## FP101x - Functional Programming

Programming in Haskell – List Comprehensions

Erik Meijer



### **Set Comprehensions**

In mathematics, the <u>comprehension</u> notation can be used to construct new sets from old sets.

$$\{x^2 \mid x \in \{1...5\}\}$$

The set  $\{1,4,9,16,25\}$  of all numbers  $x^2$  such that x is an element of the set  $\{1...5\}$ .

### **Lists Comprehensions**

In Haskell, a similar comprehension notation can be used to construct new <u>lists</u> from old lists.

$$[x \land 2 \mid x \leftarrow [1..5]]$$

The list [1,4,9,16,25] of all numbers x^2 such that x is an element of the list [1..5].

#### Note:

■ The expression  $x \leftarrow [1..5]$  is called a generator, as it states how to generate values for x.

Comprehensions can have <u>multiple</u> generators, separated by commas. For example:

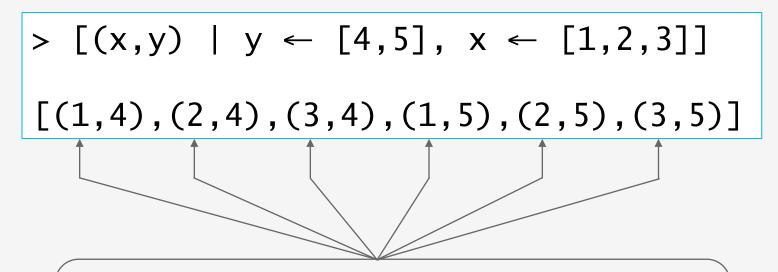
> 
$$[(x,y) \mid x \leftarrow [1,2,3], y \leftarrow [4,5]]$$
  
 $[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]$ 

Changing the <u>order</u> of the generators changes the order of the elements in the final list:

> 
$$[(x,y) | y \leftarrow [4,5], x \leftarrow [1,2,3]]$$
  
 $[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]$ 

Multiple generators are like <u>nested loops</u>, with later generators as more deeply nested loops whose variables change value more frequently.

#### For example:



 $x \leftarrow [1,2,3]$  is the last generator, so the value of the x component of each pair changes most frequently.

### **Dependant Generators**

Later generators can <u>depend</u> on the variables that are introduced by earlier generators.

$$[(x,y) \mid x \leftarrow [1..3], y \leftarrow [x..3]]$$

The list [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] of all pairs of numbers (x,y) such that x,y are elements of the list [1..3] and  $y \ge x$ .

Using a dependant generator we can define the library function that <u>concatenates</u> a list of lists:

```
concat :: [[a]] \rightarrow [a]

concat xss = [x \mid xs \leftarrow xss, x \leftarrow xs]
```

```
> concat [[1,2,3],[4,5],[6]]
[1,2,3,4,5,6]
```

#### **Guards**

List comprehensions can use <u>guards</u> to restrict the values produced by earlier generators.

[x | 
$$x \leftarrow [1..10]$$
, even x]

The list [2,4,6,8,10] of all numbers x such that x is an element of the list [1..10] and x is even.

Using a guard we can define a function that maps a positive integer to its list of <u>factors</u>:

```
factors :: Int \rightarrow [Int]
factors n =
[x | x \leftarrow [1..n], n `mod` x == 0]
```

```
> factors 15
[1,3,5,15]
```

A positive integer is <u>prime</u> if its only factors are 1 and itself. Hence, using factors we can define a function that decides if a number is prime:

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

- > prime 15 False
- > prime 7
  True

Using a guard we can now define a function that returns the list of all <u>primes</u> up to a given limit:

```
primes :: Int \rightarrow [Int]
primes n = [x | x \leftarrow [2..n], prime x]
```

```
> primes 40
[2,3,5,7,11,13,17,19,23,29,31,37]
```

## The Zip Function

A useful library function is <u>zip</u>, which maps two lists to a list of pairs of their corresponding elements.

$$zip :: [a] \rightarrow [b] \rightarrow [(a,b)]$$

```
> zip ['a','b','c'] [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Using zip we can define a function returns the list of all pairs of adjacent elements from a list:

```
pairs :: [a] \rightarrow [(a,a)]
pairs xs = zip xs (tail xs)
```

```
> pairs [1,2,3,4]
[(1,2),(2,3),(3,4)]
```

Using pairs we can define a function that decides if the elements in a list are <u>sorted</u>:

```
sorted :: Ord a \Rightarrow [a] \rightarrow Bool

sorted xs =

and [x \le y \mid (x,y) \leftarrow pairs xs]
```

```
> sorted [1,2,3,4]
True
> sorted [1,3,2,4]
False
```

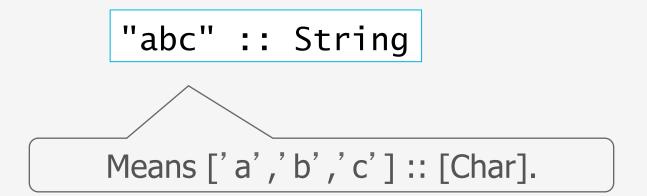
Using zip we can define a function that returns the list of all positions of a value in a list:

```
positions :: Eq a \Rightarrow a \rightarrow [a] \rightarrow [Int]
positions x xs =
 [i \mid (x',i) \leftarrow zip xs [0..n], x == x']
where n = length xs - 1
```

```
> positions 0 [1,0,0,1,0,1,1,0] [1,2,4,7]
```

## **String Comprehensions**

A <u>string</u> is a sequence of characters enclosed in double quotes. Internally, however, strings are represented as lists of characters.



Because strings are just special kinds of lists, any polymorphic function that operates on lists can also be applied to strings. For example:

```
> length "abcde"
> take 3 "abcde"
"abc"
> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

Similarly, list comprehensions can also be used to define functions on strings, such as a function that counts the lower-case letters in a string:

```
lowers :: String → Int
lowers xs =
  length [x | x ← xs, isLower x]
```

```
> lowers "Haskell"
6
```

#### **Exercises**

(1) A triple (x,y,z) of positive integers is called <u>pythagorean</u> if  $x^2 + y^2 = z^2$ . Using a list comprehension, define a function

```
pyths :: Int → [(Int,Int,Int)]
```

that maps an integer n to all such triples with components in [1..n]. For example:

```
> pyths 5 [(3,4,5),(4,3,5)]
```

(2) A positive integer is <u>perfect</u> if it equals the sum of all of its factors, excluding the number itself. Using a list comprehension, define a function

```
perfects :: Int → [Int]
```

that returns the list of all perfect numbers up to a given limit. For example:

> perfects 500

[6,28,496]

(3) The <u>scalar product</u> of two lists of integers xs and ys of length n is give by the sum of the products of the corresponding integers:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

Using a list comprehension, define a function that returns the scalar product of two lists.

# **Happy Hacking!**

