Funktionale Programmierung

- Definition
- Historie
- Motivation
- Die Programmiersprache Scala
- Einfache funktionale Programme
- Auswertung von Ausdrücken
- match .. case

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.

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.

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, die die funktionale Programmierung unterstützt, ermöglicht erheblich einfacheren Umgang mit Funktionen.

Scala ist objekt-funktional: Das Verhalten von Objekten wird durch funktionale Methoden bestimmt.

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**

Geschichte der funktionalen Programmierung

Alonso Church definierte 1936 ein 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 von 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).

Erstes Beispiel (LISP)

Ein Programm besteht in der Auswertung eines funktionalen Ausdrucks:

```
(+ (* 4 5) (/ 20 4)) → (+ 20 5) → 25
```

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

```
((lambda (x) (* 3 x)) 10) →  30
```

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

```
(define mal3 (lambda (x) (* 3 x)))   // formale Definition
(define (mal3 x) (* 3 x))            // abgekürzte Syntax
(mal3 5) → 15

(define (make-erhoehe betrag)        // eine Funktion erzeugt eine Funktion
     (lambda (x) (+ x betrag)))
(define erhoeheUm3 (make-erhoehe 3))

(define liste '(1 2 3 4))
(define liste+3 (map erhoeheUm3 liste)) → (4 5 6 7) // map = Funktion höherer Ordnung
```

Die LISP-Syntax ist sehr einfach und sehr flexibel – aber für Java/C-Programmierer ziemlich ungewohnt.

Deshalb bevorzuge ich Scala – auch um zu zeigen, dass funktionale Programmierung nicht exotisch ist!

Java 8 ...

```
public static void main(String[] args) {
    Integer[] a = {8, 3, 1, 4, 0, 7};
    Arrays.sort(a, (x,y) -> y - x);
    String[] b = {"x", "a", "b"};
    Arrays.sort(b, (x,y) -> y.compareTo(x));

    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

Entwickelt ab 2001 am EPFL (Lausanne) von Martin Odersky.
Scala enthält auch eine ganze Menge von möglichen Verbesserung von Java:

- Keine statischen Klassenelemente
- Alles ist ein Objekt (auch Zahlen und Funktionen)
- Scala fördert die Verwendung von unveränderlichen 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 erlaubt die prozedurale Programmierung.
- Scala ermöglicht die Verwendung aller Java Klassen.

- **Das Ziel ist die Erforschung der objekt-funktionalen Programmierung.**

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: Mustererkennung, toString, equals, hashCode, Generator

`object`

singuläres Objekt mit globalem Namen und anonymer Klasse

`case object`

unveränderliches Objekt mit: toString, equals, hashCode

```
class XYZ(a: Double, b: Double) {   // primärer Konstruktur
    private val c = a + b

    def getc: Double = c            // default = public
                                    // keine () nötig

}
```

Hello World (prozedural)

```
object HelloScala {
  def main(args: Array[String]): Unit =  // Unit-Funktion (void)
     printHello()

  def printHello(): Unit = println("Hello World")
}
```

Anmerkungen:

`object, def:`	Alle Deklaration beginnen mit einem Schlüsselwort Der Typname steht hinter dem Variablennamen
`object`:	Klasse mit einer einzigen Instanz (Singleton)
`Unit, Int, Long`:	Alle Typnamen werden groß geschrieben. Alles sind Objekte!
`Array[String]`:	Scala behandelt Arrays wie parametrisierte Typen.

Unit is a subtype of scala.AnyVal. There is only one value of type Unit, (), and it is not represented by any object in the underlying runtime system. A method with return type Unit is analogous to a Java method which is declared void.

Kommandozeile:

```
scalac Hello.scala
scala HelloScala
java HelloScala
```

(Voraussetzung: Scala-Bibliothek ist im CLASSPATH)

Weitere Beispiele (prozedural)

```
object IntFunctions {
  def summe(array: Array[Int]): Int = {
     var sum = 0                  // var sum: Int = 0 (Variable)
     for (x <- array)             // = foreach (es gibt kein C-for!)
         sum += x
     sum                          // Rückgabewert
  }

  def indexOf(x: Int, array: Array[Int]): Int = {
     var index = 0
     while (index < array.length && array(index) != x)
         index += 1               // kein i++, Indizierung: array(index)
     index
  }

  def max(array: Array[Int]): Int = {
     var m = array(0)
     for (i <- 1 until array.length)    // to = incl./ until excl.
       if (array(i) > m) m = array(i)   // prozedurales if
     m
  }
}
```

Weitere Beispiele (funktional)

```
object IntFunctions {
  def summe(list: Seq[Int]): Int =
     if (list.isEmpty) 0 else list.head + summe(list.tail)

  def indexOfGeneric[T](x: T, array: Array[T]): Int = {
                                              // Typparameter
     @tailrec                                 // ist garantiert endrekursiv
     def indexFrom(index: Int): Int =         // lokale Funktion
        if (index >= array.length)
          -1
        else if (x == array(index))
          index
        else
          indexFrom(index+1)

     indexFrom(0)                             // lokale Berechnung
  }

  def quadratSumme(list: Seq[Double]): Double =
     list.map(x => x * x).sum
}
```

== ruft das `equals()` des referierten Objektes auf (Pointergleichheit mit dem Operator `eq`).

Besonderheiten

```
1.toString     // Zahlen sind Objekte
x.max(y)
x max y        // Punkt kann fehlen (Operatorschreibweise)
"abc" == x     // == anstelle von equals
"abc".==(x)    // Schreibweise als Methodenaufruf (natürlich unüblich)
"abc" >= x     // Operator >= anstelle von compareTo

val a = Array(1,2,3)       // Erzeugung & Initialisierung
val b = new Array[Int](3)  // Erzeugung
val c = List(1,2,3)        // Erzeugung & Initialisierung
var d = List[Int]()        // Typparameter nötig, da keine Daten !
var e: List[Int] = Nil     // oder sp

d = 1 :: d       // "Cons"-Notation (::) ähnlich wie Prolog([ | ])
                 // ( bedeutet: d.::(1) )

d match {        // Pattern-Matching bei Listen (ähnlich Prolog)
  case Nil  => ...
  case h::t =>
}

def length[T](list: List[T]): Int = list match {
  case Nil  => 0
  case _::t => 1 + length(t)
}
```

Besonderheiten

- **val** = unveränderliche Variable, **var** = veränderliche Variable, **def** = Methode/Funktion
- Methodenparameter sind unveränderlich
- **object** = singuläres Objekt
- Typparameter in eckigen Klammern, Arrayindizes in runden Klammern
- Lokale Funktionen. Umfassende Variablen gelten innen weiter
- **if** ist funktional (bedingter Ausdruck)
- **match** .. **case** Ausdruck (Pattern-Matching)
- **For =** for-Ausdruck, foreach-Statement
- == entspricht immer equals (für Referenzvergleich gibt es eq)
- Sonderzeichen sind als Funktions-/Methodennamen erlaubt

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.)
- Die Geschichte der Programmiersprachen ist der Versuch die Nachteile der imperativen Programmierung zu beheben (modulare Programmierung, objektorientierte Programmierung ...)

- Imperative Programmierung ist rechnernah => effizient (auch bei schlechtem Compiler).
- Funktionale Programme müssen durch den Compiler in einen effizienten Ablauf umgesetzt werden.

Imperative Ablaufverfolgung ist nicht einfach

```
1   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)

Zeile	**n**	**i**	**f**
3	3	-	1
4	**3**	**0**	**1**
6	3	1	1
4	**3**	**1**	**1**
6	3	2	1
4	**3**	**2**	**2**
6	3	3	2
4	**3**	**3**	**6**
8	**3**	**3**	**6**

Um Abläufe besser zu verstehen, braucht man Invarianten. Hier: `i! = f`

Beachte Iteratoration braucht bei N Wiederholungen O(1) Speicherplatz!

Ablaufverfolgung von Rekursion ist besonders schwierig

```
int fak(int n) {
    int r;
    if (n == 0)
        r = 1;
    else
        r = n * fak(n - 1);
    return r;
}
```

fak(3)

1.		2.		3.		4.		*rekursiver Aufruf*
n	**r**	**n**	**r**	**n**	**r**	**n**	**r**	*aktuelle lokale Variablen*
3	-							*Werte der Variablen*
		2	-					
				1	-			
						0	-	
						1	**1**	
				1	**1**			
		2	**2**					
3	**6**							

Fazit: das Verständnis imperativer Programme ist schwierig.
Rekursion ist imperativ schwer verständlich.
Imperative Programme sind fehleranfällig !!!

Beachte: Rekursion braucht bei N-Wiederholungen O(n) Speicherplatz

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

```
def mal3(x: Double) = 3 * x
def plus1(x: Double) = x + 1

plus1(mal3(7)-plus1(2))
=  plus1((3*7) - (2+1))
=  plus1(21 - 3)
=  plus1(18)
=   (18 + 1)
=  19
```

- Auswertung = Gleichungsumformung
- Funktionsanwendung = Ersetzen des Aufrufs durch den Körper, und Ersetzung der Parameter durch die Argumente
- **Grundsätzlich ist die Auswertungsreihenfolge beliebig.** (lazy Evaluierung, call by name*,* Nebenläufigkeit)!

Scala ist nicht *streng* funktional! Es lassen sich Funktionen mit Seiteneffekt schreiben.

(Effekt der Funktion: Bestimmung des Ergebniswertes - *Seiteneffekt*: jede andere Wirkung)

Rekursion als Auswertung

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

- Man kann die Umformung auch kürzer schreiben.
- Sobald die Ersetzung ein Ende hat, ist man fertig!

Vergleich Endrekursion - Normalrekursion

```
factorial(4)
=  fac(4, 1)
=  fac(3, 4)
=  fac(2,12)
=  fac(1,24)
=  fac(0,24)
=  24
```

Variable: n, f

```
factorial(4)
= (4 * factorial(3))
= (4 * (3 * factorial(2)))
= (4* (3 * (2 * factorial(1))))
= (4 * (3 * (2 * (1 * factorial(0)))))
= (4 * (3 * (2 * (1 * 1))))
= (4 * (3 * (2 * 1)))
= (4 * (3 * 2))
= (4 * 6)
= 24
```

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

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))

Beachten Sie 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]    //     return f
   6  ireturn
   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 // hier stand der Aufruf!
  14  istore_1 [n]   // n = n -1
  15  goto 0         // Wiederholung
```

Ausführbarer Code ist (fast) immer imperativ. Es ist aber nicht die Aufgabe des Programmierers, sondern die Aufgabe des Compilers diesen Code zu erzeugen!

Es gab und gibt immer mal wieder 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.

Match-Case Ausdruck

Der Match-Case Ausdruck erlaubt unterschiedliche Formen der Mustererkennung (einschließlich) der Aufgaben der „normalen“ switch-Anweisung anderer Programmiersprachen.

Es ist aber wie das if keine Anweisung, sondern die Beschreibung eines funktionalen Sachverhaltes.

```
val monatsNummer = readInt
val monat: String = monatsNummer match {
    case 1 => "Januar"
    case 2 => …
}
```

Der Case-Teil kann auch `Variablen` erkennen und er kann auch mit einer Bedingung (guard) versehen sein:

```
def sign(x: Double): Double = x match {
    case 0           =>  0.0
    case _ if x < 0 => -1.0
    case _           => +1.0
}
```

Der Unterstrich _ ist ein Wildcard (entspricht dem default:)

Algebraische Datentypen und Case-Klassen / Pattern-Matching

Das Pattern-Matching der funktionalen Programmierung korrespondiert zu Algebraischen Datentypen.

Anders als bei Prolog gibt es in Scala Typdeklarationen und Vererbung.

Beispiel:

```
sealed abstract class Tree[+T] {
  def traverse: List[T] = this match {
     case Leaf(x) => List(x)
     case Node(links,rechts) => links.traverse ::: rechts.traverse
  }
}
case class Node[T] extends Tree[T] (left: Tree[T], right: Tree[T])
case class Leaf[T](value: T) extends Tree[T]
```

Oder außerhalb der Klassen durch:

```
def traverse[T](n: Tree[T]): List[T] = n match {
    case Leaf(x)   => List(x)
    case Node(l,r) => traverse(l) ::: traverse(r)
}
```

Gibt es noch weitere Möglichkeiten?

sealed = *nur in dieser Datei soll Vererbung möglich sein*

Abschließende Bemerkungen

- Pattern-Matching hat noch viele weitere Features
- Wenn normale (nicht case) Klassen das Pattern-Matching unterstützen sollen, braucht man „extractors“ (unapply)
- Case-Ausdrücke als Partielle Funktion kommt noch
- Die Besonderheiten der For-Comprehension kommen noch
- Beachten Sie „Scala by Example“ (frei auf der Scala-Seite)

Algebraische Datentypen und Case-Klassen / Objektorientierung

Objektorientierung nutzt die Methodenauswahl (dispatching) der späten Bindung.

Beispiel:

```
sealed abstract class Tree[+T] {
  def traverse: List[T]
}

case class Node[T] extends Tree[T] (left: Tree[T], right: Tree[T]) {
  def traverse = left.traverse ::: right.traverse
}

case class Leaf[T](value: T) extends Tree[T] {
  def traverse = List(value)
}
```

Objektorientierung gruppiert Aktionen zu Klassen

Funktionale / Prozedurale Programmierung konzentriert die Operationen

Am Rande: Das Konstruktionsprinzip Algebraischer Datentypen =

Summe = Attribute einer Klasse, Produkt = Varianten durch Vererbung