COMP 360, Fall 2015, Project 2

In this assignment, you will write a number of Racket functions. A few use strings and characters (which are separate data types in Racket), so you may find it useful to check the relevant Racket documentation sections.

Racket reference: http://docs.racket-lang.org/reference
Strings: http://docs.racket-lang.org/reference/strings.html
Characters: http://docs.racket-lang.org/reference/characters.html

For the examples, we use `=>` to mean “evaluates to.”

1. Write a function `only-capitals` that takes a list containing strings and returns another list containing strings that have only the strings in the argument that start with an uppercase letter. Assume all strings have at least one character. Use `filter`, `string-ref`, and `char-upper-case?` to make a 1-2 line solution.

   Ex: `(only-capitals '("I" "go" "to" "Rhodes" "College" "in" "Memphis"))
   => '("I" "Rhodes" "College" "Memphis")

2. Write a function `longest-string1` that takes a list of strings and returns the longest string in the argument. If the list is empty, return "" (the empty string). In the case of a tie, return the string closest to the beginning of the list. Use `foldr`, `string-length`, and no (other) recursion.

   Ex: `(longest-string1 '("I" "go" "to" "Rhodes" "College" "in" "Memphis"))
   => "College"

3. Write a function `longest-string2` that works exactly like `longest-string1` except in the case of a tie, return the string closest to the end of the list. Your solution should be almost an exact copy of `longest-string1`.

   Ex: `(longest-string2 '("I" "go" "to" "Rhodes" "College" "in" "Memphis"))
   => "Memphis"

4. Recall that the normal version of `map` takes a function of a single argument and a list, and returns a new list consisting of the function applied to each element of the original list. Write a function called `map2` that takes a function of two arguments, and two lists of equal length. `map2` returns a new list consisting of the function applied to corresponding pairs of elements from each of the two argument lists.

   Ex: `(map2 + '(1 2 3) '(4 5 6)) ==> '(5 7 9)
   Ex: `(map2 expt '(1 2 3) '(4 5 6)) ==> '(1 32 729)
   Ex: `(map2 cons '(1 2 3) '((4 5) (6 7 8) ())) ==> '((1 4 5) (2 6 7 8) (3))

5. Write a function called `map-any` that takes a function of any number of arguments, and a list of lists. The number of sublists in the list of lists must match the number of arguments that the function takes. `map-any` returns a new list consisting of the function applied to corresponding n-tuples of elements from each of the n sublists of the list argument.

   You will need to use the function `apply`, which takes a function and a list of arguments, and returns the value obtained by applying the function to the arguments.

   Example of apply: `(apply + '(1 2 3)) ==> 6`
6. Write a tail-recursive general-purpose comparison function called least that takes a function f and a list lst. f should be a function that specifies a partial order relation (in other words, a function of two arguments that returns #t or #f by comparing its arguments to determine which one is “smaller”), and lst should be a list of elements of the appropriate data type to pass to f. least should return the “smallest” element in lst. You may assume there will not be any duplicate items in lst (so breaking ties won’t matter).

Note that you can’t use foldr here because foldr is not tail-recursive.

Make sure you don’t accidentally write an exponential time algorithm.

Ex: (least < '(2 4 3 5 1 7 6)) ==> 1
Ex: (least > '(2 4 3 5 1 7 6)) ==> 7
Ex: (least string<? '("I" "go" "to" "Rhodes" "College" "in" "Memphis")
    ==> "College"

7. Write a tail-recursive function called lookup that takes a function and a list of lists. The function must be a predicate (a function that returns #t or #f) of a single argument, and the sublists of the list argument all must have two elements. lookup examines each sublist left to right searching for the first sublist where the car of the sublist satisfies the predicate. When such a sublist is found, the second element of the sublist is returned. If no sublist is found where the car satisfies the predicate, lookup returns #f.

Ex: (lookup (lambda (x) (< x 3)) '((4 "Alice") (2 "Bob") (1 "Carl")))
    ==> "Bob"
Ex (lookup (lambda (x) (< x 1)) '((4 "Alice") (2 "Bob") (1 "Carl")))
    ==> #f

8. Write a function called power-set that takes a list of numbers (representing a set) and returns the power set of that list. Your solution will use recursion, but should use map as well. Using foldr is optional. The order of the elements in the output list is not important (since it represents a set).

Hint: For a set S, define S' as S with some element removed. Then the power set of S can be defined recursively as the union of the power set of S' and the result of a map over the power set of S'.

Ex: (power-set '(1 2 3)) ==> '((1) (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
    [the order of elements in the power set above is not important]

9. Write a function called dot-product that takes two lists representing vectors, and returns their dot product. The dot product of the vector v = (v₁, …, vₙ) and w = (w₁, …, wₙ) is \(\sum_{i=1}^{n} v_i w_i\). Do this with foldr and map2.

Use the following skeleton code, filling in the spots with the [?]:

(define (dot-product v w)
  (foldr [?] [?] (map2 [?] [?] [?] [?] [?])))

Ex: (dot-product '1 2 3) '4 5 6) ==> 32
10. Write a function called `matrix-*-vector` that multiplies a list of lists (representing a matrix), by a list (representing a vector).

Use the following skeleton code, filling in the spots with the `[]`:

```
(define (matrix-*-vector m v)
  (map [?] m))
```

Ex: `(matrix-*-vector '((1 2 3) (4 5 6)) '(-1 1 2)) ==> '(7 13)

11. Write a function called `transpose` that takes a list of lists (representing a matrix) as an argument and returns the transpose of that matrix.

Use the following skeleton code, filling in the spots with the `[]`:

```
(define (transpose m)
  (if (null? (car m)) '()
      (cons [?] (transpose [?]))))
```

Ex: `(transpose '((1 2 3) (4 5 6))) ==> '((1 4) (2 5) (3 6))

12. Write a function called `matrix-*-matrix` that takes two lists of lists (representing matrices) as arguments and returns their product.

Use the following skeleton code, filling in the spot with the `[]`:

```
(define (matrix-*-matrix m n)
  (let ((cols (transpose n)))
    (map [?] cols)))
```

Ex: `(define A '((1 2 3) (4 5 6)))
  (define B (transpose A))
  (matrix-*-matrix A B) ==> '((14 32) (32 77))
  (matrix-*-matrix B A) ==> '((17 22 27) (22 29 36) (27 36 45))

Assessment

Solutions should be:

- Correct
- In good style, including indentation and line breaks
- Written using features discussed in class. In particular, you must not use any mutation operations nor arrays (even though Racket has them).

Turn-in Instructions

- Put all your solutions in one file, `proj2_lastname_firstname.rkt`, where `lastname` is replaced with your last name, and `firstname` is replaced with your first name.
- Upload your file to Moodle before the project deadline.