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`)
- Op computer: map ‘ghc’, open

Lets get started

Simpele expressies

```
1 + 1
21*2
not True
“hello” ++ “ “ ++ “kappa”
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

Laad bestand in ghci met

```
:l Bestand.hs
```

Functies zijn vanaf dan bruikbaar

```
inc 41
```

Pattern matching

```
not          :: Bool -> Bool
not True     =  False
not 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)
doTwice (twice 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

doTwice kan ook toegepast worden op andere types

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

Higher order functions

```
doTwice ((++) “!”) “Haskell”
doTwice inc 40
doTwice not True
```

Nieuwe datatypes

Nieuw datatype

```
data Boolean = T
             | F
```

Nieuw datatype

```
not      :: Boolean -> Boolean -> Boolean
not T    = F
not F    = T

and      :: Boolean -> Boolean -> Boolean
and T T  = T
and _ _  = F
```

Nieuw datatype

```
data MaybeInt = Just Int
              | Nothing

fromMaybe  :: MaybeInt -> Int -> Int
fromMaybe (Just i) default  = i
fromMaybe Nothing default   = default
```

Veralgemening

```
data Maybe a  = Just a
              | Nothing

fromMaybe  :: Maybe a -> a -> a
fromMaybe (Just a) default  = a
fromMaybe Nil default       = default
```

Recursieve datatypes

```
data IntList a   = Elem Int IntList
                 | Nil

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

Recursieve datatypes

Simpele lijst implementatie

```
data IntList  = Elem Int IntList
              | Nil

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

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

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

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 asInt (Elem True (Elem False Nil))
```

Oefeningen

Oefeningen

github.com/pietervdvn/haskell

Bonus:

Correctheidsbewijzen