Lists [*H2010* 3.7,3.10]

- Fundamentally lists are built from "nil" (`[]`) and "cons" (`:`)
- We use square brackets to provide syntactical sugar in a variety of ways
  - Enumeration:
    `[a,b,c,d]` for `a:b:c:d:[]`
  - Ranges:
    `[4..9]` for `[4,5,6,7,8,9]`
    also `[4,7..20]` for `[4,7,10,13,16,19]`
  - Comprehension:
    `[ x*x | x <- [1..10], even x]` for `[4,16,36,64,100]`
    Comprehensions are more complex than this (see later, or [*H2010* 3.11])
- Strings are a special notation of lists of characters
  `"Hello"` for `['H','e','l','l','o']`

Function: `head`

`head xs` returns the first element of `xs`, if non-empty

Type Signature

```
head  ::  [a] -> a
```

Non-Empty List

```
head (x:_)  =  x
```

Empty List

```
head []  =  error "Prelude.head: empty list"
```

Function: `tail`

`tail xs`, for non-empty `xs` returns it with first element removed

Type Signature

```
tail  ::  [a] -> [a]
```

Non-Empty List

```
tail (_:xs)  =  xs
```

Empty List

```
tail []  =  error "Prelude.tail: empty list"
```

`tail [] /= []` — Why Not?

Why don't we define `tail [] = []`? The typing allows it.

If we have a version of `head` specialised for lists of `Int`, i.e., `headInt :: [Int] -> Int`, why might we not choose to define `headInt [] = 0` ?

A key design principle behind Haskell libraries and programs is to have programs (functions!) that obey nice obvious laws:

```
xs = head xs ++ tail xs
sum (xs ++ ys) = sum xs + sum ys
product (xs ++ ys) = product xs * product ys
```

Imagine if we defined:

```
product xs = headInt xs * product (tail xs)
```

Would this satisfy the law above for `product`?

Function: `last`

`last xs` returns the last element of `xs`, if non-empty

Type Signature

```
last  ::  [a] -> a
```

Singleton List

```
last [x]  =  x
```

Non-Empty List

```
last (_:xs)  =  last xs
```

Empty List

```
last []  =  error "Prelude.last: empty list"
```

Function: `init`

`init xs`, for non-empty `xs` returns it with last element removed

Type Signature

```
init  ::  [a] -> [a]
```

Singleton List

```
init [x]  =  []
```

Non-Empty List

```
init (x:xs)  =  x : init xs
```

Empty List

```
init []  =  error "Prelude.init: empty list"
```

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 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)
```

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?)

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
```

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` (Prelude Version)

`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 !

List Defs

```
infixl 9  !!
infixr 5  ++
infix  4  elem, notElem

map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x : map f xs

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

filter :: (a -> Bool) -> [a] -> [a]
filter p []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs

concat :: [[a]] -> [a]
concat xss = foldr (++) [] xss
```

List Defs

```
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f

repeat           :: a -> [a]
repeat x         =  xs where xs = x:xs

replicate        :: Int -> a -> [a]
replicate n x    =  take n (repeat x)

take                   :: Int -> [a] -> [a]
take n _      | n <= 0 =  []
take _ []              =  []
take n (x:xs)          =  x : take (n-1) xs

drop                   :: Int -> [a] -> [a]
drop n xs     | n <= 0 =  xs
drop _ []              =  []
drop n (_:xs)          =  drop (n-1) xs
```