Haskell Answers 5: *map* and *filter*

Antoni Diller

4 August 2011

(1) The type *String* is the same as [*Char*]. Define a function *capitalises*, of type *String* → *String*, which takes a list of characters as its argument and returns the same list as its value except that each lower-case letter has been replaced by its upper-case equivalent. Thus, `capitalises "Minority Report" = "MINORITY REPORT"`.

```
capitalises :: String -> String
capitalises = map toUpper
```

(2) Define a function *squareall* :: [*Int*] → [*Int*] which takes a list of integers and produces a list of the squares of those integers. For example, `squareall [6, 1, (-3)] = [36, 1, 9]`.

```
squareall :: [Int] -> [Int]
squareall = map (\x -> x*x)
```

(3) Define a function *nestedreverse* which takes a list of strings as its argument and reverses each element of the list and then reverses the resulting list. Thus, `nestedreverse ["in", "the", "end"] = ["dne", "eht", "ni"]`.

```
nestedreverse :: [String] -> [String]
nestedreverse = reverse . map reverse
```

(4) Define a function *atfront* :: *a* → [[*a*]] → [[*a*]] which takes an object and a list of lists and sticks the object at the front of every component list. For example, `atfront 7 [[1,2], [], [3]] = [[7,1,2], [7], [7,3]]`.

```
atfront :: a -> [[a]] -> [[a]]
atfront x xss = map (x:) xss
```

(5) Define a function *lengths* which takes a list of strings as its argument and returns the list of their lengths. For example, `lengths ["the", "end", "is", "nigh"] = [3, 3, 2, 4]`.

```
lengths :: [String] -> [Int]
lengths = map length
```

(6) Define a function *parity* :: [*String*] $\rightarrow$ [*Int*] which takes a list of strings and returns a list of the integers 0 and 1 such that 0 is the nth element of the value if the nth string of the argument contains an even number of characters and 1 is the nth element of the value if the nth string contains an odd number of characters. For example, `parity ["one", "two", "three", "four"] = [1, 1, 1, 0]`.

```
parity :: [String] -> [Int]
parity xss = map f xss
             where f xs
                     | even (length xs) = 0
                     | otherwise        = 1
```

(7) Using the higher-order function *map* define a function *sumsq* which takes an integer n as its argument and returns the sum of the squares of the first n integers. That is to say,

$$\textit{sumsq}\ n = 1^2 + 2^2 + 3^2 + \ldots + n^2.$$

```
square :: Num a => a -> a
square x = x * x

sumsq :: Integral a => a -> a
sumsq n = sum (map square [1..n])
```

(8) Define a function *subseqs* which takes a finite list *xs* as its argument and returns the list of all the subsequences of *xs*. (A subsequence of *xs* is a selection of not necessarily adjacent elements of *xs* which appear in their original order.)

```
subseqs :: [a] -> [[a]]
subseqs [] = [[]]
subseqs (x:xs) = (subseqs xs) ++ map (x:) (subseqs xs)
```

(9) The function *filter* can be defined in terms of *concat* and *map*:

```
filter p = concat.map box where box x = ...
```

Complete this definition of *filter* by defining *box*.

```
filter1 :: (a -> Bool) -> [a] -> [a]
filter1 p = concat.map box
            where box x
                    | p x       = [x]
                    | otherwise = []
```

(10) Define a function *wc* (without capitals) which removes all the capital letters from a string. Thus, `wc "Mark Twain" = "ark wain"`.

```
wc :: String -> String
wc = filter (not.isUpper)
```

(11) Define a function *wp* (without primes) which removes all the primes from a list of numbers. Thus, `wp [1, 2, 3, 4, 5, 6, 7] = [1, 4, 6]`.

```
wp :: [Int] -> [Int]
wp = filter (not.prime)

auxprime :: Integral a => a -> a -> Bool
auxprime i 2 = i ‘rem‘ 2 == 0
auxprime i j = i ‘rem‘ j == 0 || auxprime i (j-1)

prime :: Integral a => a -> Bool
prime 1 = False
prime 2 = True
prime i = not (auxprime i (i-1))
```

(12) Define a function *wtel* (without the empty list) which removes every occurrence of the empty list from a list of lists. Thus, `wtel [[1, 2], [], [1, 3]] = [[1, 2], [1, 3]]`.

```
wtel :: [[a]] -> [[a]]
wtel = filter (not.null)
```

(13) Define a function *caen* (containing an even number) which takes a list of lists of integers as its argument and removes from it every list *not* containing an even

number. Thus, `caen [[1,3], [2,1], [7,9], [2, 4, 8]] = [[2,1], [2, 4, 8]]`.

```
caen :: [[Int]] -> [[Int]]
caen = filter (even.product)
```

(14) Define a function *afoae* (at front of all even) which takes an integer and a list of lists of integers as its two arguments. It removes every element from the list which contains at least one odd number and attaches the integer at the front of the remaining lists. For example, `afoae 7 [[2, 4], [2, 3], [3, 7], [3, 4], [6, 100]] = [[7, 2, 4], [7, 6, 100]]`.

```
afoae :: Int -> [[Int]] -> [[Int]]
afoae i jss = map (i:) (filter (and.map even) (filter (not.null) jss))
```

(15) Define a function *wvowel* (without vowels) which removes every occurrence of a vowel from a list of characters.

```
wvowel :: String -> String
wvowel xs = filter f xs where f x = not (x == 'a' ||
                                         x == 'e' ||
                                         x == 'i' ||
                                         x == 'o' ||
                                         x == 'u')
```

(16) Define a function *wiv* (without internal vowels) which takes a list of strings as its argument and removes every occurrence of a vowel from each element. For example, `wiv ["the", "end", "is", "nigh"] = ["th", "nd", "s", "ngh"]`.

```
wiv :: [String] -> [String]
wiv = map wvowel
```

(17) Define a function *ssp* (sum the squares of primes) which takes a list of integers as its argument, removes those that are *not* primes and then squares the remaining integers and then adds the results together. For example, `ssp [2, 4, 7, 1, 3] = 62`.

```
ssp :: [Int] -> Int
ssp = sum . map square . filter prime
```