Les 1

Haskell

door Pietervdvn

en Ilion

Public Service announcements

- Slides komen online
- Computer aan, hack er op los
- Raak je achterop: geen probleem
    - Oefeningen = doe de dingen op de slides

Public Service announcements

- Vragen mogen altijd

Public Service announcements

- Introductieles = warmmaker
- We beginnen terug vanaf nul vandaag

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”
```

Statisch getypeerd

- Alles heeft een vast type
- Type = verzameling van mogelijke waarden
- `Bool = {True, False}`
- `Int = {..., -1, 0, 1, 2, ...}`

Simpele expressies

```
1 + 1                      :: Int
21*2                       :: Int
not True                   :: Bool
“hello ”++ “world”         :: String
reverse “abc”              :: String
```

Statisch getypeerd

Iedere functie heeft een type

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

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

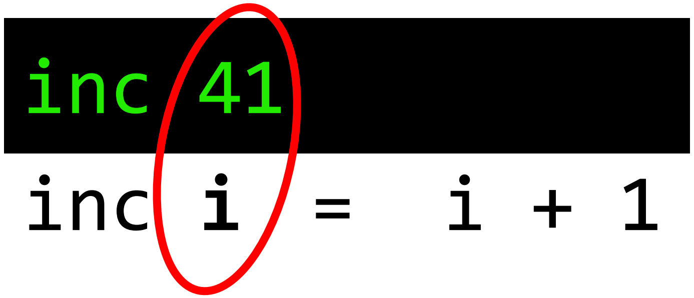
Functie berekenen

```
inc 41
```

```
inc i  =  i + 1
```

Zelf functies maken

Functie berekenen



```
inc 41
```

```
inc i  =  i + 1
```

Zelf functies maken

Functie berekenen

```
41 + 1
```

```
inc i  =  i + 1
```

Zelf functies maken

Functie berekenen

```
     42
```

```
inc i  =  i + 1
```

Pattern matching

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

Pattern matching

```
niet False
```

```
niet True  =  False
niet False =  True
```

Pattern matching

```
niet False
```

```
niet True  =  False
niet False =  True
```

Pattern matching

```
niet False
      ≠
niet True  =  False
niet False =  True
```

Pattern matching

```
niet False
```

```
niet True  =  False
niet False =  True
```

Pattern matching

```
niet False
```

```
niet False =  True
```

Pattern matching

```
niet False
```

```
niet False =   True
```

Pattern matching

```
True
```

```
niet False =  True
```

Recursie

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

Recursie

```
faculteit 3
```

```
faculteit 1   =  1
faculteit i   =  i * faculteit (i - 1)
```

Recursie

```
3 * faculteit 2
```

```
faculteit 1   =  1
faculteit i   =  i * faculteit (i - 1)
```

Recursie

```
3 * 2 * faculteit 1
```

```
faculteit 1   =  1
faculteit i   =  i * faculteit (i - 1)
```

Recursie

```
3 * 2 * 1
```

```
faculteit 1   =  1
faculteit i   =  i * faculteit (i - 1)
```

Recursie

```
6
```

```
faculteit 1    =  1
faculteit i    =  i * faculteit (i - 1)
```

Lijsten

Functies met functies

Lijsten

```
[1,2,3]
[]
1 : [2,3]
1:2:3:[]
[1] ++ [2,3] ++ []
```

Lijsten

Type van een lijst

```
 [Int]
[Bool]
[String]
```

Pattern matching op lijsten

```
sum        :: [Int] -> Int
sum []     = 0
sum (i:is) = i + sum is
```

Pattern matching op lijsten

```
sum [1,2,3]
      ≠
sum []     = 0
sum (i:is) = i + sum is
```

Pattern matching op lijsten

```
sum  [1,  2,3]
```

```
sum []     = 0
sum (i:is) = i + sum is
```

Pattern matching op lijsten

```
sum  [1,  2,3]
```

```
sum []     = 0
sum (i:is) = i + sum is
```

Pattern matching op lijsten

```
1 + sum [2, 3]
```

```
sum []     = 0
sum (i:is) = i + sum is
```

Pattern matching op lijsten

```
1 + 2 + sum [3]
```

```
sum []     = 0
sum (i:is) = i + sum is
```

Pattern matching op lijsten

```
1 + 2 + 3 + sum []
```

```
sum []     = 0
sum (i:is) = i + sum is
```

Pattern matching op lijsten

```
1 + 2 + 3 + 0
```

```
sum []     = 0
sum (i:is) = i + sum is
```

Pattern matching op lijsten

```
6
```

```
sum []       = 0
sum (i:is) = i + sum is
```

Polymorfisme

```
length          :: [Int] -> Int
length []       = 0
length (i:is)= 1 + length is
```

Polymorfisme

```
length          :: [Int] -> Int
length []       = 0
length (i:is)= 1 + length is
```

Polymorfisme

```
length       :: [String] -> Int
length []    = 0
length (i:is)= 1 + length is
```

Polymorfisme

```
length       :: [Bool] -> Int
length []    = 0
length (i:is)= 1 + length is
```

Polymorfisme

```
length       :: [a] -> Int
length []    = 0
length (i:is)= 1 + length is
```

Tail

```
tail         :: [a] -> [a]
tail []      = []
tail (_:is)  = is
```

Type inferentie

Wat is het type van

```
tail [1,2,3]
```

```
tail :: [a] -> [a]
```

Haskell ziet

```
a is in dit geval Int
```

Dus:

```
tail [1,2,3] :: [Int]
```

Type inferentie

Wat is het type van

```
tail []
```

```
   tail :: [a] -> [a]
```

Haskell ziet

```
   we weten niets over a
```

Dus:

```
tail [] :: [a]
```

Higher order functions

Functies over functies

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
```

```
doTwice f   i = f (f i)
```

Higher order functions

```
doTwice inc 40
```

```
doTwice f  i = f (f i)
```

Higher order functions

```
inc (inc 40)
```

```
doTwice f  i  = f (f i)
```

Higher order functions

```
42
```

```
doTwice f  i  = f (f i)
```

Polymorfisme

`doTwice` kan ook toegepast worden op andere types

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

Polymorfisme

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

```
doTwice yell “Haskell”
```

Polymorfisme

doTwice kan ook toegepast worden op andere types

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

Polymorfisme

doTwice kan ook toegepast worden op andere types

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

Polymorfisme

doTwice kan ook toegepast worden op andere types

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

Polymorfisme

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

Map

Tijd voor zwarte magie

Map

```
map :: (Int -> Int) -> [Int] -> [Int]
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
map inc [1,2,3]
```

```
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
map inc [1,2,3]
```

```
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
inc 1 : map inc [2,3]
```

```
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
inc 1 : inc 2 : map inc [3]
```

```
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
inc 1 : inc 2 : inc 3 : map inc []
```

```
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
inc 1 : inc 2 : inc 3 : []
```

```
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
[inc 1, inc 2, inc 3]
```

```
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
[2,3,4]
```

```
map _ [] = []
map f (i:is) = f i : map f is
```

Map

```
map  :: (Int -> Int) -> [Int] -> [Int]
map _ [] = []
map f (i:is) = f i : map f is
```

Map = doe ‘f’ op elk element van de lijst

Polymorfisme

```
map  :: (Int -> Int) -> [Int] -> [Int]
map _ [] = []
map f (i:is) = f i : map f is
```

Polymorfisme

```
map :: (Bool -> Bool) -> [Bool] -> [Bool]
map _ [] = []
map f (i:is) = f i : map f is
```

Polymorfisme

```
map  :: (a -> a) -> [a] -> [a]
map _ [] = []
map f (i:is) = f i : map f is
```

Polymorfisme

```
map :: (Int -> Bool) -> [Int] -> [Bool]
map _ [] = []
map f (i:is) = f i : map f is
```

Polymorfisme

```
map :: (Bool -> Int) -> [Bool] -> [Int]
map _ [] = []
map f (i:is) = f i : map f is
```

Polymorfisme

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (i:is) = f i : map f is
```

Higher order programming

Tijd voor meer zwarte magie

Higher order programming

```
Lijst van waarden [1,2,3]
Lijst met functies resultaten:
   [(+2),(*3)]
Elke mogelijke combinatie?
```

Higher order programming

```
doeElk _ [] = []
doeElk as   (f:fs)
         = map f as ++ doeElk as fs
```

Higher order programming

```
[1,2,3] `doeElk` [+2,*3]
```

```
doeElk _ [] = []
doeElk as   (f:fs)
         = map f as ++ doeElk as fs
```

Higher order programming

```
map (+2) [1,2,3] ++ [1,2,3] `doeElk` [*3]
```

```
doeElk _ [] = []
doeElk as   (f:fs)
         = map f as ++ doeElk as fs
```

Higher order programming

```
map (+2) [1,2,3] ++ map (*3) [1,2,3] ++ []
```

```
doeElk _ [] = []
doeElk as   (f:fs)
         = map f as ++ doeElk as fs
```

Higher order programming

```
[3,4,5] ++ [3,6,9] ++ []
```

```
doeElk _ [] = []
doeElk as   (f:fs)
         = map f as ++ doeElk as fs
```

Higher order programming

```
[3,4,5,3,6,9]
```

```
doeElk _ [] = []
doeElk as   (f:fs)
         = map f as ++ doeElk as fs
```

Higher order programming

```
[1,2,3] `doeElk` [+2,*3] `doeElk` [-1,*2]
```

Higher order programming

```
Lijst van waarden [1,2,3]
Functie met meerdere resultaten:
   f1 i  = [i+2,i*3]
Elke mogelijke combinatie?
```

Higher order programming

```
concatMap  :: [a] -> (a -> [b]) -> [b]
concatMap _ [] = []
concatMap as a2bs
         = concat (map a2bs as)
```

```
[[b]]
```

Higher order programming

```
[1,2,3] `concatMap` f1
```

```
doeElk as a2bs
         = concat (map f1 as)
```

Higher order programming

```
concat (map f1 [1,2,3])
```

```
doeElk as a2bs
           = concat (map f1 as)
```

Higher order programming

```
concat (map f1 [1,2,3])
```

```
f1 i= [i+2, i*3]
```

Higher order programming

```
concat ([[3,3], [4,6],[5,9]])
```

Higher order programming

```
[3,3,4,6,5,9]
```

Higher order programming

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

Tooling

Hoogle

Functies zoeken op basis van type signatuur:

`haskell.org/hoogle`

DuckDuckGo bang pattern !h

Geef een type in, bv “Bool -> Bool -> Bool”, krijg functies!

Wat is het type?

```
:t True
    True :: Bool
:t “abc”++”def”
    “abc”++”def” :: [Char] -- dus string
:t 1
    1 :: Num a => a -- fancy manier voor “getal”
:t inc
    inc :: Int -> Int
```

Wat meer info?

```
:i inc
  inc :: Int -> Int
  -- Defined at Untitled Haskell.hs:10:1
```

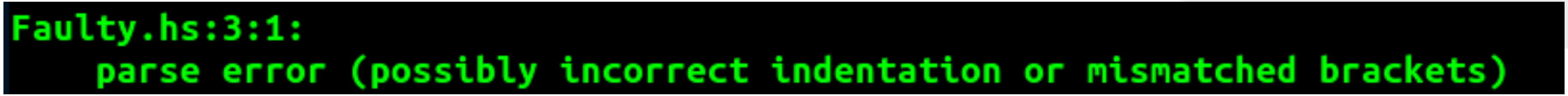
Herladen

```
:l Oplossingen1.hs
    [1 of 1] Compiling Main
        (Oplossingen1.hs, interpreted )
     Ok, modules loaded: Main.
:r
    [1 of 1] Compiling Main
        (Oplossingen1.hs, interpreted )
     Ok, modules loaded: Main.
```

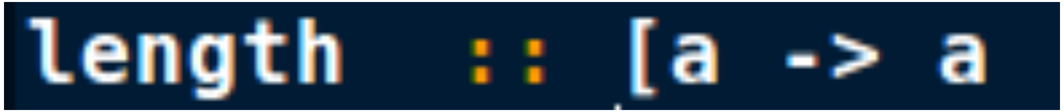
Foutmeldingen

Don’t Panic

Foutmeldingen

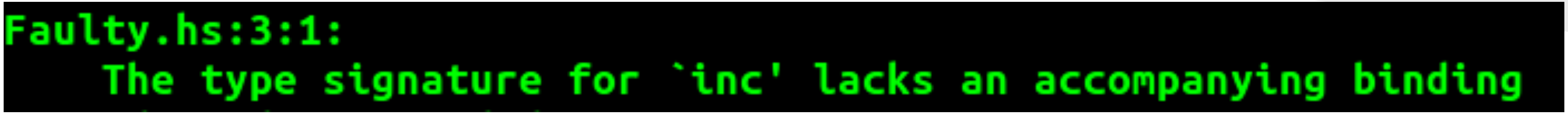


```
Faulty.hs:3:1:
    parse error (possibly incorrect indentation or mismatched brackets)
```

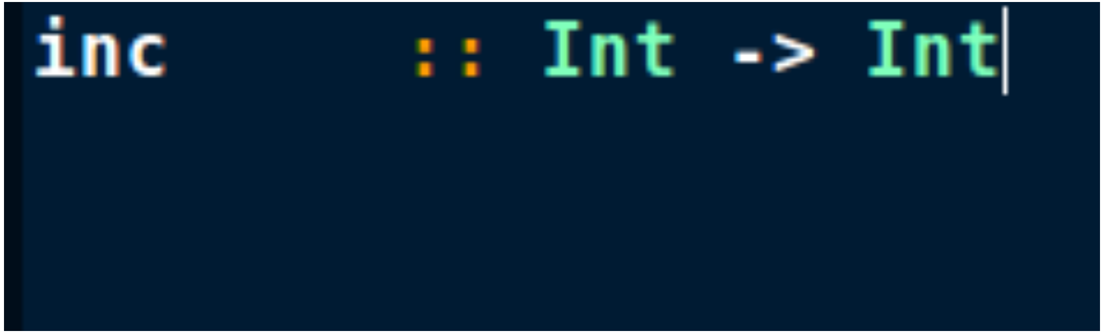


```
length  :: [a -> a
```

Foutmeldingen

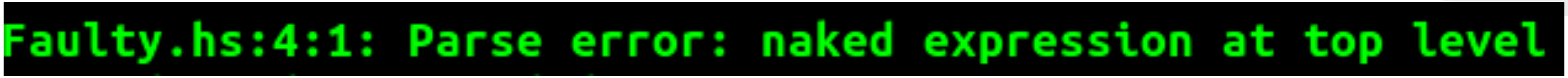


```
Faulty.hs:3:1:
    The type signature for `inc' lacks an accompanying binding
```

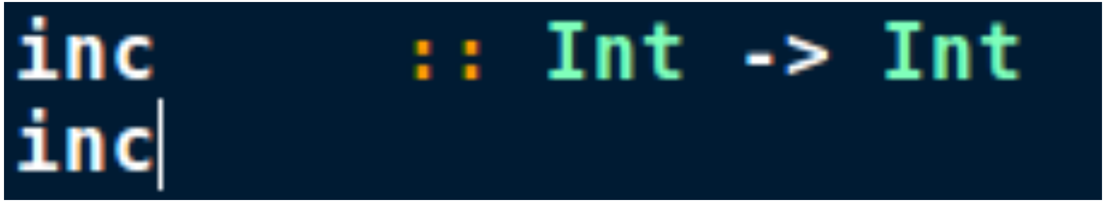


```
inc         :: Int -> Int
```

Foutmeldingen

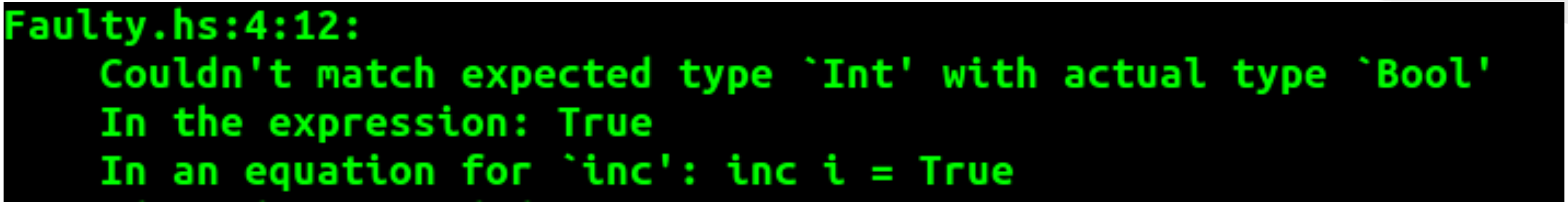


```
Faulty.hs:4:1: Parse error: naked expression at top level
```

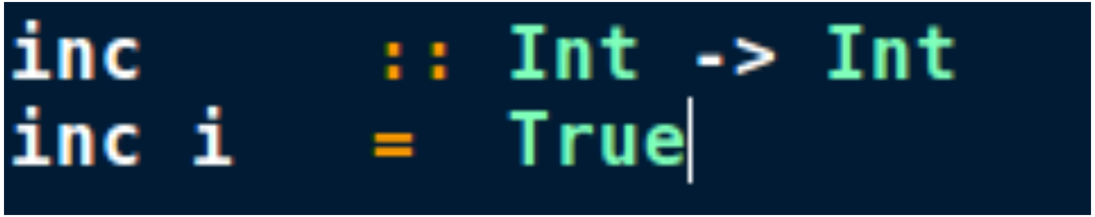


```haskell
inc         :: Int -> Int
inc
```

Foutmeldingen

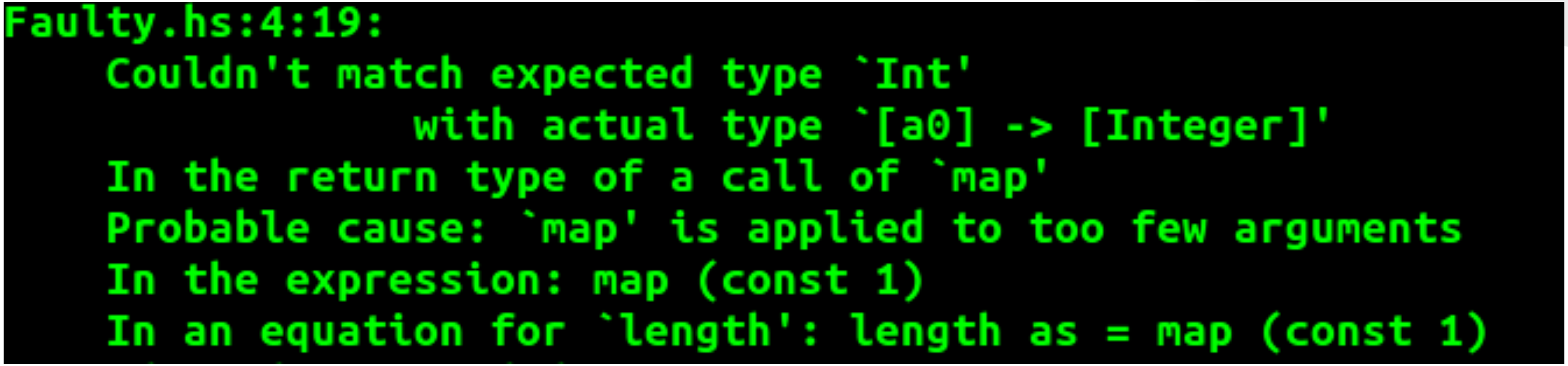


```
Faulty.hs:4:12:
    Couldn't match expected type `Int' with actual type `Bool'
    In the expression: True
    In an equation for `inc': inc i = True
```

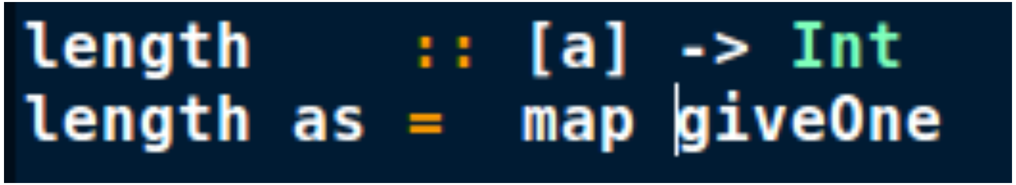


```
inc       :: Int -> Int
inc i     =  True
```

Foutmeldingen

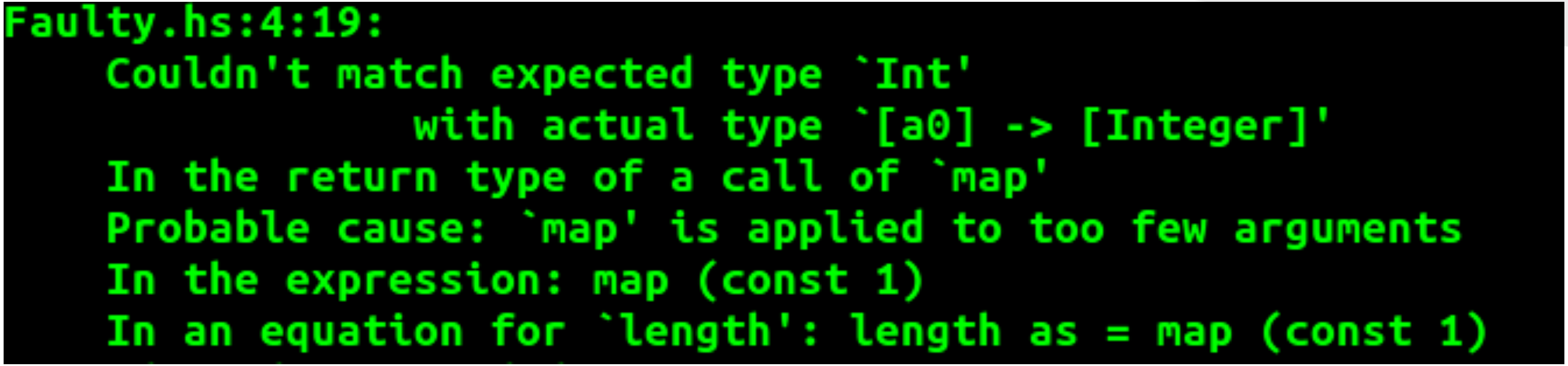


```
Faulty.hs:4:19:
    Couldn't match expected type `Int'
                with actual type `[a0] -> [Integer]'
    In the return type of a call of `map'
    Probable cause: `map' is applied to too few arguments
    In the expression: map (const 1)
    In an equation for `length': length as = map (const 1)
```

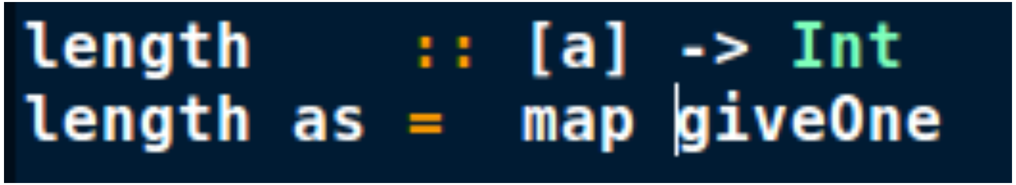


```
length     :: [a] -> Int
length as =  map giveOne
```

Foutmeldingen

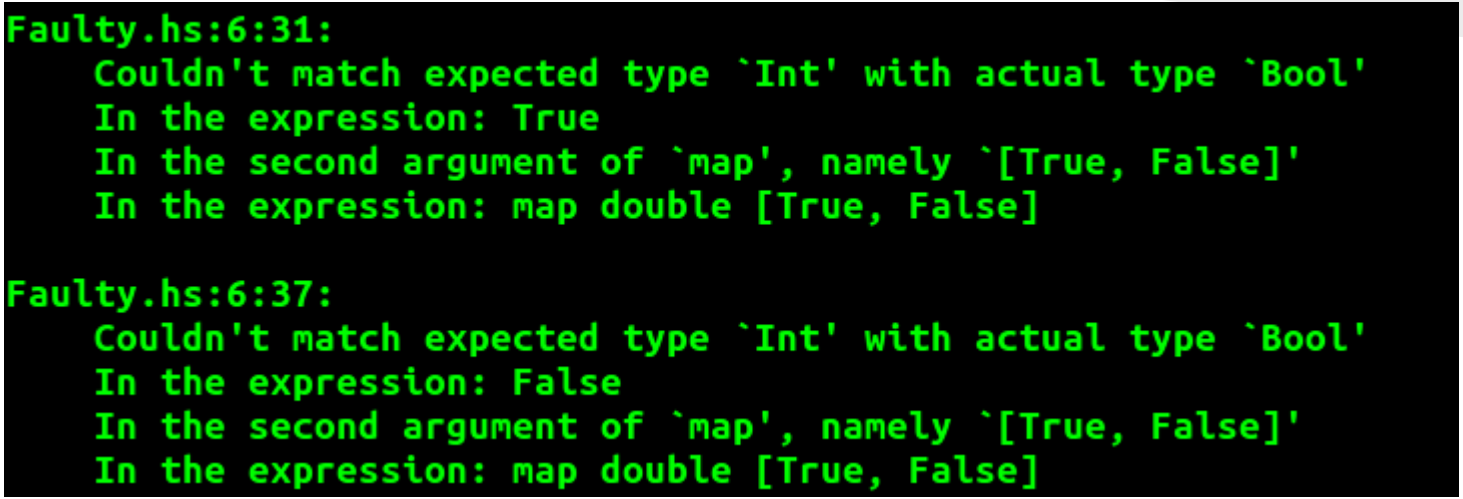


```
Faulty.hs:4:19:
    Couldn't match expected type `Int'
                with actual type `[a0] -> [Integer]'
    In the return type of a call of `map'
    Probable cause: `map' is applied to too few arguments
    In the expression: map (const 1)
    In an equation for `length': length as = map (const 1)
```



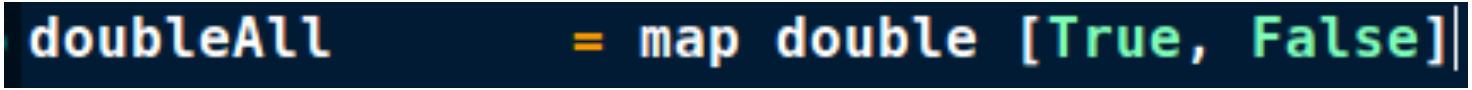
```
length     :: [a] -> Int
length as =  map giveOne
```

Foutmeldingen



```
Faulty.hs:6:31:
    Couldn't match expected type `Int' with actual type `Bool'
    In the expression: True
    In the second argument of `map', namely `[True, False]'
    In the expression: map double [True, False]

Faulty.hs:6:37:
    Couldn't match expected type `Int' with actual type `Bool'
    In the expression: False
    In the second argument of `map', namely `[True, False]'
    In the expression: map double [True, False]
```



```
doubleAll             = map double [True, False]
```

Oefeningen

Oefeningen

github.com/pietervdvn/haskell

Oplossingen komen later online