Programming I: Functional Programming in Haskell

Unassessed Exercises 3: **Lists** (Solutions)

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.

   Solution: For all these parts, a lack of solution indicates it works.

   Recall that lists are constructed of:

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

   and that `type String = [Char]`

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

   Solution: Type error: `"H" :: [Char]`, but `(: ['a', 's', 'k', 'e', 'l', 'l']) :: Char -> [Char]`

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

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

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

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

> **Solution:** Careful, the answer here is `1`!

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

> **Solution:** Type error: `tail :: [a] -> [a]`, so `tail "emu" :: [Char]` and `drop :: Int -> [a] -> [a]`, so `drop 2 "much" = "ch" :: [Char]`. This makes the two arguments incompatible for `(:)`.

(g) `head ["gasket"]`

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

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

> **Solution:** This is not a list, you cannot call `tail` on a triple!

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

> **Solution:** Wrong for multiple reasons: lists must have a homogeneous type, but `1` and `(2, 3)` are not the same time, so cannot be in a list together. Suppose this *were* allowed, say it was `fst (1, (2,3)) /= fst ((2,3), 1)`, then again, the types of the two first elements do not match, so `(/=)` makes no sense.

(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]`

> **Solution:** Likely missing parentheses around the `zip`, `zip` itself is not a valid first argument to `maximum`.

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

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

> **Solution:** The function `map :: (a -> b) -> [a] -> [b]`, here `[1..n] :: [Integer]`, say, so the function passed to `map` must be of type `Integer -> b` for some `b`, but `head` takes a list, not an `Integer`!

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

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

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

> **Solution:** The partially applied function `zipWith (:) :: [a] -> [[a]] -> [[a]]` can be applied to `"zip"` to obtain a function of type `[[Char]] -> [[Char]]`, but `"with"` does not have type `[[Char]]`!

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

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

> **Solution:** Unlike `max`, which is a two-argument function, `maximum` is only a single argument function, so does not fit into `foldr`.

(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.

> **Solution:**
>
> ```
> precedes :: String -> String -> Bool
> precedes [] _ = True
> precedes _ [] = False
> precedes (c : cs) (c' : cs')
>   | c < c'     = True
>   | c == c'    = precedes cs cs'
>   | otherwise = False
> ```
>
> There is a little redundancy in this solution however, and by generalising the second case and removing the last guard, we can get a more concise definition:
>
> ```
> precedes :: String -> String -> Bool
> precedes [] _ = True
> ```

```
precedes (c : cs) (c' : cs')
  | c < c'    = True
  | c == c'   = precedes cs cs'
precedes _ _ = False
```

This works because failure to match a guard will fall through to the next clause.

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`.

Solution:

```
pos :: Char -> String -> Int
pos c (c' : cs)
  | c == c'   = 0
  | otherwise = 1 + pos c cs
```

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?

Solution:

```
twoSame :: [Int] -> Bool
twoSame [] = False
twoCase (x : xs) = elem x xs || twoSame xs
```

The complexity is $O(n^2)$, since it walks every element of the list and each element will perform a worse case $O(n)$ call to `elem`. Here are a couple of better ways of writing this function:

```
twoSame :: [Int] -> Bool
twoSame xs = twoAdj (sort xs)

twoAdj :: [Int] -> Bool
twoAdj (x:x':xs) = x == x' || twoAdj (x':xs)
-- if the list does not have two elements
twoAdj _ = False
```

This version makes use of an $O(n \log n)$ complexity sorting function first, then looks to see if any two elements *next to each other* are the same. As `twoAdj` is $O(n)$, the implementation is dominated by the cost of `sort`.

Extra Alternatively, you could use a *Set* datastructure. The `containers` package contains one in the `Data.Set` module. This has a few notable operations (among others):

```
empty :: Set a                            -- O(1)
insert :: Ord a => a -> Set a -> Set a    -- O(log n)
member :: Ord a => a -> Set a -> Bool     -- O(log n)
```

Using just these functions, `twoSame` can be written as:

```
import Data.Set (Set, member, empty, insert)

twoSame :: [Int] -> Bool
twoSame xs = go xs empty
  where go :: [Int] -> Set Int -> Bool
        go [] _ = False
        go (x : xs) seen = member x seen || go xs (insert x seen)
```

This works by threading a set of elements that we've seen during the traversal, and asking if the current element has already been seen before. This does two $O(\log n)$ operations for each element of the list, so is $O(n \log n)$ complexity overall – same as the version with `sort`. However, this one is likely to do less work overall, so is the more efficient choice!

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?

Solution:

```
snoc :: [a] -> a -> [a]
snoc xs x = xs ++ [x]

rev :: [a] -> [a]
rev [] = []
rev (x:xs) = snoc (rev xs) x
```

As we walk the list looking at each element once, we know it will perform n `snoc`s (`cons` backwards). However, a `snoc` is a linear time operation in itself, as Haskell lists must be prepended to! As such, the overall complexity is $O(n^2)$. Resist the temptation of `snoc`!

```
rev :: [a] -> [a]
rev xs = go xs []
```

```haskell
  where go [] acc     = acc
        go (x:xs) acc = go xs (x:acc)
```

As `(:)` is $O(1)$, this will be $O(n)$: much better! In general, one `snoc` is ok, but its better to build a list with `(:)` and reverse it for $O(n)$ construction rather than building it with `snoc` in $O(n^2)$, if possible. There are better ways of building lists, but you'll learn about those next year.

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`.

Solution: Using two functions for this is much easier, one to check for prefixes, and the other for substring.

```haskell
substring :: String -> String -> Bool
substring _ [] = False
substring s s'@(_:cs) = isPrefix s s' || substring s cs

isPrefix :: String -> String -> Bool
isPrefix [] _ = True
isPrefix _ [] = False
isPrefix (c : cs) (c' : cs') = c == c' && isPrefix cs cs'
```

Using some nice list functions, it is possible to define `substring` without recursion:

```haskell
import Data.List (tails)

substring :: String -> String -> Bool
substring query search = any (isPrefix query) (tails search)
```

The function `tails` returns all the suffixes of the string, and then `isPrefix` can be used as before with the `any` function.

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.

Solution:

```haskell
transpose :: String -> String -> String -> String
transpose s key [] = []
transpose s key (c:cs) = s !! pos c key : transpose s key cs
```

Extra The `(!!)` function is, like `snoc`, one of the functions that is best avoided if done repeatedly, same as `pos`. Simply put, this function is $O(n^2)$. Without changing the asymptotics, you could improve this function by avoiding the two linear operations:

```haskell
-- This function is partial and will crash if the key is not found in the list.
lookUp :: k -> [(k, v)] -> v
lookUp k ((k', v):kvs)
  | k == k' = v
  | otherwise = lookUp k kvs

transpose :: String -> String -> String -> String
transpose s key = go
  where table = zip key s
        go []     = []
        go (c:cs) = lookUp c table : go cs
```

This function builds a lookup table in advance, but the `lookUp` function defined there is also $O(n)$, so the complexity remains $O(n^2)$. Can we do any better? Earlier, we were introduced to the idea of `Set`. Indeed, the *right* data-structure for this task is a `Map` (found in `Data.Map`)!

```haskell
(!) :: Ord k => Map k v -> k -> v          -- O(log n)
fromList :: Ord k => [(k, v)] -> Map k v -- O(n log n)
```

Using just these two functions (there are, again, many more, take a look if you want), we can give a more efficient definition:

```haskell
import Data.Map (Map, (!), fromList)

transpose :: String -> String -> String -> String
transpose s key = go
  where table :: Map Char Char
        table = fromList (zip key s)
        go []     = []
        go (c:cs) = (table ! c) : go cs
```

This is better, working in $O(n \log n)$ time. Nice! Again, it's really worth being aware of `Set` and `Map`: these are what real Haskell programs will be using, when appropriate – not *always* lists.

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.

   Solution:

   ```
   trimWhitespace :: String -> String
   trimWhitespace "" = ""
   trimWhitespace s@(c:cs)
     | isSpace c = trimWhitespace cs
     | otherwise = s
   ```

   It may be better to use a higher-order function like `dropWhile` for this.

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.

   Solution:

   ```
   nextWord :: String -> (String, String)
   -- Pre: the first character is non-whitespace
   nextWord "" = ("", "")
   nextWord (c : cs)
     | isSpace c = ("", cs)
     | otherwise = let (w, s) = nextWord xs in (c : w, s)
   ```

   Or, alternatively:

   ```
   nextWord :: String -> (String, String)
   nextWord s = (w, tail rest)
     where (w, rest) = break isSpace s
   ```

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.

    Solution:

    ```
    splitUp :: String -> [String]
    splitUp "" = []
    ```

```
splitUp s = w : splitUp rest
  where (w, rest) = nextWork (trimWhitespace s)
```

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.

    Solution:

    ```
    primeFactors :: Int -> [Int]
    -- Pre: n > 0
    primeFactors n = factors 2 n
      where factors p 1 = []
            factors p m
              | m `mod` p == 0 = p : factors p (m `div` p)
              | otherwise      = factors (p + 1) m
    ```

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.

Solution:

```
hcf :: Int -> Int -> Int
hcf m n = product (ms  \\ (ms \\ ns))
  where ms = primeFactors m
        ns = primeFactors n
```

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`.

Solution:

```
hcf :: Int -> Int -> Int
hcf m n = product (maxs \\ mins) * min m n
  where mins = primeFactors (min m n)
        maxs = primeFactors (max m n)
```

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?

Solution: You end up with `[2, 3, 7]` instead `[2, 3]`, this is because you cannot pattern match *on* a lower-case name, so the `x` inside and outside of the comprehension are unrelated, unconditionally returning all results.

```
findAll x t = [y | (x', y) <- t, x == x']
```

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.

Solution:

```haskell
qsort :: [Int] -> [Int]
qsort [] = []
qsort (x : xs) = lefts ++ [x] ++ rights
  where lefts = qsort [x' | x' <- xs, x' <= x]
        rights = qsort [x' | x' <- xs, x' > x]
```

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")]`.

Solution:

```haskell
allSplits :: [a] -> [([a], [a])]
allSplits xs = [splitAt n xs | n <- [1..length xs - 1]]
```

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?).

Solution:

```haskell
prefixes :: [a] -> [[a]]
prefixes [] = []
prefixes (x : xs) = [x] : [x : prefix | prefix <- prefixes xs]

-- Alternatively
prefixes :: [a] -> [[a]]
prefixes [] = []
prefixes (x : xs) = map (x :) ([] : prefixes xs)

-- Notice that this has the classic shape of a fold!
prefixes :: [a] -> [[a]]
prefixes = foldr (\x ps ->  map (x :) ([] : ps)) []
```

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.

Solution: In fact, we can *use* `prefixes` for this.

```haskell
substrings :: String -> [String]
substrings [] = []
substrings s@(_:cs) = prefixes s ++ substrings cs
```

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]]`.

Solution:

```haskell
perms :: [Int] -> [[Int]]
perms [] = [[]]
perms xs = [x : ps | x <- xs, ps <- perms (xs \\ [x])]
```

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]]`.

Solution:

```haskell
routes :: Int -> Int -> [(Int, Int)] -> [[Int]]
routes from to g
  | from == to = [[from]]
  | otherwise  = [from : route | from' <- [e | (s, e) <- g, s == from]
                               , route <- routes from' to g]
```

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.

Solution: We've seen previously how `Set`s can be used to track values for cheap lookup, of course we could use `elem` instead of `member`, `[]` instead of `empty`, and `(:)` instead of `insert`, but this works well.

```haskell
import Data.Set (member, empty, insert)

routes :: Int -> Int -> [(Int, Int)] -> [[Int]]
routes from to g = go from empty
  where go from seen
          | member from seen = []
          | from == to = [[from]]
          | otherwise  = [from : route | from' <- [e | (s, e) <- g, s == from]
                                       , route <- go from' (insert from seen)]
```