

Functional Programming

Lecture 4 — Lists

Pedro Vasconcelos
DCC/FCUP

2025

Lists

Lists are collections of values:

- ▶ where the order is significant
- ▶ possibly with repeated values

Lists in Haskell

A list in Haskell is either:

the empty list `[]`

a non-empty list `x:xs` whose first element is `x`, followed by
the list `xs`

Extension notation

You may write list values separated by commas (,) and
between brackets ([and]).

$$\begin{aligned}[1, 2, 3, 4] &= 1 : (2 : (3 : (4 : []))) \\ &= 1 : 2 : 3 : 4 : []\end{aligned}$$

Enumerations

Expressions of the form $[a..b]$ or $[a, b..c]$ produce lists (a , b and c are values of an *enumerable* type, e.g. numbers).

```
ghci> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

```
ghci> [1,3..10]
[1,3,5,7,9]
```

```
ghci> [10,9..1]
[10,9,8,7,6,5,4,3,2,1]
```

Enumerations (cont.)

We can also build *infinite lists* if we omit the upper limit:

```
ghci> take 10 [1,3..]  
[1,3,5,7,9,11,13,15,17,19]
```

Evaluation of an infinite list in GHCI will not terminate (stop it using *Ctrl-C*):

```
ghci> [1,3..]  
[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,  
Interrupted
```

Comprehensions

In mathematics we can define a set from another with by using a *set comprehension*.

Example:

$$\{n^2 : n \in \mathbb{N}\}$$

define the set of squares of natural numbers:

$$\{1, 4, 9, 16, 25, \dots\}$$

Comprehensions (cont.)

We can define a list from another one using a *list comprehension*.

Example:

```
ghci> [n^2 | n ← [1..5]]  
[1, 4, 9, 16, 25]
```

(NB: we type `←` as `<-` on a keyboard.)

Generators

A term of the form $pattern \leftarrow list$ is called a **generator**:

- ▶ gives values to the variables in the pattern
- ▶ determines the order in which values are accumulated

We can also specify multiple generators:

```
ghci> [(x,y) | x<-[1,2,3], y<-[4,5]]  
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

This generates all pairs (x, y) such that x takes values from $[1, 2, 3]$ and y takes values from $[4, 5]$.

Nested generators

x first, y second

```
ghci> [(x,y) | x<-[1,2,3], y<-[4,5]]  
[(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
```

y first, x second

```
ghci> [(x,y) | y<-[4,5], x<-[1,2,3]]  
[(1,4), (2,4), (3,4), (1,5), (2,5), (3,5)]
```

Analogy: nested for loops

```
for(x=1; x<=3; x++)  
  for(y=4; y<=5; y++)  
    print(x,y);
```

```
for(y=4; y<=5; y++)  
  for(x=1; x<=3; x++)  
    print(x,y);
```

Dependencies among generators

Expressions used in generators can depend on values of variables bound in *previous* generators but not in *later* ones.

```
ghci> [(x,y) | x<-[1..3], y<-[x..3]]  
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

```
ghci> [(x,y) | y<-[x..3], x<-[1..3]]  
-- error: Variable not in scope: 'x'
```

Dependencies among generators (cont.)

We can use a dependent generator to define the Prelude concatenation function. Recall its behaviour:

```
ghci> concat [[1,2,3], [4,5], [6,7]]  
[1,2,3,4,5,6,7]
```

Here is a definition of `concat` using a list comprehension:

```
concat :: [[a]] → [a]  
concat xss = [x | xs ← xss, x ← xs]
```

Naming convention:

x is a value

xs is a list of values

xss is a list of lists of values

Guards

List comprehensions may include conditions (called *guards*) to filter the values being generated.

Example: the list of integers x such that x is between 1 and 10 and is even.

```
ghci> [x | x<-[1..10], x `mod` 2==0]  
[2,4,6,8,10]
```

A larger example: primality testing

Let us start by defining an auxiliary function to list all divisors of a positive integer.

```
divisors :: Int → [Int]
divisors n = [x | x ← [1..n], n `mod` x == 0]
```

Examples:

```
ghci> divisors 15
[1,3,5,15]
ghci> divisors 19
[1,19]
```

A larger example: primality testing (cont.)

Using the auxiliary function, we can define a primality testing function: n is prime if and only if its divisors are exactly 1 e n .

```
isPrime :: Int → Bool  
isPrime n = divisors n == [1, n]
```

```
ghci> isPrime 15  
False  
ghci> isPrime 19  
True
```

(NB: this is not an efficient algorithm; the exercise sheets propose an better solution.)

A larger example: primality testing (cont.)

We can also use `isPrime` inside another list comprehension to find all primes up-to some limit.

```
primesUpto :: Int → [Int]
primesUpto n = [x | x ← [2..n], isPrime x]
```

Example:

```
ghci> primesUpto 50
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

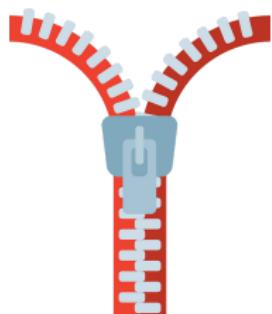
The `zip` function

The `zip` from the Prelude combines two lists into the list of corresponding pairs. If the lists have different lengths, the resulting list will have the length of the *smaller* list.

```
zip :: [a] → [b] → [(a, b)]
```

Example:

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



The function is named “zip” because of the analogy with a clothes zipper.

Using zip

We can `zip` to combine list values with its indices.

Example: look for the indices of occurrences of a value in a list.

```
indices :: Eq a ⇒ a → [a] → [Int]
indices x ys = [i | (y,i) ← zip ys [0..n], x==y]
    where n = length ys - 1
```

```
ghci> indices 'a' ['b','a','n','a','n','a']
[1,3,5]
```

Using zip (cont.)

We can also use `zip` and `tail` to list *pairs of consecutive values* in a list.

```
consecutive :: [a] → [(a,a)]
consecutive xs = zip xs (tail xs)
```

```
xs          =  [x1, x2, ..., xn-1, xn]
tail xs    =  [x2, x3, ..., xn]
zip xs (tail xs)
            =  [(x1,x2), (x2,x3), ..., (xn-1, xn)]
```

Using zip (cont.)

Examples:

```
ghci> consecutive [1,2,3,4]
[(1,2), (2,3), (3,4)]
```

```
ghci> consecutive ['a','b','b','a']
[('a','b'), ('b','b'), ('b','a')]
```

```
ghci> consecutive [1,2]
[(1,2)]
```

```
ghci> consecutive [1]
[]
```

Strings

The `String` type is predefined in the Prelude as a synonym for a list of characters.

```
type String = [Char] -- defined in the Prelude
```

For example:

```
"abba"
```

is the same as

```
['a','b','b','a']
```

(NB: GHC supports more efficient data types for strings, but we won't be using them in these lectures.)

Strings (cont.)

Since strings are special cases of lists, we can use list functions to process strings.

```
ghci> length "abcde"  
5
```

```
ghci> take 3 "abcde"  
"abc"
```

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

String comprehensions

We can also use comprehensions with strings.

Example: count characters between ‘A’ and ‘Z’.

```
countLetters :: String → Int
countLetters txt
  = length [c | c ← txt, c >= 'A' && c <= 'Z']
```

Processing characters

Specialized functions are not part of the Prelude; instead they are defined in separate *modules*.

To use a module we must use an import declaration at the beginning of our file.

```
import Data.Char -- use the Data.Char module
```

More information

We can use the `:browse` GHCi command to list exported function in a module.

```
Prelude> import Data.Char
Prelude Data.Char> :browse
digitToInt :: Char -> Int
isLetter :: Char -> Bool
isMark :: Char -> Bool
::
```