Haskell les 3:
Type magic

door Pietervdvn

Syntax:
Guards

Guards

```
fac  :: Int -> Int
fac i
   | i == 0      = 1
   | otherwise   = i * fac $ i - 1
```

Soms handig bij complexe checks

Syntax:

@

@

```
doeDingen  :: Maybe Int -> Maybe Int
doeDingen maybe@(Just i)
     = i + doeAndereDingen maybe
```

Syntax:
let

Let

```
foo :: Int -> Int -> Int -> Int
foo x y z
  = let  xy   = x + y in
        z * xy
```

Let

```
foo :: Int -> Int -> Int
foo x y z
  = let  xy   = x + y
         zxy  = xy * xy + z in
       zxy
```

Let

```
bar  :: Maybe Int -> Int
bar maybe
 | isJust maybe
     = let (Just i) = maybe in
             i * 2
 | otherwise = 0
```

Theorie

fmap

```
fmap :: (a -> b) -> [a] -> [b]
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap :: (a -> b) -> Tree a -> Tree b
fmap :: (a -> b) -> Set a -> Set b
```

There’s a suspicious pattern here

fmap

container met dingen van type `a`

Functie wordt uitgevoerd op `a` in die container

Functor

container met dingen van type `a`
Functie wordt uitgevoerd op `a` in die container

Zo'n container noemt een **`Functor`**

Functor

Volgens wikipedia:

In mathematics, a **functor** is a type of mapping between categories, which is applied in category theory. Functors can be thought of as homomorphisms between categories. In the category of small categories, functors can be thought of more generally as morphisms.

Functors were first considered in algebraic topology, where algebraic objects (like the fundamental group) are associated to topological spaces, and algebraic homomorphisms are associated to continuous maps. Nowadays, functors are used throughout modern mathematics to relate various categories. Thus, functors are generally applicable in areas within mathematics that category theory can make an abstraction of.

double fmap

```
doublefmap :: (a -> b) -> (b -> c) -> [a] -> [c]
doublefmap f g  as
        = fmap g $ fmap f as
                   :: [b]
```

double fmap

```
doublefmap :: (a -> b) -> (b -> c) -> [a] -> [c]
doublefmap f g  as
        = fmap g $ fmap f as
```

```
:: [c]
```

double fmap

```
doublefmap (*2) show [1,2,3]
```

```
doublefmap  f    g   as
        = fmap g $ fmap f as
```

double fmap

```
doublefmap (*2) show [1,2,3]
```

```
doublefmap  f    g   as
        = fmap g $ fmap f as
```

double fmap

```
fmap show $ fmap (*2) [1,2,3]
```

```
doublefmap  f    g   as
        = fmap g $ fmap f as
```

double fmap

```
fmap show [2,4,6]
```

```
doublefmap  f    g   as
        = fmap g $ fmap f as
```

double fmap

```
["2","4","6"]
```

```
doublefmap  f    g   as
        = fmap g $ fmap f as
```

double fmap

```
doublefmap :: (a -> b) -> (b -> c) -> [a] -> [c]
doublefmap f g  as
        = fmap g $ fmap f as
```

double fmap

```
doublefmap :: (a -> b) -> (b -> c) -> [a] -> [c]
doublefmap f g
          = fmap g . fmap f
```

double fmap

```
doublefmap :: (a -> b) -> (b -> c) ->
                  Maybe a -> Maybe c
doublefmap f g  as
         = fmap g $ fmap f as
```

double fmap

```
doublefmap (*2) show $ Just 21
```

```
doublefmap  f    g   as
        = fmap g $ fmap f as
```

double fmap

```
fmap show $ fmap (*2) $ Just 21
```

```
doublefmap  f    g   as
        = fmap g $ fmap f as
```

double fmap

```
fmap show $ Just 42
```

```
doublefmap  f    g   as
        = fmap g $ fmap f as
```

double fmap

```
Just “42”
```

```
doublefmap  f    g   as
        = fmap g $ map f as
```

double fmap

```
doublefmap :: (a -> b) -> (b -> c) ->
                  Tree a -> Tree c
doublefmap f g  as
         = fmap g $ fmap f as
```

double fmap

```
doublefmap :: (a -> b) -> (b -> c) ->
                  Tree a -> Tree c
doublefmap f g  as
         = fmap g $ fmap f as
```

Hoe schrijven we dit voor **elke** functor?

double fmap

```
doublefmap :: Functor functor =>
                    (a -> b) -> (b -> c) ->
                   functor a -> functor c
doublefmap f g  as
         = fmap g $ fmap f as
```

Hoe schrijven we dit voor **elke** functor?

double fmap

```
doublefmap :: Functor f =>
                  (a -> b) -> (b -> c) ->
                 f a -> f c
doublefmap f g  as
        = fmap g $ fmap f as
```

Hoe schrijven we dit voor **elke** functor?

Instance

Maybe

We hebben onze eigen `Maybe` gemaakt

Hoe vertellen we Haskell dat dit ook een `Functor` is?

Maybe

```
instance Functor Maybe where
   fmap f (Just a)= Just $ f a
   fmap _ Nothing = Nothing
```

List

We hebben onze eigen `List` gemaakt

Hoe vertellen we Haskell dat dit ook een `Functor` is?

List

```
instance Functor List where
   fmap f (Cons a as)
          = Cons (f a) $ fmap f as
   fmap _ Nil = Nil
```

class

Hoe maken we zelf `functor`

```
class Functor f where
   fmap :: (a -> b) -> f a -> f b
```

Andere nuttige klassen

Sidenote

Waarom `fmap`? Was gewoon `map` niet genoeg?

- `map` voor lijsten
- `fmap` voor functoren
- ‘noobvriendelijk’
- Wordt binnenkort veralgemeend (mss)

Show

```
class Show show where
   show :: show -> String
```

Ding met een representatie

Eq

```
class Eq eq where
  (==) :: eq -> eq -> Bool
```

Vergelijkbaar ding

Ord

```
class Ord ord where
  (>)  :: ord -> ord -> Bool
  (<)  :: ord -> ord -> Bool
  (<) a b  = not (a > b)
```

Vergelijkbaar ding

Num

```
class Num num where
   ...
```

Ding dat een getal is, zoals `Int, Float, ...`

Monoid

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
```

Ding dat een groter wordt, zoals `Int`, `[a]`,...

Monoid

```
5 `mappend` 37 == 42
[1,2] `mappend` [3,4] == [1,2,3,4]
```

Type constraints

```
(+)  :: Num a => a -> a -> a

sort :: Ord a => [a] -> [a]
```

Deriving

Deriving

Moeten we dan altijd zelf `Show, Eq, Ord` maken?

Deriving

Moeten we dan altijd zelf `Show, Eq, Ord` maken?

```
data Maybe a = Just a | Nothing
   deriving (Show, Eq, Ord)
```

Type synoniemen

HTML

Simpele html bibliotheek

```
link = “a”

inside content tag
   =  (open tag) ++ cont ++ (close tag)
```

HTML

Simpele html bibliotheek

```
link :: String
link = “a”
inside:: String -> String -> String
inside content tag
   =  (open tag) ++ cont ++ (close tag)
```

HTML

Simpele html bibliotheek

```
link :: Tag
link = “a”
inside:: String -> Tag -> HTML
inside content tag
   =  (open tag) ++ cont ++ (close tag)
```

HTML

```
type HTML  = String
type Tag   = String
```

String

```
type String  = [Char]
```

Higher order programming

Writer

Berekening die ‘en passent’ verschillende waarden berekent (bv. een logbestand)

Writer

```
berekening :: <argtypes> -> (a, [String])
berekening <args>
    = (..., [“Dingen gedaan”])
```

Writer

Berekening die ‘en passent’ verschillende waarden bijhoudt (bv. een logbestand, een teller)

Constant log teruggeven wordt onhandig

Writer

```
data Writer log a = Writer log a
```

Writer

```
data Writer log a = Writer log a

tell :: log -> Writer log ()
tell entry = Writer entry ()

return :: a -> Writer log a
return a = Writer mepmty a
```

Writer

```
andThen’:: Writer log a
             -> Writer log b -> Writer log b
andThen’ (Writer log _) (Writer log’ b)
 = Writer (log `mappend` log') b
```

Writer

```
andThen :: Writer log a
           -> (a -> Writer log b)-> Writer log b
andThen (Writer log a) a2wb
 = let    (Writer log' b)    = a2wb a in
       Writer (log `mappend` log') b
```

Writer

```
return 42
```

```
return a = Writer a mepmty
```

Writer

```
Writer mempty 42
```

```
return a = Writer a mepmty
```

Writer

```
tell [“hi”]
```

```
tell log = Writer () log
```

Writer

```
Writer [“hi”] ()
```

```
tell log = Writer () log
```

Writer

```
tell [“hi”] `andThen’` return 42
```

Writer

```
Writer [“hi”] () `andThen’` Writer mempty 42
```

Writer

```
Writer [“hi”] () `andThen’` Writer [] 42
```

```
andThen’ (Writer log _) (Writer log’ b)
 = Writer (mappend log log') b
```

Writer

```
Writer (mappend [“hi”] []) 42
```

```
andThen’ (Writer log _) (Writer log’ b)
 = Writer (mappend log log') b
```

Writer

```
Writer [“hi”] 42
```

```
andThen’ (Writer log _) (Writer log’ b)
 = Writer (mappend log log') b
```

Writer

```
tell [“hi”] `andThen’`
tell [“something happens” ] `andThen’`
someLoggingCalculation `andThen`
(\ i -> tell [“We found ”++show i] `andThen’`
      return $ i * 2)
```

Writer

```
Writer [“hi”,”something happens”, <logs>,
        “we found 21”] 42
```

Writer

```
tell 10 `andThen’` return 32 `andThen` tell
```

Writer

```
Writer 42 ()
```

Haddock

Haddock

```
-- | Docstring van functie/module/type
foo :: a -> a
```

Haddock

```
haddock File.hs --html
```

Oefeningen

github.com/pietervdvn/haskell