Exercise 4: Lambda-Calculus Interpreter

In lecture, we contrasted strict and non-strict denotational function call semantics, and saw how most programming languages perform *left-to-right eager evaluation* of a function call's arguments. We also saw that Haskell uses a different strategy to evaluate function calls, which we colloquially call "lazy evaluation" because it defers the evaluation of arguments until they are required to evaluate the function body.

In this exercise, you'll learn more about this idea by implementing an interpreter that performs eager evaluation. We're big believes in the idea that we don't really understand something until we implement it. So, this exercise should clarify the idea behind *closures* and lexical scoping, and what exactly happens when you call a function.

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/ex4/ex4.rkt`
- `Exercises/ex4/Ex4.hs`
- `Exercises/ex4/Ex4Types.hs`

Submission instructions

Submit the files `Ex4.hs` and `ex4.rkt` to MarkUs. 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 `Ex4Type.hs`. We will be supplying our own.
- Do not modify the module export code in your Haskell file.

Task 1: Calculator Eager Evaluation [5 pt]

To explore eager evaluation, let's begin by expanding the binary arithmetic grammar from the previous weeks to introduce expressions for function definition and function calls. Our new grammar is as follows:

```
Lambda Calculus with Binary Arithmetic

<expr> = NUM
       | ID
       | (<op> <expr> <expr>)
       | (let* ((ID <expr>) ...) <expr>)
```

```
       | (lambda (ID ...) <expr>)       ; function expression (function definition)
       | (<expr> <expr> ...)            ; function call
<op>   = + | - | * | /
```

Where `NUM` represents any integer literal in base 10, `ID` is a variable name (a symbol in Racket), and the arithmetic operations `+`, `-`, `*`, `/` have the standard mathematical meanings.

You already know how to evaluate numbers, identifiers and `let*` expressions from previous exercises. What remains is the evaluation of function expressions and function calls: your task is to modify the interpreter `eval-calc`, starting from the code from last week's exercise, to support valid expressions in the "Lambda Calculus with Binary Arithmetic" grammar.

As before, you can assume that expressions are semantically valid: they will *not* throw an error, and will terminate.

Evaluating Function Expressions (Defining Functions)

A function expression evaluates to a *closure*, or a data structure containing two things:

- The function expression.
- The environment at the time the function expression is evaluated. This is to implement lexical scoping.

We will store function expressions as a *list* containing three elements:

- The first element will be the token `'closure`, and will be used to identify closures during function call
- The second element will be the function body expression `<expr>`
- The third element will be the environment

For example, `(lambda (x) (+ x 1))` should evaluate to `'(closure (lambda (x) (+ x 1)) ENVIRONMENT)`.

Where `ENVIRONMENT` is the environment (hash table) at the time that the function expression is evaluated.

The idea of returning closures might be a bit strange to you. Up until now, the function `eval-calc` always returned a *number*! Now, `eval-cacl` will return either a number or a closure.

Evaluating Function Calls (Calling Functions)

A function call should be evaluated eagerly, left-to-right, like this:

- First, evaluate the function expression. (Normally, we would check that the value of the function expression is a closure. For this exercise, you may assume that the value of the function expression will always be a closure.)
- Evaluate each of the argument expressions, left-to-right
- Evaluate the function body, extending the appropriate environment with the additional argument bindings

If you are stuck, re-read the readings on "Interpreters" from last week.

Note: If you use the wrong environment in the last step, your interpreter will be dynamically scoped rather than lexically scoped.

Task 2: Calculator in Haskell [5 pt]

We will write the same interpreter in Haskell. One advantage of working in Haskell is Haskell's type system: Haskell's type system makes it clear that `evalCalc` takes an *expression* and an *environment*, and returns a *value*. (One common difficulty that students experience is differentiating between an expression and a value.)

But Haskell's type system means that our expression data structure needs to be a bit more verbose. To get you used to the data structure and review pattern matching, complete the two **warmup tasks**, which is worth 1 pt total.

We have written some of the `evalCalc` code for you. The only thing you need to do is determine how to evaluate let bindings, function expressions and function calls. Use your solution in task 1 to help you. Or, alternatively, start here with Task 2, and use your solutions to task 2 to help you with task 1.