

Haskell les 2

Functies zijn cool

door Pietervdn

Syntax

If then else

- Speciale syntax
- Geeft een waarde terug

*if boolean then expressie else
expressie*

If then else

```
fac :: Int -> Int
```

```
fac i = if i == 0 then 1
```

```
      else i * fac $ i - 1
```

Syntax met pattern matching is beter

Case

- Speciale syntax
- Geeft een waarde terug

case expressie of

pattern match -> expressie

pattern match -> expressie

...

Case

case list of

[] -> 0

(False:as) -> length as

(True:True:_) -> 2

Tuple

Soms willen we verschillende waarden
teruggeven/groeperen

(“Pieter”, 1993)

(“Ilion”, 1996)

Tuple

Soms willen we verschillende waarden
teruggeven/groeperen

```
gegevensVan :: Int -> (String, Int)
```

```
gegevensVan id = ...
```


Tuple

```
sumTuple :: (Int, Int) -> Int
```

```
sumTuple (i,j) = i + j
```

Tuple

`fst :: (a,b) -> a`

`fst (a,_) = a`

`snd :: (a,b) -> b`

`snd (_,b) = b`

Tuple

$\text{fmap} :: (b \rightarrow c) \rightarrow (a, b) \rightarrow (a, c)$

$\text{fmap } f (a, b) = (a, f b)$

Lambda functie

- Anonieme functies

\ pattern match -> expressie

Lambda functie

```
map (\i -> i + 1) [1,2,3]
```

```
map (\(a:as) -> a:a:as) [[1,2],[3]]
```

Currying

Plus: twee implementaties

```
plus :: Int -> Int -> Int
```

```
plus i j    = i + j
```

```
plus :: Int -> (Int -> Int)
```

```
plus i      = \j -> i + j
```

Plus: twee implementaties

```
map (plus 5) [1,2,3]
```


Plus: drie implementaties

```
plus :: Int -> Int -> Int
```

```
plus i j    = i + j
```

```
plus i      = \j -> i + j
```

```
plus        = \i -> \ j -> i + j
```

Elke functie is zo opgebouwd

Zipwith

```
zipWith plus [1,2,3] [4,5,6]
```

```
zipWith (+) [1,2,3] [4,5,6]
```

```
map (plus 1) [1,2,3]
```

```
map (+1) [1,2,3]
```

Algebraic Data Types

Sum types

ADT

Abstract Data Type

Volledig gedefinieerd door functies

Boolean

```
data Boolean = Fls | Tr  
  deriving (Show)
```

We maken een eigen type `Boolean` bestaande uit `{Fls, Tr}`

Boolean

We kunnen ons type `Boolean` nu gebruiken

```
and' :: Boolean -> Boolean -> Boolean
```

Boolean

We kunnen ons type Boolean nu gebruiken
En pattern matchen

```
and' :: Boolean -> Boolean -> Boolean
```

```
Tr Tr = Tr
```

```
Fls Fls = Fls
```

ADT

Wat zijn die `Fls` en `Tr` nu juist?

Het is een *functie* (zonder argumenten) die iets van het type *Boolean* geeft

```
:t Fls
    Fls :: Boolean
:t Tr
    Tr  :: Boolean
```


Sum type

Boolean is een *sum type*

Het is *ofwel* Tr, *ofwel* Fls

$\text{Boolean} = \{\text{Tr}\} + \{\text{Fls}\}$

Abstract Data Types

Product types

Person

```
data Person = P String Int  
    deriving (Show)
```

We maken een eigen type Person

Person

We kunnen ons type Person nu gebruiken

```
name :: Person -> String
```

Person

We kunnen ons type Person nu gebruiken
En pattern matchen

```
name :: Person -> String
```

```
name (P n _) = n
```

Person

Wat is die P nu juist?

Het is een *functie* (die een string en een getal neemt) en er een *Person* van maakt

```
:t P  
  P :: String -> Int -> Person
```

Person

```
P "Pieter" 1993 :: Person  
P "Ilion" 1996  :: Person
```

Person

```
P "Pieter" 1993 :: Person  
P "Ilion" 1996  :: Person  
P "Eloïse" 1996 :: Person
```

P is een **constructor**

Person

Ook constructoren kan je curryen

```
P "Pieter" 1993  :: Person
P "Ilion"        :: Int  -> Person
P                :: String -> Int  -> Person
```

Person

Ook constructoren kan je curryen

```
map (P "Kenneth") [1995, 1996]
  = [P "Kenneth" 1995, P "Kenneth" 1996]
zipWith P ["Pieter", "Ilion", "Kenneth"]
[1993, 1996, 1995]
  = [P "Pieter" 1993, P "Ilion" 1996,
      P "Ketnet" 1995]
```

Product type

Person is een *product type*

$$\begin{aligned} \text{Person} = & \{P\} * \\ & \{ "", "a", "aa", "ab", \dots \} * \\ & \{0, 1, 2, 3, \dots\} \end{aligned}$$

Abstract Data Types

Sum-Product types

MaybeInt

```
data MaybeInt = Just Int  
              | Nothing
```

MaybeInt

We kunnen ons type MaybeInt nu gebruiken

```
add :: Int -> MaybeInt -> MaybeInt
```

MaybeInt

We kunnen ons type MaybeInt nu gebruiken
En pattern matchen

```
add :: Int -> MaybeInt -> MaybeInt  
add i (Just j) = Just $ i + j  
add i Nothing  = Nothing
```

MaybeInt

Wat is die Nothing nu juist?

Het is een *functie* (zonder argumenten) die iets van het type MaybeInt geeft

```
:t Nothing  
    Nothing :: MaybeInt
```


MaybeInt

Wat is die Just nu juist?

Het is een *functie* (met een getal als argument) die iets van het type MaybeInt geeft

```
:t Just  
    Just :: Int -> MaybeInt
```

MaybeInt

```
Just 5  
Nothing
```

MaybeInt

```
Just 5
```

```
Nothing
```

Just is een **constructor**

Nothing is een **constructor**

MaybeInt

Ook constructoren kan je curryen

```
Just 5      :: MaybeInt  
Just        :: Int -> MaybeInt  
Nothing     :: MaybeInt
```

MaybeInt

Ook constructoren kan je curryen

```
map Just [1995, 1996]  
  = [Just 1995, Just 1996]
```

Sum - Product type

MaybeInt is een *sum-product type*

$$\text{MaybeInt} = \{\text{Nothing}\} + \{\text{Just}\} * \{0, 1, 2, 3, \dots\}$$

Polymorfe ADTs

MaybeInt

```
data MaybeInt = Just Int  
              | Nothing
```


MaybeBool

```
data MaybeBool = Just Bool  
               | Nothing
```

MaybeString

```
data MaybeString = Just String  
                  | Nothing
```

Maybe a

```
data Maybe a = Just a  
              | Nothing
```

(Kleine letter)

Maybe a

Wat is die Just nu juist?

Het is een *functie* (met een **a** als argument) die iets van het type `Maybe a` geeft

```
:t Just
    Just :: a -> Maybe a
```

Maybe a

`fmap :: (a -> b) -> Maybe a -> Maybe b`

`fmap _ Nothing = Nothing`

`fmap f (Just a) = Just $ f a`

Maybe a

```
fmap (+1) (Just 5)
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
fmap _ Nothing = Nothing
```

```
fmap f (Just a) = Just $ f a
```

Maybe a

```
fmap (+1) (Just 5)
```

`fmap :: (a -> b) -> Maybe a -> Maybe b`

`fmap _ Nothing = Nothing`

`fmap f (Just a) = Just $ f a`

Maybe a

```
fmap (+1) (Just 5)
```

\neq
fmap :: (a -> b) -> Maybe a -> Maybe b

fmap _ Nothing = Nothing

fmap f (Just a) = Just \$ f a

Maybe a

```
fmap (+1) (Just 5)
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
fmap _ Nothing = Nothing
```

```
fmap f (Just a) = Just $ f a
```

Maybe a

```
fmap (+1) (Just 5)
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
fmap _ Nothing = Nothing
```

```
fmap f (Just a) = Just $ f a
```

Maybe a

```
fmap (+1) (Just 5)
```

`fmap :: (a -> b) -> Maybe a -> Maybe b`

`fmap _ Nothing = Nothing`

`fmap f (Just a) = Just $ f a`

Maybe a

```
Just $ (+1) 5
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
fmap _ Nothing = Nothing
```

```
fmap f (Just a) = Just $ f a
```

Maybe a

```
Just 6
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
fmap _ Nothing = Nothing
```

```
fmap f (Just a) = Just $ f a
```

Maybe a

```
Just 6
```

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
fmap _ Nothing = Nothing
```

```
fmap f (Just a) = Just $ f a
```

Fmap voert f uit op de a in de *Maybe*

Maybe a

`join :: Maybe (Maybe a) -> Maybe a`

`join (Just (Just a)) = Just a`

`join _ = Nothing`

Maybe a

```
join (Just (Just 5))
```

```
join :: Maybe (Maybe a) -> Maybe a
```

```
join (Just (Just a)) = Just a
```

```
join _ = Nothing
```


Maybe a

```
join (Just (Just 5))
```

```
join :: Maybe (Maybe a) -> Maybe a
```

```
join (Just (Just a)) = Just a
```

```
join _ = Nothing
```

Maybe a

```
join (Just (Just 5))
```

`join :: Maybe (Maybe a) -> Maybe a`

`join (Just (Just a)) = Just a`

`join _ = Nothing`

Maybe a

```
join (Just (Just 5))
```

`join :: Maybe (Maybe a) -> Maybe a`

`join (Just (Just a)) = Just a`

`join _ = Nothing`

Maybe a

Just 5

`join :: Maybe (Maybe a) -> Maybe a`

`join (Just (Just a)) = Just a`

`join _ = Nothing`

Recursive ADTs

List a

```
data List a = Cons a (List a)  
            | Nil
```

List a

Wat is die Nil nu juist?

Het is een *functie* (zonder argumenten) die iets van het type List a geeft

```
:t Nil  
  Nil :: List a
```

List a

Wat is die Cons nu juist?

Het is een *functie* (met twee argumenten) die iets van het type `List a` geeft

```
:t Cons
```

```
Cons :: a -> List a -> List a
```


List a

```
Nil      :: List a
Cons 5 Nil      :: List Int
Cons 6 (Cons 5 Nil) :: List Int
```

Haskell's gewone lijsten hebben speciale syntax:

(:) ipv Cons

[a] ipv List a

List a

```
head :: List a -> a
```

```
head (Cons a _) = a
```

```
tail :: List a -> List a
```

```
tail (Cons _ as) = as
```

Higher order programming

List a vs Maybe a

- map, doeElk en concatMap ook definieerbaar op onze eigen List
- Equivalenten voor Maybe a?

DoeOp

Gegeven: Maybe (a -> b), Maybe a

`doeOp :: Maybe (a -> b) -> Maybe a -> Maybe b`

`doeOp (Just f) (Just a) = Just $ f a`

`doeOp _ _ = Nothing`

DoeOp

```
doeOp (Just (+1)) (Just 41)
```

```
doeOp :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
doeOp (Just f) (Just a) = Just $ f a
```

```
doeOp _ _ = Nothing
```

DoeOp

```
doeOp (Just (+1)) (Just 41)
```

```
doeOp :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
doeOp (Just f) (Just a) = Just $ f a
```

```
doeOp _ _ = Nothing
```

DoeOp

```
doeOp (Just (+1)) (Just 41)
```

```
doeOp :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
doeOp (Just f) (Just a) = Just $ f a
```

```
doeOp _ _ = Nothing
```


DoeOp

```
Just $ (+1) 41
```

```
doeOp :: Maybe (a -> b) -> Maybe a -> Maybe b
```

```
doeOp (Just f) (Just a) = Just $ f a
```

```
doeOp _ _ = Nothing
```

DoeOp

Just 42

`doeOp :: Maybe (a -> b) -> Maybe a -> Maybe b`

`doeOp (Just f) (Just a) = Just $ f a`

`doeOp _ _ = Nothing`

Higher order programming

Databank lookups

```
data :: [(String, Int)]
```

```
data = [ ("Pieter", 21),  
        ("Ilion", 18), ("Kenneth", 19) ]
```

Databank lookups

```
lookup :: a -> [(a,b)] -> Maybe a
```

```
lookup _ [] = Nothing
```

```
lookup k ((a,b):abs)
```

```
    = if k == a then Just b
```

```
      else lookup k abs
```

Databank lookups

Gemeenschappelijke leeftijd van 3 gegeven personen?

Databank lookups

```
totLeeftijd    :: (String, String) ->  
                [(String, Int)] -> Maybe Int  
totLeeftijd (p0,p1) db  
    = lookup p0 db                :: Maybe Int
```

Databank lookups

```
totLeeftijd    :: (String, String) ->
                [(String, Int)] -> Maybe Int
totLeeftijd (p0,p1) db
  = case lookup p0 db of
      Nothing -> Nothing
      Just i   -> case lookup p1 db of
                    Nothing -> Nothing
                    Just j   -> Just (i + j)
```


Databank lookups

Slechte stijl

- Nesting van cases
- “Staircase of doom”

We can do better!

Databank lookups

```
totLeeftijd    :: (String, String) ->
                [(String, Int)] -> Maybe Int
totLeeftijd (p0,p1) db
  = case lookup p0 db of
      Nothing -> Nothing
      Just i   -> case lookup p1 db of
                    Nothing -> Nothing
                    Just j   -> Just (i + j)
```

Databank lookups

```
totLeeftijd    :: (String, String) ->
                [(String, Int)] -> Maybe Int
totLeeftijd (p0,p1) db
  = case lookup p0 db of
      Nothing -> Nothing
      Just i   -> case lookup p1 db of
                    Nothing -> Nothing
                    Just j   -> Just (i + j)
```

Databank lookups

```
andThen :: Maybe a -> (a -> Maybe b) -> Maybe b
```

```
andThen maybeA f
```

```
  = case maybeA of
```

```
    Nothing -> Nothing
```

```
    (Just a) -> f a
```

Databank lookups

```
totLeeftijd    :: (String, String) ->  
                [(String, Int)] -> Maybe Int  
totLeeftijd (p0,p1) db  
    = lookup p0 db `andThen`  
      (\i -> lookup p1 db `andThen`  
        (\j -> Just (i + j) ))
```

Databank lookups

```
totLeeftijd    :: (String, String, String) ->  
                [(String, Int)] -> Maybe Int  
totLeeftijd (p0,p1,p2) db  
    = lookup p0 db `andThen`  
      (\i -> lookup p1 db `andThen`  
        (\j -> lookup p2 db `andThen`  
          (\k -> Just (i + j + k) )))
```

Tooling modules en imports

Module

Een module is stuk samenhangende code

Bestandsnaam = moduleNaam.hs

Module

```
module ModuleNaam where
```

```
... code ...
```

Module

```
module Boolean where
```

```
data Boolean = Tr | Fls
```

```
and' :: ...
```

Module

module *AndereModule* where

import Boolean

Module

```
module AndereModule where
```

```
import Boolean (Boolean, and')
```

Module

`module AndereModule where`

`import Boolean (Boolean (Tr, Fls))`

Module

`module AndereModule where`

`import Boolean hiding (Boolean, and')`

Module

module *AndereModule* where

import **qualified** Boolean as **B**

someFunction :: **B**.Boolean -> ...

someFunction (**B**.Tr)

Module

module *AndereModule* where

import **Prelude**

Module

module *AndereModule* where

import Prelude **hiding** (...)

Tooling: hlint

Oefeningen

Oefeningen

github.
com/pietervdvn/Haskell/les2

Slides en oplossingen komen online