Exercise 10: Continuation Passing Style

For this week's exercise, we'll revisit the idea of *continuations* in a slightly different context, and explore a style of programming called the **Continuation Passing Style (CPS)**. In part 1, we start by transforming recursive Haskell functions to use this programming style. In parts 2, we'll move on to transforming an interpreter to use CPS, so that continuations are accessible at each step of evaluation. As a bonus, we'll **implement Shift and Reset** in this interpreter!

Deadline: November 26th, 2019 before 10:00pm

Starter code

- `Ex10.hs`
- `Ex10Bork.hs`
- `Ex10Types.hs`

You will only be submitting `Ex10.hs` and not any of the other files. Do not make any modifications to either `Ex10Bork.hs` or `Ex10Types.hs`. We'll be supplying our own version to test your code.

Continuation Passing Style

Continuation Passing Style (CPS) is a style of programming where the control flow is passed explicitly. A function written in CPS style 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.

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

In contrast, here is the function `cpsIs3or5` that is the same function written in CPS style. 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 called. 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` or `c`):

```
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:

```
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.

```
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)
```

Task 1. 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. You function *must* call the two helper functions `cpsSplit` and `cpsMerge`
- `cpsSplit`: helper function for cpsMergeSort, to splits a list into two lists. All list elements in even indices is placed in one sub-list, and all list elements in odd indices is placed in the second sub-list.
- `cpsMerge`: helper function for cpsMergeSort, to merge two sorted lists.

Task 2. CPS Transforming The Bork Interpreter

For the second task, we introduce a language called "Bork". The expressions and values in this language are described in `Ex10Bork.hs`. Additionally, two functions are written for you:

- `eval`, which is an interpreter for the Bork language
- `def`, which takes a list of name to expression bindings and creates an environment

Interestingly, the interpreter `eval` is really just a recursive function! Like other functions we can write `eval` in CPS! Your task in this part of the assignment is to do exactly that. In `Ex10.hs`, write a function `cpsEval`, which is the function `eval` written in CPS (plus a bit more, which we'll describe below).

The file `Ex10Types.hs` defines the data structures for the Bork language we will use for `cpsEval`. These type definitions are almost identical to that of `Ex10Bork.hs`, but with two differences:

First, **the type signature of (CPS) Bork procedures/closures** is different. In particular, `cpsEval` will create procedures represented as Haskell procedures *written in CPS*.

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`. To earn full marks for this task, you do not need to correctly implement the interpreter behaviours for `Shift` and `Reset` expression. However, your implementation of `cpsEval` should still pass the Haskell type checker.

The behaviour of `cpsEval` should be almost the same as `eval`, with the exception that in `cpsEval`, if an `Error` is produced, the error is returned immediately without calling the continuation.

Hints:

- Start by looking at the different pattern matching cases in the implementation of `eval` in `Ex10Bork.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 procedure 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 necessary, you can assume that we will only test with functions with at most 5 parameters.

Bonus Task: Shift and Reset (1% bonus)

For bonus points, implement `cpsEval` to support `Shift` and `Reset` expressions. The behaviour of `Shift` and `Reset` should be identical to what you see in Racket:

- `Shift` takes a name and an expression. It binds the name to the current continuation, 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 there are no `Shift` in an expression, then `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.

Food for thought

Our treatment of logic programming via the ambiguous choice operator `-<` depended heavily on delimited continuations. But delimited continuations (shift and reset) are not frequently supported as language features. This exercise shows that we are able to replicate the behaviour of `shift` and `reset` by explicitly capturing continuations. This is one of the reasons why CPS transformation can be a powerful and interesting tool.