Exercise: Positional List Splitting

- $take :: Int \to [a] \to [a]$

  take, applied to a $k :: Int$ and a list xs, returns the longest prefix of xs of elements that has no more than k elements.

- $drop :: Int \to [a] \to [a]$

  $drop\ k\ xs$ returns the suffix remaining after $take\ k\ xs$.

Laws:

- $take\ k\ xs +\!\!+\ drop\ k\ xs = xs$
- $length\ (take\ k\ xs) \leq k$

Note: $splitAt\ \ k\ xs = (take\ k\ xs,\ drop\ k\ xs)$

Guarded Definitions

$$
\begin{array}{l}
sign\ x \mid x > 0 \ = \ 1 \\
\quad\quad \mid x == 0 \ = \ 0 \\
\quad\quad \mid x < 0 \ = \ \text{-}1
\end{array}
$$

$$
\begin{array}{l}
choose :: Ord\ a \Rightarrow (a, b) \to (a, b) \to b \\
choose\ (x, v)\ (y, w) \\
\quad \mid x > y \quad = \ v \\
\quad \mid x < y \quad = \ w \\
\quad \mid otherwise \ = \ error\ \text{"I cannot decide!"}
\end{array}
$$

If no guard succeeds, the next pattern is tried:

$$
\begin{array}{l}
take\ 0\ \_ = [\,] \\
take\ k\ \_ \mid k < 0 = error\ \text{"take: negative argument"} \\
take\ k\ [\,] = [\,] \\
take\ k\ (x : xs) = \ x : take\ (k - 1)\ xs
\end{array}
$$

$$
\begin{array}{l}
take\ 2\ [5, 6, 7] \qquad\qquad = take\ 2\ (5 : 6 : 7 : [\,]) \\
= 5 : take\ (2 - 1)\ (6 : 7 : [\,]) \\
= 5 : take\ 1\ (6 : 7 : [\,]) \\
= 5 : 6 : take\ (1 - 1)\ (7 : [\,]) \\
= 5 : 6 : take\ 0\ (7 : [\,]) \\
= 5 : 6 : [\,] \qquad\qquad = [5, 6]
\end{array}
$$

`where` Clauses

If an auxiliary definition is used only locally, it should be inside a **local definition**, e.g.:

$$
\begin{array}{l}
commaWords :: [String] \to String \\
commaWords\ [\,] = [\,] \\
commaWords\ (x : xs) = x +\!\!+\ commaWordsAux\ xs \\
\quad \textbf{where} \\
\qquad commaWordsAux\ [\,] = [\,] \\
\qquad commaWordsAux\ xs = \text{","} : commaWords\ xs
\end{array}
$$

`where` clauses are visible **only** within their enclosing clause, here “$commaWords\ (x : xs) = \ldots$”

`where` clauses are visible within all guards:

```
f x y | y >  z  =  …
      | y == z  =  …
      | y <  z  =  …
  where z = x * x
```


`let` Expressions

Local definitions can also be part of expressions:

```
f k n = let m = k `mod` n
        in if m == 0
           then n
           else f n m

h x y =  let x2 = x * x
             y2 = y * y
         in sqrt (x2 + y2)
```

Definitions can use **pattern bindings**:

```
g k n = let (d,m) = divMod k n
        in if d == 0
           then [m]
           else g d n ++ [m]
```

Guards, `let` and `where` bindings, and `case` cases all are **layout sensitive**!

`let` or `where`?

- `let` *bindings* `in` *expression*

  is an **expression**
- *fname patterns guardedRHSs* `where` *bindings*

  is a clause that is part of a **definition**
- (`where` clauses can also modify `case` cases)

Frequently, the choice between `let` and `where` is a matter of *style*:

- `where` clauses result in a top-down presentation
- `let` expressions lend themselves also to bottom-up presentations

`case` Expressions

```
sign x = case compare x 0 of
           GT ->  1
           EQ ->  0
           LT -> -1
```

The prelude datatype *Ordering* has three elements and is used mostly as result type of the prelude function *compare*:

data *Ordering* = *LT* | *EQ* | *GT*

$compare :: Ord\ a \Rightarrow a \rightarrow a \rightarrow Ordering$

Another example:

choose — *choose* $(x, v)\ (y, w)$ = **case** *compare x y* **of**
    $GT \rightarrow v$
    $LT \rightarrow w$
    $EQ \rightarrow$ *error* "I cannot decide!"

`if` … `then` … `else` … and `case` Expressions

The type *Bool* can be considered as a two-element enumeration type:

data *Bool* = **False** | **True**

Conditional expressions are “syntactic sugar” for **case** expressions over *Bool*:

if *condition* **then** *expr1* **else** *expr2*	$\equiv$	**case** *condition* **of** **True** $\rightarrow$ *expr1* **False** $\rightarrow$ *expr2*

Two ways of defining functions:

Pattern Matching	`case`
not **True** = **False** *not* **False** = **True**	*not b* = **case** *b* **of** **True** $\rightarrow$ **False** **False** $\rightarrow$ **True**

`case` Expressions are “Anonymous” Pattern Matching

commaWords :: [*String*] $\rightarrow$ *String*
commaWords [ ] = [ ]
commaWords (*x* : *xs*) = *x* ++ **case** *xs* **of**
    [ ] $\rightarrow$ [ ]
    _ $\rightarrow$ "," : *commaWords xs*

Every use of a `case` expression can be transformed into the use of an auxiliary function defined by pattern matching:

commaWords :: [*String*] $\rightarrow$ *String*
commaWords [ ] = [ ]
commaWords (*x* : *xs*) = *x* ++ *commaWordsAux xs*

commaWordsAux [ ] = [ ]
commaWordsAux xs = "," : *commaWords xs*

Some Prelude Functions — Elementary List Access

```
head                :: [a] -> a
head (x:_)           = x

last                :: [a] -> a
last [x]             = x
last (_:xs)          = last xs

tail                :: [a] -> [a]
tail (_:xs)          = xs

init                :: [a] -> [a]
init [x]             = []
init (x:xs)          = x : init xs

null                :: [a] -> Bool
null []              = True
null (_:_)           = False
```


Some Prelude Functions — List Indexing

```
length              :: [a] -> Int
length               = foldl' (\n _ -> n + 1) 0

(!!)                :: [b] -> Int -> b
(x:_)  !! 0          = x
(_:xs) !! n | n>0    = xs !! (n-1)
(_:_)  !! _          = error "PreludeList.!!: negative index"
[]     !! _          = error "PreludeList.!!: index too large"
```


Some Prelude Functions — Positional List Splitting

```
take                    :: Int -> [a] -> [a]
take 0 _                 = []
take _ []                = []
take n (x:xs) | n>0      = x : take (n-1) xs
take _ _                 = error "take: negative argument"

drop                    :: Int -> [a] -> [a]
drop 0 xs                = xs
drop _ []                = []
drop n (_:xs) | n>0      = drop (n-1) xs
drop _ _                 = error "drop: negative argument"

splitAt                  :: Int -> [a] -> ([a], [a])
splitAt 0 xs              = ([],xs)
splitAt _ []              = ([],[])
splitAt n (x:xs) | n>0 = (x:xs',xs'')
                 where (xs',xs'') = splitAt (n-1) xs
splitAt _ _        = error "splitAt: negative argument"
```


Some Prelude Functions — Concatenation, Iteration

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

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

iterate           :: (a -> a) -> a -> [a]
iterate f x        = x : iterate f (f x)

repeat            :: a -> [a]
repeat x           = xs  where  xs = x:xs
{- repeat x        = x : repeat x -}     -- for understanding

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

cycle             :: [a] -> [a]
cycle xs           = xs'  where  xs' = xs ++ xs'
```

Separation of Concerns: Generation and Consumption

```
replicate 3 '!'
= take 3 (repeat '!')                      -- replicate
= take 3 ('!' : repeat '!')                -- repeat
= '!' : take (3 - 1) (repeat '!')          -- take (iii)
= '!' : take 2 (repeat '!')                -- subtraction
= '!' : take 2 ('!' : repeat '!')          -- repeat
= '!' : '!' : take (2 - 1) (repeat '!')    -- take (iii)
= '!' : '!' : take 1 (repeat '!')          -- subtraction
= '!' : '!' : take 1 ('!' : repeat '!')    -- repeat
= '!' : '!' : '!' : take (1 - 1) (repeat '!') -- take (iii)
= '!' : '!' : '!' : take 0 (repeat '!')    -- subtraction
= '!' : '!' : '!' : []                     -- take (i)
= "!!!"
```


Exercise: Splitting with Predicates

- *takeWhile* :: $(a \to Bool) \to [a] \to [a]$

  takeWhile, applied to a predicate *p* and a list *xs*, returns the longest prefix (possibly empty) of *xs* of elements that satisfy *p*.

- *dropWhile* :: $(a \to Bool) \to [a] \to [a]$

  dropWhile p xs returns the suffix remaining after *takeWhile p xs*.

Laws:

- $takeWhile\ p\ xs \mathbin{+\!\!+} dropWhile\ p\ xs = xs$
- $all\ p\ (takeWhile\ p\ xs) = \mathbf{True}$
- $null\ (dropWhile\ p\ xs) \parallel p\ (head\ (dropWhile\ p\ xs))$

— if *p* is total (on *xs*).

Note: $span\ \ p\ xs = (takeWhile\ p\ xs,\ dropWhile\ p\ xs)$

Exercise: *zipWith*

- *zip* :: $[a] \to [b] \to [(a,\ b)]$

  zip takes two lists and returns a list of corresponding pairs. If one input list is short, excess elements of the longer list are discarded.

- *zipWith* :: $(a \to b \to c) \to [a] \to [b] \to [c]$

  zipWith generalises *zip* by zipping with the function given as the first argument, instead of a tupling function. For example, *zipWith* $(+)$ is applied to two lists to produce the list of corresponding sums.

- *diagonal* :: $[[a]] \to [a]$

  interprets its argument as a matrix, which may be assumed to be square, and returns the main diagonal of that matrix, e.g.:

  $diagonal\ [[1,2,3],[4,5,6],[7,8,9]] = [1,5,9]$

Some Prelude Functions — List Splitting with Predicates

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

dropWhile          :: (a -> Bool) -> [a] -> [a]
dropWhile p []     = []
dropWhile p xs@(x:xs')
       | p x       = dropWhile p xs'
       | otherwise = xs

span, break        :: (a -> Bool) -> [a] -> ([a],[a])
span p []          = ([],[])
span p xs@(x:xs')
       | p x       = let (ys,zs) = span p xs' in (x:ys,zs)
       | otherwise = ([],xs)

break p            = span (not . p)
```

as-Patterns

```
dropWhile           :: (a -> Bool) -> [a] -> [a]
dropWhile p []      = []
dropWhile p xs@(x:xs')
        | p x       = dropWhile p xs'
        | otherwise = xs
```

Consider matching of the third clause against *dropWhile* (< 5) [1,2,3]:

- p = (< 5)
- xs = [1,2,3]
- x = 1
- xs' = [2,3]
- $p\ x$ = (< 5) 1 = 1 < 5 = **True**

Therefore: *dropWhile* (< 5) [1,2,3] = *dropWhile* (< 5) [2,3]

as-Patterns — 2

```
dropWhile           :: (a -> Bool) -> [a] -> [a]
dropWhile p []      = []
dropWhile p xs@(x:xs')
        | p x       = dropWhile p xs'
        | otherwise = xs
```

Consider matching of the third clause against *dropWhile* (< 5) [5,4,3]:

- p = (< 5)
- xs = [5,4,3]
- x = 5
- xs' = [4,3]
- $p\ x$ = (< 5) 5 = 5 < 5 = **False**

Therefore: *dropWhile* (< 5) [5,4,3] = [5,4,3]

What We Have Seen So Far

- **Functional programming:** Higher-order functions, functions as arguments and results
- **Type systems:** type constants and type constructors, parametric polymorphism (type variables), type inference
- **Operator precedence rules:** juxtaposition as operator, "associate to the left/right"
- **Argument passing:** not by value or reference, but by name
- **Powerful datatypes** with simple interface: *Integer*, lists, lists of lists of …
- **Non-local control** (evaluation on demand): modularity (e.g., generate / prune)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\begin{array}{ll}
\mathit{length} :: [a] \to \mathit{Int} & \mathit{concat} :: [[a]] \to [a] \\
\mathit{length}\ [\,] = 0 & \mathit{concat}\ [\,] = [\,] \\
\mathit{length}\ (x : xs) = 1 + \mathit{length}\ xs & \mathit{concat}\ (xs : xss) = xs \mathbin{+\!\!+} \mathit{concat}\ xss
\end{array}$$

$$\begin{array}{ll}
(\mathbin{+\!\!+}) :: [a] \to [a] \to [a] & \mathit{sum} :: \mathit{Num}\ a \Rightarrow [a] \to a \\
[\,] \mathbin{+\!\!+} ys = ys & \mathit{sum}\ [\,] = 0 \\
(x : xs) \mathbin{+\!\!+} ys = x : (xs \mathbin{+\!\!+} ys) & \mathit{sum}\ (x : xs) = x + \mathit{sum}\ xs
\end{array}$$

$$\begin{array}{ll}
\mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \to [a] \to \mathit{Bool} & \mathit{product} :: \mathit{Num}\ a \Rightarrow [a] \to a \\
x\ \text{`}\mathit{elem}\text{`}\ [\,] = \mathbf{False} & \mathit{product}\ [\,] = 1 \\
x\ \text{`}\mathit{elem}\text{`}\ (y : ys) & \mathit{product}\ (x : xs) = x * \mathit{product}\ xs \\
\quad = x \equiv y \mathbin{\|} x\ \text{`}\mathit{elem}\text{`}\ ys &
\end{array}$$

(All these functions are in the standard prelude.)

Defining Functions Over Lists by Structural Induction

Many functions taking lists as arguments can be defined via **structural induction**:

$$\mathit{length} :: [\,a\,] \to \mathit{Int}$$
$$\mathit{length} = \mathit{foldr}\ (\mathit{const}\ (1+))\ 0$$

$$\mathit{concat} :: [\,[\,a\,]\,] \to [\,a\,]$$
$$\mathit{concat} = \mathit{foldr}\ (+\!\!+)\ [\,]$$

$$(+\!\!+) :: [\,a\,] \to [\,a\,] \to [\,a\,]$$
$$\mathit{xs}\ +\!\!+\ \mathit{ys} = \mathit{foldr}\ (:)\ \mathit{ys}\ \mathit{xs}$$

$$\mathit{sum} :: \mathit{Num}\ a \Rightarrow [\,a\,] \to a$$
$$\mathit{sum} = \mathit{foldr}\ (+)\ 0$$

$$\mathit{elem} :: \mathit{Eq}\ a \Rightarrow a \to [\,a\,] \to \mathit{Bool}$$
$$\mathit{elem}\ x = \mathit{foldr}\ (\lambda\ y\ r \to x \equiv y\ \|\ r)\ \mathbf{False}$$

$$\mathit{product} :: \mathit{Num}\ a \Rightarrow [\,a\,] \to a$$
$$\mathit{product} = \mathit{foldr}\ (*)\ 1$$

(All these functions are in the standard prelude.)

foldr

```
foldr               :: (a -> b -> b) -> b -> [a] -> b
foldr (⊗) z []       = z
foldr (⊗) z (x:xs)   = x ⊗ (foldr (⊗) z xs)
```

$$\mathtt{foldr}\ (\otimes)\ \mathtt{z}\ [x_1,\ x_2,\ x_3,\ x_4,\ x_5\ ]$$
$$= x_1 \otimes (\mathtt{foldr}\ (\otimes)\ \mathtt{z}\ [x_2,\ x_3,\ x_4,\ x_5\ ])$$
$$= x_1 \otimes (x_2 \otimes (\mathtt{foldr}\ (\otimes)\ \mathtt{z}\ [x_3,\ x_4,\ x_5\ ]))$$
$$= x_1 \otimes (x_2 \otimes (x_3 \otimes (\mathtt{foldr}\ (\otimes)\ \mathtt{z}\ [x_4,\ x_5\ ])))$$
$$= x_1 \otimes (x_2 \otimes (x_3 \otimes (x_4 \otimes (\mathtt{foldr}\ (\otimes)\ \mathtt{z}\ [x_5\ ]))))$$
$$= x_1 \otimes (x_2 \otimes (x_3 \otimes (x_4 \otimes (x_5 \otimes (\mathtt{foldr}\ (\otimes)\ \mathtt{z}\ [\,])))))$$
$$= x_1 \otimes (x_2 \otimes (x_3 \otimes (x_4 \otimes (x_5 \otimes \mathtt{z}))))$$

foldr1

```
foldr1              :: (a -> a -> a) -> [a] -> a
foldr1 (⊗) [x]       = x
foldr1 (⊗) (x:xs)    = x ⊗ (foldr1 (⊗) xs)
```

$$\mathtt{foldr1}\ (\otimes)\ [x_1,\ x_2,\ x_3,\ x_4,\ x_5\ ]$$
$$= x_1 \otimes (\mathtt{foldr1}\ (\otimes)\ [x_2,\ x_3,\ x_4,\ x_5\ ])$$
$$= x_1 \otimes (x_2 \otimes (\mathtt{foldr1}\ (\otimes)\ [x_3,\ x_4,\ x_5\ ]))$$
$$= x_1 \otimes (x_2 \otimes (x_3 \otimes (\mathtt{foldr1}\ (\otimes)\ [x_4,\ x_5\ ])))$$
$$= x_1 \otimes (x_2 \otimes (x_3 \otimes (x_4 \otimes (\mathtt{foldr1}\ (\otimes)\ [x_5\ ]))))$$
$$= x_1 \otimes (x_2 \otimes (x_3 \otimes (x_4 \otimes x_5\ )))$$

List Folding

***foldr* abstracts structural induction over lists!**

```
foldr               :: (a -> b -> b) -> b -> [a] -> b
foldr f z []         = z
foldr f z (x:xs)     = f x (foldr f z xs)

foldr1              :: (a -> a -> a) -> [a] -> a
foldr1 f [x]         = x
foldr1 f (x:xs)      = f x (foldr1 f xs)

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

foldl1              :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)      = foldl f x xs
```

Lambda-Abstraction

Named functions:	**Anonymous functions:**
add1 x = *x* + 1	(+ 1)
recip x = 1 / *x*	(1 /)
square x = *x* ∗ *x*	λ *x* → *x* ∗ *x*
	`\ x -> x * x`

In "λ *x* → *body*", the variable *x* is **bound.**

Typing rule:

If, assuming $x :: a$, we can get $body :: b$, then $(\lambda\ x \rightarrow body) :: a \rightarrow b$

Evaluation rule: **β-reduction** uses substitution:

$$(\lambda\ x \rightarrow body)\ arg \quad \rightarrow \quad body[x \mapsto \mathrm{arg}]$$

Enumeration Type Definitions

data *Bool* = **False** | **True** **deriving** (*Eq* , *Ord* , *Read* , *Show*)
data *Ordering* = *LT* | *EQ* | *GT* **deriving** (*Eq* , *Ord* , *Read* , *Show*)

data *Suit* = *Diamonds* | *Hearts* | *Spades* | *Clubs* **deriving** (*Eq* , *Ord*)

Pattern matching:

not **False** = **True**
not **True** = **False**

lexicalCombineOrdering :: *Ordering* → *Ordering* → *Ordering*
lexicalCombineOrdering LT _ = *LT*
lexicalCombineOrdering EQ x = *x*
lexicalCombineOrdering GT _ = *GT*

Simple data Type Definitions

data *Point* = *Pt Int Int* **deriving** (*Eq*) −− screen coordinates

This defines at the same time a **data constructor**:

Pt :: *Int* → *Int* → *Point*

Pattern matching:

addPt (*Pt x1 y1*) (*Pt x2 y2*) = *Pt* (*x1* + *x2*) (*y1* + *y2*)

Multi-Constructor data Type Definitions

data *Transport* = *Feet*
| *Bike*
| *Train Int* −− price in cent

This defines at the same time **data constructors**:

Feet :: *Transport*
Bike :: *Transport*
Train :: *Int* → *Transport*

Pattern matching:

cost Feet = 0
cost Bike = 0
cost (*Train Int*) = *Int*

Token Type

```
data Token = Number Integer
           | Sep Char
           | Ident String        deriving (Show)
```

Constructors:

```
Number :: Integer → Token
Sep    :: Char → Token
Ident  :: String → Token
```

Pattern Matching:

```
showToken (Number n) = "Number " ++ show n
showToken (Sep c) = "Sep " ++ show c
showToken (Ident s) = "Ident " ++ show s
```

(Defining this as "*show* :: *Token* → *String*" is the effect of "**deriving** (*Show*)".)

Lexical Analysis — Haskell Example

```
module SimpleLexer where
import Char

data Token = Number Integer
           | Sep Char
           | Ident String        deriving (Show)

simpleLexer :: String → [Token]
simpleLexer (c:cs)
  | isDigit c  =  lexNumber [c] cs
  | isAlpha c  =  lexIdent  [c] cs
  | isSep   c  =  Sep c : simpleLexer cs
  | isSpace c  =  simpleLexer cs
  | otherwise  =  error ("simpleLexer: illegal character: " ++ take 20 (c:cs))
simpleLexer [] = []

lexNumber, lexIdent :: String → String → [Token]
lexNumber prefix (c:cs) | isDigit c    =  lexNumber (prefix ++ [c]) cs
lexNumber prefix s                     =  Number (read prefix) : simpleLexer s
lexIdent  prefix (c:cs) | isAlphaNum c =  lexIdent (prefix ++ [c]) cs
lexIdent  prefix s                     =  Ident prefix : simpleLexer s

isSep c = c `elem` "(){};,+-*/"
```


Simple Polymorphic data Type Definitions

The prelude **type constructors** *Maybe*, *Either*, *Complex* are defined as follows:

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

data Either a b = Left a | Right b         deriving (Eq, Ord, Read, Show)

data Complex r = r :+ r  deriving (Eq, Read, Show)
```

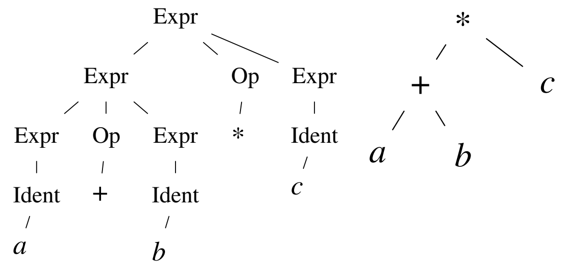
This defines at the same time **data constructors**:

```
Nothing :: Maybe a
Just :: a → Maybe a

Left :: a → Either a b
Right :: b → Either a b

(:+) :: r → r → Complex r
```


Abstract Syntax Example — Haskell

$$Expr \quad \to \quad Ident \mid Number \mid Expr\ Op\ Expr$$

```
data Op = MkOp String
  deriving Show
data Expr
  = Var String
  | Num Integer
  | Bin Expr Op Expr
 deriving Show

expr1 = Bin
         (Bin (Var "a")
            (MkOp "+")
            (Var "b") )
         (MkOp "*")
         (Var "c")
```




```
plus x y = Bin x (MkOp "+") y
mult x y = Bin x (MkOp "*") y

expr2 = (Var "a" `plus` Var "b") `mult` Var "c"
```

Showing *Expr*

```
data Op = MkOp String
  deriving Show

showOp :: Op → String
showOp (MkOp s) = s

data Expr
  = Var String
  | Num Integer
  | Bin Expr Op Expr

showExpr :: Expr → String
showExpr (Var v) = v
showExpr (Num n) = show n
showExpr (Bin e1 op e2) =
    '(' : showExpr e1 ++ showOp op ++ showExpr e2 ++ ")"
```


Exercise: Text Processing

- *lines* :: *String* → [*String*]

  lines breaks a string up into a list of strings at newline characters. The resulting strings do not contain newlines.

- *words* :: *String* → [*String*]

  words breaks a string up into a list of words, which were delimited by white space.

- *unlines* :: [*String*] → *String*

  unlines is an inverse operation to *lines*. It joins lines, after appending a terminating newline to each.

- *unwords* :: [*String*] → *String*

  unwords is an inverse operation to *words*. It joins words with separating spaces.

Some Prelude Functions — Text Processing

```
lines     :: String -> [String]
lines ""   = []
lines s    = let (l,s') = break ('\n'==) s
             in l : case s' of []      -> []
                               (_:s'') -> lines s''

words     :: String -> [String]
words s    = case dropWhile isSpace s of
                  "" -> []
                  s' -> w : words s''
                        where (w,s'') = break isSpace s'

unlines   :: [String] -> String
unlines = foldr (\ l r -> l ++ '\n' : r) []

unwords   :: [String] -> String
unwords []      =  ""
unwords [w]     = w
unwords (w:ws)  = w ++ ' ' : unwords ws
```