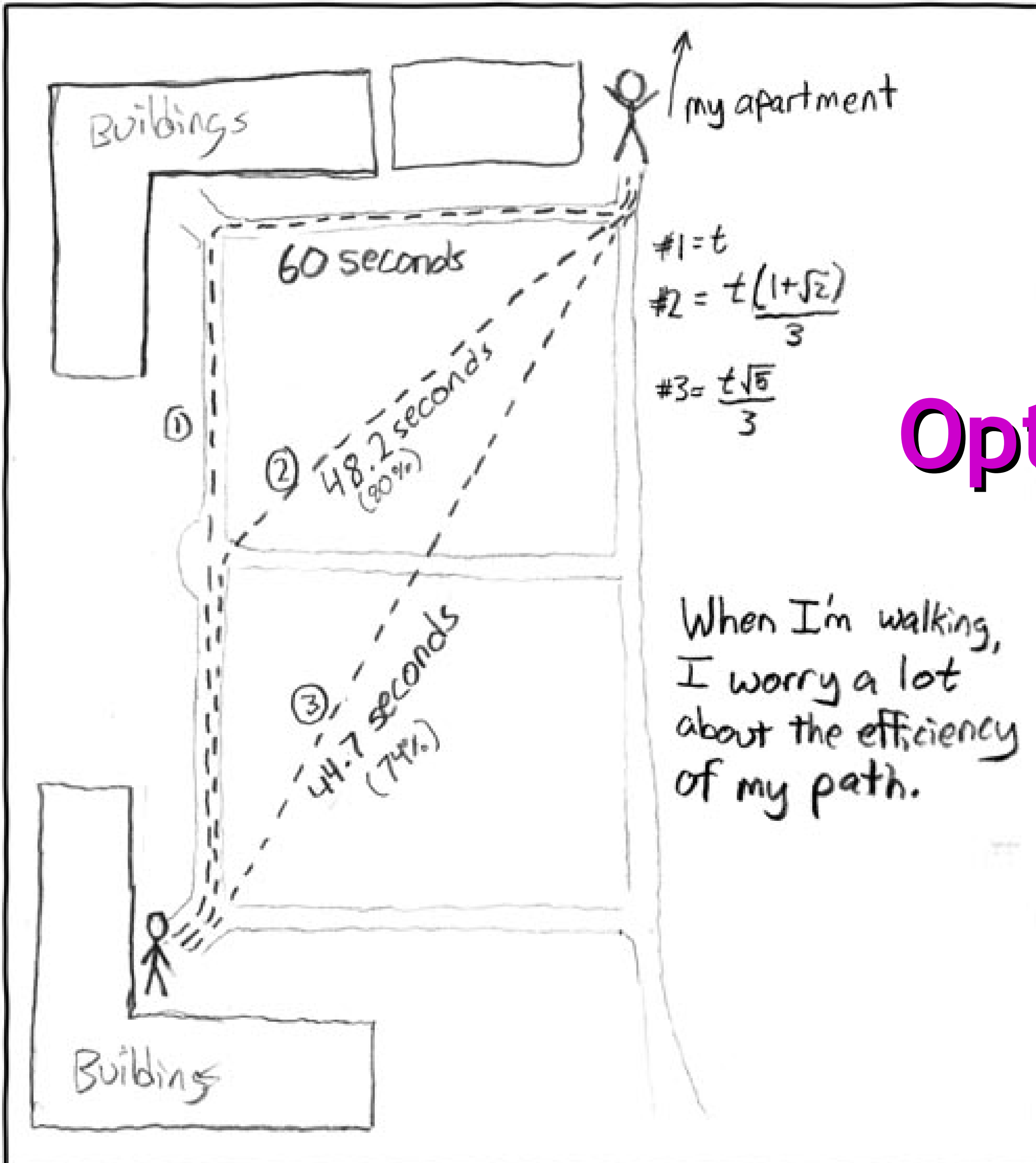


Local Optimizations

One-Slide Summary

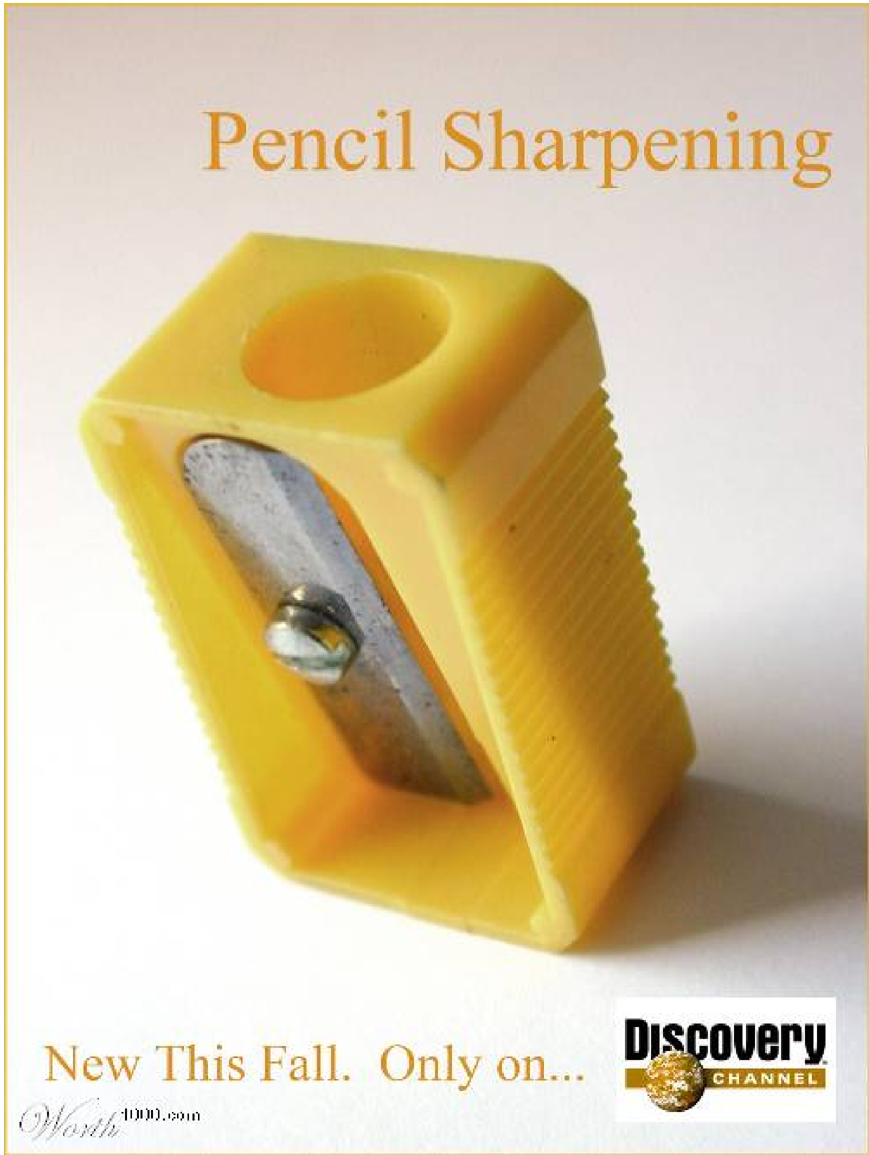
- An **optimization** changes a program so that it computes the same answer in less time (or using less of some other resource).
- We represent the program using a special **intermediate form**.
- Each method is viewed as a **control flow graph** where the nodes as **basic blocks** of instructions with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

Upcoming Events

- This Friday, 4pm, 228E/236D
  - "Halloween Fall Festival". Prizes for best costumes, trick-or-treats provided, bring an artistic dish if you like, apple dunking, music?
- **Postponed!**
  - Barbara Liskov lectures, www.liskovatuva.com
- Wednesday Nov 18, evening
  - Fireside chat with Kim "Reality TV" Hazelwood and Wes "Romance Novel" Weimer
- ???
  - The final for this class. Vote.

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 retargetting
  - 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

- igen(e, t) function generates code to compute the value of e in register t
- Example:

  igen(e_1 + e_2, t) =

  igen(e_1, t_1) *(t_1 is a fresh register)*

  igen(e_2, t_2) *(t_2 is a fresh register)*

  t := t_1 + t_2

- Unlimited number of registers

  ⇒ 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

- id's are register names
- Constants can replace id's
- 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

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

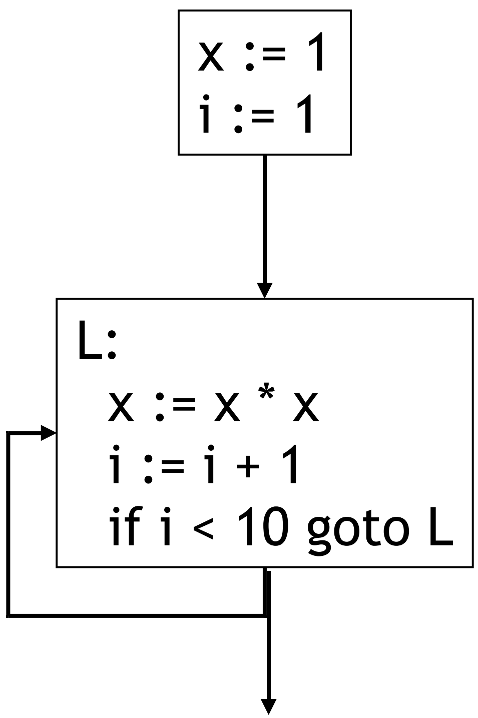
Basic Block Example

- 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?

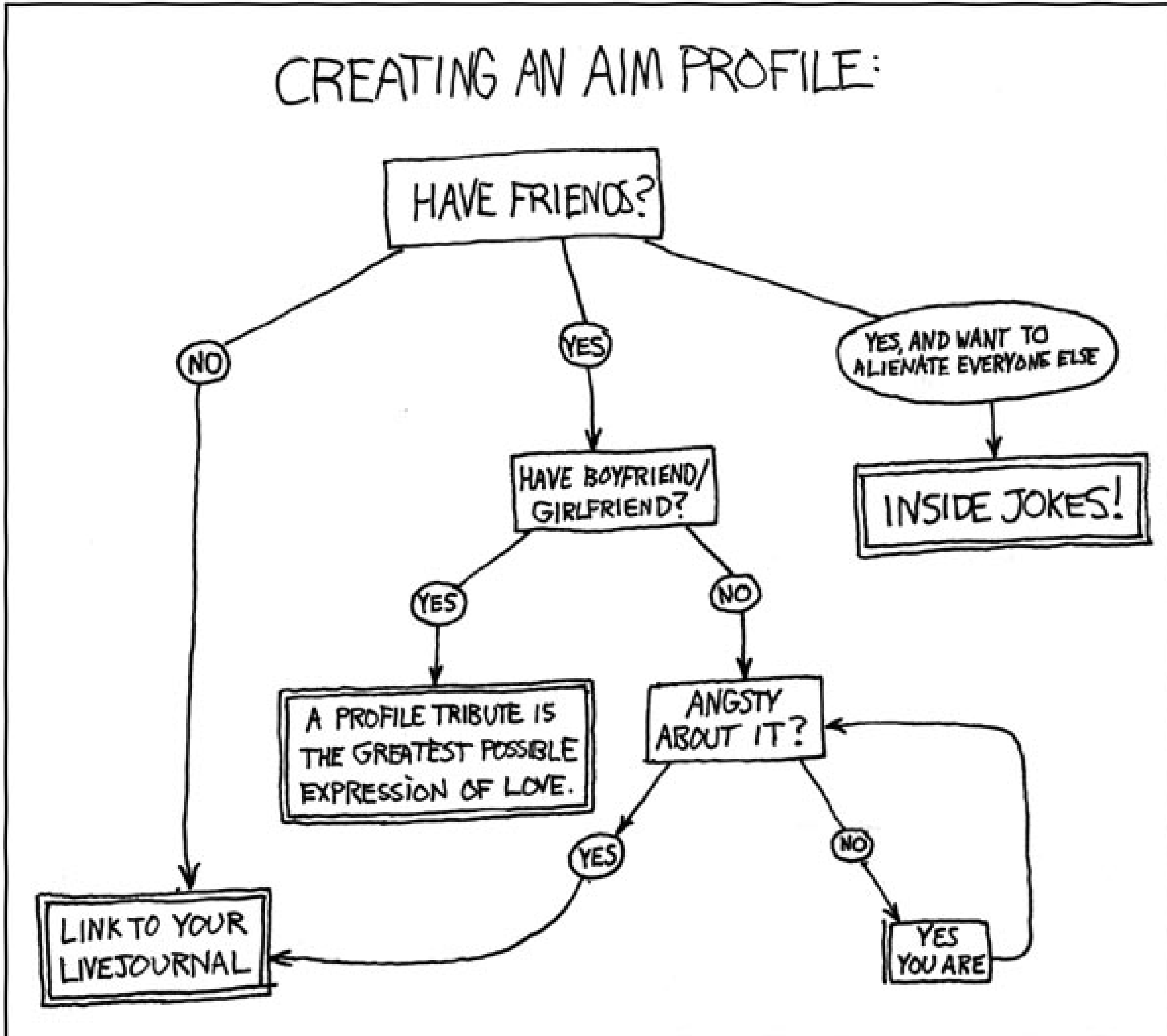
Control-Flow Graphs

- A **control-flow graph** is a directed graph:
  - Basic blocks as nodes
  - An edge from block A to block B if the execution can 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



CFG ≃ Flow Chart

Optimization Overview

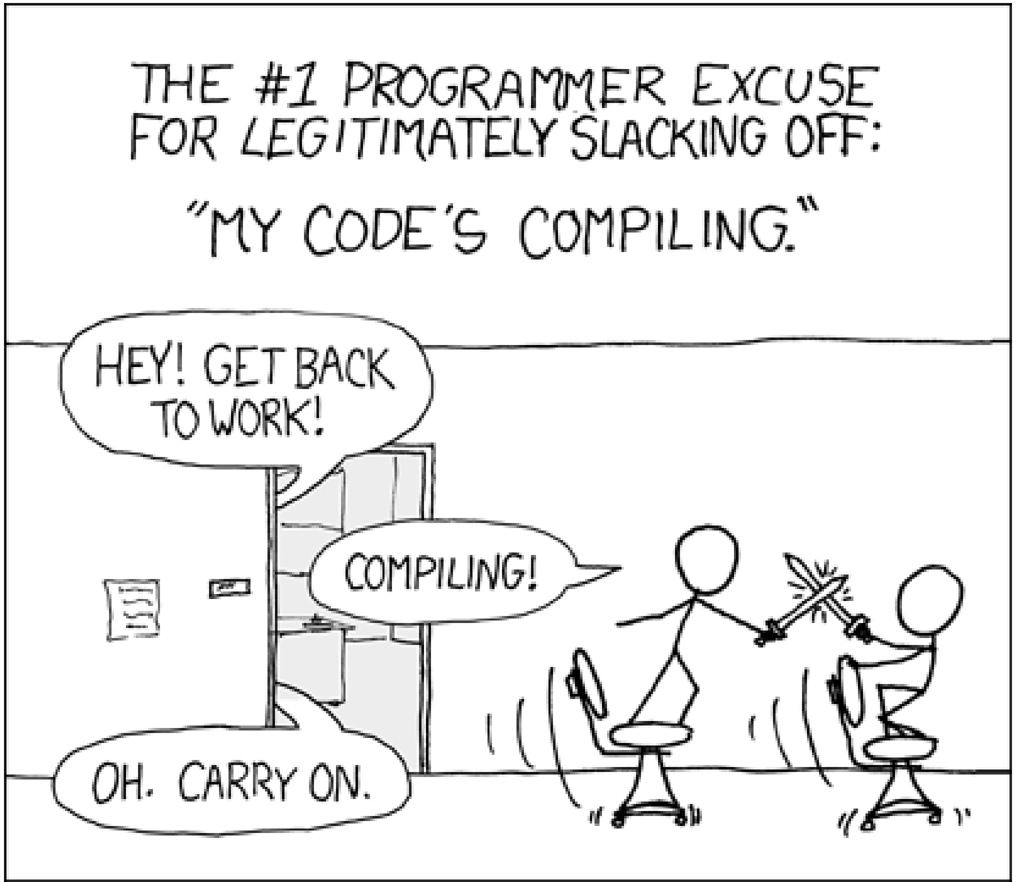
- **Optimization** seeks to improve a program's utilization of some resource
  - Execution time (most often)
  - Code size
  - Network messages sent
  - Battery power used, etc.
- Optimization should *not* alter what the program computes
  - The answer must still be the same
  - **First Rule of Optimization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

Q: Movies (363 / 842)

- This 1993 comedy film also starring Andie MacDowell "begins" with the following radio banter: *"Rise and shine, campers, and don't forget your booties 'cause it's cooooold 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: Cartoons (674 / 842)

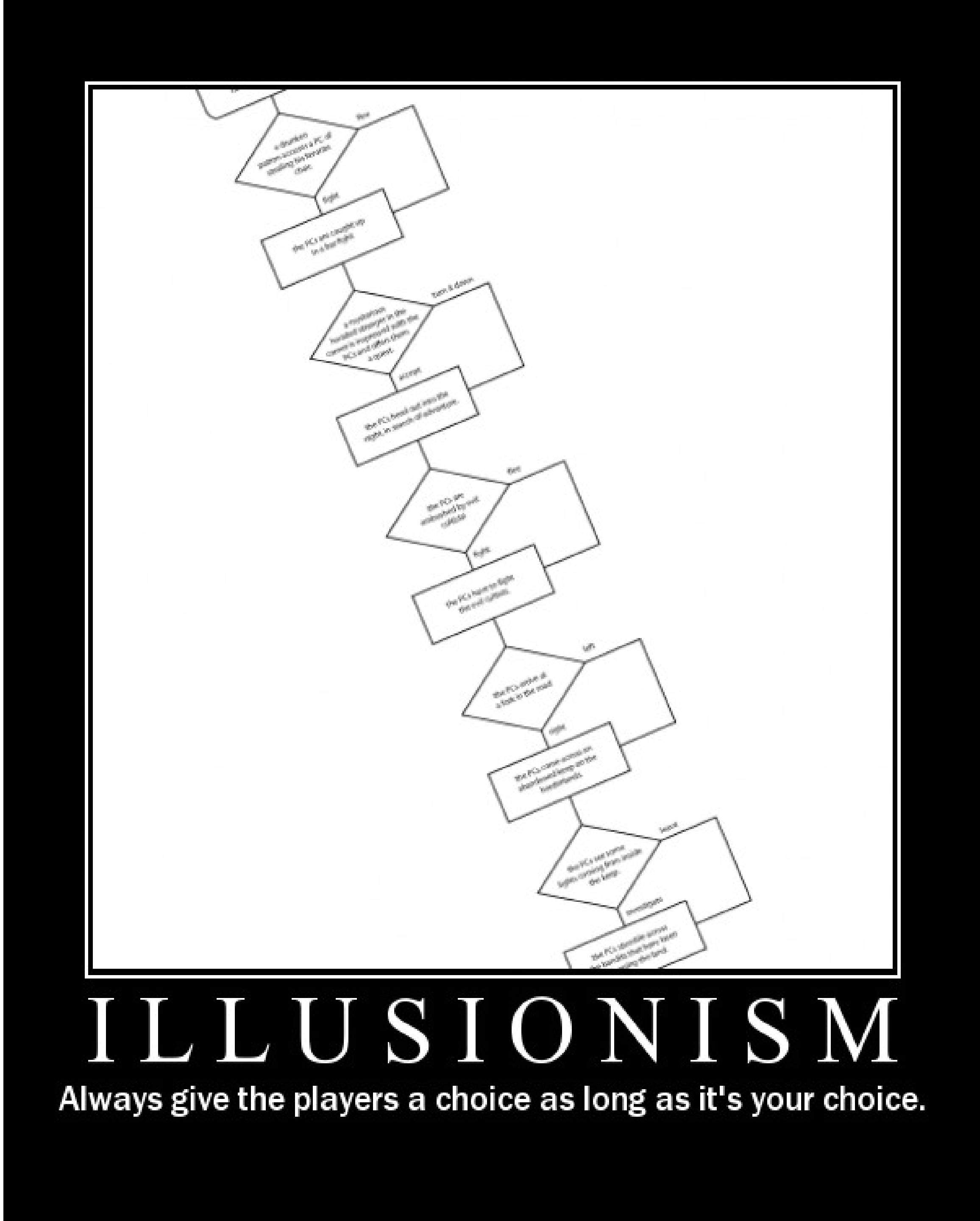
- This 1953 Warner Brothers' cartoon mouse is known for his cry of *"Arriba! Arriba! Andele!"*

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**.

CFG

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



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 "–O"

Algebraic Simplification

- Some statements can be deleted

  x := x + 0

  x := x * 1

- Some statements can be simplified

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

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

  x := x * 8 $\Rightarrow$ x := x << 3

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

  (on some machines << 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 op z

  - And y and z are constants
  - Then y op z can be computed early
- Example: x := 2 + 2 ⇒ x := 4
- Example: if 2 < 0 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		x := a + y
a := x	⇒	a_1 := x
x := a * x		x_1 := a_1 * x
b := x + a		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!

$x := a + y$		let $x = a + y$ in
$a_1 := x$	$\simeq$	let $a_1 = x$ in
$x_1 := a_1 * x$		let $x_1 = a_1 * x$ in
$b := x_1 + a_1$		let $b = x_1 + a_1$ 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                    x := y + z
…                 ⇒           …
w := 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       ⇒      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	⇒	a := b	⇒	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:

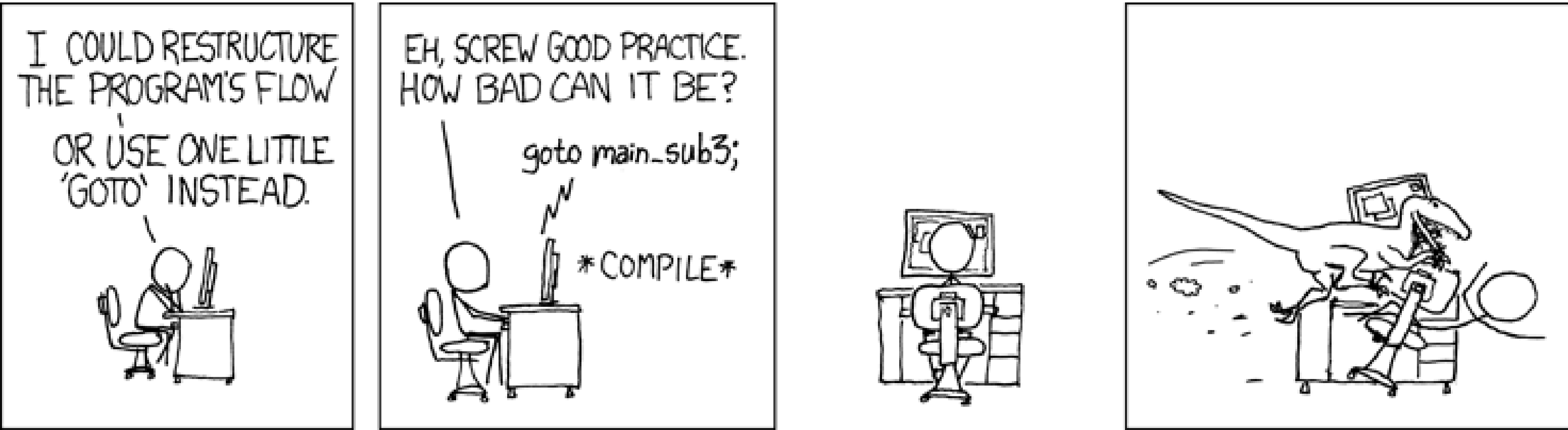
  a := x * x

  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
- WA6 due next Thursday
- Start PA5!
- Reading for next time (basic blocks, etc.)
- Midterm 2 – Tuesday Nov 24 (26 days)