

$$\#1 = t$$

$$\#2 = \frac{t(1+\sqrt{2})}{3}$$

$$\#3 = \frac{t\sqrt{5}}{3}$$

When I'm walking,
I worry a lot
about the efficiency
of my path.

Local Optimizations

One-Slide Summary

- An **optimization** changes a program so that it computes the **same answer** in **less time** (or using less of some other resource).
- We represent the program using a special **intermediate form**.
- Each method is viewed as a **control flow graph** where the nodes are **basic blocks** of instructions with **known entry** and **exit** points. The instructions have been changed so that a **single assignment** defines each variable.

Review: Dispatch

$so, E, S \vdash e_1 : v_1, S_1$

$so, E, S_1 \vdash e_2 : v_2, S_2$

...

$so, E, S_{n-1} \vdash e_n : v_n, S_n$

$so, E, S_n \vdash e_0 : v_0, S_{n+1}$

$v_0 = X(a_1 = l_1, \dots, a_m = l_m)$

$imp(X, f) = (x_1, \dots, x_n, e_{body})$

$l_{xi} = newloc(S_{n+1})$ for $i = 1, \dots, n$

$E' = [x_1 : l_{x1}, \dots, x_n : l_{xn}, a_1 : l_1, \dots, a_m : l_m]$

$S_{n+2} = S_{n+1}[v_1/l_{x1}, \dots, v_n/l_{xn}]$

$v_0, E', S_{n+2} \vdash e_{body} : v, S_{n+3}$

Evaluate arguments

Evaluate receiver object

Find type and attributes

Find formals and body

*New
environment*

New store

Evaluate body

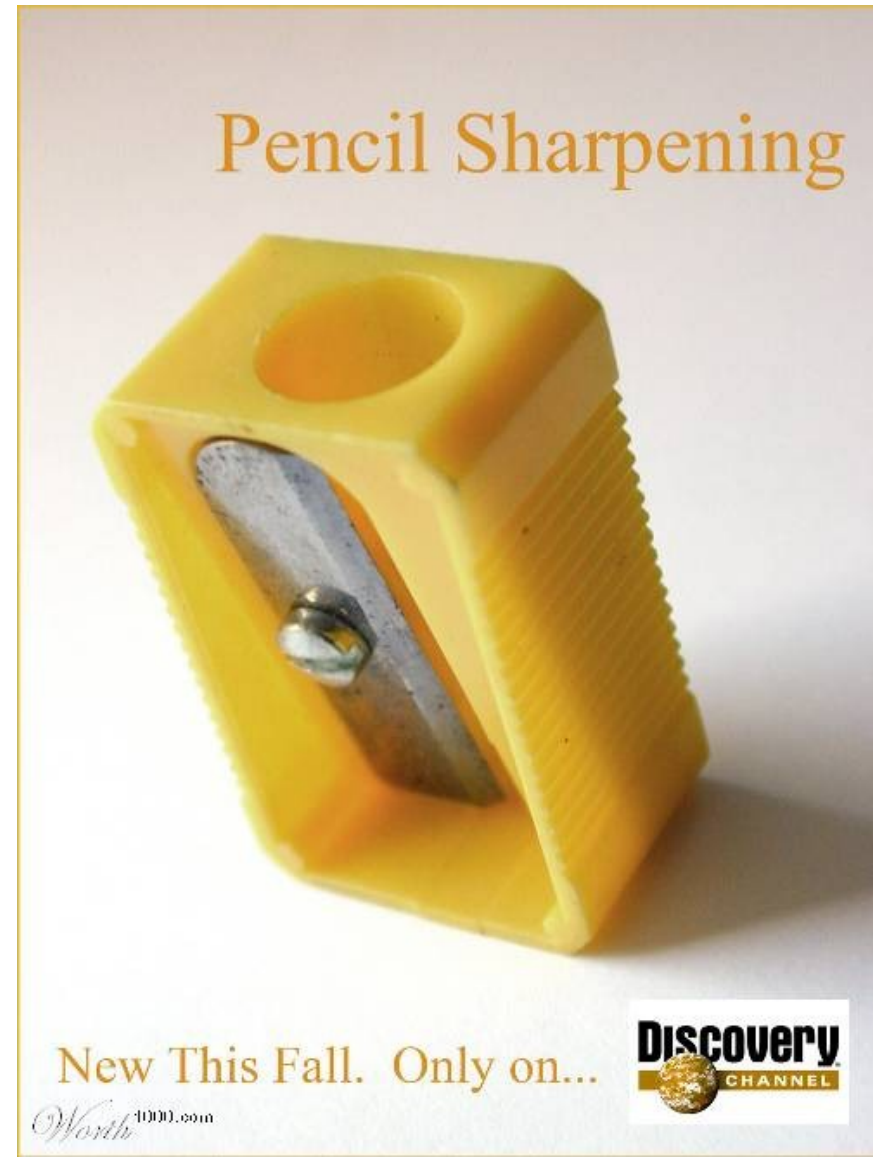
$so, E, S \vdash e_0.f(e_1, \dots, e_n) : v, S_{n+3}$

Operational Semantics: Summary

- Operational rules are very **precise**
 - Nothing is left unspecified
- Operational rules contain a lot of details
 - Read them **carefully**
- Most languages do not have a well specified operational semantics (Ruby!)
- When portability is important an operational semantics becomes essential
 - But not always using the exact notation we used for Cool

Lecture Outline

- Intermediate code
- Local optimizations
- Next time: larger-scale program analyses



Why Optimize?

- What's the point?
- Do we care about this in real life?



When To Optimize?

- When to perform optimizations:
 - On AST (just like type checking)
 - **Pro**: Machine independent
 - **Cons**: Too high level
 - On assembly language (compilers only)
 - **Pro**: Exposes optimization opportunities
 - **Cons**: Machine dependent
 - **Cons**: Must reimplement optimizations when retargeting
 - On an intermediate language
 - **Pro**: Machine independent
 - **Pro**: Exposes optimization opportunities
 - **Cons**: One more language to worry about

Intermediate Languages

- Each compiler uses its own **intermediate language**
 - IL design is still an active area of research
- Intermediate language = high-level assembly language
 - Uses register names, but has an unlimited number
 - Uses control structures like assembly language
 - Uses opcodes but some are higher level
 - e.g., **push** translates to several assembly instructions
 - Most opcodes correspond directly to assembly opcodes

Three-Address Intermediate Code

- Each instruction is of the form

$$x := y \text{ op } z$$

- y and z can be only registers, variables or constants
- Common form of intermediate code
- The AST expression $x + y * z$ is translated as
$$t_1 := y * z$$
$$t_2 := x + t_1$$
 - Each **subexpression** lives in a temporary

Generating Intermediate Code

- $\text{igen}(e, t)$ function generates code to compute the value of e in register t

- Example:

$\text{igen}(e_1 + e_2, t) =$

$\text{igen}(e_1, t_1)$ (t_1 is a fresh register)

$\text{igen}(e_2, t_2)$ (t_2 is a fresh register)

$t := t_1 + t_2$

- Unlimited number of registers
 \Rightarrow simple code generation

An Intermediate Language

$P \rightarrow S P \mid \varepsilon$
 $S \rightarrow id := id \ op \ id$
| $id := op \ id$
| $id := id$
| $push \ id$
| $id := pop$
| $if \ id \ relop \ id \ goto \ L$
| $L:$
| $jump \ L$

- ids are register names
- Constants can replace ids
- Typical operators: +, -, *

Basic Blocks

- A **basic block** is a *maximal* sequence of instructions with:
 - no labels (except at the first instruction), and
 - no jumps (except in the last instruction)
- Idea:
 - Cannot jump into a basic block (*except* at beginning)
 - Cannot jump out of a basic block (*except* at end)
 - Each instruction in a basic block is executed after all the preceding instructions have been executed (**this is the important concept**)

Basic Block Example

- Consider the basic block
 1. L1:
 2. $t := 2 * x$
 3. $w := t + x$
 4. if $w > 0$ goto L2
- No way for (3) to be executed without (2) having been executed right before

Basic Block Example

- Consider the basic block
 1. L1:
 2. $t := 2 * x$
 3. $w := t + x$
 4. if $w > 0$ goto L2
- No way for (3) to be executed without (2) having been executed right before
 - We can change (3) to $w := 3 * x$

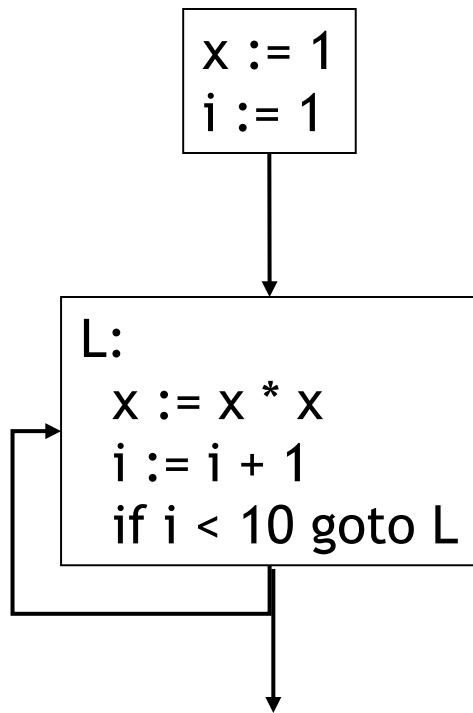
Basic Block Example

- Consider the basic block
 1. L1:
 2. $t := 2 * x$
 3. $w := t + x$
 4. if $w > 0$ goto L2
- No way for (3) to be executed without (2) having been executed right before
 - We can change (3) to $w := 3 * x$
 - Can we eliminate (2) as well?

Control-Flow Graphs

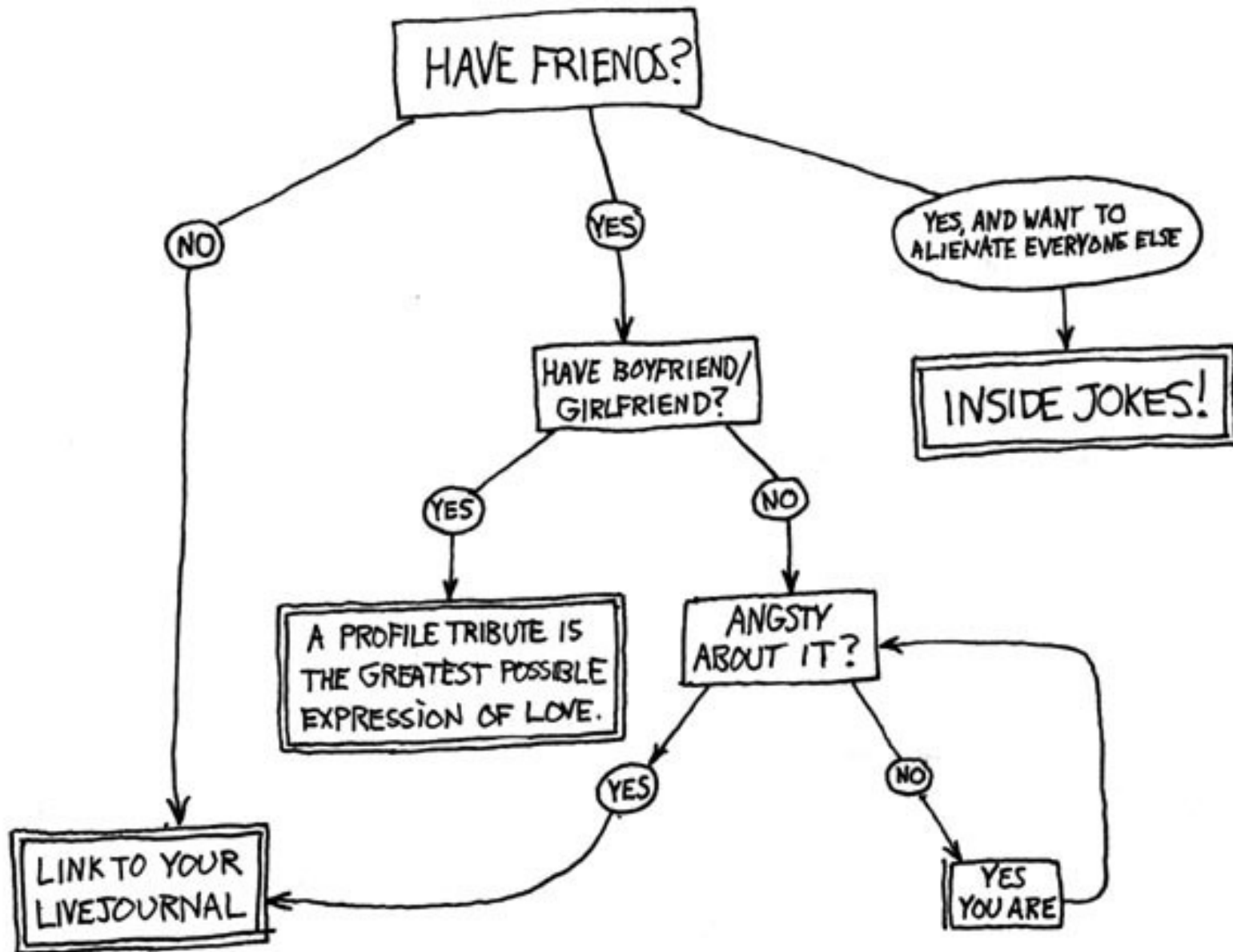
- A **control-flow graph** is a directed graph:
 - Basic blocks as **nodes**
 - An **edge** from block A to block B if the execution can (potentially) flow from the last instruction in A to the first instruction in B
 - e.g., the last instruction in A is **jump L_B**
 - e.g., the execution can fall-through from block A to block B
- Frequently abbreviated as **CFG**

Control-Flow Graphs. Example.



- The body of a method (or procedure) can be represented as a control-flow graph
- There is one initial node
 - The “start node”
- All “return” nodes are terminal

CREATING AN AIM PROFILE:



CFG
≈
Flow
Chart

Optimization Overview

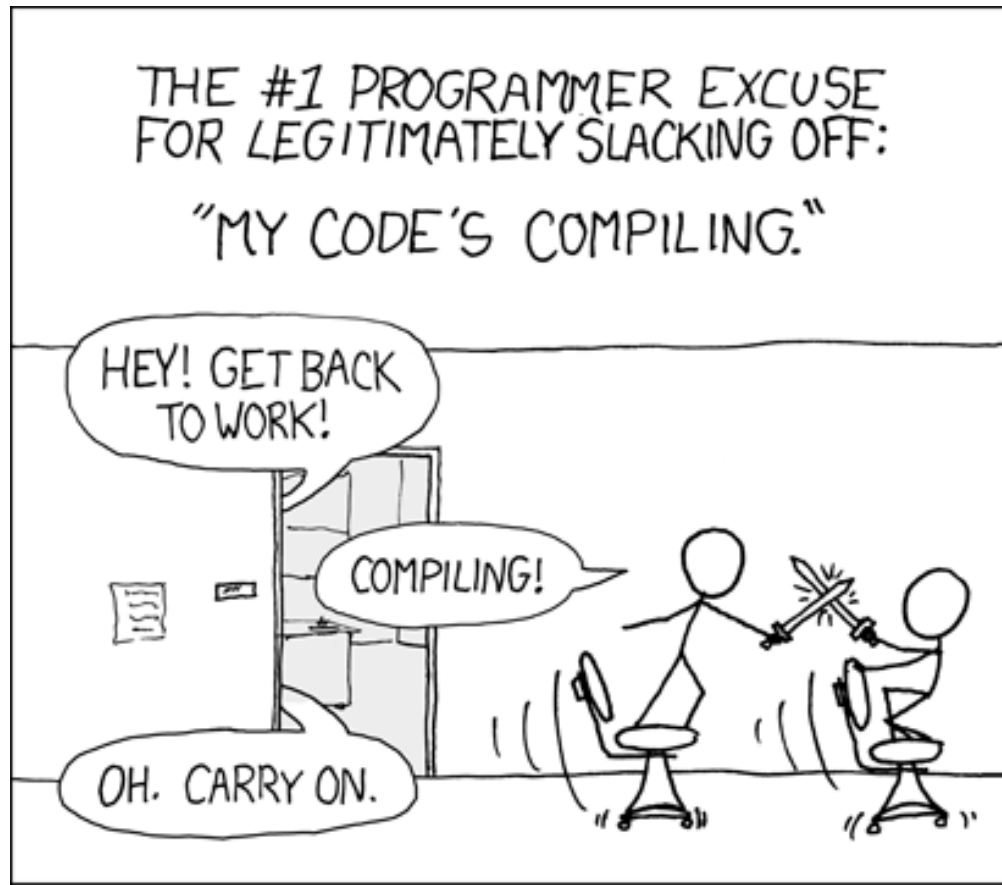
- **Optimization** seeks to improve a program's utilization of some resource
 - Execution time (most often)
 - Code size
 - Network messages sent
 - Battery power used, etc.
- Optimization should *not* alter what the program computes
 - The answer must still be the same
 - **First Rule of Optimization Club: Don't Break The Build**

A Classification of Optimizations

- For languages like C and Cool there are three granularities of optimizations
 1. **Local optimizations**
 - Apply to a basic block in isolation
 2. **Global optimizations**
 - Apply to a control-flow graph (method body) in isolation
 3. **Inter-procedural optimizations**
 - Apply across method boundaries
- Most **compilers** do (1), many do (2) and very few do (3)
- Some **interpreters** do (1), few do (2), basically none do (3)

Cost of Optimizations

- In practice, a conscious decision is made *not* to implement the fanciest optimization known
- Why?



Cost of Optimizations

- In practice, a conscious decision is made *not* to implement the fanciest optimization known
- Why?
 - Some optimizations are hard to implement
 - Some optimizations are costly in terms of compilation/interpretation time
 - The fancy optimizations are both hard and costly
- The goal: maximum improvement with minimum of cost (no risk!)

Q: Movies (363 / 842)

- This 1993 US National Film Registry inductee, a favorite of many Buddhists, "begins" with the following radio banter: *"Rise and shine, campers, and don't forget your booties 'cause it's coooold out there today. / It's cold out there every day. What is this, Miami Beach? / Not hardly. So the big question on everybody's lips / -- On their chapped lips -- / their chapped lips is, does Phil feel lucky?"*

Q: Books (763 / 842)

- This 18th- and 19th-century English poet, painter and religious printmaker believed in racial and sexual equality and rejected imposed secular authority. Peter Marshall described him as, "a revolutionary anarchist, [...] anticipating modern anarchism and social ecology. With William Godwin, he stands as a great forerunner of British Anarchism". His poem **Night** includes: "*The sun descending in the west, / The evening star does shine; / The birds are silent in their nest. / And I must seek for mine.*"

Q: Books (764 / 842)

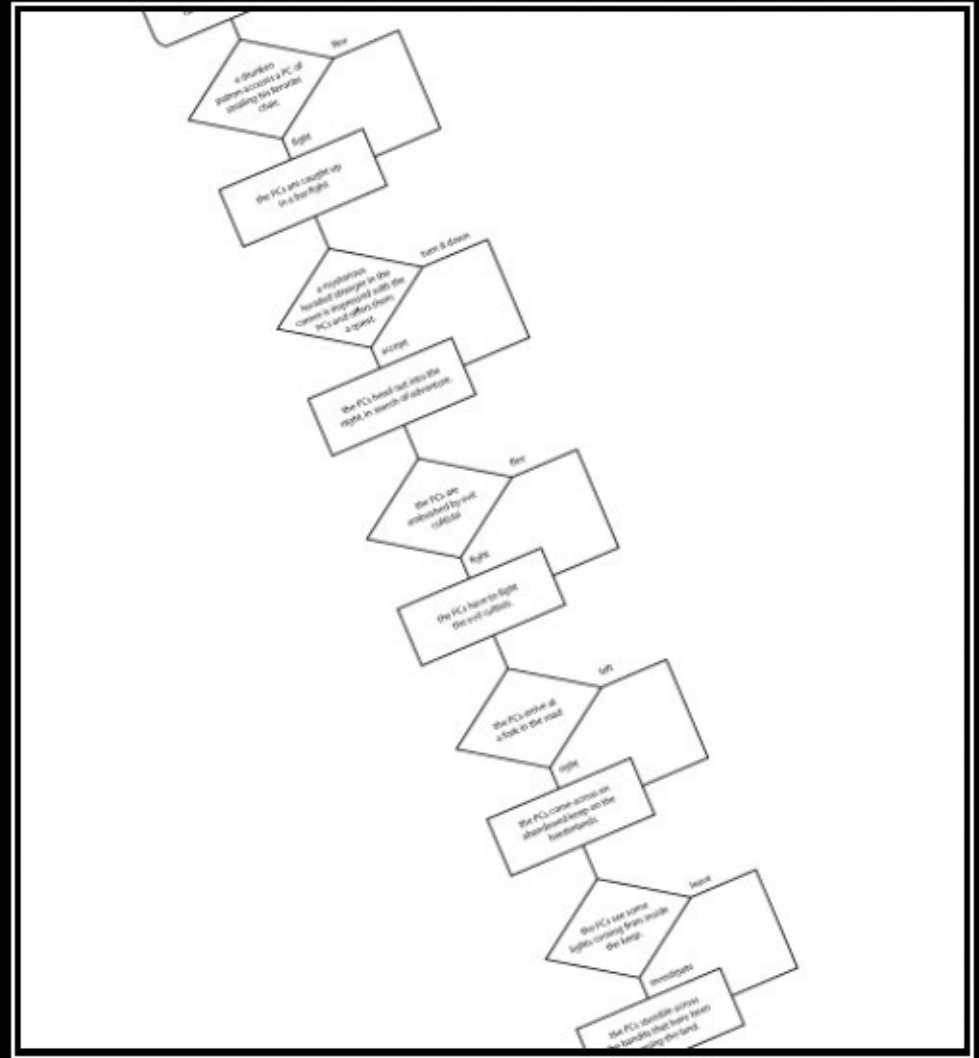
- Name the 1957 novel and the controlling protagonist associated with one of the following three companies: **Taggart Transcontinental, Rearden Steel and d'Anconia Copper.**

Real World Languages

- This language, with 5.2 million speakers, is often written either in Cyrillic or its own script. It features vowel harmony and suffices for verbs and nouns. Word order is relatively free and verbs change with voice, aspect, tense, and epistemic modality. The language is on the decline with younger speakers in favor of Mandarin Chinese. The earliest surviving text is perhaps the Stele of Yisüngge, a report on sports from 1224.

CFG

- This CFG stuff sounds complicated ...
- Can't we skip it for now?



ILLUSIONISM

Always give the players a choice as long as it's your choice.

Local Optimizations

- The simplest form of optimizations
- No need to analyze the whole procedure body
 - Just the basic block in question
- Example:
 - algebraic simplification
 - constant folding
 - Python 2.5+ does stuff like this if you say “-0”

Algebraic Simplification

- Some statements can be deleted

$x := x + 0$

$x := x * 1$

- Some statements can be simplified

$x := x * 0 \quad \Rightarrow \quad x := 0$

$y := y ** 2 \quad \Rightarrow \quad y := y * y$

$x := x * 8 \quad \Rightarrow \quad x := x \ll 3$

$x := x * 15 \quad \Rightarrow \quad t := x \ll 4; x := t - x$

(on some machines \ll is faster than $*$; but not on all!)

Constant Folding

- Operations on constants can be computed before the code executes
- In general, if there is a statement
$$x := y \text{ op } z$$
 - And y and z are **constants**
 - Then $y \text{ op } z$ can be computed early
- Example: $x := 2 + 2 \Rightarrow x := 4$
- Example: $\text{if } 2 < 0 \text{ jump } L$ can be deleted
- When might **constant folding** be dangerous?

Flow of Control Optimizations

- Eliminating **unreachable code**:
 - Code that is unreachable in the control-flow graph
 - Basic blocks that are not the target of any jump or “fall through” from a conditional
 - Such basic blocks can be eliminated
- Why would such basic blocks occur?
- Removing unreachable code makes the program smaller
 - And sometimes also faster
 - Due to memory cache effects (increased spatial locality)

Single Assignment Form

- Most optimizations are simplified if each assignment is to a temporary that *has not appeared already* in the basic block
- Intermediate code can be rewritten to be in **single assignment form**

$x := a + y$

$a := x$

$x := a * x$

$b := x + a$

\Rightarrow

$x := a + y$

$a_1 := x$

$x_1 := a_1 * x$

$b := x_1 + a_1$

(x_1 and a_1 are fresh temporaries)

Single Assignment vs. Functional Programming

- In **functional programming** variable values do not change
- Instead you make a new variable with a similar name
- Single assignment form is just like that!

$x := a + y$		$\text{let } x = a + y \text{ in}$
$a_1 := x$	\simeq	$\text{let } a_1 = x \text{ in}$
$x_1 := a_1 * x$		$\text{let } x_1 = a_1 * x \text{ in}$
$b := x_1 + a_1$		$\text{let } b = x_1 + a_1 \text{ in}$

Common Subexpression Elimination

- Assume:
 - Basic block is in single assignment form
- Then all assignments with same right-hand side compute the same value (*why?*)

- Example:

$x := y + z$

...

$w := y + z$

\Rightarrow

$x := y + z$

...

$w := x$

- Why is single assignment important here?

Copy Propagation

- If $w := x$ appears in a block, all subsequent uses of w can be replaced with uses of x

- Example:

$b := z + y$		$b := z + y$
$a := b$	\Rightarrow	$a := b$
$x := 2 * a$		$x := 2 * b$

- This does not make the program smaller or faster but might enable other optimizations
 - Constant folding
 - Dead code elimination (we'll see this in a bit!)
- Again, single assignment is important here.

Copy Propagation and Constant Folding

- Example:

a := 5

x := 2 * a

y := x + 6

t := x * y

⇒

a := 5

x := 10

y := 16

t := x << 4

Dead Code Elimination

If

$w := rhs$ appears in a basic block

w does not appear anywhere else in the program

Then

the statement $w := rhs$ is dead and can be eliminated

- **Dead** = does not contribute to the program's result

Example: (a is not used anywhere else)

$x := z + y$	$b := z + y$	$b := z + y$
$a := x$	$\Rightarrow a := b$	$\Rightarrow x := 2 * b$
$x := 2 * a$	$x := 2 * b$	

Applying Local Optimizations

- Each local optimization does very little by itself
- Typically optimizations *interact*
 - Performing one optimizations enables other opts
- Typical optimizing compilers repeatedly perform optimizations until no improvement is possible
- Interpreters and JITs must be fast!
 - The optimizer can also be stopped at any time to limit the compilation time

An Example

- Initial code:

a := x ** 2

b := 3

c := x

d := c * c

e := b * 2

f := a + d

g := e * f

An Example

- Algebraic optimization:

a := x ** 2

b := 3

c := x

d := c * c

e := b * 2

f := a + d

g := e * f

An Example

- Algebraic optimization:

a := x * x

b := 3

c := x

d := c * c

e := b + b

f := a + d

g := e * f

An Example

- Copy propagation:

a := x * x

b := 3

c := x

d := c * c

e := b + b

f := a + d

g := e * f

An Example

- Copy propagation:

a := x * x

b := 3

c := x

d := x * x

e := 3 + 3

f := a + d

g := e * f

An Example

- Constant folding:

a := x * x

b := 3

c := x

d := x * x

e := 3 + 3

f := a + d

g := e * f

An Example

- Constant folding:

a := x * x

b := 3

c := x

d := x * x

e := 6

f := a + d

g := e * f

An Example

- Common subexpression elimination:

a := x * x

b := 3

c := x

d := x * x

e := 6

f := a + d

g := e * f

An Example

- Common subexpression elimination:

a := x * x

b := 3

c := x

d := a

e := 6

f := a + d

g := e * f

An Example

- Copy propagation:

a := x * x

b := 3

c := x

d := a

e := 6

f := a + d

g := e * f

An Example

- Copy propagation:

a := x * x

b := 3

c := x

d := a

e := 6

f := a + a

g := 6 * f

An Example

- Dead code elimination:

a := x * x

b := 3

c := x

d := a

e := 6

f := a + a

g := 6 * f

An Example

- Dead code elimination:

$a := x * x$



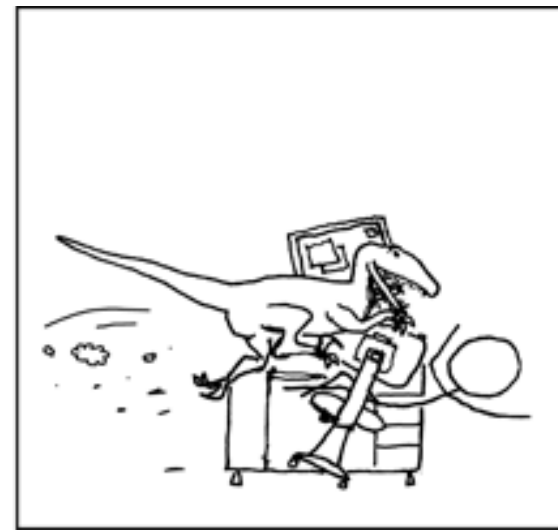
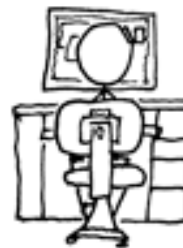
$f := a + a$

$g := 6 * f$

- This is the final form
 - Could we get to $g = 12 * a$?

Cool and Intermediate Form

- Cool does not have **goto**
- Cool does not have **break**
- Cool does not have **exceptions**
- How would you make basic blocks from a Cool AST?



Local Optimization Notes

- Intermediate code is helpful for many optimizations
 - Basic Blocks: known entry and exit
 - Single Assignment: one definition per variable
- “Program optimization” is grossly **misnamed**
 - Code produced by “optimizers” is not optimal in any reasonable sense
 - “Program improvement” is a more appropriate term
- Next: larger-scale program changes

Homework

- WA5 due today
- PA5t due today
- **Midterm 2** - Monday Apr 21 (21 days)