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 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 httpd. 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.

Solution: I have done this before, so I go over my notes from A1 and create a new directory A3 and started journalA3 similarly to the first two assignments. I opened up Drscheme, and although I was able to choose the Intermediate Student with Lambda level, I couldn't seem to get the image.ss teachpack installed at first. The help listed on the teachpacks seemed highly uninformative.
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)

**Solution:** Scheme appears to perform the usual arithmetic operations in an unusual way: the operator comes before the things it operates on. This seems pretty awkward, except in the case of (+ 1 2 3 4) it seems to save keystrokes (that adds up all the numbers that follow the "+" sign). The general rule seems to be: (operator number1 number2) means number1 operator number2. The order is important with subtraction and division: (/ 3 5) means 3 divided by 5, not 5 divided by 3. The parentheses tell me the scope of things to operate on, and I can embed subexpressions, for example: (* (/ 3 5) (/ 5 3)) means to first evaluate the inner expressions (so 3/5 and 5/3), and then multiply them (producing 1). The division operator seems to produce an exact floating point result. For example, (/ 5 3) produces a repeating decimal, and indicates which portion repeats. The expt operator seems to raise the first number to the power of the second (I had to try a few examples until I figured this one out).

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.

**Solution:** This is soooo neat. Apart from getting the string "smart" (I know I am, but what are you?), Scheme can also draw some pretty shapes. For example the second command gave me a circle, which as I expected was red. At the beginning I missed the quote before the word "red" and Scheme said I was using the identifier "red" before it's definition. It seems 'red and 'solid are recognized somehow by Scheme as being the color and how to fill in the color in the shapes drawn. The number 4 seems to define the radius of the circle (it is not the diameter, because I tried the same size for triangles and they are way smaller).
Similarly, triangle draws a triangle. The number next to it defines the size of the base of the triangle and then it tells it to fill it with the color green. The last command gave me a large, hollow, blue square. The 'outline was what made this shape hollow, 'rectangle defined the shape, and the 'blue, well it made it blue. Interestingly, the rectangle was followed by two numbers, whereas the circle by 1. I have a suspicion that if I change one of the two numbers it will no longer be a square. Correct! With (rectangle 20 40 'outline 'blue) I get a nice, tall rectangle. So the numbers after the word "rectangle" define its sides, as the number after "circle" defines its radius. This also explains the difference in size between the different shapes.

I got a bit confused about all these "expression" stuff. I'll take the TA's word that what is inside the parenthesis is an expression and that "circle" is an operator. I guess this operator needs 3 parameters, a radius, to determine if the circle is solid or just an outline, and its color.

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

SOLUTION: So this define word seems to be an operator of some kind. Its operation is to assign a label to a value. So num1 is now a label for number 3 and num2 that of number 5. Not surprisingly (+ num1 num2) gives 8. The interesting thing is the "define" can be used to assign labels to things other than numbers. For example after the third definition, when I type shape1, it is equivalent to asking Scheme to draw for me a solid red circle (where circle is another operator, that I bet someone else has defined). Just to make sure that define works with data types, other than numbers and shapes, I tried:

> (define word1 "hello")
> word1
"hello"
>
So define works for all data types that I know of in Scheme.

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))

**SOLUTION:** It seems that the word "list" is an operator that puts everything after it in a list. So with the above examples I get a list of 5 numbers and then a list of 3 shapes. I am pretty curious to see if I can combine stuff in the list (the exercise says that I can have lists in lists). I try:

> (list (circle 5 'solid 'red) (list 1 2 3) "hello")

and I get a nice combination of a circle, a list of numbers and a word, all inside a list.

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

**SOLUTION:** This is fun and useful too. By using `define` I have given an alias (num-list) to a list of 5 numbers and another (shape-list) to a list of 6 shapes. I can now call these lists by their respective alias name and I do not need to re-type them all the time. I must have made some mistake the first time I tried to define shape-list and forgot one of the shapes. When I tried it again I got the error that shape-list was already defined. After a bit of fussing around I found out that by pressing Run all the definitions get cleared and I can start again.

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"))

**SOLUTION:** This exercise was a lot of fun! The first operator returns the first element in a list of words that I defined on the spot, the word "double". Similarly in the second example I get the first item of the num-list that I defined before, the number 1. It seems that the operator second returns the second element of my shape-list, the solid square. When I try the eighth I get the error that eighth: expects argument of type list with 8 or more items. This makes sense since num-list has
only 5 elements. I tried the operators third, fourth, ... eighth and it seems they all exist, but no ninth or more. I guess nobody thought of using define to give "ninth" a special meaning. The rest operator returns a list with the first element removed (the rest of the list, I guess). reverse does what it says: it reverses a list, and if you reverse a list twice, you get the original list back. length gives you the length of a list, and append pastes two or more lists together (no matter what is inside the lists). It seems that append only works on lists, that us why appending the word "double" does not work, but that the (list "double") does. I tried out (length (list)), and got "0", so that list must be empty. I also found that (append () (list 1 2 3)) gives me (list 1 2 3), so you leave a list unchanged by appending an empty list.

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))

SOLUTION: The "operator before the things it operates on" pattern continues with Scheme. If I want to test whether 4 > 7, I put the > at the beginning of my Scheme expression, and I get false for my trouble (which is, so to speak, true). I can also test whether 4 is less than 7 by using the < operator instead. Scheme recognizes positive? as a question, and gives a true/false answer, depending on whether the number following it is positive or not. Similarly empty? checks if a list is empty and equal? checks if two things are equal. Apart from shapes, I tried empty? with numbers and words and it works there as well.

The ordinary words and and or have special meaning to Scheme. If I have and followed by a couple of things, one of which is false, I get false. I tried following it with two true things, and it produces true. Two false things produce false again. Some the and of two things is true exactly when they both are true. I tried a similar set of tests on the special word or, and it seems as though the or of two things is true unless they are both false.

By the way, the "things" in the previous paragraph have to be expressions that can only be true or false. I tried plugging in numbers, and Scheme complained.

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:

> (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"))

SOLUTION: The general pattern here seems to be (if (true-or-false-thing) do-this do-that). When the true-or-false-thing is true, we do-this, otherwise we do-that. So far as Scheme is concerned, strings such as "seven is more than four" are just objects to produce, depending on whether the
true-or-false-thing is true or false. I tried writing nonsense for both strings, and Scheme would produce the first one when the true-or-false-things is true, and the second one otherwise.

The next pattern seems to be `(cond ((and (> n2 n1) (> n2 n3)) n1) ((and (> n2 n1) (> n2 n3)) n2) (else n3))`. I tried this out, and it just returns the do# corresponding to the first thing that's true.

8. As we have mentioned 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 (max3 n1 n2 n3)
  (cond ((and (> n2 n1) (> n2 n3)) n1)
       ((and (> n2 n1) (> n2 n3)) n2)
       (else n3))
> (max3 1 10 7)
> (max3 7 1 10)

SOLUTION: It seemed odd to be defining something as obvious as maximum, so I tried `(maximum 1 10 7)` and `(max 1 10 7)`, and both gave me the correct answer 10. But max3 is not defined and I get an error when I try it.

I tried typing the given example, and suddenly max3 is defined: `(max3 1 10 7)` produces 10, as does `(max3 1 10 7)`, so the definition I created actually achieved something. I tried to trace out how it works. When a user types `(max3 1 10 7)`, the definition substitutes 1 for n1, 10 for n2 and 7 for n3, and then checks whether n1 (or 1) is both greater than n2 (or 10) and n3 (or 7). This is not true, so it next checks if n2 (10) is greater than both n1 (1) and n3 (7). This is true so n2 (10) is produced.

Doing the other example, if the user types `(max3 7 1 10)`, the definition substitutes 7 for n1, 1 for n2 and 1 for n3, then checks whether n1 is greater than n2 and n3 (false), and then if n2 is greater than both n1 and n3 (again false), so it produces n3 (10). This worked so well that I tried `(max3 1 2 3 4)`, but Scheme scolded me that I had provided 4 arguments instead of 3 (and I wasn't feeling particularly argumentative). It looks as though I can break things down and trace through why they work.

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 `solidT`, corresponding to the passage beginning with semicolons below. In your definitions pane, type a Scheme expression that defines `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 n1, n2 and 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 `solidT`. The comment on the first line indicates that `solidT` takes as input a number and a color, and produces a triangle 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 `journal1A3`. When you believe you have it working, save your file as `scheme-stuff.scm` by using the File/Save Definitions As menu.
;; 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.

SOLUTION: I began by imitating the last exercise: (define (solidT n1) ...). I actually tried running this,
(solidT 10 'red), but Scheme complained that ... is an undefined identifier. As far as I understand
I need to use the (triangle ...) operation somehow. I know from previous exercises that I can draw
a solid green triangle with base 10 for example by typing: (triangle 10 'solid 'green). In solidT
the number 10 and 'green need to be given when I run solidT. Following the max3 example I
assume these two things need to be the parameters of the function solidT. I also know that the
triangle needs to always be solid.

(define (solidT num col) (triangle num 'solid col))

I tried out the examples to make sure my drawn triangles are correct. I was a bit puzzled as to
whether I had to deal with cases of num smaller than 2, but the TA told me that we assume that
the input is always greater than 2. Nevertheless I tried out a case where num was negative and I
got a message saying num needs to be greater than 2.

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 solidT. There is another passage of comments below describing
another function square. Record your observations and explanations in journalA3.

;; square : number fill color -> object
;; Create a square with base "number" units, of color "color",
;; either solid or outline, based on fill.
;; Number is an integer greater than 2.
;; example 1: (square 10 'solid 'green) produces a solid green square with base 10 units wide.
;; example 2: (square 20 'outline 'red) produces an outline of a red square with base 20
;; units wide.

SOLUTION: I understood that I needed to define a new function, as in the previous exercise, and I had
to be able to draw a square. Although I haven’t seen a function for squares, I have encountered
a similar type of shape, a rectangle. A square is a rectangle with sizes that are equal. Based on
how I wrote solidT I started writing up square.

(define (square num fill col) (rectangle num num fill col))

I tested out my new square function and I am very happy with it. The only reason behind creating
this function, as far I understand, is to save us the trouble of writing the same number twice when
drawing a square using the rectangle function.

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.
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.

   (define (removeLast myList) (reverse (rest (reverse myList))))

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.

   ;; recursive-circle : level -> objects
   ;; Draws level number of concentric blue circles, getting smaller in size.
   ;; Number level is greater than 0.
   ;; example 1: (recursive-circle 3) produces 3 concentric circles.
   ;; example 2: (recursive-circle 1) produces 1 circle.
   (define (recursive-circle level)
      (if (equal? level 1) (circle 10 'outline 'blue)
         (overlay (circle (* level 10) 'outline 'blue) (recursive-circle(- level 1]))))

   (recursive-circle 4)
SOLUTION: Once I got the definition typed, and started up the debugger, I felt a bit overwhelmed by information. I eventually got used to the pattern: the code on the left-hand side is highlighted in green, and it becomes (after Scheme gets through with it), the code highlighted in purple on the right-hand side. The lambda keyword is strange, but I mentally substituted "recursive-circle", and it simply repeated the definition I had typed.

To begin with, level was replaced by 4 in the definition. Then (= level 1) was replaced by false (since 4 is not 1), and in the next step we were on the second option of the if statement. This called "overlay" with two parameters: one was a circle to be drawn (of radius 4^level) and the other was (recursive-circle 9). In this step we drew our circle of level 4. There doesn’t seem to be much progress being made here. However, after stepping a bit more, we have an expression to evaluate (recursive-circle 3), so the number level appears to be getting smaller and approaching 1. Stepping some more yields more circles until I reach (recursive-circle 1), and that’s the one my definition knows how to handle, it returns a final circle of radius 10. Then all the drawn centers are placed concentrically.

14. After you have understood the recursive-circle function, try to extend it so that it can draw any type of concentric shape (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.

;; recursive-shape : level shape color-> objects
;; Draws level number of concentric shapes, getting smaller in size, of a particular color.
;; The shape must take one parameter as its size (for example square, circle and triangle).
;; example 1: (recursive-shape 3 triangle 'green) produces 3 concentric green triangles.
;; example 2: (recursive-shape 5 square 'red) produces 5 concentric red squares.

SOLUTION: This problem I understand. I remember from Ex.9 and Ex.10 how to assign parameters that have to do with shape drawing. Recursion is hard to understand, but I believe I have understood enough to try and extend the previous example.

(define (recursive-shape level shape color)
  (if (equal? level 1) (shape (* level 10) 'outline color)
    (overlay (shape (* level 10) 'outline color) (recursive-shape (- level 1) shape color))))

At first I got a bit confused and in the second part of the if statement. I forgot to add the shape and color parameters in (recursive-shape) and Scheme immediately complained about the number of parameters. I fixed it and now I can draw nice concentric shapes. Triangles look weird though, not sure why, but their centers seem to not align nicely. I also forgot to close one of the parenthesis, but was scolded immediately by Scheme.

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.

;; sum : number number -> number
;; Sums up all the numbers between the first and second number.
;; First number is smaller or equal to the second number.
;; example 1: (sum 1 3) produces 6 (1+2+3).
;; example 2: (sum 4 7) produces 22 (4+5+6+7).

SOLUTION: This was harder than the previous exercise, I guess because it was the first real recursion
function that I wrote on my own. I started out with the definition of sum with parameters n₁
and n₂. From the above few exercises I know I will use an "if" and the simplest condition is
(equal? n₁ n₂), where I return n₁. Now the second part of the "if " was tricky. One thing is
for sure, I needed to add n₁ to something, so I write (+ n₁ ...). The something is a bit cryptic:
"sum of all numbers that are greater than n₁ until n₂". If this recursion has to do anything with
this exercise, I need to use sum again. Well since sum is supposed to give me the sum of all the
numbers between n₁ and n₂, I guess the sum of all the numbers that are greater than n₁ until
n₂, is the sum of n₁+1 and n₂.

(define (sum n₁ n₂)
  (if (equal? n₁ n₂) n₂
      (+ n₁ (sum (+ n₁ 1) n₂))))

Now that I look at this exercise, it seems it is very similar to the factorial exercise we did in class.
The only difference is that we do addition instead of multiplication, and we do not start adding
(multiplying) from 1, but from a given number n₁.

I tried some other numbers, and I traced the program to understand it better. If the two numbers
are equal, the sum is the number. Otherwise, we take the first number and add to it the sum of
all the numbers that are bigger than it, up to the second number. To do this we call the
"sum" function with parameters one number larger than our previous first number, and the
second number. This process of repeatedly increasing the first number will reach the second one
eventually, so a result will be produced. After this we need to add up all the first numbers we
stored at the different steps of the recursion.

I was having a lot of fun with this and I even tried a sum that included negative numbers (sum
-2 3). I got 1 and was a bit worried, until I remembered that negative numbers reduce a sum.

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.

**Solution:** At the beginning I got really scared with this exercise. I imitated the previous exercise, since I know what to do for the shortest possible output, an empty list: `(define (increase myList) (if (empty? myList) '()))`. Again, I'll need to be a bit more concrete than the ... I need to create a list of the remaining elements all increased by 1, that are a smaller list (solving a smaller problem...). Okay, suppose I could create such a list of the increased remaining elements, then I could simply add a list with the first element increased, so I replace ... with `(append (+1 (first myList))) ...`. I remember that append takes lists, so I add the keyword "list" before my addition. Now I'm stuck on how to increase the rest of the list and it doesn't seem as though I've made any progress. I ask the Prof., and he says something unhelpful about using recursion now... as if I could just type in `(increase (rest myList))` at this point! Just to make it clear how unhelpful he's been, I do type that in and try it out ... for some weird reason it works.

I trace out a few small examples with the stepper, and then with paper and pencil, and it becomes clear that it should work. Each time Scheme encounters `(increase (rest myList))` the value of `(rest myList)` gets smaller, so eventually the case where we're increasing an empty list is reached, and everything works out.

```
(define (increase myList)
  (if (empty? myList) '()
      (append (list (+ 1 (first myList)))
              (increase (rest myList))))))
```

**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:
(a) If the length of the list is 0 then palindrome? should return true.
(b) If the length of the list is 1 then palindrome? should return true.
(c) 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.
(d) 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))
;; (square 10 'solid 'blue)) produces true.

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.

SOLUTION: To solve this exercise I had to combine a lot of the things from previous questions. First of all I had to remember how to use "cond" (since a simple "if" was not enough to deal with all the conditions). The first two conditions were fairly simple, I had to check whether the length of the list is 0 or 1 and return true. The third condition was hard, so I first wrote down the forth one (else false) to take care of all other cases. Now for the third condition. I understand that this condition has an "and" and it returns true, so I wrote ( (and ... ... ) true). The first part of the "and" is easy, I have to see is the first and last element of the list are equal (equal? (first myList) (last myList)). The second part of the "and" check needed recursion. I had to see if the list of all the elements minus first and last is a palindrome. At first I wrote (palindrome? (removeLast myList)) but I was getting strange results, until I realized I was only removing the last element of the list and not the first. So I added a "rest" before that.

(define (palindrome? myList)
  (if (< (length myList) 2) true
      (and (equal? (first myList) (last myList))
           (palindrome? (rest (removeLast myList))))))

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
draw 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

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:

;; 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)
   (/ offset 2) (* (/ (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.

Solution: I had to think this exercise through a bit to understand the code. These square roots really
confused me, until I started looking at each overlay/xy segment on its own. The first thing we
want to overlay is a picture that is itself a combination (overlay/xy) of two other pictures offset
by "offset" on X and 0 on Y. So it seems these two images are next to each other. I bet they are
the base of the triangle. The other picture that is added is moved in a weird position. First of
all it is moved half way between the other two on the X axis. This makes sense, since it is the
top of the triangle. It is then shifted upwards by a very strange distance (* (/ (sqrt 3) -2) offset).
I write this down in a way I can understand it better $-\frac{\sqrt{3}}{2} \ast offset$. This does not seem very
familiar, but I trust the person who wrote it down and assume it moves the third image upwards
by the correct amount to create the triangle of the three images.

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).

(a) 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.

(b) 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.

SOLUTION: I really panicked the first time I saw the definition of sierpinski. Then I decided to start working the same way I did for all my previous work. First of all there is an "if" statement associated with this function. The first thing to check is whether (equal? steps 0), something I've done before. If this condition is true, I need to draw an equilateral triangle made up of images of three solid triangles. I know I can do that with trefoil where the images are three solid triangles of three colors. I type this down and Scheme immediately complains that the function trefoil takes 4 parameters. It seems I forgot to say that the offset needs to be "base". I write this down and start the second part of the "if" statement. This smells like recursion (sierpinski gets involved again). I believe I can use trefoil again, only now the images are actually calls to (sierpinski (- steps 1) base). The offset of this new trefoil is (* base (expt 2 steps)). I had to look back to the first exercise on arithmetic functions to figure this out.

(define (sierpinski steps base)
  (if (equal? steps 0)
      (trefoil (solidT base 'green) (solidT base 'red) (solidT base 'blue) base)
      (trefoil (sierpinski (- steps 1) base)
               (sierpinski (- steps 1) base)
               (sierpinski (- steps 1) base) (* base (expt 2 steps)))))

WHAT TO HAND IN

Under Assignments on the course web page you will find a link to the CDF 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, submit 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.