1 Lab 3 Scheme 2

Date: Mar 11, 2021

This document first describes the aims of this lab followed by exercises which need to be performed.

You should perform all exercises using the `racket` dialect of Scheme which is installed on remote.cs. Though this lab assumes that you are performing the exercises within a terminal, you can choose to use `drracket` which provides a GUI interface. If you do use `drracket` you will need to ensure that you capture your output sufficiently to convince the TA that you have indeed completed the lab.

1.1 Aims

The aim of this lab is to introduce you to Scheme. After completing this lab, you should have some familiarity with the following topics:

- More practice writing expressions using Lisp's parenthesized prefix notation.
- Writing simple Scheme functions to do arithmetic.
- Writing simple list processing functions in Scheme.
- Using functions as first-class values.

1.2 Exercises

1.2.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab3` branch and a new `submit/lab3` directory.

For this lab you will write all code in a single file `lab3-sol.scm` residing in your `submit/lab3` directory and submit that file along with a log of your terminal interaction.

1.2.2 Exercise 1: Arithmetic in Scheme

Use the well-known formula for solving a quadratic equation to implement a function `(quadratic-roots a b c)` to return a 2-element list containing the roots of the quadratic equation $ax^2 + bx + c = 0$. Scheme provides a `sqrt` function for extracting square roots.

Note that your function should evaluate the discriminant $\sqrt{b^2 - 4ac}$ only once:

```
> (load "lab3-sol.scm")
> (quadratic-roots 3 7 2)
'(-1/3 -2)
> (quadratic-roots 5 6 1)
'(-1/5 -1)
> (quadratic-roots 5 2 1)
'(-1/5+2/5i -1/5-2/5i)       ;;get complex roots for free!!!
> (quadratic-roots 1 -4 25/4)
'(2+3/2i 2-3/2i)
> (quadratic-roots 0 2 1)
; /: division by zero [,bt for context]
```

To take care of the last case above, change your function to return the symbol `'error` when a is 0.

```
> (quadratic-roots 0 1 2)
'error
> (quadratic-roots 1 -4 25/4) ;;still works
'(2+3/2i 2-3/2i)
>
```

Now add a function `(my-sqrt n)` which uses Newton's method to find the square root of a number $n > 0$. Start out with a `guess` of `1`. At each successive stage, set `guess` to the average of its current value and `n/guess`. Continue until the the relative absolute error between `guess`^{2} and n is less than `0.01%`.

Note that Scheme provides a `abs` function.

Specifically, your program should work as per this pseudo-code:

```
Number my_sqrt(n) {
  guess = 1;
  while (abs(guess*guess - n)/n < 0.0001) {
    guess = (guess + n/guess)/2;
  }
  return guess;
}
```

You can provide `guess` as an optional parameter to your Scheme `my-sqrt` function with a default value of 1, using Scheme's syntax for default values. Here is an example of the use of that syntax:

```
> (define f (lambda ((a 22) (b 44)) (+ a b)))
> (f 2 4)
6
> (f 2)
46
> (f)
```

```
66
>
```

So you can define your `my-sqrt` with initial header set up as follows:

```
(define my-sqrt
  (lambda (n (guess 1))
    ...
  ))
```

Of course, you will need to replace the loop in the above pseudo-code with recursion, since it is impossible to write a loop in the subset of Scheme we are emphasizing in this course. Once you transcribe the above pseudo-code into Scheme syntax, you should have it working:

```
> (my-sqrt 2)
577/408
> (my-sqrt 9)
65537/21845
```

Note that Scheme keeps the above computations within the domain of rational numbers.

Change the default value for the `guess` from `1` to `1.0`. Now the results change to real numbers:

```
> (my-sqrt 2)
1.4142156862745097
> (my-sqrt 9)
3.00009155413138
```

Finally, add a fourth optional parameter to `quadratic-roots` specifying the function to be used for extracting square-roots with a default value set to the Scheme `sqrt` function.

```
>  (quadratic-roots 3 7 2)
'(-1/3 -2)             ;;still works, defaulting to sqrt
>  (quadratic-roots 3 7 2 my-sqrt)
'(-0.3333294702910085 -2.0000038630423247)

;;my-sqrt cannot handle negative arguments
>  (quadratic-roots 1 -4 25/4 my-sqrt)
  C-c C-c; user break [,bt for context]

;;Scheme's sqrt handles negative arguments fine
>  (quadratic-roots 1 -4 25/4)
'(2+3/2i 2-3/2i)
>
```

1.2.3 Exercise 2

In this exercise, you will write some simple recursive list functions:

Write a Scheme function `greater-than` which when given a first argument `ls` which is a list of numbers and a second argument a number `v` which defaults to 0, returns a list having the same length as `ls` but having elements which are `#t` or `#f` depending on whether or not the corresponding element of `ls` is greater-than `v`. Your function must make use of recursion.

```
> (greater-than '(-1 3 6 -3 1 8) 2)
'(#f #t #t #f #f #t)
> (greater-than '(-1 3 6 -3 1 8))
'(#f #t #t #f #t #t)
> (greater-than '())
'()
>
```

Now write a function `get-greater-than` with the same arguments as `greater-than` but the returned value should be those elements of `ls` which are greater-than `v`.

Note that your case-analysis for this problem will have the usual cases for when `ls` is empty and when it is not empty. However, the not empty case will have two sub-cases: when `(car ls)` is greater-than `v`, and when it is not. Instead of using nested if-then-else expressions, you can use a `cond`.

```
> (get-greater-than '(-1 3 6 -3 1 8) 2)
'(3 6 8)
> (get-greater-than '(-1 3 6 -3 1 8))
'(3 6 1 8)
> (get-greater-than '())
'()
>
```

Now write a `less-than` and `get-less-than` function with an API similar to the previous two functions but using the less-than relation instead of the greater-than relation. This can be done by simply copying the previous two functions and making the obvious changes:

```
> (less-than '(-1 3 6 -3 1 8) 2)
'(#t #f #f #t #t #f)
> (less-than '(-1 3 6 -3 1 8))
'(#t #f #f #t #f #f)
> (less-than '())
'()
> (get-less-than '(-1 3 6 -3 1 8) 2)
'(-1 -3 1)
> (get-less-than '(-1 3 6 -3 1 8))
```

```
'(-1 -3)
> (get-less-than '())
'()
>
```

That was easy, but whenever you write code by copying existing code and making a few changes, you should be extremely unhappy. You should be looking for ways to generalize your original code to allow both variations.

1.2.4 Exercise 3: First-Class Functions

[This exercise does not require you to add any code to your `lab3-sol.scm` file. All you need to do is type expressions into the racket REPL with your terminal interactions logged into your `script` log.]

Functions are first-class citizens in Scheme in that they can be used like any other values:

- They can be used without being given a name; i.e. we have anonymous functions represented by `lambda` expressions.
- They can be stored in data-structures.
- They can be used as function arguments or results.

Note that a function which takes functions as arguments or returns a function result is known as a **higher-order function**. Scheme has several higher-order functions which allow you to replace the functions from the last exercise by single expressions.

The Scheme function `map` will apply the function specified by its first argument to each element of the list specified by its second argument and returns the resulting list:

```
> (map (lambda (n) (> n  2)) '(-1 3 6 -3 1 8))
'(#f #t #t #f #f #t)
> (map (lambda (n) (> n  0)) '(-1 3 6 -3 1 8))
'(#f #t #t #f #t #t)
```

1. Provide the expressions which evaluate the `less-than` equivalent.
2. Use `map` within an expression to returns a list of `#t` or `#f` depending on whether the corresponding list element is odd or even. So the list `'(1 2 3 4 5)` should map to `'(#t #f #t #f #t)`.
3. Type in an expression which returns the squares of each element of the list. So the list `'(1 2 3 4 5)` should map to `'(1 4 9 16 25)`.

Scheme contains another higher-order function `filter` which can be used to implement the `get-*` functions from the previous exercise:

```
> (filter (lambda (n) (> n  2)) '(-1 3 6 -3 1 8))
'(3 6 8)
```

1. Type in an expression which returns those elements of the above list which are less-than 2.
2. Type in an expression which returns a list containing only those elements of the list which are odd: So the list `'(1 2 3 4 5)` should be filtered to `'(1 3 5)`.

Another useful higher-order function is `foldl` which reduces a list of values to a single value by repeatedly applying a binary function to the list values. Specifically, `foldl` is called with three arguments: the binary function, an initial value and the list to be reduced:

```
> (foldl (lambda (a b) (+ a b)) 0 '(1 2 3 4 5))
15
> (foldl (lambda (a b) (* a b)) 1 '(1 2 3 4 5))
120
>
```

1. Use the exponentiation function `expt` to evaluate

$$2^{2^{2^2}}$$

   For fun, add in an extra 2!
2. Use `foldl` along with `cons` to reverse a list.
3. First class functions means that functions can also be stored in data-structures like lists. For your grand finale, use two nested applications of `map` to map a list of functions over a list.

   For example, mapping the 2-element list of functions `((lambda (n) (+ n n)) (lambda (n) (* n n)))` over the list `'(1 2 3 4 5)` should result in the 2-element list `'((2 4 6 8 10) (1 4 9 16 25))`.