



HASKELL

By Marco and Jan



QUOTE: “...”

A Haskell developer after his script
did not compile for the 42th time!

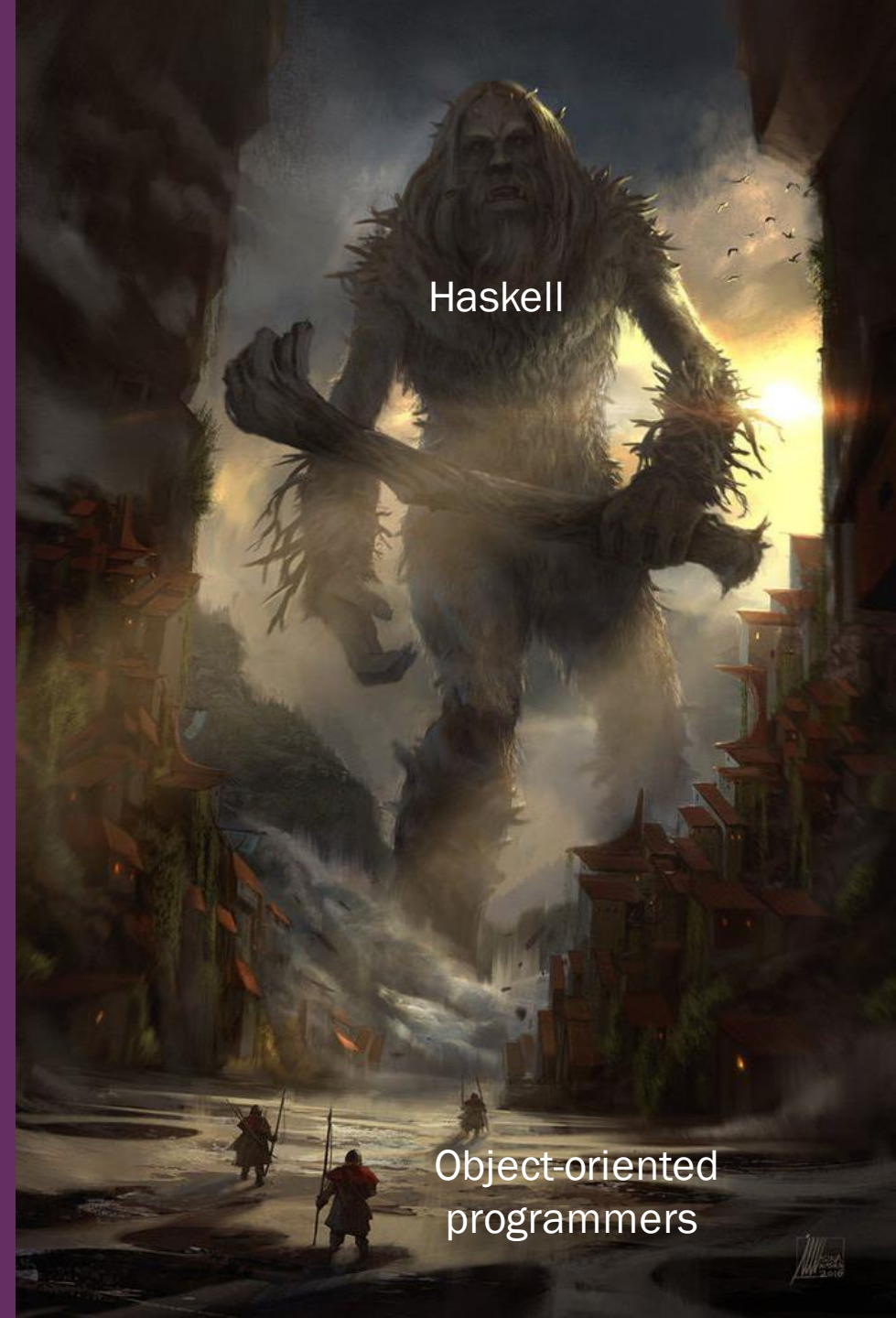
Contents

- Functional Programming: Principles
- Haskell? Why you cannot eat it.
- To Infinity and Beyond + Practical
- Where are my loops?
- Pure Functions
- We maybe have side-effects...
- Haskell in Practice: Conway's Game of Life

da!

Principles of Functional Programming

- Immutability
 - *Variables cannot be changed*
- No side-effects
- Pure Functions



Haskell

Object-oriented
programmers

So what is Haskell?

- Purely functional programming language
- You tell the computer what stuff is, rather than what to do (declarative style)
- Features:
 - *Type inference*
 - *Statically typed*
 - ***Lazyness***
 - *Many, many more...*



„You can be lazy, if you are smart!“

To Infinity and Beyond!

- Haskell
 - *evaluates lazy*
 - *supports infinite lists*

Attention: Infinite Lists might crash your IDE, if your not lazy enough!!!



INFINITE DATA STRUCTURES CHEATSHEET

```
----- lists by enumeration
x1 = [1, 9, 42]
x2 = [3 .. 7]           -- [3, 4, 5, 6, 7]
x3 = [3, 5 .. 10]       -- [3, 5, 7, 9]
x4 = [10, 9 .. 5]       -- [10, 9, 8, 7, 6, 5]
x5 = [100, 90 .. 60]    -- [100, 90, 80, 70, 60]
x6 = sum [1,2..10]      -- 55
x7 = product [1,2,3]    -- 6
x8 = length x1          -- 3

----- lists by comprehension
x9 = [ x + 1 | x <- x1 ]  -- [2, 10, 43]
x10 = [ sum [1..x] | x <- x3 ] -- [6, 15, 28, 45]
x11 = [ x | x <- [1,2,3,4,5,6], mod x 2 == 1 ] -- [1, 3, 5]

----- infinite lists
x12 = [5, 10 ..]         -- [5, 10, 15, 20, ...]
x13 = take 3 [5, 10 ..]  -- [5, 10, 15]
```



Where are my loops?

Nah, man... Use Higher Order Functions!

- Remember: imperative vs. declarative style
- But how do I work with lists then?
 - *MAP*
 - Transform elements in a list
 - *FOLD (left, right)*
 - Reduce list to one value
 - *ZIP (with)*
 - Combine elements from two lists
 - *FILTER*
 - well... removing some elements
- Do you remember **lambda** calculus?



da

while-loop

for-loop

dowhile-loop

foreach-loop

FOLD, MAP, ZIP, FILTER Cheatsheet

```
{-
Here you can find some simple examples on:
FILTER, MAP, ZIP and FOLD
-}

hundredIntegers = [1..100]

-- Filter Example
productsOfFive = filter (\ x -> x `mod` 5 == 0) hundredIntegers -- [5,10,15,20,25...]

-- Map Example
increasedInts = map (\ x -> x + 100) hundredIntegers -- [101,102,103,104,105,106...]

-- Zip Example
tuples = zip hundredIntegers [1..1000] -- [(1,1),(2,2),(3,3),(4,4)...]

-- Zipwith Example
noTuplesAnymore = zipWith (+) hundredIntegers [1..1000] -- [2,4,6,8,10,12,14...]

-- Fold Example (You have to start left or right)
reducedToOne = foldl1 (+) hundredIntegers -- 5050
```

Pure Functions and Haskell

- All functions take a parameter
- All functions must return a value
- Anytime a function is called with the same parameter it returns the same value
 - *Output depends ONLY on parameters*

Pure Functions in Practice

```
-- defines data type with two constructors
data Expr = Val Int | Sqrt Expr

-- define a evaluation function for both Expr types
eval :: Expr -> Int

-- if it is a value return it
eval (Val n) = n

-- if its a squareroot calculate it
eval (Sqrt n) = floor . sqrt $ (fromIntegral (eval n) :: Float)

-- calculate  $\sqrt{\sqrt{81}}$ 
testeval = eval (Sqrt (Sqrt (Val 81)))

-- this is a division ;)
theanswer = div 210 5
```

„No side-effects? You must be kidding!“

- In the real world things **maybe** go wrong...
- We cannot find a file **maybe**...
- **Maybe** Haskell knows a solution...



Maybe - Assignment

```
-- the constructors stay the same
data Expr = Val Int | Sqrt Expr

-- define a safe sqrt function
safesqrt :: Int -> Maybe Int
safesqrt x = if x < 0 then
    Nothing
    else
    Just (floor . sqrt $ (fromIntegral x :: Float))

-- the evaluation function maybe returns an integer now
eval :: Expr -> Maybe Int

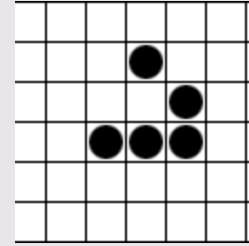
-- if it is a value just return it
eval (Val n) = Just n

-- if its a squareroot evaluate first
eval (Sqrt n) = case eval n of
    Nothing -> Nothing
    Just n -> safesqrt n

-- calculate  $\sqrt{\sqrt{81}}$ 
testeval = eval (Sqrt (Sqrt (Val 81)))

-- this is safe now
killme = eval (Sqrt (Sqrt (Val (-81))))
```

Conway's Game of Life



- Let's see, what a Haskell application could look like...
- And we come to Sebastian's task!