Exercise 1: Getting Started with Racket and Haskell

The goal for this exercise is to write some simple code in Racket and Haskell, and to get used to the syntax of both languages. You should first complete the software installations before working on this exercise. This exercise takes less time to complete than later ones, so that you have time to complete the required installations.

For this and future exercises, remember that:

- You may not import any libraries or modules unless explicitly told to do so.
- You may not use the function "eval" for any exercise or assignment unless explicitly told otherwise.
- You may not use any iterative or mutating functionality unless explicitly allowed. Remember that a big goal of this course is to learn about different models and styles of programming!
- You may write helper functions freely; in fact, you are encouraged to do so to keep your code easy to understand.
- We generally provide a set of tests for each exercise or assignment, but these tests are meant to help with understanding expected function behaviour. Please be warned that they are not nearly comprehensive enough to give confidence about correctness.

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

- 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 all of your code before your final submission, and test it on the Teaching Lab environment (which is the environment we use for testing).
- The `(provide ...)` code in your exercise file is 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.

Starter code

Download these files from Quercus files.

- `Exercises/ex1/ex1.rkt`
- `Exercises/ex1/Ex1.hs`

Submission instructions

Submit the files `ex1.rkt` and `Ex1.hs` to MarkUs: https://mcsmark.utm.utoronto.ca/csc324f20/

But before you submit, check that:

- Your code does not have any syntax error. If you only completed part of the exercise, replace any possibly syntactically incorrect code with the starter code.
- The `(provide ...)` code in your Racket file is left intact. Otherwise we won't be able to run our tests!

Note: MarkUs will be available starting September 16th.

Task 1: Working with Racket [5 pts]

Complete the functions in `ex1.rkt`, as specified by their documentation. A reminder that you may *not* use any mutating functions (e.g., `set!`), nor may you use any loop constructs (e.g., `for/list`). The purpose of this exercise is to get you writing pure recursive functions, and so you'll need to make good use of recursion to do so.

You may **not** use the built-in `count` function (this defeats the purpose of the exercise). You may, here and for the rest of the course, define your own helper functions, and use functions you're asked to implement in the implementation of other functions.

Tips:

- Use the DrRacket menu command `Racket -> Reindent All` (or keyboard shortcut) to fix code formatting frequently. This is a great way to keep track of the structure of your code, and also quickly identify missing parentheses.

- Remember that Racket, at least for now, is *expression-based.* The *body* of a function should consist of a single expression, and the value returned by the function is simply the result of evaluating the body. Because of this, you don't need a `return` keyword like you'll be familiar with from other languages.

We strongly recommend making sure this part is fully done before moving on to Task 2.

Task 2: Working with Haskell [5 pts]

In `Ex1.hs`, we've given function stubs for functions equivalent to the ones you wrote in Task 1. Your job here is to take your implementations of your functions from Racket and translate them to Haskell.

While you will need to look up built-in Haskell functions, your final code should look almost identical to the Racket code—and that's the point!

Tips:

- This section of *Learn You a Haskell* describes how to load your function definitions into the Haskell interpreter (`ghci`), for interactive testing.
- You can also use `runhaskell Ex1.hs` from the command line to run the entire module, which evaluates the `main` function, containing the provided sample property-based tests.
- The Haskell compiler error messages can be daunting. The most common beginner errors are type errors due to misunderstanding precedence (e.g., causing functions to be called on the wrong number of arguments). When starting out, we recommend *using parentheses aggressively* to mark subexpression boundaries. Once your program compiles (and is correct!), you can start removing redundant parentheses.

Task 3: Calculator [0 pts]

This task will help you understand structural recursion. While this task will not be graded for credit this week, you will need to complete an almost identical task next week.

Over the course of the next few weeks, we will build a calculator that will evaluate **binary arithmetic expressions**. The features supported by our language will grow over the next exercises. For now, our binary arithmetic expressions will have the following grammar:

```
Binary Arithmetic Expression Grammar

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

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

Arithmetic expressions will be represented as a list of elements in Racket, and operators will 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, you should *not* nest the quote operator `'`.

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

Hint: You might find the Racket built-in functions `number?` and `list?` helpful.