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\never: 2\nsay: 1"

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*

---

### 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 \( \rightarrow \) \text{words} \rightarrow ["never", "say", "never", "again"]

Predefined in Prelude

Step 2: Sort words

["never", "say", "never", "again"] \rightarrow ["again", "never", "never", "say"]

Step 3: Group equal words together

["again", "never", "never", "say"] \rightarrow [["again"], ["never", "never"], ["say"]]

Predefined in Data.List
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]
```

```
map (\(w, n) -> (w ++ ": ", n))
```

```
["again: 1", "never: 2", "say: 1"],
```

```
map (\(w) -> (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"

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
Merging maps

Can we merge two consecutive maps?

map f . map g =

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 $\text{map } f \cdot \text{map } g = \text{map } (f \cdot g)$

First we prove (why?)

$$\text{map } f \ (\text{map } g \ \text{xs}) = \text{map } (f \cdot g) \ \text{xs}$$

by induction on $\text{xs}$:

- Base case:
  $$\text{map } f \ (\text{map } g \ [] ) = []$$
  $$\text{map } (f \cdot g) \ [] = []$$

- Induction step:
  $$\text{map } f \ (\text{map } g \ (x:xs))$$
  $$= f \ (g \ x) : \text{map } f \ (\text{map } g \ \text{xs})$$
  $$= f \ (g \ x) : \text{map } (f \cdot g) \ \text{xs} \quad \text{-- by IH}$$
  $$\text{map } (f \cdot g) \ (x:xs)$$
  $$= f \ (g \ x) : \text{map } (f \cdot g) \ \text{xs}$$

$$\implies (\text{map } f \cdot \text{map } g) \ \text{xs} = \text{map } f \ (\text{map } g \ \text{xs}) = \text{map } (f \cdot g) \ \text{xs}$$

---

7. Type Classes
Remember: type classes enable overloading

Example

\[
\text{elem :: } a \rightarrow [a] \rightarrow \text{Bool}
\]

\[
\text{elem } x = \text{ any } (== x)
\]
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

where the $T_i$ may involve the type variable $a$

Instance

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

Example
Type \textit{Int} is an instance of class \textit{Eq}, i.e., $\textit{Eq} \ ~ \textit{Int}$
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 $\text{elem} :: Int \to [\text{Int}] \to \text{Bool}$

In general:
Type classes are collections of types that implement some fixed set of functions.

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 $\text{elem} :: Int \to [\text{Int}] \to \text{Bool}$

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

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 $\text{Int}$ is an instance of class $\text{Eq}$, i.e., $\text{Eq \ Int}$
Therefore $\text{elem :: Int} \rightarrow [\text{Int}] \rightarrow \text{Bool}$

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 $(\text{Bool,Int})$ is an instance of $(a,b)$.

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

Example
instance $\text{Eq \ Bool}$ where
  True $\equiv$ True $= \text{True}$
  False $\equiv$ False $= \text{True}$
  _ $\equiv$ _ $= \text{False}$

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

Example
instance $\text{Eq \ Bool}$ where
  True $\equiv$ True $= \text{True}$
  False $\equiv$ False $= \text{True}$
  _ $\equiv$ _ $= \text{False}$
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
```
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**

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

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

class Eq a => Ord a where

\( (\leq), (\prec) :: \text{a} \rightarrow \text{a} \rightarrow \text{Bool} \)

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`:

```haskell
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

\( (==), (/\neq) :: \text{a} \rightarrow \text{a} \rightarrow \text{Bool} \)

```haskell
-- default definition:
x /= y = not(x==y)
```
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, just compositions of existing types.

Example: type String = [Char]

Now: `data` defines new types.

---

8.1 data by example

Introduction by example: From enumerated types.
From the Prelude:

```haskell
data Bool = False | True

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

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

From the Prelude:

```haskell
data Bool = False | True

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

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

Otherwise Haskell cannot even print values of your type

Warning

QuickCheck does not automatically work for data types

Warning

QuickCheck does not automatically work for data types

Warning

You have to write your own test data generator.
**Season**

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

```
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**

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

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

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
From the Prelude:

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

Some values of type `Maybe`: Nothing :: Maybe a
From the Prelude:

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

Some values of type Maybe:
- `Nothing :: Maybe a`
- `Just True :: Maybe Bool`
- `Just "?" :: Maybe String`

```haskell
lookup :: Eq a => a -> [(a,b)] -> Maybe b
```
From the Prelude:

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

Some values of type Maybe:
    Nothing :: Maybe a
    Just True :: Maybe Bool
    Just "?" :: Maybe String
```

```haskell
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup key [] =
lookup key ((x,y):ys)
    | key == x    = Just y
```
Maybe

From the Prelude:

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

Some values of type Maybe:  Nothing :: Maybe a
Just True :: Maybe Bool
Just "?" :: Maybe String

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

Nat

Natural numbers:

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

Some values of type Nat:  Zero
Suc Zero
Suc (Suc Zero)
Natural numbers:

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

Some values of type Nat: Zero
  Suc Zero
  Suc (Suc Zero)
  :

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

```haskell
add Zero n = n
add (Suc m) n = Suc (add m n)
```
From the Prelude:

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

Lists

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]

The result of deriving Eq:

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

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

The result of deriving Eq:

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

Defined explicitly:

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