Functional Programming: Assignment 3

Group 60

Lucas van der Laan
s1047485

1

1.

```
xs0 = [1,2,3,4,5]
xs1 = [[1,2,3],[4,5]]
xs2 = ['a', 'b', 'c']
xs3 = []
xs4 = [[],[]]
xs5 = [[[]]]
xs6 = [[]]
xs7 = [[[]]]
```

The variables that can be [Integer] are xs0 and xs3

2.

```
xs0 = 1 : 2 : 3 : [] ++ 4 : 5 : []
xs1 = (1 : 2 : 3 : []) : (4 : 5 : []) : []
xs2 = 'a' : 'b' : 'c' : []
xs3 = []
xs4 = [] : [] : []
xs5 = ([] : []) : []
xs6 = [] : []
xs7 = ([] : []) : []
```

3.

Because it can be any Num type, not necessarily an Integer.

2

I like reverse' better, because we are simply defining a new list, instead of concatenating to a new list over and over again.

3

```haskell
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs

or :: [Bool] -> Bool
or [] = True
or (x:xs) = x || and xs

elem :: (Eq a) => a -> [a] -> Bool
elem el [] = False
elem el (x:xs) = el == x || elem el xs

drop :: Int -> [a] -> [a]
drop n [] = []
drop 0 xs = xs
drop n (x:xs) = drop (n-1) xs

take :: Int -> [a] -> [a]
take n [] = []
take 0 xs = []
take n (x:xs) = x : take (n-1) xs
```

4

[1,2,3] ++/ [4,5]
= 1 : ([4,5] ++/ [2,3])
= 1 : 4 : ([2,3] ++/ [5])
= 1 : 4 : 2 : ([5] ++/ [3])
= 1 : 4 : 2 : 5 : ([3] ++/ [])
= 1 : 4 : 2 : 5 : 3 : ([] ++/ [])
= 1 : 4 : 2 : 5 : 3 : []
= [1,4,2,5,3]

5

```haskell
uniq :: (Eq a) => [a] -> [a]
uniq [] = []
uniq (x1:x2:xs) = if x1 == x2 then x1 : uniq xs else x1 : uniq (x2 : xs)
uniq (x:xs) = x : uniq xs
```

6

1.

g0 **Name:** getCombinations
Description: It computes all the combinations between the list of as and list of bs.

g1 **Name:** fillList
Description: Fills the list n times with a value y.

g2 **Name:** take
Description: It creates an index for every item in xs, which it uses to determine how many items to take from xs.

g3 **Name:** getIndex
Description: It creates and index for every item in xs until it has found the item, which it then returns the index of.

g4 **Name:** merge
Description: It first computes all the x and y combinations, which it then uses to create lists of [x,y] which then is used to create a new list of [x1,y1,x2,y2...].

g5 **Name:** flatten
Description: It first extracts xs from xss and then extracts x from xs, which it then puts in a new list.

2.

g0 **Type:** [a] -> [b] -> [(a, b)]
Poly/Overloaded: Polymorphic

g1 **Type:** (Num t, Enum t) => t -> a -> [a]
Poly/Overloaded: Overloaded

g2 **Type:** (Num a, Enum a, Ord a) => a -> [b] -> [b]
Poly/Overloaded: Overloaded

g3 **Type:** (Num a, Enum a, Eq b) => b -> [b] -> [a]
Poly/Overloaded: Overloaded

g4 **Type:** [a] -> [a] -> [a]
Poly/Overloaded: Polymorphic

g5 **Type:** [[a]] -> [a]
Poly/Overloaded: Polymorphic

7

```haskell
removeAt :: Int -> [a] -> [a]
removeAt n [] = []
removeAt n xs = [x | (i, x) <- zip [1..] xs, i /= n]

sortWithPos :: (Ord a) => [a] -> [(a,Int)]
sortWithPos [] = []
sortWithPos xs = [(x,i) | (x,i) <- sortBy sortHelper indexPairs]
  where
    indexPairs = [(x, i) | (i, x) <- zip [0..] xs]
    sortHelper (a1, b1) (a2, b2)
      | a1 < a2 = LT
      | a1 > a2 = GT
      | otherwise = compare a1 a2

sortedPos :: (Ord a) => [a] -> [(a,Int)]
sortedPos [] = []
sortedPos xs = [
    (x, j) | x <- xs, (_, i) <- sortedWithIndex, j <- find x sortedWithIndex, i == j
  ]
  where
    sortedWithIndex = [(x, i) | (i, x) <- zip [0..] (sort xs)]
    find y xs = [i | (i, x) <- zip [0..] (map fst xs), y == x]
```

8

```haskell
module Obfuscate where

import Data.Char (isSpace)

checkpunctuation :: Char -> Bool
checkpunctuation c = c `elem` ['.', ',', '?', '!', ':', ';', '(', ')']

words' :: String -> [String]
words' s = case dropWhile isSpace' s of
  "" -> []
  s' -> if checkpunctuation (head s'')
    then (w ++ [head s'']) : words' s''
    else w : words' (tail s'')
    where
      (w, s'') = break isSpace' s'
  where
    isSpace' s = isSpace s || checkpunctuation s

```

```haskell
getRandomNumbers :: Int -> Int
getRandomNumbers n = a * a * a `mod` m
  where
    a = n * 15485863
    m = 2038074743

shuffle :: [Int] -> [a] -> [a]
shuffle [] [] = []
shuffle [] (_:_) = []
shuffle (_:_) [] = []
shuffle (i:is) xs = let (firsts, rest) = splitAt (i `mod` length xs) xs
                    in head rest : shuffle is (firsts ++ tail rest)

shuffleMiddle :: [a] -> [a]
shuffleMiddle str = if length str > 1
  then head str : shuffle randomNumbers ((init . tail) str) ++ [last str]
  else str
  where
    randomNumbers = [getRandomNumbers j | j <- [1..length str - 2]]


cambridge :: String -> String
cambridge str = unwords [shuffleMiddle word | word <- allWords]
  where
    allWords = words' str

meme :: String
meme = "According to research at Cambridge university, it doesn't matter\
       \ what order the letters in a word are, the only important thing is\
       \ that the first and last letters are at the right place. The rest can\
       \ be a total mess and you can still read it without a problem. This is\
       \ because we do not read every letter by it self but the word as a wohle."
```