Scheme Lab 00000010

*Parts of this assignment are adapted (with permission) from Professor Brent Yorgey’s CS 194 assignment.*

Please submit all responses to your Dropbox in a file called lab02.rkt.

**The next four methods will be mimicking functionality that already exists within Scheme, but for these exercises those functions will (obviously) be considered off-limits. You may use no other functions that process lists besides list, cons, and members of the car and cdr family.**

Make

**(get-element list n)**

which takes a list and returns the nth element. (Just like arrays, we will start counting from zero). This is mimicking the *list-ref* function.

Make

**(append-element list y)**

which takes a list x and an element, y, and returns list with y added to the end. This is mimicking the *append* function.

> (append-element '(1 3 5) 7)

'(1 3 5 7)

Make

**(append-list first second)**

which takes a list first and a list second, and returns a combined list. This is again mimicking the *append* function.

> (append-list '(1 3 5) '(4 6 8))

'(1 3 5 4 6 8)

Make

**(backwards list)**

which takes a list, and returns a reversed list. This is mimicking the *reverse* function.

> (backwards '(1 3 5 8))

'(8 5 3 1)

Once you are done with those four examples and their test cases, you should now feel free to use reverse, append, and list-ref. The entirety of Racket is now open to you other than mutability. You’ll know these things when you see them because, by convention, functions that have side-effects end in exclamation marks, like (set! x 5). Avoid these like a cat avoids cucumbers, but otherwise, the world is now your oyster.

Parenthesis Balancer

This exercise is a re-do of something you already did last year to help you wrap your head around Scheme recursion, linked lists, and stacks! We will take advantage of the fact that a linked list wherein you only add or remove from the head is, well, a *stack*. (Make sure that that makes sense to you before you move along.)

For this exercise, you will make a predicate function, paren-balanced? that takes a string, and decides whether the parenthesis, square braces, and curly braces are balanced. (What does “predicate function” mean, you ask? It means a function that returns a boolean!)

In this case, your function will decide if the parenthesis in the string are balanced. The only letters that we will observe closely will be "(", "[", "{", ")", "]", and "}". Any other letters should be ignored.

You will find the Racket string reference page to be necessary. (<https://docs.racket-lang.org/reference/strings.html>) Among other things, you will need a way to split a string into either characters or size-one strings. (Note that if you use characters, 'a' in Java is #\a in Scheme.)

You will also want optional parameters. These are parameters that need not be in the initial function call, but will be given an initial value if they are not called with one. Here is a function that takes an optional number with a default value 0, and a list with a default value of null.

(define (my-function (num 0) (lst null)   
 … )

This can be called in many ways, such as:

* (my-function),
* (my-function 7), or
* (my-function 9 '(a d)).

Regardless, the coder would be freely able to use num and list within the function as they will have default values if the caller does not provide one. Use this feature to begin your function with an empty stack.

Here are a few sample runs to help you understand the function. As always, this list is not even remotely meant to substitute for your own testing of edge cases, it is just a way to help you understand what the function is doing.

> (paren-balanced? "(this function {} is [] balanced!)")

#t

> (paren-balanced? "[ { ] }")

#f

Any helper methods that you create along the way should be internal to paren-balanced?. The only public-facing method in this section should be paren-balanced? itself.