CSC172 LAB

MORE SCHEMING

1 Introduction

The labs in CSC172 will follow a pair programming paradigm. Every student is encouraged (but not strictly required) to have a lab partner. Labs will typically have an even number of components. The two partners in a pair programming environment take turns at the keyboard. This paradigm facilitates code improvement through collaborative efforts, and exercises the programmers cognitive ability to understand and discuss concepts fundamental to computer programming. The use of pair programming is optional in CSC172. It is not a requirement. You can learn more about the pair programming paradigm, its history, methods, practical benefits, philosophical underpinnings, and scientific validation at http://en.wikipedia.org/wiki/Pair_programming .

Every student must hand in his own work, but every student must list the name of the lab partner (if any) on all labs.

This lab has six parts. You and your partner(s) should switch off typing each part, as explained by your lab TA. As one person types the lab, the other should be watching over the code and offering suggestions. Each part should be in addition to the previous parts, so do not erase any previous work when you switch.

The textbook should present examples of the code necessary to complete this lab. However, collaboration is allowed. You and your lab partner may discuss the lab with other pairs in the lab. It is acceptable to write code on the white board for the benefit of other lab pairs, but you are not allowed to electronically copy and/or transfer files between groups.

1. Since it has been well established in previous labs that writing computer programs in the scheme language puts you in touch with the great universal meaning of all reality, let us now move beyond cosmic transcendence and look at calculating a the square root of a number. This exercise should give you greater insight into the power, beauty and simplicity of the LISP language.

   Begin by defining a function in scheme to return the square of a number. This function should take a parameter and return that parameter times itself. Secondly, define a function be used as a way to measure if one number is close to another number. You know that you can use `(- x y)` to take the difference between two numbers. You can use `(abs x)` to get the absolute value of an expression. You can use `(< expression 0.0001)` to check if a number is “small”. So, knowing all this write a function in scheme that takes two numbers as parameters

return true if the absolute value of the difference between the value of the first and the value of the second is greater than 0.0001. So, your function should return "#f" for `(close-to 1.99 2)` and "#t" for `(close-to 1.99999 2 )`.

2. To get a square root we can use Newton's method of successive approximations. This works by starting with some number `x` that we want the square root of. We start `y` with a "guess" for the square root. Think of the guess as an approximation of the square root. With x and an approximation of the square root of x (the value y) we can get a better approximation for the square root by taking the average of y and (x/y). You can try this out by hand to see if it works – it's a rather famous method. Anyone know why it works (a picture helps, as does knowing this is a special case of solving f(x) = 0 for any differentiable f, as does incorporating the derivative (tangent, hint hint)).

   Start by writing a function that returns the average of two numbers passed in as parameters.

   Second write a function (call it "improve") that takes two numbers (x and y) as parameters and uses the average function you just wrote to compute the and return the average of y and (x/y).

3. Equipped with your "square", "close to", "average" and "improve" functions you can recursively implement Newton's method for finding square roots. Write a method (call it "sqrt-recur") that takes a number that you want the square root of and a "guess/approximation". If the guess is "close" to the square root of the number, return the guess. Otherwise return the value of a recursive call to "sqrt-recur" with the number that you want the square root of and an "improved" guess. Once you have this write a function (call it "sqrt") that starts the recursion with 1.0 as the initial guess and returns the result. Test your function on a few numbers.

4. So, square roots are fun. What if we want cube roots? It turns out that if we have a number x and a number y which is a "guess" for the cube root of x, then we can get a better guess by calculating `(((x/(y^2))+2y)/3)`. (not mysterious if you understand what's going on as mentioned in Section 2 above). Using the same basic structure that you used to define square root define a "cube-root" function using this new method of approximation. Test your function on a few numbers.
5. In scheme, the `(remainder a b)` function will return the value of a modulo b. Using this, we can write a function that returns the greatest common divisor GCD of two numbers. Write this function as follows, your function should take two numbers a and b. If b is zero, then return a otherwise return the GCD of b and (remainder a b) . This is Euclid's algorithm for finding the GCD. Test your function on a few pairs of numbers.
6. We sometimes want to print out values during the computation of a lisp/scheme function. We have two valuable functions (newline) and (display x) that can be used for "printing out" values. Try typing in and testing the following code for factorial.

```
(define (myfact x)
   (begin
     (display x)
```

```
    (newline)
    (if (< x 1) 1 (* x (myfact (- x 1))))
  )
)
```

Using these functions, modify your sqrt method to print out the successive “guesses” that occur during the computation.

2 Hand In

Hand in the source code from this lab at the appropriate location on the blackboard system at my.rochester.edu. You should hand in a single compressed/archived (i.e. “zipped” file that contains the following.)

1. A README that includes your contact information, your partner's name, a brief explanation of the lab (A one paragraph synopsis including information identifying what class and lab number your files represent.).
2. The source code files representing the work accomplished for this lab. All source code files should contain author and partner identification in the comments at the top of the file. For this lab, this is your file containing scheme code.
3. A plain text file named OUTPUT that includes author information at the beginning and shows the compile and run steps of your code. The best way to generate this file is to cut and paste from the command line. For this lab, this is a file containing examples of your code working cut and pasted from you interactive session.

3 Grading

172/grading.html
Each section (1-6) accounts for 15% of the lab grade (total 90%)

(README file counts for 10%)