Working with lists in Haskell

Vladimir Alekseichenko

Agenda

- Basic functions
- List transformations
- Reducing lists
- Special folds
- Sublists
- Searching lists
- Indexing lists
- Zipping

Basic functions

Basic functions

head

```haskell
{- get the first element in the list
--
-- head [1, 2, 3]
-- 1
-}
head         :: [a] -> a
head (x:_) = x
head []      = error "head: empty list"
```

Basic functions

tail

```
{- obtain a list of all the elements except the first
--
-- tail [1, 2, 3]
-- [2, 3]
-}
tail           :: [a] -> [a]
tail (x:xs)    = xs
tail []        = error "tail: empty list"
```

Basic functions

last

```haskell
{- get the last element in the list
--
-- last [1, 2, 3]
--  last [2, 3]
--    last [3]
-- 3
-}
last           :: [a] -> a
last [x]       = x
last (x: xs) = last xs
last []        = error "last: empty list"
```

Basic functions

init

```
{- obtain a list of all the elements except the last
--
-- init [1, 2, 3]
--   1 : init [2, 3]
--     1 : 2 : init [3]
--       1 : 2 : []
-- [1, 2]
-}
init          :: [a] -> [a]
init [x]      = []
init (x:xs)   = x : init xs
init []       = error "init: empty list"
```

Basic functions

null

```
{- test whether a list is empty
--
-- null [1, 2, 3]
-- False
-}
null     :: [a] -> Bool
null [] = True
null _  = False
```

Basic functions

length

```haskell
{- get the length of the list
--
-- length [1, 2, 3]
--   1 + length [2, 3]
--     1 + 1 + length [3]
--       1 + 1 + 1
-- 3
-}
length          :: [a] -> Int
length []       = 0
length (x:xs)   = 1 + length xs
```

List transformations

List transformations

map

```
{- applying f to each element of list
--
-- map (+3) [1, 2, 3]
--   (1+3) : map (+3) [2, 3]
--      (1+3) : (2+3): (+3) map[3]
--         (1+3) : (2+3) : (3+3) : []
-- [4, 5, 6]
-}
map            :: (a -> a) -> [a] -> [a]
map _ []       = []
map f (x:xs)   = (f x) : map f xs
```

List transformations

reverse

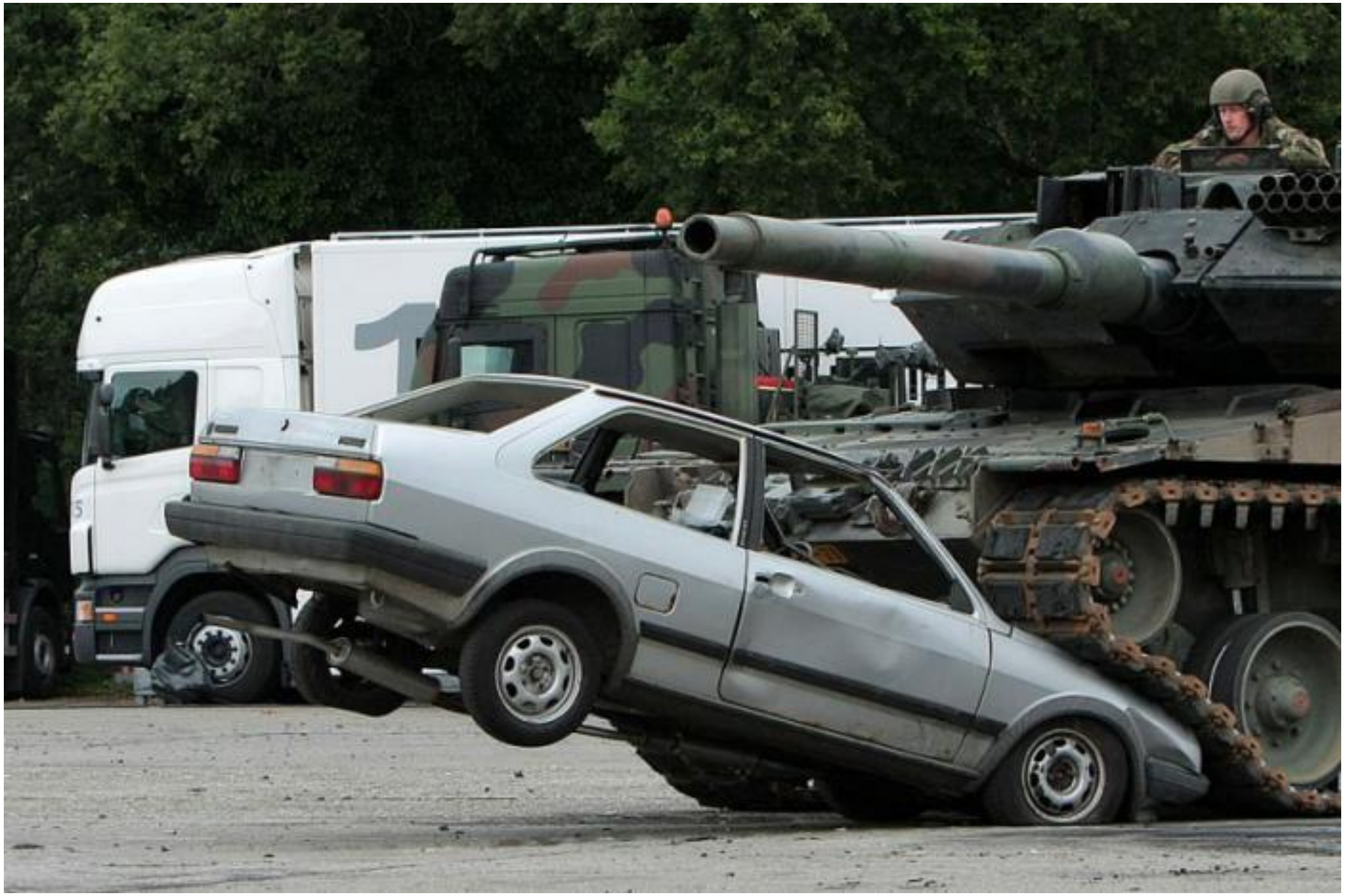
```
{- get the elements of list in reverse order
--
-- reverse [1, 2, 3]
--   reverse [2, 3] ++ [1]
--     reverse [3] ++ [2] ++ [1]
--       reverse [] ++ [3] ++ [2] ++ [1]
-- [3, 2, 1]
-}
reverse            :: [a] -> [a]
reverse []         = []
reverse (x:xs)     = reverse xs ++ [x]
```

List transformations

++

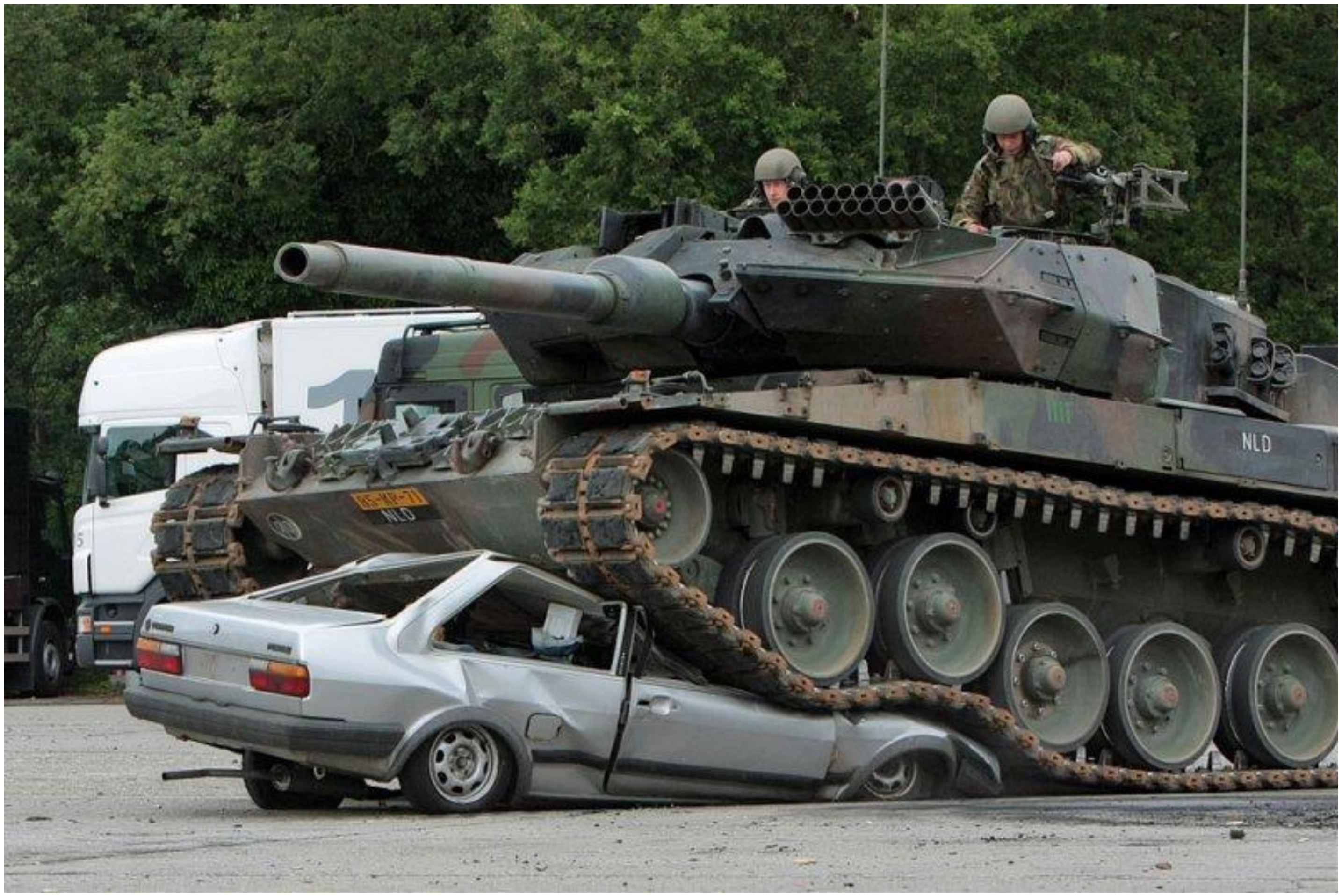
```haskell
{- connections of the two lists into one
--
-- [1, 2, 3] ++ [4, 5, 6]
--   1 : [2, 3] ++ [4, 5, 6]
--     1 : 2 : [3] ++ [4, 5, 6]
--       1 : 2 : 3 : [] ++ [4, 5, 6]
-- [1, 2, 3, 4, 5, 6]
-}
(++)               :: [a] -> [a] -> [a]
(++) x []          = x
(++) [] y          = y
(++) (x:xs) ys = x : xs ++ ys
```

Folds (0)



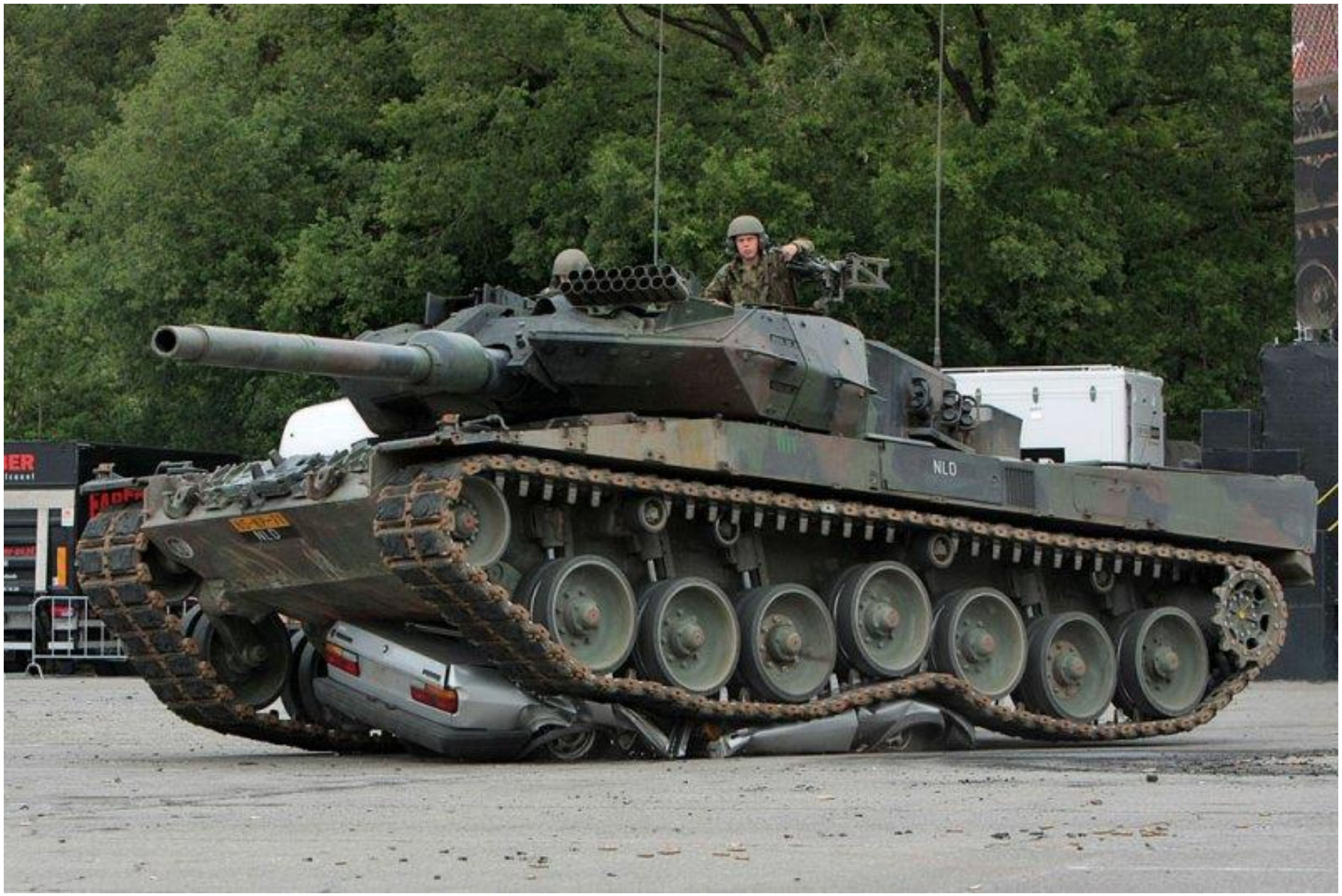
http://latrola.net/blok/tanque-militar-contra-coche-7-fotografias

Folds (1)



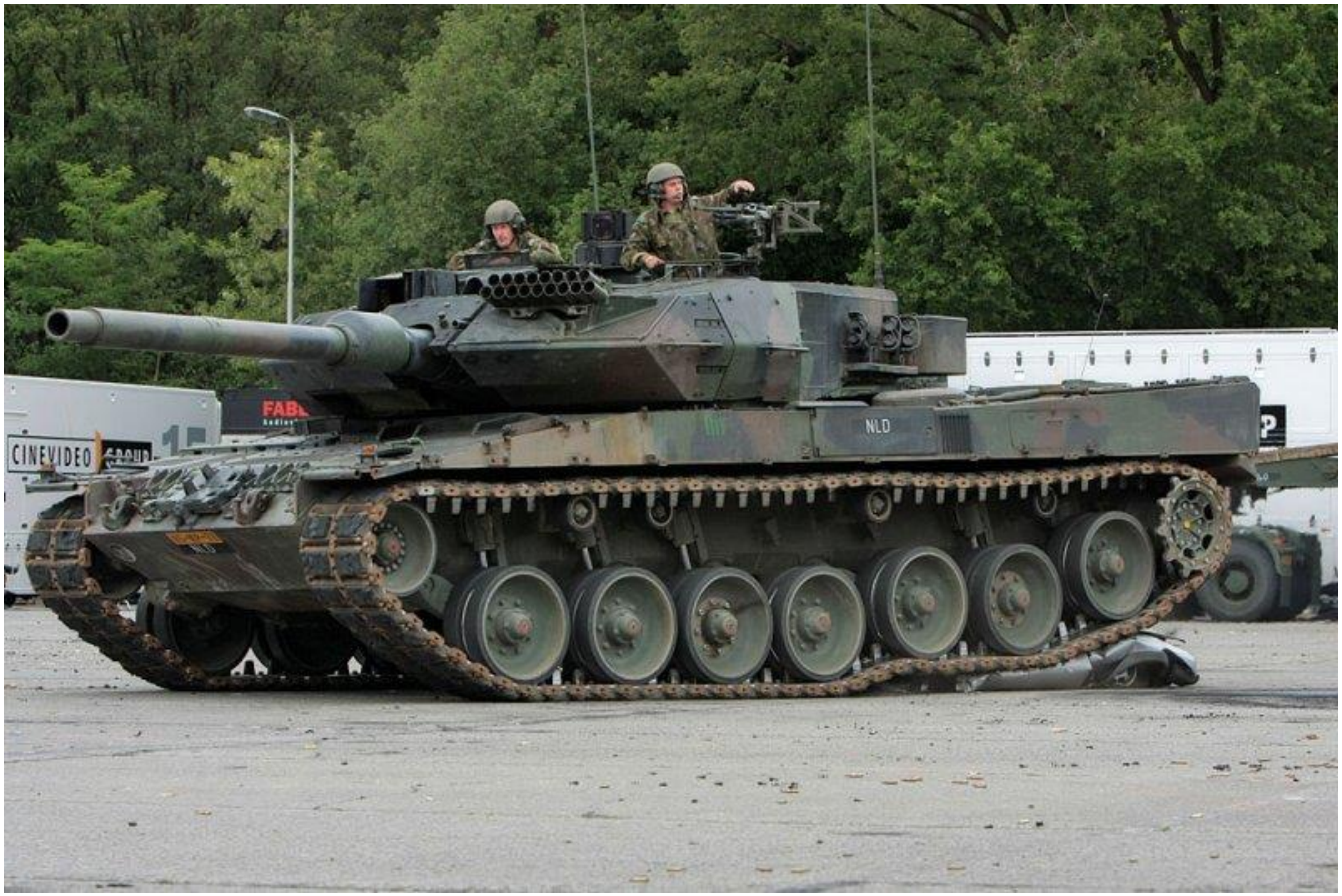
http://latrola.net/blok/tanque-militar-contra-coche-7-fotografias

Folds (2)



http://latrola.net/blok/tanque-militar-contra-coche-7-fotografias

Folds (3)



http://latrola.net/blok/tanque-militar-contra-coche-7-fotografias

Reducing lists (folds)

foldl

```haskell
{- applied to a binary operator, a starting value and a list,
-- reduces the list using the binary operator, from left to right
--
-- foldl (+) 5 [1, 2, 3]
--   foldl (+) ((+) 5 1) [2, 3]
--     foldl (+) ((+) ((+) 5 1) 2) [3]
--       foldl (+) ((+) ((+) ((+) 5 1) 2) 3) []
--       ((5 + 1) + 2) + 3
-- 11
-}
foldl              :: (a -> b -> a) -> a -> [b] -> a
foldl _ z []       = z
foldl f z (x:xs) = foldl f (f z x) xs
```

Reducing lists (folds)

foldl1

```haskell
{- it's a variant of foldl that has no starting value argument,
-- and thus must be applied to non-empty lists.
--
-- foldl1 (+) [1, 2, 3]
--   foldl (+) 1 [2, 3]
--     foldl (+) ((+) 1 2) [3]
--       foldl (+) ((+) ((+) 1 2) 3) []
--       (1 + 2) + 3
-- 6
-}
foldl1          :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs) = foldl f x xs
foldl1 _ []     = error "foldl1: empty list"
```

Reducing lists (folds)

foldr

```haskell
{- applied to a binary operator, a starting value,
-- and a list, reduces the list using the binary operator,
-- from right to left
--
-- foldr (+) 0 [1, 2, 3]
--   (+) 1 (foldr (+) 0 [2, 3])
--     (+) 1 ((+) 2 (foldr (+) 0 [3]))
--       (+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))
--       (+) 1 ((+) 2 ((+) 3 0))
--       (+) 1 ((+) 2 3)
--       (+) 1 5
-- 6
-}
foldr               :: (a -> b -> b) -> b -> [a] -> b
foldr _ z []        = z
foldr f z (x:xs)    = f x (foldr f z xs)
```

Reducing lists (folds)

foldr1

```
{- it's a variant of foldr that has no starting value argument,
-- and thus must be applied to non-empty lists.
--
-- foldr1 (+) [1, 2, 3]
--   foldr (+) 1 [2, 3]
--     2 + (foldr (+) 1 [3])
--       2 + (3 + (foldr (+1) 1 []))
--         2 + (3 + 1)
--  6
-}
foldr1            :: (a -> a -> a) -> [a] -> a
foldr1 f (x:xs) = foldr f x xs
foldr1 _ []       = error "foldr1: empty list"
```

Reducing lists (folds)
using foldl

```
{- another way to implement a reverse (using foldl)
--
-- foldl (flip (:)) [] [1, 2, 3]
--   foldl (flip (:)) ((:) 1 []) [2, 3]
--     foldl (flip (:)) ((:) 2 [1]) [3]
--       foldl (flip (:)) ((:) 3 [2, 1]) []
-- [3, 2, 1]
-}
reverse'    :: [a] -> [a]
reverse' xs = foldl (flip (:)) [] xs
```

flip

```
{- takes its (first) two arguments in the reverse order of f
--
-- flip (:) [] 1
--   1 : []
-- [1]
-}
flip :: (a -> b -> c) -> b -> a -> c
flip f x y = f y x
```

Special folds

Special folds
and

```
{- the conjunction of a Boolean list
-- and [True, True, True]
--   foldl (&&) True [True, True, True]
--      foldl (&&) (True && True) [True, True]
--         foldl (&&) (True && True) [True]
--            foldl (&&) (True && True) []
-- True
-}
and     :: [Bool] -> Bool
and xs = foldl (&&) True xs
```

Bolean „and”

```haskell
{- Boolean "and"
--
-- True && True
-- True
-}
(&&)            :: Bool -> Bool -> Bool
(&&) True True = True
(&&) _ _       = False
```

Special folds

or

```haskell
{- the disjunction of a Boolean list
--
-- or [False, False, True]
-- foldl (||) False [False, False, True]
--   foldl (||) (False || False) [False, True]
--     foldl (||) (False || False) [True]
--       foldl (||) (False || True) []
--       False || True
-- True
-}
or     :: [Bool] -> Bool
or xs = foldl (||) False xs
```

Boolean „or”

```haskell
{- Boolean "or"
--
-- False || True
-- True
-}
(||)            :: Bool -> Bool -> Bool
(||) True _     = True
(||) _ True     = True
(||) _ _        = False
```

Special folds

sum

```haskell
{- computes the sum of a finite list of numbers
--
-- sum [1, 2, 3]
--   foldl (+) 0 [1, 2, 3]
-- 6
-}
sum :: (Num a) => [a] -> a
sum = foldl (+) 0
```

Special folds
product

```
{- computes the product of a finite list of numbers
--
-- product [1, 2, 3]
--   foldl (*) 1 [1, 2, 3]
-- 6
-}
product :: (Num a) => [a] -> a
product = foldl (*) 1
```

Special folds

maximum

```haskell
{- get the maximum value from a list,
-- which must be non-empty, finite, and of an ordered type
--
-- maximum [1, 2, 3]
--   fold1 max [1, 2, 3]
--     fold max 1 [2, 3]
--       fold max (max 1 2) [3]
--          fold max (max (max 1 2) 3) []
--          max (max 1 2) 3
-- 3
-}
maximum :: (Ord a) => [a] -> a
maximum [] = error "maximum: empty list"
maximum xs = foldl1 max xs
```

Special folds

minimum

```
{- get the minimum value from a list,
-- which must be non-empty, finite, and of an ordered type
--
-- minimum [1, 2, 3]
--  foldl1 min [1, 2, 3]
--    foldl min 1 [2, 3]
--      foldl min (min 1 2) [3]
--        foldl min (min (min 1 2) 3) []
--        min (min 1 2) 3
-- 1
-}
minimum :: (Ord a) => [a] -> a
minimum [] = error "minimum: empty list"
minimum xs = foldl1 min xs
```

Extracting sublists

Extracting sublists

take

```
{- get the prefix of xs of length n,
-- or xs itself if n > length xs
--
-- take 2 [1, 2, 3]
--   1 : take (2-1) [2, 3]
--     1 : 2 : take (1-1) [3]
--     1 : 2 []
-- [1, 2]
-}
take                :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []           = []
take n (x: xs)      = x : take (n-1) xs
```

Extracting sublists

drop

```haskell
{- get the suffix of xs after the first n elements,
-- or [] if n > length xs
--
-- drop 2 [1, 2, 3, 4, 5]
--   drop 1 [2, 3, 4, 5]
--     drop 0 [3, 4, 5]
-- [3, 4, 5]
-}
drop                 :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs
drop _ []           = []
drop n (_:xs)       = drop (n-1) xs
```

Searching by equality

Searching by equality
elem

```
{- elem is the list membership predicate
--
-- elem 3 [1, 2, 3, 4, 5]
--   elem 3 [2, 3, 4, 5]
--     elem 3 [3, 4, 5]
-- True
-}
elem                          :: Eq a => a -> [a] -> Bool
elem _ []                     = False
elem e (x:xs) | e == x        = True
              | otherwise     = elem e xs
```

Searching by equality

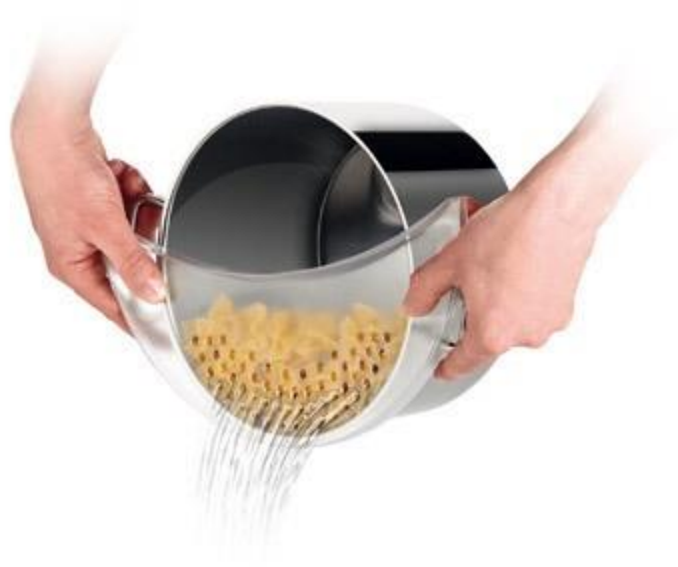
notElem

```
{- it's the negation of elem
--
-}
notElem        :: Eq a => a -> [a] -> Bool
notElem e xs = not (elem e xs)
```

Boolean „not”

```
{- Boolean "not"
--
-}
not            :: Bool -> Bool
not True       = False
not False      = True
```

Filter



http://centr-new.ru/catalog/Kukhonnie-prinadlezhnosti/mnogofunktsionalnyj-durshlag-presto-tescoma

Searching with a predicate

filter

```haskell
{- applied to a predicate and a list,
-- returns the list of those elements that satisfy the predicate
--
-- filter (>1) [1, 2, 3]
--   filter (>1) [2, 3]
--     2 : filter (>1) [3]
--       2 : 3 : filter (>1) []
--       2 : 3 : []
-- [2, 3]
-}
filter                    :: (a -> Bool) -> [a] -> [a]
filter _ []               = []
filter f (x:xs) | f x       = x : rest
                | otherwise = rest
                  where rest = filter f xs
```

Searching with a predicate

partition

```haskell
{- takes a predicate a list and returns
-- the pair of lists of elements which do and do not satisfy the predicate
--
-- partition (==2) [1, 2, 3]
-- ([2], [1, 3])
-}
partition :: (a -> Bool) -> [a] -> ([a], [a])
partition _ [] = ([],[])
partition f xs = partition' f xs [] [] where
    partition' _ [] l r                   = (l,r)
    partition' f (x:xs) l r | f x         = partition' f xs (x:l)   r
                            | otherwise = partition' f xs   l   (x:r)
```

Indexing lists

Indexing lists

(!!)

```
{- list index (subscript) operator, starting from 0
--
-- [1, 2, 3] !! 1
--   [2, 3] !! 0
-- 2
-}
(!!)                         :: [a] -> Int -> a
(!!) (x:_) 0                  = x
(!!) (x:xs) n | n > 0         = xs !! (n-1)
              | otherwise     = error "(!!): negative index"
(!!) _ _                      = error "(!!): index too large"
```

otherwise

```haskell
{- it's defined as the value True
-- it helps to make guards more readable
-}
otherwise :: Bool
otherwise = True
```

Zipping

Zipping

zip

```haskell
{- takes two lists and returns a list of corresponding pairs
--
-- zip [1, 2, 3] [4, 5, 6]
--   (1,4) : zip [2, 3] [5, 6]
--     (1, 4) : (2, 5) : zip [3] [6]
--     (1, 4) : (2, 5) : (3, 6) : zip [] []
-- [(1, 4), (2, 5), (3, 6)]
-}
zip                 :: [a] -> [b] -> [(a, b)]
zip [] _            = []
zip _ []            = []
zip (x:xs) (y:ys)   = (x,y): zip xs ys
```

Zipping

zip3

```
{- takes three lists and returns a list of triples, analogous to zip
--
-}
zip3                       :: [a] -> [b] -> [c] -> [(a, b, c)]
zip3 [] _ _                = []
zip3 _ [] _                = []
zip3 _ _ []                = []
zip3 (x:xs) (y:ys) (z:zs) = (x, y, z) : zip3 xs ys zs
```

Zipping
zipWith

```
{- zipWith generalises zip by zipping with the function
-- given as the first argument, instead of a tupling function
--
-- zipWith (,) [1, 2, 3] [4, 5, 6]
-- [(1,4),(2,5),(3,6)]
-}
zipWith                    :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith _ [] _             = []
zipWith _ _ []             = []
zipWith f (x:xs) (y:ys) = (f x y) : zipWith f xs ys
```

Resources

- Data.List
  http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-List.html
- *Source* Data.List
  http://hackage.haskell.org/packages/archive/base/latest/doc/html/src/Data-List.html