CSC324 Assignment 3: Continuation Passing Style and Delimited Continuations

For this assignment, we will use the idea of *continuations* in a slightly different context than in lecture 9, and explore a style of programming called the **Continuation Passing Style (CPS)**. In the first warm up task, you extend the "Racket<->Haskell" transpiler you wrote in the A2 warmup task to handle shift/reset expressions (more on this later). In the second warm up 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`
- `A3Orange.hs`
- `A3StarterTests.hs`

Submit only the file `A3.hs` and not any of the other files. Do not make any modifications to `A3Types.hs`. We will be supplying our own version to test your code. The file `A3Orange.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 (represented as a function) as an extra parameter. This continuation contains the work that should be done *after* the function has completed. 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. This 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 the symbol `k`):

```
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
```

Or, perhaps the task to be done after `cpsIs3or5` is to negate the result:

```
Prelude> cpsIs3or5 4 not
True
```

Passing the continuation to a function gives the function additional control: a function could choose to, for example, ignore the continuation, or do something else:

```
cpsContinueIf5 :: Int -> (Int -> Maybe r) -> Maybe r
cpsContinueIf5 x k = if (x == 5) then (k x) else Nothing
```

Converting Functions to CPS

Converting simple functions like `is3or5` to use CPS is quite straightforward: just treat `k` as you would the `return` keyword in some languages! 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. We need to carefully analyze what operations happen before the recursive call, and what operations happen *after*. Operations that happen after the recursive call goes inside the continuation function passed to the recursive call.

```
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)
```

The "then" branch is fairly straightforward. 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 this function (again, think of `k` as the "return" operation).

In the "else" branch, notice that the prepending of `x` happens *after* the recursive call. Operations that happen after the recursive call goes inside the continuation function. Thus, the continuation function passed to the recursive `cpsInsert` performs the `(x:res)` operation. The list `(x:res)` is then passed as parameter to the continuation of the original `cpsInsert` call.

In none of the cases do we ever allow a function to return to its caller. In other words, **all recursive calls to functions written in CPS are in tail position**. 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!

What is CPS good for?

As mentioned earlier, functions written in CPS inverts control, and allows a function to decide what to do with its continuation. This allows some functions to optionally exit early:

```
divide :: Float -> Float -> (Float -> Maybe r) -> Maybe r
divide x 0 k = Nothing -- error! no need to call the continuation
divide x y k = k (x / y)
```

It also allows functions to manipulate continuations, much like we see in Racket's "shift" and "reset". In fat, we will implement "shift" and "reset" in our language Orange in in the main task of this assignment.

CPS is actually closer to machine code (you may notice the similarity to `goto`). Compiled functional languages therefore sometimes use CPS as an intermediate form.

Warm up Task 1. Extending the A2 warm up task

Recall that in the A2 warm up task, you wrote a Haskell function `racketifyExpr` which turns Orange expressions into equivalent Racket programs and a Racket function `haskellify` which turns a Racket program into an equivalent Orange expression.

This assignment adds two new expressions `Shift` and `Reset` to Orange expressions. In this warm up task, you extend the solution to the A2 warmup tasks to handle these new expression types.

If you finished the warmup task in A2, then you can copy that code over and just add two (straightforward) new cases in the Haskell and Racket functions to handle Shift/Reset.

If you did not do the warmup task in A2, we again highly encourage you to do it now. Evaluating programs with continuations can be extremely unintuitive and having a reference implementation to compare against (e.g. Racket) will be invaluable.

You can use the functions just like you did in A2, but you need to be careful about controlling the "ambient" continuation. In general, if `r` is the Racket program outputted by `racketifyExpr h`, then `(reset r)` is equivalent to `cpsEval empty h id`. Likewise, if `h` is the Haskell data structure outputted by `haskellify r`, then `cpsEval empty h id` is equivalent to `(reset r)`. For example,

```haskell
putStrLn $ racketifyExpr
    (Plus (Literal (Num 1))
          (Shift "k"
             (Plus
                (App (Var "k") [(Literal (Num 1))])
                (App (Var "k") [(Literal (Num 2))]))))
-- prints (+ 1 (shift k (+ (k 1) (k 2))))
```

In Racket, you can run the above expression as

```
;; this line imports shift and reset. don't forget it!
(require racket/control)
(reset (+ 1 (shift k (+ (k 1) (k 2)))))
;; outputs 5
```

Conversely,

```
(display (haskellify '(+ 1 (shift k (+ (k 1) (k 2))))))
;; outputs
;;    (Plus (Literal (Num 1))
;;          (Shift "k"
;;             (Plus
;;                (App (Var "k") [(Literal (Num 1))])
;;                (App (Var "k") [(Literal (Num 2))]))))
```

In a correct solution to A3, you should see

```haskell
cpsEval Data.Map.empty
      (Plus (Literal (Num 1))
          (Shift "k"
             (Plus
                (App (Var "k") [(Literal (Num 1))])
                (App (Var "k") [(Literal (Num 2))]))))
      id
-- outputs (Num 5)
```

Continuing the policy from A2: **we will not answer questions about test cases that can be answered via this warm-up task.**

Warm up Task 2. 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.

Include your warm up task code in your solutions, since some of the code will be graded.

We have set up Markus so that you can check your warm up tasks, but it can be difficult to ensure that your code actually is in CPS. Check to make sure that all recursive calls are in tail position!

Main Task. CPS Transforming the Orange Interpreter

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

A Orange interpreter in direct style

An interpreter for Orange is provided to you in `A3Orange.hs`. In this file the `Expr` data type is identical to Assignment 2. However, the `Value` data type now handles closures a little differently: closures are represented using a *Haskell function* that takes a list of arguments (type `[Value]`) and produce a result (type `Value`). By leveraging Haskell's implementation of lexical scoping, the quantities that were stored explicitly in A2 (parameter list, body expression, environment) are available inside this function.

An instance of such a Haskell function (of type `[Value] -> Value`) is created when evaluating a `Lambda`. You can see in `A3Orange.hs` that inside such a function, we check if the length of the arguments matches the length of the parameters, and if so extend the environment from when the `Lambda` was created, and then evaluate the body. This Haskell function can then be called when evaluating an `App`: first, the argument expressions are evaluated, then the values are passed to this function from the closure. (If this part is still confusing, reference the code and pay special attention to what operations are happening **in Haskell** vs **in Orange**.)

Two more functions are written for you in the `A3Orange.hs` file:

- `eval`, which is an interpreter for the Orange 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 warm up 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.

Your Task

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 Orange language we will use for `cpsEval`. These type definitions are almost identical to that of `A3Orange.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*. In particular, the type signature of the Haskell function representing a `Closure` in `A3Types.hs` is now: `([Value] -> (Value -> Value) -> Value)`

In addition to the list of arguments (type `[Value]`), a Orange closure also needs to take as parameter a Haskell continuation (function of type `Value -> Value`) representing the computation to be done *after* the Orange function application—i.e. after the Orange function body is evaluated, what other evaluation steps need to be performed?

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`. Evaluating continuation-based program can be difficult and unintuitive, which is why warmup task 1 will be extremely valuable.

To help you get started, an incomplete version of the `(Literal v)` and `(Lambda params body)` cases are provided to you. **However, the provided code for these cases do not yet handle errors, so you will need to modify them!**

The way that your CPS interpreter handles errors should be identical to Assignment 2. One key difference between the CPS interpreter and the non-CPS counterparts is that when we encounter an error, we can return the error without calling the continuation! This will cut down on potential case analyses significantly.

Additional Hints:

- Start by looking at the different pattern matching cases in the implementation of `eval` in `A3Orange.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` and handling the function application case, evaluating the list of arguments is tricky. You may find it helpful to write a helper function to evaluate the list of arguments. (The `cpsMap` function you wrote in the warm up task assumes that the function being applied is written in direct style, and is therefore not sufficient.)
- 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. We recommend moving on to implementing `Shift` and `Reset` before tackling arbitrary argument function applications.

Submission and Instructions

Submit only the file `A3.hs` on Markus. We will be supplying our own version of `A3Types.hs` to test your code.

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 Markus (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)