Class 14: Miscellany, More Analysis

- Writing types
- Design recipe update
- Cond-case ordering
- Analysis of multiple versions of a single procedure
- Two-column proofs, a first look
- Lambda
- Map
"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

\[
f(x) = \begin{cases} 
4, & x < 2 \\
\frac{1}{x}, & 2 \leq x < 5 \\
11, & x \geq 5 
\end{cases}
\]

(define (f x)
  (cond
    [(< x 2) 4]
    [(< x 5) (/ 1 x)]
    [(>= x 5) 11])))
Swap cases...

\[
 f(x) = \begin{cases} 
 4, & x < 2 \\
 \frac{1}{x}, & 2 \leq x < 5 \\
 11, & x \geq 5 
\end{cases}
\]

(define (f x)
 (cond
 [(< x 5) (/ 1 x)]
 [(< x 2) 4]
 [(*= x 5) 11]]))

• BROKEN!
Better

\[
f(x) = \begin{cases} 
4, & x < 2 \\
\frac{1}{x}, & 2 \leq x < 5 \\
11, & x \geq 5
\end{cases}
\]

(define (f x)
  (cond
    [(< x 2) 4]
    [(and (>= x 2) (< x 5)) (/ 1 x)]
    [(>= x 5) 11])))
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

- Right-max code looked something like this

```scheme
(define (right-max aloi)
  (cond
    [(empty? aloi) ...]
    [(empty? (rest aloi)) ...]
    [(cons? aloi) (cons (... (right-max (rest aloi) ...) (right-max (rest aloi)...)]]))
```

\[
\begin{align*}
R(0) &= A \\
R(1) &= B \\
R(n) &= C + 2R(n - 1) \text{ for } n > 1
\end{align*}
\]
Solution

- $R(n) \leq C \ 2^n$
- We say that this procedure runs in "at worst exponential time."
- *Very* bad in general
  - Some problems – many of the ones we’re most interested in – have no solutions known to run in less than exponential time, alas.
Improved right-max

- Right-max code looked something like this

```scheme
(define (list-max aloi)
  (cond
    [(empty? (rest aloi)) (first aloi)]
    [(cons? aloi) (max2 (first aloi) (list-max (rest aloi)))]))

;; Note: list-max runtime looks like Pn + Q, where list has length n
```

```scheme
(define (right-max aloi)
  (cond
    [(empty? aloi) empty]
    [(empty? (rest aloi)) (first aloi)]
    [(cons? aloi) (cons (list-max aloi) (right-max (rest aloi)))]))
```

\[
R(0) = A \\
R(1) = B \\
R(n) = C + Pn + Q + 1R(n-1) \text{ for } n > 1
\]

Ends up quadratic!
Even better right-max

; Cons either t or (first aloi), whichever is larger, onto aloi
(define (lengthen t aloi)
  (cons (max2 t (first aloi)) aloi))

;; Note: lengthen runtime looks like L(n) = P, a constant!

(define (right-max aloi)
  (cond
   [(empty? aloi) empty]
   [(empty? (rest aloi)) (first aloi)]
   [(cons? aloi) (lengthen (first aloi) (right-max (rest aloi)))]))

\[
\begin{align*}
R(0) &= A \\
R(1) &= B \\
R(n) &= C + P + R(n - 1) \text{ for } n > 1
\end{align*}
\]

Ends up linear!
Proofs
Proofs

• Typical thing we'll ask you to prove is some claim like this:
  • "For any integer $n \geq 4$, $3n^2 - 2n + 1 \geq 41$."
• Typically the proof starts "Suppose $n$ is an integer and $n \geq 4$."
• You can rephrase the claim as "if $n$ is an integer greater than or equal to 4, then $3n^2 - 2n + 1 \geq 41$."
• When you want to prove an if...then statement, you start by assuming the "if" and see if it can lead you to the "then".
Two-column proofs

• These are a great way to be sure you're not fooling yourself.

• In the left column you write "statements", and in the right, you put "reasons".

• Acceptable reasons: prior statements, the hypothesis (the "if" part of the if...then that you're proving), arithmetic or algebra rules

• Number all statements; refer to prior statements by number
For any integer $n \geq 4$, $3n^2 - 2n + 1 \geq 41$.  

<table>
<thead>
<tr>
<th>Statement</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Suppose $n$ is an integer and $n \geq 4$.</td>
<td>Hypothesis</td>
</tr>
<tr>
<td>2 $n - 4 \geq 0$</td>
<td>S1, subtract 4 from both sides</td>
</tr>
<tr>
<td>3 $n \geq 4 \geq 0$</td>
<td>S1, arithmetic</td>
</tr>
<tr>
<td>4 $3n \geq 0$</td>
<td>S3, multiply both sides by 3</td>
</tr>
<tr>
<td>5 $10 \geq 0$</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>6 $3n + 10 \geq 0$</td>
<td>S4, S5, addition of inequalities</td>
</tr>
<tr>
<td>7 $(n - 4)(3n + 10) \geq 0$</td>
<td>S3, S6, product of nonnegative numbers is nonnegative</td>
</tr>
<tr>
<td>8 $3n^2 - 2n - 40 \geq 0$</td>
<td>S7, algebra</td>
</tr>
<tr>
<td>9 $3n^2 - 2n + 1 \geq 41$</td>
<td>S8, add 41 to both sides</td>
</tr>
</tbody>
</table>
A wrong proof

• "For any integer \( n \geq 4, 3n^2 - 2n + 1 \geq 41.\)"
• Well, 5 is bigger than 4, and \( 3 \cdot 5^2 - 2 \cdot 5 + 1 = 66 \geq 41, \) so it works.
• This "proof" only shows that the claim is true for some number that's at least 4, not for every number
• You could redo this with 6, or 12, or 81, but you'd still only have covered a few of the infinitely many truths being claimed.
• The "good" proof says "instead of a particular value, let's use a name \( n \) to denote some un-chosen value, whose only property is that it's at least 4. Then for any particular number bigger than 4, we can substitute that number for \( n \) and see a proof of the claim for that number."
How did you know what to write next?

• Experience
• Copying similar proofs I've seen before
• Working backwards
• Guesswork
Do / have to do that?

• Generally not
• The few proofs we'll ask you to do will be algebraically simpler and more obvious
• The *structure*, in which each statement follows from previous ones or facts from arithmetic/algebra, is the main idea here.
• For many analyses, we’ll do a general proof in class, and you can just cite a theorem (or “page 3 of Nov 12’s class notes”, etc.)
For proofs about recurrence relations, there's a *template*, like the one for recursive programs

- 90% of the template is unchanged in each application
- The 10% that changes is just a bit of algebra typically.
A pause from analysis for a moment

- Lambda
- "map"
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)
Higher order operations
Increment each item in a list of numbers

(define (inc-all alon)
  (cond
    [(empty? alon) empty]
    [(cons? alon) (cons
      (+ 1 (first alon))
      (inc-all (rest alon)))]))
Test each item in a list of integers to see if it's odd (produce a list of booleans)

(define (odd-all aloi)
  (cond
    [(empty? aloi)  empty]
    [(cons? aloi)  (cons
                    (odd? (first aloi))
                    (odd-all (rest aloi)))]))
Censor every item in a list of strings by replacing it with "*"

(define (censor  alos)
  (cond
    [(empty? alos)  empty]
    [(cons? alos)  (cons
                  "*
                  (censor (rest aloi))))]))
Improve a list of numbers by changing everything to 17

(define (improve alon)
  (cond
   [(empty? alon)  empty]
   [(cons? alon)  (cons
      17
      (improve (rest alon)))])
Differences

(define (improve alon)
  (cond
    [(empty? alon)  empty]
    [(cons? alon)  (cons 17
                   (improve (rest alon)))]))
"Apply a proc to each element of the list"

(define (apply-all proc alod)
  (cond
   [(empty? alod)  empty]
   [(cons? alod)  (cons
                   (proc (first alod))
                   (apply-all proc (rest alod)))])

(define (odd-all aloi) (apply-all odd? aloi))
(define (inc-all aloi) (apply-all succ aloi))
What about "censor"?

(define (apply-all proc alod)
  (cond
    [(empty? alod) empty]
    [(cons? alod) (cons
      (proc (first alod))
      (apply-all proc (rest alod)))]

  (define (star str) "*")
  (define (censor alos) (apply-all star alos))
"apply-all" is called "map"

- It's a Racket built-in!

```
(map proc lst)
```

What's the type-signature for what we wrote?

```
; map: ('a -> 'b) * ('a list) -> ('b list)
```

Note: the built-in is much fancier, and incorporates the "map2" procedure you'll be writing for homework.
"map" is...

- a **higher order procedure** (HOP)
- it consumes *functions* rather than atomic or compound data
- This week's homework goes wild on this idea
• "map" is powerful, but even more powerful is "fold"... which you'll write in this week's homework.
• It was a little annoying to have to write the "star" procedure just so that we could use it inside "map"
• We could have used a lambda-expression!
How to write "improve" using "map" and "lambda"

(define (improve aloi)
  (map (lambda (x) 17) aloi))
Analysis of map

• Suppose that largest number of operations performed in evaluating (proc x) for any argument x is a constant P.
• Then (map proc alod) takes no more than $A + n \cdot (B + P)$ operations for some nonnegative constants $A, B$.
• Let $H(n)$ be the greatest number of ops in applying (map proc alod) to any list alod of $n$ items.
• Then (by looking at the code)
  \[
  H(n) = B + P + H(n - 1) \text{ for } n > 0
  \]
• Plug-n-chug to get result.
• NB: if the runtime for the procedure is not constant, things get messier
  • We'll almost never encounter this