1 What is the type?

You can use `:t` in the interpreter to see the type of an expression, however, try to use it only after you've thought about and maybe written down the type yourself!

Try to write the type of the following values

- `['a', 'b', 'c']`
- `('a', 'b', 'c')`
- `(True, 'a')`
- `[(True, 'a'), (False, 'b')]`
- `([True, False], ['a','b'])`
- `tail`
- `reverse`
- `[tail, reverse]`

2 What was the type?

Try looking at some of the following functions that we have used previously, and understand their types, for now ignore the `Foldable t` constraint, and instead see the type `t a` as `[a]`. Make your guess first, then check it with `:t`.

- `length`
- `head`
- `null`
- `take`
- `maximum`
- `sum`
- `elem`
- `repeat`
- `cycle`
- `succ`

3 Let and where

For some functions it is highly advantageous to use `let..in` or `where`.

E.g.

```
maximum :: Ord a -> [a] -> a
maximum [x] = x
maximum (x:xs)
    | x < y     = y
    | otherwise = x
```

```
    where
        y = maximum xs
```

Or equivalently, with `let..in`

```
maximum :: Ord a -> [a] -> a
maximum [x]    = x
maximum (x:xs) =
    let
        y = maximum xs
    in case (x < y) of
        True  -> y
        False -> x
```

Note that in the `where` example, we use `guards` to do the `x<y` check, while in `let..in` we ended up using `case..of`. However, both `where` and `let..in` can, and often should, be used regardless of the need for different cases.

To familiarize yourself with `let..in` and `where` make a function that calculates the roots of a quadratic equation (andengradsligning), i.e. given the coefficients a, b and c, calculate the roots x1 and x2 (assume there are two, and let the answer be given as a tuple). The function should have a signature similar to `roots :: Floating t => t -> t -> t -> (t,t)`.

Two examples of this could be:

```
> roots 2 5 3
(-1.0,-1.5)
> roots 2 2 (-4)
(1.0,-2.0)
```

- Implement the function twice, once with `let..in` and once with `where`. You should as a minimum calculate the discriminant in your `let` or `where` part.

4 Make the function

For the following functions, write the type for the ones thatt are missing it, and implement all of them. You are welcome to use functions you have seen or made previously (this does not include `:t`).

- `second`, takes a list and returns the second element.
- `secondLast`, takes a list and returns the second to last element, e.g. `secondLast [1,2,3,4]` is 3.
- `swap`, takes a 2-tuple and returns a tuple with the elements in reverse order.
- `pair`, takes two elements and returns a tuple of them.

- `palindrome`, takes a list and returns whether it is the same forwards and backwards. (Hint: use `reverse` in the implementation)
- `twice`, takes a function and an element the function can be applied to, and applies it twice.
- `flatten`, takes a list of lists and returns a list where the elements are **not** in nested lists, e.g. `flatten [[1,2,3],[42],[],[7,6]]` should output `[1,2,3,42,7,6]`. It has the type signature `flatten :: [[a]] -> [a]`.
- `alternate`, takes a list of numbers and returns the list where every other element is negative, e.g. `alternate [1,3,4,2,5]` should output `[1,-3,4,-2,5]`.
- `setIdx`, takes a list, an element and a number (index), and returns the list where the element at the index is set to the given element.
- `modIdx`, takes a list, a function and a number (index), and returns the list where the element at the index is modified with the given function. It should have a type signature similar to `modIdx :: [a] -> (a -> a) -> Int -> [a]`.
- `unique`, takes a list and returns a list of the unique elements in the list. This can be seen as removing duplicates.

5 What is the type? (2)

Try to write the type of the following expressions. What do they do?

- `( + ) 42`
- `( : ) 3`
- `( , ) 'x'`
- `( , , ) "hi"`

6 Suggest the type

Suggest types for the following function definitions.

- `one x = 1`
- `apply f x = f x`
- `compose f g x = f ( g x )`

7 Show me what you're hiding

We usually use `show` to turn something into a `String`, this is part of what implicitly happens in the interpreter, so we can see the result on the screen. You can try entering a function with no arguments in the interpreter to see an error about this, because it doesn't know how to show a function.

- Try making a simple function that takes some input and returns a string that shows that input, e.g. `plustalk` that takes two integers and makes the string `"x plus y is z"` where `x,y,z` are the numbers in the calculation.
- Make a function `fizzbuzz` that takes an integer and prints `"Fizz"` if the number is evenly divisible by 3, `"Buzz"` if it is divisible by 5, and `"FizzBuzz"` if it is divisible by both. Otherwise it returns the number as a string, which is done with the `show` function.
  - Use `fizzbuzz` in a list comprehension to see what happens on the first 30 integers.

8 Challenges

Here are some challenges that should be possible with the part of the language we know currently. Most are taken from https://wiki.haskell.org/H-99:_Ninety-Nine_Haskell_Problems. I've chosen the ones I found interesting, but you are welcome to go look, and see if there are others you like.

Note that they fluctuate in difficulty, the difficulty is not strictly increasing.

Implement the following.

- `compress`, takes a list and drops consecutive duplicates, e.g. `compress [1,1,1,2,2,3,1,1,4]` is `[1,2,3,1,4]`.
- `pack`, takes a list and packs consecutive duplicates into sublists, e.g. `pack [1,1,1,2,2,3,1,1,4]` is `[[1,1,1],[2,2],[3],[1,1],[4]]`.
  - `encode`, takes a list (string) and makes so called run length encoding of it. This means making a list of tuples `(N,E)` where `N` is the number of times the element `E` occurs consecutively.
  - `decode`, takes a list of tuples (on the same form `encode` produces), and creates the corresponding list (string) that was encoded.
- `doubleUp`, takes a list and duplicates each element, e.g. `doubleUp [1,2,3]` is `[1,1,2,2,3,3]`.
- `slice`, takes a list and two indices, and returns the elements between the indices (you can choose whether you want the indices to be inclusive or exclusive), e.g. `slice ['a','b','c','d','e'] 1 3` is `['b','c','d']` if both indices are inclusive (remember it's 0-indexed).
- `rotate`, rotates a list k spaces to the left.
- `deleteAt`, deletes the element at the i'th position.
- `insertAt`, inserts an element at the i'th position.
- `range`, makes a list of the numbers from i to j (increasing).
- `tuples`, makes a list of all pairs of elements in a given list, where an element can be paired with itself, e.g. `tuples ['t','f']` is `[('t','t'), ('t','f'), ('f','t'), ('f','f')]` or a permutation hereof.
- `pairs`, makes a list of all pairs of elements in a given list, where an element **cannot** be paired with itself, e.g. `pairs [1,2]` is `[(1,2),(2,1)]` or a

permutation hereof.

- `combinations`, make a generalization of `pairs`, that creates lists instead of tuples and takes a k which is the number of elements chosen from the given list. E.g. the first few elements of `combinations 3 ['a'..'d']` is `[['a','b','c'],['a','b','d'],['a','c','b'],...]`