Haskell

A functional paradigm exemplar and other aspects

Ana Oprescu, Thomas van Binsbergen

UvA

March 2, 2023

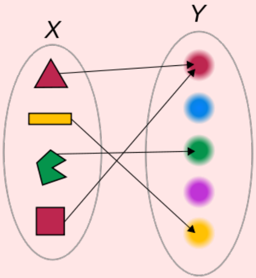
Haskell

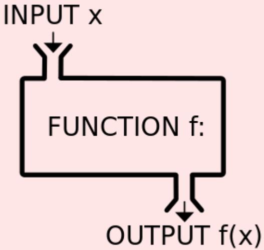
- Named after Haskell Curry
- The 90s
- Functional and declarative: you define functions (and constants) and data.. that's it
- Pure: no side-effects
- Lazy: call-by-need (= call-by-name + caching)
- Strong and statically typed
- Space/indentation sensitive (no curly braces needed!)

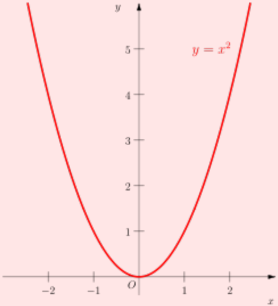
Functional programming

- Functional: write programs by defining and composing (mathematical) functions
- Functions operate on data/values (and as we shall see, functions as well)

Wikipedia examples of functions







- Math: a function is a binary relation with a mapping for every ‘input’ (first component)

Functioneel programmeren – Terminologie

Terminologie

Definieer *functies* welke gegeven een invoer (mogelijk) een bepaalde uitkomst opleveren. Functie definities bestaan uit *clausules* welke met behulp van *patroonherkenning* en *gevalsanalyse* bepalen welke *expressie* de uitkomst van een *functieaanroep* bepaalt.

```haskell
data BinIntTree = Node BinIntTree BinIntTree
                | Leaf Int

sum_tree :: BinIntTree -> Int
sum_tree (Leaf i)   = i                         -- eerste clausule
sum_tree (Node l r) = sum_tree l + sum_tree r -- tweede clausule
```

Functioneel programmeren – Leidraad

Leidraad

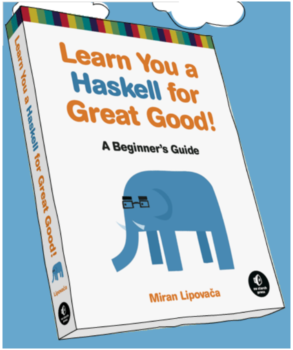
Maak een decompositie van het probleem door types en functies te introduceren voor het *oplossen van deelproblemen*. Een *compositie van deeloplossingen* geeft de eindoplossing.

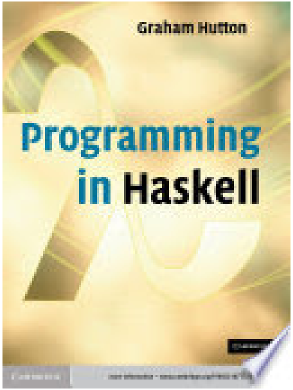
```haskell
type Sudoku = (Row,Column) -> Value
type Row    = Int
type Column = Int
type Value  = Int
type Grid   = [[Value]]

sud2grid :: Sudoku -> Grid
sud2grid = ...
```

```haskell
printSudoku :: Sudoku -> IO ()
printSudoku =
    putStr               -- print string
  . unlines              -- rows to string
  . map ( unwords        -- cells to row
        . map show)      -- cells to strings
  . sud2grid             -- sudoko to grid
```

Learning Haskell – recommended reading

- Learn You a Haskell for Great Good! **Some images and code examples in this presentation come from the book.** [1]
- Programming in Haskell [2]





[1] http://learnyouahaskell.com/chapters

[2] http://www.cs.nott.ac.uk/~pszgmh/pih.html

Contents

Starting Haskell

- Glasgow Haskell Compiler (GHC), installed as part of the 'Haskell Platform'
- Gives you a compiler (`ghc`) and an interpreter (`ghci`)
- Or experiment online: `https://repl.it/languages/haskell`

Open a terminal and start the interactive compiler **ghci**

```
$ ghci
GHCi, version 8.4.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Simple expressions

- `Prelude> 5+6`
  `11`
- `Prelude> 3.5*3`
  `10.5`
- `Prelude> 4/3`
  `1.3333333333333333`
- `Prelude> 2**8`
  `256.0`
- `Prelude> 8/2`
  `4.0`
- `Prelude> 3*3`
  `9`
- `Prelude> 2-4`
  `-2`
- `Prelude> True`
  `True`

Simple expressions (2)

- `Prelude> False`
  `False`
- `Prelude> 3 - 6 == -3`
  `True`
- `Prelude> 4 + 4 /= 8  -- 'not equal'-operator (not an assignment)`
  `False`
- `Prelude> True & True`
  `<interactive>:16:6: Not in scope: ` `` `&' ``
- `Prelude> True && True`
  `True`
- `Prelude> True || False`
  `True`
- `Prelude> / True`
  `<interactive>:19:1: parse error on input ` `` `/' ``
- `Prelude> not True`
  `False`

Variables

Haskell variables are *mathematical variables*; they are not mutable (but can be overwritten).

Example of shadowing

Definitions are not assignments!

```
Prelude> let foo = 3
Prelude> let foo = 6
Prelude> foo    {prints 6}
```

```haskell
let foo = 3
  in let foo = 6
      in print foo -- prints 6
```

Variables

Haskell variables are *mathematical variables*; they are not mutable (but can be overwritten).

Example of shadowing

Definitions are not assignments!

```
Prelude> let foo = 3
Prelude> let foo = 6
Prelude> foo    {prints 6}
```

```haskell
let foo = 3
  in let foo = 6
      in print foo -- prints 6
```

Common pitfalls

In Haskell, **let** is recursive:

```
Prelude> let foo = 3
Prelude> let foo = foo + 3
Prelude> foo     {loops}
```

```haskell
let foo = 3
  in let foo = foo + 3
      in print foo -- loops
```

Modules

FooModule.hs

```haskell
module FooModule where
-- Here variable bar is given value 4.
bar = 4
```

1. Header
2. Imports
3. Top-level function definitions

Modules

FooModule.hs

```haskell
module FooModule where
-- Here variable bar is given value 4.
bar = 4
```

1. Header
2. Imports
3. Top-level function definitions

```
Prelude> :load FooModule.hs
[1 of 1] Compiling FooModule  ( FooModule.hs, interpreted )
Ok, modules loaded: FooModule.

*FooModule> let foo = 3
*FooModule> foo + bar
7
```

Our own addition function

Lecture.hs

```haskell
module Lecture where

add :: Integer -> Integer -> Integer
add a b = a + b
```

or directly in the interpreter:

```haskell
let add a b = a + b
```

Prefix and infix functions

Infix functions

```
+ - * / ** && || == /=
```

Infix functions can be used as prefix functions:

```
*Lecture> (+) 2 3
5
```

Prefix functions – most user-defined and Prelude functions

```
not, odd, even, mod, add, ...
```

Prefix functions with two arguments can be used as infix functions:

```
*Lecture> add 2 3
5
*Lecture> 2 `add` 3
5
```

Content

Lists

- Empty list:

```
Prelude> []
[]
```

- List with 2 elements:

```
Prelude> [1,2]
[1,2]
```

- Create a list by adding an element to the front of a list:

```
Prelude> 1 : [2,3]
[1,2,3]
```

- Internal representation:

```
Prelude> 1 : 2 : 3 : []
[1,2,3]
```

Lists (2)

- List concatenation:

```
Prelude> [1,2] ++ [3,4]
[1,2,3,4]
```

- List of characters:

```
Prelude> ['a','b','c']
"abc"
```

- String concatenation:

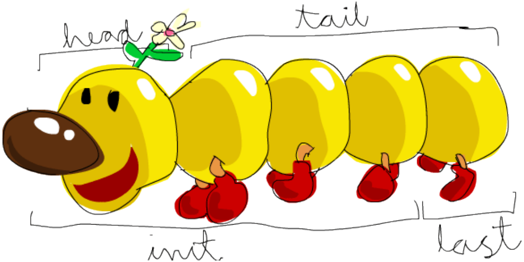
```
Prelude> "Hello" ++ " " ++ "World"
"Hello World"
```

- Length of a list:

```
Prelude> length [1,2,3,4,5,6]
6
```

Lists: head, last, init and tail

- `Prelude> head [1,2,3,4,5]`
  `1`
- `Prelude> last [1,2,3,4,5]`
  `5`
- `Prelude> tail [1,2,3,4,5]`
  `[2,3,4,5]`
- `Prelude> init [1,2,3,4,5]`
  `[1,2,3,4]`



Lists: reverse, !!, null, take

- Reverse a list

```
Prelude> reverse [1,2,3,4,5]
[5,4,3,2,1]
```

- Retrieve the *n*-th element of a list (be careful, function is partial!)

```
Prelude> [1,2,3,4,5] !! 3
4
```

- Check whether a list is empty

```
Prelude> null []
True
Prelude> null [1,2,3]
False
```

- Retrieve the first *n* elements of a list

```
Prelude> take 3 [1,2,3,4,5,6]
[1,2,3]
```

Pattern matching lists

```
Prelude> let (x:xs) = [1,2,3,4]
Prelude> x
1
Prelude> xs
[2,3,4]
```

Lecture.hs

```haskell
my_length :: [a] -> Integer
my_length []     = 0
my_length (x:xs) = 1 + my_length xs
```

Our own reverse function

Can we write our own `reverse` function?

Our own reverse function

Can we write our own `reverse` function?

Simple recursion

```
my_reverse :: [a] -> [a]
my_reverse [] = []
my_reverse (x:xs) = my_reverse xs ++ [x]
```

Our own reverse function

Can we write our own `reverse` function?

Simple recursion

```haskell
my_reverse :: [a] -> [a]
my_reverse [] = []
my_reverse (x:xs) = my_reverse xs ++ [x]
```

With an accumulator (and using ‘tail recursion’)

```haskell
my_reverse :: [a] -> [a]
my_reverse s = my_reverse' s []

my_reverse' :: [a] -> [a] -> [a]
my_reverse' []     acc = acc
my_reverse' (x:xs) acc = my_reverse' xs (x:acc)
```

Tuples

Tuples can hold multiple values of different types, however they have a finite length.

- Tuple with 2 Integers:

```
Prelude> (1,2)
(1,2)
```

- Tuple with an Integer and a Character:

```
Prelude> (1,'a')
(1,'a')
```

Lists versus Tuples

Lists	**Tuples**
Homogeneous	Heterogeneous
Variable length	Fixed length (per type)
May be infinite	Always finite

```
Prelude> :t [1, 'a']
<interactive>:16:2: error:
    • No instance for (Num Char) arising from the literal ‘1’
    • In the expression: 1
      In the expression: [1, 'a']
      In an equation for ‘it’: it = [1, 'a']
Prelude> :t (1, 'a')
(1, 'a') :: Num a => (a, Char)
```

Tuples (2)

- List of two tuples:

```
Prelude> [(1,2),(3,4)]
[(1,2),(3,4)]
```

Tuples (2)

- List of two tuples:

```
Prelude> [(1,2),(3,4)]
[(1,2),(3,4)]
```

- List of two different tuples: [(1,2),('a',4)]

Tuples (2)

- List of two tuples:

```
Prelude> [(1,2),(3,4)]
[(1,2),(3,4)]
```

- List of two different tuples: [(1,2),('a',4)]

Tuples (2)

- List of two tuples:

```
Prelude> [(1,2),(3,4)]
[(1,2),(3,4)]
```

- List of two different tuples: [(1,2),('a',4)]

```
Prelude> [(1,2),('a',4)]
<interactive>:7:3:
    No instance for (Num Char) arising from the literal `1'
    Possible fix: add an instance declaration for (Num Char)
    In the expression: 1
    In the expression: (1, 2)
    In the expression: [(1, 2), ('a', 4)]
```

Tuples: fst, snd

- First element of a tuple with 2 elements:

```
Prelude> fst (1,2)
1
```

- Second element of a tuple with 2 elements:

```
Prelude> snd (3,'d')
'd'
```

- First element of a tuple with 3 elements: fst (1,2,3)

```
Prelude> fst (1,2,3)

<interactive>:8:5:
    Couldn't match expected type `(a0, b0)'
                with actual type `(t0, t1, t2)'
    In the first argument of `fst', namely `(1, 2, 3)'
    In the expression: fst (1, 2, 3)
    In an equation for `it': it = fst (1, 2, 3)
```

Tuples: thrd function

Can we write a function that returns the third element of a tuple with 3 elements?

Tuples: thrd function

Can we write a function that returns the third element of a tuple with 3 elements?

Lecture.hs

```haskell
thrd :: (a,b,c) -> c
thrd (x,y,z) = z
```

```
*Lecture> thrd (1,2,3)
3
*Lecture> thrd ('a','b','c')
'c'
*Lecture> fst (1,2,3)          -- what happens when we try this?
```

Tuples: thrd function

Can we write a function that returns the third element of a tuple with 3 elements?

Lecture.hs

```haskell
thrd :: (a,b,c) -> c
thrd (x,y,z) = z
```

```
*Lecture> thrd (1,2,3)
3
*Lecture> thrd ('a','b','c')
'c'
*Lecture> fst (1,2,3)          -- what happens when we try this?
```

```
error:
    * Couldn't match expected type: (a, b0)
                with actual type: (a0, b1, c0)
    * In the first argument of ‘fst’, namely ‘(1, 2, 3)’
      ...
```

Content

Pattern matching

Lecture.hs

```haskell
lucky :: Integer -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck!"
```

```
*Lecture> lucky 7
"LUCKY NUMBER SEVEN!"
*Lecture> lucky 4
"Sorry, you're out of luck!"
```

Ignoring arguments

We can match arbitrary values without naming them using the ‘wildcard’ _
This is recommended over naming a variable without using it:

Lecture.hs

```haskell
lucky' :: Integer -> String
lucky' 7 = "LUCKY NUMBER SEVEN!"
lucky' _ = "Sorry, you're out of luck, pal!"
```

```
*Lecture> lucky'  7
"LUCKY NUMBER SEVEN!"
*Lecture> lucky' 4
"Sorry, you're out of luck, pal!"
```

If-then-else

Lecture.hs

```haskell
describeLetter :: Char -> String
describeLetter c =
    if c >= 'a' && c <= 'z'
        then "Lower case"
        else if c >= 'A' && c <= 'Z'
            then "Upper case"
            else "No ASCII letter"
```

```
*Lecture> describeLetter '1'
"No ASCII letter"
*Lecture> describeLetter 'a'
"Lower case"
*Lecture> describeLetter 'A'
"Upper case"
```

Guards

We can use "guards" to avoid if-spaghetti:

Lecture.hs

```haskell
describeLetter' :: Char -> String
describeLetter' c | c >= 'a' && c <= 'z' = "Lower case"
                  | c >= 'A' && c <= 'Z' = "Upper case"
                  | otherwise            = "No ASCII letter"
```

```
*Lecture> describeLetter' '1'
"No ASCII letter"
*Lecture> describeLetter' 'a'
"Lower case"
*Lecture> describeLetter' 'A'
"Upper case"
```

Where and let clauses

We can use **let** and **where** to improve code readability. However, please consult `https://wiki.haskell.org/Let_vs._Where` for a discussion.

Lecture.hs

```haskell
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
                              where (f:_) = firstname
                                    (l:_) = lastname


initials' :: String -> String -> String
initials' firstname lastname = let (f:_) = firstname
                                   (l:_) = lastname
                               in  [f] ++ ". " ++ [l] ++ "."
```

```
*Lecture> initials "John" "Doe"
"J. D."
```

Our own reverse function with **where**

With an accumulator (and using 'tail recursion')

```haskell
my_reverse :: [a] -> [a]
my_reverse s = my_reverse' s []
 where
  my_reverse' :: [a] -> [a] -> [a]
  my_reverse' []     s = s
  my_reverse' (x:xs) s = my_reverse' xs (x:s)
```

Anonymous functions

Anonymous functions

An anonymous function is a written as a ‘lambda abstraction’ and looks like: `(\x -> ...)`

Anonymous functions

Anonymous functions

An anonymous function is a written as a 'lambda abstraction' and looks like: `(\x -> ...)`

```
Prelude> (\x -> x + 1) 1
2
Prelude> (\a b -> a + b) 3 5
8
Prelude> (\_ b -> b) 17 42
42
```

Function composition

Function composition is the act of pipelining the result of one function, to the input of another, creating an entirely new function.
Mathematically, this is most often represented by the $\circ$ operator, where $f \circ g$ (often read as f of g) is the composition of f with g (perhaps counter-intuitively, g is applied first).

Function composition (found in Prelude)

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

```
Prelude> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
Prelude> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
Prelude> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
Prelude> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

Explicit function application

Normally **function application** is written by 'juxtaposition' – writing the function and argument side-by-side. However, the $ operator is an infix operator for function application.

Function application (found in Prelude)

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

Sometimes it can help you write an expression more comfortably, with less (nested) parentheses, but its usage is a bit controversial:

```
-- function application is 'left-associative'
Prelude> add 1 add 2 add 3 4 -- equivalent to ((((((add 1) add) 2) add) 3) 4)
<COMPLICATED TYPE ERROR>
Prelude> add 1 (add 2 (add 3 4))
10
Prelude> add 1 $ add 2 $ add 3 4
10
```

Infix operator definitions

The built-in operators have a precedence (or priority) and an associativity constraint:

```
Prelude> :info (.)
(.) :: (b -> c) -> (a -> b) -> a -> c  -- Defined in ‘GHC.Base’
infixr 9 .   -- right-associative, highest precedence
```

```
Prelude> :info ($)
($) :: (a -> b) -> a -> b  -- Defined in ‘GHC.Base’
infixr 0 $   -- right-associative, lowest precedence
Prelude>
```

With `infix`, `infixl` and `infixr` you can create your own infix operators.

Currying

We apply functions one argument at a time:

Lecture.hs

```haskell
add :: Integer -> Integer -> Integer
add a b = a + b
```

```haskell
let three     = add 1 2
let plus_one  = add 1
let three'    = plus_one 2
```

Currying

We apply functions one argument at a time:

Lecture.hs

```haskell
add :: Integer -> Integer -> Integer
add a b = a + b
```

```haskell
let three      = add 1 2
let plus_one   = add 1
let three'     = plus_one 2
```

```
*Lecture> map (add 3) [1,2,3,4]
[4,5,6,7]
```

Currying

We apply functions one argument at a time:

Lecture.hs

```haskell
add :: Integer -> Integer -> Integer
add a b = a + b
```

```haskell
let three     = add 1 2
let plus_one  = add 1
let three'    = plus_one 2
```

```
*Lecture> map (add 3) [1,2,3,4]
[4,5,6,7]
```

Also works for anonymous functions (be careful with parentheses!)

```
*Lecture> map ((\a b -> a + b) 3) [1,2,3,4]
[4,5,6,7]
```

Currying and higher-order functions

In Haskell, a function can receive a function as an argument and can return a function as a result. Such functions are known as *higher-order functions*.

Further reading: `http://learnyouahaskell.com/higher-order-functions`

In functional languages without lazy evaluation (such as ML and Scheme), currying is more cumbersome as the programmer needs to be aware of evaluation order.

Higher-order functions have become commonplace also in non-functional languages (e.g. in Java and Python), although the combination of currying and lazy evaluation makes higher-order functions much easier to use in Haskell

Content

Ranges

Problem: we want a list with all the integers between 1 and 15.

```
Prelude> [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Ranges

Problem: we want a list with all the integers between 1 and 15.

```
Prelude> [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

Solution: ranges!

- `Prelude> [1..15]`
  `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]`
- `Prelude> [2,4..20]`
  `[2,4,6,8,10,12,16,18,20]`
- `Prelude> [3,6..20]`
  `[3,6,9,12,15,18]`
- `Prelude> [0.1, 0.3 .. 1]`
  `[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]`
  Unexpected, different behaviour due to different types (see `https://stackoverflow.com/questions/7290438/haskell-ranges-and-floats` for an explanation).

Infinite lists

Because Haskell is lazy, we can construct infinite lists. Haskell will not fully evaluate the list unless you ask for it.

- `Prelude> [1..]`
  `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,..`
- `Prelude> take 5 [1..]`
  `[1,2,3,4,5]`
- `Prelude> take 10 $ cycle [1,2,3]`
  `[1,2,3,1,2,3,1,2,3,1]`
- `Prelude> take 10 $ repeat 1`
  `[1,1,1,1,1,1,1,1,1,1]`
- `Prelude> tail [1..]    -- what happens here?`

Infinite lists

Because Haskell is lazy, we can construct infinite lists. Haskell will not fully evaluate the list unless you ask for it.

- `Prelude> [1..]`
  `[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,..`
- `Prelude> take 5 [1..]`
  `[1,2,3,4,5]`
- `Prelude> take 10 $ cycle [1,2,3]`
  `[1,2,3,1,2,3,1,2,3,1]`
- `Prelude> take 10 $ repeat 1`
  `[1,1,1,1,1,1,1,1,1,1]`
- `Prelude> tail [1..]     -- what happens here?`
  `[2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,`

List comprehensions

$$S = \{x^2 \mid x \in \mathbb{N}, x \leq 10, x \,\%\, 2 = 0\}$$

List comprehensions

$S = \{x^2 \mid x \in \mathbb{N}, x \leq 10, x \,\%\, 2 = 0\}$

In Haskell:

```
Prelude> [x * x | x <- [1..10], x `mod` 2 == 0]
[4,16,36,64,100]
```

List comprehensions

$S = \{x^2 \mid x \in \mathbb{N}, x \leq 10, x \,\%\, 2 = 0\}$

In Haskell:

```
Prelude> [x * x | x <- [1..10], x `mod` 2 == 0]
[4,16,36,64,100]
```

A Pythagorean triple consists of three positive integers a, b and c, such that $a^2 + b^2 = c^2$:

```
Prelude> [(a,b,c) | a <- [1..20], b <- [1..20], c <- [1..20]
                    , a*a + b*b == c*c]
[(3,4,5),(4,3,5),(5,12,13),(6,8,10),(8,6,10),(8,15,17),
(9,12,15),(12,5,13),(12,9,15),(12,16,20),(15,8,17),(16,12,20)]
```

Sieve of Eratosthenes

Wikipedia

The sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite (i.e. not prime) the multiples of each prime, starting with the multiples of 2.

Lecture.hs

```haskell
primes :: [Integer]
primes = sieve [2..]
 where sieve (p:xs) = p : sieve [x | x<-xs, x `mod` p /= 0]
```

```
*Lecture> take 15 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

Content

Filter

Remove all elements from a list for which a given function returns False.

Filter (found in Prelude)

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ []              = []
filter pred (x:xs)
     | pred x            = x : filter pred xs
     | otherwise         = filter pred xs
```

```
Prelude> filter odd [1,2,3,4,5]
[1,3,5]
Prelude> filter (\x -> x > 3) [1,2,3,4,5]
[4,5]
```

Sieve of Eratosthenes (returned)

```haskell
primes = sieve [2..]
  where sieve (p:xs) = p : sieve (filter not_prime_multiple xs)
          where not_prime_multiple x = x `mod` p /= 0
```

Map

Apply a given function to each element of a list.

Map (found in Prelude)

```haskell
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

```
Prelude> map odd [1,2,3,4,5]
[True,False,True,False,True]
Prelude> map (\x -> x + 3) [1,2,3,4,5]
[4,5,6,7,8]
```

Foldr

- a binary operator, taking a list element and an 'accumulator',
- a starting value for the accumulator,
- and a list

The list is 'reduced to a value' using the binary operator, from right to left.

Definition of foldr (found in Prelude)

```
foldr                :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc []       =  acc
foldr f acc (x:xs)   =  f x (foldr f acc xs)
```

```
foldr op acc [1,2,3,4,5] == 1 `op` (2 `op` (3 `op` (4 `op` (5 `op` acc))))
```

Foldr

- a binary operator, taking a list element and an 'accumulator',
- a starting value for the accumulator,
- and a list

The list is 'reduced to a value' using the binary operator, from right to left.

Definition of foldr (found in Prelude)

```haskell
foldr               :: (a -> b -> b) -> b -> [a] -> b
foldr _ acc []      =  acc
foldr f acc (x:xs)  =  f x (foldr f acc xs)
```

```
foldr op acc [1,2,3,4,5] == 1 `op` (2 `op` (3 `op` (4 `op` (5 `op` acc))))
```

```
Prelude> foldr (:) [] [1,2,3,4,5]
[1,2,3,4,5]
Prelude> foldr (+) 0 [1,2,3,4,5]
15
Prelude> foldr (\_ acc -> 1+acc) 0 [1,2,3,4,5,6,7]
7
```

Foldl

- a binary operator, taking an accumulator and a list element ,
- a starting value for the accumulator,
- and a list

The list is ‘reduced to a value’ using the binary operator, from left to right:

Definition of foldl (found in Prelude)

```haskell
foldl              :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []     = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

```haskell
foldl op acc [1,2,3,4,5] == ((((acc `op` 1) `op` 2) `op` 3) `op` 4) `op` 5
```

Foldl

- a binary operator, taking an accumulator and a list element ,
- a starting value for the accumulator,
- and a list

The list is ‘reduced to a value’ using the binary operator, from left to right:

Definition of foldl (found in Prelude)

```haskell
foldl                :: (b -> a -> b) -> b -> [a] -> b
foldl f acc []       = acc
foldl f acc (x:xs)   = foldl f (f acc x) xs
```

```haskell
foldl op acc [1,2,3,4,5] == ((((acc `op` 1) `op` 2) `op` 3) `op` 4) `op` 5
```

```
Prelude> foldl (\acc _ -> 1+acc) 0 [1..10]
10
Prelude> foldl (flip (:)) [] [1,2,3,4,5]
[5,4,3,2,1]
```

Zip

'zip' takes two lists and returns a list of corresponding pairs (tuples of 2 elements). If one input list is short, excess elements of the longer list are discarded.

Zip (found in Prelude)

```haskell
zip :: [a] -> [b] -> [(a,b)]
zip (a:as) (b:bs) = (a,b) : zip as bs
zip _      _      = []
```

```
Prelude> zip [1,2,3] [4,5,6]
[(1,4),(2,5),(3,6)]
Prelude> zip [1,2,3,4,5] ['a','b','c','d','e']
[(1,'a'),(2,'b'),(3,'c'),(4,'d'),(5,'e')]
Prelude> import Data.IntMap
Prelude Data.IntMap> fromList $ zip [1..] "hello world!"
fromList [(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o'),(6,' '),(7,'w'),(8,'o'),(9,
```

ZipWith (found in Prelude)

'zipWith' generalises 'zip' by zipping with the function given as the first argument, instead of a tupling function.

ZipWith (found in Prelude)

```
zipWith :: (a->b->c) -> [a]->[b]->[c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _      _      = []
```

```
Prelude> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
Prelude> zipWith (*) [1,2,3] [4,5,6]
[4,10,18]
```

Content

Tips when using GHCi

Show the type of a function or variable:

```
Prelude> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Show type for all subsequent commands:

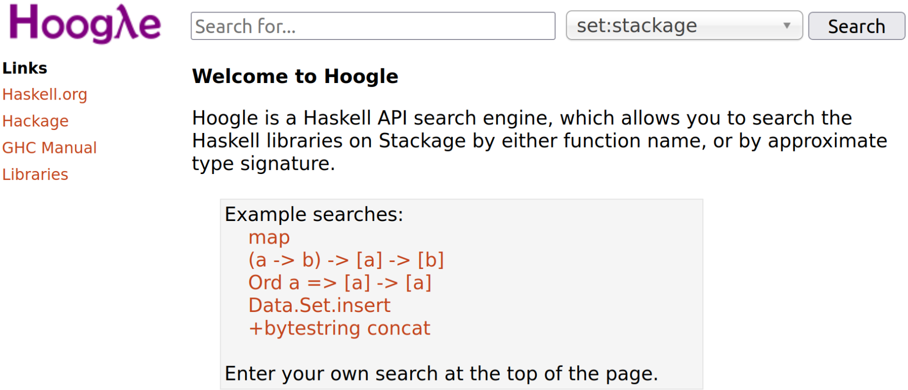
```
Prelude> :set +t
Prelude> 1
1
it :: Integer
Prelude> filter even [1..20]
[2,4,6,8,10,12,14,16,18,20]
it :: [Integer]
```

Disable:

```
Prelude> :unset +t
```

Your new best friend: Hoogle

Hoogle is a 'search engine' for Haskell functions available in the Prelude or in other packages: `https://hoogle.haskell.org/`



Remember, operators are also functions

Lecture.hs

```haskell
neg :: Integer -> Integer
neg a = - a
```

```
Prelude> neg 3
-3
Prelude> neg -3
<interactive>:2:1: error:
    • No instance for (Num (Integer -> Integer)) arising from a use of ‘-’
    • In the expression: neg - 3
```

Remember, operators are also functions

Lecture.hs

```haskell
neg :: Integer -> Integer
neg a = - a
```

```
Prelude> neg 3
-3
Prelude> neg -3
<interactive>:2:1: error:
    • No instance for (Num (Integer -> Integer)) arising from a use of ‘-’
    • In the expression: neg - 3
```

Solution: parentheses or $:

```
Prelude>
Prelude> neg (-3)
3
Prelude> neg $ -3
3
```

Why functional programming matters

- John Hughes, creator of QuickCheck and member of the committee designing Haskell says in 1989[3]

  > *Higher-order functions and lazy evaluation can contribute greatly to modularity. Since modularity is the key to successful programming, functional languages are vitally important to the real world.*

[3] https://academic.oup.com/comjnl/article/32/2/98/543535
[4] https://academic.oup.com/nsr/article/2/3/349/1427872
[5] http://infolab.stanford.edu/~olston/publications/scicloud11.pdf

Why functional programming matters

- John Hughes, creator of QuickCheck and member of the committee designing Haskell says in 1989[3]

  > *Higher-order functions and lazy evaluation can contribute greatly to modularity. Since modularity is the key to successful programming, functional languages are vitally important to the real world.*

Leidraad

Maak een decompositie van het probleem door types en functies te introduceren voor het *oplossen van deelproblemen*. Een *compositie van deeloplossingen* geeft de eindoplossing.

[3] https://academic.oup.com/comjnl/article/32/2/98/543535
[4] https://academic.oup.com/nsr/article/2/3/349/1427872
[5] http://infolab.stanford.edu/~olston/publications/scicloud11.pdf

Why functional programming matters

- John Hughes, creator of QuickCheck and member of the committee designing Haskell says in 1989[3]

  > *Higher-order functions and lazy evaluation can contribute greatly to modularity. Since modularity is the key to successful programming, functional languages are vitally important to the real world.*

Leidraad

Maak een decompositie van het probleem door types en functies te introduceren voor het *oplossen van deelproblemen*. Een *compositie van deeloplossingen* geeft de eindoplossing.

- in 2015, John Hughes looked back at whether functional programming really mattered [4]
  - lambda expressions permeated many mainstream languages (C++, Java)
  - lazy evaluation generated interest in academia and industry, e.g., leveraged in MapReduce[5]

[3] https://academic.oup.com/comjnl/article/32/2/98/543535

[4] https://academic.oup.com/nsr/article/2/3/349/1427872

[5] http://infolab.stanford.edu/~olston/publications/scicloud11.pdf