Programming Assignments – 1

Programming Languages Essentials
PUCSD – January 2016-May 2016

1. Warm up to get you into Scheme programming.

   (a) Write a Scheme procedure that returns the sum of two numbers.

   (b) Write a Scheme procedure that returns the product of two numbers.

   (c) Modify your programs in assignments 1a and 1b so that they do their job only for numbers and not for other types of data.

   (d) Write a program that displays the data type of the datum received as its argument. Use the R^{5}RS (See: `http://www.schemers.org/Documents/Standards/R5RS/`) as the reference for the data types that standard Scheme supports.

   (e) Write a Scheme procedure that accepts a natural number N as an argument and returns the sum of the first N natural numbers.

   (f) Write a Scheme procedure that accepts a natural number N as an argument and returns the product of the first N natural numbers.

   (g) Write a Scheme procedure that accepts a natural number N and a binary operator like '+', '$*$', '$\wedge$' or '$\vee$' as an argument and returns the "accumulation" due to that operator of the first N natural numbers. (e.g. The "accumulation" of the '+' operator for N natural numbers is their summation – as in problem 1e!)

   (h) The problem 1g allows us to arbitrarily specify the end point of the accumulation. However it is not general enough. It assumes that the accumulator always starts from 1. There are at least two opportunities to generalise that problem. For example, if we additionally accept the start point of accumulation as a parameter, then we can accumulate over integers too for operators like '+'. Identify all such opportunities for generalisation and write Scheme procedures for each.

2. Recursive definition and recursive execution. Study the following two Scheme procedures. Observe how they are written and imagine how they execute.

```
;; Recursive version
(define (rfact num)
   (if (= num 0)
       1
       (* num (rfact (- num 1)))
   )
)
```

```
;; Iterative version
(define (ifact num)
   (define (iter-fact n count acc)
      (if (= (+ n 1) count)
          1
          (iter-fact n
                     (count + 1)
                     (* acc count))
      )
   )
   (iter-fact num 1 1)
)
```

3. Basic List manipulation exercises

   (a) Draw a box diagram representation of the following:

      i. `(1 2 3)`

      ii. `(1 (2) (3 4) (5 6 (7 (8 9))))`

      iii. `('oses ('foss ('bsd 'linux)) ('mobile ('android 'ios 'symbian)))`

   (b) Write a Scheme procedure that returns true if a given list is a list of natural numbers, or false otherwise.

(c) Write a Scheme procedure that accepts a natural number N and a list as arguments and returns the sum of the first N members of the list. Compare your solution with your solution to problem 1e.

(d) Write a Scheme procedure that accepts a natural number N and a list of numbers as arguments and returns the list of the first N members of the given list. Compare your solution with your solution to problem 3c.

(e) Write a Scheme procedure that returns true if a list has members of the same type, or false otherwise.

4. Procedures and higher order procedures

(a) Write a Scheme procedure that accepts two arguments: a natural number N and a Scheme procedure, say "`gen`", that generates a list of numbers and returns the sum of the first N members of the list generated by "`gen`". Compare your solution with your solution to problem 3c.

(b) Write a Scheme procedure "`my-map`" that takes a procedure of one argument and a list of numbers as its arguments, and returns the list obtained by applying the argument procedure to each element of the argument list. For example: "`(my-map sin (0 (/ pi 2) pi (* 3 (/ pi 2)) (* 2 pi)))`" returns the list "`(0 1 0 -1 0)`".

(c) Write a Scheme procedure "`differentiate`" that accepts a mathematical procedure "`fx`" (e.g. `fx` could be `sin` or "`square`" etc.) and a small value (e.g. 0.00001), and returns the derivative "`f-x`" of that procedure. For example: `(differentiate sin 0.0001)`, or `(differentiate square 0.0000001)`. More useful for problem 4d would be: "`(define sin-x (differentiate sin 0.00001))`" or "`(define square-x (differentiate square 0.0001))`" etc.

(d) The objective of this problem is to test the results of your solution to problem 4c. Use a reasonably long list of numbers and the `my-map` procedure from problem 4b to generate the lists values of "`fx`" and "`f-x`". Plot the elements of these two lists and verify if your differentiation program is correct.

(e) Modify your solution to problem 1h to do the following: Given an operator as required in problem 1h, your Scheme program, called "`general-accumulator`", returns its accumulator procedure. For example: "`(define summation (general-accumulator + ...))`" where the "..." represent the other arguments that you have in your solution to probem 1h.

(f) Currying: Here is a procedure called `compose` that takes in two functions `f` and `g` and composes them: `(define (compose f g) (lambda (x) (f (g x))))`. Thus, for example: "`((compose asin sin) 1)`" returns 1. Rewrite a curried version of this procedure. What could be possible use(s) of the curried version?

(g) Define a version of `compose` that takes either two or three procedures and composes them. The composition is defined as: "`(compose3 f g h) = (compose f (compose g h))`".