CSC324 Assignment 1: Checking for Tail Recursion

You will be able to complete this assignment after lecture 2. We recommend that you complete lab 2 before starting the assignment.

In this assignment, we will write a function that checks whether a Racket definition of a function is tail recursive. This is an example of a *static analysis*, where we try to reason about a piece of code without interpreting/evaluating it.

We will use a subset of Racket defined by the following grammar:

```
<def>  ::= (define (IDENTIFIER IDENTIFIER ...) <expr>)

<expr> ::= LITERAL VALUES      ; numbers, strings, other constants
         | IDENTIFIER          ; variable names
         | (IDENTIFIER <expr> ...) ; function call. In this grammar, the function
                                   ; expression **must be an identifier**
         | (if <expr> <expr> <expr>)               ; if expressions
         | (let* ([IDENTIFIER <expr>] ...) <expr>) ; let* expressions
```

Write the function `is-tail-recursive` that takes a definition (that follows the grammar for `<def>` above), and determines whether the definition is of a tail-recursive function. You may assume that we will only use `is-tail-recursive` to check recursive definitions. You may also assume that only valid expressions that are generated from the above grammar will be used to test your function.

Recall that a function is tail recursive if all recursive calls occur in tail position. Recall also that an expression is in tail position if no other work is to be done *after* that expression is evaluated.

You can use helper functions freely. We recommend that you write two helper functions:

- a helper function `all-tail-position` that takes an expression and a function name, and returns whether all calls to the function in the expression is in tail position.
- a helper function `has-call` that takes an expression and a function name, and returns whether the expression contains a call to the function.

Remember to use structural recursion like in practice lab 2, and *trust the recursion*.

How to get started

We are intentionally giving you very little starter code, so that you get practise reasoning about the problem. Staring at an empty file can be overwhelming. If you're having trouble getting started, consider how we tackled the `subst` problem from last week. What were some of the techniques that we used to help you get started?

Do you feel like you fully understand the problem we are asking you to solve? Recall that last week, to better understand the problem, we started by writing down examples of expressions in our grammar, and determined by hand what the expected output would be. *You should start here*

In this course, it is normal to spend a long time thinking, and relatively little time writing code. If you begin writing code right away, you may end up writing a lot of code that serves little to no purpose. It's okay to work with examples for a long time. It's okay to think for a long time about a very short piece of code. It's also okay to throw out large amounts of code and start over once you understand the problem better.