Fold

As we have seen in the case of *filter* functions, sometimes a function can be abstracted. Sometimes working with abstract functions may seem more cumbersome (it is) but at the same time it is more difficult to make silly errors.

A very important class of functions are the so-called *fold* functions, which abstract the structural recursion pattern itself.

Consider the following functions:

- Computing the maximum element of a list of Nats:

```
maxl : List Nat -> Nat
maxl nil = zero
maxl (cons n ns) = max n n' where
    n' = maxl ns
```

- Computing the length of a list:

```
len : {A : Set} -> List A -> Nat
len nil = zero
len (cons _ as) = suc n' where
  n' = len as
```

- Flattening a list of lists

```
concat : {A : Set} -> List (List A) -> List A
concat nil = nil
concat (cons xs xss) = append xs ys where
  ys = concat xss
```

Note that all these functions follow this pattern:

```
fold : List A -> B
fold nil = something
fold (conx x xs) = some computation using x and y where
  y = fold xs
```

The idea of `fold` is to make the `something` part and the `some computation using x and y` part as arguments:

```
foldr : {A B : Set} -> (A -> B -> B) -> B -> List A -> B
foldr f y0 nil = y0
foldr f y0 (cons x xs) = f x y where
  y = foldr f y0 xs
```

This is what is usually called *fold-right*. To simplify:

```
foldr : {A B : Set} -> (A -> B -> B) -> B -> List A -> B
foldr f y0 nil = y0
foldr f y0 (cons x xs) = f x (foldr f y0 xs)
```

What it achieves is *aggregating* all elements of the argument list using *f*, with *y0* a special value for the empty list.

Obs: `fold cons nil xs = xs`.

This is not the only way the values of a list can be aggregated. An alternative is the so-called *fold-left*:

```
foldl : {A B : Set} -> (B -> A -> B) -> B -> List A -> B
foldl f y0 nil         = y0
foldl f y0 (cons x xs) = foldl f y0' xs where
  y0' = f y0 x
```

or, to simplify

```
foldl : {A B : Set} -> (B -> A -> B) -> B -> List A -> B
foldl f y0 nil         = y0
foldl f y0 (cons x xs) = foldl f (f y0 x) xs
```

Unlike *fold-right*, fold-left will use the default value `y` to "accumulate" the resulting `B` to this point.

Obs: `foldl (\ xs x -> cons x xs) nil xs = rev xs`.

The reason the two functions are called *fold-right* and *fold-left* is apparent if we examine a concrete example. Let `xs` be the list `[0, 1,`

`2, 3]` (i.e. `cons 0 (cons 1 (cons 2 (cons 3 nil)))`).

Let us (informally) write the steps taken by the execution:

```
foldr sum 0 xs
    = sum 0 (foldr sum 0 [1, 2, 3])
    = sum 0 (sum 1 (foldr sum 0 [2, 3]))
    = sum 0 (sum 1 (sum 2 (foldr sum 0 [3])))
    = sum 0 (sum 1 (sum 2 (sum 3 (foldr sum 0 nil))))
    = sum 0 (sum 1 (sum 2 (sum 3 0)))
    = sum 0 (sum 1 (sum 2 3))
    = sum 0 (sum 1 5)
    = sum 0 6
    = 6
```

i.e. the folding function `f = sum` associates to the *right*. On the other hand,

```
foldl sum 0 xs
  = foldl sum (sum 0 0) [1, 2, 3]
  = foldl sum (sum (sum 0 0) 1) [2, 3]
  = foldl sum (sum (sum (sum 0 0) 1) 2) [3]
  = foldl sum (sum (sum (sum (sum 0 0) 1) 2) 3) nil
  = sum (sum (sum (sum 0 0) 1) 2) 3
  = sum (sum (sum 0 1) 2) 3
  = sum (sum 1 2) 3
  = sum 3 3
```

```
  = 6
```

associates `f = sum` to the *left*.

For practical applications fold-left tends to be preferred because it does not require “deep” function calls during recursion. Note that the `sum` can be computed before “unfolding” the `foldl` recursion:

```
foldl sum 0 xs
  = foldl sum (sum 0 0) [1, 2, 3]
  = foldl sum 0 [1, 2, 3]
  = foldl sum (sum 0 1) [2, 3]
  = foldl sum 1 [2, 3]
  = foldl sum (sum 1 2) [3]
  = foldl sum 3 [3]
  = foldl sum (sum 3 3) nil
  = foldl sum 6 nil
  = 6
```

Map

Another important generic function on lists is *map*:

```
map : {A B : Set} -> (A -> B) -> List A -> List B
map f nil = nil
map f (cons x xs) = cons (f x) (map f xs)
```

This function applies `f : A -> B` to each element of `List A` to create a `List B`.

> ***Exercise:*** *Implement* `map` *using* `fold`*.*

Map-reduce

In applications “fold” is sometimes called “reduce”.
The combination of map and reduce formed the backbone of Google’s distributed architecture for a while, and is still a widely distributed solution by Apache Hadoop. The reasons for its success are:

- `fold` is a generic function for any (inductive) data type, not just lists, which is as expressive as recursion but which abstracts recursion away.
- if `A = B`, i.e. `fold : (A -> A -> A) -> List A -> A` and the argument is an associative operation, i.e. `f (f x y) z = f x (f y z)` then it doesn’t matter if folding is on the left or on the right. In fact it can be tree-like, leading to greater efficiency.
- if the function `f` is a “pure function” (no assignment) then the folding tree of `f` can be executed in parallel.
- if the function `f` is stateless then any failed computation of `f` (e.g. because hardware or communication failure) can be restarted.
- if `f` is also commutative, i.e. `f x y = f y x` then the order in which failed computations are restarted does not matter either.
- similar considerations apply to `map`.

As a result map-reduce is an ideal architecture for *efficient* and *robust* execution of massive computations on unreliable hardware.

> ***Exercise (Hard)**: Using only fold (no explicit recursion allowed!) implement the filter function. You may implement helper functions, but they also need to be defined using fold only.*