Lab 3: A Simple Interpreter

In this lab, you'll get some practice with higher-order functions, and then use what you've learned so far to build an *interpreter* for a simplified set of Racket expressions, including basic name lookup using a hash table implementation of an environment.

Starter code

- lab3.rkt

Task 1: Representing an environment

In computer science, an *interpreter* is a program that takes another program as input and executes or evaluates the contents of that program. Of course, what we mean by "execute" or "evaluate" depends on the semantics of the programming language!

You have already written an interpreter in exercise 2 (surprise!) for a simple arithmetic language. The grammar specifying valid expressions in our language looked like this:

```
Binary Arithmetic Expression Grammar (Exercise 2)

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

While this language will allow us to express any arithmetic operation, it would be nice to be able to save intermediate results in variables. For example, we may wish to use a `let*` expression as in Racket, like this:

```
(let* ((a 2)
       (b (* a 10)))
    (+ a (* a b)))
```

This expression should evaluate to 42.

In this lab, we will add local variable bindings to our language grammar. Expressions in our language will look like this:

```
Expanded Binary Arithmetic Expression Grammar

<expr> = NUM
       | ID
       | (<op> <expr> <expr>)
       | (if (<comp> <expr> <expr>) <expr> <expr>)
       | (let* ((ID <expr>) ...) <expr>)
<comp> = = | > | <
<op>   = + | - | * | /
```

Where `ID` is an identifier (variable name). Any valid variable name in Racket will also be a valid identifier in our language. You may assume that all identifiers are Racket symbols. You can use the Racket function `symbol?` to check if an expression is an identifier.

The `let*` expressions have the same semantics as in Racket. The `...` in the let expression means that there can be zero or more bindings of identifiers to expression values.

In order to evaluate such an expression, we will need to build an **environment**.

Recall from lecture that an *environment* is a mapping of identifier names to values. For example, to evaluate the expression `(* a 10)`, we must first determine the value of `a` (or raise an error if it is unbound). In an interpreter, we say that the value of `a` must be "looked up"—the environment is an abstract representation of where this lookup occurs.

But let's turn this abstract idea into concrete code. We'll use the Racket *hash table* data type to store an environment; this data type has a similar interface to a Python dictionary or Java HashMap, with the usual restriction that we won't be using any mutating functions.

What to do

Your task here is to write an interpreter `eval-calc` based on the `calculate` function you wrote in lab 2. The function `eval-calc` will take an adidtional argument `env` representing the environment as a Racket hash table.

This interpreter `eval-calc` should have the same functionality as your previous lab, but also with the ability to *evaluate* identifiers.

In this week's exercise, we'll explore how environments are built: i.e. how to evaluate a `let*` expression. For now, focus on the cases where the expression is a `NUM`, `ID`, or `(<op> <expr> <expr>)`.