>>> Functional Programming
>>> Using the example of Haskell

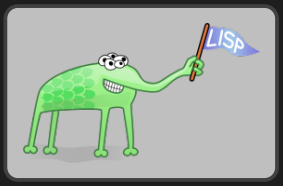
Name: Amedeo Molnár - nek0@nek0.eu
Date: 13. Januar 2020

>>> Overview

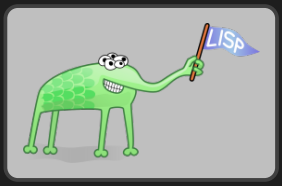
1. Introduction
2. Main differences to imperative programming
3. Examples
4. Fields of Application
5. Conclusion

>>> Languages

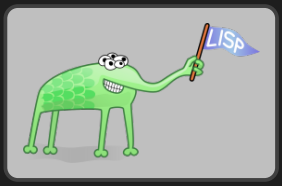
>>> Languages



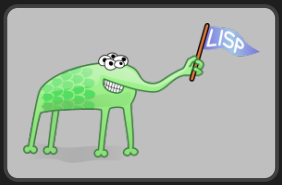
>>> Languages



>>> Languages



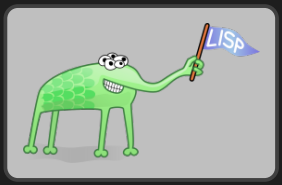
>>> Languages



OCaml

elixir

>>> Languages



OCaml

elixir

>>> Core Concepts

* Purity

* Non-imperativeness

* First Class Citizenship

* Closures

* Lambdas

>>> Core Concepts

* Purity
  A pure function must produce the same result given the same input and does not rely on or alter external state.
* Non-imperativeness

* First Class Citizenship

* Closures

* Lambdas

>>> Core Concepts

* Purity
  A pure function must produce the same result given the same input and does not rely on or alter external state.
* Non-imperativeness
  A function is not a sequence of commands, but a nesting of other functions.
* First Class Citizenship

* Closures

* Lambdas

>>> Core Concepts

* Purity
  A pure function must produce the same result given the same input and does not rely on or alter external state.
* Non-imperativeness
  A function is not a sequence of commands, but a nesting of other functions.
* First Class Citizenship
  Functions are equal to other data objects and can thus be passed as function arguments or be computation results themselves.
* Closures

* Lambdas

>>> Core Concepts

* Purity
  A pure function must produce the same result given the same input and does not rely on or alter external state.
* Non-imperativeness
  A function is not a sequence of commands, but a nesting of other functions.
* First Class Citizenship
  Functions are equal to other data objects and can thus be passed as function arguments or be computation results themselves.
* Closures
  Functions can only access variables inside context they have been created. This is possible even when the function itself has left this context. In this case the variable values are frozen at the moment of departure inside the function.
* Lambdas

>>> Core Concepts

* Purity
  A pure function must produce the same result given the same input and does not rely on or alter external state.
* Non-imperativeness
  A function is not a sequence of commands, but a nesting of other functions.
* First Class Citizenship
  Functions are equal to other data objects and can thus be passed as function arguments or be computation results themselves.
* Closures
  Functions can only access variables inside context they have been created. This is possible even when the function itself has left this context. In this case the variable values are frozen at the moment of departure inside the function.
* Lambdas
  A function definition can take place without an explicit name in the position of a function symbol.

>>> Main differences to imperative programming

>>> Main differences to imperative programming

* Avoidance of side effects

>>> Main differences to imperative programming

* Avoidance of side effects
* Immutable variables

>>> Main differences to imperative programming

* Avoidance of side effects
* Immutable variables
* No loops

>>> Main differences to imperative programming

* Avoidance of side effects
* Immutable variables
* No loops
* No state

>>> Examples

```haskell
sq :: (Floating a) => a -> a
sq x = x * x
```

```haskell
ringArea :: (Floating a) => a -> a -> a
ringArea r1 r2 = pi * (sq r1 - sq r2)
```

```haskell
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib n = fib (n - 2) + fib (n - 1)
```

>>> Advanced Example

```haskell
gifts :: [String]
gifts =
  [ "And a partridge in a pear tree!", "Two turtle doves,", "Three french hens,"
  , "Four calling birds,", "Five golden rings,", "Six geese a-laying,"
  , "Seven swans a-swimming,", "Eight maids a-milking,", "Nine ladies dancing,"
  , "Ten lords a-leaping,", "Eleven pipers piping,", "Twelve drummers drumming,"
  ]

days :: [String]
days = [
  "first", "second", "third", "fourth", "fifth", "sixth", "seventh", "eighth",
  "ninth", "tenth", "eleventh", "twelfth" ]

verseOfTheDay :: Int -> String
verseOfTheDay day =
  "On the " ++ days !! day ++ " day of Christmas my true love gave to me... \n"
  ++ concat (map (++ "\n") [dayGift day d | d <- [day, day-1..0]]) ++ "\n"
  where
    dayGift 0 _ = "A partridge in a pear tree!"
    dayGift _ gift = gifts !! gift

main :: IO ()
main = putStrLn (concatMap verseOfTheDay [0..11])
```

>>> Fields of Application

>>> Fields of Application

* Big Data

>>> Fields of Application

* Big Data
* Finance Sector

>>> Fields of Application

* Big Data
* Finance Sector
* Science

>>> Fields of Application

* Big Data
* Finance Sector
* Science
* Virtually everywhere

>>> Literature recommendations

* Learn you a Haskell for Great Good!
* Real World Haskell
* Parallel and Concurrent Programming in Haskell
* The Haskell School of Music

Thank you for your attention!

This presentation is available for download at:
https://github.com/nek0/presentation-fp-haskell