List Comprehensions

In this section we discuss a very effective approach to working with lists, namely list comprehensions. List comprehensions in essence allow us to loop over the elements of the list and describe what should happen for each element.

Reading

- Sections 5.1-5.4
- Optional reading: 5.5
- Practice exercises (5.6): 1, 2, 3, 4, 5, 9

List Comprehensions

A **list comprehension** is a special syntax that is used to construct new lists out of existing lists. A list comprehension has the form:

```
[expr | ..., ..., ...]
```

Namely there are square brackets enclosing the whole expression, then a vertical line separating the expression in two parts. The left part is an expression describing the form that the resulting values would take. The right part consists of a sequence of terms, that fall in two categories:

1. **Generator** terms look like this: `x <- lst`. They suggest that the variable `x` should traverse the elements of the list `lst`, then everything to its *right* would happen for that `x`.
2. **Guard** terms consist of a predicate (a function that returns a `Bool`) that depends on the other variables used. A guard expression specifies a filter: Any values that don't satisfy the predicate (return `False`) will not be used.

The terms on the right of the vertical line are traversed from left to right. Later entries can use previous entries.

As a first example, let us start with a list of strings and return pairs of those strings along with their lengths:

```
someNames = ["Billie", "Jo", "Peter"]
[(name, length name) | name <- someNames] -- Produces [("Billie",6), ("Jo",2), ("Peter",5)]
```

As another example, using a guard this time, let us form pairs of names from the list, but only where the first name is shorter than the second:

```
[(name1, name2) | name1 <- someNames, name2 <- someNames, length name1 < length name2]
-- Result is [("Jo","Billie"), ("Jo","Peter"), ("Peter","Billie")]
```

We saw list comprehensions early on when we looked at the quicksort algorithm. Recall what that looked like:

```
-- Quicksort in Haskell
qsort []     = []
qsort (x:xs) = qsort smaller ++ [x] ++ qsort larger
               where smaller = [a | a <- xs, a <= x]
                     larger  = [b | b <- xs, b > x]
```

This example uses two list comprehensions to pick apart the elements of the list that are smaller than the pivot x and the elements of the list that are larger than the pivot.

As a further example, we can concatenate a list of lists into one list like this:

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

So this says: Put together all the x where x goes through the elements of the list xs as xs goes through the lists contained in xss.

As another standard example, we often want to traverse all pairs (i, j) of integers where the indices come from a range from 1 to say 10, but only when j is at least equal to i. We can do this in Haskell with list comprehensions in two different ways: One is using a guard to rule out the cases we don't need, the other makes the second list depend on the first.

```
[(i, j) | i <- [1..5], j <- [1..5], i <= j]
[(i, j) | i <- [1..5], j <- [i..5]]
```

Now let us consider a more complex problem: We are given a string, i.e. a list of characters, and we wonder if any character appears twice. One way to do this via list comprehensions is as follows:

- Form the pairs (c, i) where c is a character and i is its index into the string. we can do this with zip.
- Use a list comprehension to look at all pairs of such pairs from the list, where the characters are the same but the indices are not.

```
aString = "hey␣there␣my␣good␣fellow"
indexedString = zip aString [1..]
[(c, i, j) | (c, i) <- indexedString, (d, j) <- indexedString, c == d, i < j ]
```

Let us take a different and somewhat uglier approach:

- Start with a list comprehension that goes through each character in the string.
- For each such character, for a list comprehension that finds all characters in the string that are equal to that one, and records the length minus 1. In order to use this in a list comprehension we must turn it into a list with one element.
- Only keep those with "length minus 1" positive.

```
[(c, n) | c <- aString, n <- [length [d | d <- aString, d == c] - 1], n > 0]
```

Another common application of the zip function has to do with forming consecutive pairs from a list. If we do zip lst (tail lst) we get all consecutive pairs. We can then perform some operation on those pairs. For example, we can test if a list is sorted:

```
sorted :: Ord a => [a] -> Bool
sorted lst = all [x <= y | (x, y) <- zip lst (tail lst)]
```

In fact we would ideally make the formation of the pairs into its own function:

```
pairs :: [a] -> [(a, a)]
pairs []  = []
pairs lst = zip lst (tail lst)

sorted :: Ord a => [a] -> Bool
sorted lst = all [x <= y | (x, y) <- pairs lst]
```

As one last example, let us write a function that given two lists of numbers produces what is known as the "scalar" or "dot" product of the lists. It pairwise multiplies the numbers, one from each list, then adds those products. In order to do this with a list comprehension, we will use zip to put together the pairs of numbers, then a list comprehension to multiply the pairs, then sum to add all those products:

```
dotProduct :: [Int] -> [Int] -> Int
dotProduct xs ys = sum [x*y | (x, y) <- zip xs ys]
```

Question: Would it be correct to use two generators here (i.e. x<–xs, y<–ys)? "'

Application: Finding prime numbers

We can write a primitive set of functions for finding prime numbers using list comprehensions. First of all recall what a prime number is:

> A prime number n is a number that is only divisible by 1 and itself. So a prime number has exactly two factors, 1 and n.

We will approach the problem from two different points of view:

Take 1: Compute the factors We start by writing a function that collects all the factors of a number n: these are those that divide perfectly into n, and we can use a list comprehension for that:

```
factors :: Int -> [Int]
factors n = [x | x <- [1..n], n `mod` x == 0]
```

Next we will write a function isPrime. This tests if a number if prime, by computing its factors and comparing them to the list [1, n]:

```
prime :: Int -> Bool
prime n = factors n == [1, n]
```

Lastly, we can ask the system to give us all the primes via a list comprehension. As the result is going to be infinite, we should use take to see the first few values:

```
primes :: [Int]
primes = [n | n <- [2..], prime n]
take 50 primes  -- The first 50 prime numbers
```

Take 2: Sieve of Eratosthenes Let us try a different approach, as the approach above requires computing all the factors for each number, and that can take time. This alternative approach is called the sieve of Eratosthenes, and it's based on a simple idea:

1. Start with all the numbers from 2 and on.
2. The first number on the list is prime, as there are no number before it that divide into it. Mark it as such, then proceed to delete all its multiples (e.g. pick 2 as prime then delete 4, 6, 8, etc).
3. Repeat step 2 with the remaining numbers.

The beautiful thing is that this is easy to write in Haskell. We start with a sieve function: It takes a non-empty list of all the remaining integers. Then the first integer in that list is prime, and it removes all the remaining (infinitely many) integers before calling itself on the remaining list. We then just kickstart the whole thing by providing the infinite list [2..].

This looks as follows:

```
sieve :: [Int] -> [Int]
sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]

primes :: [Int]
primes = sieve [2..]
```

You can read more about this approach in section 15.6 from the book. This is using the powerful **lazy evaluation** approach that Haskell takes. The list comprehensions in the sieve computation are actually infinite lists. But Haskell will not compute them until it absolutely has to.

We can also carry out timing tests and see that this method is a lot faster and less resource-intensive than the previous one.

Application: The Fibonacci numbers

Another favorite application of list comprehensions is the computation of the Fibonacci sequence. Recall that this sequence works as follows: We start with 1 and 1, then we generate each next number by adding the two previous numbers. So:

```
1 + 1 = 2
1 + 2 = 3
2 + 3 = 5
3 + 5 = 8
...
```

We can implement this sequence very efficiently in Haskell using a list comprehension. The trick is to start with two explicit terms, and then to form a zip of the list and its tail, which results in pairs of consecutive terms. We then simply add the numbers in the pair to get the next element. This looks as follows:

```
fib = 1 : 1 : [a + b | (a, b) <- zip fib (tail fib)]
take 40 fib
```

Practice

1. Write a function `count :: Eq a => a -> [a] -> Int` that is given an element of type `a` and a list of such elements, and it returns how many times the elment occurs. You should be able to make a list comprehension of all the occurences of that character, then compute the length of that list.

2. Pythagorean triples are triples of integers `(a, b, c)` such that $a^2 + b^2 = c^2$. Write a function `triples` that given an integer limit `n` generates all triples where the numbers are up to `n`.