CSC 261 **Lab 1: Recalling Scheme** **Spring 2020**

Assigned: Monday 20 January

Due: Monday 27 January 10:30 pm

Objectives:

- Reinforce Scheme basics
- Recall how to write recursive procedures over lists and numbers using named `let` and tail recursion
- Explore clever uses of higher-order procedures

Collaboration: This homework assignment must be completed individually.

A common problem in the AI subfield of machine learning is to do regression on a set of points. That is, assume points are observations from some unknown function, and we wish to infer the function so that we may make predictions by interpolating between these points.[1] One approach to this problem is to model the unknown function as a polynomial, which means one must determine the coefficients. In this assignment, we'll develop some useful Scheme functions for defining and evaluating "encapsulated" polynomials.

Assignment

Recall that `map`, along with many other useful Scheme habits often benefit from a technique called *sectioning* (or more general *currying*), which fixes one parameter of a binary (two-argument) procedure. For example, `left-section` takes a procedure and the first argument to that procedure, and returns another procedure that accepts the second (right) argument, with the operation and first (left) argument fixed.

```
(define left-section
  (lambda (proc left)
    (lambda (right)
      (proc left right))))
```

1. Write the companion procedure `right-section`.[2]

2. In addition to *producing* procedures as return values (as `left-section` and `compose` do), we can also use procedures as parameters (as in `map`). Recall that `(apply` *`proc args`*`)` is also in this second category. Using `map` and `apply`, write a procedure `(dot-product v1 v2)` that takes two lists of numbers having the same length and produces their dot product, that is, the sum of the products of the corresponding entries in `v1` and `v2`. For example,

   ```
   > (dot-product (list 1 2 3) (list 4 5 6)) ;; 1*4 + 2*5 + 3*6
   32
   ```

3. Using named `let`[3] and tail recursion,[4] write the procedure `(iota` *`num`*`)` that produces the list of numbers from `0` to **`num`** $-$ 1 in order. For example,

   ```
   > (iota 5)
   '(0 1 2 3 4)
   ```

[1] For example, see AIMA section 18.2, pp. 695–697, or https://en.wikipedia.org/wiki/Kriging.

[2] For future reference, the general functionality of `left-section` is available in standard Racket as `curry` and `right-section` as `curryr`. Take note!

[3] If you've forgotten how to write a named `let`, see, e.g., Dyvbig, *Recursion and Iteration* http://www.scheme.com/tspl4/control.html#./control:s20

[4] If you've forgotten about tail recursion, see e.g., Davis et al. *Helper Recursion,* http://www.cs.grinnell.edu/~weinman/courses/CSC261/2015F/misc/recursion-patterns.html

Your solution must *not* call `reverse`; if your original attempt produces results in the wrong order, you must think about building the result in the opposite order.[5]

4. Write a procedure, (`polynomial-term` *c n*) that returns the function $f(x) = c \cdot x^n$.

```
> (define two-x-cubed (polynomial-term 2 3))
> (two-x-cubed 1) ; 2 * 1 * 1 * 1
2
> (two-x-cubed 3) ; 2 * 3 * 3 * 3
54
> (define three-x-squared (polynomial-term 3 2))
> (three-x-squared 1) ; 3 * 1 * 1
3
> (three-x-squared 5) ; 3 * 5 * 5
75
> ((polynomial-term 5 4) 2) ; 5 * 2^4
80
```

5. Write a procedure (`polynomial` *coeffs*) that takes a list of coefficients for the terms x^0, x^1, x^2, ... of a polynomial and produces a function that takes a single value, evaluating the polynomial given those coefficients at that value.

```
; Create the polynomial f(x) = 1*x^0 + 4*x^1 = 1 + 4*x
> (define line (polynomial (list 1 4)))
> (line 5) ; Evaluate f(5) = 1 + 4*5
21
; Create the polynomial g(x) = 1 + 4*x + 3*x^2 - 2*x^3
> (define cubic (polynomial (list 1 4 3 -2)))
> (cubic 5) ; Evaluate g(5) = 1 + 4*5 + 3*5^2 - 2*5^3
-154
```

Note: Make your solution as concise as possible and do ***not*** (explicitly) use recursion; existing helpers may use recursion.

6. Recall that the derivative of a polynomial term may be given by $\frac{\mathrm{d}}{\mathrm{d}x} c \cdot x^n = c \cdot n \cdot x^{n-1}$ for $n \geq 1$. Write a procedure (`polynomial-derivative-coeffs` *coeffs*) that takes a list of coefficients and produces a list of coefficients of polynomial's derivative.

```
> (polynomial-derivative-coeffs (list 1 4 3 -2))
'(4 6 -6)
```

7. It can often be tedious to manually `compose` the same function with itself several times. Often, we do not know in advance how many times a procedure should be applied.

   Write a new procedure, `nest` that takes two parameters, a unary procedure *f* and an integer *n*. The value produced is a new procedure that results from composing together *n* copies of `f`.

   Note that *n* must be at least 1 for `nest` to make sense.

```
> (define l-s left-section)
> (define plus5 (nest (l-s + 1) 5))
> (plus5 6)
11
> (define duplicate (lambda (val n) ((nest (l-s cons val) n) null)))
> (duplicate "hello" 5)
'("hello" "hello" "hello" "hello" "hello")
```

[5]For future reference, the general functionality of `iota` is available in standard Racket as (`build-list` *num* `identity`).

```
> (define second-derivative (nest polynomial-derivative-coeffs 2))
> (second-derivative (list 1 4 3 -2))
'(6 -12)
```

8. Write a procedure `(polynomial-deriv coeffs n)` that takes a list of polynomial coefficients *coeffs* and a strictly positive integer *n* and produces a procedure that takes a single value and evaluates the *n*th derivative of the polynomial with the given coefficients at that value.

```
> (define d2/dx2-cubic (polynomial-deriv (list 1 4 3 -2) 2))
> (d2/dx2-cubic 5)
-54
```

9. Write a procedure `(non-zero-coefficients coeffs)` that takes a list of coefficients for the terms x^0, x^1, x^2, ... of a polynomial with terms $c_n \cdot x^n$ and produces the values of n (in ascending order) for which $c_n \neq 0$.

```
> (non-zero-coefficients (list 1 4 0 -2))
'(0 1 3)
> (non-zero-coefficients (polynomial-derivative-coeffs (list 1 4 0 -2)))
'(0 2)
```

10. Write a procedure `(argmax vals)` that takes a non-empty list of numerical values and returns the lowest index (position in the list) of the largest value from the list.

```
> (argmax (list 1 4 0 2))
1
> (argmax (iota 5))
4
```

To enable grading, you **must** include the following concluding expression in your file to export your procedures:

```
(provide right-section dot-product iota polynomial-term polynomial
         polynomial-derivative-coeffs nest polynomial-deriv
         non-zero-coefficients argmax)
```

Failure to include this line will result in a zero for the code correctness portion of your grade. To ensure credit, make sure you name your functions and have them take their arguments *exactly* as specified in the assignment.

What to turn in

In addition to `references.txt`, your submission should include the following:

- `polynomial.rkt`, your completed implementation of all procedures above
- `test-polynomial.rkt`, a separate driver program that demonstrates your procedures are correct (you must use examples *other* than those provided)
- A transcript `output.txt` file of your driver program's output

Acknowledgements

Problems 4 and 5 are adapted from S. Rebelsky and J. Weinman, *Exam 3: Sophisticated Scheming*, CSC151 2010S. Problem 7 is adapted from J. Davis and J. Weinman, *Exam 3: Sophisticated Scheming*, CSC151 2011S.