Tutorial 2: Types and More Recursion

In this tutorial we will practice more recursion on lists. We will look at recursion on two lists simultaneously, pattern-matching on more than one element, and working with lists of pairs. Finally, we will implement the merge sort algorithm. Along the way, we will introduce **type synonyms**, **type variables** and simple **type classes**.

Exercise 1:

a) Complete `before xs ys` that returns `True` if the string `xs` comes before `ys` in the dictionary. Use guards for the cases `x < y`, `x == y`, and `otherwise`.

b) Give a version `before’` using boolean operations (`||`) and (`&&`) instead of guards.

c) Complete the function `sorted` that tests if a list of strings is ordered alphabetically. Make three cases: empty; one item; and two or more items. Test your code:

```
*Main> before "Daario" "Grey Worm"
True
*Main> sorted party1
True
*Main> sorted party2
False
```

Type synonyms

The next part of the game that we will implement is the map. We will represent each game location by an integer, and the map by a list of pairs of integers, `[(Int,Int)]`. To avoid any confusion with all the other integers that are usually floating around in a program, we give a new name to these types, a **type synonym**:

```
type Node = Int
type Map  = [(Node,Node)]
```

The keyword `type` defines a new name for an existing type, and here we use it to create the types `Node` and `Map`. Using these types, you are given the map of the game, `theMap` :: `Map` (since `map` is taken). There is also a list `locations` with a name for each location, and a list `characters` of game characters at each location.

Exercise 2:

a) Create the types `Location` and `Character` as a synonym for `String`, and the synonym `Party` for a list of characters, and use these to give a type signature for `locations` and `characters`.

b) Complete the function `oneWay` which takes a `Node l` and a `Map`, and returns the list of locations `[Node]` that can be reached from `l` in one step on the map. Here, a pair `(x,y)` in the given `Map` is only one-way, from `x` to `y`.

c) Complete the function `bothWays`, which is like `oneWay` except the map can be traversed in both directions.

```
*Main> oneWay 5 theMap
[6]
*Main> bothWays 5 theMap
[2,3,6]
*Main> bothWays 0 theMap
[]
```

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]
```

Internally, the interpreter replaces the variable with a suitable type, so that in the expression `take 3 [1,2,3,4,5]` the type of `take` becomes `Int -> [Int] -> [Int]`.

Exercise 3:

a) Complete the function `at` which returns the item at the i th 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 `(!!)`. (It can be used as an **infix** operator with **backquotes** as `‘at‘`.)

b) Given a character name, we would like to find out at which `Node` in the map it is. We will do so with the function `findNode`. Given a character `c`, it should search for `c` in every `Party` in `characters` (use the built-in function `elem` or last tutorial's `member`). When found, it should return the index of that party—this means we need to keep track of the current `Node`. Since we are keeping track of a `Node` and recursing on a list `[Party]`, these are separate inputs to `findNode`, though we expect to call the function itself only with inputs `0` and `characters`.

c) Use `findNode` and `at` to complete the function `findCharacter` which tells you the location of a character as given by `locations` and `characters`. It would be good to move `findNode` into a **where**-clause. (There is also a solution without `at`, by adapting the function `findNode` to give the `Location` directly.)

```
*Main> at locations 2           -- also: locations `at` 2
"Kings Landing"                 -- or:   locations  !!  2
*Main> findNode 0 "Bran" characters
6
*Main> findCharacter "Bran"
"Winterfell"
```

There are two new, commented-out type signatures for `oneWay` and `bothWays`:

```
oneWay   :: Eq a => a -> [(a,b)] -> [b]
bothWays :: Eq a => a -> [(a,a)] -> [a]
```

The **constraint** `Eq a` means that the variable `a` may be instantiated only with types that belong to the **type class** `Eq`, "equality". That is because the functions use an equality test `x == y`. The function (`==`) exists for most types, such as `String`, and `Int`, but (importantly) not for function types, such as (`[Int] -> Int`). This would be impossible: it is **undecidable** whether two functions are equal! The **type class** `Eq` is the collection of all types for which (`==`) and (`/=`) are defined, and the **constraint** `Eq a =>` means that the type used for `a` must belong to that class.

The type class `Ord` ("ordered") is the collection of all types for which the functions (`==`) and (`<=`) are defined. This also gives the derived functions (`>=`), (`<`), and (`>`). Observe that this includes (`==`): any type belonging to `Ord` also belongs to `Eq`.

Exercise 4:

a) Replace the original signatures for `oneWay` and `bothWays` with the new ones. Remove the constraint `Eq a =>` from one of both functions, and try to load the file. What error do you get? Put back the constraint.

b) Go back to the functions of Exercise 1 and change the type of `before` to work for any list type `[a]`. What is the problem? Add the constraint `Ord a` to fix it.

c) Change the type of `sorted` so that it works for lists of lists `[[a]]` (with `Ord a`).

Merge sort

A `Party`, a group of our game characters, should be a **set**: order doesn't matter ("Jamie and Brienne" is the same group as "Brienne and Jamie") and neither do repetitions ("Sansa and Sansa” is just “Sansa” — unlike e.g. “a bag of gold and a bag of gold”). We will implement a **set** as a list that is **sorted** and **non-repeating**. The function `merge` will combine two sets, and the function `minus` will remove the elements of one set from another. 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 5:

a) Complete the `merge` function. Merging any list `xs` with an empty one returns `xs`, which gives the two base cases. For the recursive call, where `x` and `y` are the head of each input list, there are three cases. If `x < y`, put `x` at the head of the result, and recurse on the remaining lists (including `y`), and similarly for `x > y`. If `x == y`, put either `x` or `y` at the head of the result, discard the other, and then recurse on the remaining lists (without `x` and `y`).

b) Complete the `minus` function. A call `minus xs ys` on ordered, non-repeating lists should return `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 party1 (characters ‘at‘ 6)
["Bran","Catelyn","Hodor","Rickard","Robb","Sansa","Theon"]

*Main> minus it ["Arya","Bran","Sansa"]
["Catelyn","Hodor","Rickard","Robb","Theon"]

*Main> msort party2
["Daario","Daenerys","Grey Worm","Jorah","Missandei","Tyrion"]

*Main> sorted it
True
```