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.

Text Processing Prelude Functions

```
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 []      = []
unlines (l:ls)  = l ++ '\n' : unlines ls

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


map and *filter*

These functions embody aspects of list comprehension:

map f xs = [*f x* | *x* ← *xs*]
filter p xs = [*x* | *x* ← *xs*, *p x*]

Examples:

map (7 ∗) [1 .. 6] = [7, 14, 21, 28, 35, 42]
filter even [1 .. 6] = [2, 4, 6]

Definitions:

map :: (*a* → *b*) → ([*a*] → [*b*])
map f [ ] = [ ]
map f (*x* : *xs*) = *f x* : *map f xs*

filter :: (*a* → *Bool*) → ([*a*] → [*a*])
filter p [ ] = [ ]
filter p (*x* : *xs*) = **if** *p x* **then** *x* : *rest* **else** *rest*
where *rest* = *filter p xs*

Defining Functions Over Lists by Structural Induction

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

length :: [*a*] → *Int*
length [ ] = 0
length (*x* : *xs*) = 1 + *length xs*

concat :: [[*a*]] → [*a*]
concat [ ] = [ ]
concat (*xs* : *xss*) = *xs* ++ *concat xss*

(++) :: [*a*] → [*a*] → [*a*]
[ ] ++ *ys* = *ys*
(*x* : *xs*) ++ *ys* = *x* : (*xs* ++ *ys*)

sum :: *Num a* ⇒ [*a*] → *a*
sum [ ] = 0
sum (*x* : *xs*) = *x* + *sum xs*

elem :: *Eq a* ⇒ *a* → [*a*] → *Bool*
x ‘*elem*‘ [ ] = **False**
x ‘*elem*‘ (*y* : *ys*)
= *x* ≡ *y* ‖ *x* ‘*elem*‘ *ys*

product :: *Num a* ⇒ [*a*] → *a*
product [ ] = 1
product (*x* : *xs*) = *x* ∗ *product xs*

(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**:

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

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

$(+\!\!+) :: [a] \to [a] \to [a]$
$xs +\!\!+ ys = foldr\ (:)\ ys\ xs$

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

$elem :: Eq\ a \Rightarrow a \to [a] \to Bool$
$elem\ x = foldr\ (\lambda\ y\ r \to x \equiv y \parallel r)$ **False**

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

(All these functions are in the standard prelude.)

foldrX

$foldrX :: (a \to b \to b) \to b \to [a] \to b$
$foldrX\ (***)\ z\ [\,] = z$
$foldrX\ (***)\ z\ (x:xs) = x\ ***\ (foldrX\ (***)\ z\ xs)$

foldr

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

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

foldr1

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

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

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
```


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]`

Lambda-Abstraction

Named functions:

add1 $x = x + 1$

recip $x = 1 / x$

square $x = x * x$

Anonymous functions:

$(+ 1)$

$(1 /)$

$\lambda\ x \to x * x$

`\ x -> x * x`

In "$\lambda\ x \to body$", the variable x is **bound.**

Typing rule:

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

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

$$(\lambda\ x \to body)\ arg \quad \to \quad body[x \mapsto arg]$$

Some Infinite Lists

$fib :: [Integer]$
$fib = 1 : 1 : zipWith\ (+)\ fib\ (tail\ fib)$

$primes :: [Integer]$