Compound Haskell Types

Compound Types

There are a number of ways of producing more complex types out of simpler types. These are some times called **compound types**.

List Types The first example of that is the list type. As elements of the list all must have the same type, we can specify the type of a list with two pieces of information:

- The fact that it is a list. This is denoted by using a single pair of square brackets: `[...]`
- The fact that the entries have a certain type. That type goes between the brackets.

```
[False, True, True, True] :: [Bool]
['a', 'b', 'c'] :: [Char]              -- Can also be called String
"abc" :: [Char]                        -- Same as above
["abc", "def"] :: [[Char]]             -- Or also [String]
```

Practice: Write a value of type `[[Int]]`.

Tuple Types A **tuple** is a collection of values separated by commas and surrounded by parentheses. Unlike lists:

- A tuple has a fixed number of elements (fixed *arity*), either zero or at least two.
- The elements can have different types, from each other.
- The types of each of the elements collectively form the type of the tuple.

Examples:

```
(False, 3) :: (Bool, Int)
(3, False) :: (Int, Bool)      -- This is different from the one above
(True, True, "dat")  :: (Bool, Bool, [Char])
() :: ()                       -- The empty tuple, with the empty tuple type
```

We write functions for tuples by using what is known as **pattern-matching**:

```
isBetween :: (Int, Int) -> Int -> Bool
isBetween (a, b) c = a <= c && c <= b

-- Example use:    isBetween (2, 5) 3  returns true
```

What is happening in the example is that the pair `(2, 5)` is *matched* against the *pattern* `(a, b)` and as a result `a` is set to `2` and `b` is set to `5`. The pair is still considered a single input to the function (thus making two inputs together with the other integer), but it ends up having its parts bound to different variables via the pattern-matching process. We will return to this soon.

We can also mix list types and tuple types. For instance:

```
[(1, 2), (0, 2), (3, 4)] :: [(Int, Int)]        -- A list of pairs of integers
 -- A list of pairs of strings and booleans
[("Peter", True), ("Jane", False)] :: [([Char], Bool)]
```

Practice

Write the types we might use to represent the following information:

1. A person with first and last name, age, and information about whether they can drive or not.
2. Many persons as in the previous part.
3. The record of a college student, containing their name, username, and a list of the courses they have taken and the grades.
4. The ingredients for a recipe.

Function types A function type is written as `A -> B` where `A` is the type of the input and `B` is the type of the output. For example:

```
add3 x = x + 3              :: Int -> Int
add (x, y) = x + y          :: (Int, Int) -> Int
oneUpTo n = [1..n]          :: Int -> [Int]
range (a, b) = [a..b]       :: (Int, Int) -> [Int]
```

When writing functions, we tend to declare their type right before their definition, like so:

```
range :: (Int, Int) -> [Int]
range (a, b) = [a..b]
```

You may be tempted to think of this function as a function of two variables. It technically is not, and we will discuss this topic on the next section.

Type Practice

Work out the types of the following expressions:

1. `(5 > 3, 3 + head [1, 2, 3])`
2. `[length "abc"]`
3. The function `f` defined by `f lst = length lst + head lst`
4. The function `g` defined by `g lst = if head lst then 5 else 3`

Write the types for the following functions:

0. A function that takes as input a list of integers and returns the list but sorted.
1. A function that is given a list of numbers and returns the smallest and largest number.
2. A function that takes as input a pair of "ranges", where a "range" is itself a pair of integers, and returns whether the first range is contained in the other.
3. A function that takes as input a list of people names and ages as well as a cutoff age, and returns a list of the names for those people whose age passes the cutoff.