Functional Programming

Paul Mellies

Lesson 6 : Lists
Anatomy of a list

The notion of list is defined in Haskell in the following recursive way:

```haskell
a list is either empty or is an element followed by a list
```

The **head of a non empty list** is its first element:

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

The **tail of a non empty list** is what remains after taking apart the head:

```haskell
ghci> tail [1,2,3]
[2,3]
ghci> tail [3]
[]
```

where [] denotes **the empty list**.
Anatomy of a list

The recursive nature of lists in Haskell:

A list is either empty or is an element followed by a list

is witnessed by the fact that:

- the head of a non empty list is an element,
- the tail of a non empty list is again a list.

Also, calling head or tail on the empty list [] will result in an error:

```
ghci> head []  
*** Exception: Prelude.head: empty list
ghci> tail []  
*** Exception: Prelude.tail: empty list
```
How to construct lists

In order to construct lists, one uses the `cons` function noted:

```
ghci> 1:[]
[1]
```

where the term `cons` (short for `construct`) and has its origins in Lisp.

Under the hood, all lists in Haskell are represented as a bunch of consing operations and the notation `[...]` is **syntactic sugar** designed to make things easier to read:

```
ghci> 1:2:3:4:[]
[1,2,3,4]
ghci> (1,2):(3,4):(5,6):[]
[(1,2),(3,4),(5,6)]
```
Strings as lists of characters

Every string is encoded in Haskell as a list of characters.

```
ghci> ['h','e','l','l','o']
"hello"
ghci> 'h':'e':'l':'l':'o':[]
"hello"
```

In other words, a string in Haskell is syntactic sugar for a list of characters.
Strings as lists of characters

An important principle in Haskell is that

> every element of a list must be of the same type.

For that reason, the cons expression below works as expected

```ghci
ghci> 'h':"ello"
"hello"
ghci> 'h':['e','l','l','o']
"hello"
```

while any of the three equivalent expressions (up to syntactic sugar)

```ghci
ghci> "h":"ello"
ghci> ['h']:['e','l','l','o']
ghci> 'h':['']:['e':'l':'l':'o':[]]
```

will result in an error.
Concatenation of lists

If you want to concatenate two lists, use the ++ function:

```
ghci> [1] ++ [2,3,4]
[1,2,3,4]
```

The concatenation function ++ applies in particular to strings:

```
ghci> "h" ++ "ello"
"hello"
```

since strings are just lists of characters in disguise.
Lists and lazy evaluation

Lists are very important in Haskell, and there are several ways to build them.

Here follows a convenient way to generate the list of integers from 1 to 10:

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

the list of odd numbers:

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

the list of numbers from 1.0 to 5.0 incremented by 0.5 each time:

```
ghci> [1,1.5 .. 5]
[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]
```

or the decreasing list of numbers from 1 to −10:

```
ghci> [1,0 .. -10]
[1,0,-1,-2,-3,-4,-5,-6,-7,-8,-9,-10]
```
Lists and lazy evaluation

More surprisingly, it is possible to define an infinite list

   ghci> [1 .. ]
   [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18 ..

which can be then passed as parameter to a function.
Consider for instance the following Haskell code:

   simple x = x
   longList = [1 .. ]
   stillLongList = simple longList

The code compiles just fine, and produces the following behavior:

   ghci> stillLongList
   [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18 ..

The reason is that Haskell uses a special form of evaluation called lazy evaluation where some piece of the code is evaluated only when it is needed.
The `!!` operator

At this stage, we have seen the four basic functions on list

```
head    tail    :    ++
```

The **list index operator**

```
!!
```

is an infix function which picks the n-th element in a list:

```
ghci> [1,2,3] !! 0
1
ghci> "puppies" !! 4
'i'
ghci> [1..10] !! 11
*** Exception: Prelude.!!: index too large
```
Left and right section of an infix operator

We have seen how to turn an infix function like `!!` into a prefix function `(!!)` with the following behavior:

```
ghci> (!!) [1,2,3] 0
1
```

Note that `(!!)` could be equivalently defined as the lambda function:

```
\x y -> x !! y
```
Left and right section of an infix operator

It is often convenient to use the prefix functions

\[
("dog" \text{ !}) \quad \quad (!! 2)
\]

called the \textbf{left} and \textbf{right section} of \text{!!} which are handy notations for

\[
(!!) "dog" \quad \quad \text{flip} (!!) 2
\]

or equivalently using lambda functions:

\[
\lambda x \rightarrow "dog" \text{ !} x \quad \quad \lambda x \rightarrow x \text{ !} 2
\]

These two functions have the expected behavior:

```
ghci> let leftSection = ("dog" !)
ghci> leftSection 2
"g"

ghci> let rightSection = (!) 2
ghci> rightSection "dog"
"g"
```
The **length** function

The **length** function computes the length of a list:

```
ghci> length [1..20]
20
ghci> length [(10,10),(1,2),(15,16)]
3
ghci> length "quicksand"
9
```

**Exercise.** What happens when one applies the length function to an infinite list?

```
ghci> length [1..]
```
The **reverse** function

The purpose of the `reverse` function is to reverse a list:

```
ghci> reverse [1,2,3]
[3,2,1]
ghci> reverse "cheese"
"eseehc"
```

**Illustration:** the `isPalindrome` function is designed to detect palindromes:

```
isPalindrome word = word == reverse word
```

The function behaves in the following way:

```
ghci> isPalindrome "cheese"
False
ghci> isPalindrome "racecar"
True
ghci> isPalindrome [1,2,3]
False
ghci> isPalindrome [1,2,1]
True
```
The **elem** function

The **elem** function takes a value **val** and a list **list** as argument and checks whether the value is an element of the list:

```
ghci> elem 13 [1,13 .. 100]
True
ghci> elem 'p' "cheese"
False
```

Any binary function can be turned into an **infix function** using the back-quotes.

**Illustration:** this may be illustrated with the **respond** function

```
respond phrase = if '!' 'elem' phrase
    then "wow!"
    else "uh.. okay"
```

which detects the appearance of an exclamation mark in a phrase:

```
ghci> respond "hello"
"uh.. okay"
ghci> respond "hello!"
"wow!"
```
The `take` and `drop` functions

The `take` function takes a number `n` and a list as argument and then returns the list consisting of the first `n` elements of the original list:

```
ghci> take 5 [2,4 .. 100]
[2,4,6,8,10]
ghci> take 3 "wonderful"
"won"
```

When `n` is larger than the length of the list, the `take` function returns the list itself:

```
ghci> take 5 [1,2,3]
[1,2,3]
```

The `drop` function works similarly, and removes the first `n` elements of the list:

```
ghci> drop 2 [1,2,3,4,5]
[3,4,5]
ghci> drop 5 "very awesome"
"awesome"
```
The zip function

The zip function takes two lists as argument and combines them into a list of tuple pairs, in the following way:

ghci> zip [1,2,3] [2,4,6]
[(1,2),(2,4),(3,6)]

ghci> zip "dog" "rabbit"
[('d','r'),('o','a'),('g','b')]

ghci> zip ['a' .. 'f'] [1 .. ]
[('a',1),('b',2),('c',3),('d',4),('e',5),('f',6)]
The cycle function

The cycle function takes advantage of lazy evaluation to create infinite lists: given a list, the cycle function repeats that list endlessly.

Illustration: define the function

\[
\text{ones } n = \text{take } n (\text{cycle } [1])
\]

The ones function takes a number \( n \) as argument and returns the list consisting of the number 1 repeated \( n \) times:

\[
\text{ghci} > \text{ones } 2 \\
[1,1] \\
\text{ghci} > \text{ones } 4 \\
[1,1,1,1]
\]
The *cycle* function

The *cycle* function is more useful than it seems.

Consider for instance the function

```haskell
assignToGroups n aList = zip groups aList
    where groups = cycle [1..n]
```

takes a number *n* and a list *aList* as argument
and assigns a specific number between 1 and *n* to every element of the list:

```haskell
ghci> assignToGroups 3 [
  ]

ghci> assignToGroups 2 ["Bob","Kathie","Sue","Joan","Jim","Mike"]
```