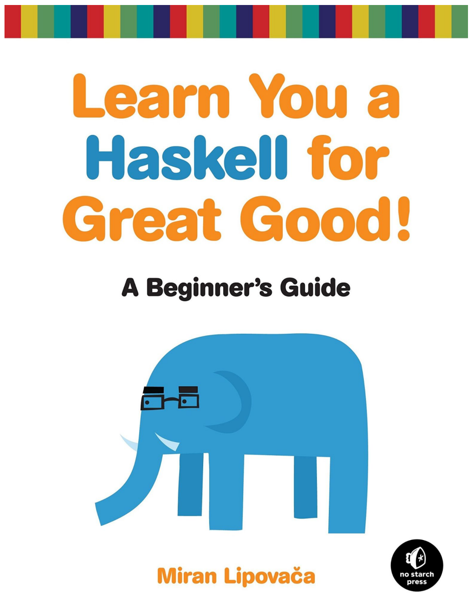
Haskell, A Purely Functional Programming Language

Authors: Joel Aumedes and Marc Cervera
Maria Teresa Alsinet - Language Processors
Superior Polytechnic School - University Of Lleida

Contents



Learn You a Haskell for Great Good!
A Beginner's Guide
Miran Lipovača
no starch press

Introduction

- Haskell is a purely functional programming language.
- Haskell is lazy, that means that unless specifically told otherwise, Haskell won't execute functions and calculate things until it's really forced to show you a result.
- Haskell is elegant and concise.

Introduction: How to use Haskell

- To dive into Haskell, you only need a text editor (Notepad++, vim, Atom, VisualStudio code...) and a Haskell compiler such as GHC.
- To start coding, you have to open a CMD and type: ghci.

```
admin@anonymous: ~$ ghci
GHCi, version 8.8.4: https://www.haskell.org/ghc/  :? for help
Prelude>
```

Starting out

Basic Arithmetic

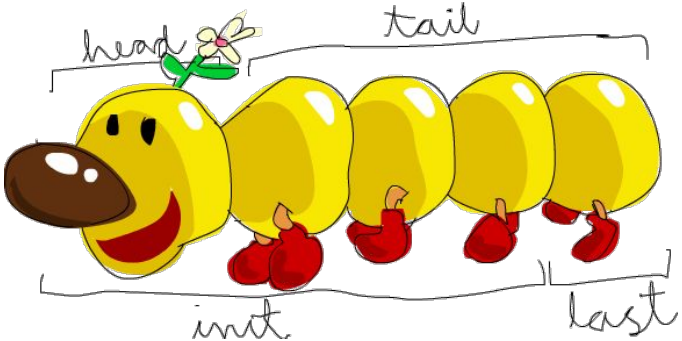
```
Prelude> 5 * 5
25
Prelude> 1000 / 2 * 3
1500.0
```

Using Lists

```
Prelude> 1:[2,3,4,5]
[1,2,3,4,5]
Prelude> “Hello“ ++ ”World”
HelloWorld
Prelude> head [0,1,2]
0
```

First functions

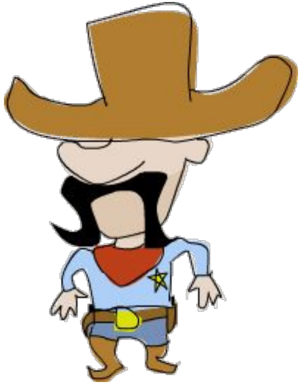
```
Prelude> max 5 12
12
Prelude> doubleMe x = x + x
Prelude> doubleMe 10
20
Prelude> isBig x = if x > 100 then “Yes!” else “No!!
```



Starting out

Ranges

```
Prelude> [1 .. 20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
Prelude> ['A' .. 'G']
"ABCDEFG"
Prelude> take 5 [10, 20..]
[10, 20, 30, 40, 50]
```



List comprehensions


```
Prelude> [x*2 | x <- [1 .. 10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> onlyUppercase xs = [c | c <- xs, c `elem` ['A' .. 'Z']]
Prelude> onlyUppercase "Hello My Friend!"
"HMF"
```

Types and Typeclasses

Everything in Haskell has a type…

```
Prelude> :t ‘a’
‘a’ :: Char
Prelude> :t False
False :: Bool
Prelude> :t “Hello!”
“Hello!” :: [Char]
```

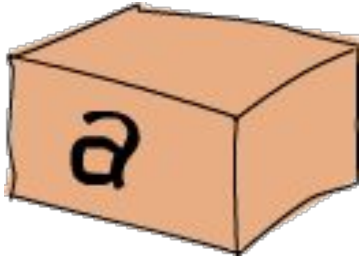
… even functions!

```
Prelude> shout x = x ++ “!!!”
Prelude> :t shout
shout :: [Char] -> [Char]
```

Int, Integer, Float, Double, Bool, Char

```
Prelude> :{
Prelude| addThree :: Int -> Int -> Int -> Int
Prelude| addThree x y z = x + y + z
Prelude| :}
Prelude> :t addThree
addThree :: Int -> Int -> Int -> Int
```

Types and Typeclasses



Type variables

```
Prelude> :t head
head :: [a] -> a
```

Typeclasses

```
Prelude> :t 19
19 :: Num p => p
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Eq, Ord, Show, Read, Num, Fractional

Syntax in Functions

Pattern matching

```
numeroCinc :: (Integral a) => a -> [Char]
numeroCinc 5 = "Number five!"
numeroCinc x = "Not a five"
Prelude> numeroCinc 5
“Number five!”
```

```
teams :: [Char] -> [Char]
teams “Ferrari” = “Very fast”
teams “McLaren” = “Not so fast”
Prelude> teams “Haas”
*** Exception: Non-exhaustive patterns in function teams
```

```
head’ :: [a] -> a
head’ [] = error “Head of an empty list!”
head’ (x:_) = x
```

Guards

```
diceResults :: (Integral a) => a -> [Char]
diceResults x
        | x == 1 = “Number 1!”
        | x == 2 = “Second!”
        | x == 3 = “The third!”
        | otherwise = “Very nice!”
```

Syntax in Functions

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
    | bmi <= 18.5 = "You are underweight!"
    | bmi <= 25.0 = "You have normal weight, not bad."
    | bmi <= 30.0 = "You are overweight, built like a croissant"
    | otherwise = "You are obese"
    where bmi = weight / height ^ 2
```

```
describeList :: [a] -> String
describeList xs = "The list is " ++ case xs of [] -> "empty."
                                               [x] -> "a singleton list."
                                               xs -> "a longer list."
```

Recursion

- Recursion is a way of defining functions in which the function is applied inside its own definition.
- Fun fact: Definitions in mathematics are often given recursively.
  - A clear example is the Fibonacci sequence, that is given recursively.
- Recursion is important to Haskell.

```
fac :: Int -> Int
fac n
        | n < 2 = 1
        | otherwise = n * fac $ n - 1
```

```
fac :: Int -> Int
fac n
        | n < 0 = error “Are you stupid?”
        | n == 0 = 1
        | n == 1 = 1
        | n > 1 = n * fac $ n - 1
```

Recursion

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
-- Very inefficient!!
```

```
invertedPair :: (Num a) => [a] -> [(a,a)]
invertedPair [] = []
invertedPair (x:xs) = (x, negate x):invertedPair xs
```

Higher order functions

Curried functions

```
Prelude> max 4 5
5
Prelude> (max 4) 5
5
```

```
Prelude> map (max 4) [1..6]
[4,4,4,4,5,6]
Prelude> applyTwice f x = f (f x)
Prelude> applyTwice (+3) 5
11
```

Lambdas

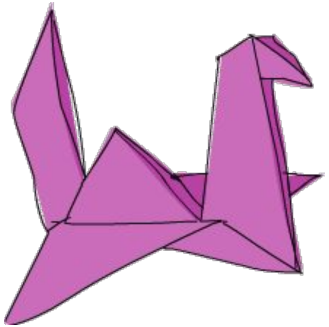
```
Prelude> filter (\x -> x > 4) [1..6]
[5,6]
Prelude> filter (>4) [1..6]
[5,6]
```

Higher Order Functions

Folds and scans

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
Prelude> sum’ [1..5]
15
```

```
sum' :: (Num a) => [a] -> a
sum' xs = foldr1 (\x acc -> acc + x) xs
Prelude> sum’ [1..5]
15
```



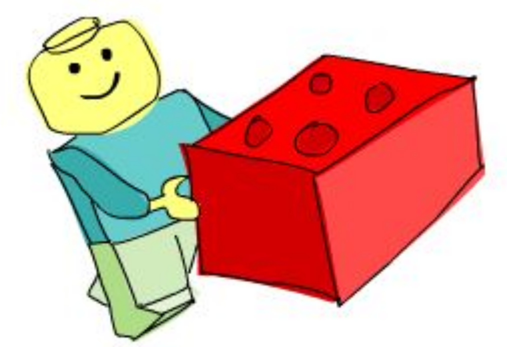
Function application

```
Prelude> length $ filter (>=10) [1..20]
11
```

Function composition

```
Prelude> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
Prelude> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

Modules



Importing a module

```
import Data.List
Prelude>:m + Data.List
```

Data.List, Data.Char, Data.Map, Data.Set, Text.Regex

Defining a module

```
module Conversions
( euroToDollar
, dollarToEuro
, celsiusToFahrenheit
, fahrenheitToCelsius)
where

euroToDollar = (*0.87)
        …
```

Making our own Types and Typeclasses

```
data Bool = False | True
```

```
data Content = Movie Int [[Char]] | TVShow Int Int [[Char]]
```

Pattern Matching

```
numberOfSeasons :: Content -> Int
numberOfSeasons (Movie _ _) = error "Not a TV Show!"
numberOfSeasons (TVShow seasons _ _) = seasons
```

Record Syntax

```
type String = [Char]
data Car = Car { model :: String,
                 make :: String,
                 year :: Int } deriving (Show)
```

Type parameters

```
data Maybe a = Nothing | Just a
```

Input and output

```
main = putStrLn “Hello, world!”
```

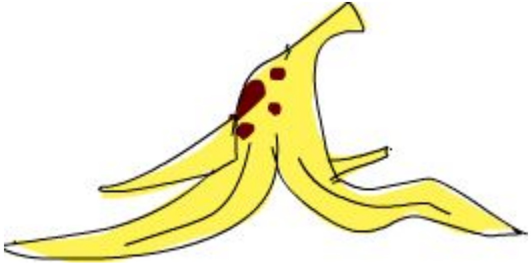
```
main = do
        putStrLn “What’s your name?”
        name <- getLine
        putStrLn “Nice to meet you, “ ++ name
```

```
main = do
        contents <- getContents
        print $ length $ filter (==’\n’) contents
```

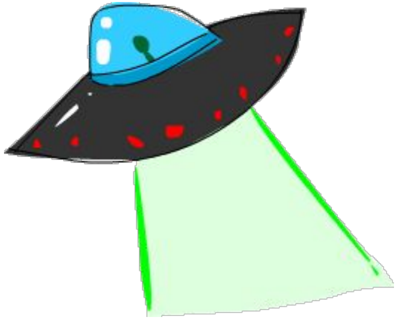
What’s next?

http://learnyouahaskell.com/chapters

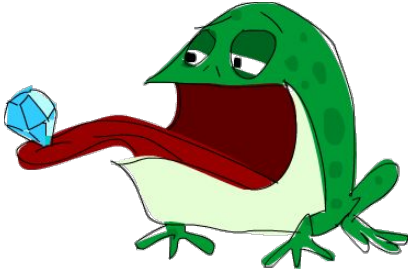
Functors



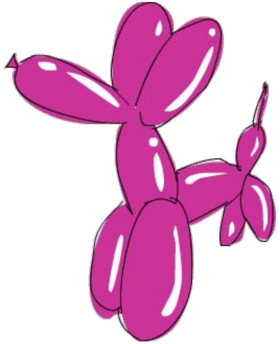
Applicative functors



Monads



Monoids



Thank you for your attention!