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 =

*Consing a list of two*

Create the following functions:

operate =

(?) makes a list of f applied to a and f applied to b.

Now use operate to make times3, which multiplies two numbers by 3.

(times3 7 15) should return '(21 . 45)

*Maps*

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

Create a function, listTimes4, that maps a list to a list that has every item multiplied by 4.

*Infix Evaluation*

Scheme works naturally in prefix. The goal of the last portion of this lab is to create an infix evaluator. We will do this in three parts

1. rpn

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

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

(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. evaluate

Then create a method **evaluate**, which takes a list and two empty stacks, and converts an infix sequence to an rpn sequence. The trick to this is to use the two stacks: the **operations** stack, and the **operand** stack, which eventually gets returned. Numbers are automatically added to the **operand** 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 **operand** 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.

The **operand** 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?

(evaluate (list 2 / 5)) --> (2 5 /)

(evaluate (list 1 \* 2 + 3)) --> (1 2 \* 3 +)

(evaluate (list 1 + 2 \* 3)) --> (1 2 3 \* +)

(evaluate (list 1 + 2 \* 3 / 4)) --> (1 2 3 \* 4 / +)

Finally, add to **evaluate** the ability to work with parentheses. The idea is that, when you find a "(" in your operations stack, just throw it onto the operands stack. If you find a ")", however, you need to move every item from operands onto operations until you find a "(" 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 evaluate on an infix list, and run rpn on it.

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

Congratulations, you are done!