Consider the following problems:

Sum a list of integers:

```scala
def sum (l:List[Int]) : Int = l match {
case Nil => 0
case h::t => h + sum(t)
}
```

Multiply a list of integers:

```scala
def prod (l:List[Int]) : Int = l match {
case Nil => 1
case h::t => h * prod(t)
}
```

Calculate the length of a list (of any type):

```scala
def len[A](l:List[A]) : Int = l match {
case Nil => 0
case _::t => 1 + len(t)
}
```

Copy a list (of any type):

```scala
def copy[A](l:List[A]) : List[A] = l match {
case Nil => Nil
case h::t => h::copy(t)
}
```

Query: How does `copy` differ from the identity function `(x => x)`?

The pattern continues...

We can abstract over the common inductive pattern displayed by these examples:

```scala
def foldr[A,B] (c: (A,B) => B, n:B) (l:List[A]) : B = l match {
case Nil => n
case h::t => c (h,foldr(c,n)(t))
}
```

Function `foldr` computes a value working from the tail of the list to the head (from right to left). Argument `n` is the value to return for the empty list. Argument `c` is the function to apply to each element and the previously computed result.

The `foldr` function is Curried to make it convenient to partially apply it.
We can view $\text{foldr} \ (c, n) \ (l)$ as replacing each :: constructor in $l$ with $x$ and the $\text{Nil}$ constructor with $n$. For example:

\begin{verbatim}
l = x_1 :: (x_2 :: (\ldots :: (x_n :: \text{Nil}))))
foldr \ (_+_\ ,0) \ (l) =
\quad x_1 + (x_2 + (\ldots (x_n + 0)))
\end{verbatim}

It is also possible to define a $\text{foldl}$ that accumulates a value from the left; sometimes this will be more efficient.

In some languages, $\text{fold}$ is called $\text{reduce}$, because we “reduce” a list of values to a single value. A similar idea appears in “map-reduce” frameworks for organizing massively distributed computations.