Haskell Unit 5: `map` and `filter`

Antoni Diller

26 July 2011

The functions `map` and `filter`

The higher-order function `map` can be defined like this:

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Intuitively, what `map` does is to apply the function `f` to each element of the list that it is applied to:

```
map f [ x1, x2, ..., xn ] = [ f x1, f x2, ..., f xn ]
```

The higher-order function `filter` can be defined like this:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter pred (x:xs)
  | pred x    = x:xs'
  | otherwise = xs' where xs' = filter pred xs
```

For example,

```
filter even [ 1 .. 20 ] = [ 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ]
```

Every list that can be defined using ZF-expressions can also be defined using `map` and `filter` and visa versa. First, I define `map` and `filter` using ZF-expressions:

```
map f xs = [ f x | x <- xs ]
filter pred xs = [ x | x <- xs, pred x ]
```

Next, I show the idea behind defining arbitrary ZF-expressions using `map` and `filter`: `[ f x | x <- xs, pred x ]` is equivalent to `map f (filter pred xs)`, which can also be written as `(map f . filter pred) xs`, using function composition.

Newton's method for finding positive square roots

Let `x` be the positive number whose square root you are trying to find. Then if `y > 0` is a guess, then `(y + x/y)/2` is a better guess. For example, say we want to find the positive square root of 27.3. Let us guess 1. Applying Newton's method, a better guess is 14.15. Applying Newton's method again, a still better guess is 8.03966. Applying Newton's method again, a still better guess is 5.71766. Applying Newton's method again, a still better guess is 5.24617. And so on. Newton's method can be programmed straightforwardly in Haskell as follows:

```
root :: Float -> Float
root x = rootiter x 1

rootiter :: Float -> Float -> Float
rootiter x y
  | satisfactory x y = y
  | otherwise        = rootiter x (improve x y)

satisfactory :: Float -> Float -> Bool
satisfactory x y = abs (y*y - x) < 0.01

improve :: Float -> Float -> Float
improve x y = (y + x/y)/2
```

This, however, is quite an "imperative" solution. A more "functional" solution uses the predefined Haskell function `iterate`:

```
iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

The function `iterate` generates an infinte list. For example, `iterate sqInt 2` would produce: `[ 2, 4, 16, 256, 65536, 4294967296, .. ]`. A more "functional" solution is, therefore:

```
root :: Float -> Float
root x = head (filter (satisfactory x) (iterate (improve x) 1))

satisfactory :: Float -> Float -> Bool
satisfactory x y = abs (y*y - x) < 0.01

improve :: Float -> Float -> Float
improve x y = (y + x/y)/2
```

The sieve of Eratosthesnes for generating primes

(1) Make a list of all the positive integers starting at 2.

(2) The first number on the list is prime; call it `p`.

(3) Construct a new list in which all multiples of `p` have been removed.

(4) Repeat the above from step (2).

For example,

```
[ [  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, ... ],
  [  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23, ... ],
  [  5,  7, 11, 13, 17, 19, 23, 25, 29, 31, 35, ... ],
  [  7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, ... ],
  [ 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, ... ], ... ]
```

In Haskell the sieve can be programmed like this:

```
primes       = map head (iterate sieve [2..])
sieve (p:xs) = [ x | x <- xs, x `mod` p /= 0 ]
```

Function composition

Composition is a binary operator represented by an infix full stop: `(f.g) x` is equivalent to `f (g x)`. The type of the section `(.)` is `(a -> b) -> (c -> a) -> c -> b`. Function composition is useful for many reasons. One of them is that `f (g ( h (i (j (k x)))))`, say, can be written as `(f . g . h . i . j . k ) x`; noting that function composition is associative. This usefulness can be illustrated by means of the following problem: Find the sum of the cubes of all the numbers divisible by 7 in a list `xs` of integers. The solution is as follows:

```
answer :: [Int] -> Int
answer xs = sum (map cube (filter by7 xs))

cube :: Int -> Int
cube x = x * x * x

by7 :: Int -> Bool
by7 x = x `mod` 7 == 0
```

But using function composition this can be written more clearly as follows:

```
answer :: [Int] -> Int
answer = sum . map cube . filter by7
```

Memoisation

In mathematics the Fibonacci numbers are usually defined like this:

```
fib 0 = 0
fib 1 = 1
fib i = fib (i - 1) + fib (i - 2)
```

Although this works in Haskell it is extremely inefficient. A more efficient definition prevents the re-evaluation of the same Fibonacci number. The values are stored in a list. The definition is as follows:

```
fib j   = fiblist !! j
fiblist = map f [ 0 .. ]
          where
          f 0 = 0
          f 1 = 1
          f i = fiblist !! (i - 1) + fiblist !! (i - 2)
```

Intuitively, `fiblist` contains the infinite list of Fibonacci numbers. Each element, say the `i`th can be expressed in at least two ways, namely as `fib i` and as `fiblist !! i`. This version of the Fibonacci numbers is very much more efficient.