Script generated by TTT

Title: Nipkow: Info2 (26.11.2013)

Date: Tue Nov 26 15:31:06 CET 2013

Duration: 88:21 min

Pages: 118

6.7 More library functions

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

6.7 More library functions

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

f . g = \x -> f (g x)

Example

```
head2  =  head . tail
```

6.8 Case study: Counting words

Input: A string, e.g. `"never say never again"`

Output: A string listing the words in alphabetical order, together with their frequency,
e.g. `"again: 1\nnever: 2\nsay: 1\n"`

6.8 Case study: Counting words

Input: A string, e.g. `"never say never again"`

Output: A string listing the words in alphabetical order, together with their frequency,
e.g. `"again: 1\nnever: 2\nsay: 1\n"`

Function `putStr` yields

```
again: 1
never: 2
say: 1
```


6.8 Case study: Counting words

Input: A string, e.g. `"never say never again"`

Output: A string listing the words in alphabetical order, together with their frequency,
e.g. `"again: 1\nnever: 2\nsay: 1\n"`

Function `putStr` yields

```
again: 1
never: 2
say: 1
```


Design principle:

> *Solve problem in a sequence of small steps*
> *transforming the input gradually into the output*

6.8 Case study: Counting words

Input: A string, e.g. `"never say never again"`

Output: A string listing the words in alphabetical order, together with their frequency,
e.g. `"again: 1\nnever: 2\nsay: 1\n"`

Function `putStr` yields

```
again: 1
never: 2
say: 1
```

Design principle:

> *Solve problem in a sequence of small steps*
> *transforming the input gradually into the output*

Unix pipes!

Step 1: Break input into words

```
"never say never again"
          ↓
["never", "say", "never", "again"]
```

Step 1: Break input into words

```
"never say never again"
```

function $\downarrow$ `words`

```
["never", "say", "never", "again"]
```

Predefined in Prelude

Step 2: Sort words

```
["never", "say", "never", "again"]
```

$\downarrow$

```
["again", "never", "never", "say"]
```

Step 3: Group equal words together

```
["again", "never", "never", "say"]
```

$\downarrow$

```
[["again"], ["never", "never"], ["say"]]
```

Step 3: Group equal words together

```
["again", "never", "never", "say"]
```

function $\downarrow$ `group`

```
[["again"], ["never", "never"], ["say"]]
```

Predefined in `Data.List`

Step 4: Count each group

```
[["again"], ["never", "never"], ["say"]]
        ↓
[("again", 1), ("never", 2), ("say", 1)]
```

Step 4: Count each group

```
[["again"], ["never", "never"], ["say"]]
        ↓ map (\ws -> (head ws, length ws))
[("again", 1), ("never", 2), ("say", 1)]
```

Step 5: Format each group

```
[("again", 1), ("never", 2), ("say", 1)]
        ↓
["again: 1", "never: 2", "say: 1"]
```

Step 5: Format each group

```
[("again", 1), ("never", 2), ("say", 1)]
        ↓ map (\(w,n) -> (w ++ ": " ++ show n)
["again: 1", "never: 2", "say: 1"]
```

Step 6: Combine the lines

```
["again: 1", "never: 2", "say: 1"]
        ↓
"again: 1\nnever: 2\nsay: 1\n"
```

Step 6: Combine the lines

```
["again: 1", "never: 2", "say: 1"]
function ↓ unlines
"again: 1\nnever: 2\nsay: 1\n"
```

Predefined in Prelude

The solution

```
countWords :: String -> String
countWords  =
  unlines
  . map (\(w,n) -> w ++ ": " ++ show n)
  . map (\ws -> (head ws, length ws))
  . group
  . sort
  . words
```

The solution

```
countWords :: String -> String
countWords  =
  unlines
  . map (\(w,n) -> w ++ ": " ++ show n)
  . map (\ws -> (head ws, length ws))
  . group
  . sort
  . words
```

Merging maps

Can we merge two consecutive maps?

```
map f . map g =
```

Merging maps

Can we merge two consecutive maps?

```
map f . map g = map (f.g)
```

The optimized solution

```
countWords :: String -> String
countWords  =
  unlines
  . map (\ws -> head ws ++ ": " ++ show(length ws))
  . group
  . sort
  . words
```

Proving `map f . map g = map (f.g)`

First we prove (why?)

```
map f (map g xs) = map (f.g) xs
```

Proving `map f . map g = map (f.g)`

First we prove (why?)

```
map f (map g xs) = map (f.g) xs
```

by induction on xs:

- Base case:
```
map f (map g []) = []
map (f.g) [] = []
```
- Induction step:
```
map f (map g (x:xs))
= f (g x) : map f (map g xs)
= f (g x) : map (f.g) xs      -- by IH
map (f.g) (x:xs)
= f (g x) : map (f.g) xs
```


Proving `map f . map g = map (f.g)`

First we prove (why?)

```
map f (map g xs) = map (f.g) xs
```

by induction on xs:

- Base case:
```
map f (map g []) = []
map (f.g) [] = []
```
- Induction step:
```
map f (map g (x:xs))
= f (g x) : map f (map g xs)
= f (g x) : map (f.g) xs      -- by IH
map (f.g) (x:xs)
= f (g x) : map (f.g) xs
```


⟹ `(map f . map g) xs = map f (map g xs) = map (f.g) xs`

Proving `map f . map g = map (f.g)`

First we prove (why?)

```
map f (map g xs) = map (f.g) xs
```

by induction on xs:

- Base case:
```
map f (map g []) = []
map (f.g) [] = []
```
- Induction step:
```
map f (map g (x:xs))
= f (g x) : map f (map g xs)
= f (g x) : map (f.g) xs      -- by IH
map (f.g) (x:xs)
= f (g x) : map (f.g) xs
```

⟹ `(map f . map g) xs = map f (map g xs) = map (f.g) xs`

⟹ `(map f . map g) = map (f.g)` by extensionality

7. Type Classes

Remember: type classes enable overloading

Remember: type classes enable overloading

Example

```
elem ::
elem x  =  any (== x)
```

Remember: type classes enable overloading

Example

```
elem ::           a -> [a] -> Bool
elem x  =  any (== x)
```

Remember: type classes enable overloading

Example

```
elem ::  Eq a => a -> [a] -> Bool
elem x  =  any (== x)
```

where Eq is the class of all types with ==

In general:

Type classes are collections of types
that implement some fixed set of functions

In general:

Type classes are collections of types
that implement some fixed set of functions

Haskell type classes are analogous to Java interfaces:
a set of function names with their types

In general:

Type classes are collections of types
that implement some fixed set of functions

Haskell type classes are analogous to Java interfaces:
a set of function names with their types

Example

```
class Eq a where
  (==) :: a -> a -> Bool
```

In general:

Type classes are collections of types
that implement some fixed set of functions

Haskell type classes are analogous to Java interfaces:
a set of function names with their types

Example

```
class Eq a where
  (==) :: a -> a -> Bool
```

Note: the type of (==) outside the class context is
`Eq a => a -> a -> Bool`

The general form of a class declaration:

```
class C a where
  f1 :: T1
  ...
  fn :: Tn
```

The general form of a class declaration:

```
class C a where
  f1 :: T1
  ...
  fn :: Tn
```

where the `Ti` may involve the type variable `a`

Instance

A type T is an instance of a class C if T supports all the functions of C.

Instance

A type T is an instance of a class C if T supports all the functions of C. Then we write C T.

Example

Type `Int` is an instance of class `Eq`, i.e., `Eq Int`

Instance

A type T is an instance of a class C
if T supports all the functions of C.
Then we write $C\ T$.

Example
Type `Int` is an instance of class `Eq`, i.e., `Eq Int`

Therefore `elem :: Int -> [Int] -> Bool`

Instance

A type T is an instance of a class C
if T supports all the functions of C.

In general:

Type classes are collections of types
that implement some fixed set of functions

Instance

A type T is an instance of a class C
if T supports all the functions of C.
Then we write $C\ T$.

Example
Type `Int` is an instance of class `Eq`, i.e., `Eq Int`

Therefore `elem :: Int -> [Int] -> Bool`

Warning Terminology clash:
Type T_1 is an *instance* of type T_2

Instance

A type T is an instance of a class C
if T supports all the functions of C.
Then we write `C T`.

Example

Type `Int` is an instance of class `Eq`, i.e., `Eq Int`

Therefore `elem :: Int -> [Int] -> Bool`

Warning Terminology clash:
Type T_1 is an *instance* of type T_2
if T_1 is the result of replacing type variables in T_2.
For example `(Bool,Int)` is an instance of `(a,b)`.

instance

The `instance` statement makes a type an instance of a class.

instance

The `instance` statement makes a type an instance of a class.

Example

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False
```


instance

The `instance` statement makes a type an instance of a class.

Example

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False
```

Instances can be constrained:

Example

```
instance Eq a => Eq [a] where
  []     == []      =  True
  (x:xs) == (y:ys)  =  x == y && xs == ys
  _      == _       =  False
```

Instances can be constrained:

Example

```
instance Eq a => Eq [a] where
  []     == []      =  True
  (x:xs) == (y:ys)  =  x == y && xs == ys
  _      == _       =  False
```

Possibly with multiple constraints:

Example

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2)  =  x1 == x2 && y1 == y2
```

The general form of the instance statement:

`instance (`*context*`) =>` *C* *T* `where`
  definitions

Instances can be constrained:

Example

```
instance Eq a => Eq [a] where
  []     == []      =  True
  (x:xs) == (y:ys)  =  x == y && xs == ys
  _      == _       =  False
```

Possibly with multiple constraints:

Example

```
instance (Eq a, Eq b) => Eq (a,b) where
  (x1,y1) == (x2,y2)  =  x1 == x2 && y1 == y2
```

The general form of the `instance` statement:

instance (*context*) => *C* *T* where
  definitions

The general form of the `instance` statement:

instance (*context*) => *C* *T* where
  definitions

T	is a type
context	is a list of assumptions C_i T_i
definitions	are definitions of the functions of class C

Subclasses

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Subclasses

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Class `Ord` inherits all the operations of class `Eq`

Subclasses

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Class `Ord` inherits all the operations of class `Eq`

Because `Bool` is already an instance of `Eq`,
we can now make it an instance of `Ord`:

Subclasses

Example

```
class Eq a => Ord a where
  (<=), (<) :: a -> a -> Bool
```

Class `Ord` inherits all the operations of class `Eq`

Because `Bool` is already an instance of `Eq`,
we can now make it an instance of `Ord`:

```
instance Ord Bool where
  b1 <= b2  =  not b1 || b2
  b1 < b2  =  b1 <= b2 && not(b1 == b2)
```

From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```

From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
```


```
  -- default definition:
  x /= y  =  not(x==y)
```

From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y  =  not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
```

From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y  =  not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
  -- default definitions:
  x < y   =  x <= y && x /= y
  x > y   =  y < x
  x >= y =  y <= x
```

From the Prelude: Eq, Ord, Show

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  -- default definition:
  x /= y  =  not(x==y)

class Eq a => Ord a where
  (<=), (<), (>=), (>) :: a -> a -> Bool
  -- default definitions:
  x < y   =  x <= y && x /= y
  x > y   =  y < x
  x >= y =  y <= x

class Show a where
  show :: a -> String
```

8. Algebraic data Types

So far: no really new types,

So far: no really new types,
just compositions of existing types

Example: `type String = [Char]`

Now: `data` defines *new* types

So far: no really new types,
just compositions of existing types

Example: `type String = [Char]`

Now: `data` defines *new* types

Introduction by example: From enumerated types

8.1 `data` by example

Bool

From the Prelude:

```
data Bool = False | True
```

Bool

From the Prelude:

```
data Bool = False | True

not :: Bool -> Bool
not False  =  True
not True   =  False
```

Bool

From the Prelude:

```
data Bool = False | True

not :: Bool -> Bool
not False  =  True
not True   =  False

(&&) :: Bool -> Bool -> Bool
False && q  =  False
True  && q  =  q
```

Bool

From the Prelude:

```
data Bool = False | True

not :: Bool -> Bool
not False  =  True
not True   =  False

(&&) :: Bool -> Bool -> Bool
False && q  =  False
True  && q  =  q

(||) :: Bool -> Bool -> Bool
False || q  = q
True  || q  = True
```

deriving

deriving

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False
```

deriving

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False

instance Show Bool where
  show True   =  "True"
  show False  =  "False"
```

deriving

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False

instance Show Bool where
  show True   =  "True"
  show False  =  "False"
```

Better: let Haskell write the code for you:

```
data Bool = False | True
            deriving (Eq, Show)
```

deriving

```
instance Eq Bool where
  True  == True   =  True
  False == False  =  True
  _     == _      =  False

instance Show Bool where
  show True   =  "True"
  show False  =  "False"
```

Better: let Haskell write the code for you:

```
data Bool = False | True
            deriving (Eq, Show)
```

`deriving` supports many more classes: `Ord`, `Read`, . . .

Warning

Do not forget to make your data types instances of `Show`

Warning

Do not forget to make your data types instances of `Show`

Otherwise Haskell cannot even print values of your type

Warning

QuickCheck does not automatically work for data types

Warning

Do not forget to make your data types instances of `Show`

Otherwise Haskell cannot even print values of your type

Warning

QuickCheck does not automatically work for data types

You have to write your own test data generator.

Season

```haskell
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq, Show)
```

Season

```haskell
data Season = Spring | Summer | Autumn | Winter
              deriving (Eq, Show)

next :: Season -> Season
next Spring = Summer
next Summer = Autumn
next Autumn = Winter
next Winter = Spring
```

Shape

```haskell
type Radius = Float
type Width  = Float
type Height = Float
```

Shape

```haskell
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Some values of type Shape: `Circle 1.0`

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Some values of type Shape: `Circle 1.0`


```
Rect 0.9 1.1
```


Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Some values of type Shape: `Circle 1.0`

```
Rect 0.9 1.1
```


```
Circle (-2.0)
```


Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Some values of type Shape: `Circle 1.0`

```
Rect 0.9 1.1
Circle (-2.0)
```


```
area :: Shape -> Float
```

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Some values of type Shape: `Circle 1.0`
`Rect 0.9 1.1`
`Circle (-2.0)`

```
area :: Shape -> Float
area (Circle r)  =  pi * r^2
```

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Some values of type Shape: `Circle 1.0`
`Rect 0.9 1.1`
`Circle (-2.0)`

```
area :: Shape -> Float
area (Circle r)  =  pi * r^2
area (Rect w h)  =  w * h
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Shape

```
type Radius = Float
type Width  = Float
type Height = Float

data Shape = Circle Radius | Rect Width Height
             deriving (Eq, Show)
```

Some values of type Shape: `Circle 1.0`
`Rect 0.9 1.1`
`Circle (-2.0)`

```
area :: Shape -> Float
area (Circle r)  =  pi * r^2
area (Rect w h)  =  w * h
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe: `Nothing ::`

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe: `Nothing ::    Maybe a`

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe: `Nothing ::    Maybe a`
`Just True ::`

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```


```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key []  =
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```


```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key []  =  Nothing
lookup key ((x,y):xys)
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```


```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key []  =  Nothing
lookup key ((x,y):xys)
  | key == x   =
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing ::   Maybe a
Just True :: Maybe Bool
Just "?" ::  Maybe String
```


```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key []  =  Nothing
lookup key ((x,y):xys)
  | key == x   =  Just y
```

Maybe

From the Prelude:

```
data Maybe a = Nothing | Just a
               deriving (Eq, Show)
```

Some values of type Maybe:

```
Nothing    :: Maybe a
Just True  :: Maybe Bool
Just "?"   :: Maybe String
```

```
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key []   =  Nothing
lookup key ((x,y):xys)
  | key == x    =  Just y
  | otherwise   =  lookup key xys
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type Nat: `Zero`

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type Nat:

```
Zero
Suc Zero
Suc (Suc Zero)
  ⋮
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type Nat:

```
Zero
Suc Zero
Suc (Suc Zero)
⋮
```

```
add :: Nat -> Nat -> Nat
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type Nat:

```
Zero
Suc Zero
Suc (Suc Zero)
⋮
```

```
add :: Nat -> Nat -> Nat
add Zero n    =  n
add (Suc m) n  =
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type Nat:

```
Zero
Suc Zero
Suc (Suc Zero)
⋮
```

```
add :: Nat -> Nat -> Nat
add Zero n    =  n
add (Suc m) n  =  Suc (add m n)
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type Nat:

```
Zero
Suc Zero
Suc (Suc Zero)
⋮
```

```
add :: Nat -> Nat -> Nat
add Zero n    =  n
add (Suc m) n  =  Suc (add m n)

mul :: Nat -> Nat -> Nat
mul Zero n    =  Zero
mul (Suc m) n  =  add n (mul m n)
```

Lists

From the Prelude:

```
data [a] = [] | (:) a [a]
```

Nat

Natural numbers:

```
data Nat = Zero | Suc Nat
           deriving (Eq, Show)
```

Some values of type Nat:

```
Zero
Suc Zero
Suc (Suc Zero)
⋮
```

```
add :: Nat -> Nat -> Nat
add Zero n   =  n
add (Suc m) n  =  Suc (add m n)

mul :: Nat -> Nat -> Nat
mul Zero n   =  Zero
mul (Suc m) n  =  add n (mul m n)
```

Lists

From the Prelude:

```
data [a] = [] | (:) a [a]
```


Lists

From the Prelude:

```
data [a] = [] | (:) a [a]
           deriving Eq
```

The result of deriving Eq:

```
instance Eq a => Eq [a] where
  []     == []     =  True
  (x:xs) == (y:ys) =  x == y && xs == ys
  _      == _      =  False
```

Lists

From the Prelude:

```
data [a] = [] | (:) a [a]
           deriving Eq
```

The result of deriving Eq:

```
instance Eq a => Eq [a] where
  []     == []      =  True
  (x:xs) == (y:ys)  =  x == y && xs == ys
  _      == _       =  False
```

Defined explicitly:

```
instance Show a => Show [a] where
  show xs  =  "[" ++ concat cs ++ "]"
```

Lists

From the Prelude:

```
data [a] = [] | (:) a [a]
           deriving Eq
```

The result of deriving Eq:

```
instance Eq a => Eq [a] where
  []     == []      =  True
  (x:xs) == (y:ys)  =  x == y && xs == ys
  _      == _       =  False
```

Defined explicitly:

```
instance Show a => Show [a] where
  show xs  =  "[" ++ concat cs ++ "]"
    where cs = Data.List.intersperse ", "  (map show xs)
```


Lists

From the Prelude:

```
data [a] = [] | (:) a [a]
           deriving Eq
```

The result of deriving Eq:

```
instance Eq a => Eq [a] where
  []     == []      =  True
  (x:xs) == (y:ys)  =  x == y && xs == ys
  _      == _       =  False
```

Defined explicitly:

```
instance Show a => Show [a] where
  show xs  =  "[" ++ concat cs ++ "]"
    where cs = Data.List.intersperse ", "  (map show xs)
```