Script generated by TTT

Title: Nipkow: Info2 (14.11.2014)

Date: Fri Nov 14 07:30:01 GMT 2014

Duration: 87:02 min

Pages: 106

TTT Fri 08:31

6.3 Combining the elements of a list: `foldr`

6.3 Combining the elements of a list: `foldr`

Example

```
sum []      =  0
sum (x:xs)  =  x + sum xs
```


6.3 Combining the elements of a list: `foldr`

Example

```
sum []      =  0
sum (x:xs)  =  x + sum xs
```


$$\mathtt{sum}\ [x_1,\ \ldots,\ x_n] = x_1 + \ldots + x_n + 0$$

6.3 Combining the elements of a list: `foldr`

Example

```
sum []        =  0
sum (x:xs)    =  x + sum xs
```

$$\texttt{sum}\ [x_1,\ \ldots,\ x_n] = x_1 + \ldots + x_n + 0$$

```
concat []          =  []
concat (xs:xss)    =  xs ++ concat xss
```


6.3 Combining the elements of a list: `foldr`

Example

```
sum []        =  0
sum (x:xs)    =  x + sum xs
```

$$\texttt{sum}\ [x_1,\ \ldots,\ x_n] = x_1 + \ldots + x_n + 0$$

```
concat []          =  []
concat (xs:xss)    =  xs ++ concat xss
```


$$\texttt{concat}\ [xs_1,\ \ldots,\ xs_n] = xs_1\ \texttt{++}\ \ldots\ \texttt{++}\ xs_n\ \texttt{++}\ \texttt{[]}$$

`foldr`

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

`foldr`

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

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
```

foldr

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

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
```

foldr

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

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

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

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

```
foldr f a []       =  a
foldr f a (x:xs)   =  x `f` foldr f a xs
```

`foldr f a` replaces
`(:)` by `f` and
`[]` by `a`

foldr

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

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

```
foldr f a []       =  a
foldr f a (x:xs)   =  x `f` foldr f a xs
```

`foldr f a` replaces		
`(:)`	by	`f` and
`[]`	by	`a`

Evaluating foldr

```
foldr f a []       =  a
foldr f a (x:xs)   =  x `f` foldr f a xs
```

Evaluating foldr

```
foldr f a []       =  a
foldr f a (x:xs)   =  x `f` foldr f a xs

foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
```

Evaluating foldr

```
foldr f a []       =  a
foldr f a (x:xs)   =  x ‘f‘ foldr f a xs

foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
= 1 + foldr (+) 0 (-2 : [])
```

Evaluating foldr

```
foldr f a []       =  a
foldr f a (x:xs)   =  x ‘f‘ foldr f a xs

foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
= 1 + foldr (+) 0 (-2 : [])
= 1 + -2 + (foldr (+) 0 [])
```

Evaluating foldr

```
foldr f a []       =  a
foldr f a (x:xs)   =  x ‘f‘ foldr f a xs

foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
= 1 + foldr (+) 0 (-2 : [])
= 1 + -2 + (foldr (+) 0 [])
= 1 + -2 + 0
```

Evaluating foldr

```
foldr f a []       =  a
foldr f a (x:xs)   =  x ‘f‘ foldr f a xs

foldr (+) 0 [1, -2]
= foldr (+) 0 (1 : -2 : [])
= 1 + foldr (+) 0 (-2 : [])
= 1 + -2 + (foldr (+) 0 [])
= 1 + -2 + 0
= -1
```

More applications of foldr

```
product xs  =  foldr  (*)   1      xs
```

More applications of foldr

```
product xs  =  foldr  (*)   1      xs
and xs      =  foldr  (&&)  True   xs
```

More applications of foldr

```
product xs  =  foldr  (*)   1      xs
and xs      =  foldr  (&&)  True   xs
or xs       =  foldr  (||)  False  xs
```

More applications of foldr

```
product xs  =  foldr  (*)   1      xs
and xs      =  foldr  (&&)  True   xs
or xs       =  foldr  (||)  False  xs
inSort xs   =  foldr  ins   []     xs
```

Quiz

What is

`foldr (:) ys xs`

Quiz

What is

`foldr (:) ys xs`

Example: `foldr (:) ys (1:2:3:[]) =`

Quiz

What is

`foldr (:) ys xs`

Example: `foldr (:) ys (1:2:3:[]) = 1:2:3:ys`

Quiz

What is

`foldr (:) ys xs`

Example: `foldr (:) ys (1:2:3:[]) = 1:2:3:ys`

`foldr (:) ys xs = xs ++ ys`

Proof by induction on `xs` (Exercise!)

Defining functions via `foldr`

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

Defining functions via `foldr`

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

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
``foldr f a (xs++ys) = foldr f a xs `f` foldr f a 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
``foldr f a (xs++ys) = foldr f a xs `f` foldr f a ys``.

Proof by induction on `xs`. Induction step:

```
foldr f a ((x:xs) ++ ys) = foldr f a (x : (xs++ys))
= x `f` foldr f a (xs++ys)
= x `f` (foldr f a xs `f` foldr f a ys)      -- by IH
foldr f a (x:xs) `f` foldr f a ys
= (x `f` foldr f a xs) `f` foldr f a ys
= x `f` (foldr f a xs `f` foldr f a ys)      -- by assoc.
```

Therefore, if `g xs = foldr f a xs`,
then ``g (xs ++ ys) = g xs `f` g ys``.

6.4 Lambda expressions

Consider

```
squares xs  =  map sqr xs  where  sqr x = x * x
```

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
`foldr f a (xs++ys) = foldr f a xs ‘f‘ foldr f a ys`.

Proof by induction on `xs`. Induction step:

```
foldr f a ((x:xs) ++ ys) = foldr f a (x : (xs++ys))
= x ‘f‘ foldr f a (xs++ys)
= x ‘f‘ (foldr f a xs ‘f‘ foldr f a ys)     -- by IH
foldr f a (x:xs) ‘f‘ foldr f a ys
= (x ‘f‘ foldr f a xs) ‘f‘ foldr f a ys
= x ‘f‘ (foldr f a xs ‘f‘ foldr f a ys)     -- by assoc.
```

Therefore, if `g xs = foldr f a xs`,
then `g (xs ++ ys) = g xs ‘f‘ g ys`.

Therefore `sum (xs++ys) = sum xs + sum ys`,
`product (xs++ys) = product xs * product ys`, ...

6.4 Lambda expressions

Consider

`squares xs  =  map sqr xs  where  sqr x = x * x`

6.4 Lambda expressions

Consider

`squares xs  =  map sqr xs  where  sqr x = x * x`

Do we really need to define `sqr` explicitly? No!

6.4 Lambda expressions

Consider

`squares xs = map sqr xs where sqr x = x * x`

Do we really need to define `sqr` explicitly? No!

`\x -> x * x`

is the anonymous function with

6.4 Lambda expressions

Consider

`squares xs = map sqr xs where sqr x = x * x`

Do we really need to define `sqr` explicitly? No!

`\x -> x * x`

is the anonymous function with

formal parameter `x` and result `x * x`

6.4 Lambda expressions

Consider

`squares xs = map sqr xs where sqr x = x * x`

Do we really need to define `sqr` explicitly? No!

`\x -> x * x`

is the anonymous function with

formal parameter `x` and result `x * x`

In mathematics: $x \mapsto x * x$

6.4 Lambda expressions

Consider

`squares xs = map sqr xs where sqr x = x * x`

Do we really need to define `sqr` explicitly? No!

`\x -> x * x`

is the anonymous function with

formal parameter `x` and result `x * x`

In mathematics: $x \mapsto x * x$

Evaluation:

`(\x -> x * x) 3 = 3 * 3 = 9`

6.4 Lambda expressions

Consider

```
squares xs  =  map sqr xs  where  sqr x = x * x
```

Do we really need to define `sqr` explicitly? No!

```
\x -> x * x
```

is the anonymous function with

formal parameter `x` and result `x * x`

In mathematics: $x \mapsto x * x$

Evaluation:

```
(\x -> x * x) 3 = 3 * 3 = 9
```

Usage:

```
squares xs  =  map (\x -> x * x) xs
```

Terminology

(\x -> e_1) e_2

x: formal parameter
e_1: result

Terminology

(\x -> e_1) e_2

x: formal parameter
e_1: result
e_2: actual parameter

Why "lambda"?

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

Terminology

(\x -> e_1) e_2

x: formal parameter
e_1: result
e_2: actual parameter

Why "lambda"?

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

Logicians write $\lambda x.\ e$ instead of `\x -> e`

Typing lambda expressions

Example

`(\x -> x > 0) :: Int -> Bool`

because `x :: Int` implies `x > 0 :: Bool`

Typing lambda expressions

Example

`(\x -> x > 0) :: Int -> Bool`

because `x :: Int` implies `x > 0 :: Bool`

The general rule:

$$\texttt{(\textbackslash}x\ \texttt{->}\ e\texttt{)}\ \texttt{::}\ T_1\ \texttt{->}\ T_2$$

Typing lambda expressions

Example

`(\x -> x > 0) :: Int -> Bool`

because `x :: Int` implies `x > 0 :: Bool`

The general rule:

$$\texttt{(\textbackslash}x\ \texttt{->}\ e\texttt{)}\ \texttt{::}\ T_1\ \texttt{->}\ T_2$$

if $x\ \texttt{::}\ T_1$ implies $e\ \texttt{::}\ T_2$

Evaluating lambda expressions

$\texttt{(\textbackslash}x\ \texttt{->}\ \textit{body}\texttt{)}\ \textit{arg}$ = *body* with x replaced by *arg*

Evaluating lambda expressions

`(\x -> body) arg` = *body* with *x* replaced by *arg*

Evaluating lambda expressions

`(\x -> body) arg` = *body* with *x* replaced by *arg*

Example

```
(\xs -> xs ++ xs) [1] = [1] ++ [1]
```

Sections of infix operators

```
(+ 1)  means  (\x -> x + 1)
```

Sections of infix operators

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

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

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
```

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

`filter`

```
[ x | x <- xs, p x ]     =     filter p xs
```

and `concat`

```
      [f x y | x <- xs, y <- ys] =
concat (                              )
```

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
```

and `concat`

```
      [f x y | x <- xs, y <- ys] =
concat (map (                        ) 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
```

and `concat`

```
[f x y | x <- xs, y <- ys] =
concat (map (\x -> map (\y ->       ) ys) 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
```

and `concat`


```
[f x y | x <- xs, y <- ys] =
concat (map (\x -> map (\y -> f x y) ys) xs)
```

6.5 Extensionality

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

6.5 Extensionality

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

f, g :: T_1 -> T:

$$\frac{\forall a.\ f\ a = g\ a}{f = g}$$

6.5 Extensionality

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

f, g :: T_1 -> T:

$$\frac{\forall a.\ f\ a = g\ a}{f = g}$$

f, g :: T_1 -> T_2 -> T:

$$\frac{\forall a, b.\ f\ a\ b = g\ a\ b}{f = g}$$

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

```
f :: Int -> Int -> Int
f x y  =  x+y
```

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

```
f :: Int -> Int -> Int
f x y  =  x+y              f x  =  \y -> x+y
```

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

```
f :: Int -> Int -> Int        f :: Int -> (Int -> Int)
f x y  =  x+y                 f x  =  \y -> x+y
```

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

`f :: Int -> Int -> Int`   `f :: Int -> (Int -> Int)`

`f x y  =  x+y`   `f x  =  \y -> x+y`

Both mean the same:

```
f a b
= a + b
```

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

`f :: Int -> Int -> Int`   `f :: Int -> (Int -> Int)`

`f x y  =  x+y`   `f x  =  \y -> x+y`

Both mean the same:

```
f a b                         (f a) b
= a + b
```

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

`f :: Int -> Int -> Int`   `f :: Int -> (Int -> Int)`

`f x y  =  x+y`   `f x  =  \y -> x+y`

Both mean the same:

```
f a b                         (f a) b
= a + b                       = (\y -> a + y) b
```

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

```
f :: Int -> Int -> Int        f :: Int -> (Int -> Int)
f x y  =  x+y                 f x  =  \y -> x+y
```

Both mean the same:

```
f a b                         (f a) b
= a + b                       = (\y -> a + y) b
                              = a + b
```

The trick: any function of two arguments

6.6 Curried functions

A trick (re)invented by the logician Haskell Curry

Example

```
f :: Int -> Int -> Int        f :: Int -> (Int -> Int)
f x y  =  x+y                 f x  =  \y -> x+y
```

Both mean the same:

```
f a b                         (f a) b
= a + b                       = (\y -> a + y) 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)

In general

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

T_1 `->` T_2 `->` T

is just syntactic sugar for

T_1 `->` `(`T_2 `->` T`)`

In general

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

T_1 -> T_2 -> T

is just syntactic sugar for

T_1 -> (T_2 -> T)

$f\ e_1\ e_2$

is just syntactic sugar for

$(f\ e_1)\ e_2$

In general

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

T_1 -> T_2 -> T

is just syntactic sugar for

T_1 -> (T_2 -> T)

$f\ e_1\ e_2$

is just syntactic sugar for

$$\underbrace{(f\ e_1)}_{::\,T_2\,\texttt{->}\,T}\ e_2$$

Analogously for more arguments

-> is not associative:

T_1 -> (T_2 -> T) $\neq$ (T_1 -> T_2) -> T

-> is not associative:

T_1 -> (T_2 -> T) $\neq$ (T_1 -> T_2) -> T

Example

```
f :: Int -> (Int -> Int)
f x y  =  x + y
```

`->` is not associative:

$$T_1 \texttt{ -> } (T_2 \texttt{ -> } T) \neq (T_1 \texttt{ -> } T_2) \texttt{ -> } T$$

Example

```
f :: Int -> (Int -> Int)     g :: (Int -> Int) -> Int
f x y  =  x + y              g h  =  h 0 + 1
```

`->` is not associative:

$$T_1 \texttt{ -> } (T_2 \texttt{ -> } T) \neq (T_1 \texttt{ -> } T_2) \texttt{ -> } T$$

Example

```
f :: Int -> (Int -> Int)
f x y  =  x + y
```

`->` is not associative:

$$T_1 \texttt{ -> } (T_2 \texttt{ -> } T) \neq (T_1 \texttt{ -> } T_2) \texttt{ -> } T$$

Example

```
f :: Int -> (Int -> Int)     g :: (Int -> Int) -> Int
f x y  =  x + y              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)`   `g (id abs)` $\neq$ `(g id) abs`

Quiz

~~`head tail xs`~~

Correct?

`head (tail xs)`

Quiz

`head tail xs`

Correct?

Quiz

~~`head tail xs`~~

Correct?

`head (tail xs)`

Partial application

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

Partial application

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

Example

Instead of	`sum xs`	=	`foldr (+) 0 xs`
just define	`sum`	=	`foldr (+) 0`

Partial application

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

Example

Instead of `sum xs = foldr (+) 0 xs`
just define `sum = foldr (+) 0`

In general:

If f `::` T_1 `->` ... `->` T_n `->` T
and a_1 `::` T_1, ..., a_m `::` T_m and $m \leq n$
then $f\ a_1 \ldots a_m$ `::` T_{m+1} `->` ... `->` T_n `->` T

6.7 More library functions

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

6.7 More library functions

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

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

head2 [1,2,3]
= (head . tail) [1,2,3]
```

6.7 More library functions

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

Example

```
head2  =  head . tail

head2 [1,2,3]
= (head . tail) [1,2,3]
= (\x -> head (tail x)) [1,2,3]
```

6.7 More library functions

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

Example

```
head2  =  head . tail

head2 [1,2,3]
= (head . tail) [1,2,3]
= (\x -> head (tail x)) [1,2,3]
= head (tail [1,2,3])
= head [2,3]
```

6.7 More library functions

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

Example

```
head2  =  head . tail

head2 [1,2,3]
= (head . tail) [1,2,3]
= (\x -> head (tail x)) [1,2,3]
= head (tail [1,2,3])
= head [2,3]
= 2
```

```
const :: a -> (b -> a)
const x  =  \ _ -> x
```

```
const :: a -> (b -> a)
const x  =  \ _ -> x

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

```
const :: a -> (b -> a)
const x  =  \ _ -> x

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

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

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

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

Example

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

```
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]
```

Example

```
any (>1) [0, 1, 2]
= True
```

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

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

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

Example

```
takeWhile (not . isSpace) "the end"
= "the"
```

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

Example

```
dropWhile (not . isSpace) "the end"
```


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

Example

```
takeWhile (not . isSpace) "the end"
= "the"
```

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

Example

```
dropWhile (not . isSpace) "the end"
= " end"
```


6.8 Case study: Counting words

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