Exercise 2: Analyzing syntax trees, a basic recursive Haskell type

In this exercise, you'll check your understanding of two programming language concepts we covered in lecture by implementing some static code analyzers for these ideas. You'll also be introduced to how to define a Haskell type, using the familiar "list" recursive definition as an example.

Starter code

- ex2.rkt
- Ex2.hs

Task 1: Detecting tail calls

We discussed last week how recursive functions can blow the call stack if we aren't careful, and that some programming languages protect against this by implementing an optimization known as *tail call elimination*. The obvious first step to implementing such an optimization is to actually check for functions in tail call position; that's what you'll do in this task.

More formally, you'll write a function that's given a Racket datum, and returns a list of (quoted) function call expressions that appear in tail position within that datum. We can formally define the *tail call position(s)* of a basic Racket expression using structural recursion:

- An atomic value has *no* tail call position.
- An expression that is a function call is itself in tail call position. None of its subexpressions are in tail call position.
- An expression that is an `(if <cond> <then> <else>)` has a set of tail call positions: the tail call positions of the `<then>` expression, and those of the `<else>` expression (in that order).
- The tail call positions of `(and <expr1> <expr2> ...)` is the set of tail call positions of the *last* the argument expression. Assume that there is at least one argument expression. The same rule applies to `(or <expr1> <expr2> ...)` expressions.

Your function must be able to handle all expressions consisting of the (arbitrarily deeply nested cominations of) elements in the items above, but that's all. You don't need to handle other types of expressions, such as `define`, `lambda`, or `cond`.

Task 2: Strict evaluation order

As we discussed in lecture, Racket (as with most other programming languages) uses *left-to-right eager evaluation* for its function call operational semantics. More formally, a function call expression `(<f> <arg-1> ... <arg-n>)` is evaluated by doing the following:

1. Evaluate each of `<f>`, `<arg-1>`, ..., `<arg-n>`, in that order.

   NOTE: `<f>` here can be one of two things: the name of a function a function call expression (the latter being a use of a higher-order function, as you saw in Lab 2). In all cases, it must evaluate to a function expression (otherwise there's a type error).

2. Substitute the values of the arguments into the body of the function value obtained from `<f>`, and then evaluate the body.

To make sure you fully understand this ordering, your task here is to write a function that takes a Racket datum consisting only of atoms (literals or identifiers) and function call expressions, and return the order in which the *function call expressions* would be evaluated.

Task 3: Making explicit recursive types

Recall the recursive definition of a list (where in this case, every item is an integer):

- The empty list is a list.
- Given an integer `x` and list `L`, "`x` cons `L`" is a list.

While this definition is built into both Racket and Haskell, we wanted to take the opportunity to make this definition more concrete, and explore the syntax for *type definitions* in Haskell. In Haskell, we can represent this definition as follows:

```
data List = Empty
          | Cons Int List
          deriving (Show, Eq)
```

Here, `Empty` is a value representing an empty List, while `Cons` is a function that takes two arguments, an integer and a List, and returns a new List. (Ignore the `deriving (Show, Eq)` for this exercise.) For example, the expression `Cons 1 (Cons 2 (Cons 3 Empty))` is logically equivalent to:

- `1:2:3:[]` in Haskell
- `(cons 1 (cons 2 (cons 3 null)))` in Racket

Your task is to implement `numEvensList` and `mapList` for this list representation in `Ex2.hs`. We've provided some starter code to guide you on how to use *pattern-matching* to deconstruct "first" and "rest" in these lists, since the built-in list functions won't work on this new datatype.