

# Grundelemente der funktionalen Programmierung

- Rekursion und Endrekursion
- Closures
- Call by Name und Kontrollabstraktion
- Currying
- partiell angewendete Funktionen
- partiell (definierte) Funktion
- Funktionen höherer Ordnung
- for-Comprehension
- Verwendung unveränderlicher Datenstrukturen
- Implementierung unveränderlicher Datenstrukturen
- Objektorientierung und funktionale Programmierung

## Rekursion

- Bei funktionaler Programmierung ist Rekursion das wichtigste Ausdrucksmittel für Wiederholung. Rekursion ist gleichzeitig eines der (formal) am einfachsten erklärbareren „Kontroll“-Konstrukte.
- Auf einer höheren Ebene macht funktionale Programmierung häufig Wiederholungskonstrukte unnötig. Man verwendet statt dessen Funktionen höherer Ordnung.
- Die „for-comprehension“ von Scala ist eine syntaktisch nette Schreibweise für besondere Funktionen höherer Ordnung (foreach, map).

## Normale Rekursion mittels Fallunterscheidung und Mustererkennung

```
def fib(n: Int): BigInt =  
  if (n == 0) 0  
  else if (n == 1) 1  
  else fib(n - 1) + fib(n - 2)
```

```
def fib(n: Int): BigInt = n match {  
  case 0 => 0  
  case 1 => 1  
  case n if n > 1 => fib(n - 1) + fib(n - 2)  
}
```

### Fazit:

- Beide Varianten sind möglich.
- Die match-case Form kommt der funktionalen Denkweise vielleicht am nächsten.
- `case .. if` ist ein Muster mit Zusatzbedingung („guard“)
- Wenn keiner der Fälle zutrifft, wird eine Exception geworfen.
- Es gibt die Möglichkeit, mit `case _` einen Default („catch-all“) zu definieren.

(Natürlich ist dieser Algorithmus ineffizient, das liegt aber nicht an der Rekursion, sondern am Algorithmus)

## Wie sieht die iterative Variante aus?

```
def fib(n: Int): BigInt = {  
  var g = BigInt(0)  
  var f = BigInt(1)  
  var i = n          // Parameter n darf nicht verändert werden  
  while (i != 0) {  
    val t = f + g  
    g = f  
    f = t  
    i -= 1  
  }  
  g  
}
```

**Anmerkung:** Scala ist vom funktionalen Paradigma beeinflusst (keine veränderlichen Parameter, kein i++ mit Seiteneffekt)

## Wie kommen wir zur Endrekursion?

- Für while-Schleifen verwenden wir eine lokale Funktion.
- Lokale Variablen werden zu Parametern der lokalen Funktion.
- Bei Abbruch der lokalen Funktion steht das Ergebnis fest.
- Die Parameter (lokale Variablen) werden beim Aufruf durch die äußere Funktion initialisiert.
- Da die Zuweisung zu Parametern (beim Aufruf) gleichzeitig erfolgt, entfallen Hilfsvariablen.
- Viele Programmiersprachen übersetzen Endrekursion in einen effizienten Ablauf.
- Die äußere Funktion kann Vorbedingungen prüfen.

```
def fib(n: Int): BigInt = {  
  require(n >= 0)  
  def ffib(i: Int, f: BigInt, g: BigInt): BigInt =  
    if (i == 0) g else ffib(i - 1, f + g, f)  
  ffib(n, 1, 0)  
}
```

## Methoden und Closures

Methoden sind Funktionen, die von einem Objekt ausgeführt werden. Das Objekt spielt dabei die entscheidende Rolle:

- Es entscheidet darüber, welche Methode ausgeführt wird.
- Die Methode kann auf Instanzvariablen des Objekts zugreifen und kann diese verändern!.

Für Funktionen und Closures gelten andere Regeln:

- Funktionen stellen (zunächst) nur die Zuordnung von Funktionsargumenten zu Resultaten dar.
- Funktionen haben gebundene Variablen, (lokale Variablen) und freie Variablen aus der Umgebung der Funktion (äußere Funktion, Klasse, Objekt).
- **Ein Closure ist ein Funktionsobjekt, das innerhalb eines Programms weitergegeben werden kann und dabei seine Umgebung von freien Variablen „mitnimmt“.**

## Beispiele

```
def defineParabola(a: Double, b: Double, c: Double) = {  
  def parabola(x: Double) = c + x * (b + x * a)  
  parabola _  
}
```

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

```
val p121 = defineParabola(1, 2, 1)
```

```
println(p121(2)) // ergibt: 9
```

### Anmerkung:

- In der ersten Form wird eine lokale Funktion definiert (freie Parameter a, b, c). Sie wird nicht angewendet, sondern direkt zurückgegeben (Scala verlangt den Unterstrich `_`. Man macht damit deutlich, dass man keine Parameter vergessen hat).
- In der zweiten Form wird eine anonyme Funktion definiert und zurückgegeben.
- Closures können an Variablen gebunden werden (auch `var`) und an Funktionen übergeben werden. Auch wenn die Umgebungsfunktion beendet ist, wird ihr Kontext mitgenommen.
- Scala erlaubt bei der Definition anonymer Funktionen eine Reihe von kontextabhängigen Abkürzungen, das ist praktisch, manchmal etwas verwirrend, aber: es ist kein entscheidendes Merkmal.

## Java unterstützt Closures nicht, kennt aber was Ähnliches

```
public void sortUpOrDown(String[] a, final boolean isAscending) {
    Arrays.sort(a, new Comparator<String>() {
        public int compare(String x, String y) {
            if (isAscending) {
                return x.compareTo(y);
            } else {
                return y.compareTo(x);
            }
        }
    });
}
```

### Anmerkungen :

- Comparator-Objekte haben die Aufgabe, die Funktion `compare` zu transportieren!
- Um Erzeugung einmaliger Objekte zu erleichtern, kennt Java anonyme Klassen.
- Freie Variable aus einer lokalen Umgebung müssen `final` deklariert werden (hier `isAscending`).

# Currying

Der Name steht für Haskell Curry, der diese Technik u.a. entwickelt hat.

**Unter Currying versteht man die Beschreibung von Funktionen mit mehreren Parametern durch eine Folge von Funktionen, die jeweils eine neue Funktion ergeben.** Ursprünglich wurde die Technik eingeführt, um Funktionen mit mehreren Parameter auf die Theorie von Funktionen mit einem Parameter zurückführen zu können. In Programmiersprachen dient die Technik der Erweiterung der Notation. Die Technik ist eine einfachste Anwendung der funktionalen Programmierung.

Häufiger Verwendungszweck: gewünschte Syntax, z.B. bei Kontrollabstraktion, DSL

## Beispiele:

```
def summe(a: Double, b: Double) = a + b  
summe(3, 4) ergibt 7
```

ist gleichbedeutend mit

```
def summe(a: Double)(b: Double) = a + b  
summe(3)(4) ergibt 7
```

wir hatten das schon zuvor:

```
defineParabola(1,0,1)(10) ergibt 101
```

## Call by Name

Es gibt viele Varianten der Parameterübergabe:

- **call by reference**: die Adresse wird übergeben, so dass man auch den Inhalt einer Variablen verändern kann: mächtig, führt zu schwer verständlichem Code (prozedural)
- **call by value**: der Wert eines Ausdrucks wird übergeben: funktional und sicher
- **copy in/copy out**: gibt es in verschiedenen Varianten. Prozedural aber einigermaßen sicher. call by value ist gleich copy in. Return entspricht copy out.
- **call by name**: es wird ein Ausdruck als Closure übergeben. Er wird nicht bei dem Aufruf ausgewertet, sondern jedesmal erneut, wenn auf den Parameter zugegriffen wird.

**Mit call by Name lassen sich eigene Kontrollanweisungen programmieren (Kontrollabstraktion)**

## Beispiel für Kontrollabstraktion 1

*Eine Anweisungsfolge wird mehrfach ausgeführt (prozedural)*

```
def repeat(n : Int)(block: => Unit) {  
    for(i <- 1 to n) block  
}
```

### Anmerkungen :

- `=>` vor dem Datentyp und bezeichnet call by name
- `Unit` steht dafür, dass kein Ergebnis ermittelt wird (wie `void`).
- Die Schleife wird n-Mal ausgeführt. Jedesmal wird `block` erneut ausgewertet.

```
var i = 10  
repeat(3) {  
    println(i)  
    i = i + i  
}
```

Scala erlaubt, dass man die Parameter in `{ .. }` einschließt.  
Der Wert von `i` ist am Ende = 80

## Beispiel für Kontrollabstraktion 2

*Eine Funktion wird bei Fehler wiederholt.*

```
def asLongAs[T] (message: String) (expression: =>T): T {  
  try {  
    expression  
  }  
  catch {  
    case _:Exception =>  
      println(message)  
      asLongAs (message) (expression)  
  }  
}
```

```
val i = asLongAs("falsche Eingabe") { readInt }
```

### Anmerkungen :

- Der Rückgabetyt wird vom Compiler ermittelt
- `catch` folgt der Syntax für `match - case`

Kontrollabstraktionen erlauben, die Sprache um höhere Konstrukte zu erweitern.

Kontrollabstraktionen bilden eine der Grundlagen für interne DSLs (domain specific language)

## Partiell angewendete Funktionen

Unter partieller Anwendung einer Funktion versteht man die teilweise Festlegung von Funktionsparametern. Dadurch die Festlegung von x-Parametern wird einer Funktion von n-Parametern eine Funktion von n-x Parametern zugeordnet.

### Beispiele:

```
def summe(a: Double, b: Double) = a + b // Funktion mit 2 Parametern
val plus3 = summe(3, _: Double)
plus3(7) ... ergibt 10
```

oder

```
def plusX(x: Double) = summe(x, _: Double)
val plus17 = plusX(17)
```

```
def sum(a: Double)(b: Double) = a + b
val curryPlus3 = sum(3)_ // Currying liefert ja immer Funktionen
```

## Partiell definierte Funktionen

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

### Beispiel:

```
val faculty: PartialFunction[Int,Int] = {  
  case 0 => 1  
  case n if n > 0 => n * faculty(n - 1)  
}
```

ist nur für nicht negative Zahlen definiert.

`faculty.isDefinedAt(-3)` ergibt: `false`

`faculty(3)` ergibt (ganz normal) 6

`faculty(-3)` wirft eine Ausnahme

## Funktionen höherer Ordnung

In funktionalen Sprachen sind Funktionsobjekte selbst vollwertig. Daraus ergibt sich, dass man Funktionen flexibel anwenden kann, in dem man ihnen ihrerseits Funktionen als Argumente übergibt.

Erstes Beispiel „sortieren eines Arrays“ (stören Sie sich nicht an der objektorientierten

### Schreibweise:

```
val a = Array(36, 18, 49)
val decreasing = (x: Int, y: Int) => x >= y
a.sortWith(decreasing)           // ergibt: a = (49, 36, 18)
a.sortWith((x,y) => x <= y)      // ergibt a = (18, 36, 49)
                                // (Funktionsliteral, Typ von x,y: INT)
a.map(x => x * 3)                 // ergibt a = (54, 108, 147)
```

### Abkürzungen:

```
a.sortWith(_ <= _)              // _: anonymer Parameter
a.map(_ * 3)
```

**Anmerkung:** Arrays sind veränderliche (*mutable*) Datenstrukturen, daher wird das Array auf der Stelle verändert.

Bei unveränderlichen Strukturen (*immutable*), wie Listen, werden neue Objekte erzeugt.

## Ein paar höhere Funktionen für Sequenz-Objekte (wie Array, List)

```
val seq1 = List(x,y,z) // x,y,z stehen für bestimmte Werte  
val seq2 = List(a,b,c) // ditto
```

```
seq1.map(f) => List(f(x), f(y), f(z))  
seq1.reduceLeft(f) => List(f(f(x,y), z)) // als Operator: (x f y)  
seq1.filter(f) => List(xi) mit f(xi) == true  
seq1.partition(f) => (List(xi), List(yi)) // f(xi) == true, f(yi) == false  
seq1.foreach(f) => wende f auf jedes Element an (mit Seiteneffekt)  
seq1.zip(seq2) => List((x,a), (y,b), (z,c))  
    ... to zip: Reisverschluss schließen
```

### Standardbeispiel: Skalarprodukt:

```
def dotProduct(a: List[Double], b: List[Double] =  
    a.zip(b).map(x => x._1 * x._2).reduceLeft((x,y) => x + y)  
  
val x = (a, b, c) ist ein Tupel. x._1 == a, x._2 == b, x._3 == c
```

## Und wie sieht quicksort für Listen aus?

```
def quicksort(list: List[Double]): List[Double] =  
  if (list.isEmpty)  
    list  
  else {  
    val (small, large) = list.tail.partition(x => x <= list.head)  
    quicksort(small) :: list.head :: quicksort(large)  
  }
```

### Anmerkungen:

**val** (small, large) = .. Bei der Tupelzuweisung kann dieses „zerlegt werden.“

a :: b = append(a, b)

a :: b = Cons(a, b) (in Prolog [a|b])

Fällt Ihnen eine Ähnlichkeit zwischen Quicksort und der Inorder-Traversierung von sortierten Suchbäumen auf?

## For-comprehension

- *to comprehend: = verstehen.*

In der funktionalen Programmierung:

*Traversieren einer Datenstruktur und Ermittlung relevanter Information*

- Gegenstück zur for-Anweisung von Java
- Funktional durch Funktion höherer Ordnung definiert

```
for(x <- liste) { println(x) }  
== liste.foreach(println(_))
```

```
for(i <- 1 to n) { val a = bearbeite(i); undDann(a, i) }  
== (1 to n).foreach{ i => {val a = bearbeite(i); undDann(a, i)} }
```

```
for(x <- liste) yield(x * x)  
== liste.map(x => x * x)
```

```
for(x <- liste if x > 10) yield(x)  
== liste.filter(x => x > 10)
```

es gibt noch ein paar weitere Varianten ...

## Unnötige Veränderlichkeit ist schlecht

```
class Rational {
    private int z, n;

    public void add(Rational summand) {
        z = z * summand.n + n * summand.z;
        n *= summand.n;
        kuerzen();
    }
}
```

```
...
Rational a = new Rational(1,2);
...
Rational b = a;
a.add(new Rational(1,2)); // gleichzeitig ändert sich b!!!
```

**Fazit:** *Brüche stellen Werte dar! Sie sollten als unveränderliche Objekte realisiert werden!!*

```
public Rational add(Rational summand) {
    return new Rational(z * summand.n + n * summand.z,
                        n * summand.n)
}
```

## Unveränderliche Datenstrukturen

Funktionale Programmierung kennt keine veränderlichen Inhalte. Konsequenterweise verwendet sie unveränderliche Datenstrukturen (das Gegenteil sind destruktive Datenstrukturen / Operationen).

Grundsätzlich hat die Beschränkung auf Unveränderlichkeit Implementierungs-Nachteile:

- Es entstehen und vergehen immer wieder neue Objekte (garbage collection!)
- Viele Operationen erfordern Kopien
- Manchmal ist der wahlfreie Zugriff aufwändig (im Unterschied zum Array)

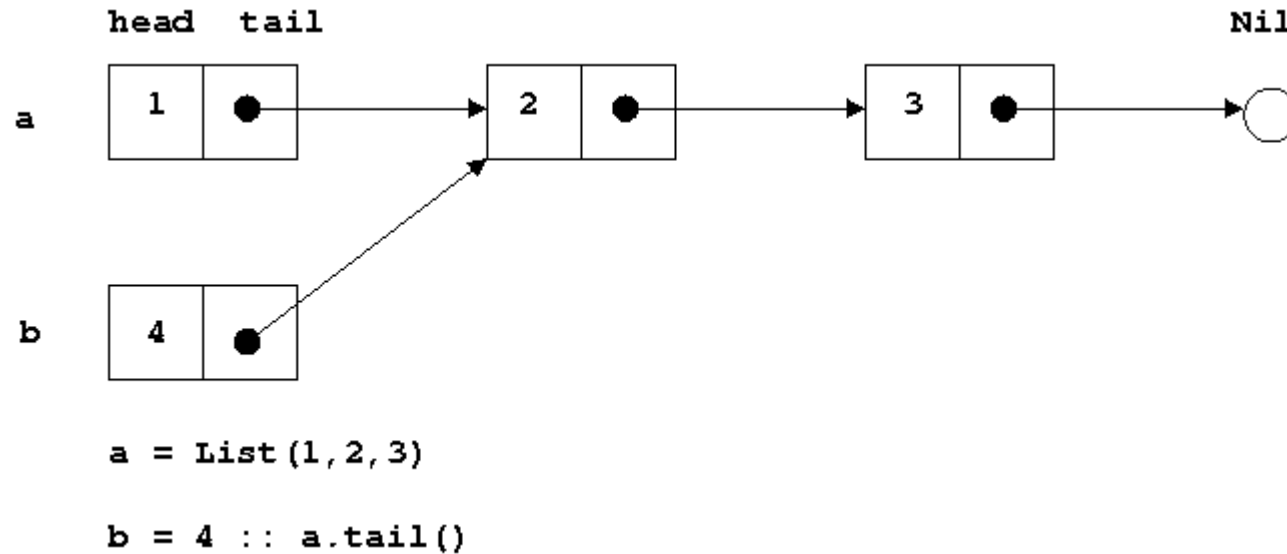
Es gibt aber auch Vorteile:

- Es sind keine „Sicherheits“-Kopien nötig.
- Mehrere Datenstrukturen können gemeinsamen Speicher verwenden.
- Im Rahmen der Nebenläufigkeit sind unveränderliche Datenstrukturen besonders robust.

In vielen Fällen werden die Nachteile durch besondere Algorithmen ausgeglichen.

Wie die Logikprogrammierung kennt auch die funktionale Programmierung Listen als grundlegende Datenstruktur. Man kann davon ausgehen, dass sie als einfach verkettete Listen implementiert sind.

## Implementierungsmodell für Listen



### Scala-Implementierung:

Klasse `::` mit `head` und `tail` und das Objekt `Nil` (der Klasse `Nil`)

Grundoperationen in  $O(1)$ : `head`, `tail`, `cons-Operation (::)`, `isEmpty`

andere Operationen sind  $O(n)$ : `append-Operation (:::)`, `reversed`, `size`, `n-tes Element`

## Verwendung unveränderlicher Datenstrukturen.

Insbesondere bei der Verwendung von Funktionen höherer Ordnung gestalten sich viele Operationen sehr elegant:

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

### Anmerkungen:

- Die Kopie beeinträchtigt die Effizienz nur unwesentlich.
- Problematischer ist die Wahl des Pivotelements.
- `val (small, large)` ist ein funktionales Konstrukt von Scala (Tupel). Es bündelt mehrere Werte.
- Vergleichen Sie den Algorithmus mit der Prolog-Variante!

## Beispiel-Implementierung von Listenklassen (vereinfacht)

Es geht hier um das Prinzip (insbesondere auch Nil-Objekt). Dafür wurde Scala-Eigenheiten weggelassen..

```
abstract class List[T] {  
  // abstrakte Methoden:  
    def isEmpty: Boolean  
    def head: T  
    def tail: List[T]  
  // konkrete Methoden: ...  
}  
  
object Nil extends List[Nothing] {  
  override def isEmpty = true  
  override def head = throw new NoSuchElementException  
  override def tail = throw new NoSuchElementException  
}  
  
case class Cons[T] (head: T, tail: List[T]) extends List[T] {  
  override def isEmpty = false  
}
```

# Objektorientierung und funktionale Programmierung

Objektorientierung hat mit der *Struktur* von Objekten zu tun, nicht mit Abläufen.  
Durch Modularisierung unterstützt OOP prozedurale Programmierung.  
Wenn sinnvoll, kann man OOP auch mit funktionaler Programmierung kombinieren.

## **Insbesondere:**

- Wertobjekte sind im Kern funktional (String, Zahlenklassen)
- Identität bei gleichen Inhalten (vgl. Scala case class)

## **Veränderliche Objekte**

- Identität bei identischer Referenz.

Aus „everything is an object“ folgt: auch Funktionen (Closures) sind Objekte!

## **Offene Probleme:**

- Komplexes statisches Typsystem.
- Bibliothek
- Optimierung