Lists

Lists are to functional programming what arrays are to imperative programming: the main collection type. Lists are a *polymorphic* type:

```
data List (A : Set) : Set where
  nil  : List A
  cons : A -> List A -> List A
```

The two constructors are:

- `nil` : the empty list constructor
- `cons` : the constructor that takes an element of `A` (called *head*) and other list (called *tail*) and creates a longer list.

For example:

- `cons 1 nil` is the list [1]
- `cons 1 (cons 2 nil)` is the list [1, 2]
- `cons true (cons false nil)` is the list [true, false]

Note that the elements of the list must be of the same type because the `cons` requires the head to be the same type as the elements of the tail. So `cons true (cons zero nil)` is illegal:

Also note that the we can have a list of lists:

```
x = cons 0 nil          ... [0]
y = cons 1 x            ... [1, 0]
z = cons x (cons y nil) ... [[1, 0], [0]]
```

Operations on lists can be defined by pattern matching on the list constructors.

A very simple function is

Empty-check

```
empty : {A : Set} -> List A -> Boolean
empty nil         = true
empty (cons x xs) = false
```

or

```
empty : {A : Set} -> List A -> Boolean
empty nil = true
empty _   = false
```

Length of a list

Structural recursion has shape:

```
f nil         = ...
```

```
f (cons x xs) = ... f xs ...
```

It is customary to denote the *head* in pattern matching by `x, y, z, ...` and the tail by `xs, ys, zs, ...`.

```
len : {A : Set} -> List A -> Nat
len nil          = zero
len (cons _ xs) = suc (len xs)
```

Appending two lists

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

As in the case of addition, the way we think recursively is trying to rearrange the constructors so that we can use the recursive call:

```
... append xs ys ...
```

Note that for more complicated functions auxiliary definitions can be introduced by `where`:

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

```
  zs = append xs ys
```

Reversing a list

```
rev : {A : Set} -> List A -> List A
rev nil = nil
rev (cons x xs) = ... rev xs ...
```

If we reverse the tail `xs` then the head `x` of the original should be added to the end:

```
rev : {A : Set} -> List A -> List A
rev nil = nil
rev (cons x xs) = append (rev xs) (cons x nil)
```

Note that `append` requires two lists as arguments, so we cannot write

```
rev (cons x xs) = append (rev xs) x
```

We first needed to “lift” the element `x` to the singleton list `cons x nil`.

Filtering a list

Let us consider the function which takes a list of `Nat`s and selects only the non-zero elements:

```
filter-zeros : List Nat -> List Nat
filter-zeros nil = nil
filter-zeros (cons x xs) = ... xs' ... where
  xs' = filter zero xs
```

Supposing that we filter the 0s out of the tail `xs` in the recursive call, what about the head `x`?

- If `x` is `zero` we should just remove it and return `xs'`.
- If `s` is not zero, i.e. `suc y` we should keep it and return `cons x xs'`.

But how can we perform this test? There is no built-in `if` function, but we could define one:

```
if : {A : Set} -> Boolean -> A -> A -> A
if true  x  _ = x
if false _ y = y
```

and use the `is-zero : Nat -> Bool` function from earlier.

```
filter-zeros : List Nat -> List Nat
filter-zeros nil = nil
filter-zeros (cons x xs) = if (is-zero x) xs' (cons x xs')
 where
  xs' = filter-zeros xs
```

However, there is another, more concise way, by applying deep pattern matching on `x`:

```
filter-zeros : List Nat -> List Nat
filter-zeros nil = nil
filter-zeros (cons zero xs) = filter-zeros xs
filter-zeros (cons (suc n) xs) = cons (suc n) (filter-zeros xs)
```

But suppose that we have a function `is-even : Nat -> Boolean` to test for even-ness or `is-prime : Nat -> Boolean` to test for primality. Filtering for some given property has very similar shape:

```
filter-zeros : List Nat -> List Nat
filter-zeros nil = nil
filter-zeros (cons x xs) = if (is-zero x) xs' (cons x xs')
 where
  xs' = filter-zeros xs

filter-evens : List Nat -> List Nat
filter-evens nil = nil
filter-evens (cons x xs) = if (is-even x) xs' (cons x xs')
 where
  xs' = filter-evens xs

filter-primes : List Nat -> List Nat
filter-primes nil = nil
```

```
filter-primes (cons x xs) = if (is-prime x) xs' (cons x xs
') where
  xs' = filter-primes xs
```

This means that we can define a *generic* filter function for any property (predicate) expressed by a function `p : A -> Boolean` where `A` is the type of list elements:

```
filter : {A : Set} -> (A -> Bool) -> List A -> List A
filter p nil = nil
filter p (cons x xs) = if (p x) xs' (cons x xs') where
  xs' = filter p xs
```

A final variation on filtering is, for a given property `p` to return two lists: the elements satisfying `p` and those not satisfying `p` ; a _partition of the list.

```
split : {A : Set} -> (A -> Bool) -> List A -> Prod (List A
) (List A)
split p nil = pair nil nil
split p (cons x xs) = ... where
   ... = split p xs
```

Here we need to be aware that the recursive call returns a `pair`, but we need access to both components. Remember, that in order to access components of a pair we need the *projections*:

```
split : {A : Set} -> (A -> Bool) -> List A -> Prod (List A
) (List A)
split p nil = pair nil nil
split p (cons x xs) = ... where
   xs' = split p xs
   ys  = proj1 xs'
   zs  = proj2 xs'
```

and now depending on what `p x` returns we decide whether to append to `ys` or `zs`:

```
split : {A : Set} -> (A -> Bool) -> List A -> Prod (List A
) (List A)
split p nil = pair nil nil
split p (cons x xs) = if (p x) (pair (cons x ys) zs)
                               (pair ys (cons x zs)) where
   xs' = split p xs
   ys  = proj1 xs'
   zs  = proj2 xs'
```

Interleaving

Let us now take the opposite operation, of "interleaving" a pair of lists into a single list by alternating. We could write it as

```
intlv : {A : Set} -> Prod (List A) (List A) -> List A
```

but remembering the currying isomorphism we could just define it as

```
intlv : {A : Set} -> List A -> List A -> List A
intlv nil ys = ys
intlv xs nil = xs
intlv (cons x xs) (cons y ys) = cons x (cons y (intlv xs ys))
```

Advanced Agda pattern-matching (*optional*)

Agda has an advanced feature to allow pattern matching on expressions which are not arguments, creating extra cases. For example, to avoid defining the *if* function we can pattern-match direclty on `p x` (note the syntax `with` and `... |`):

```
split' : {A : Set} -> (A -> Boolean) -> List A -> Prod (List A) (List A)
split' p nil = pair nil nil
split' p (cons x xs) with p x
... | true  = pair (cons x ys) zs where
   xs' = split p xs
   ys  = proj1 xs'
   zs  = proj2 xs'
... | false = pair ys (cons x zs) where
   xs' = split p xs
```

```
    ys  = proj1 xs'
    zs  = proj2 xs'
```

Finally, to avoid using projections we can pattern-match on the recursive call as well, which gives the succinct and elegant

```
split' : {A : Set} -> (A -> Boolean) -> List A -> Prod (List A) (List A)
split' p nil = pair nil nil
split' p (cons x xs) with p x | split p xs
... | true  | pair ys zs = pair (cons x ys) zs
... | false | pair ys zs = pair ys (cons x zs)
```