CMPU-101-01

Problem Solving and Abstraction

Spring 2019
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: Thu, 16 May 2019, 3pm.

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., 25 minutes on a 25 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!!!

Question	Points received	Points possible
1. Writing functions for trees		25
2. Abstracting over and parameterizing functions		25
3. Using filter with a predicate, local and lambda		25
4. Evaluating higher order functions over lists		25
Total		100

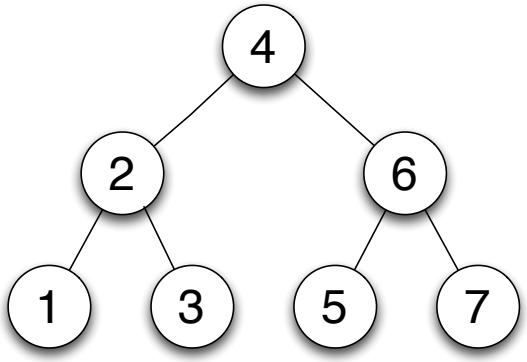
Problem 1. (25 points)

Here are the structure and data definitions for a Binary Search Tree (BST):

```
(define-struct node (value left right))
; A binary-search-tree (BST) is either
; - '()
; - (make-node Number BST BST)
; interp. all values in the left BST are less than value,
;   and all values in the right BST are greater than value.
```

Figure 1 below depicts a binary search tree. The circles represent node structs, and the lines below each circle lead to the left and right BSTs for each node. (Nodes 1, 3, 5, and 7 have `'()` left and right fields.)

Figure 1: A binary search tree



a) (7 points) Using the data and structure definitions given for a BST, translate Figure 1 into its Racket representation. *Hint: it's easiest to build the BST one level at a time, starting from the bottom. That is, define nodes n1, n3, n5, and n7 first, then use those defined node names for defining nodes n2 and n6, etc.*

b) (18 points) Develop the function `descending-order`. It consumes a binary search tree and produces a list of all the numbers in the tree. The list contains the numbers from right to left as you'd read them from the tree (e.g., for the full tree in Figure 1, the list order would be: `(list 7 6 5 4 3 2 1)`).

Your function must include a signature, purpose statement, header, and three (3) check-expects using nodes 1, 2, and 4 from the tree in Figure 1 for full credit.

Hint 1: Don't forget the template function for a BST. You should write it before you attempt to fill in the function body (if you don't know where to begin, it should help).
Hint 2: Remember the `append` function appends two *or more* lists into a single list.

Problem 2. (25 points) Consider the following data and structure definitions for a Book and two functions that consume a list of Books:

```
(define-struct book (genre words))
; A Book is a (make-book string number)


; LOB -> number
; counts number of "non-fiction" books in lob
(define (count-nonfiction lob)
  (cond
    [(empty? lob) 0]
    [(cons? lob)
     (cond
       [(string=? (book-genre (first lob))
              "non-fiction")
         (+ 1 (count-nonfiction (rest lob)))]
       [else (count-nonfiction (rest lob))])]))
```

```
; A list-of-books (LOB) is either
; - '()
; - (cons Book LOB)


; LOB -> number
; counts number of "fiction" books in lob
(define (count-fiction lob)
  (cond
    [(empty? lob) 0]
    [(cons? lob)
      (cond
        [(string=? (book-genre (first lob))
               "fiction")
          (+ 1 (count-fiction (rest lob)))]
        [else (count-fiction (rest lob))])]))
```

a) (5 points) Since both `count-nonfiction` and `count-fiction` count books of a certain genre, combine them into a single higher-order function called `count-genre` by parameterizing their differences. Be sure to include a general signature for the `count-genre` function.

b) (5 points) Rewrite `count-nonfiction` and `count-fiction` so that they both use `count-genre`. (Hint: each function's body should be a one-liner).

c) (5 points) Now we want to count the books in our collection based on more than just genre. Design a function called `count-pred` that is even more abstract than `count-genre`, which consumes an LOB and a predicate function for Books, and counts only those Books that satisfy the given predicate. Be sure to include a general signature and purpose statement.

d) (5 points) Now use `count-pred` to implement a function named `count-novellas` that counts all the books in the collection that are considered novellas. A novella is book whose genre is "fiction" and word count is between 30,000 and 60,000 words. Your `count-novellas` function should use a `local` expression to define the predicate function `novella?` that it passes to `count-pred`.

e) (5 points) Now rewrite `count-novellas`, removing the `local` expression and replacing `novella?` with an equivalent `lambda` expression in the call to `count-pred`.

Problem 3. (25 points) Recall the abstract `filter` function, shown with its signature here:

```
filter: (X -> bool) list-of-X -> list-of-X
```

a) (5 points) First, develop the predicate function, `pos-even?`, to determine whether the number it consumes is both positive and even. Be sure to include a signature and purpose statement. No tests are necessary. No `if` or `cond` expressions allowed.

b) (5 points) Next, use `filter` and `pos-even?` (i.e., no `cond` expression or recursion) to develop the function `filter-pos-even`, which consumes a list of numbers and extracts only those elements that are both positive and even. Be sure to include a signature and purpose statement.

c) (8 points) Now, consolidate the functions you wrote above so that the `pos-even?` function is defined within `filter-pos-even` using a `local` expression.

d) (7 points) Finally, simplify `filter-pos-even` by removing the `local` expression altogether, and replacing `pos-even?` with an equivalent `lambda` expression.

Problem 4. (25 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))
```

a) (15 points / 3 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 odd numbers from `nums`

- Expression that determines whether `odds` contains any even numbers

- Expression that determines whether `nums` contains only odd numbers

- Expression that adds 42 to every value in `odds`

- Expression that sums all the odd numbers in `nums` (be careful: `nums` contains numbers that aren't odd!)

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

- `(map number->string odds)`

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

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

- `(filter (lambda (n) (and (< n 8) (even?  n))) nums)`

- `(andmap odd?  (map sub1 evens))`