Programming I: Functional Programming in Haskell

Unassessed Exercises 3: **Lists**

These exercises are unassessed so you do not need to submit them. They are designed to help you master the language, so you should do as many as you can at your own speed.

There are probably more questions on these sheets than you may need in order to get the hang of a particular concept, so feel free to skip over some of the questions. You can always go back to them later if you need to.

Model answers to each set will be handed out throughout the course.

These questions are split into two groups: Basics and List Comprehensions. However, you might even find ways of using list comprehensions among the basic questions; that's fine.

1 Basics

By the end of the course you should be familiar with many of the list processing functions in the Haskell prelude and core modules. You should start with the prelude functions listed in the notes, but are strongly encouraged to browse the Haskell prelude and other modules, where you'll find many more. The `Data.List` module is particularly useful, as you'll see below.

1. By inspection (check them if you like by typing them into GHCi), determine whether the following are correctly typed. If they are, determine the resulting value. If not explain the cause of the type error.

   (a) `"H" : ['a', 's', 'k', 'e', 'l', 'l']`

   (b) `('o' : ['n']) ++ "going"`

   (c) `"Lug" ++ ('w' : "orm")`

   (d) `if "a" == ['a'] then [] else "two"`

   (e) `let xs = "let xs" in length [xs]`

   (f) `tail "emu" : drop 2 "much"`

   (g) `head ["gasket"]`

   (h) `zip "12" (1 : [2])`

   (i) `tail ((1,1), (2,2), (3,3))`

   (j) `head [1, (2,3)] /= head [(2,3), 1]`

   (k) `length [] + length [[]] + length [[[]]]`

   (l) `null ([""", "1", "11", "111"] !! 1)`

   (m) `zip [5, 3, length []] [(True, False, True)]`

   (n) `unzip [('b', 'd'), ('a', 'o'), ('d', 'g')]`

   (o) `and [1 == 1, 'a' == 'a', True /= False]`

(p) `sum [length "one", [2, 3] !! 0, 4]`

(q) `minimum [('b', 1), ('a', 2), ('d', 3)]`

(r) `maximum zip "abc" [1, 2, 3]`

(s) `concat [tail ["is", "not", "with", "standing"]]`

(t) `map head [1..n]`

(u) `filter null (map tail ["a", "ab", "abc"])`

(v) `foldr1 (||) (map even [9, 6, 8 , 2])`

(w) `zipWith (:) "zip" "with"`

(x) `foldr max 0 [0, 1]`

(y) `foldr maximum [0] [[0,1], [3,2]]`

(z) `zipWith (&&) [True] (filter id [True, False])`

2. The operators `(==)`, `(/=)`, `(<)`, `(>)`, `(<=)`, `(>=)` work on lists as well as the other numeric types you've seen. The 'dictionary' style (lexicographical) ordering is perhaps more apparent here. Try the following:

   (a) `"Equals" == "Equals"`

   (b) `"dictionary" <= "dictator"`

   (c) `"False" > "Falsehood"`

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

   (e) `[1, 2, 3] < [1, 1, 5]`

   (f) `[(1, "way")] < [(2, "ways"), (3, "ways")]`

   Now write a function `precedes` (equivalent to `(<=)`) which takes two `String`s and evaluates to `True` if the first is lexicographically smaller than or equal to the the second. Note that you can generalise the type to make it polymorphic, but you need to learn how to restrict the types to those that are *orderable*. We'll cover this later.

3. Write a function `pos` to find the position of a specified `Int` in a list of `Int`s. Assume that the integers are indexed from 0 and that the specified character will always be found. For example, `pos 4 [3, 4, 0, 1]` should evaluate to `1`.

4. Use the built-in function `elem` write a function `twoSame :: [Int] -> Bool` that delivers `True` iff the given list of integers contains at least one duplicate element. What is the complexity of `twoSame` as a function of the number of elements, n say, in the given list?

5. Define a polymorphic function `rev` (equivalent to the built-in `reverse` function) to reverse a list of objects of arbitrary type. For example, `rev "bonk"` evaluates to `"knob"`. You can add an element `x` to the rightmost end of a list `xs` thus: `xs ++ [x]`. What is its complexity, i.e. what is the cost of your function in terms of the number of `(:)` operations it performs to reverse a list of length n? Now modify your `reverse` function so that it uses repeated applications of `(:)` to an accumulating parameter (initially `[]`) instead of repeatedly adding to the rightmost end of a list. What is the complexity of the new function?

6. Write a function `substring :: String -> String -> Bool` which returns `True` iff the first given string is a substring of the second, e.g. `substring "sub" "insubordinate"` evaluates to `True` whilst `substring "not" "tonight"` evaluates to `False`.

7. Two anagrams can be considered to define a transposition of the characters of a third string. For instance, the anagrams `"abcde"` and `"eabcd"` define a transposition in which the last character of a 5-character string is moved to the start, the other characters remaining in the same order.

   Define a function transpose to transpose the characters of a string as specified by two anagrams. `transpose "UVWXYZ" "fedcba" "ecabdf"` should evaluate to `"VXZYWU"`. Hint: use your `pos` from earlier and `(!!)`, noting that all three strings are the same length and that each character of the second string is unique and appears exactly once in the third.

8. Define a recursive function `trimWhitespace :: String -> String` that will remove leading whitespace from a given string. Note: the first of these is guaranteed to be non-whitespace – an important precondition for the next function. The whitespace characters include (`' '`, `'\t'`, `'\n'`) and you can use the built-in function `isSpace` (from `Data.Char`) to check for them.

9. Write a function `nextWord` which given a string `s` returns a pair consisiting of the next word in `s` and the remaining characters in `s` after the first word has been removed. A precondition is that the first character in the input string is non-whitespace.

10. Using `trimWhitespace` and `nextWord`, write a function `splitUp` (similar to the built-in function `words`) which returns the list of words contained in a given `String`. The words may be separated by more than one whitespace character and there may be leading whitespace as well. Hint: you should only need to use `trimWhitespace` in one place in your function.

11. Define a function `primeFactors :: Int -> [Int]` that generates the list of prime factors of a given integer $n \geq 1$. To compute the prime factors of an integer n, start with the first prime, 2, and divide n by 2 repeatedly whilst the remainder is 0. This delivers zero or more factors that are all 2, e.g.:

    ```
    Main> primeFactors (2^8)
    [2,2,2,2,2,2,2,2]
    ```

    You're not done yet, though. You now have do the same again, this time with 3 and the number that remains after repeated division by 2. For example:

    ```
    Main> primeFactors (3^3 * 2^8)
    [2,2,2,2,2,2,2,2,3,3,3]
    ```

    Likewise with 4, 5, 6 and so on. As it happens you don't need to consider even numbers larger than 2 as they are certainly not prime, so you could skip 4, 6, 8 and so on, as an optimisation. What about an odd number like 9? It's actually safe to check this even though it's not prime because a multiple of 9 is also a multiple of 3, which *is* prime: the number at hand cannot therefore be a multiple of 9. Note that if a number is prime then it has only itself as a prime factor. Make sure to test your function in this case.

12. The highest common factor of two integers a and b can be expressed in terms of their prime factors. Suppose `ps` represents the list of primes factors of a, and `ps'` similarly for b. The highest

common factor of a and b is the product of common prime factors of a and b, i.e. the product of the elements that are common to `ps` and `ps'`. For example, the highest common factor of `300=2*2*3*5*5` and `375=3*5*5*5` is `3*5*5`, because `3`, `5` and `5` are the prime factors that `300` and `375` have in common. Using the `primeFactors` function above, define a function `hcf :: Int -> Int -> Int` that will compute the highest common factor of two given integers. Use the `(\\)` operator from the `Data.List` module (type `import Data.List` at the top of the program). This takes two lists and delivers the elements of the first list that remain when the elements of the second are removed from it.

13. Again using `(\\)` from module `Data.List`, write a function `lcm :: Int -> Int -> Int` that delivers the lowest common multiple of two given integers. Suppose a is the smaller of the two numbers and b the larger. The lowest common multiple of a and b is the product of the prime factors of a (i.e. a itself!) and the prime factors of b that are not included in the prime factors of a. For example the lowest common multiple of `300=2*2*3*5*5` and `375=3*5*5*5` is `300*5 = 1500`.

2 List Comprehensions

1. The following function is supposed to find all the bindings for `x` in a table of `(x, y)` pairs:

   ```
   findAll x t = [y | (x, y) <- t]
   ```

   Try it out with the application `findAll 1 [(1,2),(1,3),(4,7)]`. Oops! What's gone wrong? How would you fix it?

2. Tony Hoare's famous "quicksort" algorithm works by partitioning a (non-empty) list into those elements less than or equal to that at the head and those greater than that at the head. These two lists are then recursively sorted and the results appended together, with the head element sandwiched between the two. Define a Haskell version of quicksort that generates the two sublists using list comprehensions.

3. The built-in function `splitAt` splits a list at a specified index and returns the elements up to (and including) the indexed element and those that follow, in the form of a pair. For example, `splitAt 2 "12345"` returns `("12", "345")`. Recall that the head of a list is at index 0. Using `splitAt` define a function `allSplits :: [a] -> [([a], [a])]` that returns the result of splitting a list at all possible points. For example, `allSplits "1234"` should return `[("1","234"), ("12","34"), ("123","4")]`.

4. Define a function `prefixes :: [a] -> [[a]]` that will compute all prefixes of a given list. For example, `prefixes "123"` should return `["1", "12", "123"]`. Hint: notice in the example that `'1'` appears at the head of every element of the result (a `map` maybe?).

5. Define a function `substrings :: String -> [String]` that will compute all substrings of a given string. For example, `substrings "123"` should generate `["1", "12", "123", "2", "23", "3"]` in some order. You should be able to use a similar trick to that for `prefixes` above.

6. Recall that the operator `(\\)` in the `Data.List` module removes specified elements from a given list, e.g. `[1,2,3,4] \\ [1,3]` returns `[2,4]`. Define a function `perms :: [a] -> [[a]]` that generates all permutations of a given list using `(\\)` and a list comprehension. For example, `perms [1,2,3]` should return, in some order, `[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]`.

7. Suppose a graph is defined by a list of its edges, where each edge is specified as a pair of node identifiers. For example, if the node identifiers are (unique) integers, the following represents a circular graph with four nodes: `[(1,2), (2,3), (3,4), (4,1)]`. Define a function `routes :: Int -> Int -> [(Int, Int)] -> [[Int]]` that will compute all routes from a specified node to another in a given *acyclic* graph, i.e. a graph with no cycles. For example, `routes 1 6 [(1,2), (1,3), (2,4), (3,5), (5,6), (3,6)]` should return `[[1,3,5,6], [1,3,6]]`.

   Now extend the function so that it works with cyclic graphs. For example, `routes 1 6 [(1,2), (2,3), (3,4), (4,1)]` should return `[]`. In the previous version, you should find your function will loop indefinitely in this case. Try it!

   Hint: Define an auxiliary function that takes the list of nodes you have already visited as an additional parameter.