Scala Tutorial
für Java Programmierer

Version 1.3
5. März 2011

Michel Schinz,
Philipp Haller

Übersetzung
Ulrich Jordan

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Einleitung

Dieses Dokument ist eine kurze Vorstellung der Sprache Scala und des Scala-Compilers. Es ist für Leser gedacht, die schon Programmiererfahrung haben und die sich einen Überblick über Scala verschaffen wollen. Dabei werden Grundkenntnisse in objektorientierter Programmierung und speziell in Java vorausgesetzt.

2 Ein erstes Beispiel

Als ein erstes Beispiel verwenden wir traditionsgemäß das *Hello-World*-Programm. Es ist nicht sehr faszinierend, demonstriert aber die Verwendung einiger Scala-Werkzeuge, ohne dass man viel über die Sprache wissen muss. Und so sieht es aus:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

Die Struktur dieses Programms wird jedem Java-Programmierer bekannt vorkommen. Das Programm besteht aus einer Methode namens `main` mit den Befehlszeilenargumenten in Form eines String Arrays als Parameter. Im Methodenrumpf wird die vordefinierten Methode `println` ausgeführt, der noch eine freundliche Begrüßung als Argument übergeben wird. Die `main`-Methode ist eine prozedurale Methode, hat also keinen Rückgabewert. Deshalb muß auch kein Rückgabetyp deklariert werden.

Was Java-Programmierer weniger vertraut sein dürfte, ist die **`object`** Deklaration, die die `main`-Methode einschließt. Eine solche Deklaration beschreibt eine Klasse mit einer einzigen Instanz. So eine Klasse nennt man *Singleton-Objekt*. Die Deklaration beschreibt also eine Klasse namens `HelloWorld` und gleichzeitig eine Instanz dieser Klasse mit demselben Namen. Die Instanz wird beim ersten Aufruf erzeugt.

Der aufmerksame Leser hat vielleicht bemerkt, dass die `main`-Methode nicht als `static` deklariert wurde. Dies liegt daran, das es in Scala keine statischen Klassenmitglieder (Methoden oder Felder) gibt. Statt statische Methoden oder Felder zu deklarieren, werden diese innerhalb eines Singleton-Objektes deklariert.

2.1 Das Beispiel wird compiliert

Zum Compilieren dieses Beispiels rufen wir `scalac` auf, den Scala-Compiler. `scalac` arbeitet wie die meisten Compiler. Aus einer Quelldatei als Argument und gegebenenfalls noch weiteren Optionen werden eine oder mehrere Zieldateien erzeugt. Die Zieldateien sind normale Java-Klassendateien.

Wenn wir das obige Programm in einer Datei namens `HelloWorld.scala` speichern,

können wir es mit dem folgenden Befehl compilieren (Das Größer-Zeichen '>' steht für den Shell Prompt und wird nicht eingegeben.):

```
> scalac HelloWorld.scala
```

Dadurch werden mehrere Klassendateien im aktuellen Verzeichnis generiert. Die Datei `HelloWorld.class` enthält eine Klasse, die durch den `scala` Befehl direkt ausgeführt werden kann.

2.2 Das Beispiel wird ausgeführt

Einmal compiliert, kann das Scala-Programm mit dem Befehl `scala` ausgeführt werden. Die Verwendung von `scala` ähnelt der des `java` Befehls, mit dem Java-Programme ausgeführt werden und `scala` akzeptiert die gleichen Optionen. Das obige Beispiel wird also mit folgendem Befehl ausgeführt und liefert das zu erwartende Ergebnis:

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Interaktion mit Java

Einer der Stärken von Scala ist es die einfache Interaktion mit Java-Programmen. Alle Klassen des Pakets `java.lang` werden defaultmäßig importiert, alle anderen können explizit importiert werden.

Schauen wir uns dazu ein Beispiel an. Wir wollen ein aktuelles Datum erzeugen und dies formatiert gemäß den Konventionen eines bestimmten Landes, hier Frankreich[1], ausgeben.

Die Java-Klassenbibliotheken definieren dafür leistungsfähige Klassen, nämlich `Date` und `DateFormat`. Da Scala nahtlos mit Java interagiert, besteht nicht die Notwendigkeit, vergleichbare Klassen in der Scala-Klassenbibliothek zu definieren. Wir können einfach die Klassen der entsprechenden Java-Pakete importieren:

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

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
```

[1] Auch in andere Regionen wie dem französisch sprechenden Teil der Schweiz wird dasselbe Format verwendet.

```
    val df = getDateInstance(LONG, Locale.FRANCE)
    println(df format now)
  }
}
```

Die Scala **import**-Anweisung ähnelt sehr dem Äquivalent in Java, ist aber mächtiger. Verschiedene Klassen aus dem gleichen Paket können, wie in der ersten Zeile zu sehen ist, importiert werden, indem sie in geschweifte Klammern gesetzt werden. Ein weiterer Unterschied ist, dass beim Import aller Namen eines Pakets oder einer Klasse der Unterstrich (_) anstelle des Sternchens (*) verwendet wird. Das ist so, weil das Sternchen ein gültiger Scala Bezeichner ist und als Methodenname verwendet wird. Dazu später mehr.

Die **import**-Anweisung in der dritten Zeile importiert also alle Klassenmitglieder der `DateFormat`-Klasse. Dies macht die statische Methode `getDateInstance` und das statische Feld `LONG` sichtbar.

Innerhalb der `main`-Methode erzeugen wir zuerst eine Instanz der Java-Klasse `Date`, die per default das aktuelle Datum enthält. Als nächstes definieren wir ein Datumsformat mit der statischen `getDateInstance`-Methode, die wir zuvor importiert haben. Schließlich geben wir das aktuelle Datum formatiert gemäß der lokalisierten `DateFormat`-Instanz aus. Diese letzte Zeile zeigt eine interessante Eigenschaft der Scala Syntax. Methoden, die nur ein Argument verwenden, können in Infix-Notation geschrieben werden. Das heißt, der Ausdruck

```
df format now
```

ist nur ein anderer Ausdruck für:

```
df.format(now)
```

Dies mag nur wie ein unwichtiges syntaktisches Detail erscheinen, hat aber weitreichende Konsequenzen, die im nächsten Kapitel untersucht werden.

Zum Schluss dieses Kapitels zur Interaktion mit Java sei anzumerken, dass es auch möglich ist, direkt in Scala Java-Klassen zu beerben bzw. Java-Interfaces zu implementieren.

4 Alles ist ein Objekt

Scala ist eine reine objektorientierte Sprache in dem Sinne, dass wirklich alles ein Objekt ist, einschließlich Zahlen und Funktionen. In dieser Hinsicht unterscheidet Scala sich von Java, da in Java zwischen primitive Typen (wie `boolean` und `int`) und Referenz-Typen unterschieden wird. Auch ist es in Java nicht möglich, Funktionen wie Werte zu behandeln.

4.1 Zahlen sind Objekte

Da Zahlen Objekte sind, haben sie auch Methoden. Und in der Tat besteht ein arithmetischer Ausdruck wie der folgende:

```
1 + 2 * 3 / x
```

ausschließlich aus Methodenaufrufen. Er ist äquivalent zum folgenden Ausdruck, wie wir schon im letzten Kapitel gesehen haben:

```
(1).+(((2).*(3))./(x))
```

Dies bedeutet auch, dass `+`, `*`, etc. gültige Bezeichner in Scala sind.

Die Klammern um die Zahlen in der zweiten Version sind notwendig, da der Scala Scanner nach der Regel verfährt, jeweils den längsten möglichen Token zu ermitteln. Gemäß dieser Regel wird folgender Ausdruck:

```
1.+(2)
```

in die Token `1.`, `+` und `2` aufgeteilt. Weil `1.` ein gültiger Token ist und er länger als `1` ist, wird diese Aufteilung gewählt. Der Token `1.` steht für das Literal `1.0`, ist also von der Klasse `Double` und nicht `Int`. Bei dem Ausdruck:

```
(1).+(2)
```

verhindern die Klammern, dass `1` als `Double` interpretiert wird.

4.2 Funktionen sind Objekte

Vielleicht noch überraschender für Java-Programmierer ist die Tatsache, das Funktionen in Scala auch Objekte sind. Es ist daher möglich, Funktionen als Argumente zu übergeben, sie in Variablen zu speichern und sie von anderen Funktionen zurückgeben zu lassen. Diese Eigenschaft, Funktionen als veränderliche Werte zu behandeln, ist einer der Grundpfeiler eines äußerst interessanten Programmier-Paradigmas, nämlich dem der *funktionalen Programmierung*.

Als ein sehr einfaches Beispiel dafür, warum es nützlich sein kann, Funktionen als Werte zu behandeln, wollen wir eine Timer-Funktion betrachten, die pro Sekunde eine bestimmte Aktion durchführt. Wie geben wir diese Aktion vor? Logischerweise als eine Funktion. Diese sehr einfache Art der Funktionsübergabe wird vielen Programmierern vertraut sein. Sie wird häufig verwendet, um in User-Interface-Programmen Call-Back-Funktionen zu registrieren, die bei Eintritt eines Ereignisses ausgeführt werden.

In dem folgenden Programm wird die Timer-Funktion `oncePerSecond` definiert, und zwar mit einer Call-Back-Funktion als Argument. Die Art der Funktion wird beschrieben durch `() => Unit` und damit ist eine Funktion gemeint, die keine Argu-

mente verwendet und keinen Wert zurückgibt. (Der Typ `Unit` ist vergleichbar mit `void` in C/C++.) Die `main`-Funktion dieses Programms ruft lediglich diese Timer-Funktion mit einer Call-Back-Funktion, die einen Satz ausgibt, auf. Anders ausgedrückt gibt dieses Programm jede Sekunde den Satz “time flies like an arrow“ aus, und zwar ohne zu stoppen.

```
object Timer {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("time flies like an arrow...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

Weiter sei noch angemerkt, dass hier die vordefinierte Methode `println` verwendet wird, um den Satz auszugeben, nicht die Methode des `System.out`-Objekts.

4.2.1 Anonyme Funktionen

Dieses Programm ist leicht zu verstehen, kann aber auch noch ein wenig verfeinert werden. Zunächst ist zu bemerken, dass die Funktion timeFlies nur definiert wird, um sie später an die Funktion `oncePerSecond` weiterzugegeben. Da man diese Funktion nur einmal verwendet, scheint es unnötig zu sein, ihr einen Namen zu geben. Und es wäre schöner, wenn man die Funktion gerade dort definieren könnte, wo sie an `oncePerSecond` übergeben wird. Dies ist in Scala durch Verwendung *anonymer Funktionen*, also Funktionen ohne Namen, möglich. Die überarbeitete Version unseres Timer-Programms mit anonymer Funktion anstelle von timeFlies sieht so aus:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => Unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("time flies like an arrow..."))
  }
}
```

Das Vorhandensein einer anonymen Funktion wird in diesem Beispiel durch den Pfeil nach rechts ‘=>’ angezeigt, welcher die Liste der Argumente vom Rumpf der

Funktion trennt. In diesem Beispiel ist das Argument leer, was durch das leere Paar Klammern links vom Pfeils angedeutet wird. Der Funktionsrumpf ist der gleiche wie der von `timeFlies` oben.

5 Klassen

Wie wir oben gesehen haben, ist Scala eine objektorientierte Sprache, und dazu gehört das Konzept der Klasse[2]. Klassen werden in Scala mit einer Syntax deklariert, die der von Java sehr ähnlich ist. Ein wichtiger Unterschied besteht darin, dass Klassen in Scala Parameter haben können. Dieses wird in folgender Definition der komplexen Zahlen verwendet.

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

Die Klasse `Complex` hat zwei Argumente, den Real- und den Imaginärteil der komplexen Zahl. Diese Argumente müssen übergeben werden, wenn eine Instanz der Klasse `Complex` erzeugt werden soll, und zwar wie folgt: `new Complex(1.5, 2.3)`. Die Klasse enthält weiter zwei Methoden `re` und `im`, die den Zugriff auf Real- und Imaginärteil beschreiben.

Anzumerken ist, dass der Rückgabetyp der beiden Methoden nicht explizit angegeben wird. Dieser wird automatisch durch den Compiler ermittelt, der aus der rechten Seite der Zuweisung folgert, dass die Methode einen Wert vom Typ `Double` zurück gibt.

Der Compiler ist nicht immer in der Lage, auf den Typen zu folgern. Und leider gibt es auch keine einfache Regel, nach der genau zu ermitteln ist, wann es dem Compiler möglich ist und wann nicht. In der Praxis ist dies kein Problem, da der Compiler sich beschwert, wenn er nicht in der Lage ist, auf den Typ zu folgern, falls dieser nicht explizit angegeben wird. Als Anfänger in der Scala Programmierung sollten man nach folgender einfachen Regel verfahren. Man lässt den Typ der Deklarationen in Fällen, in denen er einfach aus dem Kontext abzuleiten scheint, weg und wartet ab, ob der Compiler dies genauso sieht. Nach einiger Zeit wird man ein Gefühl dafür entwickeln, wann man den Typ weglassen kann und wann man ihn explizit angeben muss.

5.1 Methoden ohne Argumente

Ein kleines Problem bei der Verwendung der Methoden `re` und `im` ist es, dass man bei jedem Aufruf ein leeres Paar Klammern hinter ihrem Methodennamen schrei-

[2] Vollständigerweise sollte angemerkt werden, dass es objektorientierte Sprachen gibt, die nicht das Konzept der Klasse kennen. Scala gehört aber nicht dazu.

ben muss, wie das folgende Beispiel zeigt:

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

Es wäre netter auf realen und imaginärem Teil zuzugreifen, wie wenn sie Felder wären - also ohne das leere Paar Klammern. Dies ist in Scala leicht möglich, einfach durch die Definition von ihnen als Methode *ohne Argumente*. Solche Methoden unterscheiden sich von Methoden mit null Argumenten dadurch, dass sie nach dem Namen keine Klammern stehen, weder bei der Definition noch beim Zugriff. Unsere Klasse `Complex` kann damit wie folgt umgeschrieben werden:

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
}
```

5.2 Vererbung und Überschreibung

Alle Klassen in Scala erben von einer Super-Klasse. Wenn kein Super-Klasse angegeben wird, wie beim `Complex` Beispiel im letzten Kapitel wird implizit `scala.AnyRef` verwendet.

Es ist natürlich auch in Scala möglich, Methoden, die von einer Super-Klasse geerbt wurden, zu überschreiben. Es muss dann jedoch explizit angeben werden, dass eine Methode eine andere überschreibt, um ein versehentliches Überschreiben zu verhindern. Dies geschieht durch Verwendung des **`override`** Modifizierers. Als Beispiel wird in unserer `Complex`-Klasse die `toString`-Methode, die von `Object` geerbt wurde, überschrieben.

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 Case-Klassen and Pattern Matching

Eine Datenstruktur, die oft in Programmen Verwendung findet, ist der Baum. So werden Programme von Interpretern und Compilern in der Regel intern als Baum

repräsentiert. XML-Dokumente sind Bäume und verschiedene Containern basieren auf Bäumen, etwa Rot-Schwarz-Bäumen.

Wir werden nun untersuchen, wie solche Bäume in Scala repräsentiert und verändert werden können. Und zwar am Beispiel eines kleinen Taschenrechner-Programms. Das Ziel dieses Programms ist es, einfache arithmetische Ausdrücke, die aus Summierung, Integer-Konstanten und -Variablen zusammengesetzt sind, auszuwerten. Zwei Beispiele für solche Ausdrücke sind $1+2$ und $(x+x)+(7+y)$.

Wir müssen uns zuerst für eine Darstellung von solchen Ausdrücken entscheiden. Die natürlichste ist die durch einen Baum, bei dem die Knoten Operationen (hier die Summierung) und die Blätter Werte (hier Konstanten und Variablen) sind.

In Java würde ein solcher Baum dargestellt werden durch eine abstrakten Super-Klasse für den Baum und je eine konkrete Unterklasse pro Knoten und Blatt. In einer funktionalen Programmiersprache würde man einen algebraischen Datentyp für den gleichen Zweck verwenden. Scala bietet das Konzept der *Case-Klasse* an, das irgendwo zwischen den beiden Konzepten liegt. Und so könnte man mit Case-Klassen den Baum für unser Beispiel definieren:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Weil die Klassen `Sum`, `Var` und `Const` als Case-Klassen deklariert werden, unterscheiden sie sich von anderen Klassen in folgender Weise:

- Das **`new`** Keyword muß nicht mehr angegeben werden, um Instanzen dieser Klasse zu erzeugen. (Man kann also `Const(5)` schreiben, anstelle von **`new`** `Const(5)`.)
- Es werden automatisch Getter-Funktionen für die Parameter des Konstruktors erzeugt. (Man erhält also etwa den Konstruktor Parameter `v` einer Instanz `c` einfach indem man `c.v` schreibt.)
- Standard-Definitionen für Methoden `equals` und `hashCode` werden erzeugt. Diese berücksichtigen die Struktur der Instanzen. (Und nicht nur die Identität.)
- Eine Standard-Definition für die Methode `toString` wird erzeugt. Diese liefert einen String, der so auch im Quellcode stehen könnte. (Der Baum für den Ausdruck $x+1$ liefert etwa `Sum(Var(x),Const(1))`)
- Instanzen dieser Klassen können durch *Pattern Matching* zerlegt werden, wie wir weiter unten sehen werden.

Da wir nun den Datentyp, der unsere arithmetischen Ausdrücke darstellen wird, definiert haben, können wir nun Operationen auf diesem Datentyp definieren. Beginnen wir mit einer Funktion, die einen Ausdruck in einer *Umgebung* auswertet.

In einer Umgebung werden Variablen Werte zugewiesen. So wird der Ausdruck $x+1$ in einer Umgebung, die der Variablen x den Wert 5 zuordnet (geschrieben $\{x \rightarrow 5\}$), zu 6 ausgewertet.

Wir müssen also in irgendeiner Weise eine Umgebung repräsentieren. Wir könnten dafür natürlich eine assoziative Datenstruktur (wie etwa eine Hash-Tabelle) verwenden - oder wir verwenden gleich eine Funktion! Eine Umgebung ist nämlich nichts anderes als eine Funktion, die einen Wert mit einem (Variablen-)Namen verknüpft. Die obige Umgebung $\{x \rightarrow 5\}$ kann einfach folgendermaßen in Scala definiert werden:

```
{ case "x" => 5 }
```

Diese Schreibweise definiert eine Funktion, die die Zahl 5 zurück gibt, wenn an sie der String "x" übergeben wird. Bei allen anderen Übergabewerten wird eine Ausnahme ausgelöst.

Vor dem Schreiben der Bewertungsfunktion wollen wir noch für den Typ der Umgebung einen Namen vergeben. Wir können natürlich auch die Umgebungen immer als vom Typ `String => Int` definieren. Aber es würde das Programm vereinfachen, wenn wir für diesen Typ einen Namen vergeben könnten. Außerdem werden zukünftige Änderungen dadurch erleichtert. Dies wird in Scala mit der folgenden Notation erreicht:

```
type Environment = String => Int
```

Damit kann `Environment` als eine Art Alias für Funktionen vom Typ `String => Int` verwendet werden.

Wir kommen jetzt zur Definition der Bewertungsfunktion. Konzeptionell ist es einfach: Der Wert einer Summe von zwei Ausdrücken ist einfach die Summe aus den Werten dieser Ausdrücke; der Wert einer Variablen wird direkt aus der Umgebung genommen; und der Wert einer Konstanten ist die Konstante selbst. Dies ist leicht in Scala zu formulieren:

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Diese Bewertungsfunktion wendet Pattern Matching auf den Baum `t` an. Intuitiv ist die Bedeutung obiger Definition klar:

1. Zunächst wird geprüft, ob der Baum `t` eine Summe ist und wenn er es ist, wird der linke Teilbaum an eine Variable `l` gebunden und der rechte Teilbaum an eine Variable `r`. Dann wird der Ausdruck rechts vom Pfeil ausgewertet. Dabei werden die durch das Pattern Matching gebundenen Variablen `l` und `r` ver-

wendet.

2. Wenn die erste Überprüfung nicht erfolgreich ist, wenn also der Baum keine Summe ist, geht es weiter mit einer Prüfung, ob `t` eine Variable ist. Ist dies der Fall, wird der Name des `Var`-Knotens an die Variable `n` gebunden und der rechten Ausdruck wird ausgewertet.
3. Wenn auch die zweite Prüfung nicht erfolgreich ist, also in dem Fall, dass `t` weder eine Summe noch ein Variable ist, wird überprüft, ob `t` eine Konstante ist. Und wenn dies so ist, wird der Wert in dem `Const`-Knoten an eine Variablen `v` gebunden. Dann wird wieder der rechte Ausdruck ausgewertet.
4. Schließlich, wenn alle Überprüfungen fehlschlagen, wird eine Ausnahme ausgelöst, um das Scheitern des Pattern Matching anzuzeigen. Dies könnte aber hier nur passieren, wenn inzwischen weitere Unterklassen deklariert wurden.

Die Grundidee des Pattern Matching ist es also zu versuchen, einen Wert mit einer Folge von Mustern (Pattern) zu vergleichen und sobald ein passendes (matching) Muster gefunden wurde, dieses und andere Teile des Wertes zu extrahieren, um sie dann in der anschließenden Bearbeitung verwenden zu können.

Ein in der Objektorientierung erfahrener Programmierer mag sich vielleicht fragen, warum wir die `eval`-Methode nicht als eine Klassenmethode der Klasse `tree` und ihrer Unterklassen definiert haben. Dies hätten wir auch wirklich tun können, da Scala dies bei Case-Klassen erlaubt.

Die Entscheidung, ob man das Pattern Matching direkt oder in einer Methoden verwendet, ist eine Geschmacksfrage, hat aber auch wichtige Auswirkungen auf die Erweiterbarkeit der Klasse:

- Bei der Verwendung von Methoden ist es leicht, eine neue Art von Knoten hinzuzufügen, und zwar durch die Definition einer Unterklasse von `tree`. Auf der anderen Seite ist das Hinzufügen einer neuen Operation für `tree` mühsam, da alle Unterklassen von `tree` angepasst werden müssen.
- Bei der Verwendung von Pattern Matching ist die Situation umgekehrt. Das Hinzufügen einer neuen Klasse von Knoten erfordert die Änderung aller Funktionen, die ein Pattern Matching auf `tree` anwenden, um den neue Knoten zu berücksichtigen. Auf der anderen Seite ist das Hinzufügen einer neuen Operation einfach, und zwar durch die Definition einer weiteren Funktion.

Um das Pattern Matching weiter zu erkunden, definieren wir eine weitere Operationen auf den arithmetischen Ausdruck: die symbolische Ableitung. Der Leser kennt vielleicht noch die folgenden Regeln dieser Operation:

1. Die Ableitung einer Summe ist die Summe aus den Ableitungen.
2. Die Ableitung von einer Variablen `v` ist 1, wenn nach `v` abgeleitet wird, und 0 sonst.

3. Die Ableitung einer Konstanten ist 0.

Diese Regeln können fast wörtlich in Scala übernommen werden. Damit erhalten wir die folgende Definition:

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Diese Funktion verwendet zwei neue Konzepte zum Pattern Matching. Zunächst einmal hat der Ausdruck für die Variable eine zusätzliche Klausel. Der Ausdruck nach dem Schlüsselwort **if** wird in diesem Zusammenhang *Guard* genannt. Er beschränkt den Erfolg des Vergleichs auf die Fälle, in denen dieser Ausdruck wahr ist. Hier wird er verwendet, um sicherzustellen, dass wir die Konstante 1 nur zurückgeben, wenn der Name der Variablen der gleiche ist wie der, nach der abgeleitet wird. Das zweite neue Konzept, das hier verwendet wird, ist das der *Wild-Card* _. Dieses Muster passt auf jeden Wert.

Wir haben damit noch längst nicht alle Möglichkeiten des Pattern Matching gesehen, werden hier aber aufhören, um dies Dokument kurz zu halten. Wir wollen noch zeigen, wie die beiden Funktionen auf ein reales Beispiel angewendet werden. Zu diesem Zweck schreiben wir eine einfache `main`-Funktion, die mehrere Operationen auf den Ausdruck $(x + x) + (7 + y)$ ausführt. Zuerst wird der Ausdrucks in der Umwelt $\{x \to 5, y \to 7\}$ ausgewertet. Dann wird die Ableitung bzgl. x und y gebildet:

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Bei Ausführung dieses Programms erhalten wir die erwartete Ausgabe:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Bei Durchsicht der Ausgabe erkennen wir, dass das Ergebnis der Ableitung vereinfacht werden sollten, bevor man sie dem Anwender präsentiert. Die Definition ei-

ner allgemeinen Vereinfachungsfunktion mit Hilfe des Pattern Matching ist ein interessantes (aber überraschend kniffliges) Problem. Die Lösung sei dem Leser als Übung überlassen.

7 Traits

Eine Klasse in Scala kann von ihrer Super-Klasse erben und zusätzlich auch von einem oder mehreren *Traits*.

Der vielleicht einfachste Weg für ein Java-Programmierer Traits zu verstehen ist, sie einfach als Interface mit Programmteilen zu betrachten. Wenn in Scala eine Klasse von einem Trait erbt, implementiert sie das Interface des Traits und erbt alle Programmteile des Traits.

Um die Nützlichkeit von Traits zu demonstrieren, wollen wir uns ein klassisches Beispiel ansehen: Objekte mit einer Ordnung. Es ist oft nützlich, Objekte einer bestimmten Klassen untereinander vergleichen zu können, etwa um sie zu sortieren. In Java implementieren Objekte, die vergleichbar sind, das Interface `Comparable`. In Scala können wir dies ein bisschen besser umsetzen, indem wir einen zu `Comparable` äquivalenten Trait einsetzen. Diesen Trait wollen wir `Ord` nennen.

Beim Vergleich von Objekten können uns sechs verschiedene Prädikate nützlich sein: kleiner, kleiner-gleich, gleich, ungleich, größer oder größer-gleich. Allerdings ist es umständlich, alle zu definieren, zumal vier von den sechs mit Hilfe der anderen zwei ausgedrückt werden können. Das heißt, wenn etwa das gleich und das kleiner Prädikate vorhanden sind, können die anderen durch diese zwei ausgedrückt werden. In Scala wird dieser Sachverhalt gut in folgender Deklaration wiedergegeben:

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

Diese Definition liefert einen neuen Trait namens `Ord`, der die gleiche Rolle spielt wie das Java `Comparable`-Interface und der die Standard-Implementierungen von drei Prädikate auf das vierte abstrakte Prädikat zurückführt. Die Prädikate für gleich und ungleich werden nicht deklariert, da sie per default in allen Objekten vorhanden sind.

Der Typ `Any`, der oben verwendet wird, ist Super-Typ für alle anderen Typen in Scala. Er kann also als eine allgemeinere Version von Javas `Object`-Typ angesehen werden, da er auch einen Super-Typ zu den Basistypen wie `int`, `float`, u.a. darstellt.

Um Objekte einer Klasse vergleichbar zu machen, reicht es jetzt aus, die Prädikate

für gleich und kleiner in der Klasse zu definieren und die Klasse `Ord` einzumischen. Als ein Beispiel definieren wir nun eine `Date`-Klasse, die das Datum des gregorianischen Kalenders repräsentiert. Solch ein Datum besteht aus einem Tag, einem Monat und einem Jahr, die wir alle durch ganze Zahlen repräsentieren werden. Wir beginnen also die Definition der `Date`-Klasse wie folgt:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

Der wichtige Teil hier ist die **`extends`** `Ord` Klausel nach dem Klassennamen. So wird deklariert, das die `Date`-Klasse vom `Ord`-Trait erbt.

Dann definieren wir die von der `Object` geerbte Methode `equals`, so dass sie Datums-Instanzen durch Vergleich ihrer Feldern durchführt. Die Default-Implementierung von `equals` ist nicht verwendbar, weil sie wie in Java Objekte über deren Identität vergleicht. Wir kommen auf folgende Definition:

```
  override def equals(that: Any): Boolean =
    that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
      o.day == day && o.month == month && o.year == year
    }
```

Diese Methode verwendet die vordefinierten Methoden `isInstanceOf` und `asInstanceOf`. Die erste Methode `isInstanceOf` entspricht dem Java `instanceof`-Operator und liefert **`true`**, wenn und nur wenn das Objekt, auf den er angewandt wird, eine Instanz des angegebenen Typs ist. Die zweite Methode `asInstanceOf` entspricht dem Java Cast-Operator. Wenn das Objekt eine Instanz des angegebenen Typs ist, wird es als ein solches behandelt. Andernfalls wird eine `ClassCastException`-Ausnahme ausgelöst.

Schließlich ist noch die letzte Methode wie folgt zu definieren, die das Prädikat kleiner implementiert. Es nutzt eine weitere vordefinierte Methode `error`, die eine Ausnahme mit der angegebenen Fehlermeldung auslöst.

```
  def <(that: Any): Boolean = {
    if (!that.isInstanceOf[Date])
      error("cannot compare " + that + " and a Date")

    val o = that.asInstanceOf[Date]
    (year < o.year) ||
    (year == o.year && (month < o.month ||
                        (month == o.month && day < o.day)))
  }
```

Das ist die Definition der `Date`-Klasse. Instanzen dieser Klasse können als Datumsinstanzen betrachtet werden und auch als vergleichbare Objekte. Weiter definieren sie alle sechs erwähnten Prädikate: gleich und kleiner direkt in der Definition der Date-Klasse, die anderen durch Vererbung vom `Ord`-Trait.

Traits sind natürlich auch in anderen als der hier gezeigten Situationen nützlich. Aber die Vorstellung aller Möglichkeiten kann dieses Dokument nicht leisten.

8 Generische Programmierung

Als letztes in diesem Tutorial wollen wir die generische Programmierung in Scala vorstellen. Java-Programmierer kennen die Probleme, die ohne die Möglichkeit der generischen Programmierung entstehen. Dieser Mangel wurde in Java mit Version 1.5 behoben.

Generische Programmierung ist die Möglichkeit, in Programme Typen-Parameter zu verwenden. Zum Beispiel steht ein Programmierer, der eine Bibliothek für verkettete Listen schreiben will, vor dem Problem zu entscheiden, welchen Typ die Elemente der Liste habe sollen. Da diese Liste in vielen verschiedenen Kontexten verwendet werden soll, ist es nicht möglich im voraus zu entscheiden, dass der Typ der Elemente etwa `Int` ist. Dies wäre eine willkürliche und restriktive Wahl.

Java-Programmierern blieb die Möglichkeit auf `Object` zurückzugreifen, dem Super-Typ aller Objekte. Diese Lösung ist jedoch bei weitem nicht ideal, da sie nicht für primitive Typen (`int`, `long`, `float`, u.a.) funktioniert. Weiter impliziert diese Lösung, dass viele dynamischen Typumwandlungen durch den Programmierer vorgenommen werden müssen.

In Scala ist es möglich, generische Klassen (und Methoden) zu definieren, um dieses Problem zu lösen. Betrachten wir dies an einem Beispiel einer einfachen Container-Klasse: ein Referenz, die entweder leer ist oder auf ein Objekt eines bestimmten Typs verweist.

```
class Reference[T] {
  private var contents: T = _

  def set(value: T) { contents = value }
  def get: T = contents
}
```

Die Klasse wird mit einem Referenz-Typ `T`, der den Typ des Elementes bezeichnet, parametrisiert. Dieser Typ wird im Rumpf der Klasse als Typ der Variablen `contents`, als Typ des Arguments der `set`-Methode und als Rückgabetyp der `get`-Methode verwendet.

Das obige Programmbeispiel verwendet Variablen in Scala, was an sich keiner weiteren Erläuterung bedarf. Es ist jedoch interessant zu sehen, dass der ursprüngliche

Wert, der dieser Variablen gegeben wird, _ ist. Dies steht für den Default-Wert. Dieser Default-Wert ist 0 für numerische Typen, **false** für den Boolean-Typ, () für den Unit-Typ und **null** für alle Objekttypen.

Zur Verwendung dieser Referenz-Klasse muss man angeben, welchen Typ man für den Typ-Parameter T verwenden will. Also von welchen Typ das Elements des Container sein soll. Um zum Beispiel einen Container mit einem Int-Wert zu verwenden, würde man folgendes schreiben:

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

Wie aus diesem Beispiel ersichtlich, ist es nicht notwendig, den von der get-Methode zurückgegebenen Wert zu casten, um ihn als Integer zu verwenden. Es ist natürlich auch nicht möglich, etwas anderes als einen Integer im Container zu speichern.

9 Schlussbemerkung

Dieses Dokument gibt einen kurzen Überblick über die Sprache Scala und präsentiert einige grundlegende Beispiele. Der interessierte Leser kann im Begleitdokument *Scala By Example* weitere fortgeschrittene Beispiele finden. Zur weiteren Lektüre sei die *Scala Language Specification* empfohlen.