Lists

In this week's tutorials we will practice recursion on lists, the standard data structure in Haskell. Lists are defined inductively with two **constructors**: the empty list **nil**, `[]` , and a **cons** `x:xs` of a **head** `x` and a **tail** `xs` .

```
[]      -- the empty list, "nil"
x:xs    -- a "cons" of a head x and a tail xs
```

A list with elements of type `a` has itself type `[a]` . In a cons `(x:xs)` :: `[a]` the types are then `x` :: `a` and `xs` :: `[a]` . In this way, the constructors nil `[]` and cons `(:)` can then themselves be typed:

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

Functions can **build** lists using these constructors, and also **take them apart**. This is called **pattern-matching**. For example, a function that does absolutely nothing is:

```
nothing :: [a] -> [a]
nothing []     = []
nothing (x:xs) = x : xs
```

Exercise 1:

a) Write the function `times` which multiplies the integers in a list. First, give a type signature. Use the given function `add` as an example.

b) A useful pre-defined list is `[n..m]` which counts up from `n` to `m` . Complete the function `range` so that `range n m` behaves as `[n..m]` . First, give a type signature.

c) Write `factorial` by combining `range` and `times` .

```
*Main> times [1,2,3]
6
*Main> range 4 9
[4,5,6,7,8,9]
*Main> factorial 10
3628800
```

Type variables

Some list functions work the same regardless of what type of elements the list contains. We use type variables to allow these functions to work on any kind of list.

Exercise 2:

a) Write the function `count` to count the number of elements in a list, like the built-in function `length`. Note the underscore `_` in the pattern: this is a **wildcard**, to indicate that this part of the pattern will not be used.

b) Write the function `append` to append one list to the end of another, like the built-in function `(++)`.

c) Write the function `concatenate` that takes a list of lists, and flattens it to a single list, like the built-in `concat`. Use your function `append` or `(++)`.

```
*Main> count [4,8,3,5,2]
5
*Main> append [4,5,9] [0,1,7]
[4,5,9,0,1,7]
*Main> concatenate [[9,4,5],[],[2,4],[6]]
[9,4,5,2,4,6]
```

Pattern matching + guards

Each pattern-matching case can be made conditional with **guards**.

Exercise 3:

a) Write the function `member` which determines whether an integer occurs in a list, like the built-in function `elem`. Write two versions: one using guards, and one using boolean operators (`||` , `&&`) instead.

b) Write the function `remove` which removes all occurrences of a given integer from a list. Give a type signature first.

c) Write the function `at` so that `at xs i` takes the i th element of the list `xs`, like the built-in function `(!!)`. The first element should have index 0. Throw an exception when the index is negative or too large (but don't compute the length first).

```
*Main> member 4 [1..5]
True
*Main> remove 2 [1,2,3,2,1]
[1,3,1]
*Main> at [5,7,1,0,4] 2
1
```

Note that a regular, prefix function can be used **infix** (i.e. in-between its two arguments, like an arithmetic operator) by surrounding it with **backticks**, as follows.

```
*Main> [5,7,1,0,4] `at` 2
1
```

Nested patterns

A function can pattern-match for each argument, and patterns can be **nested**: a function can match on patterns inside patterns, inside patterns, etc.

Exercise 4:

a) Write the function `final` to take the last element of a list, like the built-in function `last`. You will need an additional pattern-matching case for a list with one element, which you can write as `final [x]` or `final (x:[])`.

b) Write a function `ordered` that determines whether the integers in a list are in increasing order. You will need three cases: an empty list, a list with one element, and a list with two or more elements, which you can match as `ordered (x:y:zs)`.

c) Write a function `pair` that takes two lists and makes a list of pairs, of the first element of each list, then the second element of each, then the third etc. This is built-in as the function `zip`. You will need three pattern-matching cases: when the first list is empty (return the empty list), when the second list is empty (same), and when both lists are non-empty.

d) Write a function `find` that, given an integer `i` and a list of pairs `(Int,String)`, finds the matching pair `(i,str)` in the list and returns `str`. If there is no match, return the empty string (`""`).

```
*Main> final [3,4,2,7]
7
*Main> ordered [1..10]
True
```

```
*Main> ordered [3,6,0,4]
False
*Main> pair [1..10] ["a","b","c","d","e"]
[(1,"a"),(2,"b"),(3,"c"),(4,"d"),(5,"e")]
*Main> find 3 it
"c"
```

Type classes

We have used type variables for functions that work uniformly on lists, such as `length` and `(++)`. For functions that work for example uniformly on numbers, such as `(+)` and `(*)` which apply to `Int`s, `Integers`, `Floats`, and other number types, there is another solution. Just using a type variable is not correct, since these functions don't work on **all** types, just on **number** types. This is expressed by the **type class** called `Num`:

```
(+) :: Num a => a -> a -> a
```

The part `Num a=>` of the type declaration is a **type constraint** or **class constraint**. This type declaration states that `(+)` works on any type `a` that is a number type, `Num a`. Note that both arguments must still have the same type: `(+)` cannot add an integer to a float, for example.

We will consider four classes:

- `Num` for number types, with for example the functions `(+)`, `(*)`, and `(-)`;
- `Ord` for types whose elements have an ordering, with for example the functions `(<=)`, `(>=)`, and `max`;
- `Eq` for types where we can test elements for equality, with the functions `(==)` and `(/=)`;
- `Show` for types whose elements can be displayed as a string, with the function `show`, for printing to screen.

Note that there are types where we can't test for equality, for instance function types such as `Integer -> Integer`. Similarly, we can't display functions as strings.

Here are the type signatures with class constraints of some of the above functions.

```
 (<=) :: Ord a  => a -> a -> Bool
  max :: Ord a  => a -> a -> a
 (==) :: Eq a   => a -> a -> Bool
 show :: Show a => a -> String
```

If you try to use a function with class constraint, say `show`, on a type `a` that doesn't satisfy the constraint, you will get a type error that says:

```
* No instance for (Show a)
  arising from a use of 'show'
```

Exercise 5:

a) Try to generate a few such type class errors to see if you can understand them. For instance, try `True + False`. What about `True < False`?

b) Give appropriate type signatures with class constraints to the following functions of the previous exercises:

- `times`
- `member`
- `remove`
- `ordered`
- `find`

c) Some type classes include others. For instance, `Ord` includes `Eq`, so that a constraint `Ord a =>` allows you to use not just `(<=)` but also `(==)`. Do some experiments to figure out if `Num` includes `Ord` or `Eq`.

d) Multiple constraints can be given as a tuple, for instance: `(Eq a, Show a) =>`. Use this to give an appropriate type signature for the function `range`.

Extra: Merge sort

An important class of algorithms are **sorting algorithms**. Sorted data is much easier to handle than unsorted data, though the benefit is generally greater for other data structures such as trees, which we will see later, than for lists.

A natural and efficient sorting algorithm for lists is **merge sort**. We will implement it here. The algorithm is as follows:

- **Merge** two ordered lists into one by:
  - Inspecting the first element of each
  - Taking the smaller one as the first element of the resulting list
  - Recursively merging the remainder of the two lists
- **Merge sort** a list by:
  - Splitting it in two equal parts
  - Recursively sorting each
  - Merging the results

With ordered lists, one consideration is whether to have duplicate elements. Are we working with **repeating** lists, which may hold multiple of the same element, or with **non-repeating** lists, which hold at most one of each element? Here, we choose non-repeating.

Exercise 6:

a) Complete the `merge` function which combines two ordered, non-repeating lists into one. Give an appropriate type signature first. Decide what to do when you encounter two equal elements.

b) Complete the `msort` function that implements merge sort. Give an appropriate type signature first.

The following functions will be helpful: `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.

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 [1,2,4,5,6,9] [2,3,6,7,8]
[1,2,3,4,5,6,7,8,9]
*Main> msort [5,7,1,4,9,2,8,6,3]
[1,2,3,4,5,6,7,8,9]
```

Challenge: Towers of Hanoi

The Towers of Hanoi is a puzzle with three stacks of tokens of increasing size. Starting with one full stack and two empty stacks, the challenge is to move the full stack onto the last (empty) stack without ever putting a larger token on top of a smaller one.

Representing tokens as integers, and the three stacks as lists of integers, the solution for a stack of size three progresses as follows:

```
[1,2,3]     []       []
  [2,3]     []      [1]
    [3]    [2]      [1]
    [3]  [1,2]       []
     []  [1,2]      [3]
    [1]    [2]      [3]
    [1]     []    [2,3]
     []     []  [1,2,3]
```

We can express the solution for a given size by the following recursive algorithm. To move n tokens from stack x to stack z:

- move $n-1$ tokens from stack x to stack y;
- move one token from stack x to stack z;
- move $n-1$ tokens from stack y to stack z.

The base case is to move zero tokens, which is to do nothing. The example above, for size three, first moves two tokens to the middle stack (in the first 3 steps), moves one token to the third stack, and then moves two tokens from the middle to the third stack.

Exercise 7: Implement the function `hanoi` that solves the Towers of Hanoi puzzle for any size, returning the solution as a list of states, each state a triple of stacks.

```
*Main> hanoi 3
[([1,2,3],[],[]),([2,3],[],[1]),([3],[2],[1]),([3],[1,2],[]),
([],[1,2],[3]),([1],[2],[3]),([1],[],[2,3]),([],[],[1,2,3])]
*Main> hanoi 4
[([1,2,3,4],[],[]),([2,3,4],[1],[]),([3,4],[1],[2]),
([3,4],[],[1,2]),([4],[3],[1,2]),([1,4],[3],[2]),
([1,4],[2,3],[]),([4],[1,2,3],[]),([],[1,2,3],[4]),
([],[2,3],[1,4]),([2],[3],[1,4]),([1,2],[3],[4]),
([1,2],[],[3,4]),([2],[1],[3,4]),([],[1],[2,3,4]),
([],[],[1,2,3,4])]
```