Tutorial 2B: Type classes

Type variables

Many functions on lists work on **any** kind of list: lists of numbers, lists of strings, lists of lists of numbers, etc. Such a function will use a type `[a]`, where `a` is a **type variable**. For example, the library functions `take` and `drop`, which respectively take or drop the first n elements from a list, have the types:

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
```

When a function is applied to a list of elements of a specific type, the interpreter replaces the variable with that type. For example, in the expression `take 3 [1,2,3,4,5]` the type of `take` may become `Int -> [Int] -> [Int]`.

Exercise 1:

a) Complete the function `ditch` which behaves as `drop`: it should remove the first n items from a list.

b) Complete the function `at` which returns the item at the ith position of a list, where the head is zero. If the i is negative, or equal to or greater than the length of the list, return an error. This function exists in Haskell as `(!!)`. **Note:** a (prefix) function can be used **infix** by using **backquotes**, as in `` `at` ``.

```
*Main> ditch 2 gentlemen
["George","William"]
*Main> ladies `at` 3
"Elizabeth"
```

Type classes

If you ask the interpreter for the type of a number, rather than a definite type such as `Int` or `Integer`, you get a type variable with a **constraint** `Num a =>`.

```
*Main> :t 42
42 :: Num a => a
```

`Num` is a **type class**: a group of types, including `Int` and `Integer` but also types like `Float`, that all represent numbers of some kind. The interpreter may specialize `42` to any such type, but not others, such as `Bool` or `String`. You can ask the interpreter to do this by specifying the type you want:

```
*Main> 42 :: Int
42
*Main> :t it
it :: Int
*Main> 42 :: Bool
<interactive>: error:
    • No instance for (Num Bool) arising from the literal ‘42’
    • In the expression: (42 :: Bool)
```

The purpose of having a class `Num` is to have certain functions, such as addition and multiplication, available for all `Num` types:

```
*Main> :t (+)
(+) :: Num a => a -> a -> a
```

This is very convenient (and indeed essential in practice), because otherwise there would have to be a different addition function for each type. Making a function, such as `(+)`, available for a select group of types is called **ad-hoc polymorphism**.

Here, we will see two more type classes:

- `Eq` is the class of types for which Haskell can compute **equality**; that is, for which the equality function `==` is defined, and the related inequality `/=`.
- `Ord` is the class of types that have an **ordering**, for which the functions `<=` and `==` are defined (all types in `Ord` are also in `Eq`), and related functions `<`, `>`, and `>=`.

The classes `Eq` and `Ord` include `Bool`, `Int`, `Integer`, `String`, and more generally every type `[a]` when also `a` is in `Eq` respectively `Ord`. But typically not `a -> b`.

A **type constraint** `Eq a =>` or `Ord a =>` preceding a type indicates that the type variable `a` stands for any type belonging to the class `Eq` respectively `Ord`. These feature in the following exercises.

Exercise 2:

a) Complete the function `find` which given an item of type `a`, looks up the related item of `b` in a list of pairs `[(a,b)]`. If there is no related item, raise an error; if there are more than one, return the first.

b) Complete the function `which` that returns at which position an item first occurs in a list, starting from zero. For example, `which 20 [0,10,20,30,40]` should return `2`. If the item does not occur, raise an exception. Use the auxiliary function `aux` to count along while trying to find the matching item.

c) Copy the functions `member`, `remove`, and `before` from Tutorial 2A, and change their type so that they work for as many lists as possible, using type variables and the appropriate constraints `Eq a =>` or `Ord a =>`.

d) Complete the function `sorted` from Tutorial 2A again, but for the given type. If you had used `before` before, what do you need to adjust, and why?

```
*Main> find "Elizabeth" couples
"Fitzwilliam"
*Main> which "George" gentlemen
2
*Main> sorted couples
False
```

Merge sort

Our example lists `ladies` and `gentlemen` represent collections of fictional characters. In this context, we would expect two properties: 1) names don't occur more than once (if we happen to have two Fitzwilliams, say, we would expect to add a way to distinguish them, such as a title), 2) the order in which names are listed doesn't matter. In other words, we would like our example lists really to be **sets**.

A better way to represent sets is by making our lists **ordered** (i.e., sorted, from small to large). Several options then become computationally much cheaper: combining two sets while removing duplicates, and subtracting one set from another.

We will implement a **set** as a list that is **ordered** and **non-repeating**. Combining two sets will be by the function `merge`, and subtracting by `minus`. The `merge` function is part of the efficient **merge sort** algorithm, which is well-suited to lists. It works by splitting a list in half, recursively sorting each half, and merging them again.

Exercise 3:

a) Complete the `merge` function which combines two ordered, non-repeating lists into one, which holds all elements that occur in either.

Hint: The pattern-matching is already set up, with three cases: two where one list is `[]`, and one where both are non-empty. For the latter case, use guards to compare the heads of both lists. If one element is strictly smaller, that should become the head of the new list, and `merge` should recurse on the remaining data (so the tail of one list and the whole of the other). If they are equal, take one, discard one, and recurse on both tails.

b) Complete the `minus` function so that `minus xs ys` on ordered, non-repeating lists returns `xs` with all elements from `ys` removed.

c) Complete the `msort` function that implements merge sort. For an input list `xs` the algorithm is as follows:

- If `xs` is empty or has only one element, it is sorted.
- Otherwise, split `xs` into (almost) equal halves `ys` and `zs`.
- Recursively sort `ys` and `zs`.
- Combine the two results with `merge`.

Use the following functions: `length` to get the length of the list, `div` to divide that by two, and `take` and `drop` to get the first and second half of the list. Recursively sort the two halves, and use `merge` to combine them again. If you like, you can try to improve your solution using a **where**-clause, the function `splitAt`, or by finding a way to split the input without measuring its length first.

```
*Main> merge gentlemen ["Edward","Henry","John"]
["Charles","Edward","Fitzwilliam","George","Henry","John","William"]
*Main> it `minus` ["Charles","George","John"]
["Edward","Fitzwilliam","Henry","William"]
*Main> msort ladies
["Elizabeth","Jane","Kitty","Lydia","Mary"]
*Main> sorted it
True
```