Map, filter, and list comprehension

Now that we have a basic knowledge of lists and functions, we can start to look at some of the powerful constructs available in Haskell. The first is the `map` function, which comes originally (as far as I know) from the `mapcar` function of LISP.

The action of the map function is very simple: it takes a list and applies some function to every element of a list, obtaining a new list. Its type is

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

For example, suppose you want a list of the first ten squares:

```
Hugs> map (\x -> x*x ) [1..10]
[1,4,9,16,25,36,49,64,81,100]
```

For another example, we will do some text manipulation. It will be necessary to use some of Haskell's built-in functions. One is `toUpper`, a function of type `Char -> Char` that converts a character to upper case. If you want to use the `toUpper` function, you have to tell Hugs to import it (otherwise, it would have to import every possible library function every time it started up). Edit a file that contains the lines

```
import Prelude
import Char
```

The first line says to import `Prelude`, a library that contains many of the most frequently used functions (you might as well put `import Prelude` at the top of every Haskell script that you write). The function `toUpper` is in the `Char` library, so we import it as well. Load this file, and check that `toUpper` was correctly imported by entering

```
Hugs> :t toUpper
```

Now, to capitalize an entire string, we just enter

```
Hugs> map toUpper "Any string."
"ANY STRING."
```

The libraries such as `Char` contain many useful functions. A very convenient online reference for the libraries, as well as just about anything else you would want to know about the Haskell language, is the Haskell 98 Language and Libraries Revised Report. I have put a link to it on our links page.

By creative use of functions, you can do complex operations using `map`. For example, to capitalize all the vowels of a string, you could enter

```
Hugs> map (\x -> if elem x "aeiou" then toUpper x else x)
     "Some character string"
"SOmE ChArActEr StrIng"
```

Although this accomplishes the task in one step, it might be better to break this up into simpler pieces by creating an auxiliary function:

```
capitalizeVowels :: Char -> Char
capitalizeVowels x = if elem x "aeiou" then toUpper x else x
```

so that you can write

```
Hugs> map capitalizeVowels "Some Character String"
"SOmE ChArActEr StrIng"
```

Besides doing something to every member of a list, we often want to select some of the elements out of a list. For this, we have the `filter` function

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

which takes a boolean function, i. e. a condition, and an input list, and returns the list of elements of the input list that satisfy the condition. For example, if you want to select the capital letters out of a string:

```
Hugs> filter isUpper "Some Character String"
"SCS"
```

You could count the number of blank spaces in a string using

```
Hugs> length (filter (\x -> x == ' ') "Some Character String")
2
```

where we have used the `Prelude` function `length` (what is its type?). There is an even simpler way to write the previous operation:

```
Hugs> length (filter (== ' ') "Some character string")
2
```

What is going on in the notation `(== ' ')` is interesting. The logical equality operator is a function. Checking its type, we find

```
Hugs> :t ( == )
(==) :: Eq a => a -> a -> Bool
```

(The parentheses around == are needed to keep the interpreter from getting confused.) The result tells us that == is a function of type `a -> a -> Bool`. The `Eq a` part indicates that `a` must be a type for which equality is meaningful— more on that later. So, `== ' '` is the value of `== ' '` on the element `' '` of type `Char`, which is a function of type `Char -> Bool`. So it makes perfect sense to use `(== ' ')` as the boolean function in the statement

```
Hugs> length (filter (== ' ') "Some character string")
```

A final note on == is that it is normally written as an "infix" function, so you can enter `True == False` to Hugs (which it evaluates to `False`), but `== True False` confuses it. In the `filter` statement, either `== ' '` or `' ' ==` will work.

One of the handy devices in Haskell is list comprehension, which feels very natural to mathematicians. Let's start with an example:

```
Hugs> [ ch | ch <- "A character string", isUpper ch ]
"A"
```

The symbol `<-` is supposed to resemble the mathematical set membership symbol $\in$. This statement is intepreted as "The list consisting of all elements `ch`, drawn from the list `"A character string."`, that satisfy the boolean condition `isUpper`." Notice that it is really just a `filter` statement put into a friendlier format. But when you want to filter according to multiple conditions, perhaps from multiple lists, it is much easier to use. For example, suppose you have two character strings and you want to find every pair consisting of a small letter from the first string and the same letter appearing as a capital letter in the second string:

```
Hugs> [ (firstCh, secondCh) | firstCh <- "The first string.",
secondCh <- "The second string.", isLower firstCh, isUpper secondCh,
toUpper firstCh == secondCh ]
[('t','T'),('t','T')]
```

Notice that any expression in the list variables (in this case, the expression `(firstCh, secondCh)` in the list variables `firstCh` and `secondCh`) can be specified. For a final example and exercise, understand the following function definition:

```
allOdd :: [Int] -> Bool
allOdd ns = [ ] == [ n | n <- ns, n `mod` 2 == 0 ]
```

The backquotes around `` `mod` `` are a special notational device. The modulo function `mod` has type

```
mod :: Integral a => a -> a -> a
```

where `Integral` indicates that the type `a` must have a concept of integrality, and you can write

```
Hugs> mod 17 7
3
```

The backquotes make `mod` into an infix function, allowing you to write

```
Hugs> 17 `mod` 7
3
```

List comprehension is really nothing more than `map` and `filter` in a friendlier format:

```
[ f(x) | x <- list ]
```

is really

```
map f list
```

and

```
[ x | x <- list, P(x) ]
```

is really

```
filter P list
```

For example,

```
[ (firstCh, secondCh) | firstCh <- "The first string.",
secondCh <- "The second string.", isLower firstCh, isUpper secondCh,
toUpper firstCh == secondCh ]
```

can be written as the composition

```
filter (\pair -> toUpper (fst pair) == (snd pair) )
  ( concat
  ( map (\ch1 -> map (\ch2 -> (ch1, ch2)) (filter isUpper "The second string." ) )
  (filter isLower "The first string.") ) )
```

which is much easier to understand when written using list comprehension.