Tutorial 4: Comprehension & fold

A **list comprehension** is a convenient notation for list operations:

```
ghci> [ 3*n + 1 | n <- [0..10] , even n ]
[1,7,13,19,25,31]
```

The meaning of this example is as follows:

- Construct the list of all numbers `3*n + 1`, where
- `n` is an element of the list `[1..10]`, and
- `n` is even.

The syntax is designed after the way sets are described in mathematics. The following describes the set of all numbers $3n + 1$ where n is a natural number and n is even:

$$\{3n + 1 \mid n \in \mathbb{N}, n \text{ is even}\}$$

A list comprehension is constructed as follows:

```
[ <exp> | <qualifier_1> , ... , <qualifier_n> ]
```

where each qualifier can be a **generator** `x <- xs` that **draws** elements `x` from a list `xs` (such as `n <- [1..]` above), or a boolean **guard** (such as `prime n` above), which filters out cases that do not pass the given test. List comprehensions are intimately related to maps and filters:

```
map f (filter p xs)  ==  [ f x | x <- xs , p x ]
```

Exercise 1: Write the following functions from Tutorial 3 again, this time using a **list comprehension**:

a) `doubles`, which multiplies every number in a list by two,

b) `multiplesOfThree`, which removes any number from a list that is not a multiple of three,

c) `doubleMultiplesOfThree`, which doubles all multiples of three (removing the rest),

d) `shorts`, which removes all strings longer than 3 characters from a list,

e) `incrementPositives`, which adds one to all positive integers in a list,

f) `difference`, which removes the elements of the second list from the first,

g) `oddLengthSums`, which for a list of integer lists returns the sum of each odd-length list,

h) `everyother`, which takes every other element from a list starting with the first (use `zip` again),

i) `same`, which takes two lists and returns a list of the positions where their elements coincide; you may use `zip` and `zipWith` again, or instead try it with `zip3 :: [a] -> [b] -> [c] -> [(a,b,c)]`.

```
ghci> doubles [1..10]
[2,4,6,8,10,12,14,16,18,20]
ghci> multiplesOfThree [1..10]
[3,6,9]
ghci> doubleMultiplesOfThree [1..10]
[6,12,18]
ghci> shorts ["one","two","three","four","five","six","seven"]
["one","two","six"]
ghci> incrementPositives [-3,4,1,-2,0,3]
[5,2,4]
ghci> difference "difference" "ef"
"dirnc"
ghci> oddLengthSums [[1],[1,2],[1,2,3],[1..4],[1..5]]
[1,6,15]
ghci> everyother "Elizabeth"
"Eiaeh"
ghci> same "Charles" "Charlotte"
[1,2,3,4,5]
```

Exercise 2:

a) Complete the function `combinations` which takes two lists and produces every possible combination of items. Use a list comprehension with two generators.

b) Complete the function `selfcombinations` which takes a list and produces every possible pairing of its items, avoiding symmetric duplicates. That is, for each element, pair it only with those coming **after** it in the list, and with itself. **Hint:** in your list comprehension, use `zip` to count elements and `drop` to obtain those after that index.

c) Complete the function `pyts` so that `pyts n` generates all ordered Pythagorean triples with numbers up to (and including) `n`.

```
ghci> pairs "xy" [1,2,3]
[('x',1),('x',2),('x',3),('y',1),('y',2),('y',3)]
ghci> selfpairs [1..4]
[(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4),(4,4)]
ghci> pyts 100
[(3,4,5),(5,12,13),(6,8,10),(7,24,25),(8,15,17),(9,12,15),
(9,40,41),(10,24,26),(11,60,61),(12,16,20),(12,35,37),(13,84,85),
(14,48,50),(15,20,25),(15,36,39),(16,30,34),(16,63,65),(18,24,30),
(18,80,82),(20,21,29),(20,48,52),(21,28,35),(21,72,75),(24,32,40),
(24,45,51),(24,70,74),(25,60,65),(27,36,45),(28,45,53),(28,96,100),
(30,40,50),(30,72,78),(32,60,68),(33,44,55),(33,56,65),(35,84,91),
(36,48,60),(36,77,85),(39,52,65),(39,80,89),(40,42,58),(40,75,85),
(42,56,70),(45,60,75),(48,55,73),(48,64,80),(51,68,85),(54,72,90),
(57,76,95),(60,63,87),(60,80,100),(65,72,97)]
```

Folds

Finally, we will look at **folds** over lists. The function `foldr` ("r" for "right"—there is also a `foldl`) is a higher-order list function, like `filter` and `map`. It is defined as follows:

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

A function `foldr f u` takes a list, and replaces every **cons** (`:`) with the **folding function** `f`, and the **nil** with the **unit** `u`:

```
foldr f u (x1  :  (x2  :  (x3  :  ...  : [] )))  ==
          x1 `f` (x2 `f` (x3 `f` ... `f` u ))
```

Many list functions are naturally given as folds:

```
sum      = foldr (+) 0
product  = foldr (*) 1
concat   = foldr (++) []
xs ++ ys = foldr (:) ys xs
```

The version `foldr1 :: (a -> a -> a) -> [a] -> a` works on non-empty lists. Instead of a unit case, it uses the last element as the basis of the fold:

```
foldr1 f (x1  :  (x2  :  (x3  :  ...  : [xn] )))  ==
          x1 `f` (x2 `f` (x3 `f` ... `f` xn  ))
```

Exercise 3:

Haskell has several more built-in functions that are straightforward folds. We will write some of them here, renamed to avoid name clashes. Write the following functions with `foldr` or `foldr1` , defining the folding function in a where-clause if needed:

a) `allTrue` and `someTrue` , which behave as `and` and `or` ,

b) `largest` and `smallest` , which behave as `maximum` and `minimum` ,

c) `every` and `some` , which behave as `all` and `any` ,

d) `select` , which behaves as `filter` .

Extra

Exercise 4: The following functions count certain aspects of strings, such as how often a letter occurs. Write these with `foldr` .

a) The function `evenLength` returns `True` if a list is of even length, and `False` otherwise.

b) The function `count` returns the number of occurrences of a given character in a string.

c) The function `successive` counts the longest stretch of successive occurrences of a character in a string. **Hint:** the return value of the fold should be the **current** stretch and the **longest** (previous) stretch; then after folding a further function should select the longest only.

List comprehension together with recursion is a powerful tool in generating different combinations of the elements in a list. For example, the following function `choice` takes a list of lists, and returns every way of selecting one element from each list:

```
choice :: [[a]] -> [[a]]
choice []       = [[]]
choice (xs:xss) = [ y:ys | y <- xs, ys <- choice xss ]
```

```
ghci> choice [ [1,2] , [3,4] , [5,6] ]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
```

Exercise 5: Using `choice` as an example, write the following functions with recursion and list comprehension:

a) `selections` gives every way of selecting elements from a list (i.e. each element can be included or excluded),

b) `splits` gives every way of splitting a list into a first and second part,

c) `permutations` gives every permutation of a list (`splits` should be helpful here).

```
ghci> selections [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]

ghci> splits [1,2,3]
[([],[1,2,3]),([1],[2,3]),([1,2],[3]),([1,2,3],[])]

ghci> permutations [1..4]
[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1],[1,3,2,4],[3,1,2,4],
 [3,2,1,4],[3,2,4,1],[1,3,4,2],[3,1,4,2],[3,4,1,2],[3,4,2,1],
 [1,2,4,3],[2,1,4,3],[2,4,1,3],[2,4,3,1],[1,4,2,3],[4,1,2,3],
 [4,2,1,3],[4,2,3,1],[1,4,3,2],[4,1,3,2],[4,3,1,2],[4,3,2,1]]
```