Lists

Lists

The list is a fundamental data structure in functional programming.

A list having $x_1, \ldots, x_n$ as elements is written `List(`$x_1, \ldots, x_n$`)`

Example

```
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

```
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

```
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
```

More Functions on Lists

List Methods (1)

Sublists and element access:

`xs.length`	The number of elements of `xs`.
`xs.last`	The list's last element, exception if `xs` is empty.
`xs.init`	A list consisting of all elements of `xs` except the last one, exception if `xs` is empty.
`xs take n`	A list consisting of the first `n` elements of `xs`, or `xs` itself if it is shorter than `n`.
`xs drop n`	The rest of the collection after taking `n` elements.
`xs(n)`	(or, written out, `xs apply n`). The element of `xs` at index `n`.

List Methods (2)

Creating new lists:

`xs ++ ys`	The list consisting of all elements of `xs` followed by all elements of `ys`.
`xs.reverse`	The list containing the elements of `xs` in reversed order.
`xs updated (n, x)`	The list containing the same elements as `xs`, except at index `n` where it contains `x`.

Finding elements:

`xs indexOf x`	The index of the first element in `xs` equal to `x`, or `-1` if `x` does not appear in `xs`.
`xs contains x`	same as `xs indexOf x >= 0`

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error("last of empty list")
  case List(x) =>
  case y :: ys =>
}
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error("last of empty list")
  case List(x) => x
  case y :: ys =>
}
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error("last of empty list")
  case List(x) => x
  case y :: ys => last(ys)
}
```

Implementation of `last`

The complexity of `head` is (small) constant time.

What is the complexity of `last`?

To find out, let's write a possible implementation of `last` as a stand-alone function.

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error(”last of empty list”)
  case List(x) => x
  case y :: ys => last(ys)
}
```

So, `last` takes steps proportional to the length of the list `xs`.

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match {
  case List() => throw new Error(”init of empty list”)
  case List(x) => ???
  case y :: ys => ???
}
```

Exercise

Implement `init` as an external function, analogous to `last`.

```
def init[T](xs: List[T]): List[T] = xs match {
  case List() => throw new Error("init of empty list")
  case List(x) =>
  case y :: ys =>
}
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) =
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() =>
  case z :: zs =>
}
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case z :: zs =>
}
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case z :: zs => z :: concat(zs, ys)
}
```

Implementation of Concatenation

How can concatenation be implemented?

Let's try by writing a stand-alone function:

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case z :: zs => z :: concat(zs, ys)
}
```

What is the complexity of `concat`?

Implementation of reverse

How can reverse be implemented?

Let's try by writing a stand-alone function:

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() =>
  case y :: ys =>
}
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing a stand-alone function:

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() => List()
  case y :: ys =>
}
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing a stand-alone function:

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() => List()
  case y :: ys => reverse(ys) ++ List(y)
}
```

Implementation of `reverse`

How can reverse be implemented?

Let's try by writing a stand-alone function:

```
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() => List()
  case y :: ys => reverse(ys) ++ List(y)
}
```

What is the complexity of `reverse`?

Can we do better? (to be solved later).

Exercise

Remove the `n`'th element of a list `xs`. If `n` is out of bounds, return `xs` itself.

```
def removeAt[T](n: Int, xs: List[T]) = ???
```

Usage example:

```
removeAt(1, List('a', 'b', 'c', 'd'))
```
> *List(a, c, d)*

Exercise (Harder, Optional)

Flatten a list structure:

```scala
def flatten(xs: List[Any]): List[Any] = ???

flatten(List(List(1, 1), 2, List(3, List(5, 8))))
            >   res0: List[Any] = List(1, 1, 2, 3, 5, 8)
```

Pairs and Tuples

Sorting Lists Faster

As a non-trivial example, let's design a function to sort lists that is more efficient than insertion sort.

A good algorithm for this is *merge sort*. The idea is as follows:

If the list consists of zero or one elements, it is already sorted.

Otherwise,

- Separate the list into two sub-lists, each containing around half of the elements of the original list.
- Sort the two sub-lists.
- Merge the two sorted sub-lists into a single sorted list.

First MergeSort Implementation

Here is the implementation of that algorithm in Scala:

```scala
def msort(xs: List[Int]): List[Int] = {
  val n = xs.length/2
  if (n == 0) xs
  else {
    def merge(xs: List[Int], ys: List[Int]) = ???
    val (fst, snd) = xs splitAt n
    merge(msort(fst), msort(snd))
  }
}
```

Definition of Merge

Here is a definition of the `merge` function:

```
def merge(xs: List[Int], ys: List[Int]) =
  xs match {
    case Nil =>
      ys
    case x :: xs1 =>
      ys match {
        case Nil =>
          xs
        case y :: ys1 =>
          if (x < y) x :: merge(xs1, ys)
          else y :: merge(xs, ys1)
      }
    }
```

The SplitAt Function

The `splitAt` function on lists returns two sublists

- the elements up the the given index
- the elements from that index

The lists are returned in a *pair*.

Detour: Pair and Tuples

The pair consisting of `x` and `y` is written `(x, y)` in Scala.

Example

```
val pair = ("answer", 42)
```
> *pair : (String, Int) = (answer,42)*

The type of `pair` above is `(String, Int)`.

Pairs can also be used as patterns:

```
val (label, value) = pair
```
> *label : String = answer*
| *value : Int = 42*

This works analogously for tuples with more than two elements.

Translation of Tuples

A tuple type $(\mathsf{T}_1, ..., \mathsf{T}_n)$ is an abbreviation of the parameterized type

$$\texttt{scala.Tuple}n[\mathsf{T}_1, ..., \mathsf{T}_n]$$

A tuple expression $(\mathsf{e}_1, ..., \mathsf{e}_n)$ is equivalent to the function application

$$\texttt{scala.Tuple}n(\mathsf{e}_1, ..., \mathsf{e}_n)$$

A tuple pattern $(\mathsf{p}_1, ..., \mathsf{p}_n)$ is equivalent to the constructor pattern

$$\texttt{scala.Tuple}n(\mathsf{p}_1, ..., \mathsf{p}_n)$$

The Tuple class

Here, all `Tuple`*n* classes are modeled after the following pattern:

```
case class Tuple2[T1, T2](_1: +T1, _2: +T2) {
  override def toString = "(" + _1 + "," + _2 +")"
}
```

The fields of a tuple can be accessed with names _1, _2, ...

So instead of the pattern binding

```
val (label, value) = pair
```

one could also have written:

```
val label = pair._1
val value = pair._2
```

But the pattern matching form is generally preferred.

Exercise

The `merge` function as given uses a nested pattern match.

This does not reflect the inherent symmetry of the merge algorithm.

Rewrite `merge` using a pattern matching over pairs.

```
def merge(xs: List[Int], ys: List[Int]): List[Int] =
  (xs, ys) match {
    ???
  }
```

Implicit Parameters

Making Sort more General

Problem: How to parameterize `msort` so that it can also be used for lists with elements other than `Int`?

```
def msort[T](xs: List[T]): List[T] = ...
```

does not work, because the comparison < in `merge` is not defined for arbitrary types `T`.

Idea: Parameterize `merge` with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) = {
  ...
    merge(msort(fst)(lt), msort(snd)(lt))
}
```

Merge then needs to be adapted as follows:

```
def merge(xs: List[T], ys: List[T]) = (xs, ys) match {
  ...
  case (x :: xs1, y :: ys1) =>
    if (lt(x, y)) ...
    else ...
}
```

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val fruit = List("apple", "pear", "orange", "pineapple")

merge(xs)((x: Int, y: Int) => x < y)
merge(fruit)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `merge(xs)`:

```
merge(xs)((x, y) => x < y)
```

Parametrization with Ordered

There is already a class in the standard library that represents orderings.

```
scala.math.Ordering[T]
```

provides ways to compare elements of type `T`. So instead of parameterizing with the `lt` operation directly, we could parameterize with `Ordering` instead:

```
def msort[T](xs: List[T])(ord: Ordering) =

  def merge(xs: List[T], ys: List[T]) =
    ... if (ord.lt(x, y)) ...

  ... merge(msort(fst)(ord), msort(snd)(ord)) ...
```

Ordered Instances:

Calling the new `msort` can be done like this:

```
import math.Ordering

msort(nums)(Ordering.Int)
msort(fruits)(Ordering.String)
```

This makes use of the values `Int` and `String` defined in the `scala.math.Ordering` object, which produce the right orderings on integers and strings.

Aside: Implicit Parameters

Problem: Passing around `lt` or `ord` values is cumbersome.

We can avoid this by making `ord` an implicit parameter.

```
def msort[T](xs: List[T])(implicit ord: Ordering) =

  def merge(xs: List[T], ys: List[T]) =
    ... if (ord.lt(x, y)) ...

  ... merge(msort(fst), msort(snd)) ...
```

Then calls to `msort` can avoid the ordering parameters:

```
msort(nums)
msort(fruits)
```

The compiler will figure out the right implicit to pass based on the demanded type.

Rules for Implicit Parameters

Say, a function takes an implicit parameter of type `T`.

The compiler will search an implicit definition that

- is marked `implicit`
- has a type compatible with `T`
- is visible at the point of the function call, or is defined in a companion object associated with `T`.

If there is a single (most specific) definition, it will be taken as actual argument for the implicit parameter.

Otherwise it's an error.

Exercise: Implicit Parameters

Consider the following line of the definition of `msort`:

```
... merge(msort(fst), msort(snd)) ...
```

Which implicit argument is inserted?

- O `Ordering.Int`
- O `Ordering.String`
- O `the "ord" parameter of "msort"`

Higher-order List Functions

Recurring Patterns for Computations on Lists

The examples have shown that functions on lists often have similar structures.

We can identify several recurring patterns, like,

- transforming each element in a list in a certain way,
- retrieving a list of all elements satisfying a criterion,
- combining the elements of a list using an operator.

Functional languages allow programmers to write generic functions that implement patterns such as these using higher-order functions.

Applying a Function to Elements of a List

A common operation is to transform each element of a list and then return the list of results.

For example, to multiply each element of a list by the same factor, you could write:

```
def scaleList(xs: List[Double], factor: Double): List[Double] = xs match {
  case Nil      => xs
  case y :: ys => y * factor :: scaleList(ys, factor)
}
```

Map

This scheme can be generalized to the method `map` of the `List` class. A simple way to define `map` is as follows:

```
abstract class List[T] { ...
  def map[U](f: T => U): List[U] = this match {
    case Nil      => this
    case x :: xs => f(x) :: xs.map(f)
  }
```

(in fact, the actual definition of `map` is a bit more complicated, because it is tail-recursive, and also because it works for arbitrary collections, not just lists).

Using `map`, `scaleList` can be written more concisely.

```
def scaleList(xs: List[Double], factor: Double) =
  xs map (x => x * factor)
```

Exercise

Consider a function to square each element of a list, and return the result. Complete the two following equivalent definitions of `squareList`.

```
def squareList(xs: List[Int]): List[Int] = xs match {
  case Nil     => ???
  case y :: ys => ???
}

def squareList(xs: List[Int]): List[Int] =
  xs map ???
```

Exercise

Consider a function to square each element of a list, and return the result. Complete the two following equivalent definitions of `squareList`.

```
def squareList(xs: List[Int]): List[Int] = xs match {
  case Nil     =>
  case y :: ys =>
}
```

```
def squareList(xs: List[Int]): List[Int] =
  xs map
```

Filtering

Another common operation on lists is the selection of all elements satisfying a given condition. For example:

```
def posElems(xs: List[Int]): List[Int] = xs match {
  case Nil      => xs
  case y :: ys => if (y > 0) y :: posElems(ys) else posElems(ys)
}
```

Filter

This pattern is generalized by the method `filter` of the `List` class:

```
abstract class List[T] {
  ...
  def filter(p: T => Boolean): List[T] = this match {
    case Nil     => this
    case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
  }
}
```

Using `filter`, `posElems` can be written more concisely.

```
def posElems(xs: List[Int]): List[Int] =
  xs filter (x => x > 0)
```

Variations of Filter

Besides filter, there are also the following methods that extract sublists based on a predicate:

`xs filterNot p`	Same as `xs filter (x => !p(x))`; The list consisting of those elements of `xs` that do not satisfy the predicate `p`.
`xs partition p`	Same as `(xs filter p, xs filterNot p)`, but computed in a single traversal of the list `xs`.
`xs takeWhile p`	The longest prefix of list `xs` consisting of elements that all satisfy the predicate `p`.
`xs dropWhile p`	The remainder of the list `xs` after any leading elements satisfying `p` have been removed.
`xs span p`	Same as `(xs takeWhile p, xs dropWhile p)` but computed in a single traversal of the list `xs`.

Exercise

Write a function pack that packs consecutive duplicates of list elements into sublists. For instance,

```
pack(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(List("a", "a", "a"), List("b"), List("c", "c"), List("a")).
```

You can use the following template:

```
def pack[T](xs: List[T]): List[List[T]] = xs match {
  case Nil      => Nil
  case x :: xs1 => ???
}
```

Exercise

Using `pack`, write a function `encode` that produces the run-length encoding of a list.

The idea is to encode `n` consecutive duplicates of an element `x` as a pair `(x, n)`. For instance,

```
encode(List("a", "a", "a", "b", "c", "c", "a"))
```

should give

```
List(("a", 3), ("b", 1), ("c", 2), ("a", 1)).
```

Reduction of Lists

Reduction of Lists

Another common operation on lists is to combine the elements of a list using a given operator.

For example:

```
sum(List(x1, ..., xn))          =  0 + x1 + ... + xn
product(List(x1, ..., xn))      =  1 * x1 * ... * xn
```

We can implement this with the usual recursive schema:

```
def sum(xs: List[Int]): Int = xs match {
  case Nil      => 0
  case y :: ys => y + sum(ys)
}
```

ReduceLeft

This pattern can be abstracted out using the generic method `reduceLeft`:

`reduceLeft` inserts a given binary operator between adjacent elements of a list:

```
List(x1, ..., xn) reduceLeft op   =  (...(x1 op x2) op ... ) op xn
```

Using `reduceLeft`, we can simplify:

```
def sum(xs: List[Int])     = (0 :: xs) reduceLeft ((x, y) => x + y)
def product(xs: List[Int]) = (1 :: xs) reduceLeft ((x, y) => x * y)
```

A Shorter Way to Write Functions

Instead of `((x, y) => x * y))`, one can also write shorter:

```
(_ * _)
```

Every _ represents a new parameter, going from left to right.

The parameters are defined at the next outer pair of parentheses (or the whole expression if there are no enclosing parentheses).

So, `sum` and `product` can also be expressed like this:

```
def sum(xs: List[Int])     = (0 :: xs) reduceLeft (_ + _)
def product(xs: List[Int]) = (1 :: xs) reduceLeft (_ * _)
```

FoldLeft

The function `reduceLeft` is defined in terms of a more general function, `foldLeft`.

`foldLeft` is like `reduceLeft` but takes an *accumulator*, `z`, as an additional parameter, which is returned when `foldLeft` is called on an empty list.

```
(List(x1, ..., xn) foldLeft z)(op)   =  (...(z op x1) op ... ) op xn
```

So, `sum` and `product` can also be defined as follows:

```
def sum(xs: List[Int])      = (xs foldLeft 0) (_ + _)
def product(xs: List[Int])  = (xs foldLeft 1) (_ * _)
```

Implementations of ReduceLeft and FoldLeft

`foldLeft` and `reduceLeft` can be implemented in class `List` as follows.

```
abstract class List[T] { ...
  def reduceLeft(op: (T, T) => T): T = this match {
    case Nil     => throw new Error(”Nil.reduceLeft”)
    case x :: xs => (xs foldLeft x)(op)
  }
  def foldLeft[U](z: U)(op: (U, T) => U): U = this match {
    case Nil     => z
    case x :: xs => (xs foldLeft op(z, x))(op)
  }
}
```

FoldRight and ReduceRight

Applications of `foldLeft` and `reduceLeft` unfold on trees that lean to the left.

They have two dual functions, `foldRight` and `reduceRight`, which produce trees which lean to the right, i.e.,

```
List(x1, ..., x{n-1}, xn) reduceRight op = x1 op ( ... (x{n-1} op xn) ... )
(List(x1, ..., xn) foldRight acc)(op)    = x1 op ( ... (xn op acc) ... )
```

Implementation of FoldRight and ReduceRight

They are defined as follows

```
def reduceRight(op: (T, T) => T): T = this match {
  case Nil => throw new Error("Nil.reduceRight")
  case x :: Nil => x
  case x :: xs => op(x, xs.reduceRight(op))
}
def foldRight[](z: U)(op: (T, U) => U): U = this match {
  case Nil => z
  case x :: xs => op(x, (xs foldRight z)(op))
}
```

Difference between FoldLeft and FoldRight

For operators that are associative and commutative, `foldLeft` and `foldRight` are equivalent (even though there may be a difference in efficiency).

But sometimes, only one of the two operators is appropriate.

Exercise

Here is another formulation of `concat`:

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  (xs foldRight ys) (_ :: _)
```

Here, it isn't possible to replace `foldRight` by `foldLeft`. Why?

```
O     The types would not work out
O     The resulting function would not terminate
O     The result would be reversed
```

Back to Reversing Lists

We now develop a function for reversing lists which has a linear cost.

The idea is to use the operation `foldLeft`:

```
def reverse[T](xs: List[T]): List[T] = (xs foldLeft z?)(op?)
```

All that remains is to replace the parts `z?` and `op?`.

Let's try to *compute* them from examples.

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
Nil
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
  Nil

=   reverse(Nil)
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
  Nil

=   reverse(Nil)

=   (Nil foldLeft z?)(op)
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
  Nil
=   reverse(Nil)
=   (Nil foldLeft z?)(op)
=   z?
```

Consequently, `z? = List()`

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))

=   (List(x) foldLeft Nil)(op?)
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)
=   reverse(List(x))
=   (List(x) foldLeft Nil)(op?)
=   op?(Nil, x)
```

Consequently, `op?(Nil, x) = List(x) = x :: List()`.

This suggests to take for `op?` the operator `::` but with its operands swapped.

Deduction of Reverse(3)

We thus arrive at the following implementation of `reverse`.

```
def reverse[a](xs: List[T]): List[T] =
  (xs foldLeft List[T]())((xs, x) => x :: xs)
```

Remark: the type parameter in `List[T]()` is necessary for type inference.

Question: What is the complexity of this implementation of `reverse` ?

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  (xs foldRight List[U]())( ??? )

def lengthFun[T](xs: List[T]): Int =
  (xs foldRight 0)( ??? )
```

Reasoning About Lists

Laws of Concat

Recall the concatenation operation `++` on lists.

We would like to verify that concatenation is associative, and that it admits the empty list `Nil` as neutral element to the left and to the right:

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
       xs ++ Nil  =  xs
       Nil ++ xs  =  xs
```

Q: How can we prove properties like these?

Laws of Concat

Recall the concatenation operation ++ on lists.

We would like to verify that concatenation is associative, and that it admits the empty list `Nil` as neutral element to the left and to the right:

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
       xs ++ Nil  =  xs
       Nil ++ xs  =  xs
```

Q: How can we prove properties like these?

A: By *structural induction* on lists.

Reminder: Natural Induction

Recall the principle of proof by *natural induction*:

To show a property $P(n)$ for all the integers $n \geq b$,

- Show that we have $P(b)$ (*base case*),
- for all integers $n \geq b$ show the *induction step*:

  if one has $P(n)$*, then one also has* $P(n+1)$*.*

Example

Given:

```
def factorial(n: Int): Int =
  if (n == 0) 1            // 1st clause
  else n * factorial(n-1)  // 2nd clause
```

Show that, for all `n >= 4`

```
factorial(n) >= power(2, n)
```

Base Case

Base case: 4

This case is established by simple calculations:

```
factorial(4) = 24 >= 16 = power(2, 4)
```

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)
```

Induction Step

Induction step: n+1

We have for `n >= 4`:

```
factorial(n + 1)

>=  (n + 1) * factorial(n)    // by 2nd clause in factorial
```

Induction Step

Induction step: n+1

We have for `n >= 4`:

```
factorial(n + 1)

>=  (n + 1) * factorial(n)    // by 2nd clause in factorial

>   2 * factorial(n)          // by calculating
```

Induction Step

Induction step: `n+1`

We have for `n >= 4`:

```
factorial(n + 1)

>=  (n + 1) * factorial(n)   // by 2nd clause in factorial

>   2 * factorial(n)         // by calculating

>=  2 * power(2, n)          // by induction hypothesis
```

Induction Step

Induction step: n+1

We have for `n >= 4`:

```
factorial(n + 1)

>=  (n + 1) * factorial(n)    // by 2nd clause in factorial

>   2 * factorial(n)          // by calculating

>=  2 * power(2, n)           // by induction hypothesis

=   power(2, n + 1)           // by definition of power
```

Referential Transparency

Note that a proof can freely apply reduction steps as equalities to some part of a term.

That works because pure functional programs don't have side effects; so that a term is equivalent to the term to which it reduces.

This principle is called *referential transparency*.

Structural Induction

The principle of structural induction is analogous to natural induction:

To prove a property `P(xs)` for all lists `xs`,

- show that `P(Nil)` holds (*base case*),
- for a list `xs` and some element `x`, show the *induction step*:

  if `P(xs)` *holds, then* `P(x :: xs)` *also holds.*

Example

Let's show that, for lists `xs`, `ys`, `zs`:

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
```

To do this, use structural induction on `xs`. From the previous implementation of `concat`,

```
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() => ys
  case x :: xs1 => x :: concat(xs1, ys)
}
```

distill two *defining clauses* of `++`:

```
      Nil ++ ys  =  ys                 // 1st clause
(x :: xs1) ++ ys  =  x :: (xs1 ++ ys)   // 2nd clause
```

Base Case

$\boxed{\textbf{Base case: } \texttt{Nil}}$

For the left-hand side we have:

```
(Nil ++ ys) ++ zs
```

Base Case

$\boxed{\textbf{Base case: } \texttt{Nil}}$

For the left-hand side we have:

```
(Nil ++ ys) ++ zs

=   ys ++ zs        // by 1st clause of ++
```

Base Case

Base case: `Nil`

For the left-hand side we have:

```
(Nil ++ ys) ++ zs

=   ys ++ zs        // by 1st clause of ++
```

For the right-hand side, we have:

```
Nil ++ (ys ++ zs)
```

Base Case

Base case: `Nil`

For the left-hand side we have:

```
(Nil ++ ys) ++ zs

=   ys ++ zs        // by 1st clause of ++
```

For the right-hand side, we have:

```
Nil ++ (ys ++ zs)

=   ys ++ zs        // by 1st clause of ++
```

This case is therefore established.

Induction Step: LHS

$\boxed{\textbf{Induction step: } \texttt{x :: xs}}$

For the left-hand side, we have:

```
((x :: xs) ++ ys) ++ zs
```

Induction Step: LHS

$\boxed{\textbf{Induction step: } \texttt{x :: xs}}$

For the left-hand side, we have:

```
((x :: xs) ++ ys) ++ zs

=   (x :: (xs ++ ys)) ++ zs    // by 2nd clause of ++
```

Induction Step: LHS

$\boxed{\textbf{Induction step: } \texttt{x :: xs}}$

For the left-hand side, we have:

```
((x :: xs) ++ ys) ++ zs

=   (x :: (xs ++ ys)) ++ zs    // by 2nd clause of ++

=   x :: ((xs ++ ys) ++ zs)    // by 2nd clause of ++
```

Induction Step: LHS

Induction step: `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ++ ys) ++ zs

=   (x :: (xs ++ ys)) ++ zs    // by 2nd clause of ++

=   x :: ((xs ++ ys) ++ zs)    // by 2nd clause of ++

=   x :: (xs ++ (ys ++ zs))    // by induction hypothesis
```

Induction Step: RHS

For the right hand side we have:

```
(x :: xs) ++ (ys ++ zs)
```

Induction Step: RHS

For the right hand side we have:

```
(x :: xs) ++ (ys ++ zs)

=   x :: (xs ++ (ys ++ zs))    // by 2nd clause of ++
```

So this case (and with it, the property) is established.

Exercise

Show by induction on `xs` that `xs ++ Nil = xs`.

How many equations do you need for the inductive step?

0 2

0 3

0 4

A Larger Equational Proof on Lists

A Law of Reverse

For a more difficult example, let's consider the `reverse` function.

We pick its inefficient definition, because its more amenable to equational proofs:

```
      Nil.reverse  =  Nil                         // 1st clause
(x :: xs).reverse  =  xs.reverse ++ List(x)       // 2nd clause
```

We'd like to prove the following proposition

```
xs.reverse.reverse  =  xs
```

Proof

By induction on `xs`. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse           // by 1st clause of reverse
=   Nil                   // by 1st clause of reverse
```

Proof

By induction on `xs`. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse            // by 1st clause of reverse
=   Nil                    // by 1st clause of reverse
```

For the induction step, let's try:

```
(x :: xs).reverse.reverse
=   (xs.reverse ++ List(x)).reverse // by 2nd clause of reverse
```

Proof

By induction on `xs`. The base case is easy:

```
Nil.reverse.reverse
=   Nil.reverse            // by 1st clause of reverse
=   Nil                    // by 1st clause of reverse
```

For the induction step, let's try:

```
(x :: xs).reverse.reverse
=   (xs.reverse ++ List(x)).reverse // by 2nd clause of reverse
```

We can't do anything more with this expression, therefore we turn to the right-hand side:

```
x :: xs
=   x :: xs.reverse.reverse          // by induction hypothesis
```

Both sides are simplified in different expressions.

To Do

We still need to show:

```
(xs.reverse ++ List(x)).reverse  =  x :: xs.reverse.reverse
```

Trying to prove it directly by induction doesn't work.

We must instead try to *generalize* the equation. For *any* list `ys`,

```
(ys ++ List(x)).reverse  =  x :: ys.reverse
```

This equation can be proved by a second induction argument on `ys`.

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse

=   List(x).reverse             // by 1st clause of ++
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse

=   List(x).reverse             // by 1st clause of ++

=   (x :: Nil).reverse          // by definition of List
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse        // to show: =  x :: Nil.reverse

=   List(x).reverse             // by 1st clause of ++

=   (x :: Nil).reverse          // by definition of List

=   Nil ++ (x :: Nil)           // by 2nd clause of reverse
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse         // to show: =  x :: Nil.reverse

=   List(x).reverse              // by 1st clause of ++

=   (x :: Nil).reverse           // by definition of List

=   Nil ++ (x :: Nil)            // by 2nd clause of reverse

=   x :: Nil                     // by 1st clause of ++
```

Auxiliary Equation, Base Case

```
(Nil ++ List(x)).reverse       // to show: =  x :: Nil.reverse

=   List(x).reverse            // by 1st clause of ++

=   (x :: Nil).reverse         // by definition of List

=   Nil ++ (x :: Nil)          // by 2nd clause of reverse

=   x :: Nil                   // by 1st clause of ++

=   x :: Nil.reverse           // by 1st clause of reverse
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse            // to show: =  x :: (y :: ys).reverse
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse             // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse         // by 2nd clause of ++
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse           // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse       // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y)   // by 2nd clause of reverse
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse            // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse        // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y)    // by 2nd clause of reverse

=   (x :: ys.reverse) ++ List(y)          // by the induction hypothesis
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse           // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse       // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y)   // by 2nd clause of reverse

=   (x :: ys.reverse) ++ List(y)         // by the induction hypothesis

=   x :: (ys.reverse ++ List(y))         // by 1st clause of ++
```

Auxiliary Equation, Inductive Step

```
((y :: ys) ++ List(x)).reverse         // to show: =  x :: (y :: ys).reverse

=   (y :: (ys ++ List(x))).reverse     // by 2nd clause of ++

=   (ys ++ List(x)).reverse ++ List(y) // by 2nd clause of reverse

=   (x :: ys.reverse) ++ List(y)       // by the induction hypothesis

=   x :: (ys.reverse ++ List(y))       // by 1st clause of ++

=   x :: (y :: ys).reverse             // by 2nd clause of reverse
```

This establishes the auxiliary equation, and with it the main proposition.

Exercise (Open-Ended, Harder)

Prove the following distribution law for `map` over concatenation.

For any lists `xs`, `ys`, function `f`:

```
(xs ++ ys) map f  =  (xs map f) ++ (ys map f)
```

You will need the clauses of $++$ as well as the following clauses for `map`:

```
      Nil map f  =  Nil
 (x :: xs) map f  =  f(x) :: (xs map f)
```