

Programming Languages
CSCI-GA.2110.001 Fall 2021

Scheme Assignment
Due Tuesday, March 26 at 11:55pm

Your assignment is to write a number of small Scheme functions. All code must be purely functional (**no** use of `set!`, `set-car!`, or `set-cdr!` allowed) and should be concise and elegant.

Scheme programming will be on the mid-term exam on October 27, so it is essential that you finish this assignment on time. Do not work on the code with anyone else. Put all your code into **single file** (with a “.scm” extension) and upload it to Brightspace.

Additionally, unless otherwise noted, each recursive function that you write must be preceded by a **comment that shows your reasoning** about the recursion. For example, the comment before the solution to problem 1 below might be:

```
;; (byTwos n m) returns the list of every other integer starting with n up to m.
;; Base Case: if n > m, the result is the empty list.
;; Hypothesis: Assume (byTwos (+ n 2) m) returns the list of every other integer
;;               from n+2 up to m.
;; Recursive step: (byTwos n m) returns (cons n (byTwos (+ n 2) m))
```

1. Define the function `(byTwos n m)` that returns the list of integers starting with `n` such that each successive element is two greater than the previous element and no element is greater than `m`. For example,

```
> (byTwos 1 20)
(1 3 5 7 9 11 13 15 17 19)
```

This function should be around 3 or 4 lines of code. If you are writing more than that, you are probably thinking about it incorrectly.

2. Write the function `(compress L)` that returns a list of all the atoms (non-list values) contained in `L` or in any nested list within `L`. For example,

```
> (compress '(1 (2 3 (4 5) (6 7 (8))) 9) 10))
(1 2 3 4 5 6 7 8 9 10)
```

This function should be around 6 lines of code. Note: `(list? x)` returns true if `x` is a list (including the empty list), false otherwise.

3. Write a **linear-time** reverse function, `(rev-all L)` which reverses the elements of a list `L` and, if the `L` contains nested lists, reverses those nested lists as well. For example,

```
> (rev-all '(1 2 (3 4) (5 6 (7 8) 9) 10))
(10 (9 (8 7) 6 5) (4 3) 2 1)
```

To make sure it is linear time (in the total number of atoms in the list at any level of nesting), you should **not call append**. Hint: You should use a **helper function**, analogous to the `(rev L result)` function that I wrote in class. Feel free to adapt my **rev code**.

This code should be around 9 lines.

4. In Scheme, to compare two numbers, you can use the `=` function, as in:

```
> (= 10 3)
#f
```

To compare two atoms, you can use the `eq?` function, as in:

```
> (eq? 'hello 'goodbye)
#f
> (eq? 'hello 'hello)
#t
> (eq? 3 3)
#t
```

However, `eq?` does not perform an element-by-element comparison on lists. For example,

```
> (eq? '(1 2 3) '(1 2 3))
#f
```

Instead, Scheme provides the function `equal?` that does work on lists (and atoms, too):

```
> (equal? '(1 2 3) '(1 2 3))
#t
> (equal? 3 4)
#f
```

Write your own equality function, (`equalTo? x y`), which works the same as (`equal? x y`). You can call `eq?`, but you cannot use `equal?`.

This function should be around 5 lines of code.

5. Write a function (`equalFns? fn1 fn2 domain`), where the parameters `fn1` and `fn2` are functions and `domain` is a list of values, that returns true iff `fn1` and `fn2` always returns the same value when applied to the same element of `domain`. **Don't assume that** `fn1` and `fn2` always return atomic values. For example,

```
> (equalFns? (lambda (x) (* x 2)) (lambda (y) (+ y y))
          '(1 2 3 4 5 6 7 8 9 10 11 12))
#t
> (equalFns? (lambda (x) (* x 2)) (lambda (y) (+ y 2)) '(2))
#t
> (equalFns? (lambda (x) (* x 2)) (lambda (y) (+ y 2)) '(2 3 4 5))
#f
> (equalFns? (lambda (L) (car L)) (lambda (L) (cadr L))
          '(((2 3) (2 3)) ((4 5) (4 5))))
#t
```

This function should be around 5 lines of code.

6. Write a function, (`same-vals fn1 fn2 domain`), that returns the list of all elements x of `domain` such that (`fn1 x`) and (`fn2 x`) return the same value. For example,

```
> (same-vals (lambda (x) x)
             (lambda (y) (abs y)) ;; abs give the absolute value
             '(-3 -2 -1 0 1 2 3))
(0 1 2 3)
```

This function should be around 6 lines of code.

7. Write a function (`split x L`), where x is a number and L is a list of numbers, that returns a list containing two lists: The first list contains the numbers in L less than or equal to x and the second list contains the numbers in L greater than x . For example,

```
> (split 6 '(1 9 2 8 3 10 4 6 5))
((1 2 3 4 6 5) (9 8 10))
> (split 7 '(1 9 2 8 3 10 4 6 5))
((1 2 3 4 6 5) (9 8 10))
```

Hint: The base case is, when L is an empty list, return `'(()())`. This function should be around 8 lines of code.

8. Write a function (`psort L`) that implements a partition sort (similar to Quicksort). It should use your `split` function, above. Given a list L , if L is non-empty, then `split` should be called using the first element of L to partition the rest of L . Then, `psort` should be called recursively to sort each of the two lists returned by `split`. Finally, the sorted result is constructed from the elements of the two sorted lists, as well as the first element of L . For example,

```
> (psort '(5 3 8 6 1 0 2))
(0 1 2 3 5 6 8)
```

This function should be around 6 lines of code. Be sure to only call `split` once at each level of the recursion.

9. Write a single function (`applyToList f`), where f is a parameter that is a function, that returns a function that takes a list L as a parameter and applies f to every element of L , returning the resulting list as the result. For example,

```
> ;; define g to be the function resulting from calling applyToList
    (define g (applyToList (lambda (x) (* x 2))))

> ;; call g
    (g '(1 2 3 4 5))
(2 4 6 8 10)

> ;; call the function resulting from calling applyToList
    ((applyToList (lambda (y) (car y))) '(1 2 3) (4 5 6) (7 8 9))
(1 4 7)
```

You must use the built-in `map` function in your definition of `applyToList`. Thus, you do not need to define a recursive function and therefore no comment showing recursive reasoning is required. This function should be around 2 lines of code.

10. Write a function (`newApplyToList f`) which behaves exactly like `applyToList` above, except that you **cannot** use the built-in `map` or any other built-in function except `cons`, `car`, and `cdr`. Also, you cannot define any helper function outside of `newApplyToList`, but you can define functions within it (Note: do not use `(define ...)` inside a procedure). You do not need to provide a comment showing your recursive reasoning.

This function should be around 6 lines of code.