1 Einleitung

Mit der Version 8 wurden auch in Java Lambda-Ausdrücke eingeführt. Auf den ersten Blick erscheinen diese neben den vielen anderen Sprachfeatures von Java nur als ein weiterer kleiner Zusatz. Tatsächlich bedeutet aber ihre Einführung und die damit einhergehende Unterstützung eines funktionalen Programmierstils einen revolutionären Wandel in der Art, wie man Programme in Java gestalten kann. Funktionale Programmierung verspricht eine Programmgestaltung, die in deklarativer Form, auf hohem Abstraktionsniveau, knapp und präzise und für eine parallele Ausführung geeignet ist. Funktionale Programmierung ist grundsätzlich unterschiedlich zur imperativen Programmierung, sie steht aber nicht im Gegensatz zur objektorientierten Programmierung. Wie Brian Goetz es in seinem Vorwort zu [56] ausdrückt, arbeitet Objektorientierung mit einer Abstraktion entlang der Daten, die funktionale Programmierung erlaubt aber eine Abstraktion von Verhaltensstrukturen. Damit ergänzen sich die beiden Paradigmen zu einer neuen Qualität der Programmgestaltung.

Funktionale Programmierung hat besonders in den letzten Jahren viel Aufmerksamkeit erhalten. Dabei ist funktionale Programmierung kein neues Paradigma. Tatsächlich ist die erste funktionale Programmiersprache Lisp fast so alt wie die erste imperative Programmiersprache Fortran. Die Entwicklung der theoretischen Grundlagen zur funktionalen Programmierung, der Lambda Kalkül [17], reicht sogar bis in die 1930er-Jahre zurück. Heute wird funktionale Programmierung von den meisten Programmiersprachen unterstützt.

Mehrere Entwicklungen sind wohl dafür verantwortlich, dass die funktionale Programmierung, die über so viele Jahre ein mehr oder weniger akademisches Nischendasein fristete, jetzt vom Mainstream der Programmierwelt aufgegriffen wurde:

- Mit dem Aufkommen von Prozessoren mit mehreren Kernen werden Softwarekonzepte für eine parallele Ausführung benötigt.
- Mit der zunehmenden Vernetzung, mit Systemen, die ständig online verbunden sind, mit Serverarchitekturen, die zehntausende Klienten gleichzeitig bedienen müssen, und Internet of Things-Anwendungen, die ständig auf Ände-

rungen in der Umwelt reagieren sollen, versagen bisherige Paradigmen der Programmierung.

Und gerade die funktionale Programmierung wird als Mittel gesehen, diesen Herausforderungen zu begegnen.

Eigenschaften funktionaler Programme

Funktionale Programmierung beruht auf mathematischen Funktionen. In der Mathematik ist eine Funktion eine Abbildung einer Menge von Elementen des Definitionsbereichs *D* in eine weitere Menge von Elementen des Wertebereichs *Z*:

$$f : D \rightarrow Z, x \mapsto y$$

Jedem Element des Definitionsbereichs wird ein *eindeutiger* Wert des Wertebereichs zugeordnet. Praktisch heißt dies, dass eine Funktion bei gleichem Argumentwert immer den gleichen Ergebniswert liefern muss. Eine Funktion kann damit weder ein Gedächtnis haben, noch darf sie Seiteneffekte verursachen. Der einzige Mechanismus, der durch eine Funktion zur Verfügung gestellt wird, ist die Anwendung der Funktion auf Argumente, für die die Funktion ein eindeutiges Resultat liefert. Man spricht dann von *rein-funktionaler* Programmierung.

Eine rein-funktionale Programmierung beruht daher ausschließlich auf Funktionsdefinition und Funktionsanwendung. Die so vertrauten Konzepte der veränderlichen Variablen und Wertzuweisungen gibt es nicht, tatsächlich gibt es überhaupt keine veränderlichen Daten. Funktionale Programmierung unterscheidet sich damit grundsätzlich von imperativer Programmierung, die immer mit Zuweisungen von Werten an Variablen und Verändern eines Programmzustands arbeitet.

Funktionale Programme haben im Vergleich zu Programmen mit Seiteneffekten eine Reihe von günstigen Eigenschaften. Eine Funktion ist in sich abgeschlossen, weil sie nur von den Argumenten abhängt. Sie kann daher als eine Einheit angewendet, verstanden, getestet und verifiziert werden. Eine Funktionsanwendung steht ausschließlich für den Wert, den sie berechnet, und sie kann folglich zu jedem Zeitpunkt durch seinen Wert ersetzt werden. Das macht funktionale Programme unabhängig von einem Kontrollfluss und Funktionsanwendungen können in beliebiger Reihenfolge, parallel oder nach Bedarf erfolgen.

Aber ist dieses Modell, mit Funktionen mit Rückgabewerten und mit Funktionsanwendungen zu arbeiten, nicht viel zu restriktiv? Ist funktionale Programmierung damit im Vergleich zur imperativen Programmierung weniger mächtig und flexibel? Wie im wegweisenden Artikel von John Hughes mit dem Titel »Why functional programming matters« [32] eindrucksvoll argumentiert wird, liegt die Mächtigkeit der funktionalen Programmierung in der besseren Modularität und Kombinierbarkeit von Funktionen. Dadurch, dass Funktionen in sich abgeschlossen und nicht von einer Umgebung abhängig sind, können Funktionen frei kombiniert werden.

Besonders bemerkenswert ist auch, dass gerade John Backus, der mit der Entwicklung von Fortran [7] und mit seiner Mitarbeit bei der Definition von Algol 60 [6] ganz wesentlich zur Entwicklung der imperativen Programmiersprachen beigetragen hat, zu den Schwächen der imperativen Programmierung und zur Mächtigkeit der funktionalen Programmierung sehr früh grundlegende Einsichten gab. In seiner Ansprache zur Verleihung des Turing Awards hat er sich mit sehr drastischen Worten gegen die imperative Programmierung gewandt und eine funktionale Programmierung propagiert. Aus der begleitenden Publikation mit dem vielsagenden Titel »Can Programming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs« [8] ist dazu folgende Aussage entnommen:

> *Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor–the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.*

Backus sieht also die Schwäche der imperativen Programmierung in der engen Anlehnung an das Verarbeitungsmodell des von Neumann-Rechners. Und er vermisst mächtige Kombinationsoperatoren zur Gestaltung von Programmen auf höherer Ebene. Der Artikel propagiert dann einen funktionalen Programmierstil mit Operatoren zur Kombination von funktionalen Bausteinen. Er schreibt dazu:

> *An alternative functional style of programming is founded on the use of combining forms for creating programs. [...] Associated with the functional style of programming is an algebra of programs whose variables range over programs and whose operations are combining forms.*

Backus hat damit früh den Weg der Entwicklung funktionaler Sprachen vorgezeichnet. Sehr anschaulich ist auch das Beispiel im Artikel, mit dem imperative Programmierung und funktionale Programmierung verglichen werden. Eine imperative Lösung des Skalarproduktes von Vektoren, folgend mit zwei Listen `a` und `b` mit Länge `n`, gestaltet sich in Java folgendermaßen:

```
int c = 0;
for (int i = 0; i < n; i++) {
  c = c + a.get(i) * b.get(i);
}
```

Die Lösung baut auf der Wertzuweisung auf. Das Ergebnis wird in der Variablen `c` akkumuliert. In jedem Schleifendurchlauf wird in einer Wertzuweisung ein

nächster Wert für `c` berechnet und gespeichert. Die Berechnung ist, wie Backus es kritisiert, *»word-at-a-time style«*.

Die im Artikel dann vorgeschlagene Lösung, übersetzt auf die heute übliche Form funktionaler Sprachen, sieht im Gegensatz dazu folgendermaßen aus:

```
map2((x, y) -> x * y).andThen(reduce((r, x) -> r + x))
```

Die Operationen beziehen sich nicht auf einzelne Elemente, sondern auf die Vektoren als Ganzes. Sie sind mit Verknüpfungsoperationen in der Form von Lambda-Ausdrücken parametrisiert. Mit `map2` werden die Elemente zweier Vektoren paarweise mit der angegebenen Verknüpfungsoperation verknüpft. Die Operation `reduce` verknüpft die Elemente eines Vektors zu einem einzelnen Wert. Mit `andThen` werden Funktionen in Serie geschaltet. Somit werden in obiger Definition zuerst die Elemente von zwei Vektoren multipliziert und schließlich alle Produkte addiert. Das entspricht exakt der mathematischen Definition des Skalarprodukts. Im Gegensatz zur obigen imperativen Lösung ist diese Definition deklarativ und auf hoher Ebene. Die Operationen beziehen sich nicht auf einzelne Elemente, sondern auf die Datenobjekte selbst. Die Operationen haben keine Seiteneffekte und sind rein-funktional.

In diesem Beispiel sehen wir auch bereits die wesentlichen Konzepte, die funktionale Programmierung kennzeichnen:

- *Aggregatsfunktionen*: Funktionen, die auf komplexen Datenobjekten als Ganzes operieren und diese ohne versteckte Seiteneffekte in neue Strukturen abbilden.
- *Funktionen höherer Ordnung*: Funktionen, die durch andere Funktionen parametrisiert sind.
- *Kombination von Funktionen*: Operatoren, die eine Verknüpfung von funktionalen Bausteinen zu größeren funktionalen Einheiten erlauben.

Zur Entwicklung funktionaler Programmiersprachen

Die erste funktionale Sprache Lisp wurde von John McCarthy als eine Implementierung des Lambda-Kalküls entwickelt und bereits im Jahre 1960 veröffentlicht [51]. Wenn man bedenkt, dass Fortran als erste imperative höhere Sprache von John Backus 1957 eingeführt wurde, ist funktionale Programmierung also nur rund drei Jahre jünger. Bemerkenswert ist auch, dass diese erste Version von Lisp bereits Lambda-Ausdrücke, Funktionen höherer Ordnung und Listenverarbeitung hatte, Dinge die fast 60 Jahre später wieder sehr aktuell sind.

Seit dieser frühen Zeit wurde die funktionale Programmierung stetig weiterentwickelt. Lisp hat sich über viele Versionen und Varianten in den 1980er- und Anfang der 1990er-Jahre als die Sprache der Künstlichen Intelligenz etabliert. Die Sprachvarianten CommonLisp [81] und Scheme [83] werden auch heute noch verwendet, und mit der Sprache Clojure [44] gibt es einen populären Nachfolger, der auf der Java VM läuft.

Lisp-Sprachen sind dynamisch typisiert, das heißt, Werte tragen ihren Typ, und die Typprüfung wird zur Laufzeit durchgeführt. Robin Milner hat mit der Einführung der Sprache ML den Grundstein für statisch typisierte funktionale Sprachen gelegt [53]. Bei dynamisch-typisierten Sprachen können Funktionen mit beliebigen Datentypen arbeiten. Milner erkannte, dass man bei statischer Typisierung eine vergleichbare Flexibilität mit Typparametern erreichen kann. Gleichzeitig wurde ein Typinferenz-Algorithmus zur statischen Typprüfung eingeführt. In Standard ML, einer Weiterentwicklung von ML, wurden basierend auf der Sprache HOPE [12] polymorphe algebraische Datentypen verwendet, die auch heute noch die Basis für die Typsysteme funktionaler Sprachen bilden.

Ein weiterer wichtiger Entwicklungsschritt der funktionalen Programmierung war beginnend mit den frühen 1980er-Jahren die Erforschung von Sprachen mit nicht-strikten Auswertungsverfahren [85][86]. Bei strikten Auswertungsverfahren, wie wir diese von Java und anderen Programmiersprachen kennen, werden die Ausdrücke für die Parameter vor einem Prozeduraufruf evaluiert und die Werte übergeben. Bei einer nicht-strikten Auswertung [67] können die Ausdrücke vorerst nichtevaluiert übergeben werden, um schließlich im Kontext der aufgerufenen Funktion evaluiert zu werden. Dies führte zur Einführung von nicht-strikten funktionalen Programmiersprachen, wie zum Beispiel Miranda [87], und zur Entwicklung von effizienten Ausführungskonzepten [46][65].

Wie in [31] zu lesen ist, gab es um 1987 eine Vielzahl von Sprachentwicklungen mit nicht-strikten Auswertungsverfahren, die ganz ähnliche Ziele verfolgten. Eine Gruppe von Forschern beschloss daher, als Grundlage für Forschung und Entwicklung funktionaler Programmierkonzepte einen gemeinsamen Sprachentwurf zu schaffen. Damit wurde die Sprache Haskell geboren und der Sprachentwurf Haskell 1.0 im Jahre 1990 vom Haskell Committee veröffentlicht [30]. Haskell ist heute der State of the Art für funktionale Programmierung. Die funktionalen Programmierkonzepte, die wir in Java heute sehen, haben im Wesentlichen ihr Vorbild in Haskell. Um für dieses Buch einen entsprechenden Hintergrund zu schaffen, werden wir daher im folgenden Abschnitt die elementaren Konzepte und die wichtigsten Begriffe der funktionalen Programmierung, wie sie sich in Haskell darstellen, einführen.

1.1 Elementare Konzepte und Begriffe

Die wichtigsten Konzepte der funktionalen Programmierung sind: Lambda-Ausdrücke und Funktionsobjekte, polymorphe algebraische Datentypen mit Typinferenz, Pattern Matching sowie nicht-strikte Bedarfsauswertung. Diese Konzepte werden im Folgenden kurz eingeführt. Im Laufe des Buches werden wir uns intensiv mit der Umsetzung dieser Konzepte in Java beschäftigen.

Lambda-Ausdrücke und Funktionen höherer Ordnung

Lambda-Ausdrücke repräsentieren Funktionen mit formalen Argumenten und einem definierenden Ausdruck. Die beiden Seiten sind üblicherweise mit einem Pfeilsymbol[1] getrennt. Der definierende Ausdruck legt den Funktionswert fest und hat keine Seiteneffekte. Folgendes Beispiel zeigt einen Lambda-Ausdruck mit Argument `x` und definierendem Ausdruck `x + 1`:

```
x -> x + 1
```

Lambda-Ausdrücke sind in Programmiersprachen Literale für die Erzeugung von *Funktionsobjekten*. Funktionsobjekte können wie andere Werte an Variablen zugewiesen, als Parameter übergeben und Ergebnis von Funktionen sein. Im Sprachgebrauch der funktionalen Programmierung sagt man, Funktionsobjekte sind *first class*. Mit diesem einfachen Prinzip lassen sich Funktionen mit Funktionen als Parameter bilden, sogenannte *Funktionen höherer Ordnung*[2]. Ebenso kann man Funktionen schreiben, die aus einfacheren Funktionen komplexere Funktionen erzeugen. Zum Beispiel kann man eine Funktion `andThen` definieren, die aus zwei Funktionen `f` und `g` eine Funktion erzeugt, die zuerst auf das Argument `x` die Funktion `f` und auf das Ergebnis die Funktion `g` anwendet:

```
andThen f g = x -> g (f x)
```

Bei den Variablen, die im definierenden Ausdruck eines Lambda-Ausdrucks vorkommen, unterscheidet man *gebundene* und *freie* Variablen. Gebundene Variablen sind Argumente des Lambda-Ausdrucks, freie Variablen sind nicht Argument des Lambda-Ausdrucks und müssen damit im Kontext des Lambda-Ausdrucks gebunden sein. Im folgenden Lambda-Ausdruck ist `x` gebunden und `y` frei:

```
x -> x + y
```

Freie Variablen sind für die Bildung von Funktionsobjekten kritisch und wir werden sie im Abschnitt 2.3.4 über Closures noch ausführlich behandeln.

Algebraische Datentypen

Das Typsystem der statisch-typisierten funktionalen Sprachen beruht auf *Algebraischen Datentypen*. Algebraische Datentypen orientieren sich an Konzepten der Mengentheorie und erscheinen in einer abstrakteren Form als die Datentypen imperativer und objektorientierter Sprachen. Sie bauen auf Mengendefinitionen durch Enumeration und dem Kartesischen Produkt auf. Durch Kombination von Enumeration und Produkttypen ergeben sich sogenannte *Variantentypen* (engl.:

1. Im Lambda-Kalkül ist einem Lambda-Ausdruck der griechische Buchstabe λ vorangestellt und statt des Pfeils wird üblicherweise ein Punkt verwendet. Daher würde man den obigen Lambda-Ausdruck im Lambda-Kalkül als $\lambda\, x\, .\, x + 1$ schreiben.
2. Wir werden Funktionen höherer Ordnung auch einfach als *höhere Funktionen* bezeichnen.

v*ariant types* oder *union types*). Folgendes Beispiel eines Datentyps `Expr` zur Repräsentation von Booleschen Ausdrücken zeigt das Prinzip. Der Typ `Expr` definiert Varianten `Lit`, `Var`, `Not`, `And` und `Or` (die Varianten sind durch Oder-Striche getrennt). Jede Variante ist ein Produkttyp und besteht aus dem Namen der Variante, dem sogenannten *Datentag*, und den Typen für die Felder:

```
data Expr =
    Lit Bool      |
    Var String    |
    Not Expr      |
    And Expr Expr |
    Or Expr Expr
```

Die obige Datendefinition ist zudem rekursiv, da die Varianten `Not`, `And` und `Or` Felder vom Typ `Expr` haben. Auf diese Weise können hierarchische Strukturen gebildet werden.

Wir werden dieses Beispiel zur Repräsentation Boolescher Ausdrücke mehrmals in den Fallbeispielen zur funktionalen Programmierung in Java wieder aufgreifen und mit obiger Definition eines Algebraischen Datentyps vergleichen.

Parametrischer Polymorphismus

Typparameter dienen zur Definition von Funktionen und Datentypen, bei denen bestimmte Typen unbestimmt sind. Man spricht in der funktionalen Programmierwelt, von wo das Konzept auch ursprünglich stammt [53][13], von einem *parametrischen Polymorphismus*. Bei Java und anderen Sprachen ist das Konzept unter dem Begriff *Generizität* bekannt. Generizität ist für eine funktionale Programmierung von besonderer Bedeutung und wir werden daher bei den sprachlichen Grundlagen zur funktionalen Programmierung im nächsten Kapitel das Thema Generizität von Java detailliert behandeln.

Typinferenz

Funktionale Programmiersprachen verwenden *Typinferenz*. Darunter versteht man, dass der Typ eines Ausdrucks einer Variablen oder die Signatur einer Funktion automatisch abgeleitet wird und nicht vom Programmierer angegeben werden muss. Bei Lambda-Ausdrücken werden üblicherweise die Typen der Argumente und des Rückgabewertes nicht angegeben, sondern durch Typinferenz abgeleitet. Zum Beispiel sind bei unserem einfachen Lambda-Ausdruck

```
x -> x + y
```

keine Typen für den Parameter `x`, die freie Variable `y` oder den Rückgabewert angegeben. Darüber hinaus ist der Operator `+` überladen und somit die Typen der Operanden nicht eindeutig. In Java könnten die Typen für Argument und Ergebnis beliebige Zahlen als auch Zeichenketten sein.

Bei imperativen Sprachen wird üblicherweise nur für Ausdrücke der Typ durch den Compiler automatisch bestimmt, aber nicht für Variablen und Funktionsdefinitionen. Bei funktionalen Sprachen ist das aber auch für Variablen und Funktionsdefinitionen möglich. Der sogenannte Hindley-Milner-Algorithmus [53][27] ist in der Lage, für einen beliebigen Ausdruck und Funktionsdefinitionen den allgemeinsten Typ abzuleiten, der alle gegebenen Einschränkungen erfüllt.

Mit der Einführung von Lambda-Ausdrücken wurde die automatische Typinferenz in Java wesentlich erweitert und wir werden diese im nächsten Kapitel bei den Themen Generizität und Lambda-Ausdrücke diskutieren.

Funktionale Datenstrukturen

Eine wesentliche Eigenschaft der Algebraischen Datentypen ist, dass sie grundsätzlich *unveränderlich* sind. Auch Datenstrukturen wie Listen, Mengen oder Maps sind unveränderlich. Man spricht von *funktionalen* oder *persistenten Datenstrukturen*[3].

Die bekannteste funktionale Datenstruktur ist die funktionale Liste. In Haskell ist diese als Algebraischer Datentyp folgendermaßen definiert:

```
data [a] = []  |  a : [a]
```

Der Typbezeichner ist `[a]` mit dem Typparameter `a`. Die Variante `[]` bezeichnet die leere Liste. Die Variante `:` ist eine Liste mit dem ersten Element vom Typ a und der Restliste vom Typ [a] (der Datentag `:` ist dabei in Infix-Notation angegeben). Listen sind also rekursive Strukturen mit einem Element und einer Restliste, wobei am Ende immer die leere Liste steht. Man kann damit durch Voranstellen von Elementen aus bestehenden Listen neue Listen erzeugen, aber bestehende Listen nicht verändern. Für weitere persistente Datenstrukturen gibt es sehr effiziente Verfahren, die zum Beispiel auch einen direkten Zugriff ermöglichen. In [61] ist eine umfassende Behandlung dieses Themas zu finden.

In diesem Buch werden wir mit einer persistenten Listenimplementierung analog zum obigen Algebraischen Datentyp arbeiten. Wir führen diese in Abschnitt 3.3 ein, erweitern sie im Laufe der folgenden Kapitel und verwenden sie bei mehreren Fallbeispielen.

Pattern Matching

Eng verbunden mit den Algebraischen Datentypen ist *Pattern Matching*. Es handelt sich dabei um eine Kontrollstruktur, mit der auf Basis von Mustern die Varianten eines Datentyps unterschieden werden. Stimmt ein gegebener Wert mit einem Muster überein, ergibt sich das Ergebnis durch den dazugehörigen Ausdruck. Muster werden analog zu den Konstruktoren der Varianten gebildet. Mit

3. Wir werden die Bezeichnungen unveränderliche, persistente und funktionale Datenstruktur synonym verwenden.

dem Datentag erfolgt zuerst die Unterscheidung der Varianten. Für die Felder können Werte oder Variablen angegeben werden. Bei Werten erfolgt ein Vergleich auf Gleichheit, bei Variablen wird der Feldwert an die Variable gebunden.

Betrachten wir dazu wieder unseren Algebraischen Datentyp `Expr` von oben. Folgende Haskell-Funktion `mkString` verwendet Pattern Matching, um eine String-Repräsentation für einen `Expr`-Wert zu erstellen:

```
mkString e =
  case (e) of
    Lit True       -> "true"
    Lit False      -> "false"
    Var name       -> name
    Not sub        -> "(! " ++ mkString(sub) ++ ")"
    And left right -> "(" ++ mkString(left) ++ " && " ++
                      mkString(right) ++ ")"
    Or left right  -> "(" ++ mkString(left) ++ " || " ++
                      mkString(right) ++ ")"
```

Mit einem `case`-Ausdruck werden für einen `Expr`-Wert `e` die möglichen Varianten unterschieden. Man sieht, dass die Muster in den Fallunterscheidungen exakt den Varianten der Datentypdefinition entsprechen. Des Weiteren entsprechen die rekursiven Aufrufe der rekursiven Definition des Datentyps. Ein großer Vorteil von Pattern Matching ist, dass mit dem Mustervergleich nicht nur die Varianten unterschieden, sondern die Feldwerte den Pattern-Variablen zugewiesen werden. Das Datenelement wird sozusagen in die Bestandteile zerlegt. So werden zum Beispiel bei den Varianten `And` und `Or` die beiden Unterausdrücke an die Variablen `left` und `right` gebunden.

In Java gibt es Pattern Matching leider nicht. Wir werden aber sehen, dass man mit Lambda-Ausdrücken Strukturen ähnlich zu Pattern Matching schaffen kann.

Nicht-strikte Auswertung

Unter einer Evaluierungsstrategie versteht man die Reihenfolge, in der Ausdrücke ausgewertet werden. Aus der imperativen und objektorientierten Programmierung kennen wir ausschließlich die *strikte* Evaluierung. Es werden dabei die Ausdrücke für die Argumente von Funktionsanwendungen zuerst von links nach rechts vollständig evaluiert und dann die Funktion mit den konkreten Ergebniswerten aufgerufen. Dies entspricht einer *call-by-value* Parameterübergabe. Eine strikte Evaluierung ist bei Programmen mit Seiteneffekten auch unvermeidlich, da hier die Reihenfolge der Evaluierung der Ausdrücke und Anweisungen Einfluss auf das Ergebnis haben kann.

Arbeiten wir aber mit reinen Funktionen, hat die Reihenfolge der Funktionsanwendungen keinen Einfluss auf das Ergebnis. In der funktionalen Programmierung kann man daher auch *nicht-strikte* Evaluierungsstrategien einsetzen. Haskell ver-

wendet grundsätzlich eine nicht-strikte Evaluierungsstrategie, die sogenannte *Bedarfsauswertung* (engl.: *lazy evaluation*, *call-by-need*) [65]. Bei der Bedarfsauswertung wird ein Ausdruck erst und nur dann evaluiert, wenn das Ergebnis gebraucht wird.

Eine Bedarfsauswertung hat zum Beispiel Vorteile, wenn unendliche oder sehr große Strukturen verarbeitet werden. Das folgende Beispiel demonstriert das Arbeiten mit unendlichen Listen. In Haskell definiert der Ausdruck `[1000..]` die unendliche Liste der Integers ab dem Wert 1000. Mit dem Ausdruck

```
head (filter isPrime [1000..])
```

wird die erste Primzahl aus der Liste ermittelt. Dazu werden aber von der unendlichen Liste nur die Zahlen 1000 bis zur ersten Primzahl 1009 benötigt und somit auch nur diese generiert.

Java verwendet wie gesagt ausschließlich die strikte Evaluierung. Nicht-strikte Evaluierung kann man aber, wie wir in Abschnitt 4.6 zeigen werden, mit Funktionsparametern nachbilden und damit auch unendliche Strukturen schaffen. Und auch die funktionalen Streams in Java, die wir in Kapitel 7 behandeln, arbeiten nach dem Prinzip der nicht-strikten Bedarfsauswertung.

1.2 Funktionale Programmierung in Java

Die Erweiterungen von Java zur funktionalen Programmierung basieren auf den elementaren Konzepten, wie wir diese im letzten Abschnitt besprochen haben. Allerdings wird zurzeit nur ein Teil der Konzepte unterstützt. So gibt es in Java weder Algebraische Datentypen noch Pattern Matching. Auf der anderen Seite stehen die Konzepte zur funktionalen Programmierung nicht isoliert, sondern sind nahtlos in die objektorientierte Welt integriert. Funktionen werden durch Java-Objekte repräsentiert und können damit mit den Mitteln der objektorientierten Programmierung behandelt werden. So kann man Funktionen in herkömmlicher Weise mit objektorientierten Methoden erzeugen, aufbauen, umformen und kombinieren.

Allerdings soll auch hier angemerkt werden, dass Java natürlich keine rein-funktionale Sprache und daher ein Programmieren völlig ohne Seiteneffekte nicht sinnvoll ist. Eine solche Programmierung, bei der dann nur rekursive Funktionen möglich wären, wird in diesem Buch auch nicht verfolgt. Im Gegenteil wird man vor allem bei der internen Implementierung von funktionalen Operatoren und Strukturen auf imperative Techniken zurückgreifen, aber möglichst die funktionalen Eigenschaften nach außen bewahren. Auch die wichtigen Java-Bibliotheken, allen voran die Streams, arbeiten gezielt mit Seiteneffekten und veränderlichen Datenstrukturen. Allerdings werden die Seiteneffekte so beschränkt, dass sich Eigenschaften wie rein-funktionale Programme ergeben und damit zum Beispiel eine parallele Ausführung möglich wird.