Class 13: Miscellany, More Analysis

- Writing types
- Design recipe update
- Cond-case ordering
- Analysis of multiple versions of a single procedure
- Lambda
"Type" language: how do I write types?

• In Racket type signatures:
  num, bool, string,
  (num list), (bool list) (string list), ((num list) list), etc.
  (num -> bool), ((bool list) -> (string list)), etc.
• Note absence of hypens: never "num-list"
• I sometimes say things like "start is a num-list" when writing in English, because saying "start is a (num list)" reads badly --- the data type looks like a parenthetical remark, despite typography
Design recipe

• We now expect you to do this right
• No longer part of the points allocated to a problem
• Lose points if you mess it up.
• Never get score less than 0, though.
A thought experiment
Swap cases...
Better
CS17 style rule

- Cases *should be mutually exclusive* when feasible (almost always!), or (if not) carefully documented as depending on order.
  - Insert a comment like ";;;; this case must come last!"

- Sometimes in class, I'll ignore this rule to make code fit on one slide; I'll try to always point it out.
Back to Analysis
Brief review of standard process henceforth

- Write a (recursive) program
- Write down a recurrence relation that it satisfies
- Solution version 1
  - Use plug-and-chug to guess a solution
  - Prove your guess correct
- Solution version 2
  - Recognize the recurrence as one you’ve seen before
  - Quote the prior analysis result
- Say something about the big-O class of the result (next week?)
One more recurrence to derive/solve.
Solution
Improved right-max
Even better right-max
A pause from analysis for a moment

• Lambda
A new way to define procedures

• (define b 4)
  • Evaluates 4 to get the number value 4
  • Places "b" in the top-level environment, binding it to 4.

• (define (f x) (+ x 1))
  • Creates a closure-value with arglist: x, and body: (+ x 1)
  • Places "f" in the TLE, binding it to that closure
  • Almost like previous example, except that no "evaluation" took place, because we don't have anything we can evaluate to produce a closure value
A new procedure!

(define f (lambda (x) (+ x 1)))

• The "expression" here is a new kind of expression – a "lambda expression"
• The result of evaluating it is a closure value
• So we could write
  ((lambda (x) (+ x 1)) 3)
and the result would be "4".
• Why would we ever do this?
  • Now we only need one form of "define" (but we'll continue to use both)
  • Sometimes we'll actually use the result of a lambda-expression without
    naming it!
A (slightly) contrived example

(define (add1 x) (+ x 1))
(define (add2 x) (+ x 2))
(define (add7 x) (+ x 7))
...

;; build an "add b" function!
(define (incrementer b)
  (lambda (x) (+ x b)))

((incrementer 3) 4) => 7