CSE 1729 - Introduction to Principles of Programming	September 4, 2018
Laboratory Assignment 1	

Remember to submit your signed Honor Code agreement during the lab!

Objectives

In this short lab assignment you should

1. Become familiar with Racket, a Scheme interpreter/environment;
2. Be able to set the Scheme mode and know the default mode we will use for the course;
3. Be able to enter expressions in Racket and understand their evaluation;
4. Practice with function abstractions
5. Be able to submit your work via the Mimir site for CSE 1729.

What's All the Racket?

Racket

Scheme is the programming language we will be using for almost all programming in CSE 1729. Scheme programs are "executed" by an interpreter and constructed in a "dialog" with the interpreter. Racket is an implementation of an interpreter for Scheme and the one you will be using for the course. All lab assignments and problem sets should be completed and tested using Racket before submission. Racket is available on all workstations in School of Engineering, Learning Center computer labs. If you would like to work on your own machine, you may download Racket (for free) for your operating system at `http://racket-lang.org/download/`.

Starting Racket

Once installed properly, you should be able to start Racket by finding and double clicking the 'DrRacket' icon. On MS Windows machines you are likely to find it by clicking "All Programs"-> Applications->Racket->"Dr. Racket." On MacOS machines, you should click Applications -> Racket v6.10 -> DrRacket[1]. If you prefer a command-line interface, you can also start Racket by typing 'racket' in a Terminal window on Mac machines, or command prompt on Windows machines. You should see something similar to:

```
Welcome to DrRacket, version 7.0 [3m].
Language: R5RS; memory limit: 256 MB.
>
```

[1] 7.0 is the version number. Yours may be older, like 6.6.something

Racket runs a *read-eval-print* loop. That is, it continuously repeats a cycle of reading a Scheme expression, evaluating that expression and printing the result of evaluating that expression. Right now it is just waiting for an expression to evaluate. Since Racket supports several dialects of Scheme, we can indicate the particular dialect we want the interpreter to emulate. For most of the semester, we will follow the R5RS standard which we can indicate opening the Language -> Choose Language menu item and selecting the "Other Languages" radio button and clicking on R5RS in the dialog box and clicking on the "OK" button. Go ahead and do that now. You should use this mode unless otherwise indicated for the rest of the course.

Expressions

You will be learning more about expressions and combining expressions in lecture. For now, let's start with the simplest type of expression, a number. Go ahead and type any number that comes to mind, perhaps you have a favorite, into the upper half of the Racket window. This is the Definitions panel. Then click Run at the top of the screen. I like the numeral 9, so I'll type '99' and the interpreter responds with: `99`. So, a number evaluates to the value it represents.

A slightly more complex expression might include a mathematical operator and some operands. But, before we do that, we should note that Scheme uses *prefix notation*. So, our expression should start with the operator and be followed by all of the operands. You can add this to your Definitions window and click Run again. All expressions in the Definitions window are evaluated again and the results displayed in the bottom, Interactions, window. So if I type '+ 33 66' I might expect the interpreter to respond with `99` but instead we get:

```
#<procedure:+>
> 33
> 66
```

The interpreter evaluates each operator or operand individually as if each is a separate expression. Not quite what we had intended. To show these more complex expressions are composed of the operator and two or more operands (yes, you can include more than two) they are enclosed in parentheses and are considered a list. So, if I were to type '(+ 33 66)' in the interpreter, it would print `99` as we would expect. Next, satisfy your curiosity by entering an expression with three operands (e.g. '(+ 33 33 33)') and satisfy yourself that the operator is applied as you would expect.

In addition, we can use the result of evaluating an expression as an operand for another, larger, expression. For instance, type this expression (which can be found in your book) into the Scheme interpreter:

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

and take a moment or two to work out the result so that you can "play computer" and demonstrate to yourself that you understand how the interpreter is evaluating this expression. Now, this expression as it is shown here is fairly difficult to sort out. That is, it is difficult to determine which expressions belong to which operators. In order to help us humans visualize how an expression is composed, and therefore evaluated, we can use proper indentation (also known as *pretty printing*). The indentation we use lines up operands vertically by inserting tabs where appropriate as shown below and in your textbook. Notice, for example that `(* 2 4)` and `(+ 3 5)`, line up.

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
```

Racket provides help in pretty printing in both the lower (interaction) and upper (definition) panes. Type (or copy-paste) the one-line version of the expression into the definition pane. Now add line breaks by placing the cursor where you want the new line to start and hitting enter – notice how Racket lines things up. If you edit the indentation might get out of line, at which point you can indent an individual line relative to its predecessor using the tab key, or indent the whole definition pane by using the "reindent all" command, found under the Racket menu. SCHEME does not care if you indent or not, but properly-indented code is easier to read, and easier to understand.

Note: You must use pretty printing in all of your submissions for lab assignments and problem sets.

Using Mimir and Submitting Assignments

All SCHEME assignments in this course will be submitted using Mimir – you should be able to log into Mimir from Moodle and find the assignment under "Course Content" > "All coursework". Once your work in the Definitions pane has been written and saved as some appropriate name (e.g. lab1.rkt in the case of this lab), you may submit it by hitting the Submit button and choosing your file. Saving the definitions pane is done by selecting File -> Save Definitions and giving your solution file the correct name. You can reload your work by selecting File -> Open from the Racket menu bar and selecting the definitions file you've previously saved, at which point you can do more editing and testing. **All assignments for this course must be submitted via Mimir. No assignments will be accepted via email.**

A nice feature of Mimir is that when you submit your file, it will run tests on your work. This is used in grading your work, but also provides you with immediate feedback about what works and what doesn't.

Making it easy to read your results

Mimir will run tests on your code, but you should test your code as well–knowing the results from test cases will help you debug. We suggest using a specific labeling convention with Racket definition files to make this easier for you.

This will involve the use of strings. A string in Scheme is a sequence of characters within quotation marks. We will use strings later for various things, but for now it is sufficient to know that a string evaluates to a string that looks like itself. So if you put a string in a Racket definition pane, then hit Run, that string will show up in the lower window as the result of its evaluation.

We will use this as follows: before the code from a problem, we will identify which problem it is, and for each test case we will put in a string identifying the test case. Here is a simple example: suppose question 3 asks you to write a square function in Scheme. What you put in the Racket definitions will be something like the following:

```
"problem 3a."
(define (square x)
```

```
  (* x x))
"tests"
"(square 4)"
(square 4)
"(square -5)"
(square -5)
"(square (square 3))"
(square (square 3))
```

What you (and we) will see in the lower window when you hit the Run button is:

```
"problem 3a."
"tests"
"(square 4)"
16
"(square -5)"
25
"(square (square 3))"
81
>
```

Now it is easy to see what your test results are. Your definitions should be in the same order as in the assignment, with tests within each question.

Try This!

Practice your newly developed Scheme powers with the following expressions:

1. Use the Scheme interpreter to compute the product $34,234,567 \times 23,123,456$. Don't include the commas though.
2. Use the Scheme interpreter to compute $45,321,123^3$.

Lab Assignment Activities

Note: these activities will be graded using Mimir. You should submit partial solutions as you go to see what test cases pass. Be sure to save your definitions before submitting.

1. Currency Conversion
   (a) The exchange rate between U.S. dollars and British pounds is \$1 = £0.77 . Write a Scheme procedure `usd-gbp` to convert dollars into pounds. How many pounds will you get when you exchange \$175?
   (b) The exchange rate between British pounds and euros is £1 = €1.11 . Write a Scheme procedure `gbp-eur` to convert pounds into euros. How many euros will you get when you exchange £20?
   (c) The exchange rate between euros and Swedish krona is €1 = 10.61 SEK. Write a Scheme procedure `eur-sek` to convert euros into krona. How many krona will you get when you exchange €240?
   (d) Write a procedure `usd-sek` using your solutions from parts (a), (b) and (c) to convert U.S. dollars to Swedish Krona. What is the exchange rate from U.S. dollars to Swedish Krona (that is, how many krona can you get for each dollar?
2. A *matrix* is a rectangular grid of numbers organized into rows and columns. Matrices are an important tool in algebra and are often used to solve systems of linear equations. Below are examples of a couple of 2×2 matrices (matrices with 2 rows and 2 columns) that we will call M and N.

$$M = \begin{pmatrix} 2 & -4 \\ -6 & 12 \end{pmatrix} \qquad N = \begin{pmatrix} -3 & 1 \\ 2 & 7 \end{pmatrix}$$

   (a) A special value associated with any 2×2 matrix is the *determinant*. Given a generic 2×2 matrix, the determinant can be computed using the following formula:

$$\det \begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad - bc$$

   Using the formula, we can compute the determinant of matrix M above as $(2)(12) - (-4)(-6) = 0$. Write a Scheme procedure `det2x2` to compute the determinant of a generic 2×2 matrix. Assume that the matrix elements a. b, c and d are given as four formal parameters. Compute the determinant of N.

(b) A matrix is called *invertible* if its determinant is non-zero. Write a procedure `invertible?` that checks whether or not a generic 2×2 matrix is invertible. Use the same formal parameters as for `det2x2`. Verify that N is invertible and M is not invertible. (*Note:* you may want to use one or more of the primitive functions `=, <, >`, each of which compares two numbers and evaluates to the boolean value `#t` or `#f`. There are also logical connectives `and`, `or`, and `not` for combining boolean values.)

(c) A powerful property of matrices is that certain kinds of matrices may be meaningfully *multiplied* together to get another matrix. (It turns out that matrix multiplication is intimately related to composition of linear functions, but you won't need this interpretation to complete the assignment.) In particular, it is possible to multiply 2×2 matrices. Assume we have two matrices:

$$A = \begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \qquad B = \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix}$$

The product of these matrices is defined to be

$$A \cdot B = \begin{pmatrix} a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\ c_1a_2 + d_1c_2 & c_1b_2 + d_1d_2 \end{pmatrix} .$$

Given two 2×2 matrices, we wish to determine whether or not their product $A \cdot B$ will be invertible. There are two ways to do this

1. Compute the product, as described above; then compute its determinant.
2. It is a remarkable fact that for two matrices A and B, $\det(A \cdot B) = \det(A) \times \det(B)$. Thus, we can compute the determinant of $A \cdot B$ directly from the determinants of A and B.

Give two different Scheme procedures to compute whether the product of a pair of matrices is invertible, one for each of the two methods described above. Assume elements a_1, b_1, c_1, d_1, a_2, b_2, c_2 and d_2 are given as eight formal parameters in that order. These two functions should be named `prod-inv-direct?` for the first approach and `prod-inv-indirect?` for the second.

(d) The determinant of a 3×3 matrix can be computed from the following formula:

$$\det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = a \times \det \begin{pmatrix} e & f \\ h & i \end{pmatrix} - b \times \det \begin{pmatrix} d & f \\ g & i \end{pmatrix} + c \times \det \begin{pmatrix} d & e \\ g & h \end{pmatrix}$$

Fill in the body of the following procedure to find the determinant of a 3×3 matrix:

```
(define (det3x3 a b c
                d e f
                g h i)

  ; your code here

)
```

What is the determinant of $\begin{pmatrix} 0 & 5 & -6 \\ 8 & -11 & 4 \\ 5 & 1 & 1 \end{pmatrix}$?

Once you have finished:

1. Save your work (the definitions) to a file named `lab1.rkt`
2. Submit your lab solution file for grading via Mimir.

This lab is worth up to 10 points when completed and submitted. Please note: assignments will not be graded for credit until your Honor Code Agreement has been submitted–do it in today's lab.