Haskell Answers 2: Lists

Antoni Diller

4 August 2011

(1) Define a function *productList* :: [*Int*] → *Int* which returns the product of a list of integers. You should take the product of the empty list to be 1.

```
productList :: [Int] -> Int
productList []     = 1
productList (x:xs) = x * productList xs
```

(2) Define a function *myand* :: [*Bool*] → *Bool* which returns the conjunction of a list. Informally,

$$myand\ [e_1, e_2, \ldots, e_i] = e_1\ \&\&\ e_2\ \&\&\ \ldots\ \&\&\ e_i.$$

The conjunction of an empty list should be *True*.

```
andList :: [Bool] -> Bool
andList []     = True
andList (p:ps) = p && andList ps
```

(3) Define a function *concatList* :: [[*Int*]] → [*Int*] which flattens a list of lists of integers into a single list of integers. For example,

$$concatList\ [[3, 4], [], [31, 3]] = [3, 4, 31, 3].$$

Informally,

$$concatList\ [e_1, e_2, \ldots, e_i] = e_1\ ++\ e_2\ ++\ \ldots\ ++\ e_i.$$

```
concatList :: [[Int]] -> [Int]
concatList []       = []
concatList (xs:xss) = xs ++ concatList xss
```

(4) Define the function *while* which is such that *while pred xs* returns the longest initial segment of the list *xs* all of whose elements satisfy the Boolean-valued function *pred*. For example,

$$\textit{while even } [2,4,8,3,4,8,6] = [2,4,8].$$

```
while :: (a -> Bool) -> [a] -> [a]
while pred [] = []
while pred (x:xs)
  | pred x    = x : while pred xs
  | otherwise = []
```

(5) The function *iSort* (insertion sort) is defined as follows:

```
iSort :: [Int] -> [Int]
iSort []     = []
iSort (x:xs) = ins x (iSort xs)

ins  :: Int -> [Int] -> [Int]
ins x [] = [x]
ins x (y:ys)
  | x <= y    = x:y:ys
  | otherwise = y:ins x ys
```

Use the function *iSort* to define two functions, *minList* and *maxList*, which find the minimum and maximum elements of a non-empty list of integers.

```
minList :: [Int] -> Int
minList xs = head (iSort xs)

maxList :: [Int] -> Int
maxList xs = last (iSort xs)
```

(6) Define the functions *minList* and *maxList*, which return the minimum and maximum elements of a non-empty list of integers, respectively, without using *iSort* or any other sorting function.

```
currentMin :: Int -> [Int] -> Int
currentMin x [] = x
currentMin x (y:ys)
  | x <= y    = currentMin x ys
  | otherwise = currentMin y ys

minList1 :: [Int] -> Int
minList1 (x:xs) = currentMin x xs

currentMax :: Int -> [Int] -> Int
currentMax x [] = x
currentMax x (y:ys)
  | x >= y    = currentMax x ys
  | otherwise = currentMax y ys

maxList1 :: [Int] -> Int
maxList1 (x:xs) = currentMax x xs
```

(7) Using the function *iSort* defined in question (5) redefine the function *ins* so that the list is sorted in descending order.

```
iSort1 :: [Int] -> [Int]
iSort1 []     = []
iSort1 (x:xs) = ins1 x (iSort1 xs)

ins1  :: Int -> [Int] -> [Int]
ins1 x [] = [x]
ins1 x (y:ys)
  | x >= y    = x:y:ys
  | otherwise = y:ins1 x ys
```

(8) Using the function *iSort* defined in question (5) redefine the function *ins* so that, in addition to outputting a list in ascending order, duplicates are removed. For example, *iSort* $[2, 1, 4, 1, 2] = [1, 2, 4]$.

```
iSort2 :: [Int] -> [Int]
iSort2 []     = []
iSort2 (x:xs) = ins2 x (iSort2 xs)

ins2  :: Int -> [Int] -> [Int]
ins2 x [] = [x]
ins2 x (y:ys)
  | x < y     = x:y:ys
  | x == y    = x:ys
  | otherwise = y:ins2 x ys
```

(9) Define the function *memberNum* :: $[Int] \rightarrow Int \rightarrow Int$ such that *memberNum xs x* returns the number of times that x occurs in the list xs. For example,

$$memberNum\ [2,1,4,1,2]\ 2 = 2.$$

```
currentMemberNum :: [Int] -> Int -> Int -> Int
currentMemberNum []     y z = z
currentMemberNum (x:xs) y z
  | x == y    = currentMemberNum xs y (z+1)
  | otherwise = currentMemberNum xs y z

memberNum :: [Int] -> Int -> Int
memberNum xs x = currentMemberNum xs x 0
```

(10) The function *member* :: $[Int] \rightarrow Int \rightarrow Bool$ has the property that *member xs x* returns *True* if x occurs in the list xs and it returns *False* if x does not occur in the list xs. Give a definition of *member* which uses the function *memberNum* that you defined as the answer to question (9).

```
member  :: [Int] -> Int -> Bool
member xs x = (memberNum xs x /= 0)
```

(11) Redefine the function *member* of question (10) so that it no longer makes use of *memberNum* (from question (9)).

```
member1 :: [Int] -> Int -> Bool
member1 [] y = False
member1 (x:xs) y
  | x == y    = True
  | otherwise = member1 xs y
```

(12) Using pattern matching with : (cons), define a function *rev2* that reverses all lists of length 2, but leaves all other lists unchanged.

```
rev2 :: [a] -> [a]
rev2 (x1:[x2]) = x2:[x1]
rev2 xs = xs
```

(13) Define a function *position* which takes a number i and a list of numbers xs and returns the position of i in the list xs, counting the first position as 1. If i does not occur in xs, then *position* returns 0.

(14) Define a function *element* which takes a list xs and a positive integer i and returns the ith member of xs. Assume that the list xs is at least of length i.

(15) Define a function *segments* which takes a finite list xs as its argument and returns the list of all the segments of xs. (A segment of xs is a selection of adjacent elements of xs.) For example, *segments* $[1, 2, 3] = [[1, 2, 3], [1, 2], [2, 3], [1], [2], [3]]$.

(16) A partition of a positive integer n is a representation of n as the sum of any number of positive integral parts. For example, there are 7 partitions of the number 5: $1+1+1+1+1$, $1+1+1+2$, $1+1+3$, $1+2+2$, $1+4$, $2+3$ and 5. Define a function *parts* which returns the list of distinct partitions of an integer n. For example, *parts* $4 = [[1, 1, 1, 1], [1, 1, 2], [1, 3], [2, 2], [4]]$.

(17) A segment ys of a list xs is said to be flat if all the elements of ys are equal. Define *llfs* such that *llfs* xs is the length of the longest flat segment of xs.

(18) A list of numbers is said to be steep if each element of the list is at least as large as the sum of the preceding elements. Define a function *llsg* such that *llsg* xs is the length of the longest steep segment of xs.

(19) Define a function *llsq* such that *llsq* xs is the length of the longest steep subsequence of xs.

(20) Given a sequence of positive and negative integers define a function *msg* which returns the minimum of the sums of all the possible segments of its argument.