1 Higher-Order Functions

1.1 Map Function

Consider the function, `shout`, which takes in a `[Char]` and returns a `[Char]` where each element is uppercase.

```
shout :: [Char] -> [Char]
shout []        = []
shout (x:xs)    = toUpper x : shout xs
```

Now consider the function `squares`, which takes in a `[Int]` and returns a `[Int]` where each element is squared.

```
squares :: [Int] -> [Int]
squares []        = []
squares (x:xs)    = x * x : squares xs
```

Notice that these two functions do nearly the same thing – they go through each element and *transform* (i.e. *map*) each element to another element. This introduces the *map* pattern, where we can do something like

```
map []          = []
map (x:xs)      = f x : map f xs
```

Here, we can rewrite `shout` and `squares` like so:

```
shout   = map (\x -> toUpper x)
squares = map (\x -> x * x)
```

(Quiz.) What is the type of `map`?

(a) `(Char -> Char) -> [Char] -> [Char]`

(b) `(Int -> Int) -> [Int] -> Int`

(c) `(a -> a) -> [a] -> [a]`

(d) `(a -> b) -> [a] -> [b]`

(e) `(a -> b) -> [c] -> [d]`

The answer is **D**. Keep in mind that you can *map* one type to another type (including the original type). For example, you could map an `[Int]` to a `[Char]` by mapping each integer to its string representation (e.g. `10 -> "10"`).

The type essentially says it all. The only meaningful thing a function of this type can do is apply its first argument to elements of the list.

1.2 Folding Right

Recall that the length of the list can be found by

```
len :: [a] -> Int
len []      = 0
len (x:xs)  = 1 + len xs
```

Recall that summing a list can be done by

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs)  = x + sum xs
```

Recall that string concatenation can be done by

```
cat :: [String] -> String
cat []      = ""
cat (x:xs)  = x ++ cat xs
```

This is an example of the *fold-right* pattern. We can generalize this to the function

```
foldr f b []        = b
foldr f b (x:xs)    = f x (foldr f b xs)
```

Here, `b` is the base value. The general pattern is to recurse on the list, combining the result with the head using some binary operation. So, the above functions can be rewritten like so:

```
len = foldr (\x n -> 1 + n) 0
sum = foldr (\x n -> x + n) 0
cat = foldr (\x s -> x ++ s) ""
```

(Quiz.) What does this evaluate to?

```
foldr f b []        = b
foldr f b (x:xs)    = f x (foldr f b xs)

quiz = foldr (:) [] [1, 2, 3]
```

(a) Type error.

(b) `[1, 2, 3]`

(c) `[3, 2, 1]`

(d) `[[3], [2], [1]]`

(e) `[[1], [2], [3]]`

The answer is **B**. Recall that `(:)` is the *cons* operator. In particular, recall that `1:(2:(3:[])) => [1, 2, 3]`. Running through this example, we have

```
foldr (:) [] [1,2,3]
==> (:) 1 (foldr (:) [] [2, 3])
==> (:) 1 ((:) 2 (foldr (:) [] [3]))
==> (:) 1 ((:) 2 ((:) 3 (foldr (:) [] [])))
==> (:) 1 ((:) 2 ((:) 3 []))
==  1 : (2 : (3 : []))
==  [1,2,3]
```

thus giving us the desired answer.

To see the pattern, consider a more general example:

```
foldr f b [x1, x2, x3, x4]
==> f x1 (foldr f b [x2, x3, x4])
==> f x1 (f x2 (foldr f b [x3, x4]))
==> f x1 (f x2 (f x3 (foldr f b [x4])))
==> f x1 (f x2 (f x3 (f x4 (foldr f b []))))
==> f x1 (f x2 (f x3 (f x4 b)))
```

The reason why it's called *fold-right* is because it processes the elements from the *right* – that is, it combines the base value b with the last element, and then combines the second-to-last value with the now modified last value, and so on.

(Quiz.) Using the definition of `foldr`, what is the most general type?

(a) `(a -> a -> a) -> a -> [a] -> a`

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

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

(d) `(a -> b -> b) -> b -> [a] -> b`

(e) `(b -> a -> b) -> b -> [a] -> b`

The answer is **D**.