A3: Continuation Passing Style

Assignment 3

In Assignment 3, we will learn to write recursive functions in **continuation passing style**.

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

```
-- Function written in direct style
is3or5 :: Int -> Bool
is3or5 x = (x == 3) || (x == 5)

-- Function written in CPS
cpsIs3or5 :: Int -> (Bool -> r) -> r
cpsIs3or5 x k = k $ (x == 3) || (x == 5)
```

Continuation Passing Style

Why CPS?

- So that the programmer can make decisions about the control flow, rather than the language designer
- Additionally, in A3, we will rewrite the StagShell interpreter in CPS!

Why CPS an *interpreter*?

- To build an interpreter that supports delimited continuation operations Shift and Reset, which you will learn about in lecture 9
- Then we can add constructs like exceptions, backtracking, generators, etc. that we will use in lectures 9 and 10.

Golden Rule of CPS

No procedure is allowed to return to its caller—ever.

from http://matt.might.net/articles/by-example-continuation-passing-style/

> *The **last thing** to happen in a function is either a **call to the continuation**, or a **recursive call**. Nothing else can happen after the call to the continuation, or after a recursive call.*

CPS is like an extreme version of tail recursion.

Example: Identity Function

```
id :: a -> a
id x = x
```

Add a continuation *k*:

```
cps_id :: a -> (a -> r) -> r
cps_id x k = k x
```

(Think of *k* as the *callback* or *return* function)

Example: is3or5

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

Add a continuation k:

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

Example: Insertion

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

Add a continuation k:

```
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))
```

Rules to CPS a variable in a function body

To CPS a **variable**, apply k to the variable

```
id      x   = x

cps_id x k = k x
```

Rules to CPS a function call in a function body

To CPS a **function call**, move the continuation of the function call into its last argument:

```
f      x    = 1 + (g x)

cps_f x k = cps_g x (\result -> k (1 + result))
```

. . . because the continuation of `(g x)` in the expression `1 + (g x)` is `(\result -> 1 + result)`!

Rules to CPS a function call in a function body

To CPS a **function call**, move the continuation of the function call into its last argument:

```
f      x    = 1 + (g x)

cps_f x k = cps_g x (\result -> k (1 + result))
```

. . . because the continuation of `(g x)` in the expression `1 + (g x)` is `(\result -> 1 + result)`!

Another example:

```
f      x    = (g x) + (h x)

cps_f x k = cps_g x (\gx ->
              cps_h x (\hx ->
              k (gx + hx)))
```

Rules to CPS an if expression in a function

To CPS an **if expression**, first CPS the condition. Then, CPS the "then" and "else" branches separately.

```
f      x    = if (g x) then (h x) else (j x)

cps_f x k = cps_g x (\gx ->
              if gx
              then (cps_h x k)
              else (cps_j x k))
```

Example: Function Arguments

```
f      x    = (g (h x) (j x))

cps_f x k = cps_h x (\hx ->
              cps_j x (\jx ->
              cps_g hx jx k))
```

Let's CPS this together

```haskell
fac :: Int -> Int
fac 0 = 1
fac n = n * (fac (n - 1))

cpsFac ::
```