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)`?
We can **abstract** over the common inductive pattern displayed by these examples:

\[
\text{def foldr}[A,B] (c: (A,B) \Rightarrow B, n:B) (l:List[A]) : B = l \text{ match } \{
\text{case Nil} \Rightarrow n
\text{case h::t} \Rightarrow c (h, \text{foldr}(c,n)(t))
\}
\]

\[
\text{val sum} = \text{foldr}[\text{Int},\text{Int}] ((x,y) \Rightarrow x+y,0) _
\text{val prod} = \text{foldr}[\text{Int},\text{Int}] (_*_,1) _
\text{def len}[A] = \text{foldr}[A,\text{Int}] ((_,y) \Rightarrow 1+y,0) _
\text{def copy}[A] = \text{foldr}[A,\text{List}[A]] (_::_,\text{Nil}) _
\]

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

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

\[
\begin{align*}
l & = x_1 :: (x_2 :: (\ldots :: (x_n :: \texttt{Nil}))) \\
\text{foldr} \ (\_+\_,0) \ (l) & = \\
& x_1 + (x_2 + (\ldots (x_n + 0)))
\end{align*}
\]

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

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