Function: `null`

`null xs` returns `True` if the list is empty

Type Signature

```
null  ::  [a] -> Bool
```

Empty List

```
null []  =  True
```

Non-Empty List

```
null (_:_)  =  False
```

Function: `++`

`xs ++ ys` joins lists `xs` and `ys` together.

Type Signature

```
(++)   :: [a] -> [a] -> [a]
```

Empty List

```
[] ++ ys  =  ys
```

Non-Empty List

```
(x:xs) ++ ys  =  x : (xs ++ ys)
```

Evaluating: `++`

```
   (1:2:3:[]) ++ (4:5:[])
= -- Non-Empty List, x -> 1, xs -> 2:3:[]
   1 : ( (2:3:[]) ++ (4:5:[]) )
= -- Non-Empty List, x -> 2, xs -> 3:[]
   1 : ( 2: ( (3:[]) ++ (4:5:[]) ) )
= -- Non-Empty List, x -> 3, xs -> []
   1 : ( 2: (3: ([] ++ (4:5:[]) ) ) )
= -- Empty List, ys -> 4:5:[]
   1 : ( 2: (3: (4 : 5 :[])))
```

Note that the time taken is proportional to the length of the first list, and independent of the size of the second.

Function: `reverse` (slow)

`reverse xs`, reverses the list `xs`

Type Signature

```
reverse  ::  [a] -> [a]
```

Empty List

```
reverse []  =  []
```

Non-Empty List

```
reverse (x:xs)  =  reverse xs ++ [x]
```

Evaluating: `reverse`

```
   reverse (1:2:3:[])
= -- Non-Empty List, x -> 1, xs -> 2:3:[]
   reverse (2:3:[]) ++ [1]
= -- Non-Empty List, x -> 2, xs -> 3:[]
   (reverse (3:[]) ++ [2]) ++ [1]
= -- Non-Empty List, x -> 3, xs -> []
   ((reverse [] ++ [3]) ++ [2]) ++ [1]
= -- Empty List,
   (([] ++ [3]) ++ [2]) ++ [1]
= -- after many concatenations
   3:2:1:[]
```

This is a bad way to do reverse (why?)

Evaluating: `reverse`, again

```
   reverse (1:2:3:[])
= -- ???
   rev [] (1:2:3:[])
= -- Non-Empty List, sx -> [], x -> 1, xs -> 2:3:[]
   rev (1:[]) (2:3:[])
= -- Non-Empty List, sx -> 1:[], x -> 2, xs -> 3:[]
   rev (2:1:[]) (3:[])
= -- Non-Empty List, sx -> 2:1:[], x -> 3, xs -> []
   rev (3:2:1:[]) []
= -- Empty List, sx -> 3:2:1:[]
   3:2:1:[]
```

Much faster (why?)

Function: `reverse` (fast)

`reverse xs`, reverses the list `xs`

Type Signature

```
reverse  ::  [a] -> [a]
```

Use Helper Function (???)

```
reverse xs = rev [] xs
```

Helper: Non-Empty List

```
rev sx (x:xs) = rev (x:sx) xs
```

Helper: Empty List

```
rev sx [] = sx
```

Function: `reverse` (Prelude Version, [*H2010* 9.1])

`reverse xs`, reverses the list `xs`

Type Signature

```
reverse  ::  [a] -> [a]
```

!!!! ???

```
reverse  =  foldl (flip (:)) []
```

The Prelude doesn't always give the most obvious definition of a function's behaviour !

Function: (!!)

`(!!) xs n`, or `xs !! n` selects the `n`th element of list `xs`, provided it is long enough. Indices start at 0.

Fixity and Type Signature

```
infixl 9 !!
(!!)  ::  [a] -> Int -> a
```

Negative Index

```
xs !! n | n < 0
   =  error "Prelude.!!: negative index"
```

Empty List

```
[] !! _  =  error "Prelude.!!: index too large"
```

Zero Index

```
(x:_) !! 0  =  x
```

Non-Zero Index

```
(_:xs) !! n  =  xs !! (n-1)
```

Higher Order Functions

What is the difference between these two functions?

```
add x y = x + y
add2 (x, y) = x + y
```

We can see it in the types; `add` takes one argument at a time, returning a function that looks for the next argument.
This concept is known as "Currying" after the logician Haskell B. Curry.

```
add :: Integer -> (Integer -> Integer)
add2 :: (Integer, Integer) -> Integer
```

The function type arrow associates to the right, so `a -> a -> a` is the same as `a -> (a -> a)`.

In Haskell functions are *first class citizens*. In other words, they occupy the same status in the language as values: you can pass them as arguments, make them part of data structures, compute them as the result of functions...

```
add3 :: (Integer -> (Integer -> Integer))
add3 = add


> add3 1 2
3

(add3) 1 2
  ==> add 1 2
  ==> 1 + 2
```

Notice that there are no parameters in the definition of `add3`.

A function with multiple arguments can be viewed as a function of one argument, which computes a new function.

```
add 3 4
  ==> (add 3) 4
  ==> ((+) 3) 4
```

The first place you might encounter this is the notion of *partial application*:

```
increment :: Integer -> Integer
increment = add 1
```

If the type of `add` is `Integer -> Integer -> Integer`, and the type of `add 1 2` is `Integer`, then the type of `add 1` is?
It is `Integer -> Integer`

Some more examples of partial application:
An infix operator can be partially applied by taking a *section*:

```
increment = (1 +) -- or (+ 1)

addnewline = (++"\n")

double :: Integer -> Integer
double = (*2)
```

```
> [ double x | x <- [1..10] ]
[2,4,6,8,10,12,14,16,18,20]
```

Functions can be taken as parameters as well.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)

addtwo = twice increment
```

Here we see functions being defined as functions of other functions!

Function Composition (I)

In fact, we can define function composition using this technique:

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)

twice2 f = f `compose` f
```

We can use an infix operator definition for compose, even though it takes three results, rather than two.

```
(f ! g) x = f (g x)
twice3 f = f!f
```

We just bracket the infix application and apply that to the last (`x`) argument.

Function Composition (II) [*H2010* 9]

Function composition is in fact part of the Haskell Prelude:

```
(.) :: (b -> c) -> (a -> b) -> a -> c

(f . g) x = f (g x)
```

We can define functions without naming their inputs, using composition (and other HOFs)

```
second :: [a] -> a
second = head . tail
```

```
> second [1,2,3]
2
```