Exercise 3: Representing an Environment

This exercise is an opportunity to practice writing functions on these using structural pattern-matching, tail recursion, and pattern matching.

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/ex3/ex3.rkt`
- `Exercises/ex3/Ex3.hs`
- `Exercises/ex3/Ex3Types.hs`

Submission instructions

Submit the files `Ex3.hs` and `ex3.rkt` to MarkUs: https://mcsmark.utm.utoronto.ca/csc324f20/

But before you submit, check that your Racket code:

- 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!
- Do not modify `Ex3Type.hs`. We will be supplying our own.
- Do not modify the module export code in your Haskell file.

Task 1: Representing an environment [2 pts]

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 for a simple arithmetic language. (Surprise!) The grammar specifying valid expressions in our language looked like this:

```
Binary Arithmetic Expression Grammar (Exercise 2)

<expr> = NUM
       | (<op> <expr> <expr>)
<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 tasks 1 and 2 of this exercise, we will add local variable bindings to our language grammar. Expressions in our expanded language will look like this:

```
Expanded Binary Arithmetic Expression Grammar

<expr> = NUM
       | ID
       | (<op> <expr> <expr>)
       | (let* ((ID <expr>) ...) <expr>)
<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 the readings that an *environment* is a mapping of identifier names to *values* (review the notes on interpreters if you're not sure why this word is italicized). 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.

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. You should look up the Racket hash table documentation here: https://docs.racket-lang.org/reference/hashtables.html In particular, you should be using the functions `hash-ref` for this task, and the function `hash-set` and `hash` in the next task. **Do not use any of hash table functions that use mutation!**

For task 1, complete the `eval-calc` function to be able to evaluate binary arithmetic expressions with identifiers. You may assume that all identifiers in your expression are in the environment, and do not need to handle errors.

Task 2: Building an environment [5 pts]

For this task, we will be writing code in the `eval-calc` function to be able to evaluate expressions with `let*` bindings.

The semantics of a `let*` binding is just like in Racket. In an exression `(let* ((ID <expr1>) (ID <expr2>) ...) <body-expr>)`, we:

1. Evaluate expression `<expr1>` in the binding `(ID <expr1>)` using the current environment.
2. Bind the value from step 1 to the identifier `ID`, creating a *new* environment.
3. Evaluate the expression `<expr2>` in the binding `(ID <expr2>)` using the envrionment from step 2.
4. Bind the value from step 2 to the identifier `ID`, creating a *new, new* environment.
5. Repeat until all bindings. (Note: doesn't this sound like a perfect time to use `foldl`?)
6. Evaluate `<body-expr>` with the environment containing all the new bindings.

Modify your function `eval-calc` to handle `let*` expressions.

More about Racket Hash Tables

One of the challenges this week is understanding how to use hash tables in Racket. In particular, how can Racket "modify" a hash table without allowing mutation? The answer is that we never really "modify" a hash table. Instead, a function like `hash-set` returns a *new* hash table with all the elements of the old hash table, plus the new element.

So, Racket code like this will *not* behave like equivalent Python/Java code:

```
> (define table (hash))  ; create an empty hash table
> (hash-set table "k" 4) ; add a key "k" and value 4 to the hash table
'#hash(("k" . 4))        ; the function call returns the new hash-table ...
> (hash-ref table "k")   ; ...but doesn't mutate the old one
; hash-ref: no value found for key
;   key: "k"
; [,bt for context]
```

Instead, you will need to use the *new* hash table in subsequent computations

```
> (define table (hash))
> (let* ([new-table (hash-set table "k" 4)])
    (hash-ref new-table "k")) ; the key "k" is in the new-table!
4
```

In short, when you are using `hash-set`, make sure that you are not throwing away the new hash table!

Task 3: Calculator in Haskell [3 pts]

We will not re-implement all of task 2 in Haskell. However, the calculator grammar gives a good introduction to the *type definitions* in Haskell.

In Haskell, we can represent the definition of a binary arithmetic expression from Exercise 2 as follows:

```
data Expr = Number Float
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          deriving (Show, Eq)
```

Here, `Expr` is the name of a new data type that we defined. There are five ways of defining this data type (five different constructors), by using one of `Number`, `Add`, `Sub`, `Mul` or `Div`. (Ignore the `deriving (Show, Eq)` for this exercise.)

We use `Float` for numbers instead of `Integer` so that division works reasonably.

Now, to create an expression equivalent to `(+ 2 (- 5 1))`, we would write

```
Add (Number 2) (Sub (Number 5) (Number 1))
```

The `calculate` function should take an `Expr`, and evaluate it to return a single number.

We've provided some starter code to guide you on how to use *pattern-matching* to deconstruct the data structure.

Note: the other provided Haskell file, `Ex3Types.hs`, contains the above `Expr` definition, as well as a few helpers used for testing. Do not make any modifications to this file. You won't be able to submit this file, and we will supply our own for testing.