Assignment 1: Scheme Review

CS 442/642

Due January 16th in class

The purpose of this assignment is to give you a chance to remember all that you have forgotten about Scheme from first year (or a chance to quickly get up to speed, if you have not encountered Scheme before). This assignment is worth slightly less than the other assignments.

All of the questions below can be answered using knowledge and techniques that were (hopefully) taught in first year (except for the questions on mutation in Scheme, which may be new to some of you). However, if you have trouble remembering first year, or are otherwise struggling with certain questions, do not hesitate to visit your instructor during office hours for help reviewing.

Use the PLT Scheme distribution (i.e., Racket; set language to R5RS) to solve the following problems:

1 Pure Functional Programming; Higher-Order Functions

Part A: Two popular programming patterns fold a function across a list in order to either reduce a list to a single value, or to accumulate a single value from the elements of a list. These patterns are typically implemented with functions called foldl and foldr that each take 3 parameters: a binary function f, an initial value i, and a list L.

Given a list, \(L = (e_1 e_2 e_3 \ldots e_n)\), the function foldl folds the function F across the list from the left. It begins by applying F to the first element of the list and the initial value I and then successively applies F to the next element of the list and the previous application of F, until no more elements remain. That is,

\[
(foldl f i L) = (f e_n (f \ldots (f e_3 (f e_2 (f e_1 i))) \ldots ))
\]

The partner to foldl is foldr which performs similarly, but folds from the right, beginning with the last element of the list:

\[
(foldr f i L) = (f e_1 (f e_2 (f e_3 (f \ldots (f e_n i) \ldots ))))
\]

With both functions, if L is '\(', the empty list, the result is simply the value of i. For example, the result of \(foldl \, cons \, '(()) \, '(a b c))\) is \(c b a\). For part A, you are to implement foldl and foldr. Aim for as concise an implementation as possible, and do not use one of foldl/foldr to implement the other.

Part B: Both foldl and foldr can be used to do a variety of useful things. For each of the following, you are to select the most appropriate function (foldl or foldr) and fill in the blanks to create the desired equivalences. The evaluation of each solution should require \(O(\text{length } L)\) function applications. [Elementary built-in functions such as car, cdr, cons, eq?, +, * etc. can be considered to be \(O(1)\) while other list manipulation functions such as length, append, equal, and reverse cannot; make no assumptions about the running time of these operations].

1. \((\text{fold}\, \_\_\_\_\_\_\_\_\, L)\) sums the elements of L.

2. \((\text{fold}\, \_\_\_\_\_\_\_\_\, L)\) is equivalent to \((\text{length } L)\).
3. (fold_ ___ ___ L) gives the maximum value in L. You may assume that the list is never empty, but do not assume any known minimum or maximum possible value.

4. (fold_ ___ ___ L) is equivalent to (exists p L) [exists returns #t if there is an element x in the list L such that (p x) is true; otherwise, exists returns #f].

5. (fold_ ___ ___ L) is equivalent to (map f L) [map applies f to each element of L and returns a list of results in the same order].

6. (fold_ ___ ___ L) is equivalent to (append L M).

Part C: Answer the following questions:

1. In cases where foldl and foldr can be substituted one for the other, without affecting the result, is there any good reason to use one instead of the other? Explain.

2. Compare your implementation of exists using fold_ above with the following implementation:

```scheme
(define exists (lambda (p L)
    (if (null? L) #f
        (if (p (car L)) #t (exists p (cdr L)))
    ))
)
```

Which of them is more efficient, and why?

2 Impure Functional Programming in Scheme

For the problems in this section, you may use the Scheme assignment operator set! and other impure features to change the values of variables.

Part A: Write a Scheme function kill3 that, given a list L, modifies L so that L’s third element is removed. You may assume that only lists of length at least 3 are passed to kill3. The function kill3 does not itself return anything. For example:

```scheme
> (define my-list '(2 4 6 8 10))
> (kill3 my-list)
> my-list
(2 4 8 10)
```

Part B: Write a Scheme function countme that returns the number of times it has been called. In particular, the first call to (countme) should return 1, the second should return 2, and so on. You may include auxiliary definitions if you like, and you may assume that the client will not actively try to “break” your function (but can you think of a way, without using modules, to solve this problem robustly?).

Part C: Fill in the blank in the following expression:

```scheme
(map _________________________________ lst)
```

such that if lst is a list (a1 a2 ... an), then the result of the expression is ((+ a1 1) (+ a2 2) ... (+ an n)). For this problem, unlike in part B, your solution must consist entirely of an expression to fill in the blank. No definitions external to the map expression are permitted. Note: although the Scheme standard says that map processes the list in unspecified order, you may assume that the order actually used is left-to-right.
Submission

Question 1 should be submitted on paper in class. Question 2 should be submitted both on paper and electronically, via submit:

submit cs442 a1 .

The distribution of marks for this assignment is expected to be 60% for Question 1, 40% for Question 2.