CSC104, Assignment 3, Summer 2006
Due: Thursday July 20\textsuperscript{th}, 11:59 pm
Anastasia Bezerianos, Danny Heap
July 13, 2006

Finger Exercises

This third (and last) assignment of CSC104 will use the programming language Scheme. This assignment is more challenging than the first two and you should make sure you ask TAs and the professor a lot of questions. Part of the motivation in using Scheme is 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.

We'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."

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 dscheme. 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 btdp. You will see a number of entries with an ".ss" extension, and you should double-click image.ss. This will allow you to use some drawing commands in your programs. Although I’ll mention the necessary commands from image.ss, if you’re curious you can look under Help/Help Desk/Teachpacks.

Once you’ve selected your language level and teachpack, you may click the “run” button, and you should see your language level and teachpack listed in the bottom (interaction) pane.
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))
   > (expt 3 2)

3. Scheme allows us to represent types other than numbers. Try for example:

   > "smart"
   > (circle 4 'solid 'red)
   > (triangle 5 'solid 'green)
   > (rectangle 20 20 'outline 'blue)

Notice how strings are denoted similarly to what you’ve seen in Python. Also note that in Scheme, everything inside parenthesis is what we call an "expression", something that Scheme computes. The first word in the expression is the name of an operator or function, similarly to what you saw in Ex.2 with numbers. So Scheme recognizes (circle 4 'solid 'red) as something to compute using the operator "circle". The remaining information are the operands or parameters of the operator circle. In the given example Scheme knows that the operator circle asks for a circle to be drawn. The numbers 4, 5 and 20 in the examples are units denoting the size of the shapes.

4. We can also assign labels to different elements using define. The operator define binds the first parameter given (something not yet existing) to the second (an existing quantity). So define binds the first expression that follows it to the second expression that follows it and you can use the first expression as a name to call the second. Notice how define is inside parentheses. Try:

   > (define num1 3)
   > num1

   > (define num2 5)
   > num2

   > (+ num1 num2)

   > (define shape1 (circle 4 'solid 'red))
   > shape1
5. As well as numbers, shapes and other primitives, 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)

Lists can also have shapes or any other type of Scheme objects:

> (list (circle 5 'solid 'red) (rectangle 10 10 'solid 'blue) (triangle 10 'solid 'green))

You can also use define to label lists, as in Ex.4. Try:

> (define num-list (list 1 2 3 4 5))
> num-list

> (define shape-list (list (circle 5 'solid 'red)
(rectangle 10 10 'solid 'blue)
(triangle 10 'solid 'green) (circle 5 'outline 'red)
(rectangle 10 10 'outline 'blue)
(triangle 10 'outline 'green)))
> shape-list

6. Scheme provides some tools for manipulating lists. Try out, and then explain, the following, using the lists you have already defined in Ex.5:

> (first (list "double" "double" "toil" "and" "trouble"))
> (first num-list)
> (second shape-list)
> (eighth num-list)
> (rest (list "double" "double" "toil" "and" "trouble"))
> (rest num-list)
> (reverse shape-list)
> (reverse (reverse num-list))
> (length shape-list)
> (append shape-list num-list)
> (append shape-list num-list "double")
> (append shape-list num-list (list "double"))

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

> (> 4 7)
> (< 4 7)
> (positive? -4)
> (empty? (list 1 2))
> (equal? (circle 5 'solid 'red) (circle 5 'solid 'red))
> (and (< 4 7) (positive? -4))
> (or (< 4 7) (positive? -4))
Scheme can choose different actions, based on whether some expression is true or false, using the special keywords \texttt{if} and \texttt{cond}. Experiment with the following until you can explain it:

\begin{verbatim}
> (if (> 7 4) "seven is more than four" "seven is not more than four")
> (if (< 7 4) "seven is less than four" "seven is not less than four")
> (cond ((< 7 4) "seven is less than four")
      ((> 7 4) "seven is greater than four")
      (else "seven and four are equal"))
\end{verbatim}

8. As we have mentioned the special keyword \texttt{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:

\begin{verbatim}
> (define (max3 n1 n2 n3)
  (cond ((and (> n1 n2) (> n1 n3)) n1)
        ((and (> n2 n1) (> n2 n3)) n2)
        (else n3)))
> (max3 1 10 7)
> (max3 7 1 10)
\end{verbatim}

9. 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 we 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 \texttt{solidT}, corresponding to the passage beginning with semicolons below. In your definitions pane, type a Scheme expression that defines \texttt{solidT}. Hint: you will need to dream up names for the parameter that refers to your number and color, just as I had to dream up \texttt{n1}, \texttt{n2} and \texttt{n3} in the last exercise.

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

\begin{verbatim}
;; solidT : number color -> object
;; Create an equilateral triangle with base "number" units and solid color.
;; Number is an integer greater than 2.
;; example 1: (solidT 10 'green) produces a solid green triangle with base 10 units wide.
;; example 2: (solidT 20 'red) produces a solid red triangle with base 20 units wide.
\end{verbatim}

10. Keep working in the definitions pane (and saving your work occasionally by clicking the Save button). Don’t erase or replace your definition of \texttt{solidT}. There is another passage of comments below describing another function \texttt{square}. Record your observations and explanations in \texttt{journalA3}.
11. After drawing some shapes, it is time to do some list manipulation. Another section of comments follows for a new function last. Keep working in the definitions pane (and saving your work occasionally by clicking the Save button). You might need to refer back to Ex. 8 and use some combination of functions there to come up with your solution.

;; last : list -> element
;; Gets the last element of a list.
;; example 1: (last (list 1 2 3 4 5)) produces 5.
;; example 2: (last (list (circle 5 'solid 'red) (square 10 'solid 'blue)))
;; produces a blue rectangle.

12. A bit more list manipulation. Keep working in the definitions pane (and saving your work occasionally by clicking the Save button). This time we will define removeLast. This function returns a list minus it's last element (similarly to how rest returns the list minus its first element).

;; removeLast : list -> list
;; Removes the last element of a list.
;; example 1: (removeLast (list 1 2 3 4 5)) produces (list 1 2 3 4).
;; example 2: (removeLast (list (circle 5 'solid 'red) (square 10 'solid 'blue)))
;; produces a list with a red circle.

13. Before attempting the following exercises, you need to understand a subtle, useful, and beautiful technique called RECURSION. We 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 recursive-circle 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 (recursive-circle 4). The strange identifier lambda is a placeholder for our function, recursive-circle, at each step. You can maximize the trace window to be able to see more information. The function overlay below overlays two shapes so that they are concentric.
14. After you have understood the recursive-circle function, try to extend it so that it can draw any type of concentric circle (not only circles) of any color. This function will be very similar to the recursive-circle function seen before. Keep working in the definitions pane (and saving your work occasionally by clicking the Save button). Don’t erase or replace your previous definitions of solidT, square. There is another passage of comments below describing the recursive-shape function. Record your observations and explanations in journalA3. Make sure you ask the TAs and the instructor questions. Recursion may result in few lines of code, but it is hard to understand.

15. Again save your work occasionally by clicking the Save button and continue to work in your definitions pane. Another passage of comments below describes the function sum. Record your observations and explanations in journalA3. This is another recursive function and you should feel free to ask for help/hints from the TAs and instructors.

The idea here is that when we are given two numbers n1 and n2, we compute the sum of all the numbers between them (for example (sum 1 5) is 1+2+3+4+5). The natural way to carry out repetitive tasks in Scheme is with recursion, the technique from the previous exercises. The key idea in writing this definition is in the following paragraph:

(a) If the two numbers n1 and n2 are the same, produce the number n1 or n2 (no numbers in between to sum).

(b) Otherwise do something a bit wacky, add the first number n1 to the sum of all numbers that are greater than n1 until n2 (the sum of all numbers between n1+1 and n2).

You might want to re-read the above two sentences a few times as you work on writing the definition below. It takes a lot of staring to get used to recursion.
16. Keep working in the definitions pane (and saving your work occasionally by clicking the Save button). Don’t erase or replace your previous definitions. There is another passage of comments below describing another recursive function increase. Record your observations and explanations in journalA3. Again feel free to ask for help/hints from the TAs and instructors on this exercise.

The idea is to create a function that takes a list of numbers and produces a list where all these numbers have been increased by 1. This is another repetitive task that can be written in Scheme with recursion. The key idea in writing this definition is:

(a) If the list is empty, produce an empty list (nothing to increase).
(b) Otherwise append a list containing the first element of the given list increased by one, to a list which includes the remaining elements of the first list all increased by one.

You might want to re-read the above two sentences a few times as you work on writing the definition below. Make sure you remember how first and rest behave in Ex.6.

;; increase : list -> list
;; Increases the elements of list by one.
;; List is a list of numbers.
;; example 1: (increase (list 1 2 3 4 5)) produces (list 2 3 4 5 6).
;; example 2: (increase (list 0 -3 3)) produces (list 1 -2 4).

Note that you cannot use the parameter name "list", because Scheme already uses that for other things.

PALINDROME?

Keep working in the definitions pane. In this exercise we’ll create a function that checks if a list is a palindrome (the same when read from left or right). For example (list ”a” ”b” ”b” ”a”) is a palindrome, but (list 1 2 3) is not.

Now to define palindrome?. Here we will revisit recursion. The palindrome? function is defined as follows:

1. If the length of the list is 0 then palindrome? should return true.
2. If the length of the list is 1 then palindrome? should return true.
3. If the first and last elements of the list are equal, and the list between the first and last element is a palindrome, palindrome? should return true.
4. palindrome? should return false in all other cases.

In Scheme the words true and false have a special meaning and can be used directly in programs.

Look back in Ex.6, as well as your last and removeLast definitions to remember how to manipulate lists. If you don’t remember how to check for multiple conditions, go over Ex.7 again. Don’t forget to Save your definitions.

;; palindrome? : list -> true or false
;; Decides if a list is a palindrome.
;; example 1: (palindrome? (list 1 2 3 4 5)) produces false.
;; example 2: (palindrome? (list (circle 5 ‘solid ‘red))
After completing your definition you can check if a picture is a palindrome. Do the following:

> (circle 3 'solid 'blue)
> (image->color-list (circle 3 'solid 'blue))
> (palindrome? (image->color-list (circle 3 'solid 'blue)))

Here we created a circle and then created a list of the colors of our image, using the function `image->color-list`. Experiment with other images as well. Report your observations in `journalA3`.

**Sierpinski Gasket**

We have seen so far that recursion can be a wonderful tool to help us deal with repetitive actions. It is also seen often in nature (some plant growth patterns for example follow recursive functions). And last, but not least it can create interesting artistic effects. In this section we will create a fractal drawing called Sierpinski Gasket.

The Sierpinski Gasket is named for the Polish mathematician who first proposed it. It is a fractal image that is made from equilateral triangles. Here is a description of how it can be constructed: We start with an equilateral triangle, and replace it by three equilateral triangles, each with a base half the size of the original, stacked so the original perimeter is kept, but leaving an upside-down triangular hole in the center. We then replace each of those triangles by three more triangles, to obtain nine triangles, each with a base of one-quarter the length of the original. Continuing in this fashion infinitely many times yields the Sierpinski gasket. If you want to see an image of what it looks like you can go to http://en.wikipedia.org/wiki/Sierpinski_gasket.

In this exercise we will be using `solidT` that we defined before. This will help us draw solid triangles. In your definitions pane add the following code:

```scheme
;; trefoil : image image image number -> image
;; Place images centres in an equilateral triangle with base number units long.
;; (trefoil (solidT 10 'green) (solidT 10 'red) (solidT 10 'blue) 10)
;; produces a green, red, and blue triangle arranged in a triangle.
(define (trefoil img1 img2 img3 offset)
  (overlay/xy
   (overlay/xy img1 offset 0 img2)
   (* (/ (sqrt 3) -2) offset) img3))
```

The `overlay/xy` function is similar to the `overlay` function seen in Ex.13 , but now the two images are not concentric. The second image is offset from the first one on the X and Y axis. The `sqrt` function gives the square root of a number.

The function you need to define is called `sierpinski` and can be described as follows:

The function is given 2 parameters, the number of steps and the base of the Sierpinski Gasket (the length of the smallest triangles making up the gasket).

1. If the number of steps is equal to 0, then draw an equilateral triangle made up of images of three solid triangles (one green, one red and one blue) of size equal to the base. The offset between these three images is also equal to base.
2. If the number of steps is greater than 0, do something unexpected. Make up an equilateral triangle
made up of three smaller copies of the same image: each image is a Sierpinski triangle with steps equal
to one less than your current step and of base equal to the current base. The offset is the product of
the current base times 2 to the power of your current steps (look back to Ex.2 to remember how to
raise a number to the power of another).

The above is hard to grasp in one go. Read it through carefully, and perhaps make yourself a few sketches,
until you feel you can proceed with the code of the function. You might need to read your finger exercises
and the work you have done so far. Feel free to ask a TA or the instructor about the program if you get
stuck.

;; sierpinski : number number -> image
;; Produce a Sierpinski gasket with the given number of steps and base of smallest triangles.
;; (sierpinski 0 10) produces a trefoil with a green, red, and blue triangle of base 10.
;; (sierpinski 1 10) produces a trefoil with the trefoils from the previous line.

WHAT TO HAND IN

Under Assignments on the course web page you will find a link to the CDP submit facility. Submit the
following files:

* journalA3
* scheme-stuff.scm

You should submit your files early and often. The first time you create a file with meaningful content,
send it. You may re-submit the same file as many times as you wish, and only the last submission is
stored. A good habit is to re-submit your files each time you improve them.