CSC324 Assignment 3: Continuation Passing Style and Delimited Continuations

For all labs and assignments, 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 lab 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.

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 ...)` and `module (...) where` code in your files are 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.
- Do not change the default Haskell language settings (i.e. by writing an additional line of code in the first line of your file)

Overview

For this assignment, we will use the idea of *continuations* in a slightly different context, and explore a style of programming called the **Continuation Passing Style (CPS)**. In the warmup task, we start by transforming recursive Haskell functions to use this programming style. In the main task, we will move on to transforming an interpreter to use CPS, so that continuations are accessible at each step of evaluation. We will use these continuations to **implement Shift and Reset** in this interpreter.

Starter code

- `A3.hs`
- `A3Types.hs`
- `A3StagShell.hs`
- `A3StarterTests.hs`

You will only be submitting `A3.hs` and not any of the other files. Do not make any modifications to `A3Types.hs`. We'll be supplying our own version to test your code. The file `A3StagShell.hs` is provided to you as a reference only.

Continuation Passing Style

Continuation Passing Style (CPS) is a style of programming where the control flow is made explicit. A function written in CPS takes its continuation as an extra parameter. When the function completes its computation, it "returns" the output by calling the continuation with the computed result.

As an example, here is a function `is3or5` written in **direct style** that checks if an integer is either a 3 or a 5.

```haskell
is3or5 :: Int -> Bool
is3or5 x = (x == 3) || (x == 5)
```

In contrast, here is the function `cpsIs3or5` that is the same function written in CPS. In other words, `cpsIs3or5` is the CPS transformed version of `is3of5`. This new function takes an extra parameter, which represents the continuation: what to do *after* `cpsIs3or5` is finished. The last thing that is done in `cpsIs3or5` is to *return* the boolean value by calling the continuation (usually represented by a variable called `k`):

```haskell
cpsIs3or5 :: Int -> (Bool -> r) -> r
cpsIs3or5 x k = k $ (x == 3) || (x == 5)
```

(Note about syntax: recall that in Haskell, the expression `a $ b c` is equivalent to `a (b c)`)

If there is nothing else to do after `cpsIs3or5`, we can pass in the identity function as the argument to `k` and obtain the result:

```
Prelude> cpsIs3or5 4 (\x -> x)
False
```

Converting simple functions like `is3or5` to use CPS is quite straightforward. However, converting recursive functions to CPS requires a bit more work. Consider this function, which inserts an element into a sorted list in the correct position:

```haskell
insert :: [Int] -> Int -> [Int]
insert []     y = [y]
insert (x:xs) y = if x > y
    then y:x:xs
    else x:(insert xs y)
```

To convert this function to use CPS, we need to work with the recursive call a little more carefully. In particular, in the recursive case, we will call `cpsInsert`, and give it the appropriate continuation that prepends `x` in the right place.

```haskell
cpsInsert :: [Int] -> Int -> ([Int] -> r) -> r
cpsInsert []     y k = k [y]
cpsInsert (x:xs) y k = if x > y
    then k (y:x:xs)
    else cpsInsert xs y $ \res -> k (x:res)
```

In the "then" branch, of this function, we first pre-pend `y` to the list `(x:xs)`. Then, we call the continuation `k` with the result, since `k` is a function describing what to do *after* the computation.

In the "else" branch, we need to put `x` in front of the list that we get from inserting `y` into `xs`. In other words, we need to do some work *after* obtaining a result from recursively calling `cpsInsert`. Notice that the prepending operation `x:` is placed in the continuation argument `(\res -> k (x:res))` of the recursive call.

Notice that in none of the cases do we ever allow a function to return to its caller. In other words, the **last thing** to happen in a function is either a **call to the continuation**, or a **tail call**. Nothing else can happen after the call to the continuation, or after a tail call.

Warmup Task. CPS Transforming Haskell Functions

Write the following functions using CPS. You may find it helpful to start by writing the functions in direct style (without using tail recursion), and then writing the function again in CPS:

- `cpsFactorial`: to compute the factorial of a number
- `cpsFibonacci`: to compute the n-th Fibonacci number
- `cpsLength`: to compute the length of a list
- `cpsMap`: to apply a function (written in direct style) to every item of a list
- `cpsMergeSort`: to sort a function using merge sort. Your function *must* call the two helper functions `cpsSplit` and `cpsMerge`
- `cpsSplit`: helper function for cpsMergeSort, to split a list into two lists. All list elements at even indices are placed in one sub-list, and all list elements at odd indices are placed in the second sub-list.
- `cpsMerge`: helper function for cpsMergeSort, to merge two sorted lists.

Some of these warmup task solutions will be provided to you during the practical session in week 9 (Nov 9).

Please include your warmup task code in your solutions, since some of the code may be graded.

Main Task. CPS Transforming The StagShell

For the second task, we will revisit the StagShell language that we wrote in Assignment 2. In particular, we will CPS transform an interpreter for StagShell, and introduce delimited continuations to StagShell.

An interpreter for StagShell is provided to you in `A3StagShell.hs`. Notice that in `A3StagShell.hs`, the `Expr` data type is identical to Assignment 2. Moreover, the `Value` data type now handles closures a little differently: closures are represented using a Haskell *function*. This Haskell function takes a list of arguments to be passed to the StagShell function, and adds the parameter-to-argument bindings to the environment from when the `Lambda` was created. This definition of closures leverages Haskell's static scoping to retain the reference to this environment.

Two functions are written for you in the `A3StagShell.hs` file:

- `eval`, which is an interpreter for the StagShell language. You should understand this interpreter.
- `def`, which takes a list of names to expression bindings and creates an environment. Although it is just meant to make testing easier, it uses Haskell's lazy evaluation in an interesting way to allow recursive definitions.

Like other functions we worked with in the warmup task, the interpreter `eval` is a recursive function that can be written in CPS! Your main task in this assignment is to do exactly that.

In `A3.hs`, write a function `cpsEval`, which is the function `eval` written in CPS (plus a bit more, which we'll describe below).

The file `A3Types.hs` defines the data structures for the variation of StagShell language we will use for `cpsEval`. These type definitions are almost identical to that of `A3StagShell.hs`, but with two differences:

First, **closures are represented differently**. In particular, calling `cpsEval` on a `Lambda` expression will create closures represented as Haskell functions *written in CPS*. Please pay special attention to how we are representing closures.

Second, **cpsEval can support two new expression types**. Since `cpsEval` has access to a program's continuation at any point, we can support two new expression types: `Shift` and `Reset`. The semantics of these expressions should be identical to the Racket counterparts:

- `Shift` takes a name and an expression. It binds the name to the current continuation delimited by the most recent enclosing `Reset`, then evaluates the body expression. The current continuation is not applied to the result.
- `Reset` takes an expression. It acts as a delimiter to `Shift`. Note that if no `Shift` is evaluated during the evaluation of the `Reset` expression, then that `Reset` doesn't really do anything. If the body of a `Reset` evaluates to an error, then it should return the error with no further computation.

Your task is to complete the CPS version of the interpreter `cpsEval`. The behaviour of the CPS version of the interpreter should be identical (or almost identical) to `eval`. To help you get started, the `(Literal v)` and `(Lambda params body)` cases are provided to you.

The way that your CPS interpreter handles errors should be identical to Assignment 2 (except we can no longer check that the parameter and argument lengths are equal). One key difference between the CPS interpreter and the non-CPS counterparts is that *propagation of errors is no longer necessary*! In other words, if we encounter an error (e.g. attempting to call `First` on an expression that doesn't evaluate to a list), we can return the error *without* calling the continuation! This is one of the advantages of modeling continuations directly.

Additional Hints:

- Start by looking at the different pattern matching cases in the implementation of `eval` in `A3StagShell.hs`. Start with the simplest ones: `Literal`, `Plus`, `Times`, etc. The last three cases are the most challenging. For `If` expressions, check if your solution is consistent with `cpsInsert` in this handout.
- When writing `cpsEval`, you may find evaluating function applications challenging. In particular, evaluating the list of arguments is tricky. You may find it helpful to write a helper function to evaluate the list of arguments.
- If you are stuck on function application, start by assuming that functions will have at most 3 parameters. That way, you can get partial credit even if you don't figure out how to handle functions with an arbitrary number of arguments. (You may wish to move on to implementing `Shift` and `Reset` first.)