Scheme Lab 00000011

Please submit all responses to your Dropbox in a file called lab03.rkt.

*Warm-up: Easy lambdas*

Create the following two functions:

func1(x) =

func2 =

Create tests to show that your two functions are the same.

Create:

dec =

*Maps*

Create a function, evens, that maps a list to Booleans that describe whether each number is even or not.

Write a function called squares that takes a list of numbers as an argument and returns a list of the squares of those numbers. Use a lambda and map to define the function.

Write a function called bigs that takes a list of pairs of numbers (such as '((1 3) (7 4) (1 5))) as an argument and returns a list of the larger elements (e.g. '(3 7 5)). Use a lambda and map to define the function.

*Infix Evaluation*

Scheme works naturally in prefix. The goal of the last portion of this lab is to create an infix evaluator. Please note that you **do not** need to define every function using the lambda syntax for this portion of the lab. This lab involves passing functions as first-order objects. We will do this in three parts.

1. eval-rpn

First, create a method **eval-rpn** that will take a series of operators and run them:

(eval-rpn (list 3 4 5 + /) '()) should output 1/3

(eval-rpn (list 2 5 \*) '()) should output 10

The extra empty list is a stack that you can use as you do your evaluations. You will need to push and pop this stack as you progress through the function. Play around with this, and you will be able to determine the rules.

One word of warning – we are not using the mutable parts of Scheme (such as the **set** function), so you will have to pass any states (such as stack states) you want to each recursive call as you progress.

1. in->postfix

Then create a method **in->postfix**, which takes a list and two empty stacks, and converts an infix sequence to an rpn sequence. We will be following the Shunting-Yard algorithm to do this. A good explanation of the algorithm is here: <https://brilliant.org/wiki/shunting-yard-algorithm/>

The trick to this is to use the two stacks: the **operations** stack, and the **operands** stack. In postfix, operations are performed as soon as they are encountered. We will ultimately be returning the **operands** stack, so we need to be careful about what order we place things onto that stack.

Numbers are immediately added to the **operands** stacks as they are encountered.

Operations (+\*/-) can either be moved from the original list to the **operation** stack, or from the **operation** stack to the **operands** stack based on precedence of the operators. It may take you a little while to figure this out what these rules should be, but keep playing with it and you will get there. (Again, remember that you want to put things that have to *happen* first on **operands** first.)

The **operands** stack will contain the final answer. What do you think you should do if the original list becomes null, but there are still commands on the **operation** stack?

(in->postfix (list 2 / 5)) --> (2 5 #<procedure:/>)

(in->postfix (list 1 \* 2 + 3)) --> (1 2 #<procedure:\*> 3 #<procedure:+>)

(in->postfix (list 1 + 2 \* 3)) --> (1 2 3 #<procedure:\*> #<procedure:+>)

(in->postfix (list 1 + 2 \* 3 / 4))   
--> (1 2 3 #<procedure:\*> 4 #<procedure:/> #<procedure:+>)

Finally, add to **in->postfix** the ability to work with parentheses. The idea is that, when you find a "(" in your operations stack, just throw it onto the operations stack to save it for later. If you find a ")", however, you now need to process everything inside that parenthesis pair. Move every item from operations onto operands until you find your "(" again. Clear out that "(", and presto-bingo, you have dealt with parentheses!

1. infix

Finally, create the **infix** function. You’ve already done most of the work! **Infix** will call in->postfix on an infix list, and run eval-rpn on it.

(infix (list 2 \* "(" 3 + 4 ")")) 🡪 14

Congratulations, you are done!