Scala-Cheatsheet

Syntaktische Konstrukte

Name	VL	Beschreibung	Beispiel
Listengeneratoren	10	Generiert eine Liste. Die Liste enthält für alle Werte im kartesischen Produkt der Generatoren, für die der Wächter erfüllt ist, den durch `yield` angegebenen Ausdruck.	`for (x <- 0 to 10 if (x % 2 == 0)) yield x`

Mustervergleich	10	Vergleicht eine angegebene Instanz mit vorgegebenen Mustern. Die Anweisungen des ersten passenden Musters (von oben aus betrachtet) werden ausgeführt. Zusätzliche Angabe eines Wächters (guard) möglich.	`x match {`
			`case Nil => 0`
			`case h::ts if (h < 0) => 4`
			`case h::ts => 7 }`

Sektionen	10	Kurzschreibweise für (i. d. R. binäre) Infix-Operatoren.	`_ + 1`
Faule Auswertung	11	Wertet die rechte Seite der Definition nur aus, sobald es zwingend für die Programmausführung notwendig ist.	`lazy val x = List.range(1, 1000).product`

Algebraische Datentypen	11	Datentypen bzw. Objekte, die für Mustervergleich verwendet werden können.	`abstract class Tree`
			`case object Empty extends Tree`
			`case class Node(l: Tree, v: Int, r: Tree)`
			`extends Tree`

Generische Definitionen	10/11	Typparameter werden mit eckigen Klammern, Kovarianz durch ein + angezeigt.	`class C[+T](value: T)`

Partielle Fkt.-Auswertung	11	Funktionen mit mehreren Parameterlisten können als Funktionen, die andere Funktionen zurückgeben, betrachtet werden.	`def f: Int => Int => Int = a => b => a-b`
			`val g = f(10)`
			`g(6) == 4`

Tupel

Name	VL	Beschreibung	Beispiel
`(`x`, ... )`	10	Erzeugt ein Tupel bestehend aus den angegebenen Komponenten (maximal 22 Komponenten).	`(4, 5)`
`_1, _2, ...`	10	Liefert die n-te Komponente eines Tupels.	`(4, 5, 6)._2 == 5`

Listenfunktionen

Name	VL	Beschreibung	Beispiel
`range`	10	Erzeugt eine Liste mit allen Elementen zwischen der unteren (einschließlich) und der oberen (ausschließlich) Grenze.	`List.range(4, 7) == List(4, 5, 6)`

Listenmethoden

Name	VL	Beschreibung	Beispiel
`::`	10	Liefert eine Liste bestehend aus einem Kopfelement und einer Restliste.	`4 :: List(5, 6) == List(4, 5, 6)`
`:::`	10	Liefert eine Liste, bei der linke und rechte Liste zusammengehängt wurden.	`List(4, 5) ::: List(5, 6) == List(4, 5, 5, 6)`
`apply`	10	Liefert das n-te Element einer Liste.	`List(4, 5, 6)(2) == 6`
`head`	10	Liefert das erste Element einer Liste, sofern vorhanden.	`List(4, 5, 6).head == 4`
`tail`	10	Liefert die Elemente einer Liste ohne das erste Element.	`List(4, 5, 6).tail == List(5, 6)`
`last`	12	Liefert das letzte Element einer Liste, sofern vorhanden.	`List(4, 5, 6).last == 6`
`take`	10	Liefert die ersten n Elemente einer Liste.	`List(4, 5, 6).take(2) == List(4, 5)`
`drop`	10	Liefert eine Liste ohne die ersten n Elemente.	`List(4, 5, 6).drop(2) == List(6)`
`length`	10	Liefert die Länge einer Liste.	`List(5, 6, 7).length == 3`
`sum`/`product`	10	Liefert die Summe/das Produkt der Elemente einer Liste.	`List(4, 5, 6).sum == 15`
`reverse`	10	Liefert eine Liste, deren Elemente in umgekehrter Reihenfolge stehen.	`List(4, 5, 6).reverse == List(6, 5, 4)`
`zip`	10	Fügt die Elemente zweier Listen paarweise zusammen.	`List(4, 5, 6).zip(List('A', 'B')) == List((4, 'A'), (5, 'B'))`
`filter`	10, 11	Liefert die Elemente einer Liste, die die gegebene Bedingung erfüllen.	`List(4, 5, 6).filter(_%2 == 0) == List(4, 6)`

`map`	11	Wendet die gegebene Funktion auf jedes Element der Liste an.	`List(4, 5, 6).map(_+1) == List(5, 6, 7)`
`takeWhile`	11	Wählt Elemente aus einer Liste aus, solange die gegebene Funktion zutrifft.	`List(4, 5, 6).takeWhile(_%2==0) == List(4)`
`dropWhile`	11	Wählt diejenigen Elemente aus einer Liste aus, die `takeWhile` nicht zurückliefern würde.	`List(4, 5, 6).dropWhile(_%2==0) == List(5, 6)`
`forall`	11	Prüft, ob die gegebene Bedingung für alle Elemente zutrifft.	`List(4, 5, 6).forall(_%2==0) == false`
`exists`	11	Prüft, ob die gegebene Bedingung für mindestens ein Element zutrifft.	`List(4, 5, 6).exists(_%2==0) == true`
`foldRight`	11	Wendet eine Reduktion von rechts nach links auf die Elemente einer Liste an. Für einen Startwert s, eine Funktion f und Listenelemente $x_1, \ldots, x_n$ wird also $f(x_1, f(\ldots f(x_n, s)))$ berechnet.	`List(4, 5, 6).foldRight(0)(_-_) == 5`
`foldLeft`	11	Wie `foldRight`, aber von links nach rechts. Für einen Startwert s, eine Funktion f und Listenelemente $x_1, \ldots, x_n$ wird also $f(f(\ldots f(x_1, s), \ldots), x_n)$ berechnet.	`List(4, 5, 6).foldLeft(0)(_-_) == -15`
`min`/`max`	12	Wählt das minimale/maximale Element einer Liste aus.	`List(5, 4, 6, 5).min == 4`
`flatMap`	13	Wie `map`, allerdings muss die gegebene Funktion eine Sammlung von Elementen (z. B. eine Liste) liefern. Diese Listen werden dann verkettet.	`List(4, 5, 6).flatMap(x => (1 to x-1) .filter(x%_==0)) == List(1, 2, 1, 1, 2, 3)`

Stromfunktionen

Name	VL	Beschreibung	Beispiel
`iterate`	11	Erzeugt einen unendlichen Strom durch wiederholte Anwendung einer Funktion beginnend bei einem Startwert.	`Stream.iterate(4)(_ + 1).take(3).toList == List(4, 5, 6)`

Strommethoden

Name	VL	Beschreibung	Beispiel
`#::`	11	Liefert einen Strom bestehend aus einem Kopfelement und einem Reststrom.	`4 #:: Stream(5, 6) == Stream(4, 5, 6)`
`#:::`	11	Liefert einen Strom, bei der linker und rechter Strom zusammengehängt wurden.	`Stream(4,5) #::: Stream(5,6) == Stream(4, 5, 5, 6)`

`apply`
`head`
`tail`
`take`
`drop`
`zip`
`filter`
`map`
`takeWhile`
`dropWhile`
`flatMap`
		Siehe Listenmethoden.	

`length`
`sum`
`product`
`forall`
`exists`
`foldRight`
`foldLeft`
`min`
`max`
		Für endliche Streams wie die entsprechenden Listenmethoden. Für unendliche Streams terminiert die Auswertung nicht (oder nicht notwendigerweise).	

`toList`	11	Wandelt endlichen Strom in Liste um (terminiert nicht bei unendlichen Streams).	`Stream(4, 5, 6).toList == List(4, 5, 6)`

Parallelisierung

Name	VL	Beschreibung	Beispiel
`par`	12	Gibt eine Sicht auf die Daten des entsprechenden Container-Objekts zurück, deren Operationen unter Umständen parallelisiert sind.	`List.range(0, 1000).par.map(_ * 2).toList`
`Future`	12	Startet eine nebenläufige Berechnung.	`val fut = Future { foo() }`
`Await.result`	12	Wartet auf Beendigung einer nebenläufigen Berechnung.	`Await.result(fut, Duration.Inf)`