Base case (m=Z)

```
   nat2int (multnat Z n)
=  nat2int Z   (* defn of multnat *)
=  0 (* defn of nat2int *)
```

Induction Hypothesis: Assume that for m=*k*, *forall* n:nat,
nat2int (multnat *k* n) = (nat2int *k*) * (nat2int n)

Induction Step (m = S *k*)

```
     nat2int (multnat (S k) n)
=    nat2int (addnat n (multnat k n) // defn of multnat
=    (nat2int n) + (nat2int (multnat k n)) // correctness of addnat
=    (nat2int n) + ((nat2int k) * (nat2int n)) // IH on m=k
=    (1 + (nat2int k)) * (nat2int n)   // right distribution of * over + <-
=    (nat2int (S k)) * (nat2int n) // defn of nat2int <-
```

Exercise: State and prove that Z is a (i) left annihilator for multnat; (ii) right annihilator for multnat.

Exercise: State and prove that (S Z) is (i) a left identity for multnat; (ii) a right identity for multnat.

Exercise: State and prove that multnat is commutative.

Exercise: State and prove that multnat is associative.

Exercise: State and prove that multnat distributes left and right over addnat.

If we make a convenient assumption that $0^0 = 1$ rather than undefined, we can define exponentiation as a primitive recursive function

```
let rec expnat m n = match n with
    Z -> (S Z)     (* m^0 = 1 *)
  | S x -> multnat m (expnat m x)     (* m^(1+x) = m * (m^x) *)
;;

expnat zero one;;
expnat zero zero;;
expnat zero three;;
expnat three zero;;
expnat two three;;
expnat two one;;
expnat three two;;
```

Exercise: State and prove the correctness of expnat.

Exercise: State and prove that `(S Z)` is the right identity for `expnat`.

Exercise: Prove that `forall m: nat, forall n: nat, forall x: nat,`
`expnat x (addnat m n) = multnat (expnat x m) (expnat x n)`

Lists

OCaml supports the definition of a generic type constructions such as lists over any type. That is, for any type, we have a uniform way of building lists with elements of that type. Note however, that all elements of a given list must have the *same* type, that is one cannot have a mixed list with say integers and booleans.

A lot of reasoning about lists does not concern itself with the type of the list elements. This kind of genericity is called *"Parametric Polymorphism"*.

Lists are a built-in polymorphic type in OCaml. However, one can imagine that someone must have made a parametric type definition of the form

```
type 'a list = Nil | Cons of 'a * ('a list)
```

for two constructors traditionally called `Nil` and `Cons`. The polymorphic type is `'a list`, where `'a` stands for *any* type. *Type variables* are written by putting a quote mark before an identifier beginning with a lower-case letter. It is customary to read the "quote-a" as "alpha", "quote-b" as "beta", etc. to highlight that these are type variables.

[Mathematically, lists are the least fixed-point solution to a recursive type equation $L_\alpha = \mathbf{1}_{Nil} +_{Cons} (\alpha \times L_\alpha)$ for any type α.

OCaml interpreters come with a built-in List module which has predefined values and functions over lists. To use a values and functions in a module we refer to them using a dot notation, e.g. `List.append`. However, by "opening" the module so we can use its definitions freely, without qualifying them each time with the module name..

```
open List;;
```

There is a more intuitive way of writing the `Nil` constructor.

```
[ ];;   (* The Nil constructor *)
```

The `Cons` constructor can be thought of taking a pair — an element from a type α and a list of type α list. This constructor is asymmetric in the two arguments, one is an element of type `'a` and the other is a list of elements of that type, an `'a list`. So it is not like a monoid operator. Note also that we can only "`Cons`" an element to the *front* of a list, and this is a constant-time operation.

```
1 :: [ ];;

1 :: (2 :: []);;
```

Two lists of the same type can be concatenated to return a single list. The original lists are unchanged; a new list is created, and the elements of the first list appear in order before those of the second list.

```
append;;

(* imagine someone had defined a recursive function

 let rec append l1 l2 = match l1 with
        [ ] -> l2
     | x::xs -> x :: (append xs l2)
   ;;
*)

append [ ] [1;2;3];;
append [1;2;3] [ ];;
```

If one worked with the above imagined definition of append, one could do the following (we imagine the implementor of the List library did so).

Exercise: State and prove that `[ ]` is the left and right identity element for `append`

```
append [1] (append [2] [3]);;
append (append [1] [2]) [3];;
```

Exercise: State and prove that `append` is associative.

Exercise: Prove that appending two lists yields a list whose length is the sum of the lengths of the input lists:

```
forall l1: 'a list, forall l2: 'a list,
     length (append l1 l2) = (length l1) + (length l2)
```

It is common to use the operator `_@_` as an infix version of `append`.

Note that `_ :: _` ("cons") is a constant time operation, whereas `append` involves a function call. So never write `[1] @ [2;3;4]` but instead write `1 :: [2; 3; 4]`. However, since one can only prepend (cons) an *element* at the front of a list, if we have to place an element at the end of a list, we may have to use append.

Consider the code to reverse a (polymorphic) list (There already is a `List.rev` function).

```
List.rev [3; 2; 1];;


let rec rev s = match s with
        [ ] -> [ ]
   |  x::xs ->   (rev xs) @ [x]
;;

rev [1;2 3];;
```

Cons would not work, since the element is being placed at the *end* of the list.