Haskell Practice Sheet

Alex Rozanski

January 11, 2015

Higher-Order Functions

The skeleton file for this section is `functions.hs`.

Think about where you can use functions such as `zipWith`, `foldr1`, `takeWhile`, `dropWhile`, `map` and `concatMap` when writing your solutions.

1. Write the function `abbreviate :: [String] -> String` which, given a list of words returns a string comprised of the first character of each word. For instance, `abbreviate ["Department", "of", "Computing"]` should return `"DoC"`. Write the function **without** using recursion.

2. Write the function `myProduct :: [Int] -> Int` which, given a list of integers returns their product. For instance, `myProduct [5, 10, 3]` should return `150`. Write the function **without** using recursion (or the `product` function :))

3. Write the function `greaterThan :: [Int] -> Int -> [Int]` which, given a list of integers sorted in ascending order and a lower-bound, returns the list of those numbers which are greater than the lower bound. For instance: `greaterThan [1..10] 5` should return `[6, 7, 8, 9, 10]`. Write the function **without** using recursion.

4. Write the function `divisibleBy5 :: [Int] -> [Int]` which, given a list of integers returns a list containing those which are divisible exactly by 5. For instance, `divisibleBy5 [1..20]` should return `[5, 10, 15, 20]`. Write the function **without** using recursion.

5. Write the function `upperString :: String -> String` which, given a string returns its uppercase representation. For instance, `upperString "cat"` should return `"CAT"`. Remember you can use the `ord` function to convert a character to its ordinal value and `chr` to convert an ordinal value to its corresponding character. Also remember that in the ASCII character scheme, the characters `'A'` to `'Z'` come before the characters `'a'` to `'z'`.

   Now write `upperWords :: [String] -> String` which, given a list of words uses `upperString` to convert each word to its uppercase representation and combines all of these words into a single string at the end. For instance, `upperWords ["the", "cat"]` returns `"THECAT"`. Think about how you can write this function using a single higher-order function call.

6. Write the function `deriv :: [Int] -> [Int]` which, given a list of integers representing the coefficients of increasing powers of x (starting at 0), returns a list representing the coefficients of increasing powers of x of the derivative.

   For instance: we can represent the equation $0 + x + x^2 + 2x^3$ in this form as `[0, 1, 1, 2]`. The derivative $1 + 2x + 6x^2$ is represented in this form as `[1, 2, 6]`. As such, `deriv [0, 1, 1, 2]` should return `[1, 2, 6]`. Try writing this using `drop` and `zipWith`.

   (Hint: the coefficients can be calculated by multiplying a sub-list of the input coefficients pairwise with increasing powers of x from 1 to infinity).

Algebraic Data Types

The skeleton file for this section is `datatypes.hs`.

Trees

Given the following tree definition:

```
data Tree a = Empty | Leaf a | Node (Tree a) a (Tree a)
```

1. Write the function `treeCount :: Tree a -> Int` which, given a `Tree` holding values of type `a` returns the number of items stored in the tree. For instance: `treeCount (Node (Leaf 5) 6 (Leaf 7))` should return `3` and `treeCount Empty` should return `0`.

2. Write the function `flatten :: Tree a -> [a]` which, given a `Tree` of element type `a` returns all of the items stored in the tree. For instance: `flatten (Node (Node (Leaf 1) 2 (Leaf 3)) 4 (Node (Leaf 5) 6 (Leaf 7)))` should return `[1, 2, 3, 4, 5, 6, 7]` and `flatten Empty` should return `[]`.

3. Write the function `highest :: Tree Int -> Int` which, given a `Tree` of element type `Int` returns the highest value stored in the tree. For instance: `flatten (Node (Node (Leaf 1) 2 (Leaf 3)) 4 (Node (Leaf 5) 6 (Leaf 7)))` should return `7`. Write this function **without** using recursion.

4. Write the function `multiply :: Tree Int -> Int -> Tree Int` which, given a `Tree` of element type `Int` and a multiplier returns a new tree with every item stored multiplied by the multiplier. For instance: `multiply (Node (Node (Leaf 1) 2 (Leaf 3)) 4 (Node (Leaf 5) 6 (Leaf 7))) 3` should return `(Node (Node (Leaf 3) 6 (Leaf 9)) 12 (Node (Leaf 15) 18 (Leaf 21)))`.

5. Write the function `addTrees :: Tree Int -> Tree Int -> Tree Int` which, given two `Tree`s of element type `Int` returns a new tree with the items from each tree added pairwise. You should assume that both trees have exactly the same structure. For instance: `addTrees (Node (Leaf 1) 8 (Leaf 32)) (Node (Leaf 9) 5 (Leaf 17))` should return `(Node (Leaf 10) 13 (Leaf 49))`.

The `List` type

In the skeleton file there is a custom `List` Algebraic Data Type (analogous to Haskell's built-in lists) defined as follows:

```
data List a = Nil | Cons a (List a)
```

We can construct `List`s similar to how we construct lists in Haskell. For instance, [1, 2 3] can be constructed as follows:

```
Cons 1 (Cons 2 (Cons 3 Nil))
```

Which is equivalent to the following using lists in Haskell:

```
1 : (2 : (3 : []))
```

1. Write a function `myLength :: List a -> Int` which takes a `List` of `a` elements and returns the number of elements in the list. For instance: `myLength (Cons 1 (Cons 2 (Cons 3 Nil)))` should return `3` and `myLength Nil` should return `0`.

2. Write a function `myMap :: List a -> (a -> b) -> List b` which works analogously for our new `List` data type as `map` does for Haskell lists. `myMap` should take a `List` of `a` elements, a function which takes an `a` and returns a `b` and return a `List` of `b` elements.

   For instance: `myMap (Cons 1 (Cons 2 (Cons 3 Nil))) (+1)` should return `Cons 2 (Cons 3 (Cons 4 Nil))` and `myMap (Cons 'a' (Cons 'b' (Cons 'c' Nil))) (=='b')` should return `Cons False (Cons True (Cons False Nil))`.