Homework 13: Plan Composition

Due: 10:00 PM, Dec 5, 2014

Back when this semester began
You'd not have been able to plan
To write this procedure
But now you're so eager:
You couldn't before, now you can!

Contents

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. **For this homework specifically, this entails answering the Palindrome Detection, Sum Over Table, Adding Machine, Rainfall, Length of Triples, and Shopping Discount questions.**

In order to hand in your solutions to these problems, they must be stored in appropriately-named files. In particular, each should be named for the corresponding problem, as follows (e.g., `palindrome.rkt` corresponds to Palindrome Detection):

- `palindrome.rkt`
- `sum.rkt`
- `adding.rkt`
- `rainfall.txt`
- `triples.txt`
- `shopping.txt`

For this assignment, all files you turn in that contain code must be Racket files, so they must end with extension `.rkt`.

Overview

This assignment contains two sets of three problems each. The first set, Programming Problems, requires you to write your own programs to solve given problems. The second set, Review Problems, gives you problems and proposed solutions, and asks you to identify which solutions you prefer.

Important: You should treat this assignment as a diagnostic to assess for yourself what you have learned in this course. As such, you should treat it like an exam, and complete all the problems entirely on your own. Do not collaborate with your classmates, or seek help from the TAs. The TAs will be available to answer clarification questions only.

1 Programming Problems

1.1 Palindrome Detection

A palindrome is a string with the same letters in each of forward and reverse order (ignoring capitalization). Write a procedure called *is-palindrome* that consumes a string and produces a boolean indicating whether the string with all spaces and punctuation removed is a palindrome. Treat all non-alphanumeric characters (i.e., ones that are not digits or letters) as punctuation.

Examples:

(*is-palindrome* "a man, a plan, a canal: Panama")
⇒ *true*

(*is-palindrome* "abca")
⇒ *false*

(*is-palindrome* "yes, he did it")
⇒ *false*

1.2 Sum over Table

Assume that we represent tables of numbers as lists of rows, where each row is itself a list of numbers. The rows may have different lengths.

Write a procedure called *sum-largest* that consumes a table of numbers and produces the sum of the largest number in each row. Assume that no row is empty.

Example:

(*sum-largest* (*list* (*list* 1 7 5 3) (*list* 20) (*list* 6 9)))
⇒ 36

In this example, *sum-largest* extracted 7 from the first row, 20 from the second, and 9 from the third, and then added these numbers up.

1.3 Adding Machine

Write a procedure called *adding-machine* that consumes a list of numbers and produces a list of the sums of each non-empty sublist separated by zeros. Ignore input elements that occur after the first occurrence of two consecutive zeros.

Example:

(*adding-machine* (*list* 1 2 0 7 0 5 4 1 0 0 6))
⇒ (*list* 3 7 10)

2 Review Problems

Below you are given problem statements followed by multiple solutions to each problem. Assume that the solutions are correct; ignore any small deviations in behavior. Also assume that any missing helper functions are defined in the obvious way. Finally, ignore stylistic differences in naming. Instead, focus on the structure of the solutions.

For each problem, rank the solutions in order of your preference from most to least. Explain why you picked that ordering. You are allowed to have ties.

The solutions are labeled `A`, `B`, etc. Indicate your ordering with the labels and `>`, using commas for ties. For instance, the ordering `B > A, C > D` means you liked `B` the most, followed by `A` and `C` (tied), followed by `D`.

Remember to explain your choice!

Lastly, you should not run the code in this section. This is a reading exercise.

2.1 Rainfall

Write a procedure called *rainfall* that consumes a list of real numbers representing daily rainfall readings. The list may contain the number −999 indicating the end of the data of interest. Produce the average of the non-negative values in the list up to the first −999 (if there is one). There may be negative numbers other than −999 in the list (representing faulty readings). Assume that there is at least one non-negative number before −999.

Example:

(*rainfall* (*list* 1 −2 5 −999 8))
⇒ 3

Solution A:

```
;; rainfall-loop : (listof number) number number → number
(define (rainfall-help sub-l total days)
  (cond [(empty? sub-l) (/ total days)]
        [(cons? sub-l) (cond [(= (first sub-l) -999) (/ total days)]
                             [(< (first sub-l) 0) (rainfall-help (rest sub-l) total days)]
                             [else (rainfall-help (rest sub-l) (+ total (first sub-l)) (+ 1 days))])]))

;; rainfall : (listof number) → number
(define (rainfall l)
  (rainfall-help l 0 0))
```

Solution B:

```
;; sum-relevant : (listof number) → number
(define (sum-relevant l)
  (cond [(empty? l) 0]
        [(cons? l) (cond [(= (first l) -999) 0]
                         [(< (first l) 0) (sum-relevant (rest l))]
                         [else (+ (first l) (sum-relevant (rest l)))])]))

;; count-relevant : (listof number) → number
(define (count-relevant l)
  (cond [(empty? l) 0]
        [(cons? l) (cond [(= (first l) -999) 0]
                         [(< (first l) 0) (count-relevant (rest l))]
                         [else (+ 1 (count-relevant (rest l)))])]))

;; rainfall : (listof number) → number
(define (rainfall l)
  (/ (sum-relevant l) (count-relevant l)))
```

Solution C:

```
;; cleanse : (listof number) → (listof number)
(define (cleanse l)
  (cond [(empty? l) empty]
        [(cons? l) (cond [(= (first l) -999) empty]
                         [(< (first l) 0) (cleanse (rest l))]
                         [else (cons (first l) (cleanse (rest l)))])]))

;; rainfall : (listof number) → numbers
(define (rainfall l)
(let ((clean-data (cleanse l)))
    (/ (sum-of clean-data) (length clean-data))))
```

2.2 Length of Triples

Write a procedure called *max-triple-length* that consumes a list of strings and produces the length of the longest concatenation of three consecutive elements. Assume the input contains at least three

strings. Also assume we are given the data definition:

```
(define-struct triple (a b c))
```

and the helper procedure:

```
;; break-into-triples : (listof number) → (listof triple)
(define (break-into-triples l)
  (cond [(empty? (rest (rest l))) empty]
        [else (cons (make-triple (first l) (second l) (third l))
                    (break-into-triples (rest l)))]))
```

Example:

```
(max-triple-length (list "a" "bb" "c" "dd"))
⇒ 5
```

Solution A

```
;; max-triple-length : (listof string) → number
(define (max-triple-length l)
  (max-of (map (lambda (t)
                 (+ (string-length (triple-a t))
                    (string-length (triple-b t))
                    (string-length (triple-c t))))
               (break-into-triples l))))
```

Solution B

```
;; max-triple-length : (listof string) → number
(define (max-triple-length l)
  (let ((data-as-nums (map string-length l)))
    (max-of
     (map (lambda (t) (+ (triple-a t)
                         (triple-b t)
                         (triple-c t)))
          (break-into-triples data-as-nums)))))
```

Solution C

```
;; triple-help : (listof string) → number
(define (triple-help sub-l max-so-far)
  (cond [(empty? (rest (rest sub-l))) max-so-far]
        [(cons? sub-l) (let ((next-length (+ (string-length (first sub-l))
                                             (string-length (second sub-l))
                                             (string-length (third sub-l)))))
                         (triple-help (rest sub-l) (max max-so-far next-length)))]))

;; max-triple-length : (listof string) → number
(define (max-triple-length l)
```

(*triple-help l* 0))

2.3 Shopping Discount

An online clothing store applies discounts during checkout. A shopping cart is a list of the items being purchased. Each item has a name (a string like "shoes") and a price (a real number like 12.50). Write a procedure called *checkout* that consumes a shopping cart and produces the total cost of the cart after applying the following two discounts:

- if the cart contains at least 100 worth of shoes, take 20% off the cost of all shoes (match only items whose exact name is "shoes")
- if the cart contains at least two hats, take 10 off the total of the cart (match only items whose exact name is "hat")

Assume the cart is represented as follows:

```
;; An item is (make-item string number)
(define-struct item (name price))
```

and we are given these two helper procedures:

```
;; hat-discount : number → number
(define (hat-discount numhats)
  (if (≥ numhats 2) 10 0))

;; shoe-discount : number → number
(define (shoe-discount shoe-total)
  (if (≥ shoe-total 100)
      (* shoe-total .20)
      0))
```

Example:

```
(checkout (list (make-item "shoes" 25) (make-item "bag" 50)
                (make-item "shoes" 85) (make-item "hat" 15)))
⇒ 153
```

Solution A

```
;; shop-help : (listof item) number number number → number
(define (checkout-help cart shoe-total hat-count cart-total)
  (cond [(empty? cart)
         (- cart-total (shoe-discount shoe-total) (hat-discount hat-count))]
        [(cons? cart)
                (let ((name (item-name (first cart)))
                      (price (item-price (first cart))))
           (cond [(string=? name "shoes")
                  (checkout-help (rest cart) (+ shoe-total price) hat-count (+ cart-total price))]
```

```
                [(string=? name "hat")
                 (checkout-help (rest cart) shoe-total
                                (add1 hat-count)
                                (+ cart-total price))]
                [else (checkout-help (rest cart) shoe-total hat-count (+ cart-total price))])]))

;; checkout : (listof item) → number
(define (checkout cart)
  (checkout-help cart 0 0 0))
```

Solution B

```
;; total-cost : (listof item) → number
(define (total-cost cart)
  (cond [(empty? cart) 0]
        [(cons? cart) (+ (item-price (first cart)) (total-cost (rest cart)))]))

;; checkout : (listof item) → number
(define (checkout cart)
  (let ((shoes (filter (lambda (item) (string=? (item-name item) "shoes")) cart))
  (hats (filter (lambda (item) (string=? (item-name item) "hat")) cart)))
    (- (total-cost cart)
       (shoe-discount (total-cost shoes))
       (hat-discount (length hats)))))
```
