SET07106 - Exercise sheet 1
Getting started and propositional functions

1 Starting the interpreter

Before we can use Haskell to answer the questions on this sheet, we have to start the Haskell interpreter running. Navigate on your computer to `c:/program files/Haskell/bin` and then double click `ghci.exe`. It will automatically open a command prompt where *GHCi* is running. After loading, you should see something that looks like:

```
Prelude>
```

We can now, if we wish, use Haskell as a calculator. For example, type `5*6` and press Enter, and you should see:

```
Prelude> 5*6
30
```

If you do not like the prompt `Prelude`, you can change it to whatever you want. Simply type the following, hitting enter after the first line:

```
Prelude> :set prompt "ghci> "
ghci>
```

Even though you can change it to whatever you want, it is best if you use some character, such as >, followed by a space, at the end of your prompt name, to make it easy to see where the prompt ends and your code begins. From this point on, all code will be presented with the prompt "`ex1>` ".

Next, we will load the file containing the functions needed for this week's exercises. To do this, type:

```
ex1> :load CHANGE THIS TO CORRECT LOCATION
```

and hit Enter. Note that the Haskell interpreter will look for the file `exercise1.hs` in the current folder, and then search its path directories to find it. You can also use "`:l exercise1`" to achieve the same result. We are now ready to start using the propositional functions that are contained in the file `exercise1.hs`.

2 Worked examples

Before letting you loose on some exercises, we will see what some of the propositional functions in that file do. Remember propositional functions return a Boolean value: a value of type `Bool`, either `True` or `False`.

First, we can see whether some simple things are true or false. Type:

```
ex1> (5>6) || (10>8)
True
```

The double-pipe (||) is what Haskell uses for "or". The first propositional statement (`5>6`) is clearly not true, so would evaluate to `False`, but the second statement (`10>8`) evaluates to `True`, so by the meaning of the connective "or" (from now on, we'll use $\vee$), the compound statement evaluates to `True`.

We know that the conjunction of the two previous statements is false, since at least one of them evaluates to `False`. So, it is no surprise to find:

```
ex1> (5>6) && (10>8)
False
```

We use the double amperdand (`&&`) for $\wedge$.

The function `countElems` has two inputs: a value of type `a`, and a list of values of type `a`, which we write as `[a]`. It gives us back an integer, which tells us how many times the first value appears in the list of values. For example:

```
ex1> countElems 1 [1,1,1,1,2,3,4,5]
4
ex1> countElems 5 [1,1,1,1,2,3,4,5]
0
ex1> countElems True [False,True,3>4,4==3,3<=4]
2
ex1> countElems 'a' "The cat sat on the mat"
3
```

The last of these is because Haskell thinks of strings as lists of characters. So, internally, we have that:

```
"The cat sat"
['T','h','e',' ','c','a','t',' ','s','a','t']
```

would be treated exactly the same. Note that Haskell treats a space as a character.

```
ex1> countElems ' ' "The cat sat on the mat"
5
```

Exercise 1 *Determine, using Haskell, whether the following statements are true.*

1. $8 > 9$
2. $9 \geq 0$
3. *`True`*$\wedge$*(`True`*$\vee$*`False`)*
4. *Not not `True` is equal to `True`.*
5. *The number of b's in "It was the best of times, it was the blurst of times" is* 6*.*
6. *The number of* 2*s in* 5438275439872242142 *is less than* 3*. (Hint: do not think of them as numbers, but characters!)*

7. *There are more t's in "The Importance of Being Earnest" than "The Picture of Dorian Gray".*

8. *The number of e's in "Gadsby" is less than the number of s's in "The quick brown fox jumped over the lazy dog".*

The function `exactlyTrueIn` takes a value of type `Int` (i.e. a whole number), a value of type `[Bool]` (i.e. a list of booleans) and returns a value of type `Bool`. We use the function *infix*, by surrounding it with ticks. It tells us whether or not the list contains exactly a specified number of `True` values:

```
ex1> 3 `exactlyTrueIn` [True,False,3>=1]
False
```

We are going to change this function to do different things. Open the file `exercise1.hs` using Notepad++. The function should look like:

```
exactlyTrueIn ::  Int -> [Bool] -> Bool
exactlyTrueIn n xs = length (filter (==True) xs) == n
```

By the end of this module, we will have a good understanding of everything here. However, for now we just focus on the parts we will change, which are the `(==True)` and the `== n`. The first tells us what to look for in the list of boolean values: everything that is equal to `True`. The `filter` function takes a list and returns a (possibly) smaller list that only contains things which are equal to `True`. The `length` function tells us how long a list is. Finally, the `== n` checks whether the number returned from the `length` function is equal to `n`.

Exercise 2 *Make copies of the function in the file `exercise1.hs`. Give each copy an appropriate name, and change the definition so that is tells us whether:*

1. *The number of values equal to `True` is at least n (i.e. $\geq n$).*

2. *The number of values equal to `True` is at most n (i.e. $\leq n$).*

3. *The number of values equal to `True` is greater than n (i.e. $> n$).*

4. *The number of values equal to `True` is less than n (i.e. $< n$).*

5. *The number of values equal to `True` is not n (i.e. $\neq n$).*

6. *The number of values equal to `False` is exactly n.*

Save the file `exercise1.hs`. Go to your prompt and type `:reload` or `:r`. Using lists of booleans and numbers, check your functions work as expected.

We are now going to attempt to change the function in a more radical way: instead of a hard-coded value we are checking against, we are going to pass the value to check against to the function. In order to do that, we need to look at the type signature of the function `exactlyTrueIn`:

```
exactlyTrueIn ::  Int -> [Bool] -> Bool
```

This takes a value of type `Int` and a list of values of type `Bool` and returns a value of type `Bool`. The right-most thing is the final type of the returned value. We want to give the function another argument, with the argument being either `True` or `False`, i.e. a value of type `Bool`. Haskell is clever: sometimes we can omit the type signature of a function, and Haskell will work it out for us. That is the easy method.

Exercise 3 *Make a copy of* `exactlyTrueIn`*, and then comment out the signature by typing* `--` *in front of it:*

```
-- exactlyTrueIn ::  Int -> [Bool] -> Bool
```

Change the definition of `exactlyTrueIn` *(and the name!) so that it takes a value of type* `Int`*, a value of type* `Bool`*, and a list of values of type* `Bool` *and returns a value of type* `Bool`*. The left-hand side of your definition should look like:*

```
exactlyGivenIn n b xs
```

Save and reload the file, and check that it works as expected.

Next, we can see if Haskell thinks the type of the function is the same as we are expecting! Type:

```
ex1> :type exactlyGivenIn
exactlyGivenIn ::  (Eq a) => Int -> a -> [a] -> Bool
```

That's odd: we were expecting `Int -> Bool -> [Bool] -> Bool`, but got this unspecified type `a` instead. In fact, we've defined something much more powerful than we were intending: we can check a list for occurrences of values of any type, so long as values of that type can be checked for equality with each other. That is what the `(Eq a)` thing means. For example, we can check whether two integers are equal, so our function should work fine using integers, not just booleans. We can also check whether characters are equal, so it should work fine with strings:

```
ex1> exactlyGivenIn 3 2 [1,2,3,4,3,2,1]
False
ex1> exactlyGivenIn 9 'a' "A man, a plan, a canal:  Panama"
True
```

Just think: if we had explicitly given the type for our function we would have to rewrite it for every different type! Thinking mathematically is all about focussing on the important stuff, and leaving the trivial stuff to someone else. In other words, a good mathematical thinker knows when to be lazy, and when to not be lazy.