SET07106 - Exercise sheet 2
Predicates and Lists

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 sections *Guards! Guards!*, *An Intro to Lists*, and *I'm a list comprehension.*

RESOURCES: The file `exercise2.hs`. Download it into an appropriate place on your file system.

We saw in the lecture that the statement $\forall x.P(x)$ could be thought of as a repeated conjunction over a set. For the statement to be true, every possible x must satisfy P, i.e. $P(x)$ must be true:

$$P(x_1) \wedge P(x_2) \wedge \ldots \wedge P(x_n)$$

where $x_1, \ldots, x_n$ are all the different possibilities for x.

By contrast, $\exists x.P(x)$ could be thought of as repeated disjunction over a set. For the statement to be true, at least one x must satisfy P:

$$P(x_1) \vee P(x_2) \vee \ldots \vee P(x_n).$$

In this practical, we will look at two different ways to represent these two quantifiers. There is a third way we will look at towards the end of the module.

Using `filter` and `map`

We saw `filter` last week, but this week we will properly investigate how it works. The function `filter` has two inputs: a propositional function (i.e. a function with a Boolean output) and a list of objects. We can see this by checking its type:

```
ex2> :t filter
filter ::  (a -> Bool) -> [a] -> [a]
```

Intuitively, `filter` takes the function and the list, and returns a list where all the elements satisfy the propositional function. If no element of the original list satisfies the function, then we get the empty list back:

```
ex2> filter (==1) [1,2,3,2,1]
[1,1]
ex2> filter (>4) [1,2,3,2,1]
[]
ex2> filter (=='a') ''Now is the winter of our discontent made glorious
Summer by this sun of York''
''a''
```

Exercise 1 *The propositional function* `even` *takes an integer. The notation* `[1..10]` *gives the list of whole numbers between* 1 *and* 10*. Write a line of code using* `filter` *that returns all the even numbers between* 1 *and* 10*. Write a line of code using* `filter` *that returns all the odd numbers between* 1 *and* 50*. (There is an inbuilt function called* `odd`*.)*

How will we use `filter` to work out whether *everything* in a list evaluates to `True` when the function is applied? We will apply the function to everything in the list, turning it into a list of Booleans, and then filter out the non-True values. To do this, we require the function `map`. This function has type:

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

In other words, it takes a function that transforms values of type `a` into values of type `b`, a list of things of type `a`, and returns a list of things of type `b`. Intuitively, it applies the function to every element of the first list, and returns that list of values:

```
ex2> map (+1) [1,2,3]
[2,3,4]
```

Here the types `a` and `b` are both `Int`. We could map our propositional function `f` (or, equivalently, our predicate P) across the list of things and get back a list of Booleans:

```
ex2> map (>2) [1,2,3]
[False,False,True]
```

Now, we need to check that list is entirely composed of `True`. Using `filter`, we can remove all the non-True values, and then see if the list has got shorter. We will use the function `length`, which returns the number of elements in a list. If the list has got shorter, then there must have been at least one `False` in the list, i.e. something which did not satisfy our predicate P.

```
ex2> filter (==True) (map (>2) [1,2,3])
[True]
ex2> length (filter (==True) (map (>2) [1,2,3]))
1
```

So, in the file `exercise2.hs`, there is a function called `forAllMapFilter`:

```
forAllMapFilter ::  (a -> Bool) -> [a] -> Bool
forAllMapFilter f xs = length (filter (==True) (map f xs)) == length xs
```

Similarly, if we want the filtered list to have at least one thing in it, we just need to check it is not empty. There is a function called `thereExistsMapFilter`:

```
thereExistsMapFilter ::  (a -> Bool) -> [a] -> Bool
thereExistsMapFilter f xs = (filter (==True) (map f xs)) /= []
```

Exercise 2 *There is a function in* `exercise2.hs` *called* `isPrime` *that checks whether a number is prime. Use this to determine whether:*

- *Every number between* 5 *and* 10 *is prime,*
- *There is some number between* 5 *and* 10 *which is prime,*
- *Every prime number between* 1 *and* 10 *is odd.*

Using list comprehension

Using `filter` produced something which was not particularly nice to look at. We can also use list comprehension which produces nicer looking code and may be easier for some people to read. List comprehension is when we write a list, and write a set of conditions that things in the list must satisfy. Haskell uses notation which is very similar to set notation, which we will see in next week's lecture. It is easiest to see with an example:

```
ex2> [x | x <- [1..10], even x]
[2,4,6,8,10]
```

We can unpack this a little. It says that we have a list of `x`s, where:

- `x <- [1..10]` (`x` is a value in the list `[1..10]`), AND
- `even x` (`x` is an even number).

The AND is key here: whenever we separate conditions in a list comprehension, we do it using a comma, and all of them have to be satisfied. Here, we are looking for the even numbers between 1 and 10, so we want the numbers returned to be both even and between 1 and 10.

Exercise 3 *Using list comprehensions, give the list of odd numbers between* 30 *and* 50. *Now, give the list of odd numbers between* 40 *and* 80 *which are also prime.*

How can we use list comprehension to work out the truth values of for all and exists statements? Consider the function `fakeFilter`:

```
fakeFilter ::  (a -> Bool) -> [a] -> [a]
fakeFilter f xs = [x | x <- xs, f x]
```

This does *exactly the same thing* as `filter`: it returns a list of elements which satisfy a certain function. (Note, we do not need to write `f x == True`, since the `== True` part happens automatically.)

Now we just need to check whether this list is the same as the one we started with, or non-empty:

```
forAllListComprehension ::  (a -> Bool) -> [a] -> Bool
forAllListComprehension f xs = [x | x <- xs, f x] == xs

thereExistsListComprehension ::  (a -> Bool) -> [a] -> Bool
thereExistsListComprehension f xs = [x | x <- xs, f x] /= []
```

Exercise 4 *Determine whether there are any e's in "So no-one told you life was gonna be this way". Determine whether every multiple of* 10 *less than* 200 *is also even. (You may like to play around with the notation* `[1,3..10]` *to answer the second part.)*

Questions on maps, filters and list comprehensions

Maps and filters are used frequently in functional programming. So are list comprehensions. You should attempt the following exercises to gain familiarity with them.

Exercise 5 *Write a line of code, using* `map` *and* `filter`*, that returns all the odd numbers between* 1 *and* 50*,* ***using the function*** `even`*. In other words, you are not allowed to use the function* `odd`*. Hint: what can you do to any even number to make it into an odd number?*

Exercise 6 *Do the above exercise using* `map` *and list comprehensions. Again, you are not allowed to use* `odd`*.*

Exercise 7 *Use the function* `head` *(amongst other things) to find the first prime number greater than* 100*. Try to do it in two different ways: using* `filter`*; and using list comprehensions.*

Exercise 8 *Use the function* `last` *(amongst other things) to find the greatest prime number less than* 200*. Try to do it in two different ways: using* `filter`*; and using list comprehensions.*

Exercise 9 *Write a line of code which doubles all of the integers between* 5 *and* 10 *and prints them as a list. Next, verify that this new list contains only even numbers.*

Exercise 10 *Use* `map` *to square the first* 20 *integers. Do the same using list comprehensions.*