CSE 216 Handout 2: PS 2 September 10, 2021

This problem set is due at 11:55 pm on Friday, September 17, 2021. Don’t go by the due date that you see on Blackboard because it is in ET. Go by the one given in this handout.

Be sure to include a comment at the top of each file submitted that gives your name and email address.

Submit your solution file named ps2.scm that contains all of your solutions on Blackboard. Multiple submissions are allowed before the due date.

Note: Please read the "Cooperation vs. Cheating" (i.e., "Academic Integrity") section of the syllabus before you start this problem set and future problem sets.

General Instructions

You are free to define auxiliary functions, as many as you want, and you may assume that the functions defined earlier "carry over" into any of the functions downstream in the entire problem set. When I ask you to write a recursive function, say foo, the function foo itself does not have to be recursive if it calls an auxiliary function that is recursive. We saw many examples like that in Recitation 1.

As usual all of your programs that you submit should run without any errors. If there is any error and the program crashes, you will get no credit.

In this problem set you are not allowed to use set!, set-car!, set-cdr!, or begin unless I specifically mention that you are allowed to use them.

Problem 1

(a) Write a recursive function named count that takes one list of atoms as its argument and returns the number of atoms in the list. Add at least three calls to it including boundary cases.

(b) Write a recursive function named sum that takes one linear list of numbers as its argument and returns the sum of the elements in the list. Add at least three calls to it including boundary cases.

(c) Write a recursive function named position that takes two arguments, an atom and a list of atoms, and returns the position of the first occurrence of the atom in the list. Let’s use 1 as the first position, namely it will return 1 if the atom is the first element of the list. If the atom does not occur in the list, it should return 0. Add at least three calls to it including boundary cases.

(d) Write a recursive function named occurs? that takes two arguments, an atom and a list of atoms, and returns true if the atom occurs in the list and false otherwise. Add at least three calls to it including boundary cases.

(e) Write a function named add-to-set that takes two arguments, an atom and a list of atoms, and adds the atom into the list maintaining the set properties. Add at least three calls to it including boundary cases.

(f) Write a recursive function named union that takes two lists (sets) of atoms as arguments and returns the union of the two sets. Add at least three calls to it including boundary cases.
(g) Write a recursive function named `get-uniques` that takes one list of atoms as its argument and returns a list containing all the atoms in the list without duplicates. The input arguments may or may not have duplicate elements. Add at least three calls to it including boundary cases. Be sure to use some input lists with duplicates.

(h) Write two functions named `reverse` and `reverse-aux`. I will start them, and you complete the rest of them.

```
(define (reverse u)
  (reverse-aux u '()))

(define (reverse-aux u v)
  ...)
```

In `reverse-aux` you will accumulate the list being reversed in `v`, which will eventually be returned as its return value. You may assume the argument `u` is a list of atoms. Add at least three calls to `reverse` including boundary cases.

(i) Write a recursive function named `frequencies` that takes a list of atoms and returns a histogram (frequency table) as a two-level list. The input argument may or may not have duplicate elements. Add at least three calls to it including boundary cases. For example, if the input list is `(a b a b a c a)` then it should return `(((a 4) (b 2) (c 1)))`.

(j) Write a recursive function named `assoc` (short for association) that takes two arguments: an atom as a key and an association list of key-value pairs, and returns the value associated with the key (first argument) in the association list. If the key does not exist in the association list, it should return `-1`. Add at least three calls to it including boundary cases. Here are example calls and the expected return values:

```
(assoc 'c '(((d 3) (a 2) (c 14) (f 4) (k 23))))
returns 14

(assoc 'g '(((c 3) (a 2) (f 40) (k 23))))
returns -1
```

You may represent a pair as a `cons` pair or a list of length 2.

(k) Write a recursive function named `assoc-update` (short for association update) that takes three arguments: an atom as a key, an atom as a new value to be set, and an association list of key-value pairs, and returns the updated assoc list. If the key does not exist in the association list, it should return an assoc list that is same as before, not necessarily the same exact list in the original memory locations. That is, the one being returned may be a rebuilt one with the same contents as the original. Add at least three calls to it including boundary cases. The order of pairs in the result may or not be the same as the order of the elements in the original input argument list. Here a pair is represented as a list of length 2. Here are example calls and the expected return values:

```
(assoc-update 'c 30 '(((d 3) (a 2) (c 14) (f 4) (k 23))))
returns (((d 3) (a 2) (c 30) (f 4) (k 23)))

(assoc-update 'g 10 '(((c 3) (a 2) (f 40) (k 23))))
returns ((c 3) (a 2) (f 40) (k 23))
```

Problem 2

(a) We studied some functions that compute with trees, e.g., `scale-tree` and `enumerate-tree` to name a few, in the lecture `lec4.scm`.

Given a tree we want to compute the sum of the squares of the leaves that are odd in the tree. One could write it as follows:
(define (sum-odd-squares tree)
  (cond ((null? tree) _______
         ((not (pair? tree)) _________________
          (else (+ (sum-odd-squares _______________) (sum-odd-squares _______________)))))))

Fill in the blanks to complete the definition above. (Hint: read SICP)

(b) This time let’s write a function that constructs a list of all the even Fibonacci numbers \( F_{ib}(k) \), where \( k \) is less than or equal to the given integer \( n \):

(define (even-fibs n)
  (define (next k)
    (if (> k n)
        ’()
        (let ((f (fib k)))
          (if (even? f)
              (cons f _______________)
              (next _______________))))
  (_________ 0))

Fill in the blanks to complete the definition above.

(c) The function \( \text{sum-odd-squares} \) that you just completed above does not use any higher-order functions. You can reformulate your solution approach to take advantage of higher-order functions. So, rewrite it with a new name \( \text{sum-odd-squares-high} \) that still takes one tree argument and computes the same result as \( \text{sum-odd-squares} \) does, but uses some higher order functions. Note that you will have to write some additional functions to complete the solution in this new approach. Add all the necessary functions to make your solution run without any errors.

(d) Similarly, the function \( \text{even-fibs} \) that you just completed above does not use any higher-order functions. You can reformulate your solution approach to take advantage of higher-order functions. So, rewrite it with a new name \( \text{even-fibs-high} \) that still takes one integer argument and computes the same result as \( \text{even-fibs} \) does, but uses some higher order functions. As you did in part (c) above, you will have to write some additional functions to complete the solution in this new approach. Add all the necessary functions to make your solution run without any errors.

Problem 3

In PS 1 we wrote \( \text{plus} \), \( \text{diff} \), \( \text{times} \), and \( \text{quotient} \) using only a small number of Scheme built-in operators, functions and special forms. In PS 1, you wrote these functions to compute with numeric data. In this problem we will build them for symbolic data. This problem is meant to develop your ability to make abstractions with symbolic data, something Lisp/Scheme is very good at.

As we did in PS 1, you are restricted to use only a certain functions and special forms in this problem and they only include these:

\[
\text{car, cdr, cons, atom?, equal?, null?, if, let, define.}
\]

\( \text{atom?} \) is not provided by Racket, so I will give you a definition as follows and you can include it in your solution file:

\[
\begin{align*}
  \text{(define (atom? x)} \\
  \text{(not (pair? x)))}
\end{align*}
\]

Do not use \text{begin} in this problem set except perhaps as a tool for debugging. In other words, your submitted file should not have it.

In this problem all arguments and return values are restricted to lists of the following atoms:
These correspond to the decimal integers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Now, for example, the number 497 is represented as the list (FOUR NINE SEVEN). To keep things simple, don’t worry about negative numbers, decimal points, etc. You may use one (and only one) global definition in your entire solution, namely,

(define *digits* '(OH ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE))

In each of the following, you may find it helpful to define auxiliary functions (multiple of them in some cases) that do things like stripping leading zeros, testing for carries, etc.

(a) Define a function named a1, which adds one to a single digit modulo 10. For example (a1 ‘TWO) returns THREE, (a1 ‘NINE) returns OH, etc.

(b) Define a function named add1, which calls a1, and possibly other functions. Like 1+, add1 takes a single argument and returns a list representing a number one greater than the argument. For example, (add1 '(oh)) returns (one), (add1 ' (oh oh nine nine nine)) returns (one oh oh oh), etc.

(c) Also add s1 and sub1 in a similar manner.

(d) Now write your functions plus, diff, times, and quotient using your add1 and sub1 functions instead of the 1+ and 1- that you used in PS 1. Some sample calls are given here:

(plus '(NINE) '(FIVE)) returns (ONE FOUR)
(times '(THREE) '(SEVEN)) returns (TWO ONE)
(diff '(ONE FOUR) '(NINE)) returns (FIVE)
(quotient '(ONE ONE) '(FIVE)) returns (TWO)

For each of these functions, add a good number of test cases that include those that address boundary cases.

Optional, meaning not required: For really intense programmers Demonstrate that your functions continue to work appropriately if *digits* is changed by the following (This is the only use of set! allowed in this PS):

(set! *digits* '(OH ONE))

A good implementation should allow you to switch number bases by this simple one line change without changing anything in other parts of your program.