Haskell and Scala

Adam Szlachta

Introductio

History

programming

programming

- ..

c....

Dynicax Sammary

Algebraic data

types

Monadic feature

Summany

Haskell and Scala

comparison

Adam Szlachta

March 20, 2013

ver. 1.0 adam.szlachta@gmail.com

Introduction

Haskell and Scala

Adam Szlachta

Introduction

History

Functional programmin

Pasis syntax

Functions

Syntax summary

Algebraic dat

types

Monadic feature

ummary

■ Haskell and Scala introduction

- History
- Functional programming essence
- Syntax and features comparison
- Scala libraries influenced by Haskell
- Summary

Features incorporated in Scala

Haskell and Scala

Adam Szlachta

and the second

. . .

Functional

programming

Eupetions

Syntax summan

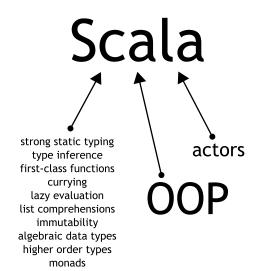
. .

Algebraic data

types

Monadic featur

Summary



Languages which influenced Scala

Haskell and Scala

Adam Szlachta

Histon

Functional

programmin

Functions

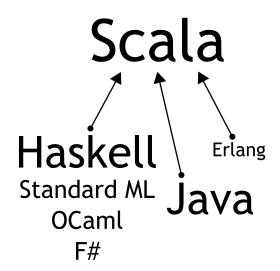
Syntax Summar

Algebraic dat

types

Monadic feature

Summary



Haskell logo and name

Haskell and Scala

Adam Szlachta

. .

History

programmir

Basic synta

Functions

Syntax summar

Laziness

Algebraic data

types

Monadic feature

Summa



From Wikipedia

Lambda calculus (also written as λ -calculus or called "the lambda calculus") is a formal system in mathematical logic and computer science for expressing computation by way of variable binding and substitution.

From Wikipedia

Haskell Brooks Curry (1900-1982) was an American mathematician and logician. Curry is best known for his work in combinatory logic.



Haskell logo and name

Haskell and Scala

Adam Szlachta

Introducti

History

programmir

Basic syntax

Functions

Syntax summar

Algobraio data

types

Classes

Monadic feature

Summar



From Wikipedia

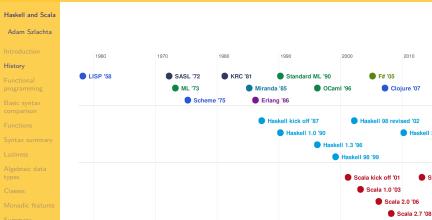
Lambda calculus (also written as λ -calculus or called "the lambda calculus") is a formal system in mathematical logic and computer science for expressing computation by way of variable binding and substitution.

From Wikipedia

Haskell Brooks Curry (1900-1982) was an American mathematician and logician. Curry is best known for his work in combinatory logic.



Functional programming languages history



2010

Haskell 2010 '10

Scala 2.9 '11

Scala 2.10 '13

2020

Haskell and Scala

Adam Szlachta

Functional

programming

avoiding side effects

- avoiding state (mutable data)
- referential transparency and lazy evaluation
- first-class functions
- based on theories
 - λ -calculus (α -conversion, β -reduction, η -conversion)
 - category theory

Haskell and Scala

Adam Szlachta

Functional

programming

- avoiding side effects
- avoiding state (mutable data)
- referential transparency and lazy evaluation
- first-class functions
- based on theories
 - λ -calculus (α -conversion, β -reduction, η -conversion)
 - category theory

Haskell and Scala

Adam Szlachta

Carlotte State

Functional programming

Basic syntax

Eupstions

Suntay summan

Algebraic dat

types

Monadic feature

ivioliadic icature

avoiding side effects

avoiding state (mutable data)

referential transparency and lazy evaluation

first-class functions

based on theories

lacktriangle λ -calculus (α -conversion, β -reduction, η -conversion)

category theory

Haskell and Scala

Adam Szlachta

Functional

programming

_ ...

Algebraic data

types

Monadic feature

_

avoiding side effects

avoiding state (mutable data)

referential transparency and lazy evaluation

first-class functions

based on theories

lacktriangle λ -calculus (α -conversion, β -reduction, η -conversion)

category theory

Haskell and Scala

Adam Szlachta

Functional

programming

F.....

Suntay summan

Algebraic data

types

Classes

Monadic feature

Summar

- avoiding side effects
- avoiding state (mutable data)
- referential transparency and lazy evaluation
- first-class functions
- based on theories
 - λ -calculus (α -conversion, β -reduction, η -conversion)
 - category theory

Haskell and Scala

Adam Szlachta

. . .

Functional

programming

companse

Syntax Samma

Algebraic data

types

.

Monadic feature

Summan

- avoiding side effects
- avoiding state (mutable data)
- referential transparency and lazy evaluation
- first-class functions
- based on theories
 - \bullet λ -calculus (α -conversion, β -reduction, η -conversion)
 - category theory

Haskell and Scala

Adam Szlachta

Histor

Functional programming

Basic syntax

Functions

Syntax summary

Algebraic data

types

Monadic feature

_

avoiding side effects

avoiding state (mutable data)

referential transparency and lazy evaluation

first-class functions

based on theories

■ λ -calculus (α -conversion, β -reduction, η -conversion)

category theory

Hello, World!

```
Haskell and Scala
```

Adam Szlachta

.....

. .

Function

Basic syntax

comparison

Functions

Syntax summary

A1 -1 -1 -1 -1 -1

types

Monadic feature

vionadic features

Summary

```
module Main where
main :: IO ()
main = putStrLn "Hello, World!"
```

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, World!")
  }
}
```

Haskell

Scala

Hello, World!

```
Haskell and Scala
```

Adam Szlachta

......

History

Functiona

Basic syntax

comparison

Functions

Syntax summary

types

Monadic feature

vionadic teatures

Summary

```
module Main where
main :: IO ()
main = putStrLn "Hello, World!"
```

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, World!")
  }
}
```

Haskell

Scala

Who is this?

Haskell and Scala

Adam Szlachta

.....

History

Functional programming

Basic syntax

Functions

Syntax summary

Algebraic data

types

Classes

Monadic features

Summany



Haskell and Scala

Adam Szlachta

History

Functiona programm

Basic synta

Functions

Syntax summary

. .

Algebraic data

types

Monadic feature

Summary

The Polish Parliament meets in the capital of Poland.

The Polish Parliament meets in Warsaw.

Warsaw has been the capital of Poland since 1815.

Haskell and Scala

Adam Szlachta

. .

Function

Basic synta

comparison

Functions

Syntax summary

..

types

Classes

Monadic feature

ummary

The Polish Parliament meets in the capital of Poland.

The Polish Parliament meets in Warsaw.

Warsaw has been the capital of Poland since 1815

Haskell and Scala

Adam Szlachta

Histor

Function

Basic synta

Functions

Syntax summary

Laziness

Algebraic data

types

Monadic feature

Summary

The Polish Parliament meets in the capital of Poland.

The Polish Parliament meets in Warsaw.

Warsaw has been the capital of Poland since 1815.

Haskell and Scala

Adam Szlachta

. .

Function

Basic synta:

comparison

Functions

Syntax summary

Algebraic data

types

Monadic featur

Summary

From Wikipedia

Referential transparency is a property whereby an expression can be replaced by its value without affecting the program.

Example:

```
text = reverse "redrum"
```

can be replaced with:

```
text = "murder'
```

Haskell and Scala

Adam Szlachta

Histo

Functional programmi

Basic synta:

Functions

Syntax summary

. .

Algebraic data

types

Monadic feature

Summary

From Wikipedia

Referential transparency is a property whereby an expression can be replaced by its value without affecting the program.

Example:

text = reverse "redrum"

can be replaced with:

text = "murder'

Haskell and Scala

Adam Szlachta

Histo

Functiona programm

Basic syntax

Functions

c .

Syricax Sammary

Algebraic data

Lypes

Monadic featur

Summary

From Wikipedia

Referential transparency is a property whereby an expression can be replaced by its value without affecting the program.

Example:

text = reverse "redrum"

can be replaced with:

text = "murder"

Who is this?

Haskell and Scala

Adam Szlachta

History

Functional

Basic syntax

Functions

Syntax summary

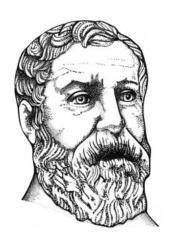
. .

Algebraic data

types

Monadic feature

Summary



Heron's formula

Haskell and Scala

Adam Szlachta

. . . .

History

Functional

Basic syntax

Functions

Syntax summan

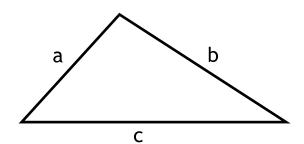
Algebraic data

types

Monadic feature

Summary

$$T = \sqrt{s(s-a)(s-b)(s-c)}$$
, where $s = \frac{a+b+c}{2}$



Function definition

Haskell and Scala

Adam Szlachta

. . . .

History

Functional

programmin

Functions

c.....

Algebraic dat

сурса

Monadic feature

Summary

```
triangleArea :: Double -> Double -> Double -> Double
triangleArea a b c =
   let s = (a + b + c) / 2 in
   sqrt (s * (s - a) * (s - b) * (s - c))
```

Haskell

```
def triangleArea(a: Double, b: Double, c: Double): Double = {
   val s = (a + b + c) / 2
   return Math.sqrt (s * (s - a) * (s - b) * (s - c))
```

Function definition

Haskell and Scala

Adam Szlachta

. . . .

History

Functional

programmin

Functions

Syntax Summar

Laziness

Algebraic dat

Classe

Monadic feature

Summary

```
triangleArea :: Double -> Double -> Double
triangleArea a b c =
  let s = (a + b + c) / 2 in
  sqrt (s * (s - a) * (s - b) * (s - c))
```

Haskell

```
def triangleArea(a: Double, b: Double, c: Double): Double = {
  val s = (a + b + c) / 2
  return Math.sqrt (s * (s - a) * (s - b) * (s - c))
}
```

Function definition

Haskell and Scala

Adam Szlachta

Introductio

Histor

Functiona

Basic syntax

Functions

Syntax summary

Syricax sammary

Algebraic data

types

Monadic featur

ummarv

```
triangleArea a b c =
  let s = (a + b + c) / 2 in
  sqrt (s * (s - a) * (s - b) * (s - c))
```

```
triangleArea a b c =
    sqrt (s * (s - a) * (s - b) * (s - c))
    where
        s = (a + b + c) / 2
```

History

Functional programming

programmin

print \$ add5 10

add x y = x + yadd5 = add 5

Functions

Syntax summary

Syntax Sammar

Algebraic data

types

Monadic feature

wionadic leatures

Summary

```
def add(x: Int)(y: Int) = x + y
```

$$def add5 = add(5)_{-}$$

println (add5(10))

ocala

```
Haskell and Scala
```

Introduction

Functional

programming

Functions

Syntax summary

Lastanas

Algebraic data

Classes

Monadic feature

Summar

```
add1 :: [Int] -> [Int]
add1 xs = map (\x -> x + 1) xs

sum :: [Int] -> Int
sum xs = foldr (\x y -> x + y) 0 xs

add1 :: [Int] -> [Int]
add1 xs = map (+ 1) xs

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

```
def add1(xs: List[Int]): List[Int] = xs.map(x => x + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)((x, y) => x + y)
    def add1(xs: List[Int]): List[Int] = xs.map(_ + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)(_ + _)
```

laskell

◆□▶◆周▶◆団▶◆団▶ ■ 釣Q@

```
Haskell and Scala
```

............

Functional

programming

Functions

- unctions

Sylicax sullillary

Laziness

types

Classes

Monadic feature

Summar

```
add1 :: [Int] -> [Int]
add1 xs = map (\x -> x + 1) xs

sum :: [Int] -> Int
sum xs = foldr (\x y -> x + y) 0 xs

add1 :: [Int] -> [Int]
add1 xs = map (+ 1) xs

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

```
def add1(xs: List[Int]): List[Int] = xs.map(x => x + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)((x, y) => x + y)

def add1(xs: List[Int]): List[Int] = xs.map(_ + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)(_ + _)
```

```
Haskell and Scala
```

......

Functional

programming

Functions

Syntay summany

-,....

Algebraic data

types

Classes

Monadic features

Summary

```
add1 :: [Int] -> [Int]
add1 xs = map (\x -> x + 1) xs

sum :: [Int] -> Int
sum xs = foldr (\x y -> x + y) 0 xs

add1 :: [Int] -> [Int]
add1 xs = map (+ 1) xs

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

```
def add1(xs: List[Int]): List[Int] = xs.map(x => x + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)((x, y) => x + y)

def add1(xs: List[Int]): List[Int] = xs.map(_ + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)(_ + _)
```

```
Haskell and Scala
```

......

. .

Functional

programming

Functions

Syntay summany

-,....,

Algebraic data

types

Monadic feature

```
Summary
```

```
add1 :: [Int] -> [Int]
add1 xs = map (\x -> x + 1) xs

sum :: [Int] -> Int
sum xs = foldr (\x y -> x + y) 0 xs

add1 :: [Int] -> [Int]
add1 xs = map (+ 1) xs

sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

```
def add1(xs: List[Int]): List[Int] = xs.map(x => x + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)((x, y) => x + y)

def add1(xs: List[Int]): List[Int] = xs.map(_ + 1)

def sum(xs: List[Int]): Int = xs.foldRight(0)(_ + _)
```

Point-free notation

Haskell and Scala

Adam Szlachta

Functions

Standard notation:

```
double x = 2*x
```

sum xs = foldr (+) 0 xs

Point-free notation

Haskell and Scala

Adam Szlachta

Introductio

Histor

Function: programm

Basic syntax

Functions

Syntax summary

Algebraic data

types

Classe

Monadic feature

Summary

Standard notation:

```
double x = 2*x

sum xs = foldr (+) 0 xs
```

Point-free notation:

Syntax

Haskell and Scala

Adam Szlachta

ntroduction

History

programmi Pasis contr

comparison

Lunctions

Syntax summary

Lazilles

Algebraic dat

types

Monadic featur

Summary

	Haskell	Scala	Python	Java
semicollons	optional	optional	optional	obligatory
curly brackets	optional	yes***	no	yes
significant indentation	yes	no	yes	no
type inference	yes	yes	dynamic	no
functions definitions	whitespace	()*	()	()
functions call	whitespace	()**	()	()
point-free notation	yes	no	no	no

^{*} optional for arity-0

^{**} optional for arity-0 and arity-1

^{***} optional for purely functional bodies (but without val definitions)

Strict and non-strict semantics

Haskell and Scala

Adam Szlachta

Little-Land

Function

Basic syntax

comparison

Functions

Syntax summary

Laziness

Algebraic data

Cl

Monadic feature

Summar

Meaning

Lazy evaluation means evaluating expression only when it is needed.

Meaning

Non-strictness means that the evaluation of expressions proceed from the outside (e.g. from '+' in (a + (b*c))). Usually identified with lazy evaluation.

Note

Useless for not purely functional computations!

Strict and non-strict semantics

Haskell and Scala

Adam Szlachta

_

programmir

Basic synta: comparison

Functions

Laziness

Syntax summary

Syntax summar

Algebraic data

Classes

Monadic feature

Summar

Meaning

Lazy evaluation means evaluating expression only when it is needed.

Meaning

Non-strictness means that the evaluation of expressions proceed from the outside (e.g. from '+' in (a + (b*c))). Usually identified with lazy evaluation.

Note

Useless for not purely functional computations!

Strict and non-strict semantics

Haskell and Scala

Adam Szlachta

Histo

Functiona

Basic synta:

Functions

Syntax summar

Laziness
Algebraic data

types

Monadic featur

Summar

Meaning

Lazy evaluation means evaluating expression only when it is needed.

Meaning

Non-strictness means that the evaluation of expressions proceed from the outside (e.g. from '+' in (a + (b*c))). Usually identified with lazy evaluation.

Note

Useless for not purely functional computations!

Lazy values

Haskell and Scala

Adam Szlachta

Laziness

```
lazyArgument = g (f x)
```

lazy val lazyValue = g(f(x))

Lazy values

Haskell and Scala

Adam Szlachta

. . .

Functional

programming

· unction.

Sylltax Sull

Laziness

Algebraic dat

types

Monadic feature

Summany

```
lazyArgument = g (f x)
```

$$lazyArgument = g$$
\$ f x

strictArgument = g \$! f x

lazy val lazyValue = g(f(x))

Haskell

Lazy values

Haskell and Scala

Adam Szlachta

.....

History

Functional

programming

c .

Syntax summary

Laziness

Algebraic data

types

Monadic features

Monadic features

Summary

```
lazyArgument = g (f x)
lazyArgument = g $ f x
```

strictArgument = g \$! f x

```
lazy val lazyValue = g(f(x))
```

val strictValue = g(f(x))

Haskell

cala

Infinite streams

Haskell and Scala

Adam Szlachta

Laziness

take 10 [1..]

[1,2,3,4,5,6,7,8,9,10]

Stream.from(1).take(10).toList

List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Haskell and Scala

Adam Szlachta

ntroduction

History

Functional programmin

Basic syntax

Functions

Syntax summary

. .

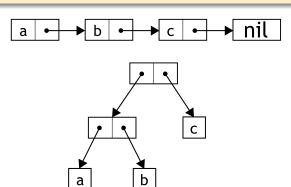
Algebraic data types

Class

Monadic features

From Wikipedia

Algebraic data type is a kind of composite type, i.e. a type formed by combining other types. Two common classes of algebraic type are product types, i.e. tuples and records, and sum types, also called tagged unions or variant types.



```
Haskell and Scala
Adam Szlachta
```

Algebraic data

types

```
data Boolean = True | False
```

```
trait Boolean
case class True extends Boolean
case class False extends Boolean
```

```
Haskell and Scala
```

Introduction

History

Functional programmin

Pacie cuntav

F.....

Syntax summary

Algebraic data

types

Monadic feature

```
trait Boolean
case class True extends Boolean
case class False extends Boolean

trait List[A]
case class Nil[A]() extends List[A]
case class Cons[A](v: A, 1: List[A]) extends List[A]

trait Tree[A]
case class Empty[A]() extends Tree[A]
case class Leaf[A](v: A) extends Tree[A]
case class Branch[A](1: Tree[A], r: Tree[A]) extends Tree[A]
```

```
Haskell and Scala
```

Introduction

History

Functional programming

programming

_ ..

Syntax summan

-,....,

Algebraic data

types

Manadia Gastina

ivionadic features

```
Summary
```

```
trait Boolean
case class True extends Boolean
case class False extends Boolean

trait List[A]
case class Nil[A]() extends List[A]
case class Cons[A](v: A, 1: List[A]) extends List[A]

trait Tree[A]
case class Empty[A]() extends Tree[A]
case class Leaf[A](v: A) extends Tree[A]
case class Branch[A](1: Tree[A], r: Tree[A]) extends Tree[A]
```

```
Haskell and Scala
```

Introduction

History

Functional

programming

. .

c .

Algebraic data

types

Monadic feature

```
trait Boolean
case class True extends Boolean
case class False extends Boolean

trait List[A]
case class Nil[A]() extends List[A]
case class Cons[A](v: A, 1: List[A]) extends List[A]

trait Tree[A]
case class Empty[A]() extends Tree[A]
case class Leaf[A](v: A) extends Tree[A]
case class Branch[A](1: Tree[A], r: Tree[A]) extends Tree[A]
```

Algebraic data

types

Note

Algebraic data type can be recursive and act as unions, structs and enums

```
data DaysOfWeek = Monday | Tuesday | Wednesday | Thursday
    | Friday | Saturday | Sunday
```

Algebraic data

Note

Algebraic data type can be recursive and act as unions, structs and enums

```
data DaysOfWeek = Monday | Tuesday | Wednesday | Thursday
    | Friday | Saturday | Sunday
data Account = Account
    { number :: Int
    . firstName :: String
    , lastName :: String
    . balance
               :: Float }
```

Algebraic data

Note

Algebraic data type can be recursive and act as unions, structs and enums

```
data DaysOfWeek = Monday | Tuesday | Wednesday | Thursday
    | Friday | Saturday | Sunday
data Account = Account
    { number :: Int
    . firstName :: String
    , lastName :: String
    . balance
                :: Float }
data Tree = Branch { left
                           :: Tree
                   , value :: Int
                     right :: Tree }
            Leaf { value :: Int }
```

History

Functional

programming

Functions

Syntax summary

Laziness

Algebraic data

types

Classes

Monadic feature

Summar

Note

Algebraic data type can be recursive and act as unions, structs and enums.

```
ta Account = Account
{ number :: Int
, firstName :: String
, lastName :: String
, balance :: Float }
```

data Account = Account Int String String Float

data Tree = Branch Tree Int Tree | Leaf Int

Hasket

```
Haskell and Scala
            data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
Adam Szlachta
            print $ treeToString $ Branch (Branch (Leaf 2) (Leaf 3)) (Leaf 4)
            trait Tree[A]
            case class Empty[A]() extends Tree[A]
Algebraic data
            case class Leaf[A](v: A) extends Tree[A]
types
            case class Branch[A](1: Tree[A], r: Tree[A]) extends Tree[A]
            println(treeToString(Branch(Branch(Leaf(2), Leaf(3)), Leaf(4))))
```

◆□▶ ◆□▶ ◆■▶ ◆■ ◆○○○

```
Haskell and Scala
            data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
Adam Szlachta
            treeToString :: Show a => Tree a -> String
            treeToString t = case t of
                Emptv -> "emptv"
                Leaf a -> "leaf " ++ show a
                Branch 1 r -> "branch[" ++ treeToString 1 ++
                                     " " ++ treeToString r ++ "]"
            print $ treeToString $ Branch (Branch (Leaf 2) (Leaf 3)) (Leaf 4)
            trait Tree[A]
            case class Empty[A]() extends Tree[A]
Algebraic data
            case class Leaf[A](v: A) extends Tree[A]
            case class Branch[A](1: Tree[A], r: Tree[A]) extends Tree[A]
            def treeToString[A](t: Tree[A]): String = t match {
              case Empty() => "empty"
              case Leaf(a) => "leaf " + a
              case Branch(1, r) => "branch[" + treeToString(1) +
                                          " " + treeToString(r) + "]"
            }
            println(treeToString(Branch(Branch(Leaf(2), Leaf(3)), Leaf(4))))
```

◆□▶ ◆□▶ ◆■▶ ◆■ ◆○○○

```
Haskell and Scala
            data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
Adam Szlachta
            treeToString t = case t of
                Emptv -> "emptv"
                Leaf a -> "leaf " ++ show a
                Branch 1 r -> "branch[" ++ treeToString 1 ++
                                     " " ++ treeToString r ++ "]"
            print $ treeToString $ Branch (Branch (Leaf 2) (Leaf 3)) (Leaf 4)
            trait Tree[A]
            case class Empty[A]() extends Tree[A]
Algebraic data
            case class Leaf[A](v: A) extends Tree[A]
            case class Branch[A](1: Tree[A], r: Tree[A]) extends Tree[A]
            def treeToString[A](t: Tree[A]): String = t match {
              case Empty() => "empty"
              case Leaf(a) => "leaf " + a
              case Branch(1, r) => "branch[" + treeToString(1) +
                                           " " + treeToString(r) + "]"
            }
            println(treeToString(Branch(Branch(Leaf(2), Leaf(3)), Leaf(4))))
```

◆□▶ ◆□▶ ◆■▶ ◆■ ◆○○○

```
Haskell and Scala
Adam Szlachta
            data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
            treeToString t = case t of
                Empty -> "empty"
                Leaf a -> "leaf " ++ show a
                Branch 1 r -> "branch[" ++ treeToString 1 ++
                                      " " ++ treeToString r ++ "]"
            treeToString Empty = "empty"
            treeToString (Leaf a) = "leaf " ++ show a
Algebraic data
            treeToString (Branch 1 r) =
                "branch[" ++ treeToString 1 ++ " " ++ treeToString r ++ "]"
            print $ treeToString $ Branch (Branch (Leaf 2) (Leaf 3)) (Leaf 4)
            "branch[branch[leaf 2 leaf 3] leaf 4]"
```

Default implementations

Haskell and Scala

Adam Szlachta

Classes

```
class Equal a where
    (===), (/==) :: a -> a -> Bool
    x /== y = not $ x === y
```

```
trait Equal[_] {
  def ===(x: Equal[_]): Boolean
  def /==(x: Equal[_]): Boolean = !(this === x)
```

Default implementations

```
Haskell and Scala
```

Adam Szlachta

Classes

```
instance Eq a => Equal (Tree a) where
    Empty === Empty = True
   Leaf x === Leaf y = x == y
    Branch 11 r1 === Branch 12 r2 = 11 === 12 && r1 === r2
    === = False
```

Default implementations

```
Haskell and Scala
```

Adam Szlachta

Introduction

HISTORY

Functional programmin

compariso

Functions

Syntax Summary

Laziness

types

Classes

ivionadic leatures

```
instance Eq a => Equal (Tree a) where

Empty === Empty = True

Leaf x === Leaf y = x == y

Branch 11 r1 === Branch 12 r2 = 11 === 12 && r1 === r2

==== _ = False
```

```
trait Tree[A] extends Equal[A]
case class Empty[A]() extends Tree[A] {
  def ===(x: Equal[_]): Boolean = x match {
    case Empty() => true
   case => false
7
case class Leaf[A](v: A) extends Tree[A] {
  def ===(x: Equal[_]): Boolean = x match {
    case Leaf(v1) => v == v1
   case _ => false
case class Branch[A](1: Tree[A], r: Tree[A]) extends Tree[A] {
  def ===(x: Equal[]): Boolean = x match {
    case Branch(11, r1) => 1 === 11 && r === r1
   case _ => false
```

Who is this?

Haskell and Scala

Adam Szlachta

History

Functional

Rasic syntax

Functions

c.....

Dynicax Sammary

.. . . .

types

-51---

Monadic features

Summany



Functional

programming

_ ...

Syntax summani

Algebraic data

types

Monadic features

Summary

```
[x \mid i \leftarrow [0..10], let x = i*i, x > 20]
```

```
genSquares :: [Int
genSquares = do
    i <- [0..10]
    let x = i*i
    guard (x > 20)
    return x
```

Works in any monadic context.

```
for { i <- List.range(0, 11); x = i*i; if x > 20 } yield x
```

```
def genSquares(): List[Int] = for
  i <- List.range(0, 11)
  x = i*i
  if x > 20
} yield x
```

Works for any type implementing map/flatMap/filter

Haskell

icala

List comprehensions

Haskell and Scala

Adam Szlachta

......

History

Functional

programmin

compariso

Functions

Syntax summary

. . .

Algebraic data

Lypes

Monadic features

Summan

```
[x | i <- [0..10], let x = i*i, x > 20]
genSquares :: [Int]
genSquares = do
    i <- [0..10]
    let x = i*i
    guard (x > 20)
    return x
```

Works in any monadic context.

```
for { i <- List.range(0, 11); x = i*i; if x > 20 } yield x

def genSquares(): List[Int] = for {
   i <- List.range(0, 11)
   x = i*i
   if x > 20
} yield x
```

Works for any type implementing map/flatMap/filter

List comprehensions

Haskell and Scala

Adam Szlachta

Introductio

History

Functional

programming

companiso

Functions

Syntax summary

Algebraic data

Lypes

Monadic features

Summan

```
[x | i <- [0..10], let x = i*i, x > 20]
genSquares :: [Int]
genSquares = do
    i <- [0..10]
    let x = i*i</pre>
```

Works in any monadic context.

guard (x > 20)
return x

```
for { i <- List.range(0, 11); x = i*i; if x > 20 } yield x

def genSquares(): List[Int] = for {
   i <- List.range(0, 11)
   x = i*i
   if x > 20
} yield x
```

Works for any type implementing map/flatMap/filter

```
4□ > 4□ > 4□ > 4□ > 4□ > 9
```

List comprehensions

Haskell and Scala

Adam Szlachta

Introductio

....

Functional

programming

Companisor

Sylicax Sullillia

Algebraic date

types

Monadic features

ivioliadic leatures

Summary

```
[x | i <- [0..10], let x = i*i, x > 20]
genSquares :: [Int]
genSquares = do
    i <- [0..10]
    let x = i*i
    guard (x > 20)
    return x
```

Works in any monadic context.

```
for { i <- List.range(0, 11); x = i*i; if x > 20 } yield x

def genSquares(): List[Int] = for {
   i <- List.range(0, 11)
   x = i*i
   if x > 20
} yield x
```

Works for any type implementing map/flatMap/filter.

Histor

Functional programming

Basic syntax

c.....

Alaskasia dae

types

Classes

Monadic features

for {

} yield z

x <- Some (8)

 $v \leftarrow fun1(x)$

 $z \leftarrow fun2(y)$

```
do
    x <- Just 8
    y <- fun1 x
    z <- fun2 y
    return z

do

x <- Just 8
    y >>= \x ->
    fun2 y >>= return

do

x <- Just 8
    y <- fun1 x
    fun2 y >>= \x ->
    fun2 y
```

```
Some(8).flatMap (x =>
fun1(x).flatMap (y =>
fun2(y).map (z =>
z)))
```

Monadic notation

Haskell and Scala

Adam Szlachta

Histor

Functional

programming

Companisor

A1 1 1 1 1 1 1

types

Ciasses

Monadic features

for {

} yield z

x <- Some (8)

 $v \leftarrow fun1(x)$

 $z \leftarrow fun2(y)$

ummary

```
do
    x <- Just 8
    y <- fun1 x
    z <- fun2 y
    return z

do
    x <- Just 8
    y <- fun1 x
    Just 8 >>= \x ->
    fun1 x >>= \y ->
    fun2 y >>= return
    return z
```

```
Some(8).flatMap (x => fun1(x).flatMap (y => fun2(y).map (z => z)))
```

skell

I/O isolation

object Console {

```
Haskell and Scala
```

Adam Szlachta

Histor

Functiona

Basic syntax

Suntay summan

Algebraic data

types

Classes

Monadic features

Summary

```
getLine :: IO String
getLine = ...
putStr :: String -> IO ()
putStr = ...

getLineWithPrompt :: String -> IO String
getLineWithPrompt prompt = do
    putStr prompt
    getLine
line :: IO String
line = getLineWithPrompt "> "
```

```
def readLine(): String = { ... }
  def print(obj: Any) { ... }
}

def getLineWithPrompt(prompt: String): String = {
  Console.print(prompt)
   Console.readLine()
}

val line: String = getLineWithPrompt("> ")
```

I/O isolation

```
Haskell and Scala
```

Adam Szlachta

Histor

Function

Basic syntax

Functions

Syntax summary

Algebraic data

types

Monadic features

Summary

```
getLine :: IO String
getLine = ...
putStr :: String -> IO ()
putStr = ...

getLineWithPrompt :: String -> IO String
getLineWithPrompt prompt = do
    putStr prompt
    getLine
line :: IO String
line = getLineWithPrompt "> "
```

```
object Console {
  def readLine(): String = { ... }
  def print(obj: Any) { ... }
}

def getLineWithPrompt(prompt: String): String = {
  Console.print(prompt)
   Console.readLine()
}

val line: String = getLineWithPrompt("> ")
```

I/O isolation

```
Haskell and Scala
            getLine :: IO String
Adam Szlachta
            getLine = ...
            putStr :: String -> IO ()
            putStr = ...
            getLineWithPrompt :: String -> IO String
            getLineWithPrompt prompt = do
                 putStr prompt
                 getLine
            line :: IO String
            line = getLineWithPrompt "> "
            object Console {
              def readLine(): String = { ... }
              def print(obj: Any) { ... }
Monadic features
            def getLineWithPrompt(prompt: String): String = {
              Console.print(prompt)
              Console.readLine()
            }
            val line: String = getLineWithPrompt("> ")
```

Features comparison

Haskell and Scala

Adam Szlachta

History

programming
Basic syntax

Functions

Syntax summ

Laziness

Laziness

types Classes

Monadic feature

	Haskell	Scala	Java
strong static typing	yes	yes	yes
type inference	yes	yes	no
higher order types	yes	yes	yes**
algebraic data types	yes	yes (verbose)	no
infinite streams	yes	yes	no*
strict semantics	optional	default	yes
lazy evaluation	default	optional	no
currying	default	optional	no
lambda expressions	yes	yes	no*
immutability	enforced	not enforced	not enforced
side effects isolation	yes	no	no
default implementations	yes	yes	no*

^{*} will be in Java 8

^{**} not as good as in Haskell/Scala

Who is this?

Haskell and Scala

Adam Szlachta

Additi Sziacitte

....

Functional

Basic syntax

Functions

c.....

Algobraic date

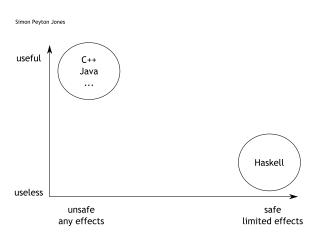
types

Monadic leature



Haskell and Scala

Adam Szlachta



Haskell and Scala

Adam Szlachta

Additi Sziacitta

History

Functional

programming

Eupetions

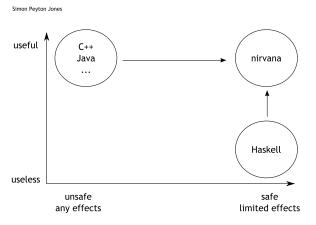
Syntax Summary

Algebraic data

types

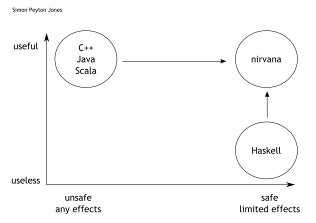
Monadic feature

ivionadic teature



Haskell and Scala

Adam Szlachta



Haskell and Scala

Adam Szlachta

History

Functional

programming

F.....

c.....

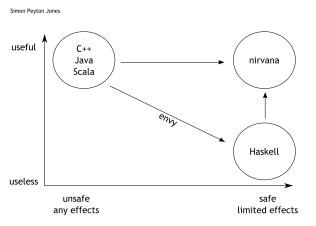
Syntax summary

Algebraic data

types

Monadic feature

mondate reactive



Libraries and tools inspired by Haskell

Haskell and Scala

Adam Szlachta

Functional

programmin

- ...

-,..---

Algebraic date

types

Monadic feature

Summary

Type classes library Scalaz (Haskell standard library)

Combinator parser
 (Haskell: Parsec, attoparsec, polypar

 Automated specification-based testing ScalaCheck (Haskell: QuickCheck)

Libraries and tools inspired by Haskell

Haskell and Scala

Adam Szlachta

......

Functional programmin

programmin

Eunstions

Syntax summan

Algebraic data

types

Monadic feature

Summary

 Type classes library Scalaz (Haskell standard library)

Combinator parser
 (Haskell: Parsec, attoparsec, polyparse)

 Automated specification-based testing ScalaCheck (Haskell: QuickCheck)

Libraries and tools inspired by Haskell

Haskell and Scala

Adam Szlachta

Histor

programmi

Basic syntax

Functions

Syntax summar

Laziness

Algebraic data

C1

Monadic feature

Summary

 Type classes library Scalaz (Haskell standard library)

 Combinator parser (Haskell: Parsec, attoparsec, polyparse)

 Automated specification-based testing ScalaCheck (Haskell: QuickCheck)

Resources

Haskell and Scala

Adam Szlachta

ntroduction

History

Functional programming

Basic syntax

Functions

Syntax summary

Laziness

types

Monadic featu

ivionadic feature

Summary

Links

- http://808Fabrik.com/scala
- http://hyperpolyglot.org/ml
- http://downgra.de
- http://hseeberger.wordpress.com
- http://code.google.com/p/scalaz/
- http://code.google.com/p/scalacheck/
- http://www.artima.com/pins1ed/combinator-parsing.html
- http://www.haskell.org/haskellwiki/Typeclassopedia
- http://typeclassopedia.bitbucket.org

Books and papers

- Eugenio Moggi, "Notions of computation and monads"
- Philip Wadler, "Comprehending Monads"
- Philip Wadler, "Monads for functional programming"
- Conor McBride, Ross Paterson, "Applicative programming with effects"
- Ross Paterson, "Arrows and computation"
- Jeff Newbern, "All About Monads"
- Brent Yorgey, "The Typeclassopedia" in "The Monad.Reader Issue 13"