

Eine Einführung in die Programmiersprache  
Haskell

Christopher Bertels

25.07.2008

Seminar Programmiersprachen SS 2008  
Fachbereich Mathematik/Informatik  
Universität Osnabrück

## Inhaltsverzeichnis

<b>1</b>	<b>Allgemeines</b>	<b>3</b>
1.1	Entstehungsgeschichte Haskells . . . . .	3
1.2	Haskell 98 - Implementationen und Zukunftsperspektiven . . .	3
1.3	Allgemeine Spracheigenschaften . . . . .	5
<b>2</b>	<b>Sprachsyntax</b>	<b>7</b>
2.1	Funktionen - Hauptbestandteil der Sprache . . . . .	7
2.1.1	Beispielfunktionen . . . . .	7
2.2	Currying . . . . .	9
2.3	Datentypen . . . . .	10
2.4	Funktionen höherer Ordnung und Lambda Ausdrücke . . . . .	13
2.4.1	Funktionen höherer Ordnung . . . . .	14
2.4.2	Lambda Ausdrücke . . . . .	16
<b>3</b>	<b>Haskell Spezialitäten</b>	<b>17</b>
3.1	Typklassen . . . . .	18
3.2	Bedarfsauswertung . . . . .	20
3.3	I/O & Monaden . . . . .	21
<b>4</b>	<b>Fazit</b>	<b>25</b>
	<b>Literatur</b>	<b>26</b>

# 1 Allgemeines

## 1.1 Entstehungsgeschichte Haskell

Haskell entstand zwischen 1987 und 1990, ausgehend von der FPCA-Konferenz<sup>1</sup> und dem dort gegründeten Komitee. Ziel des Komitees war es, eine vereinheitlichte, standardisierte rein funktionale Sprache zu entwickeln.

Eines der Hauptziele war, dass die Sprache dem Prinzip der *Lazy Evaluation* folgen sollte, der sog. *Bedarfsauswertung*. Gerade diese Eigenschaft ist bei den meisten (funktionalen) Programmiersprachen nicht vorhanden, bietet aber jede Menge Forschungsraum, und man erhoffte sich, interessante neue Erkenntnisse auf diesem Gebiet zu gewinnen. Als Grundlage sollte dafür ursprünglich die proprietäre rein funktionale Programmiersprache Miranda<sup>2</sup> dienen. Allerdings erklärten sich die Entwickler von Miranda nicht dazu bereit, den Quellcode der Sprache freizugeben<sup>3</sup>, was im Sinne des Komitees eine Grundvoraussetzung für eine erfolgreiche Programmiersprache gewesen wäre, die vor allem in der Forschung Anwendung finden sollte. Daraufhin entschied man sich schließlich dafür, eine neue offene Programmiersprache zu entwickeln (vgl. [6]). Damit war Haskell geboren.<sup>4</sup>

## 1.2 Haskell 98 - Implementationen und Zukunftsperspektiven

Haskell ist eine durch einen Standard definierte Sprache. Dieser wurde 1998 als Report veröffentlicht, 1999 nochmals überarbeitet und liegt seitdem in seiner aktuellen Fassung vor (s. [5]).

Es gibt mehrere Implementationen dieses Standards, darunter die wohl

---

<sup>1</sup> *Conference on Functional Programming Languages and Computer Architecture*; FPCA '87 in Portland, Oregon

<sup>2</sup> Trademark of Research Software Ltd.

<sup>3</sup> Im Sinne von Open Source

<sup>4</sup> *Haskell* ist der Vorname des Logikers Haskell B. Curry (1900-1982), der neben Alonzo Church (1903-1995) fundamentale Grundlagen funktionaler Programmiersprachen mitentwickelt hat.

bekanntesten, namentlich der Glasgow Haskell Compiler (GHC)<sup>5</sup> sowie Hugs<sup>6</sup>. Während Hugs ein reiner Interpreter mit einer interaktiven REPL<sup>7</sup> ist, stellt der GHC einen echten Compiler dar, der Haskell Programmcode in nativen Binärcode umwandelt. Zusätzlich bietet dieser aber auch alle Vorteile eines Interpreters, welcher GHCi heißt, der vergleichbar mit dem Hugs Interpreter ist (vgl. [8], S. 6 f.). Es gibt noch weitere Implementationen, die man allesamt auf der Haskell Website finden kann.<sup>8</sup>

Derzeit wird an einer neuen Haskell Spezifikation gearbeitet, genannt Haskell<sup>9</sup>. Dies ist in Anlehnung an die in Haskell oft verwendete Notation von Funktionennamen, die mit einem Apostroph enden<sup>10</sup>, wie es in der Mathematik auch oft vorzufinden ist.<sup>11</sup>

Haskell' soll neuere Entwicklungen in der Haskell Community wieder spiegeln, einen Großteil der bereits ohnehin verwendeten Bibliotheken in den Sprachstandard einbeziehen sowie neu entwickelte Sprachkonstrukte standardisieren<sup>12</sup> (vgl. [2]).

Vor allem in den letzten Jahren hat Haskell, wie auch viele andere funktionale Programmiersprachen, an öffentlichem Interesse gewonnen. Während diese vor einigen Jahren nur in akademischen Kreisen vorzufinden waren, zeichnet sich seit einiger Zeit ein stetiger Zuwachs an Interessenten auch im kommerziellen bzw. privaten Sektor ab. Grund dafür dürften unter anderem Bestrebungen vieler moderner objekt-orientierter mainstream Programmiersprachen sein, funktionale Elemente mit aufzunehmen.<sup>13</sup> Zusätzlich haben sich in den letzten Jahren vor allem neue Skriptsprachen durchgesetzt, welche zum Teil starke Einflüsse aus den funktionalen Programmiersprachen

---

<sup>5</sup><http://www.haskell.org/ghc/>

<sup>6</sup><http://www.haskell.org/hugs/>

<sup>7</sup>sog. *Read-Eval-Print-Loop*, wie in anderen interpretierten Sprachen üblich (z.B. Ruby, Python, Lisp etc.)

<sup>8</sup><http://www.haskell.org/haskellwiki/Implementations>

<sup>9</sup>engl. *Haskell-Prime*, zu deutsch: *Haskell-Strich*

<sup>10</sup>Beispiele: f und f', g und g'

<sup>11</sup>Vermutlich ist dies eine Anspielung auf die Namensgebung von C++, bei der der typische ++-Operator aus C verwendet wurde

<sup>12</sup>z.B. Concurrency / STM (Software Transactional Memory) uvm.

<sup>13</sup>z.B. C# (Lambda Expressions / Anonymous Functions, Type Inference, Closures ...), Java (Closures)

aufweisen<sup>14</sup>. Letztlich kommt hinzu, dass sich seit kurzem ein fundamentaler Paradigmenwechsel im Bereich der Prozessorarchitekturen vollzieht. Während bis vor kurzem Single-Core Prozessoren der allgemeine Standard waren, zeichnet sich zunehmend der Trend ab, nur noch Multi-Core Prozessoren als Grundlage moderner Computersysteme zu nehmen. Diese Hardwarerevolution wird sich auch in der Art und Weise, wie Software geschrieben wird, widerspiegeln (vgl. [9]). Grund dafür ist die Tatsache, dass heutige imperative Programmiersprachen, die aktuell den überwiegenden Industriestandard darstellen<sup>15</sup>, nicht sehr geeignet für die Entwicklung echt-paralleler Programme sind. Die Komplexität, die mit der Entwicklung von stark nebenläufigen Programmen durch *Locks* und sog. *Conditionvariables* einhergeht, übersteigt bei einer hohen Prozessorzahl die Fähigkeiten, die ein Programmierer an den Tag legen kann. Mit jedem weiteren Prozessorkern steigt die Komplexität der nebenläufigen Programmierung. Da man davon ausgehen kann, dass in einigen Jahren Prozessoren mit mehr als 50 Kernen keine Seltenheit mehr sein werden, scheint ein (zumindest in Teilen) neuer Programmieransatz notwendig zu werden. Das Stichwort zur Lösung des Problems lautet hier *Reinheit*<sup>16</sup> bzw. *Referenzielle Transparenz*<sup>17</sup>, welches ein stark vertretenes Konzept in Haskell ist. Dies ist ein weiterer Grund für das zunehmende Interesse an Haskell und dessen Anwendungsmöglichkeiten.

Wie und wodurch Haskell im Speziellen, und rein funktionale Konzepte im Allgemeinen hierbei neue Perspektiven bieten können, wird im weiteren Verlauf des Textes versucht deutlich zu machen.

### 1.3 Allgemeine Spracheigenschaften

Haskell, wie fast alle funktionalen Programmiersprachen, basiert auf dem Lambda Kalkül<sup>18</sup>, welches ein formelles System zur Berechnung beliebiger Ausdrücke ist. Dabei werden komplexere Ausdrücke durch sog. *Reduktio-*

---

<sup>14</sup>z.B. Ruby, Python, Groovy

<sup>15</sup>Beispiele: C, C++, Java, C# ...

<sup>16</sup>engl. *purity*

<sup>17</sup>bezeichnet die Eigenschaft, dass Funktionen nur von ihren Parametern abhängen (mathematisches Modell einer Funktion)

<sup>18</sup>Mathematisch-Logisches Berechnungskalkül entwickelt von Alonzo Church (1930)

*nen*<sup>19</sup> so lange vereinfacht, bis man bei elementaren Ausdrücken angelangt ist, die dann direkt im System implementiert sind (vgl. [10]). Daraus ergibt sich auch ein fundamentaler Charakter funktionaler Programmiersprachen: Programme lassen sich als Werte betrachten; als Kette von Ausdrücken, die ausgewertet werden und einen Ergebniswert liefern (in Abhängigkeit von den Eingaben bzw. Parametern) (vgl. [3], S. 2 ff.). Dies steht im Widerspruch zum Programmmodell imperativer Programmiersprachen, bei denen ein Programm als Abfolge von Befehlen verstanden wird, die die Maschine abzuarbeiten hat.

Im Gegensatz zu imperativen Sprachen steht also prinzipiell eher das *Was* im Vordergrund und weniger das *Wie*, also die Beschreibung des Problems an sich und nicht die der Problemlösung. Haskell geht diesen Schritt aber im Vergleich zu den meisten funktionalen Programmiersprachen noch ein entscheidendes Stück weiter. Während der Großteil der funktionalen Sprachen imperative Ausdrücke aus pragmatischen Gründen zulässt, verzichtet Haskell darauf. Allerdings nur prinzipiell, da schließlich immer irgendeine Art von sequenzieller Abarbeitung von Befehlen möglich sein muss um Ein- und Ausgabe (I/O) zu bewerkstelligen. Gerade hier muss nämlich eine imperative Abarbeitung stattfinden.<sup>20</sup> Wir werden später noch sehen, wie Haskell mit diesem vermeintlichen Problem, nämlich dem Wunsch, eine rein funktionale Sprache zu sein, aber zugleich Seiteneffekte zu ermöglichen, auf elegante Art und Weise umgeht.

---

<sup>19</sup><http://de.wikipedia.org/wiki/Lambda-Kalk%C3%BCl#Kongruenzregeln>

<sup>20</sup>Schließlich soll z.B. erst eine Datei ausgelesen und dann dessen Inhalt am Bildschirm ausgegeben werden - und nicht anders herum. Die Reihenfolge der Abarbeitung spielt also eine sehr entscheidende Rolle; man möchte dem Computer i.d.R. genau sagen können, was er wann (und wie) tun soll.

## 2 Sprachsyntax

### 2.1 Funktionen - Hauptbestandteil der Sprache

Im Gegensatz zu den meisten imperativen Programmiersprachen bilden Funktionen in Haskell<sup>21</sup> die Basis für ein Programm. Alles baut auf Funktionen auf und neue Funktionen definiert man als eine Komposition von bereits vorhandenen Funktionen (vgl. [8], S. 169 ff.).

Es liegt daher nahe, dass die Syntax es sehr einfach macht Funktionen zu definieren. Während in anderen Sprachen oft Klammerungen notwendig sind, verzichtet man in Haskell im Allgemeinen darauf.<sup>22</sup> Im Folgenden wird Anhand einiger Beispielfunktionen die Syntax Haskells genauer erläutert.

#### 2.1.1 Beispielfunktionen

```
length :: [a] -> Int
length [] = 0
length (x:xs) = length xs + 1
```

Hier haben wir eine Funktion **length** definiert, welche die Länge (als Integer) einer Liste beliebigen Typs zurückliefert. Das erkennt man daran, dass es sich beim Typ der Liste (`[a]`) um ein kleines `a`, einer sog. Typvariablen handelt. Die eckigen Klammern bezeichnen eine Liste vom Typ, der in den Klammern steht. Namen von Datentypen sind per Konvention groß geschrieben, Typvariablen hingegen klein. Im Gegensatz zu z.B. Java nimmt man in Haskell als Typvariablenamen typischerweise `a`, `b`, `c` usw. Eine Typdefinition erkennt man immer an dem `::` Symbol<sup>23</sup>. Die eigentliche Funktionsdefinition ist dafür umso leichter. Es handelt sich in diesem Fall um zwei einfache Gleichungen. Die erste besagt, dass **length** angewandt auf die leere Liste (`[]`) 0 ergibt. Die zweite Gleichung sagt, dass **length** angewandt auf eine Liste, deren Kopf `x` heißt und deren Rest wir mit `xs` bezeichnen die Länge des Restes (`xs`) ist plus 1. Es handelt sich hier also um eine rekursive Definition. Der :

<sup>21</sup>wie auch in jeder anderen funktionalen Sprache

<sup>22</sup>Lediglich zur Auflösung von Uneindeutigkeiten werden Klammern benutzt.

<sup>23</sup>`::` Liest sich als "has type".

Operator ist der Listenkonstruktor womit sich Listen in Kopf und Rest teilen bzw aus ihnen konstruieren lassen. Man sollte dies allerdings nicht mit Konstruktoren in objekt-orientierten Programmiersprachen verwechseln. Auf die Frage, wie man eigene Datentypen definiert, wird später noch eingegangen.

Betrachten wir eine weitere Funktion namens **maximum**. Diese gibt, wie ihr Name schon sagt, das Maximum zweier übergebener ganzer Zahlen zurück.

```

-- kleine maximums Funktion
maximum :: Int -> Int -> Int
maximum x y =
    if x < y then y else x

```

Durch zwei Bindestriche (`--`) werden einzeilige Kommentare eingeleitet. Mehrzeilige Kommentare werden zwischen `{-` und `-}` geschrieben. Natürlich lassen sich diese auch schachteln (wie z.B. in Java). Haskell lässt einem aber relativ viel Offenheit über den Programmierstil. So hätte man die selbe Funktion auch so schreiben können:

```

-- die selbe Funktion, nur mit Guards
maximum :: Int -> Int -> Int
maximum x y
    | x < y      = y
    | otherwise = x

```

Die senkrechten Striche bezeichnet man als “Guards”, wobei von ihnen gefolgt ein Boolescher Ausdruck steht, dessen rechter Teil (hinter dem `=`) ausgewertet wird, falls er wahr ist. Das Schlüsselwort **otherwise** ist immer wahr und wird daher ausgeführt, wenn kein Guard über ihm wahr war. Dieses Prinzip, dass Gleichungen von oben nach unten überprüft und, falls passend, ausgewertet werden, nennt man auch Pattern Matching. Es findet in Haskell große Anwendung und erleichtert in vielen Fällen die Programmierung von sonst recht komplexen Problemen, da man unter Umständen an bestimmten Strukturen von Parametern interessiert ist, die man relativ leicht via Pattern Matching abfragen kann (vgl. [8], S. 38 ff.). Außerdem spielt es besonders bei (selbst definierten) Datentypen eine wichtige Rolle.

Im Allgemeinen gibt es in Haskell zwei verschiedene Programmierstile. Man spricht dabei von *Declaration-* und *Expressionstyle*, welche sich vor allem in Struktur und Syntax einer Funktionsdefinition unterscheiden. Der interessierte Leser sei dabei auf die vielfältige Literatur zu Haskell verwiesen; es wird im weiteren nicht näher darauf eingegangen. Als kurze Anmerkung sei nur gesagt, dass der Deklarationsstil (*Declarationstyle*) sich bei Funktionsdefinitionen durch Gleichungen auszeichnet, während der Ausdrucksstil (*Expressionstyle*) überwiegend sog. *Lambda* und *Let Expressions* benutzt (vgl [7]). Im folgenden wird überwiegend der Deklarationsstil verwendet; es ließe sich aber jede hier vorgestellte Funktion auch im Ausdrucksstil schreiben. Auf die Lambda-Ausdrücke wird später noch gezielt eingegangen.

## 2.2 Currying

Wie wir bei der letzten Beispielfunktion `maximum` gesehen haben, beinhaltet die Typdefinition zwei Pfeile. Dies scheint auf den ersten Blick zu verwirren, da man eher folgenden Typ erwartet hätte (wie man ihn aus anderen Programmiersprachen kennt):

```
maximum :: (Int , Int) -> Int
```

Man hätte `maximum` sicherlich so definieren können. Allerdings wäre `maximum` dann keine Funktion zweiten Grades mehr. Sie wäre eine Funktion ersten Grades, deren Parameter ein 2-Tupel von Integern wäre. Man kann die Typdefinition von `maximum` mit zwei Pfeilen nämlich auch so verstehen: `maximum` ist eine Funktion mit einem Parameter (`Int`), deren Rückgabewert eine Funktion mit einem Parameter ist (`Int`), die als Rückgabewert ein `Int` liefert. Allgemeiner formuliert gilt:

*Eine Funktion n-ten Grades wird als Funktion 1. Grades aufgefasst, deren Rückgabewert eine Funktion (n-1)-ten Grades ist.*

Es gibt mehrere Gründe für diese Art von Funktionsdefinition. Zum einen die Tatsache, dass der Lambda Kalkül nur einstellige Funktionen zulässt, zum anderen aber auch die elegante Möglichkeit, Funktionen nur teilweise anzuwenden, um somit eine neue Funktion zu erhalten, deren erster Parameter z.B. schon vorgegeben ist (vgl. [8], S. 185 ff.).

Ein Beispiel wäre eine Funktion `timesThree`, die ein Argument mit 3 multipliziert:

```
timesThree :: Int -> Int
timesThree = (3 *)
```

Da der Ausdruck `(3 *)` eigentlich noch einen Parameter erwartet - nämlich den zweiten Parameter an `(*)` - ist dieser also äquivalent zu der Funktion, die ihre Argumente mit 3 multipliziert. Man kann sich hier sicherlich auch weitaus komplexere Szenarien vorstellen, in denen Currying Sinn machen kann.

## 2.3 Datentypen

Wie in jeder (stark) typisierten Programmiersprache gibt es auch in Haskell eine ganze Reihe von bereits vordefinierten (elementaren) Datentypen. Zu den wichtigsten zählen unter anderem:

```
Int, Integer, Double, Float, Char, Bool, Complex
```

Man beachte, dass der vermeintliche Datentyp `String` hier nicht auftaucht. Das liegt daran, dass `String` in Haskell lediglich Listen vom `Char`-Datentyp sind. `String` wird folgendermaßen definiert:

```
type String = [Char]
```

Das Schlüsselwort `type` bezeichnet ein Typsynonym, ähnlich der `typedefs` in C++.

Bei den anderen vordefinierten Typen handelt es sich um sog. *Algebraische Datentypen*, die im Gegensatz zu z.B. durch Klassen definierte Typen<sup>24</sup> strukturell beschrieben und typischer Weise via Pattern Matching abgefragt werden. Ein sehr einfaches Beispiel ist der Datentyp `Bool`, der boolesche Werte beschreibt und nicht wie in vielen anderen Sprachen fest in die Sprache eingebaut, sondern in Haskell selbst implementiert ist. Er ist sogar so simpel, dass man ihn auch selbst hätte implementieren können, wäre er nicht schon vorhanden (vgl. [1]). Die Typdefinition sieht folgendermaßen aus:

<sup>24</sup>im Sinne einer objekt-orientierten Programmiersprache

```
data Bool = True | False
```

Das Schlüsselwort **data** gibt an, dass eine Typdefinition eingeleitet wird, gefolgt vom Namen des Datentyps und anschließend seinen *Konstruktoren* (getrennt durch einen senkrechten Balken). Man sollte sich bei Konstruktoren in Haskell nicht zu sehr an denen in objekt-orientierten Sprachen orientieren. Konstruktoren in Haskell bieten die Möglichkeit verschiedene Strukturen des selben Datentyps zu definieren und nicht wie in z.B. Java ein “Objekt” zu initialisieren<sup>25</sup>. Der Datentyp **Bool** hat damit zwei mögliche Werte: **True** und **False**. Der Datentyp ist deswegen simpel, da er keine Parameter besitzt, weder im Typ noch in den Konstruktoren. Es handelt sich sozusagen um eine reine Aufzählung, einer *Enumeration*. Kompliziertere Datentypen besitzen meist zumindest einen Parameter (vgl. [8], S. 243 ff.). Ein Beispiel ist der vordefinierte Datentyp **Maybe**, welcher ebenfalls zwei Zustände hat:

```
data Maybe a = Nothing | Just a
```

**Maybe** hat zusätzlich einen Typparameter *a*, ähnlich wie wir es zuvor bei den Listen kennen gelernt haben. Man kann **Maybe** so verstehen, dass er entweder **Nothing** ist, also keinen Wert hat, oder aber einen Wert vom Typ *a* enthält, also **Just a**.

Eine Beispielfunktion, die von **Maybe** gebrauch macht ist folgende Funktion **last**, welche das letzte Element einer Liste zurückgibt:

```
— gibt von einer Liste das letzte Element zurück
last :: [a] -> Maybe a
last []      = Nothing
last [x]     = Just x
last (x:xs) = last xs
```

**last** ist ein schönes Beispiel für die Verwendung von **Maybe**. Denn was wäre der Rückgabewert einer leeren Liste? Entweder wirft man hier einen Fehler bzw. eine Exception und das Programm wird beendet - wie es in der tatsächlichen Implementation von **last** der Fall ist - oder aber man gibt direkt

<sup>25</sup>im Sinne von Anlegen korrekter Defaultwerte etc.

im Typ an, dass der Rückgabewert auch mal **Nothing** sein kann (vgl. [1]). Genau das wird hier gemacht. Welche Variante eleganter ist, sei hier jedem selbst überlassen.

Würde man nun eine Funktion schreiben, die ein **Maybe** als Parameter bekommt und z.B. angibt, ob es sich um einen richtigen Wert (oder um **Nothing**) handelt, würde man via Pattern Matching den übergebenen Wert abfragen:

```
hasValue :: Maybe a -> Bool
hasValue Nothing = False
hasValue (Just x) = True
```

Hier erkennt man schön, dass es sich bei **False** und **True** ebenfalls nur um Konstruktoren handelt - nämlich die des Datentyps **Bool**. Wollte man jetzt etwas mit dem Wert machen, falls es sich denn nicht um **Nothing** handeln sollte, so könnte man in der zweiten Gleichung ohne Weiteres auf das **x** zugreifen.

Beispiel:

```
f :: Maybe Int -> Int
f Nothing = 0
f (Just x) = x
```

**f** gibt in diesem Fall den Wert des Parameters zurück, falls vorhanden, ansonsten konstant 0. Der Typparameter wurde hier durch einen konkreten Typ (**Int**) ausgetauscht.

Interessanterweise kann der Haskell Compiler erkennen, wenn eine Gleichung fehlt. Würde man nämlich die zweite Gleichung bei der Definition von **hasValue** weglassen, so würde ein Konstruktor (**Just**) nicht mitbetrachtet werden - der Compiler würde eine Warnung ausgeben. Das liegt daran, dass Haskell eine stark typisierte Sprache ist. In Haskell hat jeder Ausdruck einen klar definierten Typ, auch wenn es dem Programmierer nicht immer klar sein muss. So ist es in den meisten Fällen nicht nötig, den Typ einer Funktion bzw. eines Ausdrucks explizit mit anzugeben - der Compiler erkennt diesen auch selbstständig und prüft zugleich ob das Programm korrekt

*getypt* ist<sup>26</sup> (vgl. [3], S. 9).

Ein letztes Beispiel für einen Algebraischen Datentyp sei der in Haskell eingebaute polymorphe Listentyp (`[a]`). Dieser ist zwar aus Geschwindigkeitsgründen bei den meisten Implementationen direkt eingebaut, allerdings verhält er sich identisch zu einem von Hand definierten Listentyp, der wie folgt aussehen könnte:

```
data List a = Nil | Cons a (List a)
```

So entspricht der Konstruktor `Nil` der leeren Liste (`[]`), `Cons` dem infix Listen-Konstruktor `:`. Es handelt sich selbstverständlich um eine rekursive Typdefinition. Zur Erinnerung: Der tatsächliche Listentyp wäre in etwa so deklariert:

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

Was der obigen Definition sehr nahe kommt.

Weitere Beispiele für Typdefinitionen und darauf arbeitende Funktionen finden sich im Anhang bzw. im beiliegenden Quellcode.

Zusätzlich sei hier auf die Vielzahl von bereits vordefinierten Datentypen der Haskellsprachbibliothek, der sog. *Prelude* (s. [1]), verwiesen.

## 2.4 Funktionen höherer Ordnung und Lambda Ausdrücke

Wie bereits zuvor erwähnt, beruht Haskell auf dem Lambda-Kalkül, zu dessen Grundlage Lambda Ausdrücke (*Lambda Expressions*) gehören. Außerdem gibt es in funktionalen Sprachen wie Haskell das Prinzip von Funktionen höherer Ordnung. Auf beide Aspekte soll im Folgenden näher eingegangen werden.

Funktionen höherer Ordnung (*higher-order functions*) sind Funktionen, die als Parameter Funktionen erwarten. Sie sind das zentrale Mittel der Abstraktion in funktionalen Programmiersprachen, da sie oft sehr allgemein gehalten und gerade dadurch vielseitig einsetzbar sind (vgl [3], S. 56 ff.). Lambda Expressions sind Werte, oder auch Ausdrücke, deren Typ eine Funktion

---

<sup>26</sup>Mit *getypt* ist gemeint, ob bei Funktionsaufrufen die Parameter und Rückgabewerte entsprechend zueinander passen

ist. In anderen Sprachen werden sie auch als anonyme Funktionen bezeichnet, da sie Funktionen sind, die keinen Namen besitzen<sup>27</sup> (vgl. [8], S. 171 ff.).

Da in Haskell Funktionen höherer Ordnung oft Anwendung finden, spielen gerade dort Lambda Ausdrücke eine wichtige Rolle.

### 2.4.1 Funktionen höherer Ordnung

Eine oft verwendete Funktion höherer Ordnung ist **map**, welche als Parameter eine Funktion sowie eine Liste beliebigen Typs erwartet und die übergebene Funktion auf alle Elemente der Liste anwendet (s. [1]). Der Rückgabewert ist eine Liste der Funktionswerte.

Die Definition von **map** sieht wie folgt aus:

```

— wendet Funktion f auf alle Elemente der Liste an
— und gibt die Funktionswerte als Liste zurück.
map :: (a -> b) -> [a] -> [b]
map _ []      = []
map f (x:xs) = f x : (map f xs)

```

Die Definition von **map** ist relativ einfach. Zunächst betrachten wir den Fall, dass als zweiter Parameter eine leere Liste übergeben wurde. Da uns die übergebene Funktion (erster Parameter) nicht interessiert, schreiben wir hier nur einen Unterstrich, welcher in Haskell als Platzhalter für alle möglichen Werte steht (man sagt auch, *es erfüllt jedes Pattern*)<sup>28</sup>. In diesem Fall geben wir lediglich die leere Liste zurück.

Die andere Gleichung behandelt den Fall, dass eine nicht-leere Liste übergeben wurde. Hier wird **f** auf den Kopf der Liste (**x**) angewandt und mit dem Listen-Konstruktor (**:**) mit der Liste verknüpft, die entsteht, wenn wir **map** rekursiv mit **f** und dem Rest der Liste (**xs**) aufrufen. Als Resultat liefert uns **map** also eine Liste von Werten, die durch die Anwendung einer übergebenen Funktion (hier **f**) auf die Elemente der Liste entstanden sind.

Wie interessant **map** eigentlich ist, mag einem zunächst nicht klar sein. Es sollte einem aber auffallen, dass **map** völlig allgemeingültig ist. Es wird

<sup>27</sup>z.B. in C# 3.0 (*anonymous functions*)

<sup>28</sup>s. Abschnitt 2.2, 2.3 (*Pattern Matching*)

kein direkter Typ verlangt, da **map** auf jeder Art von Liste arbeiten kann, vorausgesetzt der Typ der Funktion passt zu dem Typ der Listenelemente. Dass die übergebene Funktion auch eine Komposition von anderen Funktionen sein kann, ist hier völlig irrelevant. Das macht die Definition von **map** so einfach; ihre Anwendungsmöglichkeiten aber zugleich so weitreichend (vgl. [3], S. 59 ff.). Die Stärke liegt ganz klar in der Abstraktheit. Während eine Funktion wie **map** noch relativ einfach und doch von großem Nutzen ist, so gibt es zahlreiche Beispiele von bereits vordefinierten Funktionen, die durchaus komplexerer Natur sein können. Sie alle haben aber eines gemeinsam, nämlich die Möglichkeit der Abstraktion durch Hinzunahme von Funktionen als Parameter. Ein weiteres Beispiel für die Mächtigkeit Funktionen höherer Ordnung ist der Funktionenkompositionsoperator, welcher auch aus der Mathematik bekannt ist.

```

— Funktionenkomposition
— In der Mathematik ein kleiner runder Kreis
— in Haskell ein Punkt
— (lässt sich besser mit der Tastatur schreiben)
— Nimmt zwei Funktionen sowie einen Parameter
— und "komponiert" sie zum Ergebniswert
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
— oberes ist identisch zu: f . g (x) = f (g x)

```

Es handelt sich hier um einen infix Operator (erkennbar an den Klammern um den Funktionsnamen), der zwei Funktionen  $f$  und  $g$  mit entsprechend passenden Parametern und Rückgabetypen nimmt, sowie einen weiteren Parameter vom gleichen Typ des Eingabeparameters der zweiten Funktion erwartet. Als Rückgabetypp wird etwas vom Typ des Rückgabewertes der ersten Funktion geliefert, was exakt zu der mathematischen Definition der Komposition passt. Hier kommt auch Currying oft zum Tragen, da meist der dritte Wertparameter nicht explizit bekannt ist, weil man z.B. die Komposition als Mittel zur Erschaffung einer neuen Funktion nutzt, die dann einen Parameter von genau diesem Typ erwartet (vgl [8], S. 169 ff.).

Beispiel:

```

not :: Bool -> Bool
not True = False
not False = True
— intEven gibt an, ob Int gerade
— intOdd gibt an, ob Int ungerade
— rem gibt ganzz. Rest bei Division zurück
intEven, intOdd :: Int -> Bool
intEven n = n 'rem' 2 == 0 — infix durch '...'
intOdd    = not . intEven

```

Die Funktion **not** ist die boolesche Negation, und ist genauso wie hier in der Haskell Prelude implementiert (s. [1]). `intEven` und `intOdd` sind jeweils Funktionen, die ein `Int` nehmen und angeben, ob die übergebene Zahl gerade bzw. ungerade ist. Entscheidend für dieses Beispiel ist die Definition von `intOdd`, da hier der Parameter nicht explizit angegeben wird, sondern implizit durch die Komposition von **not** und `intEven` vorhanden ist. Der Rückgabetypp der Komposition ist nämlich eine Funktion, die ein **Int** als Parameter erwartet und ein **Bool** zurückgibt.

Erwähnenswert ist vielleicht auch noch, dass sich jede zweistellige Funktion in Haskell auch infix schreiben lässt, wie die Funktion **rem** im obigen Beispiel, indem man den Funktionsnamen einfach durch Hochkommata einschließt (vgl [3], S.45).

### 2.4.2 Lambda Ausdrücke

Lambda Ausdrücke spielen, wie bereits erwähnt, gerade bei der Verwendung von Funktionen höherer Ordnung eine wichtige Rolle, da sie den Mechanismus darstellen, um Funktion innerhalb eines Funktionsaufrufes zu definieren. Sie stellen oft eine kompaktere und zugleich elegantere Lösung dar als ihr durch normale Funktionen definiertes Pendant (vgl. [8], S. 173 f.). Als Beispiel soll hier wieder die Funktion **map** dienen:

```

zahlListe :: [Int]
zahlListe = [1,2,3,4,5]

```

```
quadratListe :: [Int]
quadratListe = map (\x -> x * x) zahlListe
— => quadratListe = [1,4,9,16,25]
```

Zunächst einmal haben wir eine nullstellige Funktion bzw. eine Konstante (`zahlListe`), die als Wert eine Liste von Integern hat. Als weitere Konstante haben wir eine Liste (`quadratListe`), die als Elemente die quadrierten Werte der ersten Liste beinhaltet. Das Quadrieren der Werte der ersten Liste übernimmt eine Funktion, die wir hier direkt in den Aufruf von `map` geschrieben haben. Es handelt sich dabei um eine anonyme Funktion bzw. einen Lambda Ausdruck, da diese Funktion nur hier verwendet wird; ihr wurde kein expliziter Name gegeben. Lambda Ausdrücke zeichnen sich durch einen vorangestellten Backslash (`\`) aus, gefolgt von einer Parameterliste, einem Pfeil und dann dem eigentlichen Funktionsrumpf. In diesem Beispiel haben wir also eine Funktion, die einen Parameter besitzt (`x`) und als Rückgabewert das Quadrat des Parameters liefert. Nun wird diese anonyme Funktion auf jedes Element von `zahlListe` angewandt und `map` gibt dann diese Werte als Liste zurück, was in unserem Beispiel `quadratListe` entspricht.

Der Lambda Ausdruck muss geklammert werden, damit der Compiler erkennen kann, wo dieser aufhört und wo der normale Funktionsaufruf weitergeht. Als Anmerkung sei noch erwähnt, dass die Wahl, den Backslash als einleitendes Zeichen eines Lambda Ausdrucks zu nehmen, daher rührt, dass dieser dem griechischen Buchstaben Lambda am nächsten kommt<sup>29</sup>. Die Lambda-Notation wurde aus dem Lambda Kalkül übernommen, in dem Funktionsausdrücke mit einem Lambda beginnen (s. [10]).

### 3 Haskell Spezialitäten

Im Folgenden werden Haskellspezifische Sprachkonstrukte und -eigenschaften besprochen, die Haskell unter den funktionalen Sprachen recht einzigartig

---

<sup>29</sup>man kann sich das so vorstellen, dass dem Backslash nur ein kleiner Strich am unteren linken Ende zu einem korrekten Lambda fehlt

machen. Dazu zählen zum einen die Typklassen (*Type Classes*), zum anderen die Bedarfsauswertung (*Lazy Evaluation*).

### 3.1 Typklassen

Wir haben bereits eine ganze Reihe von Funktionen in Haskell kennengelernt. Eine Frage dürfte sich dem aufmerksamen Leser aber immer wieder gestellt haben, und zwar die nach einer Möglichkeit der Funktionen- bzw. Operatorüberladung. Da Haskell eine strikt typisierte Sprache ist, scheint es fast zwingend, für jeden numerischen Datentyp eigene arithmetische Operatoren definieren zu müssen, da für Haskell etwas vom Typ **Int** nicht das gleiche ist wie etwas vom Typ **Float**. Dies ist nur eines der Probleme, denen Typklassen entgegenzutreten.

Betrachtet man z.B. folgende Funktion `square`:

```
square :: Num a => a -> a
square x = x * x
```

Als erstes würde man vermutlich als Funktionswerte Integer oder auch Float in Erwägung ziehen, aber schnell feststellen, dass man die gleiche Operation auch auf anderen numerischen Datentypen durchführen möchte. Genauergenommen auf allen Datentypen für die es auch einen Multiplikationsoperator (`*`) gibt.

Genau aus diesem Grund unterscheidet sich auch der Funktionstyp von denen, die wir vorher gesehen haben. Zwar handelt es sich bei `square` um eine polymorphe Funktion, was man an der Typvariablen `a` erkennen kann, allerdings wird zugleich eine Typeinschränkung vorgenommen, was durch den Doppelpfeil gekennzeichnet wird. Die Typeinschränkung steht dabei direkt davor und gibt an, welche Datentypen die Typvariable `a` annehmen kann. In diesem Fall sind das alle Datentypen, die sog. *Instanzen* der Typklasse **Num** sind. Als Instanzen einer Typklasse meint man nicht etwa instanziierte Objekte einer Klasse, wie es bei objekt-orientierten Programmiersprachen der Fall ist. Man kann Typklassen in Haskell vielmehr als eine Art Schnittstelle, ähnlich wie z.B. in Java, sehen. Die Instanzen (in Haskell) entsprechen dann den die Schnittstelle implementierenden Klassen (in Java).

So gibt eine Typklasse eine Reihe von Funktionsköpfen vor, die von den instanzierenden Datentypen typspezifisch implementiert werden müssen. Im Falle von **Num** sind das die gängigen arithmetischen Operatoren der Mathematik, abgesehen von der Division:

```
class (Eq a, Show a) => Num a where
  (+), (-), (*) :: a -> a -> a
  negate, abs :: a -> a — add. Inverses & Betrag
```

Eine Typklassendefinition beginnt mit dem Schlüsselwort **class** gefolgt von dem Namen der Typklasse sowie einem Typparameter (oftmals *a*), welcher dem die Typklasse instanzierenden Datentyp entspricht. Anschließend folgt nach dem Schlüsselwort **where** die zu implementierenden Funktionsköpfe. Es ist auch möglich, Default-Implementationen bereitzustellen, welche dann wahlweise von den instanzierenden Datentypen überschrieben werden können. Interessanterweise liegt bei der Klassendefinition von **Num** ebenfalls eine Typeinschränkung vor, welche besagt, dass jeder Datentyp *a*, welcher **Num** instanzieren will, ebenfalls eine Instanz von den Typklassen **Eq** und **Show** sein muss.

Die Typklasse **Eq** stellt Vergleichsoperatoren wie Gleichheit (**==**) und Ungleichheit (**/=**) bereit, während **Show** Funktionen zum Konvertieren in einen String bzw. zum Konvertieren von Strings in entsprechende Werte des jeweiligen Datentyps vorgibt<sup>30</sup>.

Eine solche Typeinschränkung wird oft auch als Typklassen-Vererbung bezeichnet, da alle Instanzdatentypen von **Num** auch Instanzen von **Eq** und **Show** sind. Sie “erben” quasi diese Funktionalität (da bei Nichteinhaltung der Typeinschränkung der Compiler einen Fehler melden würde).

Als Beispiel für eine Typklassendefinition mit Default-Implementationen dient hier **Eq**:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y      = not (x == y)
  x == y      = not (x /= y)
```

<sup>30</sup>vergleichbar mit `toString()` und `tryParse()` in Java

Hier wurde gleich für beide Operatoren eine Implementation vorgegeben, sodass für einen Instanzdatentyp nur einer der beiden Operatoren typspezifisch implementiert werden muss. Es bleibt einem aber auch offen, beide Operatoren zu implementieren, auch wenn das meistens nicht notwendig ist.

Eine Instanzdefinition von **Eq** könnte für folgenden Beispieldatentyp `Person` so aussehen:

```

— TypSynonyme
type Alter = Int
type Name = String
— Der eigentliche Datentyp
data Person = Person Name Alter
— Person ist Instanz von Eq
instance Eq Person where
    (Person n1 a1) == (Person n2 a2) = (n1 == n2) &&
        (a1 == a2)

```

In diesem Beispiel wäre eine Person gleich einer anderen, wenn ihre Namen und Alter gleich sind.

Weitere Beispiele für Datentypen und Typklassen finden sich im Quelltext-Anhang.

## 3.2 Bedarfsauswertung

Bedarfsauswertung (*Lazy Evaluation*) bezeichnet die Eigenschaft, dass Funktionsparameter nur ausgewertet werden, falls sie für die weitere Berechnung von Nöten sind. Dies steht im Gegensatz zu den meisten Programmiersprachen, bei denen Funktionsargumente zunächst immer ausgewertet werden, bevor sie innerhalb der Funktion zur weiteren Verarbeitung kommen.

Haskell ist eine *lazy-evaluation language*, sodass nur diejenigen Werte berechnet werden, die auch wirklich nötig sind. Das kann in vielen Fällen zu besseren Performancesituationen führen, birgt aber auch zugleich eine Reihe von Problemen. Ein solches Problem ist die nichtdeterministische Abarbeitung bei Funktionsaufrufen. Es ist aufgrund der Bedarfsauswertung relativ schwer vorhersagbar, wann welcher Teil einer Berechnung wirklich stattfindet.

den wird, da so lange mit der Berechnung eines Ausdrucks gewartet wird, bis dieser wirklich notwendig ist (vgl [8], S. 337 ff.).

Dazu ein kleines Beispiel mit der bereits bekannten Funktion `square`:

```
square 2
=> 2 * 2 => 4

square (square 2)
=> (square 2) * (square 2)
=> (2 * 2) * (2 * 2) => 4 * 4 => 16
```

An diesem Beispiel erkennt man gut die Abarbeitung eines Aufrufs einer einfachen Funktion wie `square`. Im ersten Fall liegt der übergebene Parameter bereits als Atomformel vor, da es sich um eine ganze Zahl handelt. Folglich wird der Parameter in den Funktionsrumpf eingesetzt und direkt ausgewertet.

Im zweiten Fall verläuft der Vorgang analog, mit dem Unterschied, dass hier keine Atomformel übergeben wurde, sondern ein weiterer Aufruf von `square` mit einem Parameter. Bei einer Sprache mit strikter Auswertung würde zunächst der übergebene Parameter ausgewertet und dann in den Funktionsrumpf eingesetzt werden. Da Haskell aber eine nicht-strikte Bedarfsauswertung besitzt, wird so lange mit der Auswertung der Parameter gewartet, bis diese tatsächlich berechnet werden müssen. Obwohl hier der Ausdruck `(square 2)` augenscheinlich zwei mal ausgewertet wird, ist dies tatsächlich nicht der Fall, da der Haskell Compiler weiß, dass bei beiden Aufrufen von `square` das selbe Ergebnis auftreten muss (es liegt der gleiche Parameter vor, nämlich 2), sodass der Ausdruck nur einmal ausgewertet und anschließend in beide Positionen eingesetzt wird (vgl. [8], S. 340 ff.).

Ein großes Dilemma der Bedarfsauswertung zeigt sich bei dem Willen, Ein- und Ausgaberroutinen zu verwenden, welche auf eine sequenzielle, stark deterministische Abarbeitung angewiesen sind (vgl. [8], S. 384 ff.).

### 3.3 I/O & Monaden

Die Fragestellung nach einem einheitlichen, der Sprache angemessenen Umgang mit Ein- und Ausgabeoperationen erwies sich in Haskell zunächst als

ein relativ schwieriges Problem. Zum Einen aufgrund der eben genannten Probleme der Bedarfsauswertung, welche es schwer machen, deterministisch den Zeitpunkt der Evaluation von Funktionsparametern zu bestimmen, zum Anderen die Tatsache, dass Haskell als rein funktionale Sprache entworfen worden war, welche jegliche Seiteneffekte prinzipiell unmöglich macht. Die Antwort auf diese Problemstellung stellt das Konzept der Monaden dar, dessen Ursprung in der Kategorientheorie der höheren Mathematik zu finden ist (vgl. [6, 5]).

Aufgrund der recht hohen Komplexität und dem sehr großen theoretischen Umfang dieser Problemstellung wird im Folgenden nur knapp und sehr oberflächlich darauf eingegangen. Es wird dabei auf die vielfältige Literatur (welche vor allem auch im Internet zu finden ist) verwiesen. Entsprechende Literaturverweise sind im Anhang zu finden.

Das Ziel bei der Entwicklung eines eleganten Umgangs mit Ein- und Ausgabeoperationen (*I/O*)<sup>31</sup> war es, die Sprache an sich, wenn möglich, kaum verändern zu müssen. Vor allem aber sollte sie ihren rein funktionalen Charakter beibehalten. Während die meisten funktionalen Programmiersprachen immer auch Seiteneffekte aus praktischen Gründen erlauben, versuchte man bei Haskell einen anderen Weg zu finden.

Wie genau es zu der Entwicklung von sog. *simulierten Seiteneffekten* im Allgemeinen und I/O-Operationen im speziellen gekommen ist, lässt sich am besten in den offiziellen Dokumenten der Haskell-Entwickler lesen (s. [6]).

Die grundlegende Lösung des Problems lässt sich aber so verstehen, dass Seiteneffekte in Haskell nicht wirklich existieren. Sie werden simuliert. Als Beispiel dienen hier zwei gängige Ein- und Ausgabeoperationen:

```
getLine    :: World          -> (String, World)
putStrLn  :: String -> World -> ((), World)
```

Die Typen der beiden “Funktionen” erscheinen zunächst etwas seltsam. `getLine` ist eine Operation, man sagt auch *Aktion*<sup>32</sup>, welche, wenn ausgeführt, einen String von der Standardeingabe zurückgibt. Entsprechend ist `putStrLn`

---

<sup>31</sup>engl. *Input/Output*

<sup>32</sup>engl. *Action*

eine *Aktion*, welche, wenn ausgeführt, einen String auf der Standardausgabe ausgibt. Die Betonung liegt hier bewusst auf das "... ,wenn ausgeführt, ...", da die Seiteneffekte, welche diese Aktionen hervorrufen erst entstehen, wenn das Programm ausgeführt wird (vgl. [8], S. 385 f.). Haskell an sich kennt keine wirklichen Seiteneffekte. Lediglich die Laufzeitumgebung bzw. der Compiler weiß, wenn er eine bestimmte Aktion sieht, dass er sie entsprechend umsetzen muss<sup>33</sup>. Um Haskell als rein funktionale Programmiersprache weiterhin durchgehen zu lassen, übergibt man den Aktionen einen zusätzlichen Parameter, der sog. World, welcher den aktuellen "Zustand der Welt" vor der Ausführung der Aktion darstellt. Der Rückgabetyt ist dann ein Tupel von dem Typ den die Aktion eigentlich zurückgeben soll, sowie wiederum etwas vom Typ World, welches den Zustand nach Aufruf der Aktion widerspiegelt. Die Tatsächlichen Signaturen sehen allerdings etwas einfacher aus, da man die zusätzlichen Werte der Zustände versteckt - in einem sog. *Monaden* (vgl. [8], S. 401 ff.).

So lassen sich die beiden zuvor genannten Aktionen auch folgendermaßen definieren:

```
type IO a = (World -> (a , World))
getLine  :: IO String
putStrLn :: String -> IO ()
```

Man verbirgt durch einen speziellen Typ die Zustandsparameter bzw. -rückgabewerte<sup>34</sup>. Was interessant an diesem Ansatz ist, ist die Tatsache, dass der Compiler anhand des Typs einer Funktion erkennen kann, ob diese Ein- oder Ausgabeaktionen (bzw. Seiteneffekte im Allgemeinen) durchführt, da der Typ sich von dem einer normalen Funktion unterscheidet. Man erkennt dies daran, dass der Typ z.B. nicht

```
String -> String
```

sondern

```
String -> IO String
```

<sup>33</sup>z.B. Dateien schreiben/auslesen, Standard Ein- und Ausgabe etc.

<sup>34</sup>hier ist **IO** ein Typsynonym; in Wirklichkeit eine Typklasse, was für diese Einführung aber nicht weiter tragisch ist

ist. Letzteres würde man so verstehen, dass die Aktion einen String nimmt und als Rückgabewert ebenfalls einen String hat, wobei zur Auswertung des Rückgabewertes Seiteneffekte (in diesem Fall I/O) entstehen können. Entscheidend an diesem Ansatz ist, dass Funktionen bzw. Aktionen mit Seiteneffekten niemals in rein funktionalen Teilen des Programms auftreten können, da sich die Typen nicht miteinander vertragen. Etwas vom Typ **IO String** ist niemals das gleiche wie etwas vom Typ **String**. Die Umkehrung geht aber sehr wohl, sodass Haskellprogramme typischerweise so strukturiert sind, dass der Großteil des Programms aus rein funktionalen Anteilen besteht, dem eigentlichen Kern des Programms und eine äußere Schicht als Schnittstelle zum Betriebssystem o.ä. vorliegt, welche Seiteneffekte produziert und sich um Ein- und Ausgabe kümmert (s. Abb. 1; vgl. [4]).

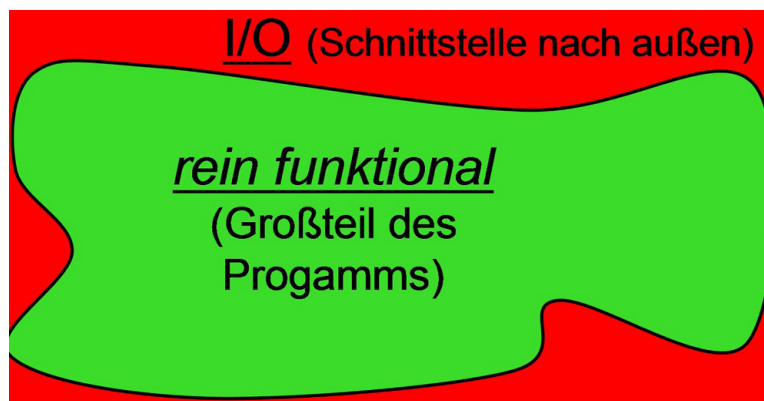


Abbildung 1: Typische Struktur eines Haskell-Programms

Als letzte Anmerkung sei noch erwähnt, dass der Begriff der Monaden eine algebraische Struktur der Kategorientheorie beschreibt, welche bei der Umsetzung von simulierten Seiteneffekten in Haskell als theoretische Grundlage diente (vgl. [4]). Was genau diese sind und wieso sie sehr wichtig sein können, sei an dieser Stelle nicht weiter erläutert. Entscheidend ist aber, dass dem Leser klar wird, inwiefern Haskell kontrolliert mit Seiteneffekten umgeht und dabei seine Reinheit nicht verliert. Das Typsystem, insbesondere die Typklassen, spielen dabei eine wichtige Rolle und erlauben es dem Laufzeitsystem bzw. dem Compiler zu erkennen, wo Seiteneffekte auftreten können und wo nicht. Das führt dazu, dass im Zuge der neuen Hardwarearchi-

tekturen, welche stark auf echt parallele Ausführung setzen, Haskell einfach und effizient zwischen evtl. problematischen Teilen eines Programms (nämlich denen mit Seiteneffekten) und denjenigen, die keine großen Probleme bezüglich paralleler Ausführung bergen, unterscheiden kann.

Während also Haskell eine solche strikte semantische Unterscheidung von Programmteilen implizit beherrscht, müssen bei vielen anderen Sprachen früher oder später Vorkehrungen getroffen werden um eine ähnliche Unterscheidung durch den Compiler möglich zu machen.

## 4 Fazit

Haskell ist eine interessante, sehr ungewöhnliche Sprache. Sie unterscheidet sich in vielen hier vorgestellten Aspekten von den gängigen imperativen, aber auch von den meisten funktionalen Programmiersprachen. Während der Einstieg<sup>35</sup> relativ schwer sein kann, hat dies sicherlich auch einen sehr positiven Nebeneffekt. Man lernt Probleme auch anders zu betrachten und zu lösen; bleibt nicht in den alten Denkmustern stecken und erweitert sein Verständnis von Programmiersprachen ungemein. In Anbetracht der aktuellen und erst recht der für die Zukunft propagierten Entwicklung moderner Computerarchitekturen scheint Haskell ungemein wichtige Konzepte zu vertreten, von denen andere Sprache sicherlich noch einige abschauen können. So scheint Haskell eine der wenigen akademischen Sprachen zu sein, die tatsächlich die Chance haben, produktiv eingesetzt zu werden. Ob diese Aussage letztlich zutrifft, wird nur die Zeit zeigen.

Als kleinen Einstieg sollte dieser Text dem interessierten Leser dienen. Er ist weder vollständig noch sonderlich umfangreich, betrachtet man den großen Umfang den eine Sprache wie Haskell ausmacht. Dennoch ist es wünschenswert, dass zumindest ein kleiner Einblick in die Sprache an sich und dessen Einsatzmöglichkeiten gegeben wurde und hoffentlich das Interesse für mehr weckt. Hier gibt es, wie bereits erwähnt, eine große Menge an sehr gutem Material vor allem kostenlos im Internet.

---

<sup>35</sup>vor allem für einen Umsteiger von imperativen bzw. objekt-orientierten Programmiersprachen hin zu Haskell

## Literatur

- [1] Haskell-Community. The haskell api search engine. <http://www.haskell.org/hoogle>, 2008.
- [2] Haskell-Community. Haskell-prime. <http://hackage.haskell.org/trac/haskell-prime/>, 2008.
- [3] P. Hudak. *The Haskell School of Expression*. Cambridge University Press, 2000.
- [4] S. Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. <http://research.microsoft.com/~simonpj/papers/marktoberdorf/>, 2000.
- [5] S. Peyton Jones. Haskell 98 language and libraries - the revised report. <http://haskell.org/onlinereport/>, 2002.
- [6] S. Peyton Jones. A history of haskell: Being lazy with class. <http://research.microsoft.com/~simonpj/papers/history-of-haskell/index.htm>, 2007.
- [7] S. Peyton Jones. Wearing the hair shirt: A retrospective on haskell. <http://research.microsoft.com/~simonpj/papers/haskell-retrospective/HaskellRetrospective-2.pdf>, 2008.
- [8] S. Thompson. *The Craft Of Functional Programming*. Addison Wesley, second edition, 1999.
- [9] W. Tichy. Die multicore-revolution und ihre bedeutung für die softwareentwicklung. *Sigs Datacom - ObjektSpektrum 04/2008* - <http://www.sigs-datacom.de/sd/publications/>, 2008.
- [10] Wikipedia. Lambda kalkül. <http://de.wikipedia.org/wiki/Lambda-Kalkül>, 2008.