Einführung in Scala

Einführung

Funktionale Programmierung

- Funktionale Programmierung basiert auf Mathematik. Sie ist deklarativ (nicht imperativ). Innerhalb des Paradigmas ist der Ablauf eines Programms nicht definiert.
- Funktionale Programme basieren auf Ausdrücken, die ausgewertet (evaluiert) werden

Kurz und Knapp

- In eingeschränkter Form ist funktionale Programmierung die Programmierung ohne veränderliche Variablen, ohne Zuweisung und ohne Kontrollstrukturen.
- In allgemeinerer Form ist es Programmierung mit besonderer Betonung von Funktionen.
- Funktionen sind Werte. Man kann durch Operationen neue Funktionen erzeugen. Man kann Funktionen an Funktionen übergeben.

Einführung

Sprachen

- Java (Version <= Java 7) ermöglicht die Implementierung von Funktionen durch Funktionsobjekte. Diese werden in der einfachsten Form durch anonyme Klassen definiert. Der Umgang mit Funktionsobjekten ist extrem schwerfällig.
- Eine Sprache, welche die funktionale Programmierung unterstützt, ermöglicht erheblich einfacheren Umgang mit Funktionen.
- Scala ist objekt-funktional: Das Verhalten von Objekten wird durch funktionale Methoden bestimmt.

Einführung

Eigenschaften funktionaler Sprachen

- Keine Variablen, keine Seiteneffekte, keine Zuweisung, keine imperativen Kontrollstrukturen
- Ausdrücke lassen sich deklarativ verstehen!
- Funktionen sind Bürger erster Klasse (first class citizens):
  - Wie alle Daten können Funktionen innerhalb von Funktionen definiert werden
  - Wie alle Daten können Funktionen an Funktionen übergeben und zurückgegeben werden
  - Funktionen lassen sich mit anderen Funktionen zu neuen Funktionen verknüpfen
  - Insbesondere gibt es Funktionen, die andere Funktionen auf Datenstrukturen anwenden (Funktionen höherer Ordnung)

Einführung

Geschichte der funktionalen Programmierung

- Alonso Church definierte 1936 umfassendes funktionales System (Lambda-Kalkül).
- Marvin Minsky entwickelte 1960 am MIT die Programmiersprache LISP (inspiriert vom Lambda-Kalkül). Ziel: mächtige Programmiersprache für Forschung in der KI. LISP war vom Komfort der Zeit weit voraus (interaktives Arbeiten, dynamische Datenstrukture mit automatischer Speicherbereinigung).
- Funktionale Programmierung war wesentlicher Ausgangspunkt von objektorientierter Programmierung (Flavors), von höheren Mechanismen der Nebenläufigkeit (Actors) und auch Programmiersprache der ersten graphisch interaktiven Benutzeroberflächen (Lisp-Machine, Loops).
- Heute spielt funktionale Programmierung eine zunehmende Rolle bei einigen Mustern der objektorientierten Programmierung und Programmiersprache für verteilte Systeme (Erlang, Haskell, Scala).

Erste Beispiele (LISP)

CommonLISP

- `https://onecompiler.com/commonlisp/`

Ein Programm besteht in der Auswertung eines funktionalen Ausdrucks

(+ (* 4 5) (/ 20 4)) $\rightarrow$ (+ 20 5) $\rightarrow$ 25

In CommonLISP

```
(write (+ (* 4 5) (/ 20 4)))
```

Erste Beispiele (LISP)

Funktionen werden durch Lambda-Ausdrücke definiert und können „angewendet“ werden

((lambda (x) (* 3 x)) 10) $\rightarrow$ 30

In CommonLISP

```
(write((lambda (x) (* 3 x)) 10))
```

Erste Beispiele (LISP)

Funktionen können (für spätere Anwendungen) in Variablen gespeichert und an andere Funktionen übergeben werden.

```
(defun mal3 (x) (* 3 x))
(write (mal3 5))
(write (mal3 10))

(defun mal9 (x) (*(mal3 x) 3))
(write (mal9 9))
```

Hinweis

Die LISP-Syntax ist sehr einfach und sehr flexibel – aber für Java/C-Programmierer ziemlich ungewohnt. Wir nutzen Scala, auch um zu zeigen, dass funktionale Programmierung nicht exotisch ist!

Erste Beispiele (LISP)

Fakulät rekursiv berechnen

```
(defun fact (n)
  (if (zerop n)
    1
    (* n (fact (1- n)))))
(write(fact 6))
```

Seit Java 8

Wiederholung

```
public class FunktionalTest {
  public static void main(String[] args) {
    Integer[] a = { 8, 3, 1, 4, 0, 7 };
    Arrays.sort(a, (x, y) -> y - x);
    System.out.println(Arrays.toString(a));
    String[] b = { "x", "a", "b" };
    Arrays.sort(b, (x, y) -> y.compareTo(x));
    System.out.println(Arrays.toString(b));
    List<Integer> lst = Arrays.asList(8, 3, 1, 4, 0, 7);
    Integer[] mp = lst.stream().parallel().filter(x -> x % 2 == 0).map(x -> x * x).toArray
        (Integer[]::new);
    System.out.println(Arrays.toString(mp));
  }
}
```

Scala

Objekt-funktionale Programmierung mit Scala

- Entwickelt ab 2001 am EPFL (Lausanne) von Martin Odersky.
- Scala enthält auch eine ganze Menge von möglichen Verbesserung
  - Keine statischen Klassenelemente
  - Alles ist ein Objekt (auch Zahlen und Funktionen)
  - Scala fördert die Verwendung von unveraänderlichen Variablen
  - Typangaben können meist unterbleiben (Typinferenz)
  - Es gibt eine Reihe von syntaktischen Verbesserungen gegenüber Java
  - High-Level Pattern-Matching Ausdruck
  - Closures (= Funktionsobjekte mit Kenntnis der Umgebung)
  - Die Scala-Bibliothek unterstützt die funktionale Programmierung mit Objekten

Scala

Objekt-funktionale Programmierung mit Scala

- Aber auch:
  - Scala erlaubt die prozedurale Programmierung.
  - Scala ermöglicht die Verwendung aller Java Klassen.

In diesem Modul

- Werkzeug: `https://www.jetbrains.com/help/idea/discover-intellij-idea-for-scala.html`
- Scala Version 3.*x*

Aufbau eines Scala-Programms

- Wie ein Java-Programm, so gliedert sich ein Scala-Programm in Pakete.
- Diese wiederum enthalten Klassen, abstrakte Klassen, Objekte und Traits (diese übernehmen die Rolle von Interfaces).
- In Scala entfällt die Forderung, dass eine Datei den gleichen Namen wie die (einzige) öffentliche Klasse tragen muss.
- Pakete werden in Scala wie in Java durch die Packetanweisung deklariert.
- Grundsätzlich können alle Java-Klassen benutzt werden. Das gleiche gilt grundsätzlich auch umgekehrt.

HalloWelt

```
object HalloWelt {
  def main(args: Array[String]) {
    println("Hallo, Welt!")
  }
}
```

Was fällt auf im Vergleich zu Java?

- `object HalloWelt` → Singleton Objekt: Klasse mit nur einer Instanz.

⇒ Im Beispiel werden mit dem Schlüsselwort `object` sowohl eine Klasse namens `HalloWelt` als auch die dazugehörige, gleichnamige Instanz definiert. Instanz wird bei erstmaliger Verwendgung erstellt

- `main`-Methode nicht als `static` deklariert

⇒ statische Mitglieder (Attribute oder Methoden) existieren in Scala nicht

Interaktion mit Java

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object GermanDate {
  def main(args: Array[String]): Unit = {
    val now = new Date
    val df = getDateInstance(LONG, Locale.GERMAN)
    println(df format now)
  }
}
```

Variablendeklarationen und Typparameter

```
val a = 1.5 // Java: int[] d = new int[5]
val b = "abc" // Java: final double a = 1.5;
val c = new Array[Int](5) // Java: int c = 0
val d = Array(1, 2, 3) // Java: int[] a = {1, 2, 3};
var e = 0 // Java: final String b = "abc";
val f = 1.toString  // Java: Zahlen sind Objekte
```

`val` = unveränderliche Variable, `var` = veränderliche Variable...

Lazy

Man kann festlegen, dass ein Wert nicht zu dem Zeitpunkt seiner Definition, sondern erst später bei der ersten Verwendung ausgewertet wird

Ausgabe?

```
object Beispiel4 {
  def main(args: Array[String]): Unit = {
    var x = 4
    lazy val y = 10 * x
    x = 5
    println(y)
  }
}
```

Top-Level Elemente

- `trait`
  Entspricht einem Java-Interface + Variablen + Methoden + Mehrfachvererbung (benötigen wir nicht)
- `abstract class`
  wie Java
- `class`
  wie Java
- `case class`
  Klasse für unveränderliche Objekte mit: toString, equals, hashCode, ...
- `object`
  singuläres Objekt mit globalem Namen und anonymer Klasse
- `case object`
  unveränderliches Objekt mit: toString, equals, hashCode

Scala-Singleton-Objekte

- In Scala gibt es keine statischen Funktionen (diese sind nicht objektorientiert)
- Für global anzusprechende Funktionen gibt es singuläre Objekte

```
object Beispiel5 {
  def main(args: Array[String]): Unit =
      {
    printHello()
  }
  def printHello(): Unit = {
    println("hello world")
  }
}
```

```
object Beispiel6 {
  def main(args: Array[String]) {
    Printer.printHello()
  }
}
object Printer {
  def printHello() {
    println("hello world")
  }
}
```

`Printer.printHello()` mit `Printer` meint ein Objekt, keine Klasse!

Scala-Klassen und Konstruktoren

Jede Klasse hat einen sogenannten primären Konstruktor. Dessen Parameter stehen unmittelbar im Kopf der Klasse, sein Körper steht als Anweisungsfolge im Klassenkörper

Java

```
public class Person {
  private final String name;
  private final int alter;
  public Person(String n, int a) {
    if (a < 0) throw new IllegalArgumentException();
    name = n;
    alter = a;
  }
  public String name() {
    return name;
  }
  public int alter() {
    return alter;
}}
```

Scala

```
class Person(n: String, a: Int) {
  require(a >= 0)
  val name = n
  val alter = a
}

// noch kuerzer
class Person(val name: String, val alter: Int) {
  require(alter >= 0)
}
```

Case-Klassen

Klassen, die im Wesentlichen nur Information transportieren, werden als `case class` definiert sind. Bei diesen werden einige Methoden, wie `toString` und `equals` automatisch definiert.

```
object Hauptprogramm {
  def main(args: Array[String]): Unit = {
    val pers = Array(Person("Karin", 17), Person("Hans", 9), Person("Karin", 17))
    println(pers(0))
    println(pers(0) == pers(2))
    val gefunden = pers.exists(_.name == "Karin")
  }
}

case class Person(name: String, alter: Int) {
  require(alter >= 0)
}
```

Methodendeklaration

- Die Methodendeklaration wird durch `def` eingeleitet
- Darauf folgt der Name der Methode und dann die optionale, in Klammern eingeschlossene Parameterliste
- Anschließend folgt der Rückgabetyp gefolgt von dem Methodenkörper (Anstelle des Schlüsselworts `void` fungiert in Scala der Typ `Unit`)

```
def intMethode(x: Int): Int = 3 * x

def fakultaet(n: Int): Int =
  if (n == 0) 1 else n * fakultaet(n -
      1)

def voidMethode(x: String): Unit =
  println(x)
```

```
def langeIntMethode(n: Int): Int = {
  var s = 0
  for (i <- 1 to n) s += i
  s
}
def langeVoidMethode(): Unit = {
  print("hello ")
  println("world")
}
```

Funktionsobjekte

```
object A {
  def m(): Int = {
    3
  }
}
object X {
  def main(args: Array[String]): Unit = {
    val hello_fkt1 = A.m _
    val hello_fkt2: ()=>Int = A.m
    val drei = A.m()
    println(hello_fkt1.apply())
    println(hello_fkt2.apply())
    println(drei)
  }
}
```

anonyme Funktion – Lambda-Ausdruck

Beispiel

```
val addiereXundY = (x: Int, y: Int) => x + y
```

Syntax

```
anonyme Funktion ::= (Parameterliste) => Ausdruck
                   | Variable => Ausdruck
                   | Ausdruck mit anonymen Variablen
```

anonyme Funktion – Lambda-Ausdruck

Beispiel

```
object Test2 {
  def main(args: Array[String]): Unit = {
    val a = Array(1, 2, 3, 4, 5)
    val summe = a.reduce((x: Int, y: Int) => x + y)
    val summeKuerzer = a.reduce((x, y) => x + y)
    val summeNochKuerzer = a.reduce(_ + _)
    val summeGanzKurz = a.sum //vordefiniert
    println(summeGanzKurz)
  }
}
```

Shorts: Imperative Programmierung versus deklarative Programmierung

Imperative Programmierung versus deklarative Programmierung

- Imperative Programmierung sagt, was der Rechner tun soll
- Deklarative Programmierung beschreibt Sachverhalte
- Imperative Programme nur verständlich, wenn man den Ablauf nachvollzieht!!
- Deklarative Programme kennen keine Seiteneffekte.
- Imperative Programme erfordern separaten Beweis (Invarianten etc.)
- Geschichte der Programmiersprachen ist der Versuch die Nachteile der imperativen Programmierung zu beheben (modulare Programmierung, objektorientierte Programmierung ...)
- Imperative Programmierung ist rechnernah $\Rightarrow$ effizient (auch bei schlechtem Compiler).
- Funktionale Programme müssen durch den Compiler in einen effizienten Ablauf umgesetzt werden.

Imperative Ablaufverfolgung ist nicht einfach

```
1  static int fak(int n) {
2    int f = 1;
3    int i = 0;
4    while (i != n) {
5      i += 1;
6      f *= i;
7    }
8    return f;
9  }
```

```
fak(3)
Z n i f
-------
3 3 - 1
4 3 0 1
6 3 1 1
5 3 1 1
6 3 2 1
5 3 2 2
6 3 1 2
5 3 1 6
8 3 1 6

// Z = Zeile
```

- Um Abläufe besser zu verstehen, braucht man Invarianten. Hier: $i! = f$
- Iteratoration braucht bei N Wiederholungen $O(1)$ Speicherplatz

Ablaufverfolgung von Rekursion ist besonders schwierig

```
1  static int fak1(int n) {
2    int r;
3    if (n == 0)
4      r = 1;
5    else
6      r = n * fak(n - 1);
7    return r;
8  }
```

```
fak(3)
1.     2.     3.     4.     rek. Aufruf
n r    n r    n r    n r lok. Variable
3 -
       2 -
              1 -
                     0 -
                     1 1
              1 1
       2 2
3 6
```

- Das Verständnis imperativer Programme ist schwierig. Rekursion ist imperativ schwer verständlich. Imperative Programme sind fehleranfällig !!!
- Rekursion braucht bei N-Wiederholungen $O(n)$ Speicherplatz

Funktionale Programmierung: kein Ablauf von Anweisungen, sondern Auswertung eines Ausdrucks

```
def fakultaet(n: Int): Int =
  if (n == 0) 1 else n * fakultaet(n - 1)
```

```
factorial(3)
= 3 * factorial(3 - 1)
= 3 * factorial(2)
= 3 * (2 * factorial(2 - 1))
= 3 * (2 * factorial(1))
= 3 * (2 * (1 * factorial(0)))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3* 2 = 6
```

- In funktionalen Sprachen kann man die Rekursion immer als Gleichungsumformung aufschreiben!
- Die Auswertungsreihenfolge ist nicht zwingend festgelegt.

Endrekursion als Auswertung

```
def factorial (n: Int) = {
  def fac(n: Int, f: Int): Int =
    if (n==0) f else fac(b-1, n*f)
  fac(n,1)
}
```

```
factorial(3)
= fac(3, 1)
= fac(3 - 1, 3 * 1)
= fac(2, 3)
= fac(2 - 1, 2 * 3)
= fac(1, 6)
= fac(1 - 1, 1 * 6)
= fac(0, 6)
= 6
```

Definition

Eine rekursive Funktion ist `endrekursiv`, wenn nach dem rekursiven Aufruf innerhalb der Funktion keine Operation mehr auszuführen ist. Die Endrekursionsoptimierung bewirkt die Übersetzung einer endrekursiven Funktion in einen iterativen Ablauf/Prozess.

Vergleich Endrekursion - Normalrekursion

```
factorial(3)
= 3 * factorial(3 - 1)
= 3 * factorial(2)
= 3 * (2 * factorial(2 - 1))
= 3 * (2 * factorial(1))
= 3 * (2 * (1 * factorial(0)))
= 3 * (2 * (1 * 1))
= 3 * (2 * 1)
= 3 * 2 = 6
```

```
factorial(3)
= fac(3, 1)
= fac(3 - 1, 3 * 1)
= fac(2, 3)
= fac(2 - 1, 2 * 3)
= fac(1, 6)
= fac(1 - 1, 1 * 6)
= fac(0, 6)
= 6
```

- Beide Algorithmen sind rekursiv fomuliert
- Ein Algorithmus wird durch einen Prozess ausgeführt. Links rekursiv, recht iterativ

iterativer Prozess vs. rekursiver Prozess

Definition

- ein iterativer Prozess kann durch einen festen Satz von Zustandsvariablen beschrieben werden.
  (Speicherkomplexität = $O(1)$)
- ein rekursiver Prozess ist durch eine Menge von aufgeschobenen Operationen charakterisiert.
  (Speicherkomplexität = $O(n)$)

Beachte den feinen Unterschied zu rekursiver und iterativer Funktionsdefinition!

Übersetzung in imperativen Java-Bytecode (scalac)

```
private final int fac(int n,
    int f)
0 iload_1 [n]
1 iconst_0
2 if_icmpne 7 // if (n == 0)
5 iload_2 [f]
6ireturn
7 iload_1 [n] // n-1
8 iconst_1
9 isub
10 iload_1 [n] // n*f
11 iload_2 [f]
12 imul
13 istore_2 [f] // f = n * f
14 istore_1 [n] // n = n - 1
15 goto 0 // Wiederholung
```

- Ausführbarer Code ist (fast) immer imperativ. Es ist Aufgabe des Compilers diesen Code zu erzeugen!
- Es gab und gibt immer Ansätze Computer mit funktionalem Befehlssatz zu entwerfen (LISP-Machine, etc.). So überzeugend dies aus theoretischer Sicht ist, scheiterte die allgemeine Verwendung bisher an einem praktischen Grund: Chipentwicklung lohnt sich nur bei sehr großen Stückzahlen! Traditionelle Architekturen sind daher fast immer überlegen.

Funktionale Programmierung in Scala

Motivation: Funktionale Programmierung in Scala

Schnapszahlen

```
import java.util.Scanner
import java.io.FileReader
object Schnapszahlen {
  def main(args: Array[String]): Unit = {
    val in = new Scanner(new FileReader("zahlen.txt"))
    val a = Array.tabulate(16, 16)((i, j) => in.nextInt)
    println("Programm zur Ueberpruefung auf Schnapszahlen")
    println("\nWerte der Testmatrix:")
    for (zeile <- a) println(zeile.mkString(" "))
    val zz = a.flatten.filter(z => z % 11 == 0).toSet[Int]
    printf("%nSchnapszahlen: %s%n", zz.mkString(" "))
    printf("Es sind %d Schnapszahlen.%n", zz.size)
  }
}
```

Von Iteration zu Endrekursion

```
def fib(n: Int): BigInt = {
  var g = BigInt(0)
  var f = BigInt(1)
  var i = n // Parameter wie n duerfen nicht veraendert werden
  while (i != 0) {
    val t = f + g // Die prozedurale Programmierung erfordert Hilfsvariable
    g = f
    f = t
    i == 1
  }
  g
}
```

Scala ist vom funktionalen Paradigma beeinflusst (keine veränderlichen Parameter, kein i++ mit Seiteneffekt) – diese Regel ist auch bei der prozeduralen Programmierung gut!

Weg zur Endrekursion

- Für while-Schleifen lokale Funktion verwenden
- Lokale Variablen werden zu Parametern der lokalen Funktion.
- Bei Abbruch der lokalen Funktion steht das Ergebnis fest.
- Parameter (lokale Variablen) werden beim Aufruf durch äußere Funktion initialisiert.
- Da Zuweisung zu Parametern (beim Aufruf) gleichzeitig erfolgt, entfallen Hilfsvariablen.
- Die äußere Funktion kann Vorbedingungen prüfen.

(End-)rekursive Lösungen

```
// rekursive Fassung (funktional)
def fibRec(n: Int): BigInt =
  if (n <= 2) 1 else fib_rec(n - 1) + fibRec(n - 2)

// endrekursive Fassung (funktional)
def fib_rec(n: Int): BigInt = {
  require(n >= 0)
  @tailrec // der Compiler prueft, ob wir recht haben
  def fb(i: Int, f: BigInt, g: BigInt): BigInt =
    if (i == 0) g else fb(i-1, f + g, f)
  fb(n, 1, 0)}
```

In Kürze ...

Was ist der Unterschied zwischen

- Methode: an ein Objekt gebundene Funktion
- Closure: eine Funktion, die an eine Umgebung gebunden ist
- Lambda-Ausdruck / Funktionsliteral: anonym definierte Funktion

Achtung

Es gibt Methoden und Funktionen!

Methoden

- Methoden sind Funktionen, die von einem Objekt ausgeführt werden
- Das Objekt spielt die entscheidende Rolle:
  - Es entscheidet darüber, welche Methode ausgeführt wird (späte Bindung).
  - Die Methode kann auf Instanzvariablen des Objekts zugreifen und kann diese verändern!.
  - Eine Methode hat einen festen Namen

Funktionen / Closures

- Funktionen gehören zu keinem Objekt – können aber in einem Objekt gespeichert sein.
- Funktionen haben gebundene Variablen (in der Funktion definiert) und freie Variablen aus der Umgebung (äußere Funktion, Klasse, Objekt).
- Eine Closure ist eine Funktion, die innerhalb eines Programms weitergegeben werden kann und dabei ihre Umgebung von freien Variablen „mitnimmt“.
  (Begriff: die Referenzen zu freien Variablen werden in der Definitionsumgebung „lexikalisch“ aufgelöst = lexical closure.)

Funktionen / Closures

Fehler: häufig liest man, dass Closures „anonyme Funktionen“ sind

- mit Closure beschreibt man nur eine Eigenschaft von Funktionsliteralen
- im Unterschied zu Java, müssen Funktionen genauso wenig einen eigenen Namen haben wie Objekte
- man kann eine Closure in einer Variablen speichern (mit Namen).

Funktionsliteral / Lambda-Ausdruck / Anonyme Funktion

- Ein Funktionsliteral (Lambda-Ausdruck) stellt die Zuordnung von Argumenten zu Resultaten dar. (Ursprung: Lambda-Kalkül von Church) – es ist eine Funktionsdefinition
- Funktionsliterale können durch Variablen referiert werden.
- Die Syntax für Funktionsliterale lautet: Parameterliste $\Rightarrow$ Funktionsausdruck
- Man kann einer Methode ein äquivalentes Funktionsobjekt zuordnen.
  Syntax: `Objektreferenz.Methodenname _`
  Die Funktion behält über die Umgebung der freien Variablen den Zugriff auf das Objekt.
- Der Typ einer Funktion wird in Scala durch die Zuordnung von Parametertypen angegeben.
  Beispiel: `(Double, Double) => Boolean`
- Häufig werden Funktionen einfach als Closure bezeichnet.

Beispiel 1: Methode / Funktion

```
class A(x: Double) {
  // Methode xMalY
  def xMalY(y: Double): Double = x * y
}
```

```
object Test {
  def main(args: Array[String]): Unit =
      {
    // Objekt a: A
    val a = new A(2.5)
    // Funktion f: (Double) => Double)
    val f = a.xMalY _
    // Aufruf von Methode mit Objekt
    println(a.xMalY(4))
    // Die Funktion ist in sich
        abgeschlossen (closure)
    println(f(4))
  }
}
```

- `def` definiert eine Methode
- Funktionen kann man in `val/var` speichern

Beispiel 2: Methode von Object / Funktion

```
object Functions {
  // Methode add
  def add(x: Double, y: Double): Double
      = x + y
}
```

```
import Functions.add
object Test {
  def main(args: Array[String]): Unit =
      {
    val f = add _
    // Aufruf von Methode mit Objekt
    println(add(3, 4))
    // Der Funktionsaufruf sieht nicht
        anders aus.
    println(f(3, 4))
  }
}
```

- Innerhalb von `object ...` verschwindet der Unterschied zwischen Funktion und Methode beinahe:

$\Rightarrow$ Funktionen sind Objekte!!

$\Rightarrow$ Methoden sind Teil eines Objekts

Beispiel 3: lokaler Kontext

```
object Text {
  def main(args: Array[String]): Unit = {
    type DoubleFkt = Double => Double
    def newParabola(a: Double, b: Double, c: Double):
          DoubleFkt = {
      def parabola(x: Double) = c + x * (b + x * a)
      parabola _
    }
```

```
    def newParabola1(a: Double, b: Double, c: Double):
          DoubleFkt =
      (x: Double) => c + x * (b + x * a)

    val p121 = newParabola(1, 2, 1)
    println(p121(2)) // ergibt: 9
  }
}
```

- 1. Funktion: Definition lokale Funktion (freie Parameter a, b, c). Sie wird nicht angewendet, sondern zurückgegeben. Mit dem _ wird sie zu einer abgeschlossenen Funktion.
- 2. Funktion: Definition eines Lambda-Ausdruck und direkte Rückgabe.
- Auch wenn die Umgebungsfunktion beendet ist, wird ihr Kontext mitgenommen (closure).
- Der Scala Compiler erlaubt einfachere Schreibweise (keine Typangabe oder kein _)

Zusammenhang zwischen Closure und anonymer Klasse

In Java ...

```
interface Function1<T, R> {
  public R apply(T x);
}
public class Test {
  public static void main(String[] args) {
    Function1<Double, Double> a = quadratic(1, 0, 0);
    Function1<Double, Double> b = quadratic(1, 0, 10);
    System.out.println(a.apply(3.0));
    System.out.println(b.apply(3.0));
  }
  static Function1<Double, Double> quadratic(
    final double a, final double b, final double c) {
      return new Function1<Double, Double>() {
        public Double apply(Double x) {
          return (a * x + b) * x + c;
        }
      };
}}
```

Nicht strikte Funktionen

Strikt

Vereinfacht: Eine Funktion ist strikt, wenn sie alle Argumente auswertet. Auch Java ist manchmal nicht strikt: `a && b, a || b, if (bed) s1 else s2`

Call by Name

Scala: wenn dem Parametertyp ein $\rightarrow$ vorangestellt wird, wird der übergebene Ausdruck erst bei Bedarf ausgewertet. Wird er mehrfach referiert, erfolgt jedes mal eine erneute Auswertung.

```
def byName(debug: Boolean, p: =>Double) {
  if (debug) println(p * p)
  // p wird 2 mal berechnet, wenn debug==true (sonst 0 mal)
}
```

Späte Evaluierung

„lazy val“ berechnet den Ausdruck erst bei Bedarf, speichert aber das Ergebnis.

```
def byName(debug: Boolean, p: =>Double) {
  lazy val q = p
  if (debug) println(q * q)
  // p wird 1 mal berechnet, wenn debug==true (sonst 0 mal)
}
```

Beispiel für Kontrollabstraktion

Eine Funktion wird bei Fehler wiederholt.

```
object MeinBsp {
  def main(args: Array[String]): Unit = {
    val i = asLongAs("falsche Eingabe") {
      readInt
  }}
  def asLongAs[T](message: String)(expression: => T): T = {
    try {
      expression
    }
    catch {
      case _: Exception =>
      println(message)
      asLongAs(message)(expression)
}}}
```

- Der Rückgabetyp wird vom Compiler ermittelt
- Kontrollabstraktionen erlauben, die Sprache um höhere Konstrukte zu erweitern

Currying

Beispiele

```
 // normale Funktion mit 2 Parametern.
def summe(a: Int, b: Int) = a + b
// Anwendung
val sum_1_plus_3 = summe(1, 3)

// Currying = definiert ueber Funktionsobjekte
def summe(a: Int) = (b: Int) => a + b
// Anwendung
val sum_2_plus_4 = summe(2)(4)

// abgekuerzte Definition
def summe(a: Int)(b: Int) = a + b
// Anwendung
val sum_7_plus_3 = summe(7)(3)
```

Currying

Unter Currying versteht man die Beschreibung von Funktionen mit mehreren Parametern durch eine Folge von Funktionen, die jeweils eine neue Funktion ergeben. Der Name erinnert an Haskell Curry, der (gleichzeitig mit anderen) diese Technik entwickelt hat. Ursprünglich wurde Currying eingeführt, um in der mathematischen Theorie Funktionen mit mehreren Parameter durch Funktionen mit einem Parameter zu beschreiben. In Programmiersprachen dient Currying der Erweiterung der Notation, der Verbesserung der Typ-Herleitungund auch der Möglichkeit, nach und nach die Parameter zu binden.

In Scala bietet Currying den Vorteil, dass dadurch die Typinferenz unterstütztwird. Datentypen, die der Compiler bei den ersten Parameterlisten erkannt hat können in den später anzuwendenden Parameterlisten zur Typinferenz herangezogenwerden.

Beispiel: Klassen und Objekte definiert durch Funktionen

```
def main(args: Array[String]): Unit = {
  val b1 = newBalance(100)
  val b2 = newBalance(200)
  println("b1 " + b1(Symbol("deposit"))(50))
  println("b2 " + b2(Symbol("withdraw"))(50))
}
type Method = Int => Int
def newBalance(initial: Int): Symbol => Method = {
  var b = initial
  def withdraw(amount: Int): Int = {
    require(amount >= 0 && b >= amount);
    b -= amount;
    b
  }
  def deposit(amount: Int): Int = {
    require(amount >= 0);
    b += amount;
    b
  }
  def dispatch(method: Symbol): Method = method match
      {
    case Symbol("withdraw") => withdraw
    case Symbol("deposit") => deposit
  }
  dispatch
}
```

- Eine Klasse ist eine Konstruktor-Funktion.
- Ein Objekt ist eine Closure mit freien Variablen und Methodenauswahl (dispatch)
- Interpretation: object.methode = eine Funktion, die anschließend aufgerufen wird

Partielle Funktion = partiell definierte Funktionen

Eine partiell definierte Funktion ist eine nur in Teilen des Definitionsbereichs definierte Funktion. In Scala istes möglich, mittels `isDefinedAt` nachzufragen, ob die Definition für ein bestimmtes Element gilt. In Scala wird eine partiell definierte Funktion durch eine `case`-Folge beschrieben.

Partielle Funktion = partiell definierte Funktionen

```
object Compute {
  def fac: PartialFunction[Int, Int] = {
    case 0 => 1
    case n if n > 0 => n * fac(n - 1)
  }

  def main(args: Array[String]): Unit = {
    print("Eingabe einer ganzen Zahl: ")
    val zahl = readInt
    berechne(fac, zahl)
  }

  def berechne[T](f: PartialFunction[T, T], n: T) =
  if (f isDefinedAt n)
  println("Der Funktionswert ist " + f(n))
  else
  println("Die Funktion ist nicht definiert")
}
```

Funktion höherer Ordnung

Definition

Eine Funktion höherer Ordnung ist eine Funktion, die ihrerseits Funktionen als Parameter oder als Ergebnis hat. Funktionen höherer Ordnung dienen oft dazu komplexe Operationen auf Datenstrukturen durchzuführen. Funktionen höherer Ordnung bieten auch die Grundlage für die Formulierung von Kontrollabstraktionen.

Java kennt Funktionen höherer Ordnung

- Umsetzung durch anonyme Klasse (diese verpackt im Beispiel die Funktion `compareTo`)
- Achtung: Anwendung ist aber nicht funktional, da die Inhalte des Array verändert werden
- Häufig bei GUI-Elementen (z.B. ActionEvent)

```
public static void main(String args[]) {
  String[] a = {"Hans", "Karin", ...};
  Arrays.sort(a, new Comparator<String>() {
    public int compare(String a, String b) {
      return -a.compareTo(b);
    }
  });
}
```

In Scala

```
def main(args: Array[String]): Unit = {
  val a: List[String] = List("Hans", "Kirsten", "Martin")
  val sortiert: List[String] =
  a.sortWith((x: String, y: String) => x > y)
  println(sortiert)
}
```

- `sortWith` ist eine Methode der Klasse `List`
- ihr wird ein Funktionsobjekt übergeben

Funktionen höherer Ordnung machen den funktionalen Stil aus!

Bestimme die Quadratsumme der Primzahlen von 2 bis 100, die auf eine durch 3 teilbare Zahl folgen

```
object Quadratsumme {
  def main(args: Array[String]): Unit = {
    val result: Int = (2 to 20).filter(x => isPrime(x) && (x - 1) % 3 == 0).map(x => x * x
        ).sum
    println(result)
  }
  def isPrime (n: Int) = (2 until n).forall(n % _ != 0)
}
```

Funktionen höherer Ordnung machen den funktionalen Stil aus!

Der Algorithmus ist nicht optimiert, aber verständlich und einfach parallelisierbar (Multicore)

```
object Quadratsumme {
  def main(args: Array[String]): Unit = {
    val result: Int = (2 to 20).par.filter(x => isPrime(x) && (x - 1) % 3 == 0).map(
        x => x * x).sum
    println(result)
  }
  def isPrime (n: Int) = (2 until n).forall(n % _ != 0)
}
```

Ein paar Funktionen höherer Ordnung für Sequenz(-ähnliche)-Objekte (wie Array, List, Stream)

Für alle diese Typen mit Typparameter A

```
def map[B](f: A => B): Seq[B] // wende f auf jedes Element an
def flatMap[B](f: A => Seq[B]): Seq[B] // konkateniere die Ergebnisse
def filter(p: A => Boolean): Seq[A] // diejenigen Elemente mit p(e) == true
def foldLeft[B](zero: B)(f: (B, A) => B): B // ... f(e2, (f (e1, zero))
def reduceLeft(f: (A, A) => A): A // ...f(e3, (f (e2, e1))
def forall(p: A => Boolean): Boolean // p gilt fuer alle Elemente
def foreach(b: A => Unit): Unit // fuehre den Block fuer jedes Element aus(mit
    Seiteneffekten !)
```

- viele andere: `collect, count, find, foldRight, lastIndexWhere, maxBy, ...`
- natürlich „normale“Funktionen: `head, tail, drop, take ...`
- Generator-Funktionen zum Erzeugen von Sequenzen: `iterate, tabulate ...`
- Mehr in der Scala API

Zum Schluss: Quicksort

```
def sort(liste: List[Double]): List[Double] = liste match {
  case Nil => Nil
  case pivot :: rest =>
  val (small, large) = rest.partition(_ <= pivot)
  sort(small) ::: pivot :: sort(large)
}
```

- `partition` ist ebenfalls eine vordefinierte höhere Funktion. Sie gibt für eine Liste ein Paar von zwei Listen zurück `(small,large)`.