Higher-Order Functions

Simple Example: `applyFunc`

```
applyFunc f x = undefined
```

Question

What does the RHS of this function definition look like?

Answer:

```
applyFunc f x = f x
```

Example function calls:

```
applyFunc not True => False
applyFunc intToChar 3 => '3'
```

Question

What is the type of `applyFunc`?

Hint: Think about the types of its arguments.

Answer:

```
applyFunc :: (a -> b) -> a -> b
```

`applyFunc` is an example of a **higher-order function** (because it takes in a function as one of its arguments).

Now think about the types of each piece of the example function calls above:

```
applyFunc not True -- applyFunc :: (Bool -> Bool) -> Bool -> Bool
applyFunc intToChar 3 -- applyFunc :: (Int -> Char) -> Int -> Char
```

$

In order to reduce the number of parentheses, Haskell allows you to write something like:

```
y = f (g (h (1 + 2)))
```

as

```
y = f $ g $ h $ 1 + 2
```

`$` is *right-associative*, so the function applications go from right to left. `$` is also a function (specifically, an infix function).

Question

Consider a simple use of `$`:

```
y = f $ 1
```

What's the type of `$`?

```
($) :: (a -> b) -> a -> b
```

This is the same type as `applyFunc`!

More Complex HoF

Question

How would you write `applyFunc2Args`, such that:

- **Input**: A function that accepts 2 args, `f`, and 2 values, `x` and `y`
- **Output**: The result of calling `f` with 2 args, `x` and `y`

Answer:

```
applyFunc2Args :: (a -> b -> c) -> a -> b -> c
applyFunc2Args f x y = f x y
```

Question

How would you write `apply2Funcs`, such that:

- **Input**: 2 functions, `f1` and `f2` and a value, `x`
- **Output**: The result of applying `f1` to `f2 x`

Answer:

```
apply2Funcs :: (a -> b) -> (b -> c) -> a -> c
apply2Funcs f g x = f $ g x
```

`map`

Haskell's `map` function is a very useful building block for more complicated functions.

Type signature:

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

Question

Based on the name of the function and type signature, what does this function do?

Answer: Applies the input function to each of the arguments in a list and returns the resulting list.

Example applications:

```
map not [True, False, True] => [False, True, False]

f x = x + 1
map f [1, 2, 3] => [2, 3, 4]
```

zip

Note: This is not a higher-order function, but it is a useful building block for more complex functions (like ***zipWith****, see below).*

```
zip :: [a] -> [b] -> [(a, b)]
```

Question

Based on the type signature of `zip`, what will the following function call reduce to? What's the type of the result?

```
zip [1, 2, 3] ['a', 'b', 'c']
```

Answer:

```
[(1, 'a'), (2, 'b'), (3, 'c')] :: Num a => [(a, Char)]
```

zipWith

Now consider:

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

Question

Based on the type signature, what do you think this function does?

Answer: Accepts a function with 2 args, two lists, and applies the function to each consecutive pair of values in the lists.

For example:

“`zipWith (+) [1,2,3] [4,5,6] => [5, 7, 9]`