HANSER

Leseprobe

Oliver Braun

Scala

Objektfunktionale Programmierung

ISBN: 978-3-446-42399-2

Weitere Informationen oder Bestellungen unter

http://www.hanser.de/978-3-446-42399-2

sowie im Buchhandel.

Kapitel 5

Funktionales Programmieren

Scala verbindet als Hybridsprache die objektorientierte Programmierung mit der funktionalen Programmierung[1]. Unter objektorientierter Programmierung können sich die meisten etwas Konkretes vorstellen. Was aber ist eine funktionale Programmiersprache? Ob eine Programmiersprache das Label „funktional" führen darf, wurde und wird viel diskutiert. In jüngster Zeit entbrannte auch eine Diskussion, ob Scala sich so nennen darf. Martin Odersky, der Schöpfer von Scala, nennt Scala stattdessen in dem Blog-Beitrag [Ode10a] aus dem Januar 2010 eine *postfunktionale* Sprache.

Allein schon für die Frage, was eine funktionale Programmiersprache überhaupt ist, gibt es verschiedene Antworten. Einigkeit herrscht über den Kern der funktionalen Programmierung: die Berechnung durch Auswertung mathematischer Funktionen und die Vermeidung von Zustand und veränderbaren Daten. Das heißt, wenn wir in Scala `val` statt `var` nutzen, sind wir schon ein bisschen funktionaler.

Mit funktionaler Programmierung gehen eine Vielzahl von Konzepten einher, deren Umsetzung in Scala Gegenstand dieses Kapitels sind. Im ersten Abschnitt stellen wir Ihnen die Bedarfsauswertung vor, mit der die Berechnung eines `val`s erst dann durchgeführt wird, wenn das erste Mal auf ihn zugegriffen wird. In Abschnitt 5.2 werden Funktionen und Rekursionen besprochen. Ein wesentliches Merkmal funktionaler Programmierung sind die Funktionen höherer Ordnung (siehe Abschnitt 5.3) , die eine Funktion als Argument oder Ergebnis haben können.

Viele funktionale Programmiersprachen unterstützen Pattern Matching, eine Verallgemeinerung des `switch-case`-Konstrukts. Pattern Matching und die sogenannten Case-Klassen führen wir in Abschnitt 5.4 ein. Das auf den amerikanischen Mathematiker und Logiker Haskell B. Curry zurückgehende Konzept

[1] siehe auch [Bir98], [Hug89], [PH06], [OSG08] und [Oka99]

der Currysierung ermöglicht es, in Scala eigene Kontrollstrukturen zu definieren (siehe Abschnitt 5.5). Anschließend werden wir in Abschnitt 5.6 noch einmal zum Schlüsselwort `for` zurückkommen, um Ihnen zu zeigen, dass mehr als eine Schleife dahintersteckt. Zu guter Letzt werfen wir in Abschnitt 5.7 noch einen Blick auf das sehr ausgefeilte Typsystem von Scala.

5.1 Lazy Evaluation

Standardmäßig werden in Scala alle Felder einer Klasse beim Erzeugen eines Objektes berechnet. Wir sprechen dabei von der sogenannten *eager evaluation*. In einigen Fällen ist es aber nicht unbedingt sinnvoll, aufwendige Berechnungen, deren Ergebnis vielleicht sogar während der gesamten Existenz eines Objektes nicht benötigt werden, sofort durchzuführen. Der Schlüssel dazu ist die *lazy evaluation*, die auch mit *Bedarfsauswertung* übersetzt werden kann. Damit wird der Wert eines Feldes genau dann berechnet, wenn zum ersten Mal darauf zugegriffen wird. Im Gegensatz zu einer Methode wird dieser Wert dann aber in ausgewerteter Form gespeichert. Das heißt, bei einem erneuten Zugriff auf das Feld muss nicht erneut berechnet werden.

Scala legt dafür den Modifier `lazy` fest, der sinnvollerweise nur für `val`s zulässig ist. Solche lazy `val`s sind überall erlaubt und nicht nur als Felder von Objekten. Mit der folgenden Sitzung wollen wir das Verhalten von lazy `val`s demonstrieren:

```
scala> val x = {
     |   print("Geben Sie eine Zahl ein: ")
     |   readInt
     | }
Geben Sie eine Zahl ein: 12
x: Int = 12

scala> lazy val y = {
     |   print("Geben Sie eine Zahl ein: ")
     |   readInt
     | }
y: Int = <lazy>

scala> print(y)
Geben Sie eine Zahl ein: 13
13

scala> print(x)
12

scala> print(y)
13
```

In Listing 4.43 auf Seite 86 haben wir den Trait `PositiveNumber` definiert. Um Ihnen das Zurückblättern zu ersparen, wollen wir ihn hier noch einmal angeben:

```
trait PositiveNumber {
  val value: Int
  require(value>0)
}
```

Beim Erzeugen einer anonymen Klasse stellten wir fest, dass ein `val` in einer Subklasse erst nach dem Aufruf des Konstruktors der Basisklasse initialisiert wird. Gelöst hatten wir das Problem mit den vordefinierten Feldern. Eine zweite Möglichkeit, dies zu lösen, sind lazy `val`s. Allerdings können nur konkrete `val`s lazy sein. Daher ist es nicht möglich, den `val value` mit dem Modifier `lazy` zu versehen.

Listing 5.1: Der Trait `PositiveNumber` mit einer lazy `val`

```
trait PositiveNumber {
  val inValue: Int
  lazy val value = {
    require(inValue>0)
    inValue
  }
}
```

Die in Listing 5.1 zugegebenermaßen etwas umständliche Version des `PositiveNumber`-Traits lässt nun eine Definition der folgenden Form zu:

```
scala> val n = new PositiveNumber {
     |   val inValue = 12
     | }
n: java.lang.Object with PositiveNumber =
   $anon$1@409a44d6
```

Zu beachten ist hier allerdings, dass ein Objekt mit dem Wert `0` für den `inValue` nun erfolgreich erzeugt werden kann. Die durch `require` ausgelöste Exception wird dann erst beim ersten Zugriff auf `value` geworfen. Die folgende Beispielsitzung soll dies verdeutlichen:

```
scala> val m = new PositiveNumber {
     |   val inValue = 0
     | }
m: java.lang.Object with PositiveNumber =
   $anon$1@2754de0b

scala> m.value
java.lang.IllegalArgumentException: requirement failed
        at scala.Predef$.require(Predef.scala:134)
        ...
```

Damit ein lazy `val x` Sinn macht, müssen auch alle anderen `val`s, die in ihrer Berechnung auf `x` zugreifen, lazy sein. Ansonsten würde der Wert von `x` sofort benötigt und damit berechnet. Nachdem `var`s nicht lazy sein können, sollten diese den Wert von `x` nicht für ihre Berechnungen benötigen oder `x` sollte nicht lazy sein.

5.2 Funktionen und Rekursionen

Ein wesentliches Merkmal funktionaler Programmierung ist die Unterstützung von Funktionen als *first class values*. Das heißt, Funktionen sind Werte und können als solche genau wie Zahlen oder Zeichenketten zum Beispiel in Listen gespeichert werden oder sogar Parameter oder Ergebnisse von anderen Funktionen sein.

Funktionen können nicht nur wie bisher besprochen als Funktionsgleichung, sondern auch als *Funktionsliteral* definiert werden. Beispielsweise kann die Funktion

```
def inc(i: Int) = i + 1
```

auch als Funktionsliteral

```
(i: Int) => i + 1
```

geschrieben werden.[2] Nachdem durch `(i: Int) => i + 1` ein Wert definiert wird, kann dieser einer Variablen zugewiesen werden. Wir können also schreiben

```
val inc = (i: Int) => i + 1
```

Weil sich damit hinter `inc` eine Funktion verbirgt, können wir `inc` auf eine Zahl anwenden:

```
scala> val inc = (i: Int) => i + 1
inc: (Int) => Int = <function1>

scala> inc(2)
res0: Int = 3
```

Wir sehen auch in obiger Sitzung den Typ der Funktion `(Int) => Int`, also eine Funktion mit einer Parameterliste, die einen `Int` enthält, und dem Ergebnistyp `Int`. Für Funktionen mit 0 bis 22 Parametern sind in der Scala-Distribution Traits definiert. Die Ausgabe `<function1>` hinter dem Typ ist das Ergebnis der `toString`-Methode des Traits `Function1`. Das heißt auch insbesondere, dass jede Funktion in Scala ein Objekt ist.

Es ist auch möglich, ein Funktionsliteral zu definieren, das freie Variablen hat, z.B.

```
val adder = (i: Int) => i + a
```

[2] Der Ansatz kommt aus dem Lambda-Kalkül, in dem diese Funktion als $\lambda x.x + 1$ geschrieben wird. Siehe auch [Pie02].

Die Variable `i` ist ein *formaler Parameter*, aber die Variable `a` ist eine Referenz auf eine Variable im umschließenden Gültigkeitsbereich. Daher muss eine Variable mit dem Bezeichner `a` bei der Definition von `adder` sichtbar sein. Der Compiler erzeugt für `adder` dann eine sogenannte *Closure*, die die Variable `a` *captured*. Nachdem sich in Scala die Werte von Variablen ändern können, wird nicht der aktuelle Wert von `a`, sondern eine Referenz auf die Variable in die Closure eingebunden. Damit wirken sich Veränderungen von `a` nach Definition von `adder` unmittelbar aus, wie die folgende Sitzung veranschaulichen soll:

```
scala> var a = 10
a: Int = 10

scala> val adder = (i: Int) => i + a
adder: (Int) => Int = <function1>

scala> adder(2)
res0: Int = 12

scala> a = 20
a: Int = 20

scala> adder(2)
res1: Int = 22
```

Die Variable `a` hat zunächst den Wert `10`. Folglich wird die Anwendung der Funktion `adder` auf den Wert `2` zu `12` evaluiert. Wird anschließend `a` auf `20` gesetzt, so ändert sich auch das Ergebnis von `adder(2)` entsprechend.

Da es nach dem Konzept der funktionalen Programmierung keine veränderlichen Variablen gibt, also nur `vals`, ist es, wenn wir rein funktional programmieren, nicht möglich, durch Schleifen Werte zu berechnen. Beispielsweise ist der Code in Listing 5.2 zur Berechnung der Summe der in der Liste `list` enthaltenen Zahlen imperativ, da er die Berechnung durch Änderung des Zustands, hier dem Wert der Variablen `sum`, durchführt.

Listing 5.2: Imperative Berechnung der Summe der Zahlen einer Liste

```
def sum(list: List[Int]) = {
  var sum = 0
  for (elem <- list) sum += elem
  sum
}
```

In der funktionalen Programmierung wird eine solche Programmieraufgabe durch *Rekursion* gelöst.[3] Eine funktionale Version von `sum` ist in Listing 5.3 dargestellt. Nachdem die Funktion einen rekursiven Aufruf enthält, kann der Ergebnistyp nicht ermittelt werden und muss daher angegeben werden.

[3] Oder besser noch durch Funktionen höherer Ordnung, die wir Ihnen in Abschnitt 5.3 vorstellen werden.

Listing 5.3: Rekursive Berechnung der Summe der Zahlen einer Liste

```
def sum(list: List[Int]): Int = {
  if (list.isEmpty) 0
  else list.head + sum(list.tail)
}
```

Rekursion bedeutet, dass die Funktion sich selbst wieder aufruft. Die Funktion `sum` in Listing 5.3 addiert das erste Element zur Summe der Restliste. Die Methode `isEmtpy` gibt `true` zurück, wenn die Liste leer ist, `head` gibt das erste Element und `tail` die Restliste zurück. Ist die Liste leer, ist das Ergebnis 0. Aufgrund der Kürze und Klarheit und der Vermeidung von Zustandsänderungen ist die rekursive Version von `sum` vorzuziehen. Aber Rekursion ist leider in der Regel weniger effizient als ein imperativer Ansatz. Dies ist auch einer der Gründe, warum sich die Sprachdesigner von Scala dazu entschieden haben, imperatives Programmieren zuzulassen: Weil es zu effizienteren Lösungen führen kann.

Dieser Nachteil des rekursiven Ansatzes lässt sich in einem bestimmten Fall vom Compiler weg optimieren. Und zwar dann, wenn die Funktion *endrekursiv* (*tail recursive*) ist, d.h. wenn das Letzte, was in der Funktion berechnet wird, der rekursive Aufruf selbst ist. Der Compiler kann in diesem Fall die Rekursion durch eine Schleife ersetzen. Die `sum`-Funktion in Listing 5.3 lässt sich nicht optimieren, da das Ergebnis des rekursiven Aufrufs zum Schluss noch zum ersten Element der Liste addiert wird. Listing 5.4 zeigt eine Funktion `sum`, die eine endrekursive Hilfsfunktion nutzt. Diese kann durch den Compiler optimiert werden. Der Hilfsfunktion wird der bisher berechnete Wert und die Restliste übergeben, sodass die Addition vor dem rekursiven Aufruf berechnet wird und somit der rekursive Aufruf die letzte Berechnung von `sumHelper` ist.

Listing 5.4: Endrekursive Berechnung der Summe der Zahlen einer Liste

```
def sum(list: List[Int]) = {
  def sumHelper(x: Int, list: List[Int]): Int = {
    if (list.isEmpty) x
    else sumHelper(x+list.head, list.tail)
  }
  sumHelper(0,list)
}
```

Vor dem Release von Scala 2.8.0 war die gängige Möglichkeit zu testen, ob die Endrekursion durch eine Schleife optimiert wurde, im Abbruchfall der Rekursion eine Exception zu werfen und dann im Stacktrace zu überprüfen, ob die Funktion sich selbst aufgerufen hat oder nicht.

Listing 5.5: Rekursive Berechnung der Summe der Zahlen einer Liste mit einer `@tailrec`-Annotation

```
import scala.annotation.tailrec
// wird nicht kompiliert, weil nicht endrekursiv
@tailrec def sum(list: List[Int]): Int = {
  if (list.isEmpty) 0
  else list.head + sum(list.tail)
}
```

Seit Scala 2.8.0 gibt es dafür eine Annotation: `@tailrec`. Versehen wir beispielsweise, wie in Listing 5.5 zu sehen, die nicht endrekursive Funktion `sum` aus Listing 5.3 mit der `@tailrec`-Annotation, so führt der Übersetzungsversuch zu folgendem Fehler

```
<console>:6: error: could not optimize @tailrec annotated
    method: it contains a recursive call not in tail
   position
       @tailrec def sum(list: List[Int]): Int = {
                    ^
```

Andererseits wird die Funktion `sum` aus Listing 5.4 mit einem `@tailrec` vor der `sumHelper`-Funktion ohne Fehlermeldung kompiliert, das heißt also, `sumHelper` konnte optimiert werden.

Wie die folgende Sitzung zeigt, kann eine endrekursive Funktion nicht immer optimiert werden:

```
scala> import scala.annotation.tailrec
import scala.annotation.tailrec

scala> class Tailrec {
     |   @tailrec def g(i: Int):Int = if (i==0) 0
     |                                 else g(i-1)
     | }
<console>:7: error: could not optimize @tailrec annotated
    method: it is neither private nor final so can be
   overridden
         @tailrec def g(i: Int):Int = if (i==0) 0
```

Wie aus der Fehlermeldung ersichtlich ist, wird eine Methode, die in einer Subklasse redefiniert werden kann, nicht optimiert.

Der Kern des Problems rekursiver Funktionen ist, das der Stack mit jedem Funktionsaufruf wächst, was letztendlich je nach Speichergröße und Anzahl der Rekursionschritte zu einem Überlauf und damit einem Absturz des Programmes führen kann. Ein Mittel, dies zu umgehen, sind sogenannte *Trampolines*[4]. Dabei werden die Funktionen so implementiert, dass sie statt eines Wertes eine Funktion zurückgeben, mit der der nächste Schritt berechnet werden kann. Das heißt, der rekursive Aufruf findet nicht in der Funktion selbst statt, sondern die Trampoline-

[4] Die Funktionen prallen wie von einem Trampolin zurück – daher der Name.

Funktion führt einen Schritt nach dem anderen aus. Mit Scala 2.8.0 werden solche Trampoline-Funktionen durch das Objekt `scala.util.control.TailCalls` unmittelbar unterstützt.

Interessant ist Trampolining insbesondere bei wechselseitiger Rekursion, also wenn zwei oder mehr Funktionen sich gegenseitig aufrufen, was nicht automatisch optimiert werden kann. Der Klassiker sind die beiden Funktion `isEven` und `isOdd`, die berechnen, ob eine Zahl gerade bzw. ungerade ist und die (wie aus Listing 5.6 ersichtlich) wechselseitig rekursiv definiert sind[5].

Listing 5.6: Wechselseitig-rekursive Funktionen `isEven` und `isOdd`

```
def isEven(n: Int): Boolean =
  if (n==0) true else isOdd(n-1)
def isOdd(n: Int): Boolean =
  if (n==0) false else isEven(n-1)
```

Die beiden Funktionen können nicht optimiert werden und führten auf einem Testrechner bei `isEven(100000)` zu einem `StackOverflowError`. Durch den Trampoline-Ansatz, der in Listing 5.7 dargestellt ist, wird zwar immer noch für jede Zahl zwischen der eingegebenen und `0` die Funktion `isEven` oder `isOdd` aufgerufen, aber jetzt hintereinander.

Listing 5.7: Wechselseitig-rekursive Funktionen `isEven` und `isOdd` unter Verwendung des Tramponline-Ansatzes

```
import scala.util.control.TailCalls._
def isEven(n: Int): TailRec[Boolean] =
  if (n==0) done(true) else tailcall(isOdd(n-1))
def isOdd(n: Int): TailRec[Boolean] =
  if (n==0) done(false) else tailcall(isEven(n-1))
```

Statt eines `Boolean`s geben diese beiden Funktionen eine `TailRec[Boolean]` zurück. Um die eingebettete Rekursion zu beenden, dient die Funktion `done`, für den Rekursionsschritt die Funktion `tailcall`. Der Ausdruck `isEven(100000)` gibt damit eine Funktion zurück, die mit `result` ausgewertet werden kann. Der Ausdruck `isEven(100000).result` berechnet ohne Stack Overflow den Wert `true`.

5.3 Higher-Order-Functions

Funktionen, die Funktionen als Parameter haben oder als Ergebnis zurückgeben, werden *Funktionen höherer Ordnung* genannt. Diese sind eine unmittelbare Folge daraus, dass Funktionen *first class values* sind, also gleichberechtigt neben anderen

[5] Durch die wechselseitige Abhängigkeit lassen sich die beiden Funktionen nicht nacheinander in die Scala-Shell eingeben.

Werten stehen. Funktionen höherer Ordnung sind ein sehr praktisches Mittel zur Abstraktion, wie wir Ihnen im folgenden Beispiel zeigen wollen.

Betrachten wir eine Funktion `incList`, die alle Zahlen in einer Liste inkrementiert. Wir könnten diese Funktion rekursiv definieren, beispielsweise wie in Listing 5.8 angegeben.

Listing 5.8: Die Funktion `incList` zum rekursiven Inkrementieren aller Elemente einer Liste

```
def incList(list: List[Int]): List[Int] = {
  if (list.isEmpty) List()
  else list.head+1 :: incList(list.tail)
}
```

Weiter könnten wir vielleicht auch eine Funktion benötigen, die alle Elemente einer Liste verdoppelt. Die entsprechende Funktion ist in Listing 5.9 dargestellt.

Listing 5.9: Die Funktion `doubleList` zum rekursiven Verdoppeln aller Elemente einer Liste

```
def doubleList(list: List[Int]): List[Int] = {
  if (list.isEmpty) List()
  else list.head*2 :: doubleList(list.tail)
}
```

Die beiden Funktionen `incList` und `doubleList` unterscheiden sich nur sehr wenig. Wo bei `incList` der Ausdruck `list.head+1` steht, heißt es bei `doubleList` stattdessen `list.head*2`, und natürlich steht beim rekursiven Aufruf einmal `incList` und einmal `doubleList`.

Definieren wir zwei Funktionen:

```
def inc(x: Int) = x + 1
def double(x: Int) = x * 2
```

und nutzen diese, wie in Listing 5.10 dargestellt, sehen die Funktionen sich noch ein bisschen ähnlicher.

Listing 5.10: Die Funktionen `incList` und `doubleList` mit Nutzung der Funktionen `inc` und `double`

```
def incList(list: List[Int]): List[Int] = {
  if (list.isEmpty) List()
  else inc(list.head) :: incList(list.tail)
}
def doubleList(list: List[Int]): List[Int] = {
  if (list.isEmpty) List()
  else double(list.head) :: doubleList(list.tail)
}
```

Mit einer Funktion höherer Ordnung könnten wir die Funktion `inc` bzw. `double` als Argument übergeben. Das heißt, statt der beiden nahezu identischen Funktionen `incList` und `doubleList` können wir die Funktion `funList` (siehe Listing 5.11) definieren, die als Parameter eine Funktion erwartet, mit der die Listenelemente verändert werden sollen.

Listing 5.11: Die Funktionen `funList` mit einer Funktion zur Veränderung der einzelnen Elemente als Parameter

```
def funList(fun: Int => Int ,list: List[Int])
    : List[Int] = {
  if (list.isEmpty) List()
  else fun(list.head) :: funList(fun, list.tail)
}
```

Mit der Funktion `funList` sowie `inc` und `double` sind wir in der Lage, die Funktionen `incList` und `doubleList` wie folgt zu definieren:

```
def incList   (list: List[Int]) = funList(inc   ,list)
def doubleList(list: List[Int]) = funList(double,list)
```

Tatsächlich wird dieser Ansatz, eine Funktion über eine Liste zu *mappen*, als Methode bereits zur Verfügung gestellt. Statt `funList` selbst zu definieren, können wir die Higher-Order-Function `map` nutzen:

```
def incList   (list: List[Int]) = list map inc
def doubleList(list: List[Int]) = list map double
```

Als nächsten Schritt können wir uns die Definition von `inc` und `double` sparen, indem wir stattdessen Funktionsliterale übergeben:

```
def incList   (list: List[Int]) = list map (i => i + 1)
def doubleList(list: List[Int]) = list map (i => i * 2)
```

Und zu guter Letzt können wir sogar noch die gebundene Variable `i` im Ausdruck rechts vom Pfeil durch einen Platzhalter ersetzen und schreiben:

```
def incList   (list: List[Int]) = list map (_ + 1)
def doubleList(list: List[Int]) = list map (_ * 2)
```

An die Stelle des Unterstrichs setzt der Compiler dann das jeweilige Element. Damit sparen wir uns die Einführung eines Namens für die Variable. Mit dem Unterstrich definieren wir eine sogenannte *partiell angewandte Funktion* (*partially applied function*). Dies funktioniert auch mit mehreren Variablen und mehreren Unterstrichen. Beispielsweise definieren wir durch

```
scala> val add = (_: Int)+(_: Int)
add: (Int, Int) => Int = <function2>
```

eine Funktion `add`, die eine Parameterliste mit zwei Argumenten erwartet. Der Typinferenzmechanismus kann für die beiden Argumente von `add` keinen Typ

ermitteln, sodass wir Typinformationen hinzufügen müssen. Gäben wir keinen Typ an, sähe das in der Scala-Shell folgendermaßen aus:

```
scala> val add = _+_
<console>:5: error: missing parameter type for expanded
   function ((x$1, x$2) => x$1.$plus(x$2))
       val add = _+_
                 ^
<console>:5: error: missing parameter type for expanded
   function ((x$1: <error>, x$2) => x$1.$plus(x$2))
       val add = _+_
                   ^
```

Die korrekt definierte Funktion `add` kann auf eine Argumentliste, bestehend aus zwei `Int`s, angewendet werden:

```
scala> add(2,3)
```

Was hier direkt wie ein Funktionsaufruf aussieht, entspricht eigentlich `add.apply(2,3)`, also das Objekt mit dem Namen `add` reagiert auf die Nachricht `apply` mit einer Argumentliste, die zwei `Int`s enthält.

In der funktionalen Programmierung dominiert die Liste als Datenstruktur. Für sie sind in der Regel eine Vielzahl von Funktionen höherer Ordnung definiert, um die Liste in eine neue zu transformieren. Das neue Collection-Framework (siehe Abschnitt 6.2) von Scala 2.8 definiert viele dieser Funktionen für alle Collections. Wir erläutern einige der Funktionen im Folgenden am Beispiel der Liste.

Neben dem bereits vorgestellten `map`, das alle Elemente einer Liste mit der übergebenen Funktion verändert und die neue Liste zurückgibt, gibt es auch die Funktion `foreach`, die eine Funktion vom Typ `A => Unit` als Argument hat und diese mit allen Elementen ausführt. Beispielsweise können mit

```
list.foreach(e => println(e))
```

alle Elemente einer Liste `list` zeilenweise ausgegeben werden. Auch hier kann die gebundene Variable `e` wieder durch den Unterstrich ersetzt werden:

```
list.foreach(println(_))
```

Statt `println(_)` kann auch `println _`[6] geschrieben werden. Damit steht der Platzhalter nicht mehr für den einen Parameter, sondern für die gesamte Parameterliste, die in diesem Fall einen Parameter enthält. Wird im Kontext eine Funktion erwartet, kann ein Platzhalter für die gesamte Parameterliste sogar weggelassen werden, wie beispielsweise bei

```
list.foreach(println)
```

[6] Es ist wichtig, zwischen `println` und dem Unterstrich ein Leerzeichen einzufügen. Zusammengeschrieben, also `println_`, ist es ein gültiger Bezeichner.

In Infix-Operatorschreibweise kann dann auch geschrieben werden:

```
list foreach println
```

Eine ganze Reihe von Higher-Order-Functions haben Prädikate, also Funktionen mit dem Ergebnistyp `Boolean`, als Parameter. Damit lässt sich beispielsweise ein Teil einer Liste ermitteln, für den ein Prädikat gilt oder nicht. Mit `list filter (_<3)` wird eine neue Liste berechnet, die alle Elemente enthält, die kleiner als 3 sind. Die Funktion `filterNot` invertiert das Prädikat. `dropWhile` entfernt die Elemente, solange eine Bedingung gilt, `takeWhile` nimmt nur diese, `span` teilt die Liste gemäß des Prädikats in zwei Teillisten. Für jedes Prädikat `p` und jede Liste `l` gilt:

$$\texttt{l span p} \equiv \texttt{(l takeWhile p, l dropWhile p)}$$

wie das folgende Listing verdeutlichen soll:

```
scala> val list = List(1,2,3,4,5)
list: List[Int] = List(1, 2, 3, 4, 5)

scala> list span (_%2==1)
res0: (List[Int], List[Int]) = (List(1),List(2, 3, 4, 5))

scala> list takeWhile (_%2==1)
res1: List[Int] = List(1)

scala> list dropWhile (_%2==1)
res2: List[Int] = List(2, 3, 4, 5)
```

Die Funktion `partition` teilt die Liste in zwei Listen, bei denen alle Elemente der ersten Liste das Prädikat erfüllen und alle der zweiten Liste nicht, z.B.

```
scala> list partition (_%2==1)
res3: (List[Int], List[Int]) = (List(1, 3, 5),List(2, 4))
```

Weitere nützliche Funktionen sind `forall` und `exists`, die ermitteln, ob ein Prädikat für alle bzw. für mindestens ein Element der Liste erfüllt ist, z.B.

```
scala> list forall (_<3)
res4: Boolean = false

scala> list exists (_<3)
res5: Boolean = true
```

Eine typische Anwendung für Listen ist es, die Elemente mithilfe einer Operation zu einem Wert zusammenzufassen, z.B. die Addition aller Elemente. Dieses Zusammenfassen wird *Falten* genannt, da die Liste zu einem Wert zusammengefaltet wird. Wie viele funktionale Programmiersprachen stellt auch Scala dafür Funktionen bereit, und zwar für das Falten einer Liste von links beginnend `foldLeft` und von rechts beginnend `foldRight`. Üblicherweise wird beim Fal-

ten keine Rücksicht genommen, ob es sich um eine leere oder eine nicht-leere Liste handelt. Daher wird als Parameter zusätzlich zur Operation ein Wert benötigt, der zum einen Ergebnis der Anwendung der Faltungsfunktion auf die leere Liste ist und zum anderen am Ende bzw. am Anfang der Faltung mithilfe der Operation mit dem Rest verknüpft wird.

Der Ausdruck

```
list.foldLeft(0)(_+_)
```

berechnet die Summe der Elemente der Liste `list` mit dem Startwert `0`, indem er sukzessive alle Elemente mit der Operation `+` verknüpft.

In Scala werden statt der Funktionen `foldLeft` und `foldRight` die Operatoren `/:` und `:\` verwendet. Die beiden Operatoren entsprechen den beiden Funktionen, wobei gemäß der Regel für Operatoren, die mit einem `:` enden, bei `/:` die Liste rechts und der Startwert links stehen. Damit sieht die typische Anwendung natürlicher und leichter verständlich aus, wie beispielsweise bei

```
(list :\ 0)(_+_)
```

oder

```
(0 /: list)(_+_)
```

Im ersten Fall wird die Zahl `0`, die rechts vom Operator steht, von rechts zum Rest addiert und im zweiten Fall entsprechend von links. Mit der folgenden Sitzung wird es noch etwas deutlicher:

```
scala> ("Anfang -> " /: list)(_+_)
res6: java.lang.String = Anfang -> 12345

scala> (list :\ " <- Ende")(_+_)
res7: java.lang.String = 12345 <- Ende
```

Die Verwendung von `foldLeft` bzw. `foldRight` führt zwar zum selben Ergebnis, der Ausdruck ist aber weniger eingängig:

```
scala> (list foldLeft "Anfang -> ")(_+_)
res8: java.lang.String = Anfang -> 12345

scala> (list foldRight " <- Ende")(_+_)
res9: java.lang.String = 12345 <- Ende
```

Für nicht-leere Listen gibt es noch die Varianten `reduceLeft` und `reduceRight` die keinen Startparameter benötigen. Die Funktionen `scanLeft` und `scanRight` berechnen die Liste aller Zwischenergebnisse, wie im folgenden Listing zu sehen:

```
scala> (list scanLeft "Anfang -> ")(_+_)
res10: List[java.lang.String] = List(Anfang -> , Anfang
    -> 1, Anfang -> 12, Anfang -> 123, Anfang -> 1234,
    Anfang -> 12345)
```

```
scala> (list scanRight " <- Ende")(_+_)
res11: List[java.lang.String] = List( <- Ende, 5 <- Ende,
    45 <- Ende, 345 <- Ende, 2345 <- Ende, 12345 <- Ende)
```

Auch wenn wir noch nicht alle standardmäßig verfügbaren Funktionen höherer Ordnung angesprochen haben, hören wir nun damit auf und verweisen für weitere Informationen auf das Studium des Scala-APIs.

Funktionen höherer Ordnung können Funktionen nicht nur als Parameter haben, sondern auch als Ergebnis zurückgeben. Beispielsweise können wir mit

```
def mkAdder(x: Int) = (y: Int) => x + y
```

eine Funktion `mkAdder` definieren, die angewendet auf einen Parameter `x` eine Funktion zurückgibt, die `x` zum jeweiligen Parameter addiert, zum Beispiel:

```
scala> val plus2 = mkAdder(2)
plus2: (Int) => Int = <function1>

scala> plus2(5)
res12: Int = 7
```

Um Ihnen zum Abschluss der Abschnitts noch einen Eindruck zu geben, wie mächtig das Programmieren mit Funktionen höherer Ordnung sein kann, betrachten Sie den folgenden Ausdruck:

```
(1 to 10).toList map (x => x+(_: Int)) map (_(3))
```

Der erste Teil, `(1 to 10).toList map (x => x+(_:Int))`, erzeugt aus der Liste der Zahlen 1 bis 10 Funktionen entsprechend der Funktion `mkAdder` (siehe oben). Der Teil `map (_(3))` wendet dann jede Funktion aus der Liste auf die Zahl 3 an, d.h. das Ergebnis des Ausdrucks lautet `List(4, 5, 6, 7, 8, 9, 10, 11, 12, 13)`. Beim Teilausdruck `_(3)` ist der Platzhalter für die Funktion da. Das Argument, die Zahl 3, steht bereits in Klammern.

5.4 Case-Klassen und Pattern Matching

Das *Pattern Matching*, der Vergleich mit einem Muster, ist eine Art verallgemeinertes `switch-case`, wie es aus den C-ähnlichen Sprachen bekannt ist. Wie mit `switch` und `case` kann eine Variable auf verschiedene Werte überprüft werden, z.B.

```
number match {
  case 1 => println("eine Eins")
  case 2 => println("eine Zwei")
  case _ => println("etwas anderes")
}
```

Hier wird der Wert der Variablen `number` zuerst mit dem Wert `1` verglichen. Wenn der Vergleich erfolgreich ist, wird der Code nach dem Doppelpfeil `=>` ausgeführt. Damit wird beim Wert `1` `eine Eins` ausgegeben. Anschließend wird der gesamte Block beendet. Ist `number` 2, passt der Wert nicht auf das Muster 1. Daher wird als Nächstes geprüft, ob er auf das Muster 2 passt. Nachdem dies der Fall ist, wird `eine Zwei` ausgegeben. Passt ein anderer Wert weder auf das Muster 1 noch auf 2, wird mit dem dritten Muster, dem Unterstrich, gematcht. Der Unterstrich ist das *Wildcard-Pattern*, auf das alles passt.

Scala überprüft beim Kompilieren, ob es Fälle gibt, die nicht erreicht werden können. Setzen wir das Wildcard-Pattern an den Anfang, so würde das bedeuten, dass bei jedem beliebigen Wert von `number` der erste Fall passt und damit immer `etwas anderes` ausgegeben wird. Allerdings schlägt schon der Versuch fehl, diesen Ausdruck zu übersetzen, wie die folgende Sitzung zeigt:

```
scala> number match {
     |   case _ => println("etwas anderes")
     |   case 1 => println("eine Eins")
     |   case 2 => println("eine Zwei")
     | }
<console>:9: error: unreachable code
         case 1 => println("eine Eins")
                           ^
<console>:10: error: unreachable code
         case 2 => println("eine Zwei")
                           ^
```

Wollen wir eine Funktion definieren, die mittels Pattern Matching ihre Argumente unterscheidet, so können wir das machen wie in Listing 5.12.

Listing 5.12: Definition der Funktion `checkNumber` mit Pattern Matching

```
def checkNumber(number: Int) = number match {
  case 1 => println("eine Eins")
  case 2 => println("eine Zwei")
  case _ => println("etwas anderes")
}
```

Es ist natürlich auch möglich, den Ausdruck als Funktionsliteral anzugeben (siehe Listing 5.13).

Listing 5.13: `checkNumber` mit Pattern Matching als Funktionsliteral

```
val checkNumber = (number: Int) => number match {
  case 1 => println("eine Eins")
  case 2 => println("eine Zwei")
  case _ => println("etwas anderes")
}
```

Scala erlaubt hier eine noch etwas prägnantere Schreibweise, entsprechend den partiell angewandten Funktionen (siehe Seite 110), bei der wir uns die Benennung der Variablen `number` sparen können. Die kürzere Form ist in Listing 5.14 dargestellt.

Listing 5.14: `checkNumber` mit Pattern Matching als Funktionsliteral ohne explizite Parameterdarstellung

```
val checkNumber: Int => Unit = {
  case 1 => println("eine Eins")
  case 2 => println("eine Zwei")
  case _ => println("etwas anderes")
}
```

Wollen wir im letzten Fall auch die übergebene Zahl ausgeben, können wir auch auf eine Variable matchen. Auf eine Variable kann wie auf den Unterstrich alles gematcht werden. In Listing 5.15 wird im dritten Fall auf die Variable `number` gematcht. Diese damit neu eingeführte Variable kann dann auf der rechten Seite des Doppelpfeils genutzt werden.

Listing 5.15: `checkNumber` mit Pattern Matching auf eine Variable

```
val checkNumber: Int => Unit = {
  case 1      => println("eine Eins")
  case 2      => println("eine Zwei")
  case number => println("etwas anderes: "+number)
}
```

Listing 5.16 zeigt die Möglichkeit, über Pattern Matching den Typ eines Wertes zu ermitteln. Die Funktion `checkValue` hat einen Parameter vom Typ `Any`. Damit kann `checkValue` beispielsweise auf einen `Int`, einen `Double`, einen `String` oder irgendetwas anderes angewendet werden. Das Pattern `i: Int` passt dann für einen beliebigen Wert vom Typ `Int`.

Listing 5.16: `checkValue` mit Pattern Matching auf Typen

```
val checkValue: Any => Unit = {
  case i: Int    => println("ein Int: "+i)
  case d: Double => println("ein Double: "+d)
  case _         => println("etwas anderes")
}
```

Selbstverständlich können wir auch die verschiedenen Muster gemischt in einer Funktion nutzen, wie in Listing 5.17 dargestellt ist.

Listing 5.17: `checkValue` mit Pattern Matching auf einen Wert und auf Typen

```
val checkValue: Any => Unit = {
  case 1         => println("eine Eins")
  case _: Int    => println("irgendein Int (keine Eins)")
  case d: Double => println("irgendein Double: "+d)
  case any       => println("etwas anderes: "+any)
}
```

Die bisher verwendete Darstellung von Sequenzen wie z.B. Listen als `List(1, 2, 3)` oder `1 :: 2 :: Nil`[7] steht auch als Pattern zur Verfügung. Damit kann aber nicht nur auf die gesamte Liste, sondern auch auf Listen mit bestimmten Werten gematcht werden (siehe Listing 5.18).

Listing 5.18: `checkValue` mit Pattern Matching auf einen Wert und auf Typen

```
val checkLists: List[Any] => Unit = {
  case Nil =>
    println("eine leere Liste")
  case List(_,2,_) =>
    println("eine dreielementige Liste"+
            " mit einer 2 an Position 2")
  case List(_,_,_) =>
    println("eine dreielementige Liste")
  case _ :: _ :: Nil =>
    println("eine zweielementige Liste")
  case _ :: _ =>
    println("eine nichtleere Liste")
}
```

Da Typ-Parameter in Scala genauso wie in Java beim Übersetzen entfernt werden, ist es nicht möglich, mit einem Pattern bezüglich des Typ-Parameters zu matchen.[8] In Listing 5.19 steht der Versuch, bei Listen von Strings etwas anderes auszugeben wie bei Listen von Ints.

Listing 5.19: Beispiel für Pattern Matching, das sich aufgrund von Type Erasure unerwartet verhält

```
// Achtung: Type Erasure
val doesNotWorkAsExpected: List[Any] => Unit = {
  case _: List[String] =>
    println("eine Liste mit Strings")
  case _: List[Int] =>
    println("eine Liste mit Ints")
}
```

[7] Der Wert `Nil` steht für die leere Liste.
[8] Mehr Informationen zum Typsystem und zur Type Erasure erwarten Sie in Abschnitt 5.7.

Wie die folgende Sitzung zeigt, wird der zweite Fall nie erreicht, weil der Typ-Parameter zur Laufzeit nicht mehr vorhanden ist und beide Muster dann auf jede beliebige Liste passen:

```
scala> val listOfInts = List(1,2,3)
listOfInts: List[Int] = List(1, 2, 3)

scala> doesNotWorkAsExpected(listOfInts)
eine Liste mit Strings
```

Wird die Scala-Shell mit dem Flag `-unchecked` gestartet, sehen wir auch die entsprechenden Warnungen:

```
<console>:6: warning: non variable type-argument String
   in type pattern List[String] is unchecked since it is
   eliminated by erasure
         case _: List[String] =>
                 ^
<console>:8: warning: non variable type-argument Int in
   type pattern List[Int] is unchecked since it is
   eliminated by erasure
         case _: List[Int] =>
                 ^
```

Ist es nur wichtig zu wissen, ob das erste Element ein `String` ist, kann das Pattern wie im ersten Fall von Listing 5.4 aufgebaut sein. Allerdings passt das Pattern dann auch beispielsweise auf die Liste `List("Hallo",1,true,3.4): List[Any]`.

```
val worksAsExpected: List[Any] => Unit = {
  case (x:String)::_ =>
    println(
      "eine Liste mit einem String als erstes Element"
    )
  case _ => println("eine andere Liste")
}
```

Natürlich kann auch auf Tupel gematcht werden, wie das folgende Listing 5.20 zeigt.

Listing 5.20: `checkValue` mit Pattern Matching auf Tupel

```
val checkValue: Any => Unit = {
  case (1,"Hallo") =>
    println("ein Tupel mit einer 1 und \"Hallo\"")
  case (_,1.2)     => println("Ein Tupel mit dem Wert"+
                              " 1.2 an zweiter Stelle")
  case (_,_,_)     => println("Ein Tripel")
  case i: Int      => println("ein Int: "+i)
  case _           => println("etwas anderes")
}
```

Zusätzlich zum Pattern können noch sogenannte *Guards* angegeben werden. Ein Guard ist ein boolescher Ausdruck, der mit `if` eingeleitet wird. Wenn das Pattern passt, wird der dazugehörige Guard ausgewertet. Nur wenn dessen Ergebnis `true` ist, wird der Code hinter dem Pfeil ausgeführt.

Listing 5.21: Pattern Matching mit Guards

```
val checkValue: Any => Unit = {
  case (x:Int,y:Int) if x*y > 0 =>
    println("gleiche Vorzeichen")
  case (_:Int,_:Int) =>
    println("verschiedene Vorzeichen")
  case _ => println("kein Int-Tupel")
}
```

In Listing 5.21 wird ein Guard verwendet, um zu überprüfen, ob die beiden Komponenten des Tupels das gleiche Vorzeichen haben. Vorher wird durch das Pattern sichergestellt, dass es sich bei beiden Komponenten um `Int`s handelt.

5.4.1 Case-Klassen

Bisher haben wir nur Pattern mit vordefinierten Typen verwendet. Pattern Matching ist aber auch mit Objekten selbst definierter Klassen möglich und sogar mit sehr wenig zusätzlichem Aufwand. Wir müssen nur eine *Case-Klasse* durch Voranstellen des Schlüsselwortes `case` definieren. Listing 5.22 zeigt ein einfaches Beispiel.

Listing 5.22: Definition der Case-Klasse `Movie`

```
case class Movie(title: String, fsk: Int)
```

Durch die Definition der Klasse `Movie` als Case-Klasse ist es möglich, auf ein Muster der Form `Movie(t,f)` mit einem `String t` und einem `Int f` zu matchen. Listing 5.23 zeigt die Funktion `printAllowedAtAgeOf`, die den `Movie` mittels Pattern Matching in seine Komponenten zerlegt und dabei unterscheidet, ob die FSK-Zahl eine `0` ist oder nicht.

Listing 5.23: Die Funktion `printAllowedAtAgeOf` mit Pattern Matching für `Movie`-Objekte

```
val printAllowedAtAgeOf: Movie => Unit = {
  case Movie(t,0) =>
    println(t+" (keine Altersbeschraenkung)")
  case Movie(t,i) => println(t+" (frei ab "+i+")")
}
```

Die Nutzung sieht dann wie folgt aus:

```
scala> val movie = Movie("Am Limit",0)
movie: Movie = Movie(Am Limit,0)

scala> printAllowedAtAgeOf(movie)
Am Limit (keine Altersbeschraenkung)

scala> printAllowedAtAgeOf(new Movie("Matrix",16))
Matrix (frei ab 16)
```

Die Funktion `printAllowedAtAgeOf` funktioniert also wie erwartet. Werfen wir aber noch einmal einen Blick auf die ersten beiden Zeilen der obigen Sitzung, so fallen uns zwei Dinge auf: Erstens wir haben den `Movie` ohne `new` erzeugt, und die Ausgabe in der zweiten Zeile sieht auch nicht wie sonst aus.

Das alles verdanken wir dem Schlüsselwort `case` vor der Klassendefinition von `Movie`. Der Compiler ermöglicht damit nicht nur Pattern Matching, sondern generiert auch kanonische Implementierungen von `toString`, `equals` und `hashCode`. Außerdem werden die Klassen-Parameter automatisch zu `val`-Feldern. Bei Bedarf kann aber explizit `var` vorangestellt werden. Das heißt, wir können mit dem Movies sofort einiges anfangen, wie die folgende Sitzung zeigt:

```
scala> println(movie.title)
Am Limit

scala> if (movie == Movie("Am Limit",0))
     |   println("der selbe Film")
der selbe Film

scala> print(movie)
Movie(Am Limit,0)
```

Darüber hinaus erzeugt der Compiler automatisch ein Companion-Objekt mit einer `apply`- und einer `unapply`-Methode. Durch die Factory-Methode `apply` ist es möglich, `Movie("Am Limit",0)` statt `new Movie("Am Limit",0)` zu schreiben. Die `unapply`-Methode ist eine sogenannte *Extraktor-Methode* und wird für das Pattern Matching in Scala benötigt. Mit `unapply` wird ein `Movie`-Objekt in seine Bestandteile in Form eines Tupels der Klassen-Parameter zerlegt:

```
scala> Movie.unapply(movie)
res8: Option[(String, Int)] = Some((Am Limit,0))
```

Achtung: Die automatisch generierten Methoden beziehen sich alle auf die Klassen-Parameter. Es ist natürlich trotzdem möglich, weitere Felder und zusätzliche Konstruktoren zu einer Case-Klasse hinzuzufügen.

Im der obigen Sitzung taucht ein neuer Datentyp auf: `Option[(String,Int)]` mit dem Wert `Some((Am Limit, 0))`. Dieser `Option`-Typ wird in Scala immer dann verwendet, wenn das Ergebnis ein optionaler Wert ist, d.h. wenn auch etwas

schief gehen kann.[9] Allgemein hat `Option[T]` zwei mögliche Werte `Some(x)`, wobei `x` ein Wert vom Typ `T` ist und den erfolgreichen Fall repräsentiert, und den Wert `None`, der dem Fehlerfall entspricht.

Der `Option`-Typ wird beispielsweise in der Scala Collections-Bibliothek verwendet, wie die folgende Sitzung beispielhaft verdeutlichen soll:

```
scala> val cities =
     |   Map("A" -> "Augsburg", "B" -> "Berlin")
cities: scala.collection.immutable.Map[java.lang.String,
   java.lang.String] = Map((A,Augsburg), (B,Berlin))

scala> cities get "A"
res0: Option[java.lang.String] = Some(Augsburg)

scala> cities get "C"
res1: Option[java.lang.String] = None
```

Nachdem wir eine `Map` erzeugt haben, gibt der Zugriff auf das Paar mit dem Schlüssel `"A"` den Wert `"Augsburg"` eingebettet in `Some` zurück. Der versuchte Zugriff auf das Paar mit dem Schlüssel `"C"`, das nicht vorhanden ist, führt zum Ergebnis `None`.

Ab Scala 2.8.0 wird für Case-Klassen auch eine `copy`-Methode generiert, mit der eine Kopie eines Objektes erzeugt werden kann, z.B.

```
scala> val movie2 = movie.copy()
movie2: Movie = Movie(Am Limit,0)
```

Dank der mit Scala 2.8 neu eingeführten *named* und *default arguments* (siehe Seite 36) können einzelne Felder, auch wieder beschränkt auf die Klassen-Parameter, elegant mit einem anderen Wert belegt werden, z.B.

```
scala> val movie3 = movie.copy(title = "Biene Maja")
movie3: Movie = Movie(Biene Maja,0)
```

Um Laufzeitfehler zu vermeiden, ist es wichtig, beim Pattern Matching alle möglichen Fälle zu beachten. Wird ein Fall nicht behandelt, wird bei Datentypen aus der Scala-Standard-Bibliothek eine entsprechende Warnung ausgegeben:

```
scala> val errorOnEmptyList: (List[Any]) => Any = {
     |   case x::_ => x
     | }
<console>:6: warning: match is not exhaustive!
missing combination            Nil
errorOnEmptyList: (List[Any]) => Any = <function1>
```

Diese hilfreiche Warnung taucht natürlich nicht auf, sobald ein Default-Fall mit dem Unterstrich definiert wird:

[9] In Java werden für diese Art von Fehlern üblicherweise `null`-Werte verwendet.

```
scala> val throwExceptionOnEmptyList
     |  : (List[Any]) => Any = {
     |   case x::_ => x
     |   case _ => throw new Exception
     | }
throwExceptionOnEmptyList: (List[Any]) => Any =
    <function1>
```

Im folgenden Abschnitt zeigen wir Ihnen, wie wir eine solche Warnung auch für selbst definierte Klassen bekommen können.

5.4.2 Versiegelte Klassen

Wir definieren zunächst eine abstrakte Klasse `Lecture` für Lehrveranstaltungen. Bei den Lehrveranstaltungen unterscheiden wir zwischen Vorlesungen, Übungen und von Studenten gehaltenen Tutorien. Um Pattern Matching zu nutzen, definieren wir die drei Subklassen `Course`, `Exercise` und `Tutorial` als Case-Klassen. Die Implementierung der vier Klassen ist in Listing 5.24 wiedergegeben.

Listing 5.24: Die Klassen `Lecture`, `Course`, `Exercise` und `Tutorial`

```
abstract class Lecture
case class Course(title: String) extends Lecture
case class Exercise(belongsTo: Course) extends Lecture
case class Tutorial(belongsTo: Course) extends Lecture
```

Anschließend können wir die Funktion `courseTitle` (wie in Listing 5.25 gezeigt) mit Pattern Matching definieren.

Listing 5.25: Die Funktion `courseTitle`

```
def courseTitle(lecture: Lecture) = {
  lecture match {
    case Course(title) => title
    case Exercise(Course(title)) => title
    case Tutorial(Course(title)) => title
  }
}
```

Definieren wir nur Pattern für `Course` und `Exercise`, so kann die Funktion trotzdem ohne Fehlermeldung übersetzt werden:

```
scala> def courseTitle(lecture: Lecture) = {
     |   lecture match {
     |     case Course(title) => title
     |     case Exercise(Course(title)) => title
     |   }
     | }
courseTitle: (lecture: Lecture)String
```

Die Anwendung der Funktion auf ein `Tutorial` führt dann aber zu einem `MatchError`, wie die folgende Sitzung zeigt:

```
scala> courseTitle(Tutorial(Course(
     |   "Fortgeschrittene funktionale Programmierung")))
scala.MatchError: Tutorial(Course(Fortgeschrittene
   funktionale Programmierung))
        at .courseTitle(<console>:11)
        ...
```

Der Scala-Compiler kann ohne zusätzliche Information auch nicht warnen, dass ein Fall beim Matching in der Funktion `courseTitle` vermutlich vergessen wurde, da es in einer objektorientierten Sprache grundsätzlich möglich ist, eine weitere Klasse von `Lecture` abzuleiten.

Die Erzeugung weiterer Subklassen von `Lecture` kann aber unterbunden werden. Damit kann der Compiler die gewünschte Warnung ausgeben. Dazu müssen wir die Klasse `Lecture` mit dem Schlüsselwort `sealed` *versiegeln*. Für eine *Sealed Class* gilt, dass alle davon abgeleiteten Klassen in derselben Sourcecode-Datei wie die versiegelte Klasse stehen müssen. Also definieren wir die vier Klassen wie in 5.26 dargestellt in einer Datei.

Listing 5.26: Die versiegelte Klasse `Lecture` und die davon abgeleiteten Case-Klassen `Course`, `Exercise` und `Tutorial`

```
// in einer gemeinsamen Sourcecode-Datei
sealed abstract class Lecture
case class Course(title: String) extends Lecture
case class Exercise(belongsTo: Course) extends Lecture
case class Tutorial(belongsTo: Course) extends Lecture
```

Die anschließende Definition von `courseTitle` nur für `Course` und `Exercise` liefert dann die gewünschte Warnung:

```
scala> def courseTitle(lecture: Lecture) = {
     |   lecture match {
     |     case Course(title) => title
     |     case Exercise(Course(title)) => title
     |   }
     | }
<console>:11: warning: match is not exhaustive!
missing combination       Tutorial
         lecture match {
         ^
courseTitle: (lecture: Lecture)String
```

Allerdings sind die Möglichkeiten des Compilers bei versiegelten Klassen nur auf die direkt abgeleiteten Case-Klassen beschränkt. Definieren wir `courseTitle`, wie in der folgenden Sitzung zu sehen, bekommen wir keine Warnung, können aber dennoch einen `MatchError` verursachen:

```
scala> def courseTitle(lecture: Lecture) = {
     |   lecture match {
     |     case Course(title) => title
     |     case Exercise(Course(title)) => title
     |     case Tutorial(Course("Prog 2")) => "Prog 2"
     |   }
     | }
courseTitle: (lecture: Lecture)String

scala> courseTitle(Tutorial(Course("Prog 1")))
scala.MatchError: Tutorial(Course(Prog 1))
        at .courseTitle(<console>:13)
        ...
```

Verwenden wir zwar eine versiegelte Klasse, wollen aber bewusst einzelne Fälle auslassen, z.B. weil wir wissen, dass sie nicht vorkommen können, können wir dies dem Compiler mit der `@unchecked`-Annotation mitteilen. Wir bekommen in dem Fall keine Warnung mehr. Zum Beispiel würden wir bei der folgenden Funktion ohne die `@unchecked`-Annotation eine Warnung bekommen, dass der Fall `Nil` fehlt. Mit der Annotation können wir die Funktion ohne Warnung übersetzen:

```
def nonEmptyListMatch(l: List[Any]) =
  (l: @unchecked) match {
    case 1::_ => println("Liste beginnt mit der 1")
    case _::_ => println("Liste beginnt nicht mit 1")
  }
```

5.4.3 Partielle Funktionen

Wenn wir, wie im vorherigen Abschnitt gezeigt, zwar alle Case-Klassen abdecken, aber innerhalb der Klassen nicht alle Fälle, so können wir `courseTitle` als *partielle Funktion*[10] definieren, wie in Listing 5.27 dargestellt.

Listing 5.27: Die partielle Funktion `courseTitle`

```
def courseTitle: PartialFunction[Lecture,String] = {
  case Course(title) => title
  case Exercise(Course(title)) => title
  case Tutorial(Course("Prog 2")) => "Prog 2"
}
```

Die `PartialFunction` hat zwei Typ-Parameter: den Argumenttyp und den Ergebnistyp. Im obigen Listing wird also eine partielle Funktion von `Lecture` nach `String` definiert. Nutzen wir die partielle Funktion wie die zuvor definierte

[10] Eine Funktion, die nicht für alle Werte ihres Definitionsbereichs definiert ist, heißt *partielle* (also partiell definierte) `Funktion`.

Funktion `courseTitle`, so können wir damit immer noch einen `MatchError` erzeugen:

```
scala> courseTitle(Tutorial(Course("Prog 1")))
scala.MatchError: Tutorial(Course(Prog 1))
        at .courseTitle(<console>:13)
        ...
```

Es ist aber möglich, mit der Methode `isDefinedAt` dynamisch (also zur Laufzeit) zu überprüfen, ob die partielle Funktion an der entsprechenden Stelle definiert ist, *bevor* wir unwissentlich in den `MatchError` laufen:

```
scala> courseTitle.isDefinedAt(
     |   Tutorial(Course("Prog 1")))
res5: Boolean = false
```

Der Trait `PartialFunction` definiert noch einige andere nützliche Methoden:

- `andThen` komponiert die partielle Funktion mit einer Funktion, die das Ergebnis transformiert, z.B.

  ```
  scala> (courseTitle andThen (_.map(_.toUpper)))(
       |   Course("Programmierung 2"))
  res6: String = PROGRAMMIERUNG 2
  ```

- `orElse` komponiert die partielle Funktion mit einer Funktion, die berechnet wird, wenn die partielle Funktion für das Argument nicht definiert ist, z.B.

  ```
  scala> val unknown:PartialFunction[Lecture,String] = {
       |   case _ => "unknown title"
       | }
  unknown: PartialFunction[Lecture,String] = <function1>

  scala> (courseTitle orElse unknown)(
       |   Course("Programmierung 2"))
  res7: String = Programmierung 2

  scala> (courseTitle orElse unknown)(
       |   Tutorial(Course("Programmierung 2")))
  res8: String = unknown title
  ```

- `lift` macht aus der partiellen Funktion eine totale Funktion, die für definierte Werte `x` das Ergebnis `Some(x)` und für nicht definierte Werte das Ergebnis `None` zurückliefert, z.B.

  ```
  scala> (courseTitle lift)(Course("Prog 1"))
  res9: Option[String] = Some(Prog 1)

  scala> (courseTitle lift)(Tutorial(Course("Prog 1")))
  res10: Option[String] = None
  ```

5.4.4 Variablennamen für (Teil-)Pattern

In manchen Fällen wird ein Objekt mit einem Pattern zerlegt, aber das gesamte Objekt auf der rechten Seite des Doppelpfeils wieder genutzt. Um nicht so etwas wie

```
case Tutorial(Course("Prog 2")) =>
  "In diesem Semester: Tutorial zur Vorlesung "+
    Course("Prog2")
```

schreiben zu müssen, bei dem das `Course("Prog 2")` der linken Seite auf der rechten Seite wiederholt wird, kann mit dem Pattern `Tutorial(course @ Course("Prog 2"))` der Name `course` für das Teil-Pattern vergeben werden, sodass statt oben angegebener Zeile

```
case Tutorial(course @ Course("Prog 2")) =>
  "In diesem Semester: Tutorial zur Vorlesung "+course
```

geschrieben werden kann.

5.4.5 Exception Handling

Auch wenn *Exception Handling* (*Ausnahmebehandlung*) nicht der funktionalen Programmierung entstammt, passt es gut in den Abschnitt über Pattern Matching. Müssen wir in einem Code-Block damit rechnen, dass eine Exception geworfen werden kann, z.B. weil wir versuchen, eine Datei zu öffnen, umschließen wir den Block mit `try` und reagieren in dem darauf folgendem Catch-Block auf die möglichen Ausnahmen. Im Gegensatz zu anderen Programmiersprachen wie Java, die für verschiedene Exceptions mehrere Catch-Blöcke vorsehen, wird in Scala in einem einzigen Catch-Block mittels Pattern Matching zwischen den Exceptions unterschieden.

Listing 5.28 zeigt ein Beispiel, bei dem eine Zahl von der Kommandozeile eingelesen werden soll. Gibt der Benutzer eine Zeichenkette ein, die sich nicht in einen `Int` umwandeln lässt, so wird eine Exception geworfen. Diese wird dann in dem Catch-Block abgefangen.

Listing 5.28: Exception-Handling in der Funktion `readIntOption`

```
def readIntOption() = {
  try {
    Some(readInt)
  } catch {
    case _: NumberFormatException => None
  }
}
```

Im Try-Block können wir davon ausgehen, dass alles gut geht, und den Wert, den `readInt` einliest, in ein `Some`-Objekt einpacken. Geht etwas schief, wird

der Block verlassen und der Catch-Block betreten. In diesem wird die Exception mit Pattern untersucht. Trifft das Pattern zu, wird der entsprechende Code ausgeführt. Passt kein Pattern, wird die Exception nicht behandelt und weiter geworfen. In Listing 5.28 gibt es nur ein einziges Pattern, auf das eine `NumberFormatException` passt. Alle anderen Exceptions bleiben von der Funktion `readIntOption` unbehandelt.

Für das Behandeln verschiedener Exceptions werden keine zusätzlichen Catch-Blöcke benötigt. Durch das Pattern Matching können mehrere Exceptions in einem Block behandelt werden. Listing 5.29 zeigt eine alternative Implementierung der Funktion `readIntOption`, bei der bei allen Exceptions außer der `NumberFormatException` eine zuvor selbst definierte `ReadIntException` geworfen wird.

Listing 5.29: Unterschiedliches Handling von Exceptions in `readIntOption`

```
case object ReadIntException extends
    Exception("unexpected error in readIntOption")

def readIntOption() = {
  try {
    Some(readInt)
  } catch {
    case _: NumberFormatException => None
    case _ => throw ReadIntException
  }
}
```

Nachdem das Objekt `ReadIntException` als Case-Objekt definiert wurde, kann vom Aufrufer darauf gematcht werden. Es gibt außerdem auch einen Finally-Block, der immer ausgeführt wird, egal ob der Try-Block problemlos abgearbeitet werden konnte oder eine Exception geworfen wurde. Bei der Exception ist es auch ohne Belang, ob sie in einem Catch-Block gefangen wurde oder nicht. Der Finally-Block schließt sich mit dem Schlüsselwort `finally` an den Catch-Block oder, falls nicht vorhanden, an den Try-Block an. Ein `try` ohne `catch` mit einem `finally` ist dann sinnvoll, wenn keine Exception behandelt werden soll, aber eine oder mehrere Anweisungen wie z.B. das Schließen eines Dateihandles unbedingt ausgeführt werden sollen, bevor die Funktion beendet wird.

Listing 5.30: Die Funktion `readIntOption` mit `try`, `catch` und `finally`

```
def readIntOption() = {
  try {
    Some(readInt)
  } catch {
    case _: NumberFormatException => None
    case _ => throw ReadIntException
  } finally {
```

```
      println("readIntOption beendet")
    }
  }
```

Listing 5.30 zeigt ein Beispiel, in dem am Ende der Funktion immer auf der Console `readIntOption beendet` ausgegeben wird und zwar, *bevor* `Some`, `None` oder die Exception weitergegeben wird.

Scala kennt keinen Unterschied zwischen sogenannten *checked* und *unchecked exceptions*. Java macht aber diese Unterscheidung und verlangt, das Checked Exceptions in der Signatur einer Methode angegeben werden. Dies ist wiederum in Scala zunächst gar nicht möglich. Wird aber in Java eine Scala-Methode genutzt, die eine Checked Exception werfen kann, und hat die Java-Methode einen Catch-Block dafür, wird der Java-Compiler einen Fehler anzeigen, der besagt, dass die Exception gar nicht geworfen wird – sonst würde sie ja in der Signatur stehen.

Der Ausweg in Scala ist die `@throws`-Annotation, die die Klasse der Exception als Argument hat, z.B.

```
class Reader(fname: String) {
  private val in = new BufferedReader(new FileReader(
      fname))
  @throws(classOf[IOException])
  def read() = in.read()
}
```

Wenn die Chance besteht, dass Ihr Scala-Code aus Java heraus genutzt werden soll, sollten Sie sich angewöhnen, die `@throws`-Annotation zu nutzen. Innerhalb von Scala ist sie unnötig.

5.4.6 Extraktoren

In Abschnitt 5.4.1 haben wir Case-Klassen und ihre zahlreichen Vorteile erläutert. Case-Klassen haben aber auch einen Nachteil: Sie machen ihre konkrete Datenrepräsentation explizit. Wenn es nur um die Eleganz beim Pattern Matching geht, müssen wir keine Case-Klasse nutzen. Wir können stattdessen ein Objekt mit einer `unapply`-Methode implementieren. Ein Objekt mit einer `unapply`-Methode nennen wir *Extraktor*.

Listing 5.31 zeigt das Extraktor-Objekt `HeaderField`, das ein Schlüssel-Wert-Paar aus einem E-Mail-Header-Feld extrahiert, z.B. aus der Zeichenkette `From: scala@obraun.net` das Tupel `(From,scala@obraun.net)`.

Listing 5.31: Der Extraktor `HeaderField`

```
object HeaderField {
  def unapply(s: String) = {
    val (key,value) = s.span(_!=':')
    if (key.isEmpty || value.isEmpty)
```

```
        None
      else
        Some((key,value.tail trim))
    }
  }
```

Die `unapply`-Methode versucht, die Zeichenkette beim ersten Vorkommen des Zeichens `:` in zwei Teile zu zerlegen. Ist eines der beiden Teile leer, so war der Doppelpunkt das erste oder letzte Zeichen in der Zeile oder kam überhaupt nicht vor. Andernfalls werden vom zweiten Teil das erste Zeichen, der Doppelpunkt, und anschließend alle Leerzeichen am Anfang und am Ende entfernt.

In Listing 5.32 ist die Klasse `Email` dargestellt. Ein `Email`-Objekt besteht aus einer `Map` mit Paaren aus dem Namen eines Headerfeldes und dem dazugehörigen Wert und aus einem `String`, der den Text der E-Mail repräsentiert. Die `toString`-Methode ist zu Demonstrationszwecken redefiniert (siehe Scala-Shell-Sitzung weiter unten).

Listing 5.32: Die Klasse `Email`

```
class Email(val header: Map[String,String],
            val body: String) {
  override def toString: String = header+"\n"+body
}
```

Die Funktion `stringToEmail` (siehe Listing 5.33) nutzt den Extractor `HeaderField`, um mittels Pattern Matching eine Headerzeile in ihre Bestandteile zu zerlegen. Dazu wird der Parameter `s` zuerst mit `lines.toList` in eine Liste von Zeilen zerlegt. Anschließend wird die Liste in zwei Teile zerlegt. Die leere Zeichenkette dient dabei als Trenner, denn zwischen Header und Body einer E-Mail ist eine Leerzeile.

Mit der Header- und Bodyliste erzeugen wir ein neues `Email`-Objekt, wobei wir die beiden Listen noch in eine `Map` bzw. einen `String` umwandeln müssen. Die Liste der Headerfelder bearbeiten wir mit einem Fold von rechts nach links mit einer leeren `Map` als Startwert. Die Funktion zum Falten nimmt eine Zeile und die bisher berechnete `Map`. Kann die Zeile mit dem `HeaderField`-Extraktor in ein Schlüssel-Wert-Paar zerlegt werden, wird aus dem Paar und der `Map` mit der `Map`-Methode `+` eine neue[11] `Map` erzeugt. Die Liste, die den E-Mail-Body repräsentiert, enthält noch die führende Leerzeile. Diese wird entfernt[12], und aus der Liste von Zeilen wird wieder ein einziger String mit Zeilenumbrüchen erzeugt.

[11] Die ohne speziellen Import verwendete `Map` ist `scala.collection.immutable.Map`, also unveränderbar.

[12] Genau genommen wird auch hier wieder eine neue Liste ohne die führende Leerzeile berechnet.

Listing 5.33: Die Funktion `stringToEmail`

```
implicit def stringToEmail(s: String) = {
  val (headLines,body) = s.lines.toList span (_!="")
  new Email(
    (headLines :\ Map[String,String]())(
      (field,map) =>
        field match {
          case HeaderField(k,v) => (map + (k -> v))
          case _ => map
        }
     ),
     body.tail mkString "\n"
   )
}
```

Nachdem die Funktion `stringToEmail` mit dem Schlüsselwort `implicit` markiert wurde, müssen wir die Funktion nicht einmal explizit anwenden, sondern können einen `String` einer Variablen vom Typ `Email` zuweisen. Nachdem dies zu einem Typfehler führt, darf der Compiler alle impliziten Funktionen überprüfen, ob genau eine davon in der Lage ist, den `String` in eine `Email` zu transformieren. Die folgende Sitzung zeigt dies im Beispiel:

```
scala> val email: Email = """From: scala@obraun.net
     |                     |Subject: Hallo Leser
     |                     |
     |                     |Viel Spass mit Scala.
     |                     |
     |                     |Gruss
     |                     |Oliver Braun""".stripMargin
email: Email =
Map(Subject -> Hallo Leser, From -> scala@obraun.net)
Viel Spass mit Scala.

Gruss
Oliver Braun
```

5.4.7 Pattern Matching mit regulären Ausdrücken

Scala unterstützt wie viele moderne Programmiersprachen auch reguläre Ausdrücke. Mit regulären Ausdrücken können Zeichenketten zerlegt werden. Ein regulärer Ausdruck passt auf eine Zeichenkette oder er passt nicht. Insofern ist er den bisherigen Pattern sehr ähnlich. Daher können reguläre Ausdrücke in Scala genau wie Pattern genutzt werden.

Um einen regulären Ausdruck zu erzeugen, kann die Klasse `scala.util.matching.Regex` genutzt werden. Diese steht nicht über das `Predef`-Objekt zur Verfügung, sondern muss explizit importiert werden. In der folgenden Sitzung wird ein regulärer Ausdruck für ein Datum in der Form `Tag.Monat.Jahr`