
double circleArea(double radius) {
    return pi * (radius * radius);
}
…
    double r = 5.0;
    double a = circleArea(r);
```

- After Inlining

```
    double r = 5.0;
    double a = pi * r * r;
```

Actually, closer to this

```
double t = r;
double a = pi * t * t;
```

i.e. we need to be careful about scopes and names

Outline

Intro to Optimization

Local Optimization

Intra-Procedural Optimization

Inter-Procedural Optimization

Data-Structures, Analysis, Transformation

Program Analysis

- Optimizations have various preconditions that have to be met in order for them to be applied
  - ✦ Is variable x used later on?
  - ✦ Where does the value of x flow to?
  - ✦ Is y constant? (and of what value?)
  - ✦ was the expression “a + b” computed earlier?
  - ✦ Can statements 1 & 2 be reordered?
- We need some systematic ways to answer such questions

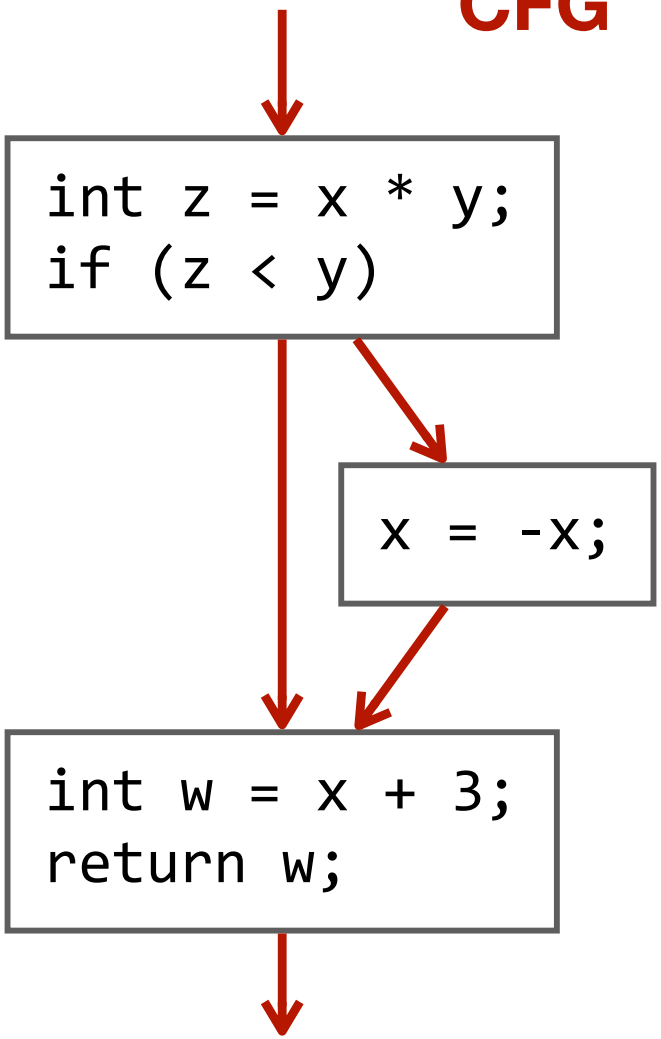
Data Structures (Graph IRs)

- 3 big graph IRs we can use to analyze code
- **Control Flow Graphs** — paths in the graph represent possible program trajectories
- **Data Flow Graphs** — paths represent possible flows of data
- **Dependency Graphs** — ***the absence*** of a path from A to B indicates that changes to A ***cannot influence*** B
- Static Single Assignment (SSA) form will also be a very useful *linear IR* — will cover next week

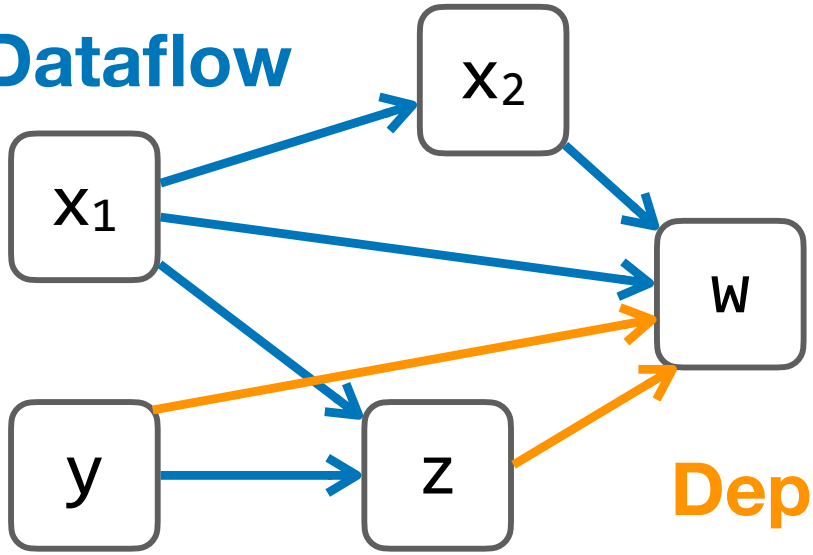
3 Different Graphs

```
public int foo(int x, int y) {
  int z = x * y;
  if (z < y) {
    x = -x;
  }
  int w = x + 3;
  return w;
}
```

CFG



Dataflow



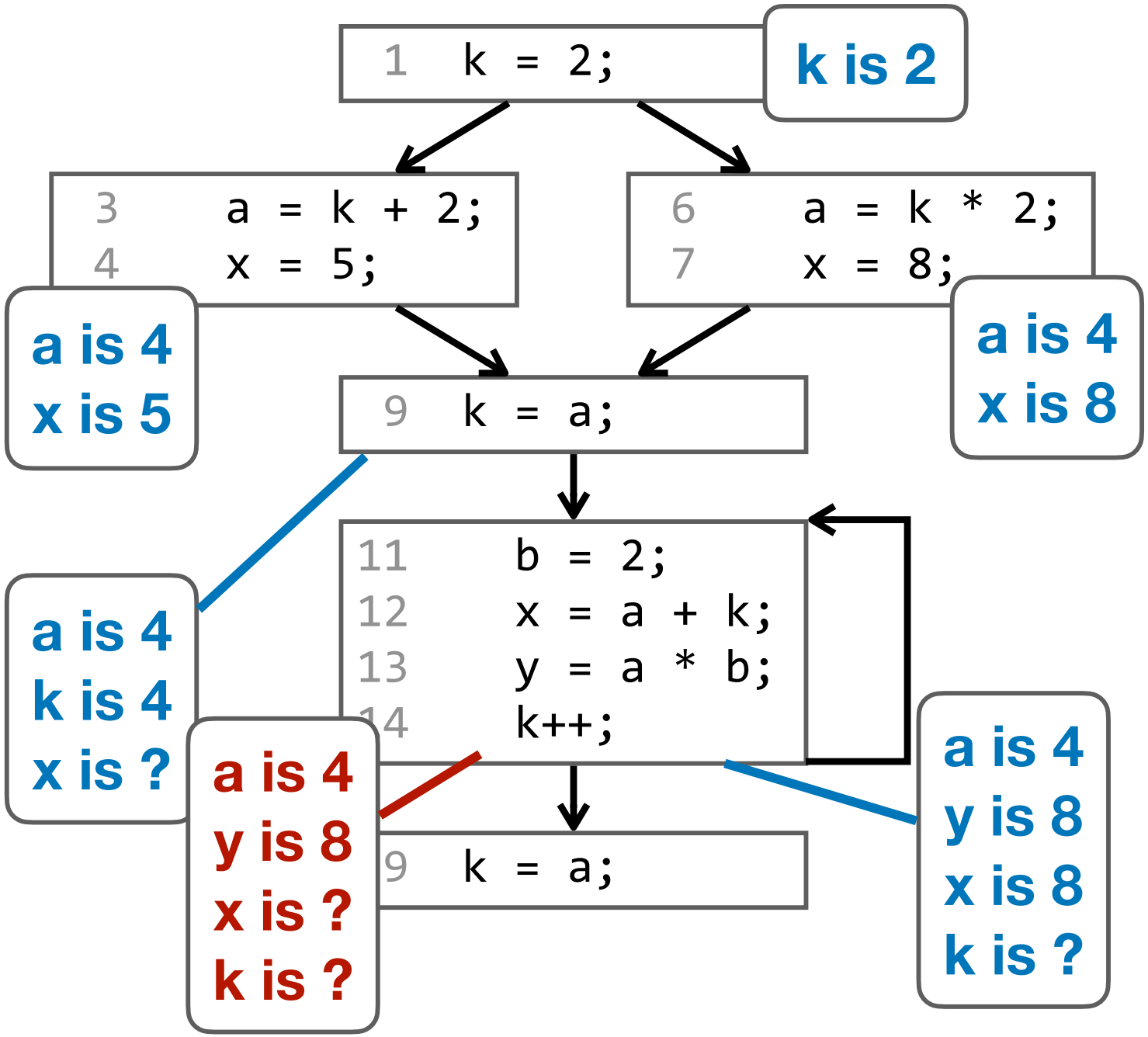
Dependency

Preview — Analysis

- Many analyses work by pushing values through the program
  - i.e. a ***non-standard interpretation*** of the program
- The values being pushed through are stand-ins for ***properties about the program***, e.g.
  - "this variable is always 3"
  - "the variables x,y and z are defined"
- Analyses make ***conservative approximations*** of these properties rather than trying to be perfectly precise
  - e.g. "this variable is positive" ***over-approximates*** "this variable must always be 3"

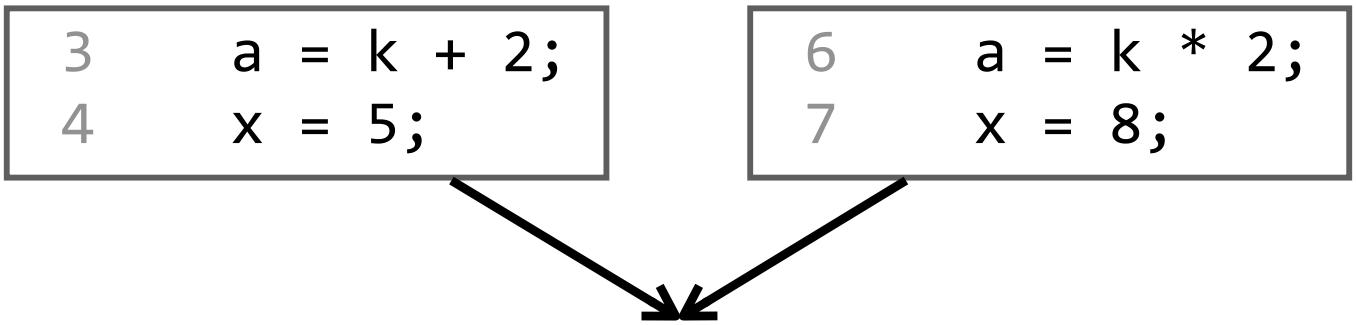
Preview — Constants

```
1   k = 2;
2   if (...) {
3     a = k + 2;
4     x = 5;
5   } else {
6     a = k * 2;
7     x = 8;
8   }
9   k = a;
10  while (...) {
11    b = 2;
12    x = a + k;
13    y = a * b;
14    k++;
15  }
16  print(a+x);
```



Merging Values

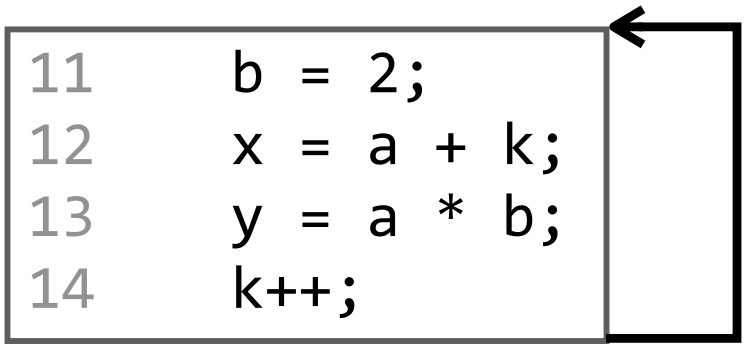
- When values from two different branches flow into the same point, we ***merge*** them
- e.g.



- What is the value of x?
- Unknown! Not Constant!

Loops

- When we encounter a loop, we keep running it over and over again (merging when coming in from the back-edge)
- e.g.



a is 4
y is 8
x is 8
k is ?

a is 4
y is 8
x is ?
k is ?

a is 4
y is 8
x is ?
k is ?

- When the values stop changing, we can continue
- How do we know whether or not the values will stop changing…

Next Time…

- Wednesday
  - ✦ Bootstrapping for Codegen
  - ✦ Dataflow Analysis (start)
- Friday
  - ✦ Main Course of Dataflow Analysis
- Monday — Holiday!
- Next Wednesday — SSA, an important IR
- Then backend and end of the quarter!