Introductie tot Haskell

door Pietervdvn

Wat is Haskell?

- Functionele programmeertaal
- Geen procedures, methodes of objecten
- Enkel functies en functies op functies

Waarmee

- Haskell compiler
- Interactieve omgeving
- linux: ‘`ghci <bestand.hs>`’ (`apt-get install ghc`)
- Computers hier: Haskell Platform > winGHCi
- Via athena: academic > Hugs

Let’s get started

Simpele expressies

```
1 + 1
21*2
not True
“hello” ++ “ “ ++ “world”
reverse “abc”
length “abcde”
```

Statisch getypeerd

- Alles heeft een vast type
- Type = verzameling van mogelijke waarden
- bv. `Bool = {True, False}`

Statisch getypeerd

Iedere waarde heeft een type

```
1       :: Int
True    :: Bool
“abc”   :: String
```

Statisch getypeerd

Iedere functie heeft een type

```
faculteit  :: Int -> Int
not        :: Bool -> Bool
(+)        :: Int -> Int -> Int
```

Zelf functies maken

```
-- Telt 1 op bij een getal
inc    :: Int -> Int
inc i  =  i + 1
```

Zelf functies maken

Schrijf in bestand.hs

Laad bestand in ghci met

```
:l Bestand.hs
```

of met openen in WinGHCi

Zelf functies maken

Functies zijn vanaf dan bruikbaar

```
inc 41
```

Pattern matching

```
niet       :: Bool -> Bool
niet True  =  False
niet False =  True

asInt         :: Bool -> Int
asInt True    =  1
asInt False   =  0
```

Recursie

```
faculteit     :: Int -> Int
faculteit 0   =  1
faculteit i   =  i * faculteit (i - 1)
```

Higher order functions

Functies kunnen ook functies als argument krijgen

```
doTwice        :: (Int -> Int) -> Int -> Int
doTwice f i    = f (f i)
```

Higher order functions

```
doTwice inc 40
inc (inc 40)
```

Higher order functions

doTwice kan ook toegepast worden op andere types

```
doTwice  :: (String -> String)-> String-> String
doTwice f i   = f (f i)
```

Higher order functions

```
yell      :: String -> String
yell str  = str ++ “!”

doTwice yell “Haskell”
```

Higher order functions

doTwice kan ook toegepast worden op andere types

```
doTwice  :: (String -> String)-> String-> String
doTwice f i   = f (f i)
```

Higher order functions

doTwice kan ook toegepast worden op andere types

```
doTwice  :: (Int -> Int)-> Int -> Int
doTwice f i   = f (f i)
```

Higher order functions

doTwice kan ook toegepast worden op andere types

```
doTwice        :: (a -> a) -> a -> a
doTwice f i    = f (f i)
```

Higher order functions

```
doTwice yell “Haskell”
doTwice inc 40
doTwice not True
```

Nieuwe datatypes

Nieuw datatype

```
data Geslacht = Jongen | Meisje
                    deriving (Show)


-- deriving (Show) is nodig om af te
printen
```

Nieuw datatype

```
ander :: Geslacht -> Geslacht
ander Jongen     = Meisje
ander Meisje     = Jongen

isJongen         :: Geslacht -> Bool
isJongen Jongen  = True
isJongen _       = False
```

Nieuw datatype

```
-- voor de kerk
magTrouwen    :: Geslacht -> Geslacht -> Boolean
magTrouwen Jongen Meisje = True
magTrouwen Meisje Jongen = True
magTrouwen _ _      = False
```

Nieuw datatype

```
data MisschienInt  = Een Int
                   | Geen

ontMisschien  :: MisschienInt -> Int -> Int
ontMisschien (Een i) default  = i
ontMisschien Geen default  = default
```

Nieuw datatype

```
veiligeDeling :: Int -> Int -> MisschienInt
veiligeDeling _ 0  = Geen
veiligeDeling deeltal deler
                   = Een (div deeltal deler)
```

Veralgemening

```
data Misschien a = Een a
                 | Geen

ontMisschien  :: Misschien a -> a -> a
ontMisschien (Een a) default  = a
ontMisschien Geen default     = default
```

Veralgemening

```
geslachtStudent  :: Databank -> Name ->
                                Misschien Geslacht
```

inMisschien

```
inMisschien   :: (a -> a) ->
                 Misschien a -> Misschien a
inMisschien f (Een a) = Een (f a)
inMisschien f Geen    = Geen

inMisschien ander (vindGeslacht ugentDb “Ilion”)
```

inMisschien

```
inMisschien   :: (a -> b) ->
                    Misschien a -> Misschien b
inMisschien f (Een a) = Een (f a)
inMisschien f Geen    = Geen

inMisschien isJongen
   (vindGeslacht ugentDb “Ilion”)
```

Recursieve datatypes

Recursieve datatypes

```
data IntList  = Elem Int IntList
              | Nil


Nil
Elem 3 Nil
Elem 2 (Elem 3 Nil)
Elem 1 (Elem 2 (Elem 3 Nil)))
```

Recursieve datatypes

```
length      :: IntList -> Int
length Nil              = 0
length (Elem _ tail)    = 1 + length tail
```

Recursieve datatypes

```
element    :: Int -> IntList -> Misschien Int
element ind Nil           = Geen
element 0 (Elem e _)      = Een e
element ind (Elem _ ls)   = element (ind - 1) ls
```

Recursieve datatypes

```
onEach  :: (Int -> Int) -> IntList -> IntList
onEach _ Nil  = Nil
onEach f (Elem i tail)
      = Elem (f i) (onEach f tail)


onEach inc (Elem 1 (Elem 2 (Elem 3 Nil)))
```

Recursieve datatypes

We kunnen opnieuw algemeen maken!

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

Recursieve datatypes

```
onEach  :: (a -> a) -> List a -> List a
onEach _ Nil  = Nil
onEach f (Elem a tail)
      = Elem (f a) (onEach f tail)

onEach inc (Elem 1 (Elem 2 (Elem 3 Nil)))
onEach not (Elem True (Elem False Nil))
```

Map

Wat als we een functie van `a -> b` toelaten?

```
map   :: (a -> b) -> List a -> List b
map _ Nil  = Nil
map f (Elem a tail)
      = Elem (f a) (onEach f tail)

map isMeisje (Elem Jongen (Elem Meisje Nil))
```

Oefeningen

Oefeningen

github.com/pietervdvn/haskell