CSC104, Assignment 3, Winter 2006
Due: Thursday March 30th, 11:59 pm

Danny Heap

This third (and last) assignment of this offering of CSC104 will use the programming language Scheme. Part of my motivation in using Scheme is that I expect that the structure of this language is unusual enough to put all students (those with considerable computer experience, and those without) on an equal footing. Scheme is widely-used as a first programming language for both Computer Science, and for students in the humanities.

I'll try to provide documentation for Scheme as you need it. We'll be using a relatively friendly Scheme environment called DrScheme (pronounced "doctor scheme"), and if you pursue its help menu you will find a rather dry manual for Scheme, plus links to an online book teaching Scheme called "How to Design Programs."

Finger exercises

1. Create directory *A3*, make it your working directory, and create a file called *journalA3* in it. This will be where your record the joys and frustrations of working through this assignment, plus how you go about solving (or not) the exercises, and observations or investigations that occur to you on the way.

   While still in directory *A3*, start up DrScheme, by typing *drscheme*. Once you have a DrScheme window, you can set a comfortable font under *Edit/preferences*, and set the level of the Scheme language to "Intermediate student with lambda" by going to the *Language/Choose Language* menu, and looking under "How to Design Programs." The idea of selecting a language level is to make enough of Scheme available to allow you to do some interesting things, but to turn off some other features that might get you into trouble. Of course, you are free to set the language level to whatever you decide, so that you can get into as much, or little, trouble as you choose.

   While under the *Language* menu, switch to the *Teachpack* submenu, and select *htdp*. You will see a number of entries with an ".ss" extension, and you should double-click *draw.ss*. This will allow your to use some drawing commands in your programs. Although I'll mention the necessary commands from *draw.ss*, if you're curious you can look under *Help/Help Desk/Teachpacks*.

2. The bottom part of your DrScheme is an interactions pane, where you can experiment with Scheme expressions. Arithmetic uses some familiar operators such as + - * and / together with a couple of probably-unfamiliar features: expressions are wrapped in parentheses, and operators come before the things they operate on (operands). Try typing the following in the DrScheme interactions pane, and record your experience and explanations or conjectures in *journalA3* (if you get a message about the interactions pane not being synchronized with the definitions pane, click the *run* button and continue):

```
> (+ 2 3)
> (+ 1 2 3 4)
> (* 2 3)
> (* 2 3 4)
> (/ 3 5)
> (* (/ 3 5) (/ 5 3))
```

3. As well as numbers, Scheme has lists. Various types of objects (including lists themselves) can be members of lists, but numbers are probably the most concrete sort of object to begin with. Here's how you can create a list of the first five positive integers:

   ```
   (list 1 2 3 4 5)
   ```

   This is exactly the same as:

   ```
   '(1 2 3 4 5)
   ```

   Scheme provides some tools for manipulating lists. Try out, and then explain, the following:

   ```
   (first (list 1 2 3 4 5))
   (second (list 1 2 3 4 5))
   (eighth (list 1 2 3 4 5))
   (rest (list 1 2 3 4 5))
   (rest (list 6 7 8 9 10))
   (reverse (list 1 2 3 4 5))
   (reverse (reverse (list 1 2 3 4 5)))
   (length (list 1 2 3 4 5))
   (append (list 1 2 3 4) (list 5 6 7 8))
   ```

4. Scheme can tell *true* from *false*, combine them with *and* and *or*. Try out the following, and record your observations and explanations:

   ```
   > (> 3 5)
   > (< 3 5)
   > (positive? 5)
   > (positive? -3)
   > (and (< 3 5) (positive? -3))
   > (or (< 3 5) (positive? -3))
   ```

   Scheme can choose different actions, based on whether some expression is *true* or *false*, using the special keywords *if* and *cond*. Experiment with the following until you can explain it (note that strings are denoted similarly to what you've seen in Python):

   ```
   > (if (> 5 3) "five is more than three" "five is not more than three")
   > (if (< 5 3) "five is less than three" "three is not less than three")
   > (cond ((< 5 3) "five is less than three")
           ((> 5 3) "five is greater than three")
           (else "five and three are equal"))
   ```

   The special keyword *define* binds the first expression that follows it to the second expression that follows it, so you can use the first expression as a name to call the second. Try out the following example, and chat with a TA or an instructor until you can record some sort of explanation:

```
> (define (maximum n1 n2)
    (if (> n1 n2) n1
        n2))
> (maximum 3 5)
> (maximum 5 3)
```

5. The semicolon tells Scheme to ignore the remainder of a line it is on, and is thus useful for comments. In the next few exercises I will provide comments that describe the function that you are to try to create. Included in the comments are examples of how the function behaves.

   Your first task is to define the function *lcycle*, corresponding to the passage beginning with semicolons below. In your definitions pane, type a Scheme expression that defines *lcycle*. HINT: you will need to dream up a name for the parameter that refers to your list, just as I had to dream up n1 and n2 in the last exercise. You can't use the parameter name *list*, because Scheme already uses that for other things.

   You should read the comments very closely, plus refer back to the last few exercises, to guide you in writing *lcycle*. The comment on the first line indicates that *lcycle* takes a list as input, and produces a list as output. You should type your definition in the top (definition) pane (include the comments), and then test it by clicking the *run* button, and trying the new command out in the bottom (interactions) pane of DrScheme. Record your observations and explanations in *journalA3*. When you believe you have it working, save your file as *sneeze.scm* by using the *File/Save Definitions As* menu.

   ```
   ;; lcycle : list -> list
   ;; To produce a version of the list with the first element moved to the end.
   ;; example 1: (lcycle '(1 2)) produces '(2 1)
   ;; example 2: (lcycle '(1)) produces '(1)
   ;; example 3: (lcycle '(1 2 3)) produces '(2 3 1)
   ```

6. Keep working in the definitions pane (and saving your work occasionally by clicking the *Save* button). Don't erase or replace your definition of *lcycle*. There is another passage of comments below describing a similar function *rcycle*. You could create *rcycle* from scratch, or perhaps think of how to create it using *lcycle* and *reverse*. Record your observations and explanations in *journalA3*.

   ```
   ;; rcycle : list -> list
   ;; To produce a version of the list with the last element moved to the beginning.
   ;; example 1: (rcycle '(1)) produces '(1)
   ;; example 2: (rcycle '(1 2)) produces '(2 1)
   ;; example 3: (rcycle '(1 2 3)) produces '(3 1 2)
   ```

7. The functions you're working on are components of a program that will simulate and display the progress of a respiratory infection through a population. We use a positive number to indicate an individual in our population who is infected. Your next job is to define *sick-left* to decide whether the left-most number in a triple (list of three) is positive. Keep working in the definitions pane where your previous two definitions are, test your work by clicking *Run* and then trying things out in the interactions pane, and save your work by clicking the *Save* button. As always, observations, explanations, and conjectures go in *journalA3*

   ```
   ;; sick-left : list -> Whether left-most number of list is positive.
   ;; To decide whether the left-most number of list is positive.
   ```

```
;; example 1: (sick-left '(5 3 1)) produces true.
;; example 2: (sick-left '(-5 3 1)) produces false.
```

8. Now you should define the similar function *sick-right*. Once again, you could define this from scratch, or you could use *sick-left* and *reverse*. Record your observations and explanations. Save your definitions by clicking the *Save* button, and record the process of solving the exercise in *journalA3*.

```
;; sick-right : list -> Whether the right-most number of list is positive.
;; To decide whether the right-most number of list is positive.
;; example 1: (sick-right '(1 3 5)) produces true.
;; example 2: (sick-right '(1 -3 -5)) produces false.
```

9. Before attempting the last (tenth) exercise, you need to understand a subtle, useful, and beautiful technique called RECURSION. I will discuss this in class, but here is an example you should experiment with until you can record an explanation.

   In the open definitions pane, type the definition of *GCD* below, followed by the given example. Then click the *Step* button, and then (again) the *Step>* button on the *Stepper* window that pops up. You can step backwards and forwards through the recursive evaluation of *(GCD 5 35)*. The strange identifier *lambda* is a placeholder for our function, GCD, at each step.

   It may help your thinking to accept (without proof) the following mathematical facts: Fact 1: The GCD of two positive integers $n1$ and $n2$ is the same as the GCD of $n1$ and the remainder of $n2$ after dividing by $n1$. Fact 2: The GCD of any non-negative integer and zero is the given non-negative integer.

```
;; GCD : number number -> number
;; To find the Greatest Common Denominator (GCD) of two non-negative whole numbers.
;; example 1: (GCD 5 7) produces 1.
;; example 2: (GCD 15 35) produces 5.
(define (GCD n1 n2)
  (if (zero? n2) n1
      (GCD n2 (remainder n1 n2))))

(GCD 15 35)
```

10. Please feel free to ask for help/hints from the TAs and instructors on this exercise, creating a definition for the function *make-list*. Keep working in the same definitions pane as your previous definitions, saving your work occasionally by clicking the *Save* button. Observations, conjectures, and explanations go in *journalA3*.

    The idea is to create a function that produces lists with n copies of some object. The natural way to carry out repetetive tasks in Scheme is with recursion, the technique from the previous exercise. The key idea in writing this short definition is in the following paragraph:

    If n is less than 1, produce an empty list. Otherwise append a list containing a single copy of the object to a list of $n - 1$ copies of the object.

    You might want to re-read the above two sentences a few times as you work on writing the definition below. I believe it almost spells out the code you have to write to create the function, but it takes a lot of staring to get used to recursion.

```
;; make-list : number object -> list
;; To produce a list consisting of number copies of object.
;; example 0: (make-list 0 5) produces '()
;; example 1: (make-list 3 5) produces '(5 5 5)
```

Sneeze and sensibility

Keep DrScheme open to the definitions pane *sneeze.scm*, which has the definitions you created, as you read the next few paragraphs.

Every winter respiratory infections sweep through our population, carried on a sneeze. Depending on how strong your resistance is, your neighbour's sneeze today will be your sneeze tomorrow.

I have built a simplified model of a sneeze epidemic. The population is represented as a list of zeros (representing healthy individuals) with a single 1 (an infected individual). As each day passes, a healthy individual with an infected individual neighbouring them (to the right or left) will become infected if their resistance is less than the virulence of the infection. Once infected, they remain sick (and infectious) for a number of days (which I call duration), at which point they recover and become immune for a number of days (which I call immunity).

Depending on the value of *virulence* (a number between 0 and 100), *immunity*, and *duration*, the course of the epidemic will be different. For each individual, *resistance* is randomly chosen, each day, to be a number between 0 and 100. If a person has a sick neighbour, and their *resistance* is less than virulence, they become sick.

Open a separate terminal where you can copy `~heap/pub/epidemic.scm` to your current (A3) directory. In DrScheme, open *epidemic.scm* using the *File/New* menu. You will now have one definitions pane for the definitions you created, and saved in *sneeze.scm*, and another definitions pane with *epidemic.scm*. You will need to set the language level to *Advanced Student* in the *Language/How to Design Programs* menu, in order for some of the commands I have put in *epidemic.scm* to be recognized.

In *epidemic.scm*, hunt for the comments (the passages beginning with semicolons) that correspond to each of the definitions you created. Cut-and-paste (or carefully retype) your definitions immediately below those passages — no semicolons in front of your definitions, please!. When you're done, click *Run*.

If you complete the Scheme code correctly, you will be prompted for values of *virulence*, *duration*, *immunity*, and *days*. Try starting with 10, 15, 5, and 100, respectively. If you're successful, a two-dimensional simulation of the sneeze epidemic runs each time you click the *Run* button. You will get different simulations if you change the values for *virulence*, *duration*, *immunity*, or *days*. Record your observations and conjectures in *journalA3*.

What to submit

Please submit the following files:

- *journalA3*
- *sneeze.scm*