Lab 7

0. Goals and Instructions

Relevant sections of text: *HtDP/2e through Section 19.2.*

Goals:

- abstracting functions
- analyzing and improving performance
- using the built-in abstraction functions (`foldr`, `map`, etc.)
- using `local` and `lambda`

Instructions. Please submit all answers via Sakai in a single file named `lab7.rkt`. Do all work in pairs. Include the names of both partners at the top of the file in a comment. You need only submit one lab per pair.

Also submit your current version of Project 1. Call it `fish.rkt`. You should have completed Milestone 3. The starter file rubric shows exactly which features will be graded.

1. Basic abstraction problems

Problem 1. Abstract the following two functions into a single function:

```
;; mini : NELON  ->  Number
;; to determine the smallest number
;; on alon
(define (mini alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [(cons? (rest alon))
     (cond
       [(< (first alon)
           (mini (rest alon)))
        (first alon)]
       [else
         (mini (rest alon))])]))
```

```
;; maxi : NELON  ->  Number
;; to determine the largest number
;; on alon
(define (maxi alon)
  (cond
    [(empty? (rest alon)) (first alon)]
    [(cons? (rest alon))
     (cond
       [(> (first alon)
           (maxi (rest alon)))
        (first alon)]
       [else
         (maxi (rest alon))])]))
```

Both consume non-empty lists of numbers [“NELON”] and produce a single number. The left one produces the smallest number in the list, the right one the largest. Name your abstracted function `extreme`. Remember to write a general, parametric/polymorphic type signature/contract for your abstract function!!!

Define `mini1` and `maxi1` in terms of `extreme`. Test each of them with the following three lists:

```
(list 3 7 6 2 9 8)

(list 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
```

```
(list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30)
```

Why are they slow on the long lists? Note you can use the function `time` to time a program, e.g.,

```
  (time (mini1 LIST1))
```

Explain in a comment why the times are as long as they are.

Problem 2. In this problem you will improve the performance of the abstracted function from problem 1. The new and improved version of the abstract function should be named `extreme.v2`. Define `mini.v2` and `maxi.v2` to use `extreme.v2`.

Here is how to design the improved abstract function. First, introduce a `local` name for the result of the natural recursion. Then introduce a local, auxiliary function that picks the "correct" one of the two numbers to produce as the result in the `cons?` clause (basically, this auxiliary function replaces the `cond` in your first abstract function).

Test `mini1.v2` and `maxi1.v2` with the same inputs again, and observe the timing. Report on any timing improvements in a comment with a brief explanation of why the performance improved.

2. Bank account operations

Problem 3. Define a type Account, which is a structure with three fields: an account `number` (a Number), the account holder's `name` (a String), and the account `balance` (a Number).

In the following, LoA = [List-of Account].

Design the following functions. Each definition should use one or more of the list abstraction functions together with a `local` auxiliary function definition or a `lambda` function. Do not use explicit recursion. Do not use an external auxiliary function.

The functions:

- `get-bigs`: given an LoA and a threshold, returns all Accounts in the LoA whose balance is greater than or equal to threshold,
- `get-names`: given an LoA, returns the list of names of the accounts,
- `total-holdings`: given an LoA, returns the sum of their balances,
- `some-negative?`: given an LoA, determines whether the list contains an account with a negative balance,
- `name-check?`: given an LoA, returns `#true` if and only if all name fields are non-empty,
- `sort-accounts`: given an LoA, returns the LoA sorted by decreasing account balance,
- `count-multiples`: given an LoA and a name, returns the number of accounts in the LoA whose account name matches the given name,
- `add-interest`: given an LoA and an interest rate, returns the list of accounts obtained by adding interest to the balances of each account at the specified rate. For each account, if the original balance was b and the rate r, the new balance will be $(1+r)b$.

Problem 4. Design function `create-accounts`, which, given a natural number n and real number m, returns an LoA of length n. The names on the accounts are "`N1`", "`N2`", . . . ; the account numbers are 101, 102, ... The balance on each account is a random rational number b in $[0, m]$ with at most 2 decimal places, i.e., $100b$ is an integer. For example, if m is 10, then 0, 1.57, 3, 9.99, and 10 are all possible balances. Your function should be capable of producing any balance satisfying these constraints, and should choose them with approximately equal probability. Your definition should use `build-list` and should not use explicit recursion.

Hint: consider choosing the number of cents randomly, and then adjusting to get a dollar amount.