

# Parallele und funktionale Programmierung

Felix Leidl

16. Februar 2024

# Grundlagen

## Terminologie

- Seiteneffekt
- Werttreue: gleiche Eingabe ergibt gleiche Ausgabe
- REPL: Read-Eval-Print-Loop
- Objektorientiert: In Scala ist alles ein Objekt
- statisch typisiert: Typen werden zur Übersetzung festgelegt
- implizite Typinferenz
- Funktionsabstraktion: Funktionsdefinition
- Currying
- Paradigmen: imperativ vs. deklarativ
- Funktion höherer Ordnung: Parameter oder Rückgabewert sind Funktion
- anonyme Funktion: kann einer Variable zugeordnet werden
- n-stellige Funktion: n Parameterlisten
- Lazy Evaluation: Berechnung erst bei Verwendung
- LazyList: Unendlichkeit möglich
- Overhead: Mehrkosten durch Parallelisierung

## Syntax

### List

```
List(4,5,6): List[Int]
List(42, 'a', false): List[Any] // mehrere Typen in Liste => Supertyp Any
List.range(4,7) // 7 exklusiv
4 to 6
4::5::6::Nil // :: adds or removes the first element
List(4,5)::List(6) // adds two Lists together
List().length
List().head // first element
List().tail // List, without head
List().drop(3) // drops the first n elements
List().sum // adds all elements
List().product
List(4,5,6)(0) // returns first element
```

```

List(4,5,6).take(2) // returns first n elements
List().reverse
List(1,2,3).zip(List(4,5,6,7)) // => List((1,4),(2,5),(3,6)) drops the tail
ls.foldRight(Nil)((e, r) => if (r == Nil || e != r.head) e :: r else r)
List(1,2,3).filter(_ == 3) // returns List without 3
List(List(2), List(4,3)).flatten // returns List(2,3,4)
List(1,2,3).equals(List(1,2,3))
List().map(_ + 1) // increases every Element by one
"Scala".toList
List(1,2,3,4,5).takeWhile(_ < 3) // returns List(1,2)
List().dropWhile(_ < 5)
List('S', 'c', 'a').mkString
List(2,4,6).forall(_ % 2 == 0) // returns true, when all are true
List(2,3,4).exists(_ == 3) // returns true, when on is true

```

## LazyList

```

1#::LazyList()
lazy val dikrim = b*b - 4 * a * c // calculation at first use
LazyList.iterate(BigInt(2))(_ + 1) // LazyList from 2 to infinity

```

## Tupel

```

(false, 10): (Boolean, Int)
(false, 10, 'a'): (Boolean, Int, Char)
(false, 10)._1 // gives you the first element

```

## Funktionen

```

def foo: Boolean => Char = a => if a 'a' else 'b'
def pair[A]: A => (A,A) = x => (x, x) // Generische Typparameter
def addd: Int => Int => Int => Int = x => y => z => x + y + z // Currying
def bar: List[Int] => Int = { // Mustervergleich
  case a::as => 1 + bar(as)
  case a => 0
}
def lol: Unit => Int = _ => 42 // Unit is nothing and _ ignores the constant
val f = (x: Int) => x + 1 // anonymus function
val f = _ + 1 // anonymus function

```

## If-Else

```

if n>2 then n else -n
if (n>2) n else -n
if n>2 then n
else if n==2 then 2

```

```

else -n
case x if x > 2 // guard

```

## For-comprehension

```

for x <- List.range(1,10) yield x*x
for x <- List.range(1,10) if x > 1 yield x*x
for x <- List.range(1,5)
  y <- List.range(x, 4) yield (x,y)
for xs <- xss; x <- xs yield x // flatten an list

```

## Type Aliasing

```

type Coordinate = (Int, Int)
type Move = Coordinate => Coordinate // function typ
def left: Move = {case (x, y) => (x-1, y)}
type Pair[T] = (T,T)
def mult: Pair[Int] => Int = (x, y) => x * y
def copy[T]: Pair[T] => T = x => (x, x) // generic typing

```

## Typverbund

```

trait Shape // interface
case class Circle(r: Double) extends Shape // constructor is r: Double
case class Rect(l: Double, w: Double) extends Shape
case object Dot extends Shape // Singleton
def square: Double => Shape =
  n => Rect(n, n)
def area: Shape => Double = {
  case Circle(r) => math.Pi * r * r
  case Rect(l, w) => w * l
}

```

## Enum

```

enum Answer:
  case Yes, No, Unknown // allows Yes instead of Answer.Yes
import Answer._
def flip: Answer => Answer = {
  case Yes => No
  case No => Yes
  case Unknown => Unknown
}

```

## MapReduce

```

val Letters: List[Char] = List.range('a', ('z' + 1).toChar)
def mapLetterCount: String => List[(Char, Int)] =

```

```

    string => count(letters, string.toList)
def reduceLetterCount: ((Char, List[Int])) => (Char, Int) = {
    case (key, values) => (key, values.sum)
}
mapReduce(mapLetterCount)(reduceLetterCount)(bsp1) // maps first, then reduces

```

## Parallelisierung

### Vorteile

Alle parallelisierbaren Operationen arbeiten wegen „divide and conquer“ automatisch parallel

Programmierende muss sich nicht um Lastverteilung/Synchronisierung kümmern

Wird im Hintergrund mit einem schlafenden fork/join-Pool umgesetzt

### .par

- Liefert Sicht auf Ursprüngliche Daten
- Die Verarbeitungsoperationen der Sicht sind jedoch parallelisiert
- Benötigt `import scala.collections.parallel.CollectionConverters._`
- Beispiele: `List(1,2,3).par.map(_ + 1)`

### Future

- Einpacken einer Berechnung `f` in ein Futur: `val c = Future {f}`
- Warten auf das Ergebnis: `Await.result(c, Duration.Inf)`

### Akka-Aktoren

- externe Bibliothek

## Datentypen

- Boolean
- Int
- String
- Char
- BigInt

- BigDecimal
- Long
- Any: Übertyp aller Typen
- Nothing: Untertyp aller Typen

## ParVector

Ist die parallele Version der Liste

```
342 += 42 += ParVector.fill(10)(100)
```

## ParRange

```
ParRange(30, 47, 1, true) // == (30 to 47)
```

## Future

```
val a = Future(x*x)
Await.result(a, Duration.Inf)
```