wada

Higher Order function

Definition

Haskell functions can take functions as parameters and return functions as return values. A function that does either of those is called a higher order function. Higher order functions aren't just a part of the Has[OBJ][OBJ][OBJ]kell experience, they pretty much are the Haskell experience. It turns out that if you want to define computations by defining what stuff *is* instead of defining steps that change some state and maybe looping them, higher order functions are indispensable. They're a really powerful way of solving problems and thinking about programs.

Example

```
mapping1 :: (a ->b) -> [a] -> [b]
mapping2 :: (a ->b) -> ([a] -> [b])
filtering1 :: (a -> Bool) -> [a] -> [a]
filtering1 :: (a -> Bool) -> ([a] -> [a])
zippingWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Associativity

Associativity is a property of an operator describing how multiple usages of the same operator are grouped in the absence of parentheses. In other words, associativity describes how brackets are placed when you don’t write them explicitly. This property applies to binary operators.

Let’s look at an example to understand better what associativity means. If you have a binary operator ◯ and you write the following expression:

a ◯ b ◯ c ◯ d

There are several ways how () can be placed:

1. ((a ◯ b) ◯ c) ◯ d
2. a ◯ (b ◯ (c ◯ d))

Depending on the resulting order of parentheses, if you don't write them explicitly, we can define the associativity type of an operator:

Left-associativity

If () are placed like in option 1, the operator is called left-associative (the brackets are accumulated on the left).

Right associativity

If () are placed like in option 2, the operator is called right-associative (the brackets are accumulated on the right).

Example:

1) - has left associativity. Which means that 1 - 2 - 3 - 4 gets parsed as ((1 - 2) - 3) - 4 = -8 and not as 1-(2-(3-4)) = -2
2) : has right associativity. Which means that 1 : 2 : 3 : 4 : [] get parsed as 1 : (2 : (3 : (4 : [])))
3) -> has right associativity. Which means that a -> b -> c -> d gets parsed as a -> (b -> (c -> d))

Curried function

Definition

Every function in Haskell officially only takes one parameter. So how is it possible that we defined and used several functions that take more than one parameter so far? Well, it's a clever trick! All the functions that accepted *several parameters* so far have been **curried functions**. What does that mean? It means that the right associativity has been applied on the (->) operator.

wada

f :: a -> b -> c is the same as f :: a -> (b -> c)

Example:

Let's take our good friend, the max function. It looks like it takes two parameters and returns the one that's bigger. Doing max 4 5 first creates a function that takes a parameter and returns either 4 or that parameter, depending on which is bigger. Then, 5 is applied to that function and that function produces our desired result. That sounds like a mouthful but it's actually a really cool concept. The following two calls are equivalent:

```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

Putting a space between two things is simply function application. The space is sort of like an operator and it has the highest precedence. Let's examine the type of max. It's `max :: (Ord a) => a -> a -> a`. That can also be written as `max :: (Ord a) => a -> (a -> a)`. That could be read as: max takes an a and returns (that's the ->) a function that takes an a and returns an a. That's why the return type and the parameters of functions are all simply separated with arrows.

Partially applied function

Simply speaking, if we call a function with too few parameters, we get back a **partially applied** function, meaning a function that takes as many parameters as we left out. Using partial application (calling functions with too few parameters, if you will) is a neat way to create functions on the fly so we can pass them to another function or to seed them with some data.

Example: (+) 3, 5 (*)

wada

Anonymous function

Lambdas are basically anonymous functions that are used because we need some functions only once. Normally, we make a lambda with the sole purpose of passing it to a higher-order function. To make a lambda, we write a \ (because it kind of looks like the greek letter lambda if you squint hard enough) and then we write the parameters, separated by spaces. After that comes a -> and then the function body. We usually surround them by parentheses, because otherwise they extend all the way to the right.

Example:

```
\xs -> length xs > 15
```

is an anonymous function that takes as input à list and returns a boolean. It the same as the function

```
moreElemThan15 :: [a] -> Bool
moreElemThan15 xs = length xs > 15
```

Example of higher order function

Map

map is a function that takes as input a function and a list and applies that function to every element in the list, producing a new list. Let's see what its type signature is and how it's defined.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

The type signature says that it takes a function that takes an a and returns a b, a list of a's and returns a list of b's. It's interesting that just by looking at a function's type signature, you can sometimes tell what it does. map is one of those really versatile higher-order functions that can be used in millions of different ways. Here it is in action:

Example

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

Filter

filter is a function that takes a predicate (a predicate is a function that tells whether something is true or not, so in our case, a function that returns a boolean value) and a list and then returns the list of elements that satisfy the predicate. The type signature and implementation go like this:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

wada

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
```

Pretty simple stuff. If p x evaluates to True, the element gets included in the new list. If it doesn't, it stays out. Some usage examples:

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) in filter notNull
[[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) "u LaUgH aT mE BeCaUsE I
aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) "i lauGh At You BecAuse
u r aLL the Same"
"GAYBALLS"
```

Flip

Flip simply takes a function and returns a function that is like our original function, only the first two arguments are flipped. It signature is defined as follow:

```
flip :: (a -> b -> c) -> b -> a -> c
```

We can implement it like so:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f = g
         where g x y = f y x
```

Fold (Foldl, Foldr)

A fold takes a function, a starting value and a list to fold up. The function itself takes two parameters. The function is called with the accumulator and the first (or last) element and produces a new accumulator. Then, the binary function is called again with the new accumulator and the now new first (or last) element, and so on. Once we've walked over the whole list, only the accumulator remains, which is what we've reduced the list to.

Fold right

Backtracking

Backtracking is an algorithmic technique for solving problems recursively by putting in standby some task, looking deeper to get a partial solution and use it for previously left tasks.
Example: factorial function.

```
factorial :: Integral a => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

We call foldr the “right fold” because the fold is right associative; that is, it associates to the right. This is syntactically reflected in a straightforward definition of foldr as well:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z [] = z
```

wada

```
foldr f z (x:xs) = f x (foldr f z xs)
```

The “rest of the fold,” (foldr f z xs) is an argument to the function f we’re folding with. The z is the zero of our fold. It provides a fallback value for the empty list case and a second argument to begin our fold with. The zero is often the identity for whatever function we’re folding with, such as 0 for (+) and 1 for (*).

Example:

```
foldr (+) 0 [1, 2, 3]
```

How it evaluate it:

foldr (+) 0 [1, 2, 3] = (+) 1 (foldr (+) 0 [2, 3]) ⇒ 6
foldr (+) 0 [2, 3] = (+) 2 (foldr (+) 0 [3]) ⇒ 5
foldr (+) 0 [3] = (+) 3 (foldr (+) 0 []) ⇒ 3
foldr (+) 0 [] = 0

foldr (+) 0 [1, 2, 3] ⇒ (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))

⇒ 1 + (2 + (3 +(foldr (+) 0 [])))

Fold left

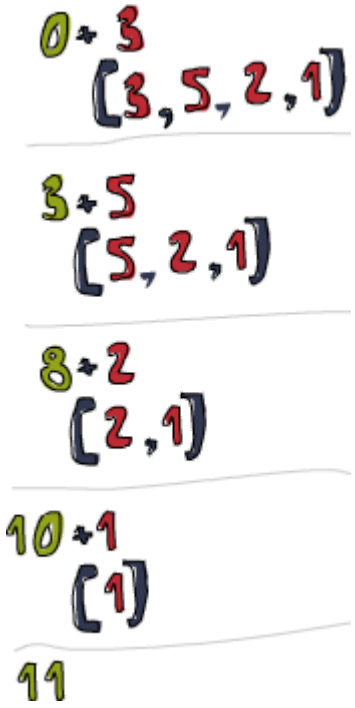
Because of the way lists work, folds must first recurse over the spine of the list from the beginning to the end. Left folds traverse the spine in the same direction as right folds, but their folding process is left associative and proceeds in the opposite direction as that of foldr.
Here’s a simple definition of foldl. Note that to see the same type for foldl in your GHCi REPL you will need to import Data.List for the same reasons as with foldr.

```
foldl :: (b -> a -> b) -> b -> [a] -> b
```

wada

```
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

Example:
foldl (+) 0 [3, 5, 2, 1] = foldl (+) 0 (3 : 5 : 2 : 1 : [])



Scan

Scans work similarly to maps and also to folds. Like folds, they accumulate values
instead of keeping the list's individual values separate. Like maps, they return a list of results. In this case, the list of results shows the intermediate stages of evaluation, that is, the values that accumulate as the function is doing its work. Scans are not used as frequently as folds, and once you understand the basic mechanics of folding, there isn't a whole lot new to understand. Still, it is useful to know about them and get an idea of why you might need them.

wada

First, let’s take a look at the types. We’ll do a direct comparison of the types of folds and scans so the difference is clear:

```
foldr :: (a -> b -> b) -> b -> [a] -> b
scanr :: (a -> b -> b) -> b -> [a] -> [b]
foldl :: (b -> a -> b) -> b -> [a] -> b
scanl :: (b -> a -> b) -> b -> [a] -> [b]
```

example

scanr (+) 0 [1, 2, 3]
[1 + (2 + (3 + 0)), 2 + (3 + 0), 3 + 0, 0]
[6, 5, 3, 0]

scanl (+) 0 [1, 2, 3]
[0, 0 + 1,0 + 1 + 2, 0 + 1 + 2 + 3]
[0, 1, 3, 6]

Function application ($)

Alright, next up, we'll take a look at the $ function, also called *function application*. First of all, let's check out how it's defined:

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Whereas normal function application (putting a space between two things) has a really high precedence, the $ function has the lowest precedence. Function application with a space is left-associative (so f a b c is the same as ((f a) b) c)), function application with $ is right-associative.
That's all very well, but how does this help us? Most of the time, it's a convenience function so that we don't have to write so many parentheses. Consider the expression sum (map sqrt [1..130]). Because

$ has such a low precedence, we can rewrite that expression as sum $ map sqrt [1..130], saving ourselves precious keystrokes! When a $ is encountered, the expression on its right is applied as the parameter to the function on its left. How about sqrt 3 + 4 + 9? This adds together 9, 4 and the square root of 3. If we want to get the square root of 3 + 4 + 9, we'd have to write sqrt (3 + 4 + 9) or if we use $ we can write it as sqrt $ 3 + 4 + 9 because $ has the lowest precedence of any operator. That's why you can imagine a $ being sort of the equivalent of writing an opening parentheses and then writing a closing one on the far right side of the expression.

zipWith

The zipWith function makes a list, its elements are calculated from the function and the elements of input lists occuring at the same position in both lists. It signature is as follow:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

function composition (.)

In mathematics, function composition is defined like this: $(f \circ g)(x) = f(g(x))$, meaning that composing two functions produces a new function that, when called with a parameter, say, *x* is the equivalent of calling *g* with the parameter *x* and then calling the *f* with that result.

In Haskell, function composition is pretty much the same thing. We do function composition with the . function, which is defined like so:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)   or (.) f g x = f (g x)
```

Mind the type declaration. f must take as its parameter a value that has the same type as g's return value. So the resulting function takes a

parameter of the same type that g takes and returns a value of the same type that f returns.

uncurry and curry

Haskell is curried by default, but you can uncurry functions. Uncurrying means unnesting the functions and replacing the two functions with a tuple of two values (these would be the two values you want to use as arguments). If you uncurry (+), the type changes from `Num a => a -> a -> a` to `Num a => (a, a) -> a` which better fits the description "takes two arguments, returns one result" than curried functions. Some older functional languages default to using a product type like tuples to express multiple arguments.

- Uncurried functions: One function, many arguments
- Curried functions: Many functions, one argument apiece

You can also desugar the automatic currying yourself, by nesting the arguments with lambdas, though there's rarely a reason to do so. We'll use anonymous lambda syntax here to show you some examples of uncurrying. You may want to review anonymous lambda syntax or try comparing these functions directly and thinking of the backslash as a lambda:
index anonymous function ! syntax

```
nonsense :: Bool -> Integer
nonsense True = 805
nonsense False = 31337
curriedFunction :: Integer -> Bool -> Integer
curriedFunction i b = i + (nonsense b)
uncurriedFunction :: (Integer, Bool) -> Integer
uncurriedFunction (i, b) = i + (nonsense b)
```

wada

```
anonymous :: Integer -> Bool -> Integer
anonymous = \i b -> i + (nonsense b)
anonNested :: Integer -> Bool -> Integer
anonNested = \i -> \b -> i + (nonsense b)
```

Then when we test the functions from the REPL:

```
Prelude> curriedFunction 10 False
31347
Prelude> anonymous 10 False
31347
Prelude> anonNested 10 False
31347
```

They are all the same function, all giving the same results. In anonNested, we manually nested the anonymous lambdas to get a function that was semantically identical to curriedFunction but didn't leverage the automatic currying. This means functions that seem to accept multiple arguments such as with a -> a -> a -> a are higher-order functions: they yield more function
values as each argument is applied until there are no more (->) type constructors and it terminates in a non-function value.

References:

1. https://riptutorial.com/haskell/example/16493/associativity
2. https://kowainik.github.io/posts/fixity
3. http://learnyouahaskell.com/higher-order-functions
4. https://en.wikipedia.org/wiki/Operator_associativity
5.