6.1 Applying functions to all elements of a list: `map`

Example

```haskell
map even [1, 2, 3]
= [False, True, False]
```

6.2 Filtering a list: `filter`

Example

```haskell
filter even [1, 2, 3]
= [2]

filter isAlpha "R2-D2"
= "RD"

filter null [[], [1, 2], []]
= [[] , []]
```
6.3 Combining the elements of a list: foldr

Example

\[
\begin{align*}
\text{sum } [] & = 0 \\
\text{sum } (x:xs) & = x + \text{sum } xs \\
\text{sum } [x_1, \ldots, x_n] & = x_1 + \ldots + x_n + 0 \\
\text{concat } [] & = [] \\
\text{concat } (xs:xss) & = xs ++ \text{concat } xss \\
\end{align*}
\]
foldr

foldr (⊕) z [x₁, ..., xₙ] = x₁ ⊕ ... ⊕ xₙ ⊕ z

Defined in Prelude:
foldr :: (a -> a -> a) -> a -> [a] -> a
foldr f a [] = a
foldr f a (x:xs) = x 'f' foldr f a xs

Applications:
sum xs = foldr (+) 0 xs
concat xss = foldr (++) [] xss

What is the most general type of foldr?
foldr

\[ \text{foldr} (\oplus) z [x_1, \ldots, x_n] = x_1 \oplus \ldots \oplus x_n \oplus z \]

Defined in Prelude:

\[
\begin{align*}
\text{foldr} & : (a \to a \to a) \to a \to [a] \to a \\
\text{foldr} f a [] & = a \\
\text{foldr} f a (x:xs) & = x \cdot f \cdot \text{foldr} f a x \\
\end{align*}
\]

Applications:

\[
\begin{align*}
\text{sum} \; xs & = \text{foldr} (+) \; 0 \; xs \\
\text{concat} \; xss & = \text{foldr} (+++) \; [] \; xss \\
\end{align*}
\]

What is the most general type of foldr?

foldr

\[
\begin{align*}
\text{foldr} f a [] & = a \\
\text{foldr} f a (x:xs) & = x \cdot f \cdot \text{foldr} f a x \\
\end{align*}
\]

foldr f a replaces

[ (:) by f and
[ [ ] by a ] ]
Evaluating foldr

\[
\text{foldr } f \ a \ [\ ] \ = \ a \\
\text{foldr } f \ a \ (x : xs) \ = \ x \ 'f' \ \text{foldr } f \ a \ xs
\]

\[
\text{foldr } (+) \ 0 \ [1, -2] \\
= \text{foldr } (+) \ 0 \ (1 : -2 : [])
\]

Evaluating foldr

\[
\text{foldr } f \ a \ [\ ] \ = \ a \\
\text{foldr } f \ a \ (x : xs) \ = \ x \ 'f' \ \text{foldr } f \ a \ xs
\]

\[
\text{foldr } (+) \ 0 \ [1, -2] \\
= \text{foldr } (+) \ 0 \ (1 : -2 : [])
\]

Evaluating foldr

\[
\text{foldr } f \ a \ [\ ] \ = \ a \\
\text{foldr } f \ a \ (x : xs) \ = \ x \ 'f' \ \text{foldr } f \ a \ xs
\]

\[
\text{foldr } (+) \ 0 \ [1, -2] \\
= \text{foldr } (+) \ 0 \ (1 : -2 : [])
\]

More applications of foldr

\[
\text{product } xs \ = \ \text{foldr } (+) \ 1 \ xs
\]

\[
\text{product } xs \ = \ \text{foldr } (+) \ 1 \ xs
\]
More applications of foldr

\[
\begin{align*}
\text{product } xs &= \text{foldr} \ (\ast) \ 1 \ \text{xs} \\
\text{and } xs &= \text{foldr} \ (\&\&) \ \text{True} \ \text{xs}
\end{align*}
\]

\[
\begin{align*}
\text{product } xs &= \text{foldr} \ (\ast) \ 1 \ \text{xs} \\
\text{and } xs &= \text{foldr} \ (\&\&) \ \text{True} \ \text{xs} \\
\text{or } xs &= \text{foldr} \ (\|\|) \ \text{False} \ \text{xs}
\end{align*}
\]

More applications of foldr

\[
\begin{align*}
\text{product } xs &= \text{foldr} \ (\ast) \ 1 \ \text{xs} \\
\text{and } xs &= \text{foldr} \ (\&\&) \ \text{True} \ \text{xs} \\
\text{or } xs &= \text{foldr} \ (\|\|) \ \text{False} \ \text{xs} \\
\text{inSort } xs &= \text{foldr} \ \text{ins} \ [\text{}] \ \text{xs}
\end{align*}
\]

Quiz

What is \( \text{foldr} \ (:) \ \text{ys} \ \text{xs} \)
What is $\text{foldr} (:) \ ys \ xs$

Example: $\text{foldr} (:) \ ys \ (1:2:3:[]) = \quad$ $\text{foldr} (:) \ ys \ xs$

Example: $\text{foldr} (:) \ ys \ (1:2:3:[]) = 1:2:3:ys$

---

Defining functions via $\text{foldr}$

- means you have understood the art of higher-order functions

What is $\text{foldr} (:) \ ys \ xs$

Example: $\text{foldr} (:) \ ys \ (1:2:3:[]) = 1:2:3:ys$

$\text{foldr} (:) \ ys \ xs = \ xs \ ++ \ ys$
Defining functions via `foldr`
- means you have understood the art of higher-order functions
- allows you to apply properties of `foldr`

Example
If $f$ is associative and $a \ f \ x = x$ then

$\text{foldr} \ f \ a \ (\text{x}++\text{ys}) = \text{foldr} \ f \ a \ \text{x} \ 'f' \ \text{foldr} \ f \ a \ \text{ys}.$

Proof by induction on $\text{x}s$. Induction step:

$\text{foldr} \ f \ a \ ((x:xs) ++ ys) = \text{foldr} \ f \ a \ (x : (xs++ys))$
Defining functions via `foldr`

- means you have understood the art of higher-order functions
- allows you to apply properties of `foldr`

**Example**

If \( f \) is associative and \( a \) a `f` \( x = x \) then
\[
foldr \ f \ a \ (xs++ys) = foldr \ f \ a \ xs \ \langle f \rangle \foldr \ f \ a \ ys.
\]

Proof by induction on \( xs \). Induction step:
\[
foldr \ f \ a \ ((x:xs) ++ ys) = foldr \ f \ a \ (x : (xs++ys))
= x \ \langle f \rangle \foldr \ f \ a \ (xs++ys)
= x \ \langle f \rangle \ (foldr \ f \ a \ xs \ \langle f \rangle \foldr \ f \ a \ ys) \quad \text{-- by IH}
foldr \ f \ a \ (x:xs) \ \langle f \rangle \foldr \ f \ a \ ys
\]
Defining functions via \texttt{foldr}

- means you have understood the art of higher-order functions
- allows you to apply properties of \texttt{foldr}

**Example**

If \( f \) is associative and \( f \ x = x \) then
\[
\text{foldr } f \ a \ (x:s+y) = \text{foldr } f \ a \ (x : (x:s+y))
\]
= \( x \ \text{foldr } f \ a \ (x:s+y) \)
= \( x \ (\text{foldr } f \ a \ x \ (f \ \text{foldr } f \ a \ y)) \)  -- by IH
\[
\text{foldr } f \ a \ (x:s) \ (f \ \text{foldr } f \ a \ y)
= \ (x \ (\text{foldr } f \ a \ x \ (f \ \text{foldr } f \ a \ y)) \)  -- by assoc.
\]

Proof by induction on \( x \). Induction step:
\[
\text{foldr } f \ a \ ((x:x)s) = \text{foldr } f \ a \ (x : (x:s+y))
\]
= \( x \ (f \ \text{foldr } f \ a \ (x:s+y)) \)
= \( x \ f \ \text{foldr } f \ a \ x \ (f \ \text{foldr } f \ a \ y)) \)  -- by IH
\[
\text{foldr } f \ a \ (x:x) \ (f \ \text{foldr } f \ a \ y)
= \ (x \ (f \ \text{foldr } f \ a \ x) \ (f \ \text{foldr } f \ a \ y)) \)  -- by assoc.
\]
Therefore, if \( g \ xs = \text{foldr } f \ a \ xs \),
then \( g \ (xs ++ ys) = g \ xs \ f \ g \ ys. \)

Therefore \( \text{sum } (x:s+y) = \text{sum } x + \text{sum } y, \)
\( \text{product } (x:s+y) = \text{product } x \ast \text{product } y, \ldots \)
6.4 Lambda expressions
Consider
\[
squares \; xs \; = \; map \; sqr \; xs \; where \; sqr \; x \; = \; x \times x
\]
Do we really need to define \(sqr\) explicitly? No!

\[
\lambda \; x \to \; x \times x
\]
is the anonymous function with
formal parameter \(x\) and result \(x \times x\)
6.4 Lambda expressions

Consider

\[
\text{squares } \text{xs} = \text{map } \text{sqr } \text{xs} \text{ where } \text{sqr } x = x \times x
\]

Do we really need to define \text{sqr} explicitly? No!

\[
\text{x } \rightarrow x \times x
\]

is the anonymous function with formal parameter \text{x} and result \text{x} \times \text{x}

In mathematics: \[ x \mapsto x \times x \]

Evaluation:

\[(\text{x } \rightarrow x \times x) \ 3 = 3 \times 3 = 9\]

---

6.4 Lambda expressions

Consider

\[
\text{squares } \text{xs} = \text{map } \text{sqr } \text{xs} \text{ where } \text{sqr } x = x \times x
\]

Do we really need to define \text{sqr} explicitly? No!

\[
\text{x } \rightarrow x \times x
\]

is the anonymous function with formal parameter \text{x} and result \text{x} \times \text{x}

In mathematics: \[ x \mapsto x \times x \]

Evaluation:

\[(\text{x } \rightarrow x \times x) \ 3 = 3 \times 3 = 9\]

Usage:

\[
\text{squares } \text{xs} = \text{map } (\text{x } \rightarrow x \times x) \text{xs}
\]
Terminology

\( (\lambda x \rightarrow e_1) e_2 \)

- \( x \): form. par.
- \( e_1 \): result
- \( e_2 \): act. par.

Why “lambda”?

The logician Alonzo Church invented *lambda calculus* in the 1930s.

Logicians write \( \lambda x. e \) instead of \( \lambda x \rightarrow e \)

---

Terminology

\( (\lambda x \rightarrow e_1) e_2 \)

- \( x \): form. par.
- \( e_1 \): result
- \( e_2 \): act. par.

Why “lambda”?

The logician Alonzo Church invented *lambda calculus* in the 1930s.

Logicians write \( \lambda x. e \) instead of \( \lambda x \rightarrow e \)
**Terminology**

\((\lambda x \to e_1) \ e_2\)

- **x**: formal parameter

**Typing lambda expressions**

**Example**

\((\lambda x \to x > 0) :: \text{Int} \to \text{Bool}\)

because \(x :: \text{Int}\) implies \(x > 0 :: \text{Bool}\)

**Typing lambda expressions**

**Example**

\((\lambda x \to x > 0) :: \text{Int} \to \text{Bool}\)

because \(x :: \text{Int}\) implies \(x > 0 :: \text{Bool}\)

**The general rule:**

\((\lambda x \to e) :: T_1 \to T_2\)

**Typing lambda expressions**

**Example**

\((\lambda x \to e) :: T_1 \to T_2\)

if \(x :: T_1\) implies \(e :: T_2\)
Sections of infix operators

(+ 1) means (\x -> x + 1)
(2 *) means (\x -> 2 * x)
(2 ~) means (\x -> 2 ~ x)

Sections of infix operators

(+ 1) means (\x -> x + 1)
(2 *) means (\x -> 2 * x)
(2 ~) means (\x -> 2 ~ x)
(~ 2) means (\x -> x ~ 2)
Sections of infix operators

(+ 1) means  (\x -> x + 1)
(2 *) means  (\x -> 2 * x)
(2 ^) means  (\x -> 2 ^ x)
( ^ 2) means  (\x -> x ^ 2)
etc

Example
squares xs = map ( ^ 2) xs

List comprehension

Just syntactic sugar for combinations of map

[ f x | x <- xs ] = map f xs

filter

[ x | x <- xs, p x ] = filter p xs
List comprehension

Just syntactic sugar for combinations of map

\[ [ f \cdot x \mid x \leftarrow xs ] = \text{map} \ f \ xs \]

filter

\[ [ x \mid x \leftarrow xs, p \ x ] = \text{filter} \ p \ xs \]

and concat

\[ [ f \cdot x \cdot y \mid x \leftarrow xs, y \leftarrow ys ] = \text{concat} \ ( \text{map} \ \ ( ) \ ys) \ xs \]
List comprehension

Just syntactic sugar for combinations of `map`:

\[
[f \; x \mid x \leftarrow xs] = \text{map } f \; xs
\]

`filter`:

\[
[x \mid x \leftarrow xs, p \; x] = \text{filter } p \; xs
\]

and `concat`:

\[
[f \; x \; y \mid x \leftarrow xs, y \leftarrow ys] = \text{concat } (\text{map } (\lambda x \rightarrow \text{map } (\lambda y \rightarrow f \; x \; y) \; ys) \; xs)
\]

6.5 Extensionality

Two functions are equal if for all arguments they yield the same result:

\[f, g :: T_1 \rightarrow T: \forall a. f \; a = g \; a\]

\[\frac{f = g}{f = g}\]

\[f, g :: T_1 \rightarrow T_2 \rightarrow T: \forall a, b. f \; a \; b = g \; a \; b\]

\[\frac{f = g}{f = g}\]
6.6 Curried functions
A trick (re)invented by the logician Haskell Curry

Example
\[
\begin{align*}
f &: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \\
f \ x \ y &= x + y \\
f \ x &= \ \lambda \ y \rightarrow x + y
\end{align*}
\]

Both mean the same:
\[
f \ a \ b = a + b
\]
6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

\[ f :: \text{Int} \to \text{Int} \to \text{Int} \quad f :: \text{Int} \to (\text{Int} \to \text{Int}) \]

\[ f \ x \ y = x + y \quad f \ x = \ \lambda y \to x + y \]

Both mean the same:

\[ f \ a \ b \quad (f \ a) \ b \]

\[ = a + b \quad = (\lambda y \to a + y) \ b \]

The trick: any function of two arguments
6.6 Curried functions
A trick (re)invented by the logician Haskell Curry

Example
\[ f :: \text{Int} \to \text{Int} \to \text{Int} \quad \text{and} \quad f :: \text{Int} \to (\text{Int} \to \text{Int}) \]
\[ f \ x \ y = x+y \quad \text{and} \quad f \ x = \lambda y \to x+y \]
Both mean the same:
\[ f \ a \ b = \lambda y \to a + y \quad \text{and} \quad (f \ a) \ b = a + b \]

The trick: any function of two arguments can be viewed as a function of the first argument.

In general

Every function is a function of one argument (which may return a function as a result)
\[ T_1 \to T_2 \to T \]
is just syntactic sugar for
\[ T_1 \to (T_2 \to T) \]
**In general**

Every function is a function of one argument (which may return a function as a result)

\[ T_1 \rightarrow T_2 \rightarrow T \]

is just syntactic sugar for

\[ T_1 \rightarrow (T_2 \rightarrow T) \]

\[ f \; e_1 \; e_2 \]

is just syntactic sugar for

\[ (f \; e_1) \; e_2 \]

\[ :: T_2 \rightarrow T \]


-> is not associative:

\[ T_1 \rightarrow (T_2 \rightarrow T) \neq (T_1 \rightarrow T_2) \rightarrow T \]

**Example**

\[ f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]

\[ f \; x \; y = x + y \]


-> is not associative:

\[ T_1 \rightarrow (T_2 \rightarrow T) \neq (T_1 \rightarrow T_2) \rightarrow T \]

**Example**

\[ f :: \text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}) \]

\[ f \; x \; y = x + y \]

\[ g :: (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Int} \]

\[ g \; h = h \; 0 \; + \; 1 \]
-> is not associative:
\[ T_1 \to (T_2 \to T) \neq (T_1 \to T_2) \to T \]

Example
\[
f :: \text{Int} \to (\text{Int} \to \text{Int}) \quad g :: (\text{Int} \to \text{Int}) \to \text{Int}
f x y = x + y \quad g h = h 0 + 1
\]

Application is not associative:
\[ (f \ e_1) \ e_2 \neq f \ (e_1 \ e_2) \]

Example
\[
f :: \text{Int} \to (\text{Int} \to \text{Int}) \quad g :: (\text{Int} \to \text{Int}) \to \text{Int}
f x y = x + y \quad g h = h 0 + 1
\]

Application is not associative:
\[ (f \ e_1) \ e_2 \neq f \ (e_1 \ e_2) \]

Example
\[
(f 3) 4 \neq f \ (3 \ 4)
\]

Quiz
head tail xs
Correct?
Partial application

Every function of \( n \) parameters can be applied to less than \( n \) arguments

Example

Instead of \( \text{sum} \; xs = \text{foldr} \; (+) \; 0 \; xs \)
just define \( \text{sum} = \text{foldr} \; (+) \; 0 \)

Partial application

Every function of \( n \) parameters can be applied to less than \( n \) arguments

Example

Instead of \( \text{sum} \; xs = \text{foldr} \; (+) \; 0 \; xs \)
just define \( \text{sum} = \text{foldr} \; (+) \; 0 \)

Partial application

Every function of \( n \) parameters can be applied to less than \( n \) arguments

Example

Instead of \( \text{sum} \; xs = \text{foldr} \; (+) \; 0 \; xs \)
just define \( \text{sum} = \text{foldr} \; (+) \; 0 \)

In general:
If \( f :: T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T \)
and \( a_1 :: T_1, \ldots, a_m :: T_m \) and \( m \leq n \)
then \( f \; a_1 \ldots \; a_m :: T_{m+1} \rightarrow \ldots \rightarrow T_n \rightarrow T \)
6.7 More library functions

\[ f \cdot g = \lambda x \to f (g \; x) \]

\[ (.) \; :: \; (b \to c) \to (a \to b) \to (a \to c) \]
\[ f \cdot g = \lambda x \to f (g \; x) \]

**Example**

\[ \text{head2} = \text{head} \cdot \text{tail} \]

\[ \text{head2} [1,2,3] \]
const :: a -> (b -> a)
const x = \_ -> x

curry :: ((a,b) -> c) -> (a -> b -> c)
curry f = \ x y -> f(x,y)

all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]

Example
all (>1) [0, 1, 2]
= False

any :: (a -> Bool) -> [a] -> Bool
any p = or [p x | x <- xs]
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile p [] = []