Class 10: Code review, a little more recursion, and analysis, part 1
Announcements

• Today’s topics
  • Some code review
  • Operation counting
  • Writing recurrences for operation-counting functions
  • Two-argument recursion, part 1

• Remember to hand the TAs a note about things you’d like to re-visit
Rosh Hashannah, Yom Kippur

• The lecture-capture videos for those two classes will be available on the course website.
• As mentioned in the syllabus, HW will still be due at the usual time.
Homework hint

• For problem 5, last part, you must show that if $T$ and $S$ are two functions and $T(n) = 3 \cdot S(n)$ then (using $M = 1$), $T$ is eventually larger than $S$, and folks have had trouble with this
  • It’s a lot simpler when you notice that both functions have $\mathbb{R}_+$ as their codomain, i.e., for every $n$, the number $S(n)$ is positive.
  • Without this, the claim is actually false!
List construction simplified

• You may, after tonight’s HW, use `(list 1 2 3)` to produce the list we’d formerly have produced using `(cons 1 (cons 2 (cons 3 empty)))`

• Use language-level “Beginning Student with List Abbreviations” to see prettier printed representations of lists.
Design recipe 1

- You no longer need to provide data examples for atomic data (num, bool, string)
- You do need to provide examples of list data; use the standard form:
  
  ;; a num list is either
  ;;    the empty list
  ;;    (cons n b), where n is a num, and b is a num list
  ;;    nothing else is a num list
Design Recipe 2

• The design recipe requires you to write the template for your procedure before you write the input-output specification, so that you have a name to use for your input
  • The input spec should explicitly use this name:

```scheme
;;; input:
;;; aloi, a non-empty list of student heights
;;; ...
(define (my-proc aloi) …)
```
“Generic lists”

• Some procedures will work on a num list, a bool list, a string list, an anything-list!
• We indicate this by writing 'a list; the 'a is read “alpha”, and stands for “some unspecified type”
• If you need more, you can use 'b 'c 'd
• The type signature for “first” is then
  first: 'a list -> 'a
because if it consumes a bool-list, it produces a bool, etc.
Student code

(define (odds-only) alon
  (cond
   [(empty? empty)]
   [(cons?
      [(if odd?
         (first alon
          first (rest alon) odds-only))]
   )))
Student code

(define (odds-only) alon
  (cond
   [(empty? empty)]
   [(cons?
      [(cons?
         (if odd?
          (if odd?
            first alon
            first (rest alon) odds-only)))]]))

• What else is wrong here?
Student code

(define (odds-only) aloi
  (cond
   [(empty? empty)]
   [(cons?
       (if odd?
           first aloi
           first (rest aloi) odds-only)])
   []))

• What else is wrong here?
  • Odds-only should consume a list of integers, so aloi is a better name-choice
  • Doesn’t use the template!
(define (fname arg)
  (cond
   [(empty? arg) ...]
   [(cons? arg) ... (first arg) ... (fname ... (rest arg) ...)]

• Guarantees that the procedure arguments get placed right (and named)
• Guarantees that the tests are applied to the arguments
• Makes sure that the recursive result gets set up almost exactly right for almost all cases
• Matches structure of recursion diagram to make coding easier!
Student code, template version

(define (odds-only aloi)
  (cond
   [(empty? aloi) empty]
   [(cons? aloi) (if odd? (first aloi) (odds-only (rest aloi)))]))

• Still broken
  • The “odd?” needs to be part of a proc-appl-expr
  • There needs to be a “cons” somewhere
  • Lots easier to debug!
Topic switch

• *Analysis*
• Know how to write basic recursive programs
• It's time to understand the speed with which such programs work.
Demo: how slow can you go?
Last time

• A procedure that operates on lists of varying length may take different amounts of time on different lists.
• For a list of length $n$, we measure the "time" in units of elementary operations
• For list-length procedure, we saw that the number of elementary ops increased as the size of the input list increased
Next step: define an operation-counting function

• First draft:
  • "Let $L(n)$ be the number of elementary operations involved in applying our procedure to a list of length $n$."
• Domain: natural numbers (because they’re list-lengths!)
• Codomain: number of operations used.
  • Surely at least 1
  • So: positive integers
  • To make some algebra easier, we’ll say “positive reals” instead
  • So $T: \mathbb{N} \to \mathbb{R}_+$
  • Kind of like the homework problem you’re working on!
A small hitch

"Let $T(n)$ be the number of elementary operations involved in applying our procedure to a list of length $n$."

• What if the procedure takes different amounts of time on two different 100-element lists?
  • Example: `contains17?` is very fast on `(cons 17 (cons ... )))))))), but slow on `(cons 18 (cons 19 ( .....(cons 118 empty)...)))))))`)

• Revised:

"Let $T(n)$ be the largest number of elementary operations involved in applying our procedure to any list of length $n$."

Studying $T$ is called worst-case performance analysis

It's what we do because it's relatively easy!
Something surprising

"Let $T(n)$ be the largest number of elementary operations involved in applying our procedure to any list of length $n$."

• Surprise: it's easier to talk about this version of T than the other one
• Disappointment: it's harder to actually compute $T(n)$ for any particular $n \neq 0$, because it requires an infinite number of tests
What we *can* do

• We can often look at our programs and say "The function $T$ must satisfy certain equations/inequalities" (today)

• Then (with practice) say something about all *solutions* of those equations, hence about the function $T$ (next week)

• First: elementary operations
Elementary operations

- During evaluation of an expression, we count elementary operations
  - Evaluate a number-expression, string-expression, or bool-expression
  - Evaluate a name-expression (i.e., look up a binding in an environment)
  - Evaluate empty
  - Apply cons to two values
  - Apply +, *, -, / to two values
  - Apply cons?, empty?, zero?, >, <, =, etc.
  - Apply first or rest to a list
  - Add a binding to an environment
  - Remove a binding from an environment
  - Compare a boolean to true/false
Count elementary operations

i. (cons 13 (cons 4 empty))  
   7

ii. (if (> 3 4) 1 2)  
   (define (len aloi)
      (cond
         [(empty? aloi) 0]
         [(cons? aloi) (+ 1 (len (rest aloi)))]))

iii. (len empty)  
   lookup len; eval empty; bind aloi to empty: 3
   evaluate (empty? aloi): lookup, lookup, apply: 3
   check if it's T/F: 1
   evaluate 0: 1
   unbind aloi: 1
   Total: 9
Count elementary operations

i. (len empty) took 9 operations

ii. (define lst1 (cons 1 empty))
   (len lst1)

iii. (define lst2 (cons 1 (cons 2 empty)))
    (len lst2)

(define (len aloi)
    (cond
        [(empty? aloi) 0]
        [(cons? aloi) (+ 1 (len (rest aloi)))]))
Count elementary operations (len lst1)

(define lst1 (cons 1 empty))
(len lst1)

Lookup len, lst1: 2

bind aloi to lst1: 1

evaluate (empty? aloi), test vs true: 4 [same as before]
evaluate (cons? aloi), test vs true: 4

evaluate (+ 1 (len (rest aloi))):
    evaluate +, 1: 2
    evaluate (len (rest aloi)):
        lookup len: 1
            evaluate (rest aloi): 2 lookups, 1 application
            apply len to empty: 9 [from before]
        apply +: 1
    unbind aloi: 1

Total: 19 + 9 (or 19 + L(0))

(define (len aloi)
  (cond
    [(empty? aloi) 0]
    [(cons? aloi) (+ 1 (len (rest aloi)))]))
Count elementary operations (len lst2)

Total: 19 + 19 + 9 (or 19 + L(1))

(define (len aloi)
  (cond
   [(empty? aloi) 0]
   [(cons? aloi) (+ 1 (len (rest aloi)))]))
A pattern

• "Let $L(n)$ be the largest number of elementary operations involved in applying len to any list of length $n."$

• $L(0) = 9$
• $L(1) = 19 + L(0)$
• $L(2) = 19 + L(1)$
• $L(n) = 19 + L(n - 1)$ for $n > 0$

$L(0) = 9$
$L(n) = 19 + L(n - 1)$ for $n > 0$

Call this a "recurrence relation for $L."$
Recurrence relation

- Multiple equations/inequalities
- First one or two give known facts, like $L(0) = 8$; called "base cases"
- Base case may be 0 or 1 or something else; usually 0 or 1.
  - Sometimes have two or more base cases
- Last equations/inequality says something (often inequality) about $L(n)$ in terms of prior values:
- Examples:
  - $S(n) \leq 2S(n - 1)$ for $n > 0$
  - $H(n) \leq H(n - 1) + H(n - 2)$ for $n > 1$
  - $R(n) \leq n + R \left( \left\lfloor \frac{n}{2} \right\rfloor \right) + 3$ for $n > 1$
Procedure → Recurrence

- *Don't count exactly: use constants*
- Base case usually looks like
  
  \[
  H(0) = A
  \]

- For the other case (usually the "cons?" clause of the main cond expression)
  - Find op count (on an input of size \( n \)) for all work done *except in recursive calls*
    - * Might be a constant, might depend on \( n \)*
  - Figure out the *argument size* in all recursive calls
    - For us, for now: typically \( n - 1 \)
  - Express total work in all recursive calls in terms of \( H \)
    - For us, typically: \( H(n - 1) \)
  - Sum these three, and write \( H(n) \leq \) this sum.
Activity

(define (contains17? aloi)
  (cond
    [(empty? aloi) false]
    [(cons? aloi) (or (= 17 (first aloi))
                     (contains17? (rest aloi)))]))

For the other case (usually the "cons?" clause of the main cond expression)

  Find op count (on an input of size $n$) for all work done except in recursive calls
    * Might be a constant, might depend on $n$
  Figure out the argument size in all recursive calls
    * For us, for now: typically $n - 1$
  Express total work in all recursive calls in terms of $H$
    * For us, typically: $H(n - 1)$
  Sum these three, and write $H(n) \leq$ this sum.
Reminder

• Today’s topics
  • Some code review
  • Operation counting
  • Writing recurrences for operation-counting functions
  • Two-argument recursion, part 1

• Remember to hand the TAs a note about things you’d like to re-visit