CSE 341, Autumn 2010
Assignment #6 - Scheme: Basics / Random Art (100 points)
Due: Friday, November 12, 2010, 9:30 AM

This homework focuses on basic Scheme programming. The first part of this assignment involves writing simple Scheme functions to practice the language and its syntax. The second part of the assignment involves generating random expressions that can be used to produce art. Include your answers in a file named `hw6.scm`.

Part A - Scheme Functions:

1. `repl`

Define a function `repl` that takes a string and an integer as parameters and that returns the given string repeated the given number of times. For example, `(repl "hi!" 4)` should return `"hi!hi!hi!hi!"`. If the integer passed is 0 or negative, return the empty string. *(Use the library function `(string-append` **str1 str2**`)` to help you.)*

2. `list-twos`

Define a function `list-twos` that takes an integer *n* as a parameter and that returns a list of a factorization of that integer with all possible 2s factored out. The list should begin with as many 2s as can be divided out of *n*, followed by the integer that is left after all such 2s have been extracted (which is by definition an odd integer). You may assume that n is a positive integer (greater than 0).

- `(list-twos 56)` should return `(2 2 2 7)` because 56 is 2*2*2*7
- `(list-twos 80)` should return `(2 2 2 2 5)` because 80 is 2*2*2*2*5
- `(list-twos 19)` should return `(19)` because 19 is already odd

3. `collapse`

Define a function `collapse` that takes a list of numbers and that collapses successive pairs of numbers by replacing the pair with the sum of the numbers. If the list has an odd length, then the last value is not collapsed. For example:

- `(collapse '(2 8 3.1 2.4 3 6.5))` should return `(10 5.5 9.5)`
- `(collapse '(13 4 27 9 48))` should return `(17 36 48)`

4. `stretch`

Define a function `stretch` that takes a list of integers as its parameter and that doubles its length by replacing each integer with two numbers that add up to the integer. The two numbers should each be half of the original, but if the original number is odd, then the first number in the new pair should be one higher in absolute value than the second. You might want to call the functions `quotient` and `modulo`. For example:

- `(stretch '(38 4 19 7 -5))` should return `(19 19 2 2 10 9 4 3 -3 -2)`

5. `sum-squares`

Define a function `sum-squares` that takes a list of numbers as its parameter and computes the sum of the squared values of all the numbers in the list. You may assume that the list contains at least one element. For full credit, you may *not* use recursion on this problem; you must solve it using higher-order functions. For example:

- `(sum-squares '(1 2 -3 4))` should return `30`, because $(1^2 + 2^2 + (-3)^2 + 4^2 = 30$.

6. `all-pairs`

Define a function `all-pairs` that takes two lists as parameters and that returns a list of all of the ordered pairs that can be formed with one value from the first list and one value from the second list. The list should be ordered first by the order of values from the first list and within those groups, ordered by the order of values from the second list. You can assume that the lists contain no duplicates. For example:

- `(all-pairs '(3 19 7) '(a b c d))` should return
  `((3 a) (3 b) (3 c) (3 d) (19 a) (19 b) (19 c) (19 d) (7 a) (7 b) (7 c) (7 d))`

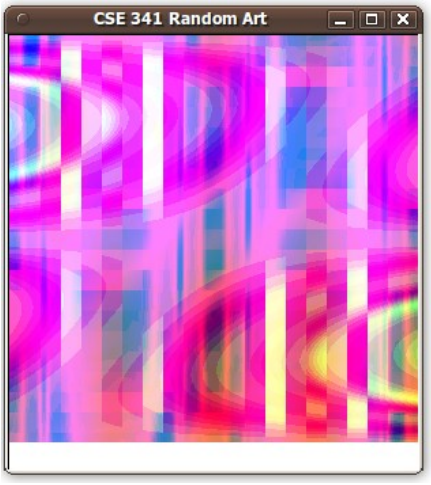
Part B - Random Art:

The second part of the assignment involves generating random expressions that can be used to produce art, such as the example image at right. Random art can be produced by mapping various mathematical functions and expressions onto the x/y plane and mapping various random numbers into the RGB color space. You won't actually need to do any drawing on this assignment. Instead, you will write code to generate random mathematical expressions that are fed to an instructor-provided file `graphics.scm` that draws the actual graphics.

The expressions that you generate will all produce values in the range of -1 to +1. The simplest expression is just the letter *x* or *y*. But we can build up complex expressions from simpler expressions by applying functions that produce values in the -1 to +1 range, when given arguments in that range. Below are several examples of such functions:

```
sin(pi * value)
cos(pi * value)
(value + value) / 2
value * value
```

How does it work?



One of the great strengths of Scheme is that Scheme code itself can produce other Scheme code that can be executed later. We will take advantage of that feature in this second part of this assignment. You will write a procedure that generates and returns random mathematical expressions. Rather than generating these as strings, you will generate actual code representing expressions with Scheme procedures to call, such as sine, cosine, multiplication and division. The expressions that you return will be used by the procedures in `graphics.scm`.

The math expressions that you produce will refer to *x* and *y* coordinates. To produce random art, our instructor-provided code generate three such math expressions, one each for the red, green, and blue components. Then our code scales the range of -1 to +1 so that it matches the width and height of our picture and we pick a set of red/green/blue colors for each pixel of our picture. The results are quite interesting.

The picture at right was produced with expressions nested 8 levels deep.

Implementation Details:

7. `generate`

Define a function `generate` that takes an integer *n* as a parameter and that generates a random expression of depth *n*. You may assume that *n* is greater than or equal to 0. For an expression of depth 0, your function should produce either the symbol `x` or the symbol `y`, each with equal probability. For an expression of depth greater than 0 you should produce one of the following five Scheme expressions, each with equal probability:

```
(cos (* pi expr))
(sin (* pi expr))
(/ (+ expr expr) 2.0)
(/ (round (* expr 10.0)) 10.0)
(* expr expr)
```

Each occurrence of **expr** above should be replaced with a random expression of depth (*n* - 1). For the expressions that have two occurrences of **expr**, the two expressions should be generated by two separate expressions (rather than using the same expression twice). To get random numbers, call the Scheme procedure `random` that takes no arguments and returns a random real number from 0.0 to 1.0. Here are some example calls to your procedure and possible results:

- `(generate 0)` could return `x` or `y`
- `(generate 1)` could return `(sin (* pi y))`, or `(/ (round (* x 10.0)) 10.0)`, etc.
- `(generate 2)` could return `(cos (* pi (* y y)))`, or `(/ (+ (/ (+ y x) 2.0) (sin (* pi y))) 2.0)`
- `(generate 3)` could return `(/ (+ (sin (* pi (sin (* pi y)))) (cos (* pi (/ (+ y x) 2.0)))) 2.0)`

You will be returning list expressions that contain *symbols*. In the preceding example output, when you see a symbol such as `sin`, `cos`, or `/`, it isn't a string (a string would have had quotes around it). Instead, you are actually returning references to the Scheme procedures for computing a sine, cosine, division, etc. To return a symbol, you must precede that symbol by an apostrophe, `'` . For example, to return the expression containing the `cos` or `+` procedure, what you would actually write at that point in your code is `'cos` or `'+` . (You have also seen the `'` used in Scheme at the start of lists to distinguish a list of literal values from a function call.)

When you complete this part of the assignment, load the provided file `graphics.scm` and run the function `draw`, passing it a width, height, and expression depth. For example, the image in this document was produced by the call:

```
(draw 300 300 8)
```

The `draw` function assumes that you have correct definitions of `all-pairs` and `generate`. There is a variation of the `draw` function called `draw2` that updates the image every few seconds.

Note: On both parts of the assignment, some of the functions you are asked to write have a precondition/assumption about the type of value passed to the function. You should document these preconditions. If you want to, you can include code to check for errors and call the function `error` if a precondition is violated; but this is not required.

Grading and Submission:

Submit your finished `hw6.scm` file electronically using the link on the class web page.

For reference, our solution is 35 "substantive" lines long according to the class Indenter page, excluding blank lines and comments. You don't need to match this number or even come close to it; it is just a rough guideline.

Your program should not produce syntax **errors** when executed. You may lose points if you name your functions or top-level values incorrectly, even if their behavior is correct.

Your code should work for both basic and advanced cases. Perform your own **testing**, and remember to test edge cases.

You should not use any of Scheme's language features that involve **mutation**, such as the `set!`, `set-car!`, or `mcons` procedures. You also should not use **quasi-quotes** with the back-tick (`` ` ``) character (note that this is not the same as the apostrophe character, `'`, which *is* allowed.) Do not define Scheme **macros**. Your code should compile/execute properly using the **"Pretty Big"** language level of PLT Scheme. Otherwise you may use any Scheme library constructs you like to help you solve a problem unless those constructs are explicitly forbidden by this spec or the problem description.

As always, if the solution to one function is useful in helping you solve a later function, you should call the earlier function from the later function. You can include any testing code you develop (although it should be labeled as such). Other than testing data/code, do not define any other **global symbols** not described in this document.

If you write inner helper functions, you should choose a suitable set of **parameters** for each one. Don't pass unnecessary parameters or parameters that are unmodified duplicates of existing bound parameters from the outer function.

You are expected to use good programming **style**, such as naming, indentation/spacing, and avoiding redundancy when possible. Avoid long lines of over 100 characters in length; if you have such a line, split it into multiple lines.

In general, favor the `cons` procedure to grow lists versus the `concat` procedure (though the `concat` procedure is sometimes necessary and is not forbidden entirely). Favor the use of higher-order functions such as `map` over manually iterating over / recursively processing a list.

Place a descriptive **comment** header at the top of your program along with a comment header on each procedure, including a description of the meaning of its parameters, its behavior/result, and any preconditions it assumes. If you declare a non-trivial or non-obvious inner helper, also briefly comment the purpose of that helper in a similar fashion.

Efficiency on a fine level of detail is not crucial on this assignment. But code that is unnecessarily computationally inefficient (such as code that performs an algorithm that should be $O(n)$ in $O(n^2 \log n)$ time) might receive a deduction.

Redundancy, such as recomputing a value unnecessarily or unneeded recursive cases, should be avoided. Avoid repeating large common subexpressions as much as possible; for example, the pseudo-code *"if (test) then (1 + really long expression) else (2 + really long expression)"* is better written as *"((if test then 1 else 2) + really long expression)"*.