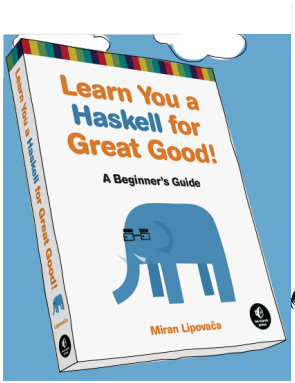
Inhoud

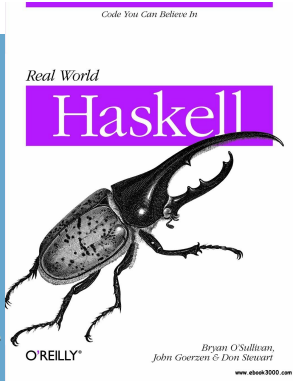
Haskell

- Vernoemd naar Haskell Curry
- Verschenen in 1990
- Puur functioneel
- Lui
- Sterk en statisch getypeerd

Haskell leren

- Learn You a Haskell for Great Good![1] `http://learnyouahaskell.com/chapters`
- Real World Haskell `http://book.realworldhaskell.org/read`
- A Gentle Introduction to Haskell '98 `https://www.haskell.org/tutorial`





[1] Sommige voorbeelden in deze slides zijn afkomstig van dit e-book.

Haskell starten

Open een terminal en start de interactieve compiler met **ghci**

```
robin@robin-pc ~ $ ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ ?: for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

Basis rekenen

- `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`

Basis rekenen (2)

- `Prelude> False`
  `False`
- `Prelude> 3 - 6 == -3`
  `True`
- `Prelude> 4 + 4 /= 8`
  `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`

Variabelen, commentaar, bestanden en modules

```
Prelude> let foo = 3
Prelude> foo + 3
6
```

FooModule.hs

```
module FooModule where
-- Hier wordt de waarde 4 toegekend aan de bar variabele.
bar = 4
```

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

*FooModule> foo + bar
7
```

Functies

Hoeveel functies heb je gezien?

Prefix en infix functies

Prefix functies

```
not
```

Prefix functies met twee argumenten kun je gebruiken als infix functie:

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

Infix functies

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

Infix functies kun je gebruiken als prefix functies d.m.v. haakjes:

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

Een eigen optel-functie

Lecture.hs

```
module Lecture where

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

of direct in de interpreter:

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

Wat doet deze functie?

Lecture.hs

```
foo :: Integer -> Integer
foo 0 = 1
foo n = n * foo (n-1)
```

```
*Lecture> foo 0
1
*Lecture> foo 3
6
```

Inhoud

Lijsten

- Lege lijst:

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

- Lijst met 2 elementen:

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

- Een element toevoegen aan een lijst:

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

- Interne representatie:

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

Lijsten (2)

- Lijst concatenatie:

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

- Lijst van karakters:

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

- String concatenatie:

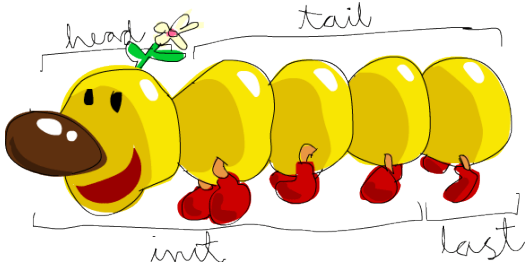
```
Prelude> "Hello" ++ " " ++ "World"
"Hello World"
```

- Lengte van een lijst:

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

Lijsten: head, last, init en 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]`



Lijsten: reverse, !!, null, take

- Draai een lijst om

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

- Pak het *n*-de element van een lijst

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

- Kijk of een lijst leeg is

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

- Pak de eerste *n* elementen van een lijst

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

Lijsten uitpakken

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

Lecture.hs

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

Onze eigen reverse functie

Kunnen we onze eigen `reverse` functie schrijven? Natuurlijk!

Lecture.hs

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

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

Tuples

Tuples kunnen meerdere waarde van verschillende types bevatten, maar hebben altijd een vaste lengte.

- Tuple met 2 Integers:

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

- Tuple met een Integer en een Character:

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

Tuples (2)

- Lijst van twee tuples:

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

- Lijst van twee verschillende tuples:

```
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

- Het eerste element van een tuple met 2 elementen:

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

- Het tweede element van een tuple met 2 elementen:

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

- Het eerste element van een tuple met 3 elementen:

```
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 functie

Kunnen we een functie schrijven dat het derde element van een tuple met 3 elementen teruggeeft?

Lecture.hs

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

```
*Lecture> thrd (1,2,3)
3
*Lecture> thrd ('a','b','c')
'c'
```

Inhoud

Pattern matching

Lecture.hs

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

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

Argumenten negeren

Met behulp van _ kun je een argument negeren:

Lecture.hs

```
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

```
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

Je kunt if-spaghetti voorkomen met behulp van “guards”:

Lecture.hs

```
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 en let clauses

Met behulp van `let` en `where` kun je de leesbaarheid van je code verbeteren.

Lecture.hs

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

Inhoud

Ranges

Probleem: we willen een lijst met alle gehele getallen tussen de 1 en 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]
```

Oplossing: 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]`
  `[0.1,0.3,0.5,0.7,0.89999999999999,1.09999999999999]`

Lijsten van oneindige lengte

Omdat Haskell lui is kun je een lijst aanmaken van oneindige lengte. Haskell zal de lijst niet volledig evalueren als je dat niet vraagt.

- `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]`

List comprehensions

$S = \{x \cdot x | x \in \mathbb{N}, x \leq 10, x \mod 2 = 0\}$
In Haskell:

```
Prelude> [x * x | x <- [1..10], even x]
[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)]
```

Verwijdere dubbele triplets:

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

Sieve of Eratosthenes

Wikipedia

The sieve of Eratosthenes, one of a number of prime number sieves, 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

```
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]
```

Inhoud

Anonieme functies

Anonieme functies

An anonymous function is a function without a name. It is a lambda abstraction and it might look like this: `\x -> x + 1`

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

Filter

Verwijder alle elementen uit een lijst waarvoor een gegeven functie False teruggeeft.

Filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _pred []    = []
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]
```

Map

Pas een functie toe op ieder element van een lijst.

Definition of map

```
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]
```

(Ab)using lazyness

We kunnen argumenten van te voren in een functie stoppen:

Lecture.hs

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

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

Ook als we anonieme functies gebruiken (vergeet de haakjes niet!)

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

Currying

Dit “luie” gedrag hoe functies in Haskell werken heeft te maken met dat functies eigenlijk maar één parameter hebben in Haskell. Functies met meerdere argumenten zijn “curried” functies.

Lees meer hierover (voor het tentamen), bijvoorbeeld hier:
`http://learnyouahaskell.com/higher-order-functions`

Foldr

'foldr', applied to a binary operator, a starting value (typically the right-identity of the operator), and a list, reduces the list using the binary operator, from right to left.

Definition of foldr

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

```
Prelude> foldr (+) 0 [1,2,3,4,5]
15
Prelude> foldr (\x y -> x + 2 * y) 4 [1,2,3]
49
```

Foldl

'foldl', applied to a binary operator, a starting value (typically the left-identity of the operator), and a list, reduces the list using the binary operator, from left to right:

Definition of foldr

```
foldl        :: (b -> a -> b) -> b -> [a] -> b
foldl f z0 xs0 = lgo z0 xs0
             where
                lgo z []     =  z
                lgo z (x:xs) = lgo (f z x) xs
```

```
Prelude> foldl (\x y -> 2*x + y) 4 [1,2,3]
43
```

Zip

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

Definition of zip

```
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')]
```

ZipWith

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

Definition of zipWith

```
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]
```

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.

```
(.) :: (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]
```

Inhoud

Types

Zoals we hebben gezien, hebben variabelen een type:

```
Prelude> :t True
True :: Bool
Prelude> :t 'a'
'a' :: Char
Prelude> :t "hooi"
"hooi" :: [Char]
```

In Haskell zijn dat:

- Int: gehele getallen, maar beperkt
- Integer: gehele getallen, onbeperkt (dus langzamer!)
- Float: single precision floats
- Double: double precision floats
- Bool: True of False
- Char: een enkel karakter en `[Char]` is een string.

Type signatures

En kunnen we bij het definiëren van een functie een type-signature meegeven:

Lecture.hs

```
fac :: Integer -> Integer
fac 0 = 1
fac n = n * foo (n-1)
```

of een niet bekend type:

Lecture.hs

```
third :: (a,b,c) -> c
third (_,_,z)  = z
```

Type aliases

Met behulp van het **type** keyword kunnen we aliases aanmaken:

Lecture.hs

```
type Feedback = (Int, Int)
```

Eigen datatypes

Met behulp van het **data** keyword kunnen we ons eigen type definiëren:

Lecture.hs

```
data Color = Red | Yellow | Blue | Green | Orange | Purple
             deriving (Eq, Show, Bounded, Enum)
```

```
Prelude> [minBound..maxBound] :: [Color]
[Red,Yellow,Blue,Green,Orange,Purple]
Prelude> let foo = Red
Prelude> foo
Red
Prelude> :t foo
foo :: Color
```

Typeclasses

Daarnaast kunnen types onderdeel zijn van typeclasses:

```
Prelude> :t elem
elem :: Eq a => a -> [a] -> Bool
```

Door aan te geven in welke typeclass het type zat.

- Eq: type ondersteunt == en /=
- Ord: type is geordend
- Show: type kan als een string worden gerepresenteerd
- Read: string kan als het type worden gerepresenteerd
- Enum: type is sequentieel geordend
- Bounded: type heeft een ondergrens en bovengrens
- Num: type gedraagt zich als een getal
- Integral: type is een geheel getal (Int/Integer)
- Floating: type is een gebroken getal (Float/Double)

Inhoud

Tips voor het gebruiken van GHCi

Toon het type van een functie of variabele:

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

Toon voor alle volgende commando's:

```
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]
```

Schakel het weer uit:

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

Nogmaals, operatoren zijn ook functies

Lecture.hs

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

```
Prelude> neg 3
-3
Prelude> neg -3
<interactive>:9:1:
No instance for (Num a0) arising from a use of 'neg'
The type variable 'a0' is ambiguous
Possible fix: add a type signature that fixes these type variable(s)
```

Oplossing: haakjes or $:

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