```haskell
head :: [a] -> a                  -- returns the first element
last :: [a] -> a                  -- returns the last element
tail :: [a] -> [a]                -- returns everything except the first element
init :: [a] -> [a]                -- returns everything except the last element
take :: Int -> [a] -> [a]         -- returns the n first elements
drop :: Int -> [a] -> [a]         -- returns everything except the n first elements
(++) :: [a] -> [a] -> [a]         -- lists are catenated with the ++ operator
(!!) :: [a] -> Int -> a           -- lists are indexed with the !! operator
reverse :: [a] -> [a]             -- reverse a list
null :: [a] -> Bool               -- is this list empty?
length :: [a] -> Int              -- the length of a list
```

***Imutabilitate - in HASKELL functiile nu pot modifca Inputurile (doar returneaza o valoare a inputurilor modificate!)

```
Prelude> list = [1,2,3,4]
Prelude> reverse list
[4,3,2,1]
Prelude> list
[1,2,3,4]
Prelude> drop 2 list
[3,4]
Prelude> list
[1,2,3,4]
```

Exista tipul Maybe introduce conceptul de TIP PARAMETRIZAT

- De obicei folosit in gestionarea situatiilor cand o operatie nu returneaza ceva valid
- Constructori : Nothing, Just

You use a `Maybe` value by pattern matching on it. Usually you define patterns for the `Nothing` and `Just something` cases. Some examples:

```haskell
-- Multiply an Int with a Maybe Int. Nothing is treated as no multiplication at all.
perhapsMultiply :: Int -> Maybe Int -> Int
perhapsMultiply i Nothing = i
perhapsMultiply i (Just j) = i*j   -- Note how j denotes the value inside the Just
```

```
Prelude> perhapsMultiply 3 Nothing
3
Prelude> perhapsMultiply 3 (Just 2)
6
```

Constructori:

Maybe: Nothing, Just

Bool: True, False

Nota! Constructorii care nu iau un parametru (e.g. Just a), sunt constante”: **Nothing, True, False**. Pe cand cei care iau un parametru (e.g. Just a) se comporta ca niste functii.

```
ghci> :t Just
Just :: a -> Maybe a
ghci> :t Nothing
Nothing :: Maybe a
```

Either: Left, Right

`Either Int Bool` `Left 0`, `Left 1`, `Right False`, `Right True`,

`Either Integer Integer` `Left 0`, `Right 0`, `Left 1`, `Right 1`,

Infix/Prefix

Un infix operator poate fi apelat ca o functie daca il punem intre paranteze

`(+) 1 2 ==> 1 + 2 ==> 3`

As an example, the function `zipWith` takes two lists, a binary function, and joins the lists using the function. We can use `zipWith (+)` to sum two lists, element-by-element:

```
Prelude> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
Prelude> zipWith (+) [0,2,5] [1,3,3]

[1,5,8]
```

Aplicarea unei functii binare asemenea unui operator infix -> folosim `map`- backtics

```
6 `div` 2 ==> div 6 2 ==> 3
(+1) `map` [1,2,3] ==> map (+1) [1,2,3] ==> [2,3,4]
```

LAMBDAS:

\ -> inseamna litera lambda

```
Prelude> filter (\x -> reverse x == x)
["ABBA","ACDC","otto","lothar","anna"]
output:
["ABBA","otto","anna"]

(\x y -> x^2+y^2) 2 3            -- multiple arguments
13
```

Operatori (.), ($)

Adesea folositi pentru functii care iau ca argument alte functii (high order functions)

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

double x = 2*x
quadruple = double . double   -- computes 2*(2*x) == 4*x
f = quadruple . (+1)          -- computes 4*(x+1)
g = (+1) . quadruple          -- computes 4*x+1
third = head . tail . tail    -- fetches the third element of a list
```

Tipuri recursive:

Definim o lista de Int

```haskell
data IntList = Empty | Node Int IntList
                deriving Show

ihead :: IntList -> Int
ihead (Node i _) = i

itail :: IntList -> IntList
itail (Node _ t) = t

ilength :: IntList -> Int
ilength Empty = 0
ilength (Node _ t) = 1 + ilength t


Prelude> ihead (Node 3 (Node 5 (Node 4 Empty)))
3
Prelude> itail (Node 3 (Node 5 (Node 4 Empty)))
Node 5 (Node 4 Empty)
Prelude> ilength (Node 3 (Node 5 (Node 4 Empty)))
3
```

O lista de elemente de oricare tip:

```haskell
data List a = Empty | Node a (List a)
               deriving Show

lhead :: List a -> a
lhead (Node h _) = h

ltail :: List a -> List a
ltail (Node _ t) = t

lnull :: List a -> Bool
lnull Empty = True
lnull _     = False
```

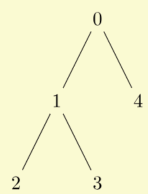
```
llength :: List a -> Int
llength Empty = 0

llength (Node _ t) = 1 + llength t
```

Construim un arbore:

```
data Tree a = Node a (Tree a) (Tree a) | Empty
```

In case you're not familiar with binary trees, they're a data structure that's often used as the basis for other data structures (`Data.Map` is based on trees!). Binary trees are often drawn as (upside-down) pictures, like this:




```
example :: Tree Int
example = (Node 0 (Node 1 (Node 2 Empty Empty)
                          (Node 3 Empty Empty))
                  (Node 4 Empty Empty))

treeHeight :: Tree a -> Int
treeHeight Empty = 0
treeHeight (Node _ l r) = 1 + max (treeHeight l) (treeHeight r)

lookup :: Int -> Tree Int -> Bool
lookup x Empty = False
lookup x (Node y l r)
  | x < y = lookup x l
  | x > y = lookup x r
  | otherwise = True
```

```haskell
insert :: Int -> Tree Int -> Tree Int
insert x Empty = Node x Empty Empty
insert x (Node y l r)
  | x < y = Node y (insert x l) r
  | x > y = Node y l (insert x r)
  | otherwise = Node y l r
```