Lists

Lists

The list is a fundamental data structure in functional programming.

A list having $x_1, ..., x_n$ as elements is written $List(x_1, ..., x_n)$

Example

```scala
val fruit  = List("apples", "oranges", "pears")
val nums   = List(1, 2, 3, 4)
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty  = List()
```

There are two important differences between lists and arrays.

▶ Lists are immutable — the elements of a list cannot be changed.

▶ Lists are recursive, while arrays are flat.

Lists

```scala
val fruit  = List("apples", "oranges", "pears")
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
```

The List Type

Like arrays, lists are homogeneous: the elements of a list must all have the same type.

The type of a list with elements of type `T` is written `scala.List[T]` or shorter just `List[T]`

Example

```scala
val fruit: List[String]    = List("apples", "oranges", "pears")
val nums : List[Int]       = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Nothing]   = List()
```

Constructors of Lists

All lists are constructed from:

- the empty list `Nil`, and
- the construction operation `::` (pronounced *cons*):
  `x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`.

For example:

```
fruit = "apples" :: ("oranges" :: ("pears" :: Nil))
nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
empty = Nil
```

Right Associativity

Convention: Operators ending in ":" associate to the right.

    `A :: B :: C` is interpreted as `A :: (B :: C)`.

We can thus omit the parentheses in the definition above.

Example

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

Operators ending in ":" are also different in the they are seen as
method calls of the *right-hand* operand.

So the expression above is equivalent to

```
Nil.::(4).::(3).::(2).::(1)
```

Operations on Lists

All operations on lists can be expressed in terms of the following three operations:

head	the first element of the list
tail	the list composed of all the elements except the first.
isEmpty	'true' if the list is empty, 'false' otherwise.

These operations are defined as methods of objects of type list. For example:

```
fruit.head      == "apples"
fruit.tail.head == "oranges"
diag3.head      == List(1, 0, 0)
empty.head      == throw new NoSuchElementException("head of empty list")
```

List Patterns

It is also possible to decompose lists with pattern matching.

`Nil`	The `Nil` constant
`p :: ps`	A pattern that matches a list with a `head` matching `p` and a `tail` matching `ps`.
`List(p1, ..., pn)`	same as `p1 :: ... :: pn :: Nil`

Example

`1 :: 2 :: xs`	Lists of that start with `1` and then `2`
`x :: Nil`	Lists of length 1
`List(x)`	Same as `x :: Nil`
`List()`	The empty list, same as `Nil`
`List(2 :: xs)`	A list that contains as only element another list that starts with `2`.

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length `L` of the lists it matches?

O	`L == 3`
O	`L == 4`
O	`L == 5`
O	`L >= 3`
O	`L >= 4`
O	`L >= 5`

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length L of the lists it matches?

O	L == 3
O	L == 4
O	L == 5
O	L >= 3
O	L >= 4
O	L >= 5

Sorting Lists

Suppose we want to sort a list of numbers in ascending order:

▶ One way to sort the list List(7, 3, 9, 2) is to sort the tail List(3, 9, 2) to obtain List(2, 3, 9).

▶ The next step is to insert the head 7 in the right place to obtain the result List(2, 3, 7, 9).

This idea describes *Insertion Sort* :

```
def isort(xs: List[Int]): List[Int] = xs match {
  case List() => List()
  case y :: ys => insert(y, isort(ys))
}
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List() => ???
  case y :: ys => ???
}
```

What is the worst-case complexity of insertion sort relative to the length of the input list N?

O   the sort takes constant time
O   proportional to N
O   proportional to N log(N)
O   proportional to N * N

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List() =>
  case y :: ys =>
}
```

What is the worst-case complexity of insertion sort relative to the length of the input list N?

```
O       the sort takes constant time
O       proportional to N
O       proportional to N * log(N)
O       proportional to N * N
```