Haskell les 4:

Lazyness

Een goede informaticus moet lui zijn

door Pietervdvn

Syntax:
Record syntax

Record Syntax

```
data Person  =
  Person Int String Int Bool Bool
```

Record Syntax

```
data Person  =
  Person ID Name Year Gender Bool

type ID = Int
type Name = String
...
```

Record Syntax

```
data Person  =
  Person ID Name Year Gender Bool

type ID = Int
type Name = String
...
```

Record Syntax

```
data Person  =
  Person {id::ID,
          name::Name,
          birth::Year,
          sex:: Gender,
          member::Bool }
```

Record Syntax: getters

```
:t id
  id     :: Person -> Id
:t name
  name   ::Person -> Name
:t member
  member :: Person -> Bool
```

Record Syntax: getters

```
person = Person 42 “Pieter” 1993 True


member person
   True
birth person
   1993
```

Record Syntax: setters

```
person = Person 42 “Pieter” 1993 True


person {id = 43}
   Person 43 “Pieter” 1993 True
person
   Person 42 “Pieter” 1993 True
```

Theorie

Luiheid

Lazyness

Dingen worden maar uitgerekend wanneer het echt nodig is

Lazyness

```
42 `div` 0
   *** Exception: divide by zero
const 5 (42 `div` 0)
   5
```

Lazyness

```
const 5 (42 `div` 0)
```

Lazyness

```
((\x -> (\y -> x)) 5) (42 `div` 0)
```

Lazyness

```
(\y -> 5) (42 `div` 0)
```

Lazyness

```
5
```

Lazyness

Meeste talen:

Call by Name

Call by Name

```
f = \x -> x * x
```

```
f (1 + 1)
```

Call by Name

```
f = \x -> x * x
```

```
f (1 + 1)
(\x -> x * x) (1 + 1)
```

Call by Name

```
f = \x -> x * x
```

```
f (1 + 1)
(\x -> x * x) (1 + 1)
(1 + 1) * (1 + 1)
```

Call by Name

```
f = \x -> x * x
```

```
f (1 + 1)
(\x -> x * x) (1 + 1)
(1 + 1) * (1 + 1)
2 * (1 + 1)
```

Call by Name

```
f = \x -> x * x
```

```
f (1 + 1)
(\x -> x * x) (1 + 1)
(1 + 1) * (1 + 1)
2 * (1 + 1)
2 * 2
```

Call by Name

```
f = \x -> x * x
```

```
f (1 + 1)
(\x -> x * x) (1 + 1)
(1 + 1) * (1 + 1)
2 * (1 + 1)
2 * 2
4
```

Lazyness

Dingen worden maar uitgerekend wanneer het echt nodig is:

Call by Need

Lazyness

Dingen worden maar uitgerekend wanneer het echt nodig is:

Call by Need = Call by Name + Sharing

Call by Need

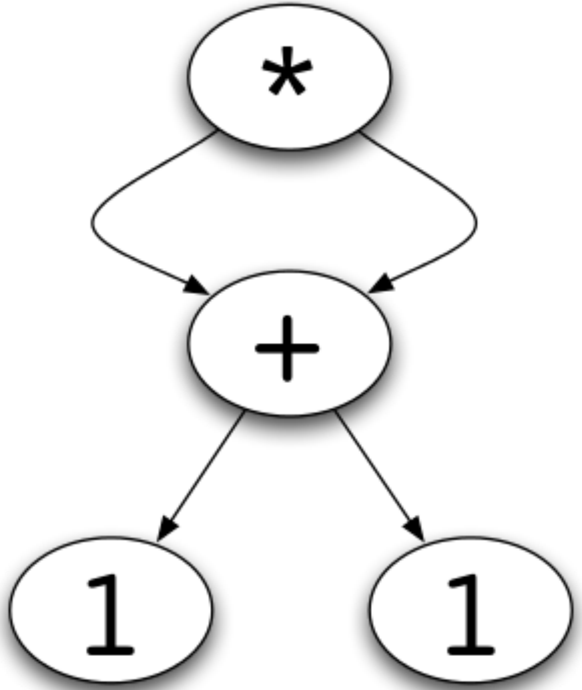
```
f = \x -> x * x
```

```
f (1 + 1)
(\x -> x * x) (1 + 1)
```

Call by Need

```
f = \x -> x * x
```

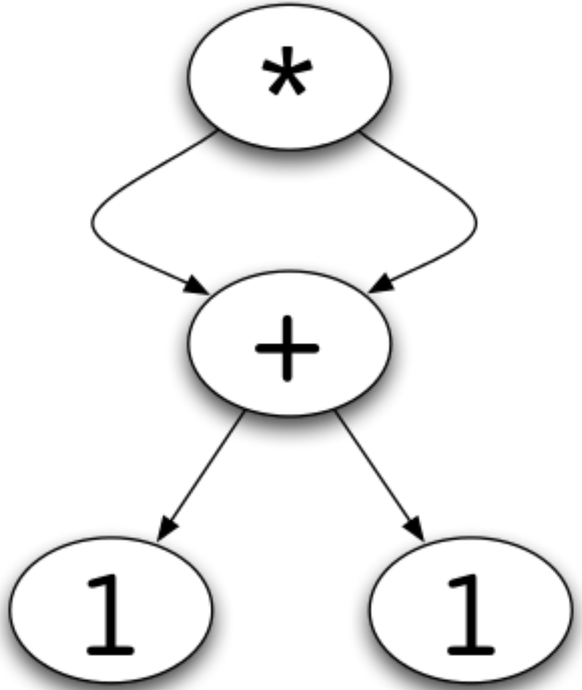
```
f (1 + 1)
(\x -> x * x) (1 + 1)
```



Call by Need

```
f = \x -> x * x
```

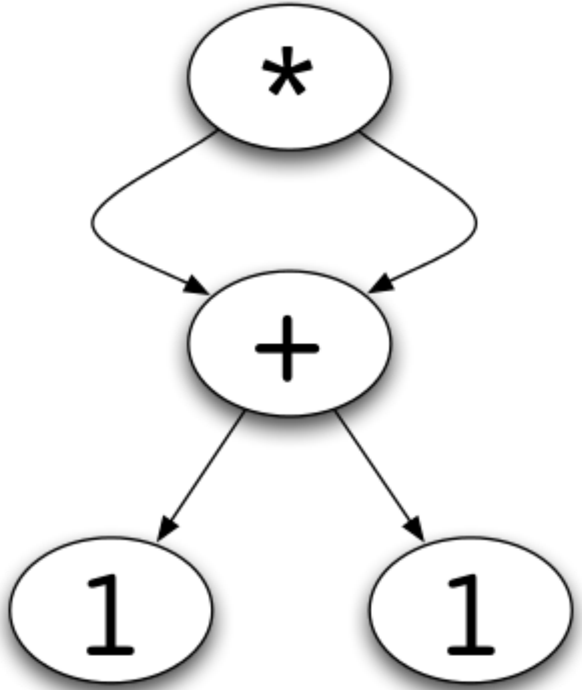
```
f (1 + 1)
(\x -> x * x) (1 + 1)
(1 + 1) * (1 + 1)
```



Call by Need

```
f = \x -> x * x
```

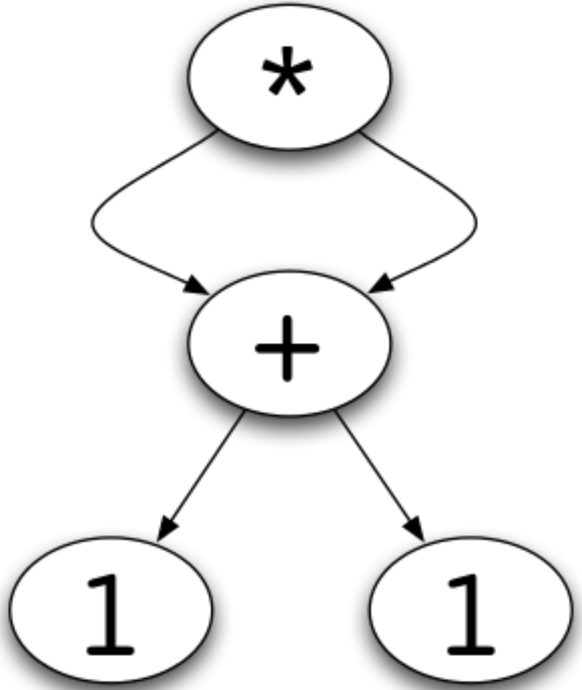
```
f (1 + 1)
(\x -> x * x) (1 + 1)
(1 + 1) * (1 + 1)
2 * 2
```



Call by Need

```
f = \x -> x * x
```

```
f (1 + 1)
(\x -> x * x) (1 + 1)
(1 + 1) * (1 + 1)
2 * 2
4
```



Lazyness: if

```
ifthenelse :: Bool -> a -> a -> a
ifthenelse True a0 _  = a0
ifthenelse False _ a1 = a1
```

Call by Name

```
ifthenelse True 42 (1 * 2 + fac 43 * fib 4
```

Call by Name

```
ifthenelse True 42 (1 * 2 + fac 43 * fib 4
42
```

Call by Name

```
ifthenelse True 42 (5 `div` 0)
42
```

Call by Name

```
ifthenelse False 42 (5 `div` 0)
***Exception: divide by zero
```

Lazyness: oneindige lijst

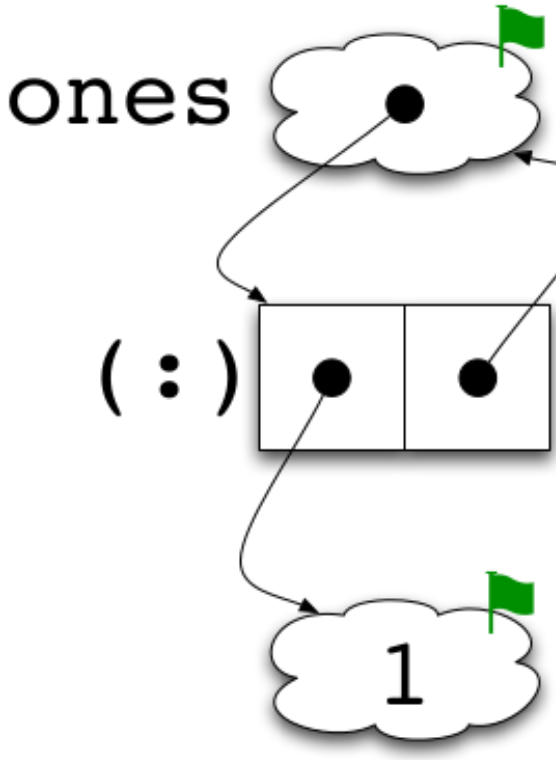
```
ones = 1:ones
```

Lazyness: oneindige lijst

```
ones
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

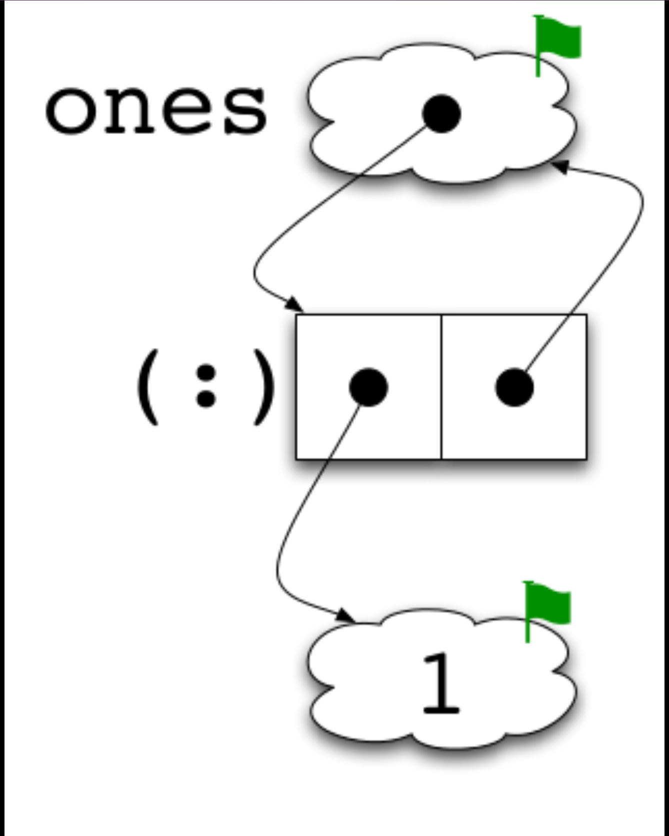
Lazyness: oneindige lijst

```
ones = 1:ones
```



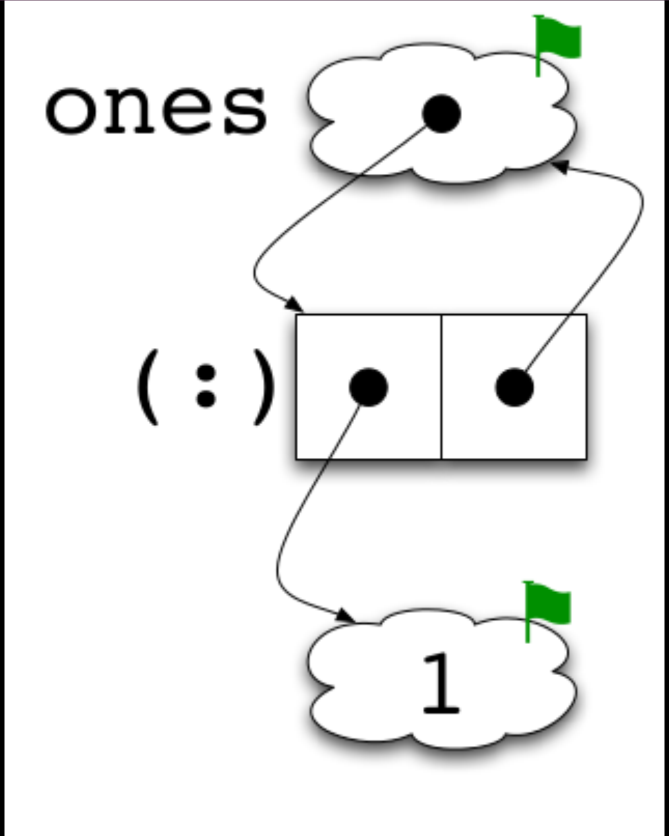
Lazyness: oneindige lijst

```
ones
1:ones
```



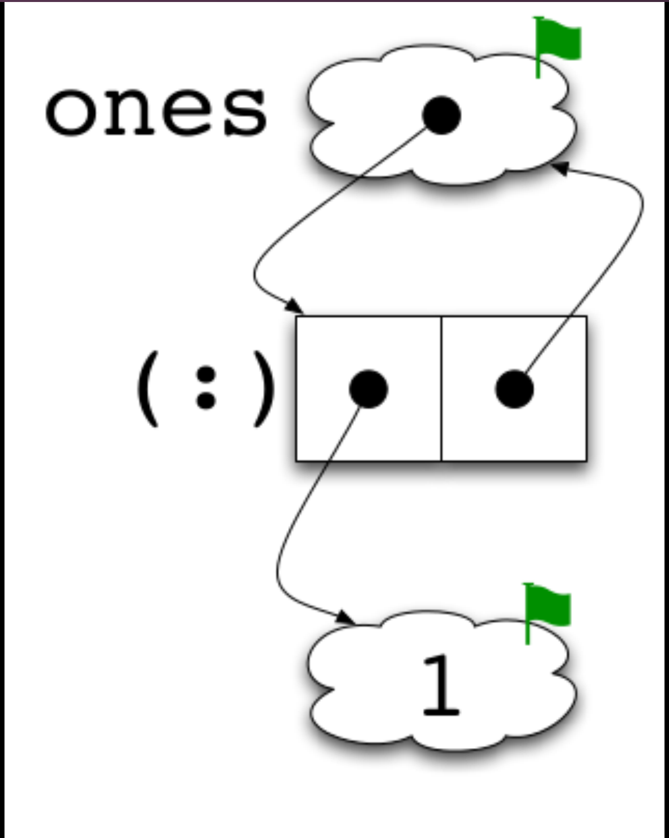
Lazyness: oneindige lijst

```
ones

1:ones

1:1:ones
```



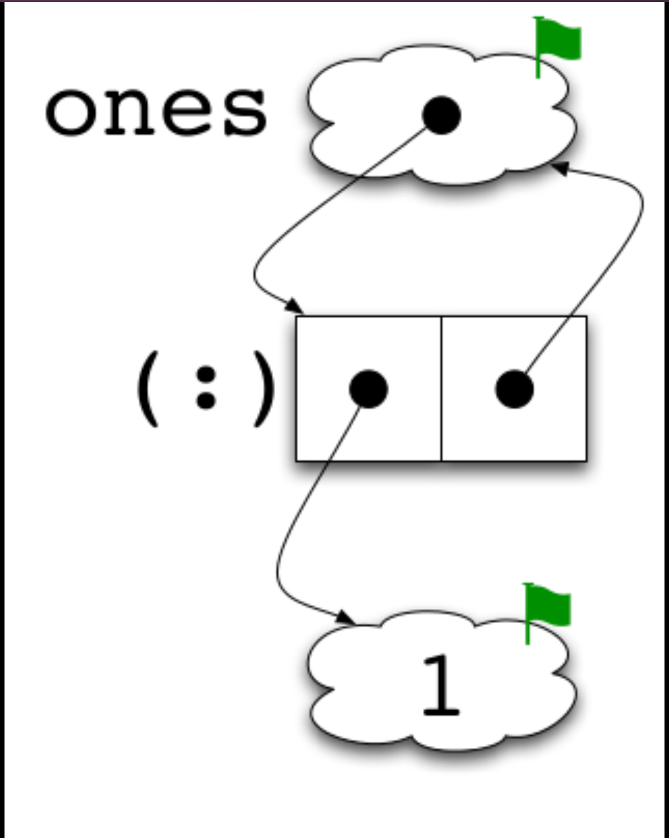
Lazyness: oneindige lijst

```
ones

1:ones

1:1:ones

1:1:1:ones
```



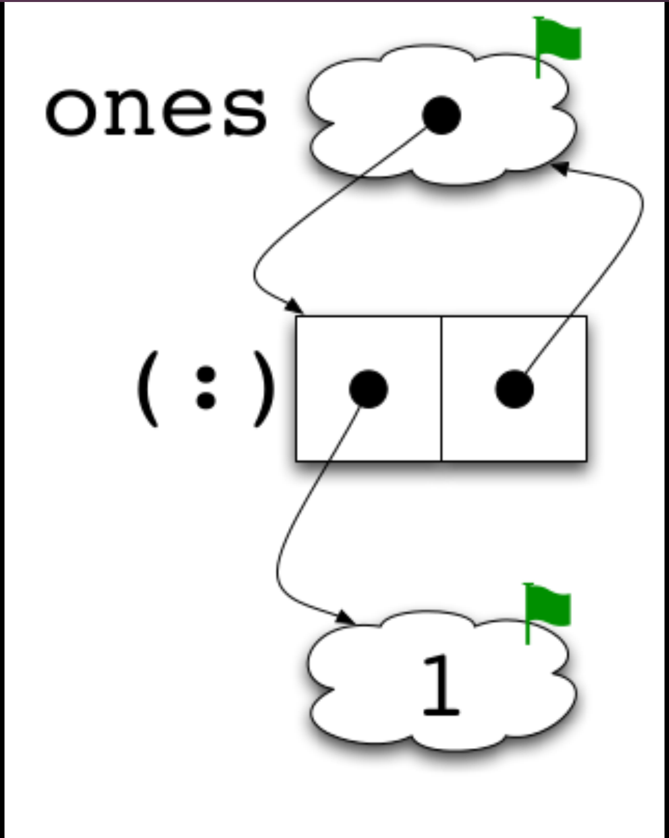
Lazyness: oneindige lijst

```
ones

1:ones

1:1:ones

1:1:1:ones

1:1:1:1:ones
```



Lazyness: oneindige lijst

```
ones
1:ones
1:1:ones
1:1:1:ones
1:1:1:1:ones
… ad infinitum
```



Lazyness: oneindige lijst

```
take 10 ones
  [1,1,1,1,1,1,1,1,1,1]
```

Lazyness: oneindige lijst

```
nats = 0:map (+1) nats
```

Call by Name

```
nats

0:map (+1) nats
```

Call by Name

```
nats

0:map (+1) (0:map (+1) nats)
```

Call by Name

```
nats

0:1:map (+1) (drop 1 nats)
```

Call by Name

```
nats

0:1:map (+1) (drop 1 (0:1: … ):
   map (+1) drop 2 nats
```

Call by Name

```
nats

0:1:map (+1) (1: … ):
  map (+1) drop 2 nats
```

Call by Name

```
nats

0:1:2:map (+1) drop 2 nats
```

Call by Name

```
nats

0:1:2:3:4:5:...
```

Call by Name

```
nats
0:1:2:3:4:5:...

-- syntactische suiker:
[0..]
```

Lazyness: oneindige lijst

Compacte notatie

Lazyness: oneindige lijst

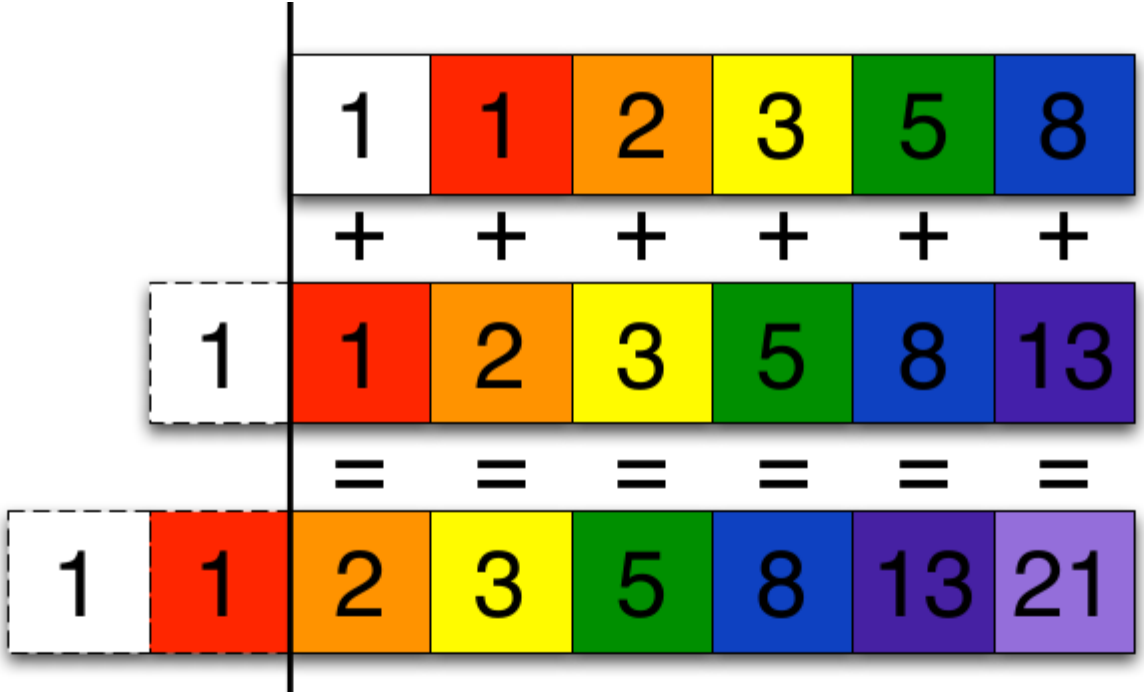
```
sum $ take 10 [1..]
  55
take 10 $ map fac [1..]
  [1,2,6,24,120,720,5040,...,3628800]
```

Lazyness: oneindige lijst

```
repeat 1
  [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
zip [1..5] $ repeat 1
  [(1,1),(2,1),(3,1),(4,1),(5,1)]
zipWith (+1) [1..5] $ repeat 1
  [2,3,4,5,6]
```

Lazyness: oneindige lijst

```
fibs =1 :1 :zipWith (+)fibs (tail fibs)
```



One pass

Boom datastructuur

```
data Tree a  = Leaf a
             | Node a (Tree a) (Tree a)
```

Boom datastructuur

```
-- Zoekt de kleinste waarde
minimum :: Ord a => Tree a -> a
minimum (Leaf a) = a
minimum (Node a left right)
     = min3 a (minimum left) (minimum right)
```

Boom datastructuur

```
-- Vervangt elke waarde door a
repl  :: a -> Tree a -> Tree a
repl a (Leaf _)  = Leaf a
repl a (Node _ left right)
     = Node a (repl a left) (repl a right)
```

Boom datastructuur

Is het mogelijk om een boom te overlopen in 1 pass, en overal het minimum in te stellen?

Boom datastructuur

```
-- Vervangt elke waarde door het minimum a
repMin  :: a -> Tree a -> (Tree a, a)
repMin a (Leaf b)  = (Leaf a, b)
repMin a (Node b left right)
     = let (left’, minLeft) = repMin a left
           (right’, minRight) = repMin a right
     (Node a left’ right’,
           min3 b minLeft minRight)
```

Boom datastructuur

```
(newTree, min) = repMin min tree
```

Lazyness: issues

Strictness

Lazyness is traag voor sommige algoritmen (vooral numerieke)

Gebruik stricte code

Strictness

e.g

```
Node (count + 1) …
Node (0 +1 +1 +1 +1 …)
```

Strictness

```
let count = oldCount + 1 in
    seq count $ foo count
```

-- evalueert ‘count’ voordat deze als parameter wordt doorgegven

Higher order programming

State

Toestand

Voor sommige algoritmes is toestand handig (bv. Dijkstra, queue)

Toestand

```
doeDing :: <args> -> [Node] -> ([Node], <res>)
doeDing … queue
   = let toVisit = head queue
         …              in
         (tail queue, …)
```

Toestand

```
data State s a  = State
                    {runstate :: s -> (s,a) }
```

Toestand

```
data State s a   = State (s -> (s,a))

runstate:: State s a -> s -> (s,a)
runstate (State (s,a))   = \s -> (s,a)
```

Toestand

```
data State s a   = State
                    {runstate :: s -> (s,a) }

return   :: a -> State s a
return a    = State (\s -> (s,a))
```

Toestand

```
put   :: s -> State s ()
put s = State (\_ -> (s,()) )

get   :: State s s
get   = State (\s -> (s,s))
```

Toestand

```
andThen    :: State s a -> (a -> State s b) -> State s b
andThen action    a2actionb
 = State $ \beginState ->
   let (midState, a)   = runstate action beginState
       (endState, b)   = runstate (a2actionb a) midState in
       (endState, b)
```

Toestand

```
runstate (put 5) 0
```

```
put s = State (\_ -> (s,()) )
```

Toestand

```
runstate (State $ \_ -> (5,()) 0
```

```
put s = State (\_ -> (s,()) )
```

Toestand

```
(\_ -> (5,()) ) 0
```

```
put s = State (\_ -> (s,()) )
```

Toestand

```
(5,())
```

```
put s = State (\_ -> (s,()) )
```

Toestand

```
runstate get 42
```

```
get   = State (\s -> (s,s) )
```

Toestand

```
runstate (State $ \s -> (s,s)) 42
```

```
get   = State (\s -> (s,s) )
```

Toestand

```
(\s -> (s,s)) 42
```

```
get    = State (\s -> (s,s) )
```

Toestand

```
(42,42)
```

```
get   = State (\s -> (s,s) )
```

Toestand

```
runstate (put 5 `andThen` const get) 42
```

```
andThen    :: State s a -> (a -> State s b) -> State s b
andThen action    a2actionb
 = State $ \beginState ->
   let (midState, a)   = runstate action beginState
       (endState, b)   = runstate (a2actionb a) midState in
       (endState, b)
```

Toestand

```
runstate (put 5 `andThen` const get) 42

andThen     :: State s a -> (a -> State s b) -> State s b
andThen action    a2actionb
 = State $ \beginState ->
   let (midState, a)   = runstate (put 5) beginState
       (endState, b)   = runstate ((\_ -> get) a) midState in
       (endState, b)
```

Toestand

```
runstate (put 5 `andThen` const get) 42

andThen     :: State s a -> (a -> State s b) -> State s b
andThen action    a2actionb
 = State $ \beginState ->
   let (5, ())    = runstate (put 5) beginState
       (5, 5)    = runstate ((\_ -> get) ()) 5 in
       (5, 5)
```

Toestand

```
runstate (put 5 `andThen` const get) 42

andThen     :: State s a -> (a -> State s b) -> State s b
andThen action    a2actionb
 = State $ \beginState -> (5,5)
```

Toestand

```
runstate (State $ \_ -> (5,5,)) 42
```

Toestand

```
(5,5)
```

Toestand

```
foo   :: State ([Node], Map Node Node,
Graph) ()
foo   =  get (\ (toVisit:queue, paths, graph) -
>
         let cn = getClosest toVisit graph in
         put (queue, insert toVisit cn, graph)
```

Toestand

```
data Context  = Ctx {queue :: [Node],
                     paths :: Map Node Node,
                     graph :: Graph}

foo   :: State Context ()
foo   =  get (\ ctx ->
         let node= head $ queue ctx in
         let cn = getClosest node (graph ctx) in
         put $ ctx {queue = tail queue,
                    paths = insert node cn}
```

Tooling

quickcheck

Quickcheck

Unit-test voor haskell

Quickcheck

~~Unit-test voor haskell~~

Unit-test = quickcheck voor Java

Quickcheck

Unit-test voor haskell

Stel een aantal eigenschappen op

```
\s -> s == s
```

Quickcheck

Unit-test voor haskell

Stel een aantal eigenschappen op

```
\s -> s == s
```

Test deze

QuickCheck

```
import Test.QuickCheck
quickCheck ((\str -> str == str) :: [Char] -> Bool)
+++ OK, passed 100 tests.
```

Copyright

Met dank aan Prof. Tom Schrijvers, van wie ik de afbeeldingen (en oefeningen) overgenomen heb

Oefeningen

github.com/pietervdvn/Haskell/Les4