Folds

I Did It My Way

Implement the functions below. Just as in the previous exercise, avoid using the versions of these functions in the standard library and do it your way.

- Write a function `mySum :: [Integer] -> Integer` which calculates the sum of the numbers in a list.
- Write a function `myProduct :: [Integer] -> Integer`, which, similarly to the previous function, calculates the product of the numbers in a list.[1]
- After writing these two functions, you should have noticed they look very similar, and only differ in a few places. Write a function `foldInts` which has the common characteristics of `mySum` and `myProduct`, and accepts the different characteristics as parameters.

  ```
  foldInts :: (Integer -> Integer -> Integer)
           -> Integer -> [Integer] -> Integer
  ```

  Using this `foldInts` function, `mySum` and `myProduct` could be implemented as follows:

  ```
  mySum     = foldInts (+) 0
  myProduct = foldInts (*) 1
  ```

Examples

```
Main> mySum [1,4,7,10]
22

Main> mySum []
0

Main> myProduct [1,2,3]
6

Main> myProduct []
1

Main> foldInts (+) 0 [1,2,3,4]
10

Main> foldInts (*) 1 [1,2,3,4]
24
```

[1] We use `Integer` (arbitrary-precision-integers) instead of plain `Int`, since the size of the product can rise rapidly.

Associativity and Folds

Your `foldInts` function implicitly puts the operator between the elements of the list. So:

```
foldInts (+) 0  [1,2,3,4] = 0+1+2+3+4
```

The `(+)`-operator is commutative, so the order of execution is not relevant. However, what should happen when we execute `foldInts (-) 0 [1,2,3,4]`? Here, there are two options, either we associate to the left: $((((0-1)-)2-)3-4) = -10$ or we associate to the right: $(1-(2-(3-(4-0)))) = -2$. Since this is a very common operation in functional programming, the `Prelude` predefines the following 2 functions:

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

- Implement the two functions yourself:

```
myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldr :: (a -> b -> b) -> b -> [a] -> b
```

- Write a function `readInBase :: Int -> [Int] -> Int` using one of the fold functions that takes a list of digits in base B and outputs the number in base 10.

```
Main> myFoldl (+) 0 [1,2,3]
6

Main> myFoldl (-) 0 [1,2,3]
-6

Main> myFoldr (-) 0 [1,2,3]
2

Main> myFoldl (++) "" ["Hello", " ", "World"]
"Hello World"

Main> myFoldr (+) 0 [1,2,3]
6

Main> myFoldr (:) [] [1,2,3]
[1,2,3]

Main> myFoldr (++) "" ["Hello", " ", "World"]
"Hello World"

Main> readInBase 2 [1,0]
2
```

```
Main> readInBase 6 [1,3,0]
54
```

Hint: Use Horner's method in combination with a fold function. Horner's method calculates polynomials using the following scheme:

$$a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n = a_0 + x(a_1 + x(a_2 + x(\ldots + x(a_n))))$$

Remember that the number 130 in base 6 can be written as:

$$1 * 6^2 + 3 * 6^1 + 0 * 6^0$$

Map

Function `map` is also a common function in Haskell (it is available in the Haskell Prelude under the name `map`). It takes a function and a list of elements and applies the function to all elements:

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

- Implement function `myMap :: (a -> b) -> [a] -> [b]`, that is your own implementation of `map`. Do not use folds for this implementation.
- Implement function `myMapF :: (a -> b) -> [a] -> [b]`, this time using a fold function.

Examples

```
Main> myMap (+1) [1,2,3,4]
[2,3,4,5]

Main> myMap not [True, False]
[False, True]

Main> myMap not []
[]

Main> myMapF (+1) [1,2,3,4]
[2,3,4,5]

Main> myMapF not [True, False]
[False, True]

Main> myMapF not []
[]
```