3 Lists

Exercise 3.1 (*Warm-up*: List constructors, list types).
Haskell list notation is actually syntactic sugar for more low-level expressions consisting of the *list constructors* `[]` and `(:)` (sometimes called nil and cons). It is important to be able to mentally switch between those two notations.

```
xs0 = [1,2,3] ++ 4:[5]
xs1 = [1:2:[3],[4,5]]
xs2 = "abc"
xs3 = []
xs4 = [[],[]]
xs5 = [[]:[]]
xs6 = [[]++[]]
xs7 = [[[]]]
```

1. Write each of the above expressions in **list notation**, and give a possible type for it. For example, for `[1..3]`, the list notation is `[1,2,3]`, which can have type `[Integer]`.

2. Write each of the above expressions using **list constructors**, and underline its head and its tail. For example, for `[1..3]` it is `1:2:3:[]`. We can check this answer in GHCi:

   ```
   ⋙ [1..3] :: [Integer]
   [1,2,3]
   ⋙ 1:2:3:[]
   [1,2,3]
   ```

3. If we use the :type command to ask for the type of `[1,2,3]`, we get:

   ```
   ⋙ :type [1,2,3]
   Num a ⇒ [a]
   ```

   Why is that different?

Exercise 3.2 (*Warm-up*: Modes of recursion, accumulators, `Reverse.hs`).
In the slides, the function `reverse :: [a] → [a]` was defined as:

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

In the source code for the Haskell Prelude, it is however defined differently:

```
reverse xs = rev xs []
  where rev []     acc = acc
        rev (y:ys) acc = rev ys (y:acc)
```

These functions are equivalent but not identical. Try to compute `reverse [1,2,3]` using either definition. What do you see as the advantages and disadvantages of either definition? Which do you prefer? (Also see Hint 1)

Exercise 3.3 (*Warm-up*: list design pattern, `ListFunctions.hs`).
Give **recursive definitions** of these standard Haskell library functions. Use the *list design pattern* discussed in the lectures.

1. `and :: [Bool] → Bool`, that determines whether all elements in a list of Booleans are true;
2. `or :: [Bool] → Bool`, that determines whether there exists a Boolean in a list that is true;
3. `elem :: (Eq a) ⇒ a → [a] → Bool`, that tests whether an element occurs in a list;
4. `drop :: Int → [a] → [a]`, that discards the first n elements from a list;
5. `take :: Int → [a] → [a]`, that returns the first n elements of a list;

The purpose of this exercise is to train the list design pattern for defining simple list consumers and producers. Normally, whenever you write small functions that seem like they perform some standard list function, look at the Data.List library first: https://hackage.haskell.org/package/haskell2010/docs/Data-List.html

Exercise 3.4 (*Warm-up*: List constructors, evaluation, *pen-and-paper*, `Mingle.hs`).
Given this suspicious definition of a custom operator `++/`:

```
(++/) :: [a] → [a] → [a]
[]     ++/ ys = ys
(x:xs) ++/ ys = x:(ys ++/ xs)
```

(Note the difference with standard `++` in the recursive step!)

Compute the result of:

```
[1,2,3] ++/ [4,5]
```

Perform this computation **manually, using reduction**, as shown during the lecture. That is, put every reduction step on a new line, and underline what you rewrite. You are allowed to use an extra rewrite step for changing between *list notation* and using *list constructors*.

Exercise 3.5 (*Mandatory*: Extended list design pattern, `Uniq.hs`).
Sometimes the basic *list design pattern* of writing cases for `[]` and `(x:xs)` can feel limiting. You are always free to add more case distinctions! For example, you can decide to use a pattern of three cases: the empty list, the singleton list and a list consisting of two or more elements. With that in mind, write a function `uniq :: (Eq a) ⇒ [a] → [a]`, which removes duplicate contiguous items in a list, for example:

```
uniq "goodbye, hello" ⟹ "godbye, helo"
```

Exercise 3.6 (*Mandatory*: List comprehensions, `ListComprehensions.hs`).
List comprehensions are powerful tools, but reading them takes practice.

1. For each of these functions, think of a better name and explain briefly what it computes:

   ```
   g0 as bs  = [ (a, b) | a ← as, b ← bs ]

   g1 n y    = [ y | i ← [1..n] ]

   g2 n xs   = [ x | (i, x) ← zip [0..] xs, i < n ]

   g3 a xs   = [ i | (i, x) ← zip [0..] xs, x == a]

   g4 xs ys  = [ e | (x, y) ← zip xs ys, e ← [x,y] ]

   g5 xss    = [ x | xs ← xss, x ← xs ]
   ```

   (You already have seen some of them as a recursive definition. Whether you prefer those over a list comprehension can be a matter of taste.)

2. What types do these have? Which functions are fully polymorphic, and which ones are overloaded using a type class?

Exercise 3.7 (*Mandatory:* "Think big", Functional LEGO, `Lego.hs`).
Write the following functions **without explicit recursion**: they can be built out of list comprehensions and/or functions from the Haskell standard library. Using helper functions is allowed, but these too must be non-recursive.

1. The function `removeAt :: Int → [a] → [a]` that removes a single list element at a given index. For example: `removeAt 2 "abc" ⟹ "ac"`

2. A function `sortWithPos :: (Ord a) ⇒ [a] → [(a,Int)]` that sorts a given list, but also records the position each element had in the original list. For example:

   ```
   sortWithPos "haskell" ⟹ [('a',1),('e',4),('h',0),('k',3),('l',5),('l',6),('s',2)]
   ```

3. A function `sortedPos :: (Ord a) ⇒ [a] → [(a,Int)]` that returns the list given as an argument, but where each element is paired with the position it gets *if the list is sorted*:

   ```
   sortedPos "haskell" ⟹ [('h',2),('a',0),('s',6),('k',3),('e',1),('l',4),('l',5)]
   ```

   (Stuck? See Hint 3.)

Exercise 3.8 (*Mandatory*: Programming, `Obfuscate.hs`). Perhaps you know this old meme:[1]

> *Aoccdrnig to rscheearch at Cmabrigde uinervtisy, it deosn't mttaer waht oredr the ltteers in a wrod are, the olny iprmoetnt tihng is taht the frist and lsat ltteres are at the rghit pclae. The rset can be a tatol mses and you can sitll raed it wouthit a porbelm. Tihs is bcuseae we do not raed ervey lteter by it slef but the wrod as a wlohe.*

Implement a function `cambridge :: String → String` that transforms a text into this style; that is, for every word, jumble all the letters except the first and the last. Leave white space, punctuation and numbers as is. You can use any moethd you lkie to mix the lrteets in a wrod.

- This is a programming exercises, so divide this problem into smaller ones. The obvious approach is to break the string in a list of words, jumble each word, then put the string back together again.
- The `words` function is useful for breaking sentences, but only breaks on white space. You can use it to get a prototype solution, but to get `cambridge` perfect you may need to replace it with a `String → [String]` function you write yourself. *(If you don't succeed at that, please do use `words` and work on the other parts.)*
- Since Haskell functions must be *referentially transparent*, using unpredictable random numbers is impossible. Be creative and invent some way to deterministically shuffle a list!

Exercise 3.9 (*Extra*: Collapsing lists). A 'folding operation' is one in which a list is reduced to a single value. Examples are `sum` and `concat`. We will explore two patterns to create such functions.

1. Write a recursive function `concatr :: [[a]] → [a]` that for any list $[l_0, l_1, l_2 \ldots]$ computes $l_0{+}{+}(l_1{+}{+}(l_2{+}{+}\ldots)))$. So for example:

   ```
   concatr [l0]           ⟹ l0
   concatr [l0, l1]       ⟹ l0 ++ l1
   concatr [l0, l1, l2]   ⟹ l0 ++ (l1 ++ l2)
   concatr [l0, l1, l2, l3] ⟹ l0 ++ (l1 ++ (l2 ++ l3))
   ```

   and so on. For the case `concatr []`, make an appropriate choice yourself.

2. Write a recursive function `concatl :: [[a]] → [a]` that for any list $[l_0, l_1, l_2 \ldots]$ computes $(((l_0{+}{+}l_1){+}{+}l_2){+}{+}\ldots)$. So for example:

   ```
   concatl [l0]           ⟹ l0
   concatl [l0, l1]       ⟹ l0 ++ l1
   concatl [l0, l1, l2]   ⟹ (l0 ++ l1) ++ l2
   concatl [l0, l1, l2, l3] ⟹ ((l0 ++ l1) ++ l2) ++ l3
   ```

   and so on. For the case `concatl []`, make an appropriate choice yourself.

3. You probably guessed that `concatr` and `concatl` are equivalent. But which is more efficient?
4. Would `concatr` and `concatl` they still be equivalent if we replace `++` with the `++/` operator from Exercise 3.4?

[1] https://www.cambridgebrainsciences.com/more/articles/deos-it-mttaer-waht-oredr-the-ltteers-in-a-wrod-are

Exercise 3.10 (*Extra:* Problem solving, `DNA.hs`). Recall the representation of bases and DNA strands introduced in the lectures.

```
data Base  =  A | C | G | T
  deriving (Eq, Ord, Show)

type DNA      =  [Base]
type Segment  =  [Base]
```

1. Define a function `contains :: Segment → DNA → Bool` that checks whether a specified DNA segment is contained in a DNA strand. Can you modify the definition so that a list of positions of occurrences is returned instead? *(The function `tails` is useful here!)*
2. Define a function `longestOnlyAs :: DNA → Integer` that computes the length of the longest segment that contains only the base `A`.
3. *(Challenging!)* Define a function `longestAtMostTenAs :: DNA → Integer` that computes the length of the longest segment that contains at most ten occurrences of the base `A`.

   Friendly note: this exercise is properly difficult to solve efficiently! An efficient solution for it can rely on a sudden clever insight. Sadly, the more you concentrate, the less likely you are to get those insights... It is easier with a brute-force approach: write a function that generates all possible segments and work from there.

To test your functions on a large data set, use the mm1.dna file and the `'fileDNA'` operator:

```
>>> contains [C,A,T,G,A,G,A,C,T] `fileDNA` "mm1.dna"
True
>>> longestOnlyAs `fileDNA` "mm1.dna"
9
>>> longestAtMostTenAs `fileDNA` "mm1.dna"
182
```

Exercise 3.11 (*Extra:* Recursive comprehensions, `Partitions.hs`). The function `partitions` computes all *partitions* of a list *xs*. A partition of *xs* is a list $[A_1, \ldots, A_n]$ ($n \geq 0$), such that A_1 `++` $\ldots$ `++` A_n = *xs* and every A_i is non-empty. The empty list only has itself as possible partition. For example:

`partitions` $[\,]$	$= [[\,]]$
`partitions` $[x_0]$	$= [[x_0]]$
`partitions` $[x_0, x_1]$	$= [[x_0], [x_1]], [[x_0, x_1]]]$
`partitions` $[x_0, x_1, x_2]$	$= [[[x_0], [x_1], [x_2]], [[x_0], [x_1, x_2]], [[x_0, x_1], [x_2]], [[x_0, x_1, x_2]]]$

1. Give the most general type of `partitions`.
2. If *xs* is a list of *n* elements, how many elements does (`partitions` *xs*) have?
3. *(Challenging!)* Implement `partitions`. The result of this function should have the *same elements* as above. The *order of the elements* may differ.

Friendly note: this exercise can be a brain twister, since a 'very nearly correct' solution still computes completely wrong results, and there exist several creative ways to solve it.

Hints to practitioners 1. The helper function `rev` is written in a style called *tail recursive*, meaning that its recursive step does not involve any "post-processing" after the function calls itself recursively; in this style an *accumulator* is necessary to produce a function result.

Tail recursive functions can be compiled into ordinary loops, which can be important in other functional languages such as StandardML, OCaml, Scheme, Racket, Scala... In Haskell, things are more subtle, and worrying about creating tail-recursive functions is usually not worth the effort.

For more, see https://wiki.haskell.org/Tail_recursion.

Hints to practitioners 2. Do not confuse `[Base]` with the singleton list `[base]`. The first is a type; the second is a value, a shorthand for `base:[]`. Identifiers for values and identifiers for types live in different name spaces. Hence it is actually possible to use the same name for a type variable and for a value variable, for example

```
insert :: (Ord a) ⇒ a → [a] → [a]
insert a []    =  [a]
insert a (b : xs)
  | a ≤ b      =  a : b : xs
  | otherwise  =  b : insert a xs
```

The occurrence of `a` on the first line is a type variable; the occurrence of `a` on the second line is a value variable of type `a :: a`. If you think that this is confusing (we agree!), simply use different names, e.g.

```
insert :: (Ord obj) ⇒ obj → [obj] → [obj]
```

However, keep this "feature" in the back of your head, if you read other people's code. There is an unfortunate tendency to use the same names both for types and elements of types.

Hints to practitioners 3. If you've found simple and elegant solutions to the first two, you may expect there is also one for `sortedPos`. We ourselves haven't found one yet. Try constructing a functional pipeline, and you are allowed to re-use other functions you wrote.

Note that everything you can do with `map`/`filter`, you can also do with a list comprehension, but sometimes `map`/`filter` may produce more readable code.