Reduction of Lists

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

For example:

\[
\begin{align*}
  \text{sum}(\text{List}(x_1, \ldots, x_n)) &= 0 + x_1 + \ldots + x_n \\
  \text{product}(\text{List}(x_1, \ldots, x_n)) &= 1 \times x_1 \times \ldots \times x_n
\end{align*}
\]

We can implement this with the usual recursive schema:

```python
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:

\[
\text{List}(x_1, \ldots, x_n) \ \text{reduceLeft} \ op \ = \ (\ldots(x_1 \ op \ x_2) \ op \ \ldots) \ op \ x_n
\]

Using `reduceLeft`, we can simplify:

```scala
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 \(((x, y) \Rightarrow x \times y)\), one can also write shorter:

\((\_ \times \_\))

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:

```scala
def sum(xs: List[Int]) = (0 :: xs) reduceLeft (\_ + \_)
def product(xs: List[Int]) = (1 :: xs) reduceLeft (\_ \times \_)
```
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.

\[
\text{(List}(x_1, \ldots, x_n) \text{ foldLeft } z)(\text{op}) = (\ldots(\text{z op } x_1) \text{ op } \ldots) \text{ op } x_n
\]

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

```scala
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.

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

```python
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`:

```python
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?

0 The types would not work out
0 The resulting function would not terminate
0 The result would be reversed
We now develop a function for reversing lists which has a linear cost. The idea is to use the operation foldLeft:

```scala
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.
To start computing $z$, let's consider $\text{reverse}(\text{Nil})$.

We know $\text{reverse}(\text{Nil}) = \text{Nil}$, so we can compute as follows:

$\text{Nil}$
To start computing $z$, let’s consider reverse(Nil).

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

\[
\text{Nil} = \text{reverse}(\text{Nil})
\]
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)
```
To start computing $z?$, let's consider $\text{reverse(Nil)}$.

We know $\text{reverse(Nil)} = \text{Nil}$, so we can compute as follows:

\[
\begin{align*}
\text{Nil} \\
= & \quad \text{reverse(Nil)} \\
= & \quad (\text{Nil foldLeft } z?)(\text{op}) \\
= & \quad z\? \\
\end{align*}
\]

Consequently, $z\? = \text{List()}$
We still need to compute $op\,?$. To do that let’s plug in the next simplest list after $\text{Nil}$ into our equation for reverse:

$$\text{List}(x)$$
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:

\[
\text{List}(x)
\]

\[
= \text{reverse(List}(x))
\]
We still need to compute $\text{op}$. To do that let’s plug in the next simplest list after $\text{Nil}$ into our equation for reverse:

\[
\text{List}(x) = \text{reverse}(\text{List}(x)) = (\text{List}(x) \text{foldLeft} \text{Nil})(\text{op})
\]
We still need to compute $\text{op}\,?\,$. To do that let's plug in the next simplest list after $\text{Nil}$ into our equation for $\text{reverse}$:

\[
\text{List}(x)
\]

\[
= \text{reverse}\left(\text{List}(x)\right)
\]

\[
= \left(\text{List}(x) \text{ foldLeft} \text{ Nil}\right)(\text{op}\,?)
\]

\[
= \text{op}\,?(\text{Nil}, x)
\]

Consequently, $\text{op}\,?(\text{Nil}, x) = \text{List}(x) = x :: \text{List}()$.

This suggests to take for $\text{op}\,?$ the operator $::$ but with its operands swapped.
Deduction of Reverse(3)

We thus arrive at the following implementation of reverse.

```scala
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`:

```python
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)( ??? )
```