CMPU-101-52

Problem Solving and Abstraction

Spring 2017
Final Exam

Name: ______________________________ Signature: ______________________________

With my signature, I promise that I haven't discussed and will not discuss the contents of this exam with anyone until after the officially scheduled exam period is over: Tue, 23 May 2017, 7pm.

Instructions:

1. This is an open book, open notes exam. You may use your CS account and browser to access the book and course notes. You may use DrRacket, but are on your honor not to use the Interactions Pane, or Run any programs.
2. Budget your time. There are 100 points on this exam, so you should spend about 1 minute per point (e.g., 10 minutes on a 10 point question). Don't spend too much time on any one question.
3. Each question indicates the total number of points.
4. Please don't hesitate to ask any questions.
5. Good luck!!!

	Points	
Question	received	possible
1. Writing functions for trees		30
2. Abstracting and parameterizing functions		30
3. Using filter with a predicate, local and lambda		20
4. Evaluating higher order functions on lists		20
Total		100

Problem 1. (30 points) A semanticist—that is, an eccentric Computer Scientist like your professor—has developed a strange perception of the entire world as tree-shaped mazes, and the paths we follow through these mazes, using the following data definitions:

Imagine yourself following a path, which is really just a sequence of directions for exploring a tree. If the path says `"left"` and the tree is a `node`, follow the tree in the left field; if the path says `"middle"` and the tree is a `node`, follow the tree in the middle field; if the path says `"right"` and the tree is a `node`, follow the tree in the right field. Finally, if a path is empty, you have arrived.

```
; Dir is one of:     ; Path is one of:
; - "left"           ; - '()
; - "middle"         ; - (cons Dir Path)
; - "right"

(define-struct node (left middle right))
; Tree is one of:
; - Number
; - (make-node Tree Tree Tree)
```

(7 points) Given the following tree (rendered in my best ASCII art), define nodes `t1` through `t7`. Advice: it will be easier to define the tree bottom-up, using seven separate defines (starting with `t1`), than to define the tree in one big define.

```
                                            t7
                                          / | \
                                         /  |  \
                                        /   |   \
                      _________________/    |    \
                     /                      |     \
                    /                       |      \
                   t5                      t6       \
                 / | \                    / | \      \
       _________/  |  \__________        /  |  \      \
      /            |             \      1   1   1      |
     /             |              \                    |
    t1             t2             t3                   t4
  / | \          / | \          / | \                / | \
 /  |  \        /  |  \        /  |  \              /  |  \
4   8   5      2   7   3      6   9   7            8   9   0
```

(2 points) What Numbers do paths `p1` and `p2` below lead to in the Tree rooted at `t7`?

```
(define p1 (list "left" "middle" "right"))

(define p2 (list "left" "right" "middle"))
```

(1 point) Define path `p3` that leads from node `t7` to the only number 5 in the Tree.

(20 points) Design a function called `search` that follows a given Path into a given Tree, and returns the Tree at the end of the given Path. If the Path is too long (meaning you've reached a Number in the Tree, but the Path is not empty), return the string `"error"`. To help you get started, I've provided the signature and purpose statement.

```
; search : Path Tree -> Tree (or "error")
; follow path p through tree t and return the tree at end of p, or "error"
```

Problem 2. (30 points) Consider the following data definition for a CD and two functions that work over a list of CDs:

```
(define-struct CD (genre price))       ;; A list-of-CDs (LOCD) is either
;; A CD is a (make-CD string number)   ;; - empty
                                       ;; - (cons CD LOCD)

;; count-classical : LOCD -> number
;; counts the number of "classical" CDs in given locd
(define (count-classical locd)
  (cond [(empty? locd) 0]
        [(cons? locd)
         (cond [(string=? (CD-genre (first locd)) "classical")
                (+ 1 (count-classical (rest locd)))]
               [else (count-classical (rest locd))])]))

;; count-indie : LOCD -> number
;; counts the number of "indie" CDs in given locd
(define (count-indie locd)
  (cond [(empty? locd) 0]
        [(cons? locd)
         (cond [(string=? (CD-genre (first locd)) "indie")
                (+ 1 (count-classical (rest locd)))]
               [else (count-classical (rest locd))])]))
```

(10 points) Since both `count-classical` and `count-indie` count CDs of a certain type, combine them into a single higher-order function called `count-CD-type` by parameterizing their differences. Be sure to include a general signature for the `count-CD-type` function.

(5 points) Rewrite `count-classical` and `count-indie` so that they both use `count-CD-type`. (Hint: each function's body should be a one-liner).

(10 points) Now we want to count the CDs in our collection that are our favorites. Design a function called `count-faves` that is even more abstract than `count-CD-type`, which consumes an LOCD and a predicate function for CDs, and counts only those CDs that satisfy the given predicate. Be sure to include a general signature and purpose statement for `count-faves`.

(5 points) Now use `count-faves` to implement a function named `count-classy` that counts Professor Smith's favorite CDs in his collection. A classy CD is one that is "classical" and costs over 15 dollars. Your `count-classy` function should use a `local` expression to define the predicate function that it passes to `count-faves`.

Problem 3. (20 points) Recall the abstract `map` function, shown with its signature here:

```
map: (X -> Y) list-of-X -> list-of-Y
```

(5 points) First, develop the function, `double`, which doubles the value of the number it consumes. Be sure to include a signature and purpose statement. No tests are necessary.

(5 points) Next, use `map` and `double` (i.e., no `cond` expression or recursion) to develop the function `double-nums`, which consumes a list of numbers and doubles the value of each number. Be sure to include a signature and purpose statement.

(5 points) Now, consolidate the functions you wrote on the previous page so that the `double` function is defined within `double-nums` using a `local` expression.

(5 points) Finally, simplify `double-nums` by removing the `local` expression altogether, and replacing `double` with its equivalent `lambda` expression.

Problem 4. (20 points) Recall the abstract, higher order functions given below:

```
filter: (X -> bool) list-of-X -> list-of-X
map: (X -> Y) list-of-X -> list-of-Y
foldr: (X Y -> Y) Y list-of-X -> Y
ormap: (X -> bool) list-of-X -> bool
andmap: (X -> bool) list-of-X -> bool
```

The following examples of lists are used below:

```
(define nums (list 1 2 3 4 5 6 7 8 9 10))
(define evens (list 2 4 6 8 10))
(define odds (list 1 3 5 7 9))
```

(10 points / 2 points each) Using only the higher order functions given above, built-in functions (like `odd?`, `even?`, . . .), or lambda expressions you write yourself, write expressions for the following:

- Expression that removes the even numbers from `nums`
- Expression that determines whether `evens` contains any odd numbers
- Expression that determines whether `odds` contains only odd numbers
- Expression that adds 42 to every value in `nums`
- Expression that sums all the even numbers in `nums` (be careful: `nums` contains numbers that aren't even!)

(10 points / 2 points each) Evaluate the following expressions:

- `(foldr string-append "" (map number->string nums))`

- `(map number->string nums)`

- `(andmap string?  (list "1" "2" "3" "4" "5" 6 7 8 9 10))`

- `(filter (lambda (n) (and (>= n 5) (odd?  n))) nums)`

- `(ormap zero?  (map sub1 nums))`