Week 1 Lab: Introduction to Racket and Haskell

In this lab, you will be introduced to the new programming languages used in this course: Racket and Haskell.

Your TAs are prepared to help you get your software setup on your own computer, and will also be able to answer questions about the programming languages as you get started.

However, one of the major goals of this course is to get you comfortable learning new programming languages independently, and so encourage you to look up documentation on your own for things like syntax and built-in functions.

Task 1: Software installation

If you're using your own computer, your first step should be installing the relevant software on your computer. Please follow the *Software Installation Instructions* on Quercus.

You can skip this if you're working on a departmental machine, but we find most students want to use their own computer at some point during the term.

Task 2: Some interactive guides

We've provided two recommended interactive readings to orient yourself with the basics of Racket and Haskell:

- An Introduction to Racket with Pictures: https://download.racket-lang.org/releases/8.6/doc/quick/index.html
- Chapters 1 and 2 of "Learn You a Haskell for Great Good!": http://learnyouahaskell.com/chapters

Note that for maximum learning, it's not good enough to just read the webpages; actually follow along by *typing any provided code on your computer*. Even though the contents of these references may seem pretty basic right now, it's best to ensure that you're extremely comfortable with the new languages before moving onto more complex material.

Task 3: Writing some basic functions (9 points)

Write the following functions, first in Racket and then in Haskell:

- A function that takes a decimal number, and converts it into a percentage value, rounded to the nearest integer.
- A function that takes a string `s` and an integer `n`, and returns a new string that repeats `s` a total of `n` times.
- A function that takes a symbol (String in Haskell) and a list, and returns whether the symbol (String in Haskell) appears in the list.

We recommend tackling each question in order. Write the function in both Racket and Haskell before moving on to the next function.

Write at least one additional test for each function.

Note To run the Haskell tests, you will need to have QuickCheck installed. Please check the installation instructions for instructions on how to install QuickCheck.

Task 4: A Calculator (1 points)

In this more advanced, Racket-only task, we build a calculator that will evaluate **binary arithmetic expressions**. Our binary arithmetic expressions will have the following grammar:

```
Binary Arithmetic Expression Grammar

<expr> ::= NUM | (<op> <expr> <expr>)
<op>   ::= + | - | * | /
```

where `NUM` any integer literal in base 10, and the arithmetic operations `+`, `-`, `*`, `/` have the standard mathematical meanings.

Arithmetic expressions are be represented as a list of elements in Racket, and operators are be represented using symbols. For example, `(list '+ 2 3)` and `(list '/ 8 2)` are examples of valid expressions.

Note that with the way quoting distributes in Racket, `'(+ 2 3)` represents the same data structure as `(list '+ 2 3)`. Likewise, `'(+ 2 (+ 4 5))` represents `(list '+ 2 (list '+ 4 5))`. In other words, if you want start quoting lists this week, do *not* nest the quote operator `'`.

Your task is to write a function `calculate` that returns the numeric value of an expression.

Your code does not need to handle errors: you may assume that the input is a syntactically correct expression, and that no division by 0 will occur in the input.

Hint: The input data is a *recursive data structure*, and the structure of our code should follow the recursive structure of our data. In other words **use recursion**, and **trust the recursion**!

You might find the Racket built-in functions `number?` and list functions like `list?`, `first`, `second`, `third` helpful.

Submission and Instructions

Submit the files `w01lab.rkt` and `W01lab.hs` on Markus. To give you faster feedback on labs, you will be able to test your own code on Markus once every hour. However, it is up to you to debug and determine why your code pass or fail tests. On labs, the tests that you are able to run on Markus are the same tests that we will be running to determine your grade. (This will not be true for assignments, so that there will be a chance to develop testing skills.)

Please test your labs early. Markus sometimes hangs when running tests, particularly if you submit code that does not terminate. If so, you will not be able to continue to submit additional tests until the test times out.

For all labs and assignments:

- You may not import any libraries or modules unless explicitly told to do so.
- You may not use the function "eval" unless explicitly told otherwise.
- In Racket, you may not use any iterative or mutating functionality unless explicitly allowed. Iterative functions in Racket are functions like `loop` and `for`. Mutating functions in Racket have a `!` in their names, like `set!`. If use the materials discussed in class and do not go out of your way to find these functions, your code should be okay.
- The `(provide ...)` and `module (...) where` code in your files are very important. Please follow the instructions, and don't modify those lines of code! If you do so, your code will not be able to run and you will receive a grade of zero.
- Do not change the default Haskell language settings (i.e. by writing an additional line of code in the first line of your file)

Breaking any of these above rules can result in a grade of 0.

Moreover, code that cannot be imported (e.g., due to a syntax error, compilation error, or runtime error during import) will receive a grade of zero. Please make sure to run and test all of your code before your final submission.

You are permitted (and encouraged!) to write helper functions freely.