Haskell les 5:

Monads, Error

Hard Truths, Warm Fuzzy things

door Pietervdvn

Syntax:
Do-notation

Do Syntax

Eerst nog wat theorie

Theorie

Undefined, Error, Bottom

Bottom

Sommige dingen kunnen niet uitgerekend worden: **undefined**

```
length [0..]
f x = f x
```

Error

Sommige dingen zijn fout: **undefined**

```
5 `div` 0
head []
error “Message”
```

Error

Sommige dingen zijn fout: **undefined**

```
5 `div` 0
head []
error “Message”
```

Error

```
:t error
  error :: String -> a
error “Hello Error”
  *** Exception: Hello Error
```

Theorie

Kinds

Kind

`Maybe` neemt een type, bv `Maybe Int`, `Maybe Bool`

maw `Maybe` neemt een type en beeld dit af op een ander type

Kind

```
:k Maybe
   Maybe :: * -> *
import Data.Map --Map = Dictonary
:k Map
   Map : * -> * -> *
:k Map Key
   Map Key :: * -> *
```

Theorie

Warm, Fuzzy Things

Suspicious pattern

```
concatMap::[a] ->   ([a] -> [b])    -> [b]
```

Suspicious pattern

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

Suspicious pattern

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

Suspicious pattern

```
andThen::List a ->   (a -> List b)    -> List b
andThen::Maybe a ->  (a -> Maybe b)   -> Maybe b
andThen::Writer m a->(a -> Writer m b)->Writer m b
```

Suspicious pattern

```
andThen::List a ->   (a -> List b)    -> List b
andThen::Maybe a ->  (a -> Maybe b)   -> Maybe b
andThen::Writer m a->(a -> Writer m b)->Writer m b
andThen::State s a ->(a -> State s a) ->State s b
```

Suspicious pattern

```
andThen::List a ->   (a -> List b)    -> List b
andThen::Maybe a ->  (a -> Maybe b)   -> Maybe b
andThen::Writer m a->(a -> Writer m b)->Writer m b
andThen::State s a ->(a -> State s a) ->State s b
return :: a -> List a
return :: a -> Maybe a
return :: a -> Writer m a
return :: a -> State s a
```

Suspicious pattern

```
andThen::List a ->   (a -> List b)    -> List b
andThen::Maybe a ->  (a -> Maybe b)   -> Maybe b
andThen::Writer m a->(a -> Writer m b)->Writer m b
andThen::State s a ->(a -> State s a) ->State s b
return :: a -> List a
return :: a -> Maybe a
return :: a -> Writer m a
return :: a -> State s a
```

Class

Dit handig patroon zit al in Haskell:

```
class Monad m where
   (>>=)    :: m a -> (a -> m b) -> m b
   return   :: a -> m a
```

Class

Dit handig patroon zit al in Haskell:

```
class Monad m where
   (>>=)    :: m a -> (a -> m b) -> m b
   return   :: a -> m a
```

`>>=` wordt uitgesproken als “bind”

Kind

```
[1,2,3] >>= (\i -> [i-1,i+1])
  [0,2,1,3,2,4]
Just 5 >>=
    (\i -> if i == 0 then Nothing
                else Just $ 10 `div` i)
  Just 2
```

Kind

```
runwriter (tell [“hi”] >>= return 5)
  ([“Hi”], 5)
runstate (put 42 >>= return 5) 0
  (42, 5)
```

Laws

```
return a >>= f  = f a
```

(Left identity)

Laws

```
return a >>= f  = f a
m >>= return    = m
```

(Left identity)

(Right identity)

Laws

```
return a >>= f  = f a                (Left identity)
m >>= return    = m                  (Right identity)
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
                                     (Associativity)
```

Intuïtie

`>>=` voegt effecten samen

`return` brengt *pure code* in een context *met effecten*

Intuïtie

`>>=` voegt effecten samen

`return` brengt *pure code* in een context *met effecten*

`Endofunctor:` functie met als type `a -> a`

Intuïtie

`>>=` voegt effecten samen

`return` brengt *pure code* in een context *met effecten*

`Endofunctor:` functie met als type `a -> a`

Dus: een monad is een monoid binnen de categorie der endofunctoren

Syntax

```
sum   :: State [Int] ()
sum =  get >>= (\is ->
         if null is then return 0 else
         let i = head is in
         put (tail is) >>
         sum’ >>= \recSum ->
         return $ i + recSum
```

Syntax

Constant met >>= en >> werken wordt onhandig: do

```
sum   = do  is <- get
            if null is then return 0 else do
            let i = head is
            put $ tail is
            recSum <- sum
            return $ recSum + i
```

Syntax: hoe werkt het?

```
foo   :: State Int ()
foo   = do     i <- get
               put i
```

Syntax: hoe werkt het?

```
foo   :: State Int ()
foo   = get >>= (\i -> put i)
```

Syntax: hoe werkt het?

```
foo   :: State Int ()
foo   = do  stm0
            var1 <- stm1
            var2 <- stm2
            return $ f var1 var2
```

Syntax: hoe werkt het?

```
foo   :: State Int ()
foo   = do  stm0
            var1 <- stm1
            stm2 >>= (\var2 ->
            return $ f var1 var2)
```

Syntax: hoe werkt het?

```
foo   :: State Int ()
foo   =     stm0 >>  (
            stm1 >>= (\var1 ->
            stm2 >>= (\var2 ->
            return $ f var1 var2)))
```

Syntax

Listcomprehensions

Lijst

```
cartesian  :: [a] -> [b] -> [(a,b)]
cartesian as bs
      = do  a  <- as
            b  <- bs
            return (a,b)
```

Lijst

```
cartesian  :: [a] -> [b] -> [(a,b)]
cartesian as bs
      = [(a,b) | a <- as, b <- bs]
```

Lijst

```
cartesian’ :: Eq a => [a] -> [a] -> [(a,a)]
cartesian’ as as’
      = do  a  <- as
            a’ <- as’
            if a == a’ then []
               else return (a,a’)
```

Lijst

```
cartesian’ :: Eq a => [a] -> [a] -> [(a,a)]
cartesian’ as as’
      = [(a,a’) | a <- as, a’ <- as’, a != a’]
```

Lijst

```
cartesian’ :: Eq a => [a] -> [a] -> [(a,a)]
cartesian’ as as’
      = [(a,a’) | a <- as, a’ <- as’, a != a’]
```

HLint helpt

Higher order programming

Parser

Parser

Parser: zet een string om in data

Parser

Parser: zet een string om in data

```
data Parser a = Parser
            {runParser :: String -> [(a,String)]}
```

`runParser` zet een String om in mogelijke uitkomsten

```
runParser parseInt “10”
        = [(0,”10”),(1,”0”),(10,””)]
```

Parser

```
return  :: a -> Parser a
return a= Parser (\str -> [(a,str)] )

abort   :: Parser a
abort   = Parser (\_ -> [])
```

Parser

```
(>>=) :: Parser a -> (a -> Parser b) -> Parser b
(>>=) pa a2pb = Parser (\beginStr ->
          let midResults  = runParser pa beginStr in
            concatMap (combine a2pb) midResults)

combine::(a->Parser b)->(a, String)->[(b,String)]
combine a2pb (a,unparsed)
      = runParser (a2bp a) unparsed
```

Parser

```
instance Monad Parser where
   return  = ...
   (>>=)   = ...
```

Parser

```
-- Gets the next char, or aborts if there is none
getChar :: Parser Char
getChar = Parser
              (\str -> if null str then []
                       else [(head str, tail str])
```

Parser

```
-- Tries the first parser. If it fails, the second
(\/)  :: Parser a  -> Parser a -> Parser a
(\/) a b
      = Parser (\str ->
            let midResult = runParser a str in
            if null midResult then runParser b str
            else midResult)
```

Parser

```
-- Tries both parsers
(/\)  :: Parser a  -> Parser a -> Parser a
(/\) a b
      = Parser (\str ->
            let midResultA   = runParser a str in
            let midResultB   = runParser B str in
            midResultA ++ midResultB)
```

Parser

De primitieven `getChar, return, >>=, (\/)` en `abort` zijn genoeg om elke parser te maken

Parser

De primitieven `getChar`, `return`, `>>=` en `abort` zijn genoeg om elke parser te maken

zonder de interne structuur van `Parser` te kennen!

Parser

```
isString :: String -> Parser Bool
isString []  = return True
isString (c:cs)
        = do   char <- getChar
               if c != char then abort
               else isString cs
```

Parser

```
getBool :: Parser Bool
getBool = (isString “True” >> return True)
          \/ (isString “False” >> return False)
```

Parser

```
getInt  :: Parser Int
getInt = return 0 /\
         (do  first <- getChar
              if first `elem` [‘0’..’9’] then
              do tail  <- getInt
                 return (tail + factor*asInt first)
              else abort
          )
```

Higher order programming

IO

IO

We willen bestanden lezen/schrijven, dingen versturen over netwerken, …

IO

vb:

```
(getLine, getLine)
```

IO

vb:

```
(getLine, getLine)
```

Lazyness: evaluatievolgorde ligt niet vast:

```
(“Eerste lijn”, “Tweede lijn”)
(“Tweede lijn”, “Eerste lijn”)
```

IO

vb:

```
(getLine, getLine)
```

Lazyness: evaluatievolgorde ligt niet vast:

```
(“Eerste lijn”, “Tweede lijn”)
(“Tweede lijn”, “Eerste lijn”)
(“Eerste lijn”, “Eerste lijn”)
```

IO

Volgorde van operaties moet vastgelegd worden

IO

Volgorde van operaties moet vastgelegd worden

Elk statement moet uitgevoerd worden

```
... ; print “hi”; ...
```

IO

State monad: geef de buitenwereld als toestand door

```
foo   :: State World a
foo   = do    print “hi”       :: State World ()
              readFile “path” :: State World String
```

IO

Expliciet World aanpassen is een slecht idee

```
foo   :: State World a
foo   = do     spareWorld <- get
               launchNuclearMissiles
               put spareWorld  -- nothing happened!
```

IO

```
foo   :: IO a
foo   = do    print “bla”                  :: IO ()
              launchStuff                  :: IO a
              str   <- readFile “path”     :: IO String
              ...
```

IO

Elke functie met side-effects heeft het type **IO ...**

IO

Elke ~~functie met side-effects~~ heeft het type **IO ...**

IO

Elke **procedure** heeft het type **IO ...**

IO: hoe uitvoeren

Er bestaat geen “`runIO`” , die impure code kan uitvoeren in een pure context*

* dit is een leugen. Echter, code met side-effects uitvoeren is gevaarlijk, onvoorspelbaar en ongelooflijk slechte stijl.

IO: hoe uitvoeren

Er bestaat geen “`runIO`” , die impure code kan uitvoeren in een pure context*

Een gecompileerd programma wordt echter opgestart in IO

IO: hoe uitvoeren

```
main :: IO ()
main =  putStrLn “Hello World”
```

Tooling

Compiling

Hoe compileren

```
module Main where


main :: IO ()
main =  putStrLn “Hello World”
```

Tooling

Cabal

Cabal

Cabal = package manager voor Haskell

Veel bibliotheken beschikbaar (hackage.org)

```
cabal install <library>
```

Cabal

Zelf package maken:

```
cabal init
```

Dit was de laatste les

Nu is het aan jullie

Andere vakken

voor wie meer wil

Andere vakken

- ➔ Functionele en logische programmeertalen
  - Marko Van Dooren
- ➔ Programmeertalen
  - Eric Laermans
- ➔ Compilers
  - Bjorn De Sutter

Master Wiskundige Informatica

- ➔ Fundamenten van programmeertalen
  - Marko Van Dooren
- ➔ Herschrijfsystemen
  - Kris Coolsaet
- ➔ Berekenbaarheid en complexiteit
  - Gunnar Brinkmann

Project

Nu is het aan jullie

Project

Niet op punten

Liefst alleen

Heb je zelf een idee: zeg het

Project

Maak een simpele programmeertaal *while*

Interpreteer die

Geef het resultaat en tussenstappen van het programma

Planning

Volgende week:

projecthulpavond, mss nog een onderwerp

Paasvakantie

22/04 (week na paasvakantie)

projecthulpavond, deadline* project

29/04

Voorstelling projecten, feedback **

Oefeningen

github.com/pietervdvn/Haskell/Les5
github.com/pietervdvn/Haskell/Project