3 Week of 02/27: curried functions

In this recitation we spent some time playing around with simple examples of curried functions and how to combine/manipulate them in order to get more comfortable with the idea of currying. Understanding this deeply will really come in handy once we start studying Haskell, since multi-argument functions in Haskell are curried by default! We can start with the definitions of the `curry` and `uncurry` functions:

```
(define curry
  (lambda (f)
    (lambda (x) (lambda (y) (f x y)))))

(define uncurry
  (lambda (f)
    (lambda (x y) ((f x) y))))
```

The `curry` function is a higher-order function that accepts a single function of two arguments (or a variadic function that is *capable* of taking two arguments in addition to other numbers of arguments - for instance, `(curry +)` will work) and returns a curried version of that function. You can think of the curried version of a two-argument function as supporting **partial evaluation**, that is, it's possible to supply "one argument at a time" to a curried function, leaving the other arguments "indeterminate". Let's look at a simple example with the addition function `+`. We can define a curried version of the function like this:

```
(define +c (curry +))
```

This new function `+c` *is a function of one argument.* To be completely explicit about how this function behaves - it takes a single number as its input, and returns a function as output, and that output function has the behavior of taking a single number as input and returning a number as output. We can use this function to add two concrete numbers, like this:

```
((+c 2) 3)
```

which, of course, will evaluate to `5` - the subexpression `(+c 2)` evaluates to a function which adds `2` to its input, and that function returns `5` when passed the input `3`. However, we can also manipulate the partially evaluated results of this curried function. For instance, we can add `2` to each element of a list using `map` as follows:

```
(map (+c 2) '(1 2 3 4 5 6 7))
```

Note that, in the past, we might have accomplished something like this by building a lambda function `(lambda (x) (+ x 2))` and mapping it onto a list. Behaviorally, that's *exactly* what this code accomplishes - if you take a look at the definition of `curry`, you will see that the function that we receive from evaluating `(+c 2)` is essentially the same as this lambda function, but we're now able to construct it with less hassle. We can also consider interesting expressions like this:

```
(compose (+c 3) (+c 5))
```

Since `(+c 3)` behaves like a "plus three" function, and `(+c 5)` behaves like a "plus five" function, *composing* these two functions gives us a function that will add *eight* to any given number. That is, `(compose (+c 3) (+c 5))` behaves just like `(+c 8)`.

Here are two more simple functions whose curried forms are fun to play with:

```
(define fst (lambda (x y) x))
(define snd (lambda (x y) y))
```

The function `fst` takes two arguments, and returns the first one, whereas `snd` takes two arguments and returns the second one. Let's think about what the curried versions of these two functions do, starting with `(curry fst)`. This function will accept one argument `x`, and return a *function* that accepts one argument `y`, and returns `x`. We might denote this behavior more compactly like this:

$$x \mapsto (y \mapsto x)$$

Notice that the function that we get from `((curry fst) x)` is always a *constant* function - no matter what input we pass it, we always get back the value of `x`. So, an appropriate name for this function might be `const`:

```
(define const (curry fst))
```

This function takes a value as input and gives back *the constant function for that value*. Now, let's think about how `(curry snd)` behaves. Given any input `x`, it returns a function that takes an input `y` and gives back that same value `y`. In other words:

$$x \mapsto (y \mapsto y)$$

Notice that the function $y \mapsto y$ that takes an input and returns that same input value already has a name - it's called the *identity function*, and can be written as `identity` or `(lambda (x) x)` in Racket. So what we've just described is a function that takes any input and returns *the identity function*:

$$x \mapsto \texttt{identity}$$

This means that `(curry snd)` actually behaves the same as `(const identity)` - can you see why?

The function `(const const)` has some interesting behavior too. As we discussed, `(const x)` will always give us a constant function that always evaluates to `x`, so `(const const)` gives us a constant function that always evaluates to `const`:

$$x \mapsto \texttt{const}$$

or, since `const` is the function $y \mapsto (z \mapsto y)$,

$$x \mapsto \big(y \mapsto \big(z \mapsto y\big)\big)$$

In other words, `(const const)` is a function that *takes three curried arguments in succession and returns the third one.* A good descriptive name for this function might be `2nd-of-3`. The function `(const (const const))` can be described in similar terms: it is the constant function that always returns the function `(const const)`:

$$w \mapsto (\texttt{const const})$$

or, in other words,

$$w \mapsto \big(x \mapsto \texttt{const}\big)$$

or

$$w \mapsto \big(x \mapsto \big(y \mapsto \big(z \mapsto y\big)\big)\big)$$

so it would be apt to call this function `3rd-of-4`, since it takes four curried arguments and returns the third one. In fact, we might define a whole sequence of curried functions similarly:

```
(define 1st-of-2 const)
(define 2nd-of-3 (const 1st-of-2))
(define 3rd-of-4 (const 2nd-of-3))
(define 4th-of-5 (const 3rd-of-4))
...
```

Here is another higher-order function that's fun to mess with:

```
(define swapc (lambda (f) (lambda (y) (lambda (x) ((f x) y)))))
```

This function allows us to swap the order in which a curried function takes its arguments. To be specific, if `f` is a function that takes at least two arguments in curried fashion (i.e. a function of one argument that returns a function of one argument) then `(swapc f)` will return the curried function that accepts the first two arguments in reverse order, so that `((f x) y)` and `(((swapc f) y) x)` always evaluate to the same thing. We can illustrate how this function works by considering the curried version of the exponential function `expt` in Racket:

```
(define expc (curry expt))
```

The built-in function `expt` takes two arguments (not curried) a and b and returns a^b. The partially evaluated results of this curried function give us "powers of x" functions, for instance these two functions:

```
(define power2 (expc 2))
(define power3 (expc 3))
```

are the functions $n \mapsto 2^n$ and $n \mapsto 3^n$ respectively, that give powers of two and powers of 3. The partially-evaluated results of `(swapc expc)` are different, though:

```
(define square ((swapc expc) 2))
(define cube ((swapc expc) 3))
```

These are the functions $x \mapsto x^2$ and $x \mapsto x^3$, respectively. We can also use `swapc` to modify some of the variations on the constant function that we saw earlier. For instance, `(swapc 2nd-of-3)` gives us back a function that accepts three curried arguments and returns the *first* one (since the roles of the first and second arguments are swapped). One of the lab exercises was to write a function called `4th-of-7` that generalizes this pattern by writing a function accepting `7` curried arguments and returning the fourth one, which can be accomplished like this:

```
(define 4th-of-7 (const (const (const (swapc (const (swapc (const const))))))))
```

This solution can be better understood by considering each of the nested subexpressions involved and understanding what they do:

```
(define 2nd-of-3 (const const))
(define 1st-of-3 (swapc (const const)))
(define 2nd-of-4 (const (swapc (const const))))
(define 1st-of-4 (swapc (const (swapc (const const)))))
(define 2nd-of-5 (const (swapc (const (swapc (const const))))))
(define 3rd-of-6 (const (const (swapc (const (swapc (const const)))))))
(define 4th-of-7 (const (const (const (swapc (const (swapc (const const))))))))
```

Finally, let's talk a little bit about curried function composition:

```
(define compc (curry compose))
```

Just as we coukd think of `(+c x)` as an "add x" function, we can think of `(compc f)` as a "compose with f" function. However, unlike with the operation of addition, the order in which we compose two functions matters, so it's important to know that `(compc f)` gives us a *post-composition* function. That is, it acts on functions g like this:

$$g \mapsto f \circ g$$

For clarity, I like to refer to this function as $\texttt{postComp}_f$:

$$\texttt{postComp}_f \;=\; \left(g \mapsto f \circ g\right)$$

To get familiar with how these functions work, we might consider some more examples involving mapping. For instance, consider how the following expression is evaluated:

```
(map (compc add1) (map +c '(1 2 3 4 5)))
```

Evaluating the subexpression `(map +c '(1 2 3 4 5))` maps the curried addition function `+c` onto the list `'(1 2 3 4 5)`, giving us a five-element list whose elements *are themselves functions*, namely the "add one" function, the "add two" function, and so on. Then, mapping `(compc add1)` (the post-composition function that we're also denoting $\texttt{postComp}_{\texttt{add1}}$) onto this list post-composes each of these functions with the `add1` function. Composing "add one" with "add one" gives us "add two", composing "add one" with "add two" gives us "add three", and so on - so the final result will be a list of five functions, which are the "add two" through "add six" functions.

If we're denoting the post-composition function by a fixed function f as $\texttt{postComp}_f$, then we can represent the behavior of the curried composition function `compc` like this:

$$f \mapsto \texttt{postComp}_f$$

i.e. `compc` transforms any function into the function which realizes *post-composition by that function.* Using this perspective, we can try to solve the second lab exercise, which was to write a curried "compose three" function `comp3` only in terms of `compc`, that is, a curried function of three arguments f, g, h that returns $f \circ g \circ h$. That is, we want to write a function that carries out

$$\texttt{comp3} \ = \ f \mapsto \big(g \mapsto \big(h \mapsto f \circ g \circ h\big)\big)$$

The function `compc` that we have at our disposal is capable of composing two functions, like this:

$$\texttt{compc} \ = \ f \mapsto \big(g \mapsto f \circ g\big)$$

or, equivalently, maps each function to its corresponding post-composition function, as we discussed before. Let's see if we can manipulate the desired function so that it can be expressed just in terms of `compc`. First, have a look at what the partially-evaluated result of `comp3` looks like when we pass in just two of the three arguments f and g:

$$((\texttt{comp3 f})\ \texttt{g}) \ = \ h \mapsto f \circ g \circ h$$

Since function composition is associative, this is the same as $(f \circ g) \circ h$. In other words, $((\texttt{comp3 f})\ \texttt{g})$ is just *a post-composition function* by $f \circ g$:

$$((\texttt{comp3 f})\ \texttt{g}) \ = \ \texttt{postComp}_{f \circ g}$$

or:

$$\texttt{comp3} \ = \ f \mapsto \big(g \mapsto \texttt{postComp}_{f \circ g}\big)$$

Now, since `compc` is precisely the function that maps a function to its corresponding post-composition function, this is the same as writing

$$\texttt{comp3} \ = \ f \mapsto \big(\texttt{compc} \circ \big(g \mapsto f \circ g\big)\big)$$

or, since $g \mapsto f \circ g$ is just the post-composition by f function:

$$\texttt{comp3} \ = \ f \mapsto \big(\texttt{compc} \circ \texttt{postComp}_f\big)$$

Now we can manipulate this expression as follows:

$$\begin{aligned}
&\texttt{comp3} = f \mapsto \big(\texttt{compc} \circ (\texttt{compc}\ f)\big) \\
&\texttt{comp3} = f \mapsto \texttt{postComp}_{\texttt{compc}}(\texttt{compc}\ f) \\
&\texttt{comp3} = \texttt{postComp}_{\texttt{compc}} \circ \texttt{compc} \\
&\texttt{comp3} = (\texttt{compc compc}) \circ \texttt{compc} \\
&\texttt{comp3} = ((\texttt{compc (compc compc)})\ \texttt{compc})
\end{aligned}$$

Hence, we can define our `comp3` function in Racket as follows:

```
(define comp3 ((compc (compc compc)) compc))
```

This is a really tough puzzle, and I realize that the switching between Racket syntax and mathematical notation can be tough to follow. See if you can convince yourself of each of the steps/manipulations that we've used here to understand why this implementation works. We can actually generalize this further by writing a function called `comp4` that composes *four* given functions in curried fashion:

```
(define comp4 ((compc (compc (compc compc))) comp3))
```

Can you figure out how to generalize this to five or more functions?