SET07106 - Exercise sheet 4
Recursion II and Functions

NOTE: This practical introduces some new Haskell syntax, but in a very focussed way. It should be read in conjunction with one of the Haskell textbooks found on Moodle, or with this online version of Learn You a Haskell, in particular the section on tuples.

The type signatures of all of the functions you will need to define already exist inside the file `exercise4.hs`. You just need to fill in the gaps then uncomment the function. The gaps for some functions are larger than for others. There are also a collection of lists pre-defined, that you can use for testing.

Recursion on the natural numbers

We saw some examples last week of recursion using natural numbers, for example `sumUpTo` and `fac`. This week we will look at some more complicated functions in which the natural number input to a function will be used as an index variable. Consider the problem of returning the element at a particular index in a list. We could create a counter that starts at 0, and increments as we move through a list until it reaches a given value. However, that is cumbersome, since we would need to pass the function around with two number inputs: the value of the counter and the value it needs to get to.

Consider the recursive approach. We count down from the index we want, at each stage throwing elements away from our list, until the counter reaches 1, at which point we return the element:

```
elemAt ::  Int -> [a] -> a
elemAt 1 (x:xs) = x
elemAt n (x:xs) = elemAt (n-1) xs
elemAt n [] = error ''Not enough objects in list!''
```

Note that, because of the way the compiler works, the bottom clause will only be called if the top two are **not** matched.

The next few exercises develop the above pattern to help us adapt lists.

Exercise 1 *Write the function* `insertAt`*. This function takes an index n, an object x of type* `a` *and a list ys with type* `[a]`*, and returns a list with type* `[a]`*. The behaviour is that the returned list is the same as the original, except containing x at index n. The remainder of ys should appear. For example,* `insertAt 3 7 [2,4,6,8]` *should return* `[2,4,3,6,8]`*.*

Exercise 2 *Write the function* `deleteAt`*. This function takes an index n and a list ys with type* `[a]`*, and returns a list with type* `[a]`*. The behaviour is that the returned list is the same as the original, except the element at index n from ys has been removed. For example,* `removeAt 3 [2,4,6,8]` *should return* `[2,4,8]`*.*

Exercise 3 *Write the function* `takeUpTo`*. This function takes an index* n *and a list* ys *with type* `[a]`*, and returns a list with type* `[a]`*. The behaviour is that the returned list is the first* n *elements of* ys*. For example,* `takeUpTo 4 [1..10]` *should return* `[1,2,3,4]`*. (There is an inbuilt function called* `take` *which does the same thing.)*

Exercise 4 *Write the function* `takeAfter`*. This function takes an index* n *and a list* ys *with type* `[a]`*, and returns a list with type* `[a]`*. The behaviour is that the returned list is the list* ys ***without*** *the first* n *elements. For example,* `takeAfter 4 [1..10]` *should return* `[5,6,7,8,9,10]`*.*

Exercise 5 *Using the functions* `takeUpTo` *and* `takeAfter`*, write the function* `takeBetween`*. This function takes an index* n*, an index* m*, and a list* ys *with type* `[a]`*, and returns a list with type* `[a]`*. The behaviour is that the returned list is the sublist of* ys *but which starts at index* n *and ends at index* m*. For example,* `takeBetween 3 6 [1..10]` *should return* `[3,4,5,6]`*.*

Exercise 6 *(Harder)* ***Without*** *using the functions* `takeUpTo` *and* `takeAfter`*, write the function* `takeBetween`*. In other words, do it by recursion on the two indices themselves.*

Functions

We saw in the lectures that functions could be thought of as sets of tuples. A function with one input and one output would be a set of pairs. A function with two inputs and one output would be a set of triples, and so on. In this practical, we are going to write a series of functions that test whether or not a given list of pairs represents a function, and further whether that function is injective, surjective or bijective.

We will focus on unary functions (functions with one input and one output, i.e. of the form $f : A \to B$) for two reasons. Firstly, they are the easiest to deal with. Secondly, every function with a higher arity can be turned into a series of unary functions. Read Curried functions for an explanation of this phenomenom[1]. Since we focus on unary functions, we will need to work with pairs.

We can access the first and second elements of a pair using the functions `fst` and `snd`:

```
ex4> fst (1,2)
1
ex4> snd (1,2)
2
```

Since we will be looking at lists of pairs, it is plausible to think we would want the list of all the first elements, and a list of all the second elements.

Exercise 7 *Write two recursive functions,* `allFst` *and* `allSnd` *that take a list of pairs and return a list of the first elements or second elements of each pair, respectively.*

Rewrite the same functions as `allFstMap` *and* `allSndMap` *that do the equivalent thing, except use the* `map` *function. (Note: you could also define these functions using list comprehension.)*

We will now put together the function `isFn`. This function will take a list of pairs (representing our function to be checked), a list of values (representing the domain of the function), a second list of values (representing the codomain of the function), and returns a Boolean:

[1] Understanding this idea is beyond the scope of the module, so don't worry if you don't understand it.

```
isFn ::  [(a,b)] -> [a] -> [b] -> Bool
```

Remember that the definition of a function from A to B is that it has to send *each* element of A to exactly one element of B. We can break that down into three parts:

1. Given a list, `fs`, of pairs of type `(a,b)`, and a list of values, `xs`, of type `a`, is the *set* of first values of `fs` equal to the set of values `xs`? If yes, then we can say that everything in `xs` appears as the first elements of `fs`, and there is nothing in the first elements of `fs` which is not in `xs`.

2. Treating `fs` as a set, does anything appear twice in the list of first values of `fs`? If yes, then `fs` maps the same value of type `a` to two different values of type `b`. In other words, `fs` is not a function.

3. Is the set of second values of `fs` a subset of `ys` (the list of values of type `b`)? If yes, then the image of the function is a subset of the codomain of the function, as required.

Note the ordering of what is treated as a set in item 2. If we take the first values of the list `fs`, and then look at that set, then clearly everything will occur at most once, since sets have no multiplicity. If instead we take remove duplicates from `fs` and *then* take the first elements, we will identify any elements appearing twice. As an example, consider:

$$[(1,2),(1,3),(2,3),(3,4)] \text{ and } [(1,2),(1,2),(2,3),(3,4)]$$

The left-hand list is *not* a function, since 1 is mapped to two different elements. If we take the first elements of that list, we get $[1,1,2,3]$, and then if we treat that as a set we get $\{1,2,3\}$. By contrast, if we treat the list as a set first we get $\{(1,2),(1,3),(2,3),(3,4)\}$ and then take the first elements, we get the list $[1,1,2,3]$, showing that 1 is mapped to two different things.

Now, consider the right-hand list. This list looks, at first glance, to not represent a function, since 1 appears twice in the first elements. However, both instances of 1 get mapped to 2, so it is a function. Take the *set* of the list of pairs gives $\{(1,2),(2,3),(3,4)\}$. When we take the first elements of that as a list, then, we get $[1,2,3]$, which is duplicate free.

Exercise 8 *Write an (inductive) function called `allDifferent`, which takes a list of values of type `a` and returns a value of type `Bool`. If a value appears twice (or more) in the list, the function should return `False`, otherwise it should return `True`.*

Exercise 9 *Write the function `isFn`. It should be a conjunction with 3 parts, corresponding to the three items above. Hint: try to get each part working separately before trying to put it all together.*

Test your function on the pre-defined sets `f1` to `f10`, with associated domains and codomains `x1` and `y1`. In other words, test your function by typing:

```
ex4> isFn f1 x1 y1
```

The demonstrators know which ones are functions and which are not, so check with them to see if your function is doing what you hoped.

Injections

Now we know we can test for being a function, we are going to test whether a function is injective. Remember that an injective function $f : A \to B$ is one where:

$$f(x) = f(y) \to x = y.$$

So, how will we encode a test for such a function? It needs to identify all pairs in the function where the second elements are equal, and then check whether the corresponding first elements are the same.

Exercise 10 *Write an inductive function called* `mapTo`*, which takes an value of type* `b`*, a list of pairs of type* `(a,b)`*, and returns a list of values of type* `a`*. The interpretation is that* `mapTo y fs` *returns a list of all things that map to* `y` *in* `fs`*.*

We now have a list of values that they all get mapped, by our function, to the same given value. We want to perform this operation for every thing in the codomain of our function[2]. In other words, we could write:

```
map (mapTo fs) ys
```

This function will return a list of lists. We show it applied to `f0 = [(1,2),(2,3),(3,4),(4,3)]` and `ys = [1..4]`, which is already defined in the file `ex4.hs` :

```
ex4> map (mapTo f0) y0
[[],[1],[2,4],[3]]
```

How would we read this? It says that, in `f0`:

- nothing is mapped to `1`;
- the element `1` is mapped to `2`;
- the elements `2` and `4` are mapped to `3`; and
- the element `3` is mapped to `4`.

In other words, `f0` is not an injection because two things map to `3`. So, if any of these lists have more than one element, we do not have an injection.

Exercise 11 *Write a function that takes a list of lists and returns* `False` *if any have length greater than* 1*. There are many ways to write this function: try to write it in two different ways.*

We can now decide whether a given set is an injective function: it needs to be a function, and it needs to be an injection.

Exercise 12 *Write the function* `isInjection`*. It needs to satisfy the functions from Exercises 9* ***and*** *11. Test it on the functions* `f1` *to* `f10`*. The demonstrators know which ones are injections, so ask them to check your answers.*

[2]There is a more efficient way to write perform this operation: we only check those elements of our codomain to which something maps. Can you work out how to do that?

Surjections

Testing for surjections is relatively straightforward. Given a codomain B, does every element of B appear at least once in the second elements of f? Or, put another way, do we have set equality between B and the second elements of f?

Exercise 13 *Write the function* `isSurjection`*. Remember that we want to check for being a function as well! Test it on the functions* `f1` *to* `f10`*. The demonstrators know which ones are surjections, so ask them to check your answers.*

Bijections

A bijection is a function which is both a surjection and an injection.

Exercise 14 *Write the function* `isBijection`*. Test it on* `f1` *to* `f10`*. Given the answers to Exercises 12 and 13, you should be able to determine whether you are correct.*