Grundelemente der funktionalen Programmierung

- Einstieg
- Lambda-Ausdrücke, Closure-Eigenschaft
- Späte Evaluierung
- Currying
- Partielle Funktion
- Funktionen höherer Ordnung
- for-Comprehension
- Verwendung unveränderlicher Datenstrukturen / ihre Implementierung
- Unvollständige Datenstrukturen (Streams)
- Objektorientierung und funktionale Programmierung

Vorbemerkung

Wie immer, kann man Funktionen in verschiedener Form schreiben. Das macht syntaktische Unterschiede aus, ändert aber nichts am Konzept!

a) als Methode einer Datenklasse

```
class List[+A] {
  def foldLeft[B >: A](z: B)(f: A => B): B = …
```

b) als Methode eines Objekts (annähernd eine Funktion)

```
object ListFunctions {
  def foldLeft[A, B>:A](lst: List[A], z: B)(f: A=>B): B = …
```

c) mit mehreren Parameterlisten (Currying):

```
  def foldLeft[A, B>:A](lst: List[A])(z: B)(f: A=>B): B = …
```

d) Definition einer Funktion mit einer Methode

```
  val fold = ListFunctions.foldLeft _
```

e) durch einen Lambda-Ausdruck:

```
  val add = (x:Double,y: Double) => x + y
```

f) **durch eine Funktion ...**

Von Iteration zu Endrekursion

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

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

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.
- Die äußere Funktion kann Vorbedingungen prüfen.

```
def fib(n: Int): BigInt = {
  require(n >= 0)

  @tailrec   // der Compiler prüft, 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)
}
```

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

Etwas genauer …

Vorweg wir haben 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***.)
- **Fehler:** häufig liest man, dass Closures „~~anonyme Funktionen~~“ sind:
  a) mit Closure beschreibt man nur *eine Eigenschaft* von Funktionsliteralen
  b) im Unterschied zu Java, müssen Funktionen genauso wenig einen eigenen Namen haben wie Objekte.
  c) 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 => 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.

- Ich betrachte Methoden eines Scala-Objects manchmal (formal nicht ganz korrekt) als Funktion.
  (Scala ordnet zwecks Kompatibilität mit Java den Objektfunktionen Klassenfunktionen zu)

Beispiel 1: Methode / Funktion

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

// Objekt a: A
val a = new A(2.5)
// Funktion f: (Double) => Double)
val f = a.MalY _

// Aufruf von Methode mit Objekt
println(a.malY(4))
// Die Funktion ist in sich abgeschlossen (closure)
println(f(4))
```

Fazit:

`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
}

// Funktion f: (Double,Double) => Double)
import Function.add
val f = add _

// Aufruf von Methode mit Objekt
println(add(3,4))
// Der Funktionsaufruf sieht nicht anders aus.
println(f(3,4)
```

Fazit:
Innerhalb von **`object`** … verschwindet der Unterschied zwischen Funktion und Methode beinahe:

– Funktionen sind Objekte!!
– Methoden sind Teil eines Objekts

Beispiel 3: lokaler Kontext

```
type DoubleFkt = Double => Double  // die Typangaben dienen der Lesbarkeit
(type führt Namen für Typen ein)

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

def newParabola(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
```

Anmerkungen:

- In der ersten Funktion wird eine lokale Funktion definiert (freie Parameter a, b, c). Sie wird nicht angewendet, sondern direkt zurückgegeben. Mit dem _ wird sie zu einer abgeschlossenen Funktion.
- In der zweiten Form wird die Funktion durch einen Lambda-Ausdruck definiert und zurückgegeben.
- Auch wenn die Umgebungsfunktion beendet ist, wird ihr Kontext mitgenommen (closure).
- Der Scala Compiler erlaubt manchmal eine einfachere Schreibweise (keine Typangabe oder kein _)

Nicht strikte Funktionen, späte Evaluierung

Vereinfachte operationale Definition: 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 => 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)
}
```

Nicht strikte Funktionen sind Grundlage von Kontrollabstraktion und von unvollständigen Datenstrukturen!

Beispiel für Kontrollabstraktion

Eine Funktion wird bei Fehler wiederholt.

```
@tailrec
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 Rückgabetyp wird vom Compiler ermittelt
- `catch` folgt der Syntax für `match - case`

Kontrollabstraktionen erlauben, die Sprache um höhere Konstrukte zu erweitern

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-Herleitung und auch der Möglichkeit, nach und nach die Parameter zu binden.

Scala: Currying unterstützt die Typinferenz!

Beispiele:

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

wir hatten das schon zuvor:

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

Frage:

Welchen Typ hat `summe(3)` ?

Beispiel: Klassen und Objekte definiert durch Funktionen

- Eine Klasse ist eine Konstruktor-Funktion.
- Ein Objekt ist eine Closure mit freien Variablen und Methodenauswahl (`dispatch`)

```
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 'withdraw => withdraw
    case 'deposit  => deposit
  }
  dispatch
}

val b1 = newBalance(100)
val b2 = newBalance(200)
println("b1 " + b1('deposit)(50)) // b1.deposit(50)
println("b2 " + b2('deposit)(50))
println("b2 " + b2('withdraw)(50))
```

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 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

Wichtige Anwendung: Funktionsobjekt mit Pattern-Matching, z.B. bei Actors:

```
receive {                    // Botschaften, die nicht passen, werden nicht gelesen
    case Message(a, b) => …
    case Botschaft(x)  =>
    …
}
```

Anwendung von Funktionen höherer Ordnung

Definition:

Eine Funktion höherer Ordnung ist eine Funktion, der Funktion als Parameter übergeben werden.

Beispiel „Sortieren eines Arrays“

(stören Sie sich nicht an der objektorientierten **Schreibweise**):

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

Abkürzungen:

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

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:

```
def isPrime(n: Int) = (2 until n).forall(n % _ != 0)
(2 to 100).filter(x => isPrime(x) && (x - 1) % 3 == 0)
          .map(x => x * x)
          .sum
```

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

```
(2 to 100).par
          .filter(x => isPrime(x) && (x - 1) % 3 == 0)
          .map(x => x * x)
          .sum
```

Schon mal als „for-comprehension“:

```
val numbers = for {
       x <- (2 to 100).par
       if isPrime(x) && (x - 1) % 3 == 0
    } yield x * x
numbers.sum
```

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 für alle Elemente

def foreach(b: A=>Unit): Unit                führe den Block für jedes Element aus
                                             (mit Seiteneffekten !)
```

und viele andere: `collect, count, find, foldRight, lastIndexWhere, maxBy, ...`

und natürlich „normale“ Funktionen: `head, tail, drop, take, ...`
und Generator-Funktionen zum Erzeugen von Sequenzen: `iterate, tabulate ...`

Für die wichtigsten Dinge braucht man etwas Ordnung (Mathematik?) ...

Monoid (= Menge mit assoziativer Verknüpfung)

Mathematik: Ein Monoid ist eine Menge mit einer **assoziativen Operation** und einem **neutralen Element**:
(Addition, Multiplikation, Minimum, Maximum, …)

Objekte vom Typ A.
Neutrales Element **`zero: A`**
Verknüpfung **`f: A x A => A`**

Man kann eine Sequenz von Objekten zu einem einzigen Wert zusammenfassen (*fold*).
Wenn `f: B x A => B` und `zero: B` spricht man von monoidalen Mengen.

Scala:

```
def fold[A] (x: Seq[A]) (zero: A) (f: (A,A) => A): A
def foldLeft[A,B] (x: Seq[A]) (zero: B)(f: (B,A) => B): B
```

(Varianten: `foldLeft`, `foldRight` (monoidal) und `reduce`, `reduceLeft`, `reduceRight`)

Es gibt viele Anwendungen, zB:

```
def sum(l1: List[Double]): Double =
    l1.foldLeft(0.0) ( (s,x) => s + x )
def append(l1: List[T], l2: List[T]): List[T] =
    l1.foldRight(l2) ( (e, list) => e :: list )
```

Monaden

= formales Konzept zur Komposition von Funktionsanwendungen (moderne Algebra)

Eine monadische Operation verknüpft eine Monade mit einer Funktion zu einer neuen Monade

Gegeben: Grundtypen A, M[A] (ein Behälter für A).

Elementare Grundoperationen:
- *Konstruktor für ein Monadenobjekt (**unit**)*
- *Verknüpfung von Funktionen und Anwendung auf Monaden (**bind**)*

In Scala heißt die Grundoperation *bind* `flatMap`.

In Scala: *Grundtypen A,B und Monaden M[A], M[B], objektorientiert definiert in M[A].*

```
class M[A] {
  def filter(  f: A=>Boolean ): M[A]  // filtert Elemente
  def map(     f: A=>B       ): M[B]  // Berechnung
  def flatMap( f: A=>M[B]    ): M[B]  // Berechnung mit Schachtelung
}
```

Alle Funktionen haben als Ergebnistyp M[_].
Dadurch mögliche Komposition aller Funktionen.

```
firma.flatMap(
    standort => standort.filter(abt => abt.machtGewinn).flatMap(
    abt => abt.map(
    person => (person.vorname, person.nachname))))
```

`flatMap` kann andere Monaden-Aufrufe enthalten! (der Parameter `f` liefert `M[B]`)

For-comprehension = Schreibweise für monadische Operationen

- *to comprehend*: = *verstehen, umfassen*

  Herkunft: in der Mathematik heißt die Mengenschreibweise M = { x | P(x) } auch *set-comprehension*!

  In Programmiersprachen: list-comprehension oder for-comprehension.

- In Scala durch monadische Funktionen definiert (vgl. auch C#/LinQ)

```
// { x * x | x ∈ list}
for(x <- list) yield x * x
== list.map(x => x * x)

// { x | x ∈ list & x > 10}
for(x <- list if x > 10) yield x
== list.filter(x => x > 10)

// { x * y | x ∈ list1 & y ∈ list2}
for(x <- list1; y <- list2) yield x * y
== list1.flatMap(x => list2.map(y => x * y))
```

Das Beispiel der vorhergehenden Folie:

```
for { standort <- firma
      abt      <- standort if abt.machtGewinn
      person   <- abt
    } yield (person.vorname, person.nachname)
```

Und das prozedurale Foreach?

```
for(x <- list) println(x)
==
list.foreach(println(_))

for(i <- 1 to n) {
   val a = aFunction(i)
   aProcedure(a, i)
}
==
(1 to n).foreach {
   i => {
          val a = aFunction(i)
          aProcedure(a, i)
        }
}
```

Man kann Foreach auch mit anderen Operationen kombinieren:

```
for(x <- list if isPrime(x)) println(x)
==
list.filter(isPrime).foreach(println)
```

Option[A] und Monaden

Motivation: Java-Fehlerbehandlung ist entweder prozedural (Exception) oder umständlich (null)

```
List<String> ansprechbar …
Map<String, Ort> wohnort …
List<String> partner = new LinkedList<>();
for (String p : ansprechbar) {
    Ort ort = wohnort.get(p);
    if (ort != null && ort.einwohner > 1000000) partner.add(p);
}
```

Problem:
- Es ist nicht einfach erkennbar, welche Methode **`null`** liefert.
- Wenn ja, muss nach jedem dieser Aufrufe eine Abfrage (und Fehlerbehandlung) stehen
- Ähnliches gilt, für (checked) Exceptions

Scala-Ansatz: „Container"-Typ **`Option[A]`**, mit den Werten **`Some[A]`** und **`None`**.

```
def mittelwert(s: Seq[Double]): Option[Double] =
    if (s.isEmpty)
        None
    else
        Some(s.sum / s.length)
```

Option-Methoden: `get`, `getOrElse`, `isEmpty`, …, Pattern-Matching

Frage:
Wie verknüpft man mehrere Option-Funktionen?

Option[A] als Monade

Die List[A]-Monade steht für 0-N Elemente, die Option[A]-Monade steht für 0-1 Elemente.

Beispiel:

In Schnittstelle `Map[A,B]`:

```
def get(key: A): Option[B]     // Some(value) oder None

val wohnort = Map("Hans" -> bonn, ...)
val ansprechbar = List("Karin", "Hans")

val partner = for {
    person   <- ansprechbar
    ort      <- wohnort.get(person) if ort.einwohner > 1000000
} yield person

for(f <- partner) printf("%s in %s%n", f, wohnort(f).name)
```

Die Operationen `map`, `flatMap`, und `filter` verknüpfen Funktionen, ohne dass man abfragen muss, ob ein Wert vorhanden ist.

`Option` verfügt über alle weiteren Collection-Methoden.

Es gibt eine unendliche Anwendungsvielfalt für das Monaden-Muster:

Datenbank-Abfragen, Fehlermeldungen, Parser-Combinators, funktionale IO, Zustandskapselung, …

Monaden bilden in .NET den Kern der Bibliothek LINQ !

Implementierung von Option als algebraischer Datentyp

```
sealed abstract class Option[A] {
  ...
}

case class Some[+A](x: A) extends Option[A]  {
    def isEmpty = false
    def get = x
    def flatMap[B](f: A => Option[B]): Option[B] = f(x)
    def map[B](f: A => B): Option[B] = Some(f(x))
    def filter(f: A => Boolean): Option[A] = if(f(x)) Some(x) else None
}

case object None extends Option[Nothing]  {
    def isEmpty = true
    def get = throw new UnsupportedOperationException
    def flatMap[B](f: A => Option[B]) = None
    def map[B](f: A => B) = None
    def filter(f: A => Boolean) = None
}
```

Future als Monade

Man kann Futures – als Monaden – direkt zu weiteren Futures kombinieren.

```
import scala.concurrent._
import ExecutionContext.implicits.global

// nebenläufige Ausführung nur, wenn Futures vor for gestartet wurden

val berechnung1 = future {
      …  // 1. nebenläufige Berechnung
    }
val berechnung2 = future {
      …  // 2. nebenläufige Berechnung
    }
val z: Future[Int] = for { // = flatMap und map
        x <- berechnung1
        Y <- berechnung2
    } yield x + y   // Verknüpfung der Ergebnisse

    ...

z onSuccess { // Callback für das Ergebnis
    case ergebnis => println(ergebnis)
}
```

z ist ein Future-Objekt, das die Summe von x und y verfügbar macht

Zusammenfassung der Monaden-Verknüpfung

```
map:      M[A] => ( A => B       ) => M[B]
flatMap:  M[A] => ( A => M[B]    ) => M[B]
filter:   M[A] => ( A => Boolean ) => M[A]
```

Monadische Funktionen wenden auf eine Monade eine Funktion an mit dem Ergebnis: neue Monade.
Sie können damit problemlos hintereinander ausgeführt werden:

```
liste.filter(x => x % 2 == 0).map(x => x * x)
```

Dagegen geht die geschachtelte Iteration nur mit `flatMap`:

```
for (x ← L1; y ← L2) yield f(x,y)

L1.flatMap(x =>   L2.map (y => f(x,y) ) )
                  M[C]=> (C =>    B) => M[B]
M[A] =>    A =>                       M[B] => M[B]
```

Die geschachtelte Funktion erhält `y` durch `map` und `x` als freien Parameter.
`map` als Funktion von `x` ist eine Funktion `A => M[B]`.
(Anstelle von `map` kann hier jede andere monadische Funktion stehen!)

Unveränderliche Datenstrukturen: unnötige Veränderlichkeit ist immer schlecht

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

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

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

Fazit: *Brüche stellen unveränderliche Werte dar!*

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

Unveränderliche Datenstrukturen (Vor- und Nachteile)

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

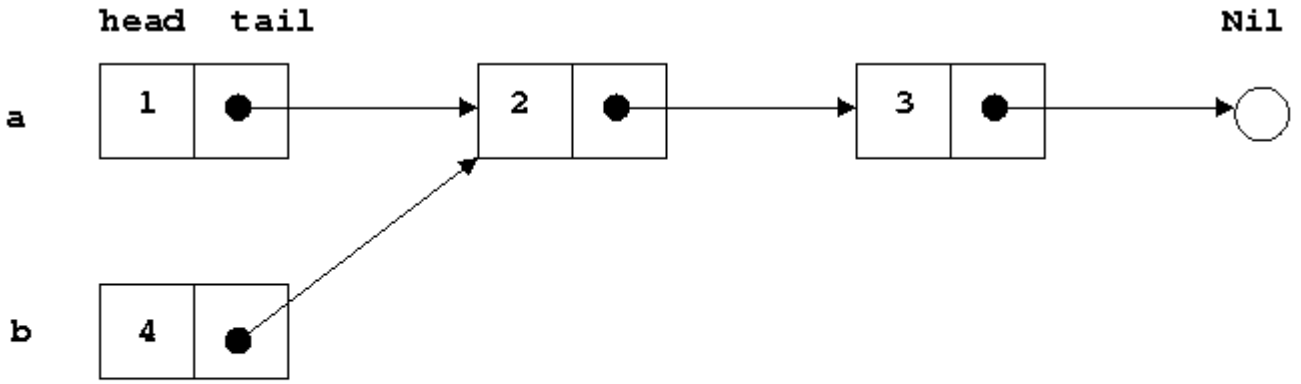
Unveränderlichkeit hat Implementierungs-**Nachteile**:

- Es entstehen und vergehen immer wieder neue kurzlebige Objekte (garbage collection!)
- Viele Operationen erfordern häufige 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.
- Die Daten können bei Bedarf evaluiert (lazy) werden

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

Beachten Sie: `Nil` ist nicht **`null`**, sondern ein singuläres Objekt! (Algebraische Datenstruktur!)

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.

s. Prolog Praktikum (t2l – Baum nach Liste)

```
append(LeftList, [X|RightList], List)
```

Beispiel-Implementierung von Listenklassen (vereinfacht)

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

```
sealed abstract class List[+T] {
// abstrakte Methoden:
    def isEmpty: Boolean
    def head: T
    def tail: List[T]
// konkrete Methoden: ...
}

case object Nil extends List[Nothing] {
    override def isEmpty = true
    override def head = throw new NoSuchElementException
    override def tail = throw new NoSuchElementException
}

final case class Cons[T] (head: T, tail: List[T]) extends List[T] {
    override def isEmpty = false
}
```

Anmerkung:

In Scala heißt die Cons-Klasse `::`. Dies ermöglicht die Schreibweise `head::tail`.

Operationen, die sich besser prozedural lösen lassen, werden an die Klasse `ListBuffer` delegiert.
(vgl. Java: `String` und `StringBuilder`/`StringBuffer`)

Nicht evaluierte Datenstrukturen (Streams) – Anwendung

Streams sind Behälter, die nur soweit wie nötig konstruiert wurden. Bei vielen Algorithmen werden so ganz einfach unnötige Operationen vermieden.

Man kann damit sogar unendliche Datenstrukturen definieren!

`#::` ist der Cons-Operator für Streams

```
val somePrimes = (2 to 1000).toStream.filter(isPrime)
println(somePrimes)

val allPrimes = Stream.from(2).filter(isPrime)    // das sind alle Primzahlen
val b = allPrimes.takeWhile(_<100).toList         // die Liste bis 100

val fib: BigInt = 0 #:: 1 #::
       ( fib.zip(fib.tail) map { case (x,y) => x + y } )
println(fib(10000))

val parallelPrimes = allPrimes.par
```

Vorteile:

- Rekursiv definierte Datenstrukturen
- Es wird nur das berechnet, was notwendig ist
- Flexibilität (z.B. parallele Berechnung)

Streams stellen die Neuerung in Java 8 dar!

Implementierungsidee von Streams

Es geht hier um das Prinzip.
Dafür wurden Scala-Eigenheiten weggelassen..

```
trait Stream[+A] {
// abstrakte Methode:
    def cell: Option((A, Stream[A]))
// konkrete Methoden: ...
    def isEmpty: Boolean = cell.isEmpty
    def take(n: Int): Stream[A] = cell match {
      case Some(h, t) if n > 0 => Stream.cons(h, t.take(n-1))
      case None => Stream.empty
    }

}

object Stream  {
    def empty[A]: Stream[A] = new Stream{
        def cell = None
    }

    def cons[A](hd: A, tl: =>Stream[A]): Stream[A] = new Stream[A]{
        lazy val cell = Some((hd, tl))
    }
}
```

Anmerkung: `head` und `tail` gehören natürlich auch dazu.