Gutes Programmieren
ist wie gutes Kochen

Das Curry-Buch

Funktional programmieren lernen mit JavaScript

Hannes Mehnert, Jens Ohlig
& Stefanie Schirmer

O'REILLY®

Das Curry-Buch – Funktional programmieren lernen mit JavaScript

Hannes Mehnert, Jens Ohlig & Stefanie Schirmer

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Sebastopol · Tokyo

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
E-Mail: kommentar@oreilly.de

Bibliografische Information Der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

Lektorat: Volker Bombien, Köln
Korrektorat: Tanja Feder, Bonn
Satz: Tim Mergemeier, Reemers Publishing Services GmbH, Krefeld; www.reemers.de
Produktion: Karin Driesen, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN 978-3-86899-369-1

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Einleitung

Dieses Buch ist für Amateur-Köche, Amateur-Köchinnen, Programmiererinnen und Programmierer gedacht, die sich weiterbilden wollen.

Funktionale Programmierung als Konzept aus der akademischen Welt der Programmiersprachenforschung ist vielleicht nicht unbedingt relevant für die Praxis, in der es Deadlines gibt, zu denen ein Programm fertig sein muss oder eine Webseite schick aussehen soll. Trotzdem sind wir der Meinung, dass wir nicht über abgehobene Theorien aus dem Elfenbeinturm berichten. Irgendwann reicht es nicht mehr, Pizza beim Lieferdienst zu bestellen. Irgendwann möchte selbst die praktischste Programmiererin über den Tellerrand der vorgefertigten Funktionen aus dem Web-Framework blicken und verstehen, was in aller Welt die abgehobenen Forschenden in der Universität da eigentlich machen. Mit diesem Buch wird ein Blick über den Tellerrand gewagt. Wir haben ein Kochbuch zum Schmökern geschrieben, das eine Reise in die Welt exotischer Gewürze enthält.

Dieses Buch muss nicht von Anfang bis Ende durchgelesen werden, es ist eher ein Lesebuch als eine schrittweise Anleitung. In jedem Kapitel wenden wir uns einem Konzept zu. Jedes Kapitel ist in sich selbst größtenteils geschlossen dargestellt und kann separat gelesen werden. Wer sich für Rekursion interessiert, kann bei Kapitel 5 anfangen, wer das geheimnisvolle Lambda-Kalkül verstehen will, findet in Kapitel 7 Erklärungen dazu. Kapitel 1 und Kapitel 10 fallen ein bisschen aus dem Rahmen, sie sind ein Rundumschlag zu all den Kleinigkeiten und dem Wissenswerten, was es zu JavaScript und funktionaler Programmierung gibt.

Dieses Buch will vor allem vermitteln, dass funktionale Programmierung Spaß macht und eine sinnliche Erfahrung sein kann. Deshalb wird in jedem Kapitel auch gekocht und es werden Vergleiche zur indischen Küche gezogen. Wenn die Theorie zuviel wird, gibt es immer noch die Möglichkeit, sich mit dem Kochen eines leckeren Gerichts zu belohnen. Am Ende steht eine doppelte Wissensvermehrung: Neben interessanten theoretischen Konzepten gibt es auch einiges aus der Welt der Currys zu lernen, was möglicherweise vorher nicht bekannt war. Viel Vergnügen dabei!

Wie dieses Buch aufgebaut ist

Kapitel 1: Hinein ins Vergnügen

Wir beginnen mit unserem Arbeitswerkzeug und stellen die Besonderheiten der Sprache JavaScript vor, mit der wir uns in die Welt der Currys und vor allem der funktionalen Programmierung wagen wollen. Neben einem kurzen Blick auf die Geschichte der Sprache geht es vor allem um ihre Besonderheiten. Wir beschäftigen uns mit anonymen Funktionen und Closures und gehen auf automatisches Einfügen von Semikolons, Typenumwandlung und Scoping ein. Auch ein Blick auf das ungewöhnliche prototypenbasierte Objektsystem von JavaScript gehört zur Orientierung dazu. Das Kapitel schließt mit einem Beispiel für funktionale Programmierung: Mit unserem Rezeptor können wir automatisch über 100 leckere indische Curry-Rezepte generieren. Der Rezeptor lässt sich zwar auch imperativ programmieren, aber wir sehen am Ende des Kapitels, warum eine funktionale Lösung eleganter ist.

Kapitel 2: Abstraktion

Hier geht es es um die grundlegenden Gewürze und die Theorie, die hinter dem Aufteilen von Code steckt. Variablen und Funktionen werden, nach einem kurzen Exkurs zur indischen Ayurveda, als Möglichkeiten der Abstraktion vorgestellt, bevor wir uns Funktionen höherer Ordnung zuwenden.

Kapitel 3: Ein Topf mit Curry

Wir haben so viel über Curry geredet, dass es langsam Zeit wird, ein Standardrezept zu kochen. Wir erfahren etwas über den Namensgeber der Currifizierung, Haskell B. Curry, und implementieren eine Curry-Funktion. Dann wenden wir uns der teilweisen Funktionsanwendung zu. Zu guter Letzt lernen wir die Komposition kennen. In unserem Topf mit Curry köcheln nun verschiedene Strategien, um mächtige Funktionen zu erschaffen, die Eingabedaten verarbeiten können. Diese Funktionen sind wiederum aus kleineren Funktionen zusammengesetzt, die wir immer wiederverwenden. Langsam wird es magisch!

Kapitel 4: Gemüse, Map, Reduce und Filter

Reduce, Map und Filter als weitere Funktionen höherer Ordnung können wir nun einsetzen und ein leckeres Auberginen-Curry kochen. Diese drei Funktionen sind so grundlegend, dass sie ein eigenes Kapitel verdient haben – sie sind sozusagen das Gemüse, das als Hauptzutat in unser Curry kommt. Jede davon sehen wir uns genau an, bevor wir uns mit der Verallgemeinerung dieser drei Funktionen höherer Ordnung auf Arrays beschäftigen. Mithilfe der vorgestellten Funktionen höherer Ordnung können wir Berechnungen komplett ohne die Verwendung von imperativen Kontrollstrukturen wie Schleifen strukturieren. Wir denken, kochen und programmieren jetzt völlig funktional.

Kapitel 5: Rekursion

Rekursion ist die Denkweise der funktionalen Programmierung von elementarer Wichtigkeit. Wir schauen uns die Motivationen und Strategien für den Einsatz von Rekursion an und blicken auf Divide and Conquer. Dann gehen wir die einzelnen Schritte an einem konkreten Beispiel durch. Dabei achten wir auf mögliche Fallstricke. Wir erläutern Tail-Rekursion und beschäftigen uns ausgiebig mit dem Stack. Listen sind in diesem Zusammenhang ein besonders interessanter Datentyp, auf dem wir strukturelle Rekursion anwenden und die wir mit Pattern Matching bearbeiten. Nachdem wir zirkuläre Listen und die strukturelle Rekursion auf Bäumen betrachtet haben, entwickeln wir ein allgemeines Rekursionsschema zum Durchlaufen von verschiedenen Datentypen.

Kapitel 6: Event-basierte Programmierung und Continuations

Hier geht es um das Leben ohne Stack. Durch moderne Webprogrammierung ist eine sehr spezielle Form der Programmierung in den Fokus vieler Programmierer und Programmiererinnen geraten, die sich für viele asynchrone Prozesse im Netz anbietet: Programmierung mit Continuations. Mit dem Continuation-Passing-Style lassen sich Probleme, bei denen auf Rückgaben gewartet werden muss, verschachteln, ohne dass alles auf dem Funktionsaufruf Stack landet. Pyramidenförmiger Code, in dem eine Funktion mit einem Callback aufgerufen wird, der mit einem Callback aufgerufen wird, der mit einem Callback aufgerufen wird, und so weiter – Continuations sind nicht nur eine interessante Besonderheit von Node.js und JavaScript, sie sind praktisch viel besser einsetzbar, als wir vielleicht zunächst annehmen.

Kapitel 7: Vom Lambda-Kalkül und Lammcurry

Hier geht es um Grundnahrungsmittel der funktionalen Küche: Reis, Bohnen und das Lambda-Kalkül. In diesem Kapitel erklären wir die rätselhafte minimalistische Sprache mit dem Zeichen λ und implementieren sie uns selbst in JavaScript.

Kapitel 8: Typen

Bei JavaScript geht es in Sachen Typen nicht allzu streng zu, es handelt sich um eine dynamisch und schwach typisierte Sprache. Ein bisschen theoretisches Hintergrundwissen zu Typen ist aber ratsam, auch im Vergleich zu anderen Sprachen, von denen wir vielleicht interessante Konzepte übernehmen können. Wir blicken auch ein bisschen in die Zukunft und schauen uns Varianten von JavaScript mit stärkeren Typen an.

Kapitel 9: Kochen als Monade

Monaden, die mysteriösen Kochbehälter in rein funktionalen Sprachen, können wir auch in JavaScript erfassen, ohne den Verstand zu verlieren. Wir programmieren uns zunächst ein Semikolon als Monade, dann schauen wir uns Monaden als Muster und ihr Vorkommen in der Welt der Programmierung genauer an, z.B. als IO-Monade und Writer-

Monade. Tiefer im Kaninchenloch finden wir die Listen-Monade, Monaden als abstrakte Datentypen oder jQuery als Monade. Ganz zum Schluss – und nicht am Anfang, wie so häufig, wenn in der Informatik von Monaden geredet wird – wenden wir uns dann noch den gefürchteten Monaden-Gesetzen zu.

Kapitel 10: Nachtisch und Resteessen

Unsere Reise in die Welt der Gewürze und der funktionalen Programmierung ist fast beendet. Wir schauen noch einmal auf ein paar Besonderheiten von JavaScript und wenden uns dann Sprachen wie CoffeeScript und Verwandten zu, die JavaScript verbessern oder verbessern wollen. Dabei öffnen wir auch andere Kochtöpfe, die die Küche der Welt uns zu bieten hat und schauen kurz bei Lisp, Haskell, Agda, Idris und Coq vorbei.

Anhang: Link- und Literaturliste

Wir haben für Sie im Anhang noch einmal zusammenfassend alle wichtigen Links und unserer Meinung nach wichtige Literatur aufgeführt.

Besuchen Sie uns auf der Webseite zum Buch

Wir haben alle Programme mehrfach selbst getestet, bevor wir sie für Sie niedergeschrieben haben. Sollten Sie dennoch einmal nicht weiterkommen, nehmen Sie doch einfach Kontakt zu uns auf. Am besten geht das über die Webseite zu diesem Buch:

http://www.currybuch.de

Über diese Seite erreichen Sie auch sämtliche Code-Beispiele aus diesem Buch.

Lizenzbestimmungen

Im Falle einer Verbreitung müssen Sie anderen die Lizenzbedingungen, unter die dieser Inhalt fällt, mitteilen. Jede dieser Bedingungen kann nach schriftlicher Einwilligung des Rechtsinhabers aufgehoben werden.

Die gesetzlichen Schranken des Urheberrechts bleiben hiervon unberührt.

Typografische Konventionen

In diesem Buch werden die folgenden typografischen Konventionen verwendet:

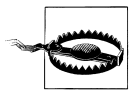
Kursivschrift
: Kennzeichnet URLs, Dateinamen, Dateinamen-Erweiterungen und Verzeichnis-/Ordnernamen. Ein Pfad im Dateisystem wird zum Beispiel als */Entwicklung/Anwendungen* erscheinen.

`Nichtproportionalschrift`
: Wird verwendet, um Code-Beispiele, den Inhalt von Dateien, Konsolenausgaben sowie Namen von Variablen, Befehlen und andere Code-Ausschnitte anzuzeigen. Das Symbol »>« kennzeichnet in einem Code-Abschnitt eine Ausgabe auf der Konsole oder den Rückgabewert eines Programmes.

`Nichtproportionalschrift fett`
: Wird zur Hervorhebung von Code-Abschnitten verwendet, bei denen es sich normalerweise um neue Ergänzungen zu altem Code handelt.

Sie sollten besonders auf Anmerkungen achten, die mit den folgenden Symbolen vom Text abgehoben werden:

Das ist ein Tipp, ein Hinweis oder eine allgemeine Anmerkung. Er enthält nützliche ergänzende Informationen zum nebenstehenden Thema.



Das ist eine Warnung oder Ermahnung zur Vorsicht, die oftmals anzeigt, dass Sie besonders aufpassen müssen.

Danksagungen

Dieses Buch wäre nicht ohne Volker Bombien vom O'Reilly Verlag entstanden. Vielen Dank Volker, dass du dich für dieses Buch so eingesetzt hast und uns immer mit Hilfe zur Seite standest. Wir danken auch dem O'Reilly-Verlag dafür, dass dieses Buch unter einer CC-BY-SA-NC-Lizenz erscheinen kann und somit die Welt ein bisschen mehr mit freiem Wissen befüllt.

Dank geht auch an unsere Reviewer, die uns viel gutes inhaltliches Feedback gegeben haben. Erste Schritte in den Text haben wir vor längerer Zeit mit Adrian Lang unternom-

men, dem wir auch für einige Inspirationen zu großem Dank verpflichtet sind. Die große Last des technischen Reviews lag auf den Schultern einer fantastischen Gruppe von Probelesern. Hier sind besonders in alphabetischer Reihenfolge Claus Schirmer, Florian Holzhauer, Jan Anlauff, Jan Lehnhardt, Jan Ludewig, Jan Mehnert, Myf Ma, Peter Sestoft und Robert Giegerich zu nennen.

Zu guter Letzt müssen wir dem Internet danken, das dafür gesorgt hat, dass dieses Buch zwischen Kopenhagen, Montreal und Berlin entstehen konnte. Es ist unsere Hoffnung, dass das Internet auch weiterhin Menschen und Daten für eine wunderbare, selbstbestimmte Zukunft zusammenbringt.

Make Datalove, not Cyberwar.

KAPITEL 1

Hinein ins Vergnügen

Im Code und in der Küche finden sich ganz ähnliche Mysterien. Es gibt viele Leute, die kochen oder bzw. und programmieren können, und für die es dennoch, sei es in der Welt der Gewürze oder in der funktionalen Programmierung, noch viel zu entdecken gibt. Beides eröffnet einem eine neue Welt, wenn man sich mit der Materie beschäftigt. Plötzlich kann alles ganz anders aussehen.

Warum funktionale Programmierung?

Eine zentrale, aber oft nur oberflächlich betrachtete Aufgabe bei der Programmierung ist das Organisieren des Programmcodes. Der Schlüssel beim Organisieren liegt in der Abstraktion. Dazu gibt es ein berühmtes Zitat von E.W. Dijkstra: »*Die Einführung geeigneter Abstraktionen ist unsere einzige mentale Hilfe, um Komplexität zu organisieren und zu meistern.*« Im Alltag der imperativen Programmierung mit JavaScript bringen ungeplante Programmänderungen die gewohnten Abstraktionsmechanismen mitunter an ihre Grenzen. In diesem Buch stellen wir die funktionale Programmierung vor, deren Ansatz sich von den übrigen Arten der Programmierung leicht unterscheidet und die eine ganze Reihe von Abstraktionsmechanismen bietet. Zudem ist sie auch ohne allzu tiefgreifende mathematische Kenntnisse leicht zugänglich.

Besonderheiten der funktionalen Programmierung

Die auffälligste Besonderheit bei der funktionalen Programmierung ist, dass Programmfunktionen wie mathematische Funktionen aufgefasst werden, so wie wir sie aus dem Mathematikunterricht in der Schule kennen. In rein funktionalen Sprachen entspricht eine Funktion genau dem Wert, den sie zurückgibt, und es ist daher oft kein `return`-Statement vorhanden. In Sprachen wie JavaScript erhalten wir den Rückgabewert erst nach Anwendung der Funktion. Eine Funktion ist in der funktionalen Programmierung völlig unabhängig von der Zeit und dem Zustand des Computers und liefert für das gleiche Argument immer wieder das gleiche Resultat. Nebenwirkungen müssen nicht berücksichtigt werden und dadurch ist es einfacher, das zum Arbeiten mit dem Programm erforderliche Verständ-

nis zu erwerben bzw. sich die betreffenden Programmstrukturen zunutze zu machen. Außerdem trennt man bei der funktionalen Programmierung die Daten selbst von dem betreffenden Vorgang: Funktionen werden aus kleineren, allgemeineren Funktionen zusammengesetzt, und die Daten werden dann von der aufgebauten Gesamtfunktion verarbeitet.

Auch in der Küche gibt es Aspekte, die wir mit der funktionale Programmierung vergleichen können. Eine Pfeffermühle zerkleinert, wenn sie betätigt wird, beim Kochen immer die Pfefferkörner, egal ob wir gerade ein Curry oder Nudeln mit Tomatensauce kochen. Die Zubereitung von Speisen ist ohnehin eine überraschend gut passende Metapher für das Programmieren. So ist eine bestimmte Art von Funktionen, die uns im weiteren Verlauf noch beschäftigen wird, nach dem Mathematiker Haskell B. Curry benannt. Aber neben dieser Namensgleicheit mit einem indischen Gericht lässt sich die Analogie noch weiter fortsetzen. In gewisser Weise ist ein Currygericht auch selbst »funktional«.

Curry bedeutet ursprünglich schlicht Sauce (von den Malayalam-Wort കുട്ടാ), aber heutzutage versteht man in Indien unter Curry ein Hauptgericht, das zu Reis oder Brot, wie zum Beispiel Chapati oder Naan, serviert wird. Das Kochen eines Currys ist gewissermaßen das Anwenden von Funktionen: Zwiebeln und Kokosmilch bilden die Basis des Gerichts, dazu kommt ein Hauptbestandteil (Hühnchen oder Gemüse) und dann wird auf diese neue Funktion eine Gewürzmischung angewendet. Man könnte sagen, das Gericht wird curryfiziert. Über die Funktion `f(x)` werden die Gewürze auf das Curry `x` angewendet.

JavaScript

Wir wollen funktionale Programmierung nicht anhand von Sprachen aus der akademischen Welt betrachten, obwohl wir auf unserer kulinarischen Reise auch in diese Kochtöpfe schauen werden (Kapitel 10), sondern anhand von JavaScript. Da jeder moderne Internetbrowser JavaScript unterstützt, ist diese Sprache momentan sehr weit verbreitet. JavaScript läuft in vielen Umgebungen – neben dem Browser auch in Fernsehgeräten und Mobiltelefonen. JavaScript wurde für Laien entwickelt, daher sind die Grundlagen einfach zu erlernen. Aktuelle Web-Applikationen nutzen JavaScript, um dynamische Benutzer-Interfaces mit einem ansprechenden Erscheinungsbild zu realisieren. In gewisser Weise bringt JavaScript das »BASIC-Gefühl« der 1980er Jahre zurück: anschalten und loslegen. JavaScript wird seit einiger Zeit auch als alleinstehende Sprache für Kommandozeilen- oder Serveranwendungen immer wichtiger und auch bei Spezialanwendungen wie der Steuerung von fliegenden Robotern verwendet (*http://nodecopter.com/*). JavaScript integriert dabei viele funktionale Aspekte, stellt aber auch einiges an Funktionalität aus der objektorientierten Programmierung zur Verfügung. Es besteht also hier die Möglichkeit, in vielen verschiedenen Paradigmen zu programmieren. Mit diesem Buch möchten wir einen »sanften Einstieg« in die funktionale Programmierung mit JavaScript bieten.

Die Geschichte von JavaScript

JavaScript wurde 1995 von Brendan Eich unter dem Namen Mocha entwickelt. In weniger als zwei Wochen sollte der Browser Netscape Navigator eine eigene Skript-Sprache erhalten und dann mit dieser ausgeliefert werden.

Interessanterweise hat JavaScript trotz seiner übereilten Entstehungsgeschichte einige Elemente der funktionalen Programmierung erhalten – functional by accident. Da die Sprache JavaScript nicht am Reißbrett, sondern von einem Menschen aus der Praxis entwickelt wurde, lässt sie sich nur sehr schwer formalisieren, was aber für die Beschreibung der Semantik Voraussetzung ist. Syntax und Semantik sind entscheidend für die Ausdrucksstärke einer Sprache. Die Syntax umfasst alle Elemente einer Sprache, wie sie im Sprachstandard festgelegt sind, also Schlüsselwörter usw. Die Semantik gibt der Syntax eine Bedeutung. Die Forschung, die sich mit der Semantik von JavaScript befasst, ist also eher eine Forschung am lebenden Objekt.

Für Design inklusive Implementierung der Sprache waren die für dieses Projekt angesetzten zwei Wochen kein besonders großzügiger Zeitplan. Er wurde aber dennoch eingehalten. Zu Brendan Eich, der in kürzester Zeit diese ungewöhnliche Programmiersprache erfand und dabei Elemente von Lisp und Self in einer gefälligen, C-ähnlichen Syntax, vereinte, gibt es allerdings auch eine negative Fußnote. 2008 spendete Eich 1000 US-Dollar für die Kampagne *»California Proposition 8«*, einer Initiative zum Verbot von gleichgeschlechtlichen Ehen. Sowohl Eichs Spende als auch die seines Arbeitgebers Mozilla sind aus Transparenzgründen öffentlich einsehbar, und diese Spende wird, wie sich denken lässt, seit März 2012 in der JavaScript-Community heftig diskutiert. Die politische Einstellung eines Menschen und seine Leistung als Softwareingenieur sollten wir aber vermutlich getrennt betrachten, so schwer es auch fällt.

JavaScript wurde bei der ECMA, einer privaten internationalen Organisation zur Normung von Computertechnik, zur Standardisierung eingereicht. Früher stand ECMA einmal für »European Computer Manufacturers Association«, seitdem sich die Organisation aber internationaler begreift, wird ECMA nicht mehr in erster Linie als Akronym für etwas verstanden. Bei der ECMA wurden verschiedene Versionen der Skript-Sprache spezifiziert – Haarspalter sprechen daher gerne von »ECMAScript«, wenn sie über die Programmiersprache sprechen, die wir in diesem Buch verwenden und die in allen wichtigen Browsern implementiert wurde. Aktuell ist die Version ECMAScript 5, aber seit 2009 zeigt sich ECMAScript 6 schon am Horizont und wird möglicherweise besonders im Zusammenhang mit dem kommenden Standard HTML5 wichtig werden. Wenig erfolgreich war der Standardisierungsversuch ECMAScript 4, der insbesondere in Sachen Typisierung neue Wege beschritt, aber letztendlich nie implementiert wurde. In Kapitel 8 im Abschnitt »ECMAScript 4« auf Seite 145 und in Kapitel 10 im Abschnitt »ECMAScript 6« auf Seite 176 werden wir noch einmal auf ECMAScript 4 und ECMAScript 6 zu sprechen kommen, aber wir greifen in diesem Buch immer auf den etablierten Standard ECMAScript 5 zurück.

Eigenschaften und Eigenheiten von JavaScript

JavaScript besitzt von Haus aus bereits einige funktionale Sprachelemente. Schauen wir uns daher einmal an, was der funktionalen Programmiererin oder dem funktionalen Programmierer bereits mitgeliefert wird.

Funktionen sind first-class citizens

Funktionen in JavaScript sind vollwertige Objekte: Sie können als Argumente an andere Funktionen übergeben werden. Außerdem können Variablen auf Funktionen verweisen. Funktionen sind also First Class Citizens oder First Class Functions. Hierbei handelt es sich um eine wichtige Grundlage der funktionalen Programmierung, denn es stellt eine notwendige Voraussetzung für weitere funktionale Konzepte wie Closures, anonyme Funktionen und Currying dar. Eine herkömmliche Funktionsdeklaration in JavaScript sieht etwa so aus:

```
function hello () {
  console.log('Hallo Welt');
}
hello();
> 'Hallo Welt'
```

Anonyme Funktionen

Ein populäres Konzept, das JavaScript von Natur aus mit funktionalen Programmiersprachen teilt, ist die anonyme Funktion. Bei einer anonymen Funktion wird – wie der Begriff schon andeutet – einfach den Namen nach dem Schlüsselwort `function` weggelassen. Damit entsteht ein Funktionsobjekt, das einer Variablen zugewiesen werden kann.

```
var hello = function () {
  console.log('Hallo Welt');
}
hello();
> 'Hallo Welt'
```

Bis jetzt stellt sich dies wie eine spleenige Syntaxvariante einer normalen Funktionsdeklaration dar.

Closures

Ihren Zauber und das Aroma ihrer Gewürze entwickelt eine anonyme Funktion als Closure, bekannt aus anderen funktionalen Sprachen wie Lisp, Haskell, Scheme und im Deutschen auch Funktionsabschluss genannt. Bei einer Closure handelt es sich eine Funktion, die sich den Zustand der Welt, in der sie erstellt wurde, merkt.

```
var greeting = function (x) {
  return function () {
    console.log(x);
  }
}
```

```
var hello = greeting("Hallo");
var world = greeting("Welt");
hello();
> 'Hallo'
world();
> 'Welt'
```

Die der Variable `greeting` zugewiesene anonyme Funktion umschließt den Wert, den sie ausgeben soll. Es handelt sich also um eine Grußfunktion mit einem eigenen Gedächtnis über den Zustand der Welt – in unserem Beispiel ist dieser Zustand nur ein String. Dabei greift die Closure implizit auf Variablen zu: Das Umschließen der Closure bezieht sich darauf, dass sie in ihrer Definition Variablen verwendet oder erzeugt und dann einen Teil des Environments, nämlich die zuvor definierten Variablen, umschließt.

Link: *http://www.haskell.org/haskellwiki/Closure*

Überraschende Sprachelemente

JavaScript ist allerdings nicht einfach eine Programmiersprache mit funktionalen Elementen, sondern mitunter ziemlich kurios.

Semikolons

JavaScript hat einige Eigenheiten, die mitunter so sehr überraschen können, dass sie möglicherweise als Fehler aufgefasst werden. Das automatische Einfügen von Semikolons (Automatic Semicolon Insertion, ASI) ist eine der verwirrendsten und umstrittensten Eigenschaften von JavaScript. Kurz gesagt können in vielen Fällen Semikolons am Zeilenende weggelassen werden, was als Erleichterung gedacht war. Allerdings werden bei manchen Zeilenenden dann auch wieder Semikolons an unerwarteten Stellen eingefügt, was besonders bei Berechnungen mit den Operatoren `++` und `--` oder bei Schlüsselwörtern wie `return` überraschen kann. Douglas Crockford, Autor des wahrscheinlich meistgelesenen JavaScript Stil-Handbuchs »JavaScript: The Good Parts« und Entwickler von Technologien wie JSON (JavaScript Object Notation, Serialisierung von Daten in JavaScript), empfiehlt, ASI vollständig zu ignorieren und jedes Statement mit einem Semikolon abzuschließen, ohne auf die eingebaute Magie des Parsers zu vertrauen. Wir halten uns in diesem Buch hauptsächlich an Crockford. Allerdings nehmen wir zur Kenntnis, dass es auch andere, mit beinahe religiöse Eifer verfochtene Ansichten zu Semikolons gibt, die wir respektieren.

Coercion

Die zweite überraschende Eigenschaft von JavaScript ist die automatische Typumwandlung. Bei Vergleichsoperationen mit `==` und `!=` werden die Argumente nach bestimmten Regeln automatisch in Wahrheitswerte umgewandelt. So wird unter Umständen 0 oder ein leerer String zu `falsch`. Auch bei der Verwendung des Operators `+` kommt es zur Typumwandlung, da `+` sowohl Strings verketten als auch Zahlen addieren kann. Welche

Operation ausgeführt wird, wird dabei anhand der Argumente entschieden. Für den Ausdruck (`4 + 6 + ' Portionen'`) beispielsweise wird in JavaScript das Ergebnis `'10 Portionen'` ausgerechnet, für den Ausdruck (`'Portionen: ' + 4 + 6`) hingegen erhalten wir den String `'Portionen: 46'`. Warum das so ist, werden wir in Kapitel 8 erläutern.

Scoping

Ein gutes Essen besteht oft aus mehreren Gängen: Vorspeise, Hauptgericht und Nachspeise. Vor dem Curry im Hauptgang gibt es vielleicht leckere Teigtaschen, Samosas. Anschließend kommen möglicherweise Rasgulla auf den Tisch, eine indische Nachspeise, die aus geronnener Milch zubereitet wird. Die kleinen Käsebällchen werden in Sirup gekocht und mit Rosenwasser verfeinert. Dabei hat jeder Gang seinen Platz innerhalb der Menüfolge. Süßer Sirup wäre im scharfen Curry unangemessen – der Hauptgang ist ein anderer »Scope« als der Nachtisch.

Bei einem festlichen Essen in einem feinen Restaurant hierzulande gibt es ebenfalls Scoping-Regeln. Zum Fisch gibt es Weißwein, zum Braten Rotwein und zum Nachtisch wird Portwein gereicht. Der Wein wird also immer im Kontext des gerade servierten Gangs ausgewählt.

Was ist ein Scope?

Der Scope ist der Sichtbarkeitsbereich eines Bezeichners oder einer Variablen, wobei wir diese beiden Begriffe hier synonym verwenden. Das Konzept des Scopings regelt, in welchem Programmabschnitt ein Bezeichner gültig ist und welchen Wert er hat. Dies ist wichtig, um Kollisionen zwischen Bezeichnern zu vermeiden. Der Vorgang, bei dem einer Variablen oder einem Bezeichner ein Wert zugewiesen wird, wird als Variablenbindung bezeichnet. Hier wird eine Variable also an einen bestimmten Wert gebunden.

Shadowing

In dem folgenden Programmfragment verdeckt (verschattet, »shadowed«) die Variable `x` in der bedingten Ausführung die Variable `x` aus dem Hauptprogramm:

```
function where () {
  var x = "Indien";
  if (true) {
    var x = "Neu-Delhi";
    console.log("Innerhalb: " + x);
  }
  console.log("Ausserhalb: " + x);
}
```

In vielen verwendeten Programmiersprachen wird hier als Ausgabe Folgendes erwartet:

```
"Innerhalb: Neu-Delhi"
"Ausserhalb: Indien"
```

Ausflug: Variablenbindung und Zuweisung sind zwei paar Schuhe

Die Variablenbindung ist in JavaScript untrennbar mit der Zuweisung von Werten an Variablen verbunden. Wenn wir hier `var x = 1` schreiben, bedeutet dies, dass die Variable `x` an einen *Speicherbereich* gebunden wird. Gleichzeitig wird diesem Speicherbereich der Wert `1` zugewiesen. Die Variable verweist dadurch nun auf den Wert `1` – wir sagen, der Wert wurde der Variable zugewiesen. In den folgenden Zeilen können wir `x` beliebige neue Werte zuweisen, dabei wird der Wert im Speicher in den neuen Wert geändert.

In vielen funktionalen Sprachen gibt es keine Zuweisung, sondern nur Variablenbindung. Wenn wir in Haskell `x = 1` schreiben, bedeutet das, das die Variable `x` an den *Wert* `1` gebunden wird, und nicht an einen Speicherbereich. Die Variable `x` verweist dadurch für immer auf den Wert `1`, der nicht veränderbar ist. Somit ist also auch immer klar wie Kloßbrühe, auf welchen Wert eine Variable verweist. Dieser Umstand wird daher auch als Referential Transparency bezeichnet. Bei Berechnungen bauen wir in Funktionen Ergebnisse aus den Eingabewerten zusammen und geben sie zurück, anstatt Werte im Speicher zu verändern.

Die Veränderung von Werten im Speicher ist sozusagen ein »heimlicher Effekt«, der nebenbei auftritt und sich nicht in der Rückgabe der Funktion zeigt. In einer mathematisch aufgefassten Funktion wäre er daher gar nicht möglich. Deshalb sprechen wir von einem Seiteneffekt oder einer Nebenwirkung. Bei der funktionalen Programmierung vermeiden wir diese Effekte.

Statisches Scoping: Die Sichtbarkeitsregelung, in der die Variable x je nach Code-Block auf Neu-Delhi oder Indien verweist, wird als Block Scoping bezeichnet. Variablen, die innerhalb eines Blocks eingeführt werden, sind nur in diesem sichtbar. Wenn, wie im obigen Beispiel, in einer bedingten Anweisung eine neue Variable eingeführt wird, verdeckt sie andere Variablen gleichen Namens und macht sie damit unsichtbar. Dies gilt allerdings nur innerhalb des besagten Blocks.

Die Erweiterung dieser Sichtbarkeitsregelung auf ein ganzes Programm wird als lexikalisches oder auch statisches Scoping bezeichnet. Beim lexikalischen Scoping legt der umgebende Programmcode die Bindung der Variablen fest. Auf der höchsten Ebene der Sichtbarkeit befinden sich die globalen Variablen, die im gesamten Programm sichtbar sind. Unser Beispiel ist auf der niedrigsten Sichtbarkeitsebene angesiedelt, die Variable ist nur innerhalb des lokalen Blocks sichtbar. Lexikalisches Scoping wird unter anderem von den Programmiersprachen C++, C, ML, Haskell, Python und Pascal unterstützt und verwendet.

Scoping in JavaScript ist dynamisch: In JavaScript wird die Variable `x` hingegen mit dynamischen Scoping gebunden. Die Funktion `where` gibt Folgendes aus:

```
> "Innerhalb: Neu-Delhi"
> "Ausserhalb: Neu-Delhi"
```

In JavaScript befinden wir uns immer in Neu-Delhi und nicht auf dem umgebenden indischen Subkontinent.

Der Unterschied zwischen lexikalischem und dynamischen Scoping liegt darin, in welchem Codeabschnitt der Wert einer Variablen gesucht wird: beim lexikalischen Scoping wird zuerst der aktuelle Codeblock durchsucht und dann der Block, der diesen umschließt und so weiter. Beim dynamischen Scoping wird der während der Ausführung zuletzt zugewiesene Wert verwendet.

Die Funktion `whereAmI` gibt das Gleiche aus wie die Funktion `where`. Die Variable `x` ist auch außerhalb des `if`-Blocks gültig.

```
function whereAmI () {
  if (true) {
    var x = "Neu-Delhi";
    console.log("Innerhalb: " + x);
  }
  console.log("Ausserhalb: " + x);
}
```

Um in JavaScript sicherzustellen, dass eine Variable keinen unerwartete Wert enthält, können wir eine Closure erzeugen und sie sofort anwenden:

```
var x = function () { var a = 42; return a * a; }();
```

In Kapitel 5 sehen wir einen entsprechenden Anwendungsfall.

Andere Sprachen mit dynamischem Scoping: Dynamisches Scoping wird auch in anderen Programmiersprachen als JavaScript verwendet, beispielsweise in Emacs Lisp, der Erweiterungssprache des Editors Emacs. Seit neuestem unterstützt Emacs Lisp zusätzlich lexikalisches Scoping. Die logische Programmiersprache Prolog und die Programmiersprache Logo für Kinder nutzen ebenfalls dynamisches Scoping.

Perl und Common Lisp besitzen ein optionales dynamisches Scoping. Beim Programmieren wird dabei für jede Variable festgelegt, welche Sichtbarkeitsregel genutzt werden soll.

Vor- und Nachteile der Scoping-Varianten: Lexikalisches Scoping fördert das modulare Programmieren, da jede Funktion als eine in sich abgeschlossene Einheit gelesen und verstanden werden kann, wenn ihre Ein- und Ausgabe bekannt sind. Beim dynamischen Scoping hängt das Verhalten der Funktion hingegen immer von den vorherigen Variablenbelegungen ab und ist daher durch bloßes Lesen einer Funktion oft nicht vollständig erkennbar.

Ein Anwendungsfall, in dem dynamisches Scoping besonders nützlich ist, ist die Speicherung von Daten, die für unterschiedliche Funktionen verfügbar sein sollen. Die Antwort auf eine Anfrage über ein Netzwerk ist hierfür ein gutes Beispiel. Statt die Netzwerkschnittstelle (das Socket) als zusätzliches Argument durch alle Funktionen mitzuschleifen, können wir diese Information einmalig in einer Variablen mit dynamischen Scope ablegen. Wenn nun mehrere Anfragen erfolgen, weist die Variable für jede Anfrage einen unterschiedlichen Wert auf. Programmiersprachen mit lexikalischem Scoping, wie zum Beispiel C, sind in diesem Fall auf externe Bibliotheken angewiesen. Sie benötigen mehr Zeilen für den gleichen Vorgang: statt einer Variablendeklaration in Perl sind mehrere Funktionsaufrufe für die Antwort auf eine Netzwerkanfrage in C erforderlich.

Prototypenbasiertes Objektsystem

Als letzte Kuriosität besitzt JavaScript noch ein ungewöhnliches Objektsystem. Anders als klassenbasiertes Objektsystem, das in vielen anderen objektorientierten Programmiersprachen wie Java üblich ist, basiert es auf Prototypen, ähnlich wie in der Programmiersprache Self aus den 90ern. Haskell und andere rein funktionale Sprachen bieten keine Objektsysteme, sondern ermöglichen die gewünschten Abstraktionen durch ein Modulsystem, Typklassen und Datenstrukturen.

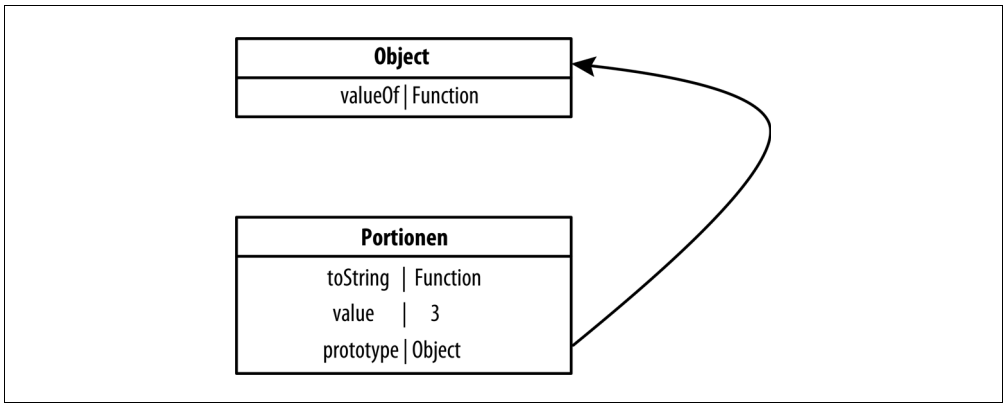


Abbildung 1-1: Prototypen und Eigenschaften in JavaScript

Auf der Abbildung zu sehen ist ein Objekt `Portionen`, dass eine Tabelle mit Eigenschaften und deren Werten besitzt: eine Funktion `toString`, den `value 3` und `prototype` – dieser zeigt auf `Object`, das eine Funktion `valueOf` enthält. Wenn `Portionen.valueOf()` aufgerufen wird, werden die betreffenden im Prototyp von `Object` vorhandenen Informationen abgerufen.

Jedes Objekt in JavaScript besteht aus einer Tabelle mit Symbolen, die auf Werte verweisen, in unserem Beispiel sind das andere Objekte oder Funktionen. Ein spezielles Symbol ist der Prototyp des Objekts, in dem allgemeine Eigenschaften für alle Objekte dieses Prototyps gespeichert sind. Jedes Objekt kann beliebige Symbole des Prototyps über-

schreiben. Eine Eigenschaft wird zuerst im konkreten Objekt gesucht. Wenn sie dort nicht gefunden wird, dann wird im Prototyp des Objekts nach Informationen gesucht. Wenn die Eigenschaft auch im Prototyp nicht gefunden werden kann, wird im Prototyp des Prototyps gesucht. Diese rekursive Kette von Prototypen wird auch als Prototyp-Chain bezeichnet.

Die Umgebung node.js und praktische Beispiele

In diesem Buch haben wir die Beispiele mit *node.js* erstellt. Diese Programmierumgebung ist besonders nützlich, wenn es darum geht, JavaScript außerhalb des Webbrowsers auszuführen, beispielsweise auf Webservern. Zudem gibt es weitere Vorteile, die node.js interessant machen:

Die Umgebung node.js nutzt Googles schnelle V8 JavaScript-Laufzeitumgebung, die auch im Google Chrome Browser eingesetzt wird. Damit V8 besser in Kontexten außerhalb des Browsers funktioniert, stellt node.js zusätzliche Schnittstellen bereit, wie zum Beispiel eine Abstraktion, die den Zugriff auf das Dateisystem ermöglicht.

Die Umgebung node.js ist zudem freie Software und für die Betriebssysteme Linux, Mac OS, Windows, FreeBSD und Solaris erhältlich. Eine Installationsanweisung und Download-Möglichkeit findet sich unter *http://nodejs.org/*.

Mit node.js werden neben einem Interpreter, der auf der Kommandozeile gestartet wird, und dem Paketmanager npm einige grundlegende Module mitgeliefert, die Funktionen zu Events, Buffer, Streams, TLS/SSL, Dateisystem-Zugriff, DNS, UDP und HTTP bereitstellen. Man kann mit wenigen Zeilen Code manuell seinen eigenen Webserver in JavaScript schreiben – dabei handelt es sich dann um das allerseits bekannte »Hello World«, das in fast keinem node.js-Tutorials fehlt. Es gibt natürlich schon eine ganze Menge Bibliotheken, die die Arbeit vereinfachen, so z.B. für Datenbank-Anbindungen, API-Handler, Parser, Template-Engines und vieles vieles mehr, wie etwa das Web-Framework »Express«. Die meisten dieser Bibliotheken können mit `npm install <Bibliotheksname>` nachinstalliert werden. Auf *https://npmjs.org/* finden sich Module für fast alle Gelegenheiten der serverseitigen JavaScript-Programmierung.

Die Beispiele aus diesem Buch können von der Seite *http://www.currybuch.de/* heruntergeladen werden.

Funktionale Sprachelemente und Konzepte

Bisher haben wir über die konkreten Sprachelemente gesprochen, die für funktionale Programmierung erforderlich sind, wie Closures und Funktionen als First-Class-Citizens. Sie machen es erst möglich, dass wir in JavaScript funktional programmieren können. Daneben gibt es die unterliegenden gedanklichen Konzepte und Ideen des funktionalen Paradigmas. Einige dieser Konzepte sind die Seiteneffektfreiheit, Zustandslosigkeit, Variablenbindung statt Zuweisung, Funktionskomposition und Funktionen höherer Ordnung, die wir alle in diesem Buch kennenlernen werden.

Unterscheidungsmerkmale von Programmiersprachen

Wie beim Kochen gibt es auch bei Programmiersprachen verschiedene grundlegende Herangehensweisen: Einige Köchinnen oder Köche kochen das Gemüse für ein Gratin oder eine Quiche lieber separat, die anderen backen alles zusammen im Ofen.

Wir können Programmiersprachen anhand von unterschiedlichen Merkmalen klassifizieren. Das wichtigste Unterscheidungsmerkmal haben wir bereits kennengelernt, das Programmierparadigma: objektorientierte Programmierung im Gegensatz zu funktionaler Programmierung. Im weiteren Verlauf werden wir einige Merkmale näher beleuchten, um JavaScript qualifiziert mit anderen Programmiersprachen vergleichen zu können. Als Erstes werfen wir einen Blick auf die Unterscheidung zwischen kompilierten und interpretierten Sprachen. Danach werden wir uns noch die Typisierung zu Gemüte führen.

Kompilierte und Interpretierte Sprachen

Um ein Programm auf einem Computer ausführen zu können, muss dieses in Maschinensprache übersetzt werden. Es gibt zwei verschiedene Wege, zur Maschinensprache zu gelangen: entweder mit Hilfe eines Compilers, der ein Programm in Maschinensprache übersetzt, oder mit Hilfe eines Interpreters, der den Programmcode direkt ausführt.

Ein Vorteil des ersten Ansatzes besteht in der hohen Geschwindigkeit, da der Compiler optimierten Maschinencode erzeugt. Ein Nachteil ist allerdings der, dass der Entwicklungszyklus »Edit-Compile-Run-Debug« umständlich ist, da zwischen dem Verändern und Ausführen immer wieder der Kompilierungsschritt nötig ist.

Der Interpreter bringt den Vorteil, dass mit ihm der Entwicklungszyklus nur noch aus »Edit-Run-Debug« besteht. Ein weiterer Vorteil besteht in der Portabilität von interpretiertem Code. JavaScript wird beispielsweise über das Internet verteilt und in dem jeweiligen Webbrowser ausgeführt. Ein kompiliertes Programm ist weniger portabel, da es zum Zwecke der Optimierung auf eine bestimmte Rechnerarchitektur festgelegt ist. Interpretierter Code hat den Nachteil, in der Ausführung langsamer zu sein, da die Laufzeitumgebung bei der Ausführung eines Programms zwischen der Programmiersprache und der Maschinensprache vermitteln muss.

Mittlerweile gibt es keine klare Grenze mehr zwischen den beiden beschriebenen Ansätzen. Es gibt inzwischen interpretierte Sprachen mit einer recht schnellen Laufzeitumgebung, z.B. V8 von Google für JavaScript. Auf der anderen Seite werden auch kompilierte Programmiersprachen wie Java und C# inzwischen in eine plattformunabhängige Zwischensprache übersetzt, was man eigentlich von interpretierten Sprachen kennt.

Typisierung

Programmiersprachen unterscheiden sich außerdem durch ihre Typisierung, d. h. ihre Art, nach der Daten wie Variablen, Funktionen und Objekten jeweils ein entsprechender Datentyp zugewiesen wird. Verschiedene Programmiersprachen unterscheiden auch, ob diese Typinformationen zur Compilezeit oder zur Laufzeit verwendet werden. In Kapitel 8 werden wir genauer auf die Typisierung eingehen.

JavaScript ist dynamisch typisiert und verwendet daher Typen vor allem zur Laufzeit. Haskell hingegen ist statisch typisiert und nutzt Typen bereits zur Compilezeit, was eine frühere Fehlererkennung ermöglicht.

JavaScript im Vergleich mit anderen Programmiersprachen: Ist das Java?

Bei JavaScript handelt es sich um eine interpretierte Programmiersprache. Der Name legt vielleicht eine Verwandtschaft mit Java nahe, denn irgendwie haben beide Sprachen etwas mit dem Internet zu tun und wurden auch zur gleichen Zeit populär. Tatsächlich sind die beiden Sprachen aber nicht verwandt, außer dass die Syntax bei beiden sehr große Ähnlichkeit mit der von C aufweist – wobei JavaScript automatisch Semikolons einfügt. Der Name JavaScript wurde aus Marketinggründen gewählt, um vom Hype um die Sprache Java zu profitieren. Java verhält sich zu JavaScript wie Ham zu Hamster.

Der erste Unterschied ist der, dass JavaScript interpretiert, Java aber kompiliert wird. Ein weiterer Unterschied besteht in der Typisierung: Während JavaScript dynamisch typisiert, erfolgt die Typisierung bei Java statisch. Außerdem unterscheiden sich die beiden Sprachen durch ihr Objektsystem.

JavaScript und rein funktionale Programmiersprachen

Im Gegensatz zu rein funktionalen Programmiersprachen wie Haskell, Miranda oder ML verfügt JavaScript über ein Objektsystem und verwendet Prototypen zur Modularisierung. In JavaScript können die Werte von Variablen verändert werden, was in Haskell als rein funktionale (purely functional) Sprache nicht möglich ist. Wenn in Haskell einmal eine Liste erzeugt wird, können die einzelnen Elemente nicht mehr abgeändert werden. Jeder Ausdruck kann in Haskell dabei direkt mit einem unveränderbaren Wert gleichgesetzt werden. Dieses Konzept wird auch Referential Transparency genannt. Statt die Liste zu verändern, wird hier eine neue Liste erstellt, die aus den Resultaten der Funktionsanwendung auf jedes Element der ursprünglichen Liste besteht (Kapitel 4 im Abschnitt »Parallele Berechnung auf Arrays mit Map« auf Seite 48). Funktionen verhalten sich daher in rein funktionalen Programmiersprachen wie mathematische Funktionen. Sie hängen nur von Definition und Eingabe ab und liefern für gleiche Eingaben stets gleiche Ergebnisse. In gemischten Sprachen mit funktionalen Anteilen wie JavaScript verhalten sie sich jedoch abhängig vom umgebenden Kontext und von vorher ausgeführten Programmteilen.

Aufbruch in die Welt der funktionalen Programmierung und der Gewürze

In diesem Buch beschreiben wir einige praktische Grundlagen des funktionalen Programmierens, und stellen immer wieder Analogien zum Kochen eines Currys her. Ähnlich wie bei den Currys gibt es viele verschiedene funktionale Programmiersprachen, die sich in bestimmten Aspekten unterscheiden. Bekannte funktionale Programmiersprachen sind Lisp, Haskell oder ML. Oft entstammen diese der akademischen Welt und sind beim täglichen Brötchenerwerb der Programmiererin und Programmierer nur in bestimmten Bereichen relevant, so schade das auch im Sinne der reinen Lehre sein mag.

Im folgenden Kapitel 2 werden wir uns verschiedene Abstraktionen vor Augen führen. Ein wichtiger Aspekt beim funktionalen Programmieren sind Funktionen höherer Ordnung. Dabei handelt es sich um Funktionen, die wiederum Funktionen als Argumente erhalten. In Kapitel 3 werden wir diese als Basisgrundlage kennenlernen, um dann in Kapitel 4 Funktionen höherer Ordnung auf Arrays anzuwenden.

Anschließend führt uns unsere kulinarische Reise zum Thema Rekursion in Kapitel 5, und in Kapitel 6 werden wir event-basierte Programmierung und Continuations kennenlernen. In den ersten Kapiteln können wir ohne Paradigmenwechsel und ohne das Erlernen einer neuen Programmiersprache in die funktionale Programmierung einsteigen.

In den anschließenden Kapiteln werden wir uns der theoretischen Seite widmen, zuerst in Kapitel 7 dem Lambda-Kalkül, der Grundlage fast aller Programmiersprachen. In Kapitel 8 werden wir mehr über Datentypen lernen und in Kapitel 9 werden wir auf Monaden eingehen, mit denen in rein funktionalen Programmiersprachen Seiteneffekte gekapselt werden und mit denen JavaScript einen weiteren Abstraktionsmechanismus erhält, mittels dessen sich zum Beispiel abstrakte Datentypen gut umsetzen lassen.

Im letzten Kapitel 10 schauen wir über den Tellerrand hinaus: wir erkunden noch unbekanntes Gebiet und führen interessantes weiterführendes Material an, von dem wir der Meinung sind, dass es sich lohnt, sich damit auseinanderzusetzen.

JavaScript ist auf eine gute Erweiterbarkeit des Sprachkerns ausgelegt. Daher sind mehrere Bibliotheken vorhanden, um JavaScript nachträglich an einen funktionaleren Programmierstil anzupassen. Diese Anpassungen können sogar als ganz eigene Sprachen daherkommen, wie wir im Kapitel 10 im Abschnitt »CoffeeScript« auf Seite 174 am Beispiel von CoffeeScript sehen werden. In diesem Buch möchten wir vor allem das Paradigma der funktionalen Programmierung und deren Konzepte kennenlernen und nicht eine bestimmte Bibliothek vorstellen. Auch wenn es sich lohnt, mit den funktionalen Bibliotheken für JavaScript herumzuspielen, haben wir auf ihren Einsatz verzichtet und zeigen lieber, wie die grundlegenden Funktionen selbst implementiert werden können.

Wir stellen fest, dass JavaScript einzigartig ist und Ideen aus verschiedenen Programmiersprachen und deren Ideologien vereint. JavaScript ist für viele ungewohnt und hält einige Fallstricke bereit, daher gibt es viele Programmiersprachen, die auf JavaScript aufbauen

und bei gleicher Expressivität/Semantik eine verständlichere Syntax versprechen. Die Syntax ist immer Geschmackssache! JavaScript und darauf aufbauende Sprachen bieten an überraschenden Stellen syntaktischer Zucker, also Syntaxerweiterungen zur Vereinfachung der Sprache. Als syntaktischen Zucker bezeichnet man alternative Schreibweisen, die sich durch umformulieren ("desugaring", entsüßen) auf die zugrundeliegende Kernsprache zurückführen lassen.

Wir genießen JavaScript am liebsten so wie Captain Jean-Luc Picard seinen Earl Grey: heiß und ungesüßt, so wie es dem britischen Adligen Charles Grey, dem zweiten Earl Grey, aus Indien vermittelt wurde.

Rezeptor: Über hundert Curry-Gerichte, funktional und automatisch erstellt

Um unsere Einführung abzuschließen, stellen wir hier noch ein JavaScript-Programm vor, das Curry-Rezepte generiert – zuerst eine »normale« imperative Version, und dann wird die Entwicklung hin zu einer funktionalen Version vollzogen.

Imperative Lösung

Ein Curry besteht aus drei verschiedenen Zutaten: einer feuchten Basis, verschiedenen Gewürzen und einer Hauptzutat. Wir haben als Beispiel einige Zutaten folgendermaßen definiert:

```
var base = [
  "Glasig angebratene Zwiebeln",
  "Naturjoghurt mit Crème fraîche versetzt",
  "Kokosnussmilch"
];
var spices = [
  "Zimt",
  "Kreuzkümmel",
  "Koriander",
  "Gewürznelke",
  "Kardamom – am besten sowohl grün als auch schwarz",
  "Schwarze Pfefferkörner",
  "Rote Chilischoten oder Chilipulver",
  "Ingwerpaste oder frischen Ingwer – kein Ingwerpulver",
  "Garam Masala",
  "Kurkuma"
];
var ingredients = [
  "Geflügel",
  "Lamm",
  "Blumenkohl und Kartoffeln",
  "Paneer",
  "Auberginen"
];
```

Der Programmcode des Rezeptgenerators sieht nun folgendermaßen aus:

```
console.log("Curry benötigt zunächst eine feuchte Basis.",
            "Ich schlage vor, folgende Basis zu benutzen:\n");
console.log( "* ", base[Math.floor(Math.random() * base.length)], "\n");
console.log("Die Gewürze werden in einer Pfanne aromageröstet",
            "Als mögliche Gewürze bieten sich an: \n");

for (var i = 0; i < 5; i++) {
  console.log("* ", spices[Math.floor(Math.random() * spices.length)], "\n");
}

console.log("Wir kommen jetzt zur Hauptzutat. Bei unserem Curry ist das\n");

console.log("* ", ingredients[Math.floor(Math.random() * ingredients.length)], "\n");

console.log("Die Hauptzutat wird in einer gesonderten Pfanne bei hoher Hitze angebraten",
            "und dann mit der flüssigen Basis und den Gewürzen zusammen bei niedriger",
            "Hitze geschmort,bis das Curry gut eingekocht und servierfertig ist.\n");
console.log("Guten Appetit!");
```

Unser Programmcode enthält durch `console.log` einige Ausgaben auf die Konsole und greift zudem mit Hilfe von `Math.random` auf zufällige Werte in den Arrays zu. Mittels `Math.random` wird hierbei ein zufälliger Wert zwischen 0 und 1 erzeugt. Multipliziert mit der Länge des jeweiligen Arrays wird damit ein Index des Arrays errechnet und mit Hilfe von `Math.floor` zu einer Ganzzahl gerundet.

Die Funktion `console.log` erhält eine beliebige Anzahl von Argumenten und gibt diese nacheinander auf der Konsole aus.

Die einzige Kontrollstruktur im Rezeptor ist die Schleife, mittels derer fünf Gewürze auswählt und ausgegeben werden. Leider müssen wir feststellen, dass es sich dabei nicht zwingend um fünf unterschiedliche Gewürze handelt. Ein weiteres Problem besteht darin, dass an drei verschiedenen Stellen das gleiche Codefragment enthalten ist, nämlich ein zufälliger Zugriff auf ein Array.

Als Nächstes stellen wir imperative Lösungsansätze für beide Probleme vor, bevor wir eine funktionale Lösung entwickeln:

Unterschiedliche Gewürze

Um unterschiedliche Gewürze auszuwählen, erstellen wir ein neues Array, in dem wir die schon ausgewählten Gewürze ablegen:

```
var taken = [];
while (taken.length < 5) {
  var choice = Math.floor(Math.random() * spices.length);
  if (taken.indexOf(choice) === -1) {
    taken.push(choice);
    console.log("* ", spices[choice], "\n");
  }
}
```

Anstelle der `for`-Schleife verwenden wir nun eine `while`-Schleife und prüfen bei jeder Gewürzwahl, ob das Gewürz bereits gewählt wurde. Wenn das nicht er Fall ist, fügen wir es in das `taken`-Array ein und geben es aus. Diese Lösung ist gleich mit mehreren Nachteilen verbunden: Es ist ein neues Array erforderlich, in dem wir nur vorübergehend Daten speichern, und zusätzlich ist die Laufzeit der Schleife nicht mehr klar zu bestimmen, da sie von der Rückgabe des Zufallsgenerators abhängt. Falls dieser immer den gleichen Wert zurückgibt (https://xkcd.com/221/), wird die Schleife nie fertig abgearbeitet.

Abstraktion in eine Funktion

Hier wollen wir Abstraktion nutzen, um nicht immer wieder die gleiche Zeile mit dem Zufallszahlenaufruf schreiben zu müssen, und ein neues abstraktes Element `rand` hinzufügen. Mit Abstraktionen werden wir uns ausführlicher in Kapitel 2 beschäftigen.

Für die Abstraktion definieren wir eine Variable, der wir dann eine Funktion zuweisen:

```
var rand = function (xs) {
  return xs[Math.floor(Math.random() * xs.length)];
};

rand(spices);
> 'Zimt'
```

Diese Funktion erwartet als Parameter ein Array `xs`, aus dem ein zufälliges Element ausgewählt und zurückgegeben wird. Die drei ähnlichen Aufrufe in unserem ursprünglichen Code können wir nun durch jeweils einen Aufruf von `rand` mit dem entsprechenden Array als Argument ersetzen.

Funktionale Lösung

Die funktionale Lösung unseres Curry-Problems sieht deutlich anders aus. Anstelle einer Schleife verwenden wir Funktionen höherer Ordnung, also Funktionen, die Funktionen als Argumente erhalten, siehe Kapitel 2 im Abschnitt »Abstraktion von einer Funktion: Funktionen höherer Ordnung« auf Seite 25.

Zunächst benötigen wir zwei Funktionen – die erste, `shuffle`, mischt ein Array wie ein Kartenspiel, und die zweite Funktion, `random_sample`, wählt zufällig ein Element aus einem gemischten Array aus:

```
function shuffle (xs) {
  var index = 0;
  var shuffled = [];
  xs.forEach(function (value) {
    var rand = Math.floor(Math.random() * index++);
    shuffled[index - 1] = shuffled[rand];
    shuffled[rand] = value;
  });
  return shuffled;
}
```

```
var random_sample = function (n, xs) {
  var shuffledList = shuffle(xs);
  return shuffledList.filter(function (i) {
    return shuffledList.indexOf(i) < n;
  });
};
```

Die Funktion `random_sample` erhält dabei zwei Argumente, eine ganze Zahl `n`, und ein Array `xs`. Dieses Array wird mittels der Funktion `shuffle` durchmischt. Danach werden alle Array-Elemente, die kleiner als `n` sind, zurückgegeben. Dazu wird das durchmischte Array gefiltert: der Array-Index wird mit `n` verglichen, und nur wenn dieser kleiner ist, wird das Element zurückgegeben.

In JavaScript funktionieren auf Strings auch die Array-Methoden.

Alternativ können wir auch die Funktion `slice` nutzen, die ein Start und optional auch ein Ende erhält und ein Teilstück eines Array mit den Elementen inklusive Start bis exklusive Ende zurückgibt. Für `"functional".slice(0, 3)` lautet die Rückgabe `fun`.

```
var random_sample = function (n, xs) {
  return shuffle(xs).slice(0, n);
}
```

Die Hilfsfunktion `pick` erzeugt Funktionen zur Auswahl einer Anzahl zufälliger Elemente aus einem Array. Sie erhält ein Array und gibt eine Funktion zurück, die eine Ganzzahl erwartet und nun diese Anzahl von Elementen aus dem Array unter Verwendung der `random_sample`-Funktion auswählt und auf der Konsole ausgibt:

```
var pick = function (xs) {
  return function (n) {
    return random_sample(n, xs).forEach(function (a) { console.log("* ", a, "\n"); });
  };
};
```

Im Programmcode rufen wir die Funktionen auf, die durch die Übergabe eines Arrays wie `spices` an die Funktion `pick` erzeugt wurden. Dabei übergeben wir die Anzahl der Array-Elemente, die wir auswählen möchten:

```
pick(base)(1);
> *  Naturjoghurt mit Crème fraîche versetzt

pick(spices)(5);
> *  Kurkuma
> *  Garam Masala
> *  Kardamom, am besten sowohl grün als auch schwarz
> *  Zimt
> *  Kreuzkümmel

pick(ingredients)(1);
> *  Blumenkohl und Kartoffeln
```

Schon bei diesem kleinen Programm sind wir der Ansicht, dass die funktionale Lösung einfacher zu lesen ist.

Weitere, viel komplexere Töpfe mit dem Gewürz der funktionalen Programmierung stehen noch auf dem Herd. Wir werden bei allen die Deckel lüften und hineinschnuppern. Zunächst soll es aber um die Grundlagen gehen: um grundlegende Gewürze und darum, was die Welt der indischen Kochkunst mit Ayurveda zu tun hat.

Komplettes Programm: Curry imperativ

```
var spices = [
  "Zimt",
  "Kreuzkümmel",
  "Koriander",
  "Gewürznelke",
  "Kardamom – am besten sowohl grün als auch schwarz",
  "Schwarze Pfefferkörner",
  "Rote Chilischoten oder Chilipulver",
  "Ingwerpaste oder frischen Ingwer – kein Ingwerpulver",
  "Garam Masala",
  "Kurkuma"
];
var base = [
  "Glasig angebratene Zwiebeln",
  "Naturjoghurt mit Crème fraîche versetzt",
  "Kokosnussmilch"
];
var ingredients = [
  "Geflügel",
  "Lamm",
  "Blumenkohl und Kartoffeln",
  "Paneer",
  "Auberginen"
];

var rand = function (xs) {
  return xs[Math.floor(Math.random() * xs.length)];
}

console.log("Curry benötigt zunächst eine feuchte Basis. Ich schlage vor, folgende",
            "Basis zu benutzen:\n");
console.log("* ", rand(base), "\n");
console.log("Die Gewürze werden in einer Pfanne aromageröstet. Als mögliche Gewürze",
            "bieten sich an:\n");

var taken = [];
while (taken.length < 5) {
  var choice = rand(spices);
```

– Fortsetzung –

```
  if (taken.indexOf(choice) === -1) {
    taken.push(choice);
    console.log("* ", choice, "\n");
  }
}

console.log("Wir kommen jetzt zur Hauptzutat. Bei unserem Curry ist das\n");

console.log("* ", rand(ingredients), "\n");

console.log("Die Hauptzutat wird in einer gesonderten Pfanne bei hoher Hitze angebraten",
            "und dann mit der flüssigen Basis und den Gewürzen zusammen bei niedriger",
            "Hitze geschmort, bis das Curry gut eingekocht und servierfertig ist.\n");
console.log("Guten Appetit!");
```

Komplettes Programm: Curry Funktional

```
function shuffle (xs) {
  var index = 0;
  var shuffled = [];
  xs.forEach(function (value) {
    var rand = Math.floor(Math.random() * index++);
    shuffled[index - 1] = shuffled[rand];
    shuffled[rand] = value;
  });
  return shuffled;
}

var random_sample = function (n, xs) {
  return shuffle(xs).slice(0, n);
};

var pick = function (xs) {
  return function (n) {
    return random_sample(n, xs).forEach(function (a) { console.log("* ", a, "\n");
});
  };
};

var spices = [
  "Zimt",
  "Kreuzkümmel",
  "Koriander",
  "Gewürznelke",
```

– Fortsetzung –

```
    "Kardamom – am besten sowohl grün als auch schwarz",
    "Schwarze Pfefferkörner",
    "Rote Chilischoten oder Chilipulver",
    "Ingwerpaste oder frischen Ingwer – kein Ingwerpulver",
    "Garam Masala",
    "Kurkuma"
  ];
  var base = [
    "Glasig angebratene Zwiebeln",
    "Naturjoghurt mit Crème fraîche versetzt",
    "Kokosnussmilch"
  ];
  var ingredients = [
    "Geflügel",
    "Lamm",
    "Blumenkohl und Kartoffeln",
    "Paneer",
    "Auberginen"
  ];

  console.log("Curry benötigt zunächst eine feuchte Basis. Ich schlage vor, folgende",
              "Basis zu benutzen:\n");

  pick(base)(1);

  console.log("Die Gewürze werden in einer Pfanne aromageröstet. Als mögliche Gewürze",
              "bieten sich an:\n");

  pick(spices)(5);

  console.log("Wir kommen jetzt zur Hauptzutat. Bei unserem Curry ist das\n");
  pick(ingredients)(1);

  console.log("Die Hauptzutat wird in einer gesonderten Pfanne bei hoher Hitze angebraten",
              "und dann mit der flüssigen Basis und den Gewürzen zusammen bei niedriger",
              "Hitze geschmort, bis das Curry gut eingekocht und servierfertig ist.\n");
  console.log("Guten Appetit!");
```

Nach diesem Überblick über unser Werkzeug, die Programmiersprache JavaScript, können wir die funktional erzeugten Curry-Rezepte genießen. Im nächsten Kapitel wird es grundsätzlicher: Wir sehen uns an, welche Voraussetzungen es für das Design funktionaler Programme gibt und wie wir Techniken der Abstraktion nutzen können, um komplexe Programme klar zu strukturieren.

```
λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s z)))))))))))))))))))))))))))))))))))))))))))) (λ z → z (λ s → λ
z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
z))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

KAPITEL 2

Abstraktionen

Denken wir in Europa an die indische Küche und ihre Gewürze, fällt uns oft das gelbe Curry-Gewürz auf der Currywurst ein. Und damit beginnt schon das erste Missverständnis, denn als eigenes Gewürz gibt es »Curry« in Indien gar nicht. Das Curry-Gewürz wurde erst durch die englische Kolonialmacht als Instantmischung der indischen Küche exportiert. Curry wird aus verschiedenen Bestandteilen komponiert, wobei der Geschmack der Zutaten aufeinander abgestimmt wird. Es gibt zwar einen Currybaum, dessen Blätter auch beim Kochen benutzt werden, aber die Curryblätter sind nur ein Bestandteil unter vielen bei der Currymischung.

Im indischen Verständnis ist Kochen gleichbedeutend mit der Komposition von Gewürzen und stellt damit eine ganz andere Frage als die europäische Küche. Entsprechend kommt die indische Küche zu anderen Lösungswegen, dass heißt Gerichten. Dies ist dem Programmieren sehr ähnlich: bei der objektorientierten Programmierung geht es eher um das Zusammensetzen von Objekt- oder Prozedurblöcken. Bei der funktionalen Programmierung steht das Komponieren von flüchtigen Funktionen ohne Seiteneffekte im Vordergrund. Zusammen ergeben sie ein Aromenkonzert, das aber ausserhalb des Gesamtarrangements nicht unbedingt genießbar ist. Hier wird also nicht ein Stück Fleisch oder eine Gemüsesuppe zubereitet und nachgesalzen, sondern allein aus dem Zusammenspiel ergibt sich die Mischung, die in Indien »Masala« heisst.

Ayurveda (Sanskrit आयुर्वेद) bedeutet auf Deutsch »Wissen vom Leben« und bildet die Grundlage einer Gewürzkomposition. Mit der Theorie dahinter lassen sich ganze Bücher füllen, die von naturwissenschaftlich bis esoterisch-religiös alle Aspekte dieser Lehre abdecken. Aber für ein Buch über das Programmieren beschränken wir uns hier auf eine vereinfachte Form: Gewürze besitzen Eigenschaften, die sie auszeichnen und miteinander kombinierbar machen. Dabei können diese sich ergänzen und ausgleichen, wirken aber nicht alleine, sondern nur gemeinsam als Mischung.

Zimt, Nelken und Kardamom gelten im Ayurveda als »warme« Gewürze. Beim Plätzchenbacken im Winter geben sie das weihnachtliche Aroma, was vielleicht sogar eine Erinnerung an die Klassifizierung dieser Gewürze nach dem Konzept des Ayurveda ist – sie sind dafür verantwortlich, dass uns wohlig warm ums Herz wird.

Kreuzkümmel, Koriander und Kurkuma bilden zusammen, besonders in der nordindischen Küche, eine fertige Mischung. Koriander ist süßlich und wird durch den leicht bitteren Kurkuma geschmacklich ergänzt. Beide wiederum reagieren aromatisch mit den herben Kreuzkümmelsamen. Diese drei Gewürze formen im Zusammenspiel also eine Art Funktionskomposition, auf die ein Gericht aufbauen kann.

Knoblauch und Ingwer haben mit ihren Wirkungen ähnliche Funktionen für ein Gericht. Sie sind frisch, aromen-verstärkend und leicht zitronig-sauer. Sie können also andere Gewürzkompositionen unterstreichen oder auch erst sichtbar machen.

Bockshornklee (Methi) ist in Deutschland nicht sehr bekannt. In Nordamerika ist der seltsam bittere Geschmack oft in Zahnpasta zu finden. In der Küche des mittleren Ostens finden wir das Gewürz häufig, da die Pflanze auch auf Böden mit sehr hohem Versandungsgrad wächst. Als bittere Medizin mit typisch gelber bis oranger Farbe ist Bockshornklee in vielen Currys zu finden.

Chilis geben das Feuer und entscheiden darüber, ob ein indisches Gericht scharf oder sehr scharf wird. Dabei kam die Chili-Pflanze erst mit portugiesischen Seefahrern aus Amerika nach Indien. Vielleicht ist das eine gute Erinnerung daran, dass es in der Küche und beim Programmieren so etwas wie eine »reine Lehre« nur selten gibt. Auf neue Ereignisse oder Zeiten muss immer auch spontan reagiert werden können.

Ayurveda ist die theoretische Grundlage der indischen Kochkunst, auf deren Basis Gerichte komponiert werden. Wie bei jeder Theorie sollte sie umfassend genug sein, um die Welt möglichst facettenreich abzubilden, aber in der Praxis können sich durchaus interessante Differenzen ergeben.

Modellierung: Theorie und Praxis

Computerprogramme können als Modelle gesehen werden, die einen Vorgang der realen Welt beschreiben (mehr dazu in Kapitel 5 im Abschnitt »Vom Problem zum Programm: Spezifikation, Ein- und Ausgabe« auf Seite 65). Auch wenn man vor Beginn eines Projektes über dessen Modellierung ausgiebig nachgedacht hat, sieht die Umsetzung oft ganz anders aus: Die Planung des Projekts in der Theorie unterscheidet sich von den Anforderungen in der Praxis.

In jedem gewachsenen Programmierprojekt kommt irgendwann der Zeitpunkt, an dem die Funktionalität erweitert werden soll, aber der Code nicht auf eine Erweiterung ausgelegt ist. Der Code muss nun entweder umorganisiert und abstrahiert werden (Refactoring), oder kopiert und an der entsprechenden Stelle verändert werden (Copy und Paste). Zwar muss man beim Refactoring etwas nachdenken, aber dafür ist der Code dann wiederverwendbar. Bequem Copy und Paste zu benutzen bringt auf lange Sicht nichts, da der Code durch Wiederholung unübersichtlich und nicht mehr wartbar ist.

Wenn nicht genügend Tests implementiert sind, ist es nach dem Umorganisieren fraglich, ob das Programm noch funktioniert. Einige Programmteile funktionierten vielleicht

nur aufgrund von anderem fehlerhaften Stellen, oder unter Annahmen, die vom umorganisierten Programm missachtet werden. Tests können jedoch immer nur zeigen, dass ein Programm Fehler enthält, und nicht, dass es fehlerfrei ist. *Testing shows the presence, not the absence of bugs* – Edsger W Dijkstra, 1969.

Beim Kopieren und Anpassen wächst der Programmcode ohne großen zusätzlichen Informationsgehalt. Wird nun ein Fehler in dem ursprünglichen Codestück gefunden, müssen wir auch alle Kopien anpassen, die den Fehler enthalten. Dies kann relativ einfach sein, wenn der Code vor nicht allzu langer Zeit geschrieben wurde und noch präsenter ist. In den meisten Fällen können die kopierten Codefragmente jedoch nur durch mühsames Suchen und Lesen des Codes gefunden werden. Der Programmcode ist durch das Kopieren unverständlich und lässt sich nicht mehr pflegen und aktuell halten. Im schlimmsten Fall muss das Programm neu geschrieben werden. Deshalb ist das Refactoring dem Kopieren und Anpassen auf lange Sicht vorzuziehen, auch wenn es im ersten Moment mehr Arbeit bedeutet. Dabei gilt in der Regel, den Code so konkret wie nötig und so abstrakt wie möglich zu halten, damit der Code aufgeräumt bleibt und sich leicht wiederverwenden lässt.

Beim Essen ist das ähnlich. Wenn wir gerne Müsli mit Nüssen essen, unser neuer Mitbewohner aber lieber Müsli mit Rosinen, weil er Nussallergie hat, werden wir kein Nussmüsli für uns kaufen und die Nüsse für ihn heraussuchen. Wir könnten dabei ja leicht eine übersehen, und schon ginge es dem Mitbewohner schlecht. Anaphylaktischer Schock! Stattdessen kaufen wir ein Basismüsli, abstrahieren also zunächst von allen speziellen Bedürfnissen, und fügen Nüsse für uns und Rosinen für ihn hinzu. Das Basismüsli können wir uns alle teilen, gemeinsam einen grossen Sack kaufen, und dadurch Geld sparen.

Im Folgenden schauen wir uns nun einige Abstraktionsmechanismen an, die uns helfen, unseren Programmcode kürzer, übersichtlicher und wiederverwendbarer zu schreiben.

Abstraktion von einem Wert: Variablen

Die einfachste Art der Abstraktion in der Programmierung ist so allgegenwärtig, dass man sie oft gar nicht bemerkt. Anstatt einen Wert immer wieder hinzuschreiben, können wir ihn in einer Variable speichern und über den Variablennamen aufrufen:

Stellen wir uns vor, wir haben eine Einkaufsliste von Gemüsen, repräsentiert als Array von Strings. Wir möchten dieses Array zusammenfassen und ausgeben.

```
["Kartoffeln", "Auberginen", "Kichererbsen", "Kaiserschoten", "Paprika"].join(", ");
> "Kartoffeln, Auberginen, Kichererbsen, Kaiserschoten, Paprika"
```

Dann möchten wir ein neues Gemüse hinzufügen.

```
["Kartoffeln", "Auberginen", "Kichererbsen", "Kaiserschoten", "Paprika"].push("Karotten");
> 6
```

Es ist ganz schön umständlich, das Gemüse-Array noch einmal hinzuschreiben, oder? Als Programmierfüchse definieren wir uns eine Variable, um dies zu vermeiden:

```
var veggies = ["Kartoffeln", "Auberginen", "Kichererbsen", "Kaiserschoten", "Paprika"];
```

Die Variable können wir nun bequem und beliebig oft wiederverwenden:

```
veggies.join(", ");
> "Kartoffeln, Auberginen, Kichererbsen, Kaiserschoten, Paprika"
veggies.push("Karotten");
> 6
```

Diese Abstraktion ist bereits sehr mächtig, da sie uns viel Schreibarbeit ersparen kann. Außerdem können wir durch die Namensgebung der Variablen das Programm strukturieren und erklären.

Abstraktion von einem Codeblock: Funktionen

Die zweite Art der Abstraktion ist ebenfalls allgegenwärtig. Wir wollen sie uns nur noch einmal klar vor Augen führen. Anstatt einen Codeblock zu wiederholen, können wir ihn in eine Funktion verpacken, und über den Funktionsnamen mit verschiedenen Parametern für verschiedene Speziallfälle aufrufen.

Stellen wir uns vor, wir haben für unser Curry folgende Gewürz-Objekte:

```
var spices = [{
   "name": "Nelken",
   "state": "gemahlen",
   "quantity_in_grams": "2",
   "active_compound": "Eugenol"
}, {
   "name": "Zimt",
   "state": "Gemahlen",
   "quantity_in_grams": "1",
   "active_compound": "Cinnamaldehyde"
}, {
   "name": "Schwarzer Pfeffer",
   "state": "gekörnt",
   "quantity_in_grams": "11",
   "active_compound": "Piperine"
}];
```

Nun möchten wir ausgeben, welche Gewürze vorrätig sind:

```
console.log("Ich habe " + spices[0].quantity_in_grams + " Gramm " +
             spices[0].name + " und zwar " + spices[0].state + ".");
console.log("Ich habe " + spices[1].quantity_in_grams + " Gramm " +
             spices[1].name + " und zwar " + spices[1].state + ".");
console.log("Ich habe " + spices[2].quantity_in_grams + " Gramm " +
             spices[2].name + " und zwar " + spices[2].state + ".");
```

Mithilfe der Abstraktion in eine Funktion wird dieser Codeblock zu:

```
function showSpice (spice) {
  console.log("Ich habe " + spice.quantity_in_grams + " Gramm " +
              spice.name + " und zwar " + spice.state + ".");
}

showSpice(spices[0]);
showSpice(spices[1]);
showSpice(spices[2]);
```

In diesem einfachen Beispiel hätte es auch eine Schleife getan. Sobald es komplizierter wird, weil die Funktionen länger werden oder Rekursion (in Kapitel 5) ins Spiel kommt, wird die Funktionsabstraktion aber sehr wichtig.

Auch hier erleichtert ein treffender Funktionsname das Verständnis des Programms sehr.

Abstraktion von einer Funktion: Funktionen höherer Ordnung

Ein häufig auftretendes Programmierproblem ist das Sortieren. Wir möchten dabei Arrays verschiedener Typen sortieren können (siehe auch Kapitel 8 im Abschnitt »Polymorphismus« auf Seite 137). Dies können Arrays von Zahlen, Strings oder gar zusammengesetzten Objekten mit mehreren Eigenschaften sein. Dabei haben die unterschiedlichen Datentypen der Felder, nach denen sortiert werden soll, jeweils andere Ansprüche an eine Sortierfunktion.

Stellen wir uns vor, wir möchten die Gewürz-Objekte nach verschiedenen Feldern sortieren.

Für das aufsteigende Sortieren von Strings, wie etwa dem Gewürznamen, stellt JavaScript schon die Methode sort bereit, die wir direkt benutzen können.

```
['Nelken', 'Zimt', 'Schwarzer Pfeffer'].sort();
> [ 'Nelken', 'Schwarzer Pfeffer', 'Zimt' ]
```

Was aber, wenn wir die Gewürze nach ihrer vorhandenen Menge in Gramm sortieren möchten? Dann müssen wir zuerst die Angaben in den richtigen Typ umwandeln, indem wir sie als Zahl, hier einen Integer (Ganzzahl), einlesen oder auch parsen. Diese Funktion muss auf jeden Eintrag des zu sortierenden Arrays angewandt werden. Es könnte auch sein, dass wir die Namen unabhängig von Groß- und Kleinschreibung sortieren möchten. Auch hier müssen wir vor dem Vergleichen eine Konvertierungsfunktion auf jeden Eintrag des Arrays anwenden, die den Eintrag in Upper- oder Lowercase umwandelt, damit die Groß- und Kleinschreibung keine Rolle beim Vergleichen spielt.

Von dieser Konvertierungsfunktion können wir abstrahieren, und sie als Argument in die Sortierungsfunktion übergeben. Dies geht in JavaScript ohne Probleme, da Funktionen als Werte aufgefasst werden, und deshalb Funktionen wiederum andere Funktionen als Argumente haben können. Solche Funktionen nennt man Funktionen höherer Ordnung.

Unsere Sortierfunktion `sort_by` bekommt den Namen der Eigenschaft, nach der wir sortieren, und optional eine Funktion `primer`, die vor dem Sortieren auf die Werte des Feldes angewandt wird. Falls wir `primer` nicht übergeben, wird direkt der Wert des Feldes verglichen.

In JavaScript sind alle Argumente optional. Wenn ein Argument nicht übergeben wird, ist der Wert `undefined`, daher funktioniert der Test auf `primer` hier.

Ternärer Operator: Die Syntax `test ? a : b` ist eine Kurzschreibweise für eine Bedingung. Wenn der Test `test` erfüllt ist, wird `a` ausgeführt, andernfalls `b` -- genau wie `if (test) a else b`.

```
var sort_by = function (field, primer) {
  var key = function (x) { return (primer ? primer(x[field]) : x[field]); };
  return function (y,z) {
    var A = key(y);
    var B = key(z);
    return ( (A < B) ? -1 : ((A > B) ? 1 : 0));
  };
};
```

Beim Sortieren nach der Menge in Gramm fällt auf, dass die Zahlen als String sortiert werden, und daher der schwarze Pfeffer vor den Nelken kommt:

```
// erster Versuch der Sortierung
spices.sort(sort_by('quantity_in_grams'));
```

Die Elemente des Feldes können als Integer eingelesen werden, indem die Funktion `parseInt` als zweites Argument an `sort_by` übergeben wird. Besser ist hier eine Extrafunktion `parseIntBase10`, um eine Zahl zur Basis 10 zu parsen. Die Übergabe der Basis an `parseInt` verhindert, dass `parseInt` das Zahlensystem aus dem Format des Eingabestrings errät.

```
// ParseInt mit Basis 10
var parseIntBase10 = function (x) { return parseInt(x, 10); };
```

Um Überraschungen zu vermeiden, geben wir die Basis immer mit an. In neueren JavaScript-Versionen soll deshalb auch das Erraten der Basis eingeschränkt werden (*https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/parseInt*).

```
// Richtige Sortierung nach Menge in Gramm als Zahl
spices.sort(sort_by('quantity_in_grams', parseIntBase10));
  { name: 'Zimt',
    state: 'Gemahlen',
    quantity_in_grams: '1',
    active_compound: 'Cinnamaldehyde' },
> [ { name: 'Nelken',
    state: 'gemahlen',
    quantity_in_grams: '2',
    active_compound: 'Eugenol' },
```

```
{ name: 'Schwarzer Pfeffer',
  state: 'gekörnt',
  quantity_in_grams: '11',
  active_compound: 'Piperine' } ]
```

Beim Zimt haben wir *Gemahlen* groß geschrieben, was die Sortierung nach dem Feld `state` durcheinanderbringt. Wir wollen hier unabhängig von Groß- und Kleinschreibung sortieren, indem wir vor dem Vergleichen nach Uppercase konvertieren.

```
// Sortierung nach Zustand, unabhängig von Groß- und Kleinschreibung
spices.sort(sort_by('state', function (a) { return a.toUpperCase(); }));
```

(Quelle: *http://stackoverflow.com/questions/979256/how-to-sort-an-array-of-javascript-objects*)

Funktionen höherer Ordnung wie in unseren Beispielen gibt es auch in vielen anderen Programmiersprachen wie Scala, Lisp und Haskell, jedoch nicht in Java oder C#.

Wir greifen auf Funktionen höherer Ordnung zurück, um Funktionen möglichst klein und übersichtlich zu halten und sie einfacher für wiederkehrende Operationen auf Datenstrukturen benutzen können. In Kapitel 4 kombinieren wir Arrays und Listen mit Funktionen höherer Ordnung. Auch auf anderen Datenstrukturen wie Bäumen oder Graphen sind Funktionen höherer Ordnung anwendbar. Solche Funktionen durchlaufen zum Beispiel die Datenstruktur und führen eine Funktion auf jedem Element aus. Das rekursive Durchlaufen einer Datenstruktur, die strukturelle Rekursion, besprechen wir in Kapitel 5.

Auch beim Kochen werden größere Funktionen in kleinere aufgebrochen. Wenn man ein amerikanisches Frühstück bestellt, wird man gefragt, wie man seine Eier haben möchte: als Rühr- oder Spiegelei? Das Spiegelei gewendet? Die Funktion »Frühstück machen« bekommt also die Funktion »Ei zubereiten« als Argument.

Ein Beispiel aus der Welt des Currys ist ein Grundgericht, das immer wieder neu variiert werden kann, etwa Vindaloo. Das Gericht entstand in Goa im Süden Indiens, das bis in die 1960er Jahre unter portugiesischer Kolonialverwaltung stand. Aus Portugal importiert wurde dabei eine typische Zubereitungsart für Schweinefleisch, das Marinieren des Fleisches in Wein bzw. Weinessig zusammen mit Knoblauch und Gewürzen, auf portugiesisch *carne em vinha de alhos*.

Im christlich geprägten Goa kommt das sauer-scharfe Schweinefleisch an besonderen Festtagen auf den Tisch. Von dort breitete es sich über den gesamten indischen Subkontinent aus. Dabei erfuhr Vindaloo eine linguistische und kulinarische Transformation: aus *vinha de alhos* wurde *vindalho* und später *Vindaloo*. Und statt Schweinefleisch wird heute auch oft Geflügel oder Lamm (bei Hindus und Muslimen) verwendet. Als Gewürze zum Einlegen werden heute Ingwer, Chilischoten, Kreuzkümmel, Pfefferkörner, Kardamom, Nelken, Piment, Tamarinde, Zimt, Senfsamen, Bockshornklee, Koriander und Kurkuma verwendet. Es gibt zudem viele Variationen von Vindaloo-Currypasten.

Die Funktion `Vindaloo` kann also Schweinefleisch-Vindaloo, Lamm-Vindaloo oder Geflügel-Vindaloo erzeugen.

Rezept: Schweinefleisch-Vindaloo aus Goa

Für 4 Personen:

- 4 Knoblauchzehen
- 1 Stück frischer Ingwer (oder Ingwerpüree)
- mindestens 4-6, gerne noch mehr getrocknete Chilischoten
- 6 grüne Kardamomkapseln
- 1 TL schwarze Pfefferkörner
- 6 Nelken
- 1 Stück Zimtrinde (5-7 cm)
- 1 TL Kurkumapulver
- Salz
- 1 EL brauner Zucker
- 800g Schweinegulasch, in größere Würfel geschnitten
- 1 EL Garam Masala
- 3 große Zwiebeln
- mindestens 2-3, gerne noch mehr frische grüne Chilischoten
- 2 EL Öl
- Essig

Den Knoblauch und Ingwer schälen und zusammen mit den getrockneten Chilischoten grob zerhacken. Die Samen aus den Kardamomkapseln herauskratzen und mit den Pfefferkörnern, Nelken und Zimt in einem Mörser fein zerreiben. Knoblauch, Ingwer, Chilis und die zermahlenen Gewürze mit 100 ml Essig mit einem Pürierstab fein pürieren. Mit Kurkuma, etwas Salz und dem Zucker verrühren.

Die Schweinefleischwürfel mit der Mischung bedecken und in einer Schüssel gut mischen. Zum Marinieren kommt das Fleisch etwa 12 Stunden in den Kühlschrank, bis es durchgezogen ist.

Am nächsten Tag Öl in einer Pfanne erhitzen und die Zwiebeln und die grünen Chilischoten klein schneiden. Bei mittlerer Hitze Zwiebeln mit Garam Masala und Chilis glasig anbraten. Dann das marinierte Fleisch zusammen mit der Marinade hinzugeben und etwa 5 Minuten anbraten. Mit einem ordentlichen Schuss Essig und 450 ml Wasser auffüllen und bei geschlossenem Deckel etwa 45 Minuten köcheln lassen. Danach den Deckel abnehmen und 10-15 Minuten bei hoher Hitze einkochen lassen, bis eine sämige, gulaschartige Sauce entsteht. Guten Appetit!

Vindaloo ist ein relativ aufwändiges Gericht, das vor allem zu Festtagen gegessen wird. Im Alltag kommen in Indien bescheidenere Gerichte auf den Tisch. Mit den Grundlagen, die wir für das Kochen mit Funktionen und indischen Gewürzen brauchen, wollen wir uns im nächsten Kapitel beschäftigen.

```
(λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s
z))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

KAPITEL 3

Ein Topf mit Curry

In Gurinder Chadhas Film »Bend it like Beckham« aus dem Jahr 2002 bewegt sich Jesminder »Jess« Bhamra in Großbritannien zwischen zwei Kulturen: Es gibt ihre indische Mutter, die ihr die Grundbegriffe der indischen Küche beibringen will, damit sie irgendwann eine gute indische Braut abgibt. Zudem ist Jess von Fußball begeistert. Heimlich schleicht sie sich zum Training und hofft, irgendwann als Fußballprofi Karriere machen zu können. »Wer will schon Aloo Gobi kochen, wenn man den Ball wie Beckham kicken kann?« seufzt sie. Dieses einfache Curry aus Kartoffeln (Aloo) und Blumenkohl (Gobi) ist nicht nur in diesem Film sehr prominent, sondern auch generell eine gute Einführung in die indische Küche.

Abbildung 3-1: Aloo Gobi (Foto von Paul Goyette)

Rezept: Aloo Gobi

Für 4 Personen:

- 500g Blumenkohl
- 400g Kartoffeln
- 3 reife Tomaten
- 1 Zwiebel
- 1 Stück Ingwer (ca. 5cm)
- Ghee
- Kreuzkümmelsamen
- Koriander (gemahlen)
- Garam Masala
- Chilipulver
- Koriandergrün (gehackt)

Die Blätter von Blumenkohl abschneiden und die einzelnen Röschen vom Strunk brechen. Dabei die großen Röschen in mundgerechte Stücke schneiden. Die Kartoffeln schälen, waschen, und in etwa 4 cm große Stücke schneiden. Tomaten waschen und klein würfeln. Die Zwiebel schälen und ebenfalls klein würfeln. Den Ingwer schälen und in dünne Stifte schneiden. In einer beschichteten Pfanne zwei Esslöffel Ghee heiß werden lassen. Den Blumenkohl hineingeben und einmal gut rühren. Salzen und bei mittlerer Hitze etwa fünf Minuten rundum leicht braun anbraten. Aus der Pfanne nehmen und noch einmal zwei Esslöffel Ghee heiß werden lassen. Die Kartoffeln darin schwenken, fünf Minuten anbraten und salzen. Die Zwiebel und einen halben Teelöffel Kreuzkümmelsamen dazugeben. Dann etwa fünf Minuten weiter braten. Ingwer, einen dreiviertel Teelöffel gemahlenen Kreuzkümmel, einen dreiviertel Teelöffel gemahlenen Koriander, einen dreiviertel Teelöffel Garam Masala und einen viertel Teelöffel Chilipulver unterrühren und eine Minute mitbraten. Tomaten dazugeben und zwei Minuten weiterrühren. 300 ml Wasser hinzugießen, den Blumenkohl unterheben, Deckel auflegen und alles 25 Minuten bei kleiner Hitze gar kochen. Mit Salz und Zucker abschmecken und mit zwei Esslöffel gehacktem Koriandergrün garnieren.

Niemand fängt beim Fußballspiel gleich als David Beckham an und auch auf dem Weg zur indischen Köchin gilt es zunächst, die einfachen Rezepte für die tägliche Mahlzeit zu meistern, wie etwa das Aloo Gobi.

Auch bei der funktionalen Programmierung halten wir, bevor wir mächtige Funktionen höherer Ordnung kennenlernen, kurz inne und betrachten einige grundlegende Konzepte.

Präfix, Infix, Stelligkeit

Da wir im Folgenden viel mit Funktionen arbeiten werden, müssen wir uns einig werden, wie wir diese notieren wollen. Dazu sollten wir zunächst festlegen, wo der Name der Funktion stehen soll. Beim Programmieren und im Mathematikunterricht schreiben wir meistens in Präfix-Notation, also den Funktionsnamen zuerst, und die Argumente dahinter.

Hier ein Beispiel in Javascript:

```
function hello (x) { return "Hello!"; }
```

Hier ein Beispiel aus der Mathematik, eine Funktion, die konstant `1` für alle Eingaben zurückgibt:

```
f(x) = 1
```

Was ist nun aber mit einem Ausdruck wie `f(x) = x + 1`? Ist hier + nicht auch eine Funktion? Als mathematischer Operator erhält + eine Sonderstellung, und wird in der Infix-Notation geschrieben. Der Operator steht hier zwischen seinen Argumenten. Und in JavaScript ist dies genau so, wie wir es aus der Mathematik kennen:

```
function f (x) { return x + 1; }
```

Durch eine Funktionsdefinition können wir die Infix- in eine Präfixschreibweise umwandeln. Dies geht beispielsweise mit einer anonymen Funktion (siehe Kapitel 1 im Abschnitt »Anonyme Funktionen« auf Seite 4) im umgebenden Kontext:

```
[..] function (a, b) { return a + b; } [..]
```

Schlauer ist es allerdings, die Funktion `plus` separat zu definieren, da wir sie dann wiederverwenden können.

```
function plus (a, b) { return a + b; }
```

Somit erhalten wir durch `2 + 3` das gleiche Ergebnis wie durch `plus(2, 3)`. Wir schreiben es nur ein wenig anders.

Die Definition eigener Infix-Operatoren wird in vielen Sprachen (C++, Haskell und anderen) unterstützt, ist allerdings in JavaScript nicht vorgesehen.

Sehr ungewöhnlich, aber nicht unmöglich ist die Emulation von Infix-Operatoren für JavaScript. *http://www.techtangents.com/emulating-infix-operator-syntax-in-javascript/* Alternativ gibt es auch Tricks, um vorhandene Operatoren zu überladen. h*ttp://web.archive.org/web/20090406105627/ http://blogger.xs4all.nl/peterned/archive/2009/04/01/462517.aspx*

Neben der Präfix-Schreibweise für Funktionen, der Infix-Schreibweise für Operatoren, und der Mixfix-Schreibweise für `if .. then .. else` und den ternären Operator kann man auch die Stelligkeit von Funktionen unterscheiden. Dabei klassifiziert man Funktionen in unäre oder einstellige, binäre oder zweistellige Funktionen, und so weiter, jeweils anhand der Anzahl ihrer Argumente.

So wie die Schreibweise kann auch die Stelligkeit von Funktionen umgewandelt werden. Dies kann sinnvoll sein, um die Programmierung auf ein einfacheres Grundmodell zurückzuführen. Im Lambda-Kalkül, dem grundlegenden Mechanismus aller funktionaler Programmiersprachen, das wir in Kapitel 7 behandeln, gibt es ausschließlich Funktionen mit je einem Argument.

Curryfizierung und teilweise Anwendung von Funktionen

Jede `n`-stellige Funktion (Funktion mit `n` Argumenten) lässt sich in `n` einstellige Funktionen zerlegen. Dieser Vorgang wird Curryfizierung genannt, nach Haskell B. Curry, mehr zu Herrn Curry kann man im Kasten nachlesen.

Curry? Was ist Curry? *Wer* ist Curry?

Curry ist nicht nur ein leckeres Eintopfgericht mit unzähligen Variationen.

Haskell Brooks Curry war ein Mathematiker, dessen Forschung im Bereich der Logik zu der Theorie der formalen Systeme und zum Logikkalkül führte, also zu Regelsystemen, mit denen man Aussagen aus Startbedingungen (Axiomen) ableiten kann.

Geboren wurde Curry 1900 in Millis, Massachusettes. Seine Eltern waren Präsident und Dekanin einer Rhetorikschule, der School of Expression (deshalb gibt es auch das Buch »The Haskell School of Expression« ;-)) in Boston. Dort wurde Eluktion gelehrt, das ist die Kunst der Übertragung der Gedanken in Worte, ein Teilgebiet der Rhetorik. Curry interessierte sich in der High School noch nicht für Mathematik. Mit 16 ging er nach Harvard, und schrieb sich dort für Medizin ein. Und im ersten Jahr hörte er im Rahmen dieses Studiums seine erste Mathematikvorlesung.

Als der 1. Weltkrieg ausbrach, war Curry der Meinung, Mathematik sei für die Kriegsführung seines Landes nützlicher als Medizin. Er wechselte in die Mathematik und absolvierte zudem eine Armeeausbildung für Studenten. Als der Krieg vorbei war, beendete Curry sein Studium im Alter von 20 Jahren vorerst mit einem Bachelor of Arts, und trat aus der Armee aus.

Er arbeitete im Folgenden bei General Electric, wobei er neben seiner Arbeit Elektrotechnik am MIT (Massachusetts Institute of Technology) studieren durfte. Jedoch stellte ihn die Vorgehensweise der Lehrenden in den Vorlesungen nicht zufrieden. Es wurde ihm nur beigebracht, wie man ein Ergebnis bekommt. Er aber wollte auch genau wissen, *warum* ein Ergebnis korrekt ist. Er merkte, dass er sich von der Anwendung der Mathematik entfernen, und zu den reinen Wissenschaften wenden wollte. Dazu ging er zurück nach Harvard, um dort Physik zu studieren. Seinen Master in Physik absolvierte er im Alter von 24 Jahren.

– Fortsetzung –

Während seiner Entdeckungsreise durch die Wissenschaften hatte Curry nicht nur mit der Themenfindung alle Hände voll zu tun. Nach dem Tod von Vater und Mutter übernahm er auch noch die Verwaltung der School of Expression bis zum Verkauf der Schule. Schließlich erkannte er, dass sein Thema eigentlich doch eher die Mathematik war, und begann seine Doktorarbeit in der Mathematik in Harvard.

Hatte Curry mit seiner Doktorarbeit in Mathematik endlich ein Thema gefunden? Nach der Meinung seines Betreuers sollte er sich mit der Theorie der Differentialgleichungen beschäftigen. Stattdessen las er aber lieber Bücher über Logik. Curry fragte diverse Professoren in Harvard und am MIT, ob er zu ihnen in das Themengebiet der Logik wechseln könne. Jedoch rieten ihm alle davon ab. Dann las er das Buch »Principia mathematica« von Alfred Whitehead und Bertrand Russell, und dies brachte ihn auf die Idee, spezielle Funktionen höherer Ordnung, die Kombinatoren (Kapitel 5 im Abschnitt »Rekursion über die Ein- und Ausgabedaten« auf Seite 76 und Kapitel 7 im Abschnitt »Rekursion im Lambda-Kalkül: Fixpunktkombinatoren« auf Seite 122) zu benutzen, die im ersten Teil des Buches vorgestellt wurden. Er fragte erneut herum, bei wem er eine Doktorarbeit in der Logik schreiben könne, und schilderte dabei seine neue Idee. Diesmal bekam er eine erfreulichere Antwort: Norbert Wiener vom MIT sagte ihm »Vermeide Logik, bis du damit etwas zu sagen hast, aber jetzt hast du etwas zu sagen«.

Curry hängte die Differentialgleichungen an den Nagel, und wandte sich der Logik zu. Zuvor lehrte er aber erst einmal ein Jahr in Princeton, um seine Idee zu konkretisieren. Dort entdeckte er Moses Schönfinkels Artikel »Über die Bausteine der mathematischen Logik«, welcher ähnlich seiner eigenen Idee Kombinatoren in der Logik benutzt. Der Artikel beschreibt im Grunde, was wir in diesem Kapitel als Curryfizierung kennenlernen.

Sein Professor Oswald Veblen, mit dem er seine Pläne diskutierte, bestärkte ihn in seiner Idee. Moses Schönfinkel war jedoch inzwischen in der Psychiatrie. Daher konnte er nicht weiter forschen, und auch nicht Currys Doktorvater werden. Auf Veblens Rat ging Curry schließlich zu den Professoren Bernays und Hilbert nach Göttingen. Vor seiner Reise heirateten Mary Virginia Wheatley und Haskell Brooks Curry. Sie hatten sich in der School of Expression kennengelernt, und gingen zusammen nach Deutschland.

Nach einem weiteren Jahr Forschung verteidigte Curry im Sommer 1929 seine Doktorarbeit »Grundlagen der kombinatorischen Logik« in Göttingen. Seine Dissertation wurde 1930 im American Journal of Mathematics veröffentlicht.

Die Currys gingen in die USA zurück und Haskell begann, an der Pennsylvania State University zu arbeiten. Sie gründeten eine Familie, und hatten Glück, dass Haskell eine Stelle an der Universität bekommen hatte. Seit 1929 hatte die »große Depression« in Amerika die Job-Aussichten für Logiker nicht gerade verbessert.

Curry baute in den folgenden Jahren die Theorie der Kombinatoren immer weiter aus und legte den Grundstein für das Feld der kombinatorischen Logik, einer Logik ohne Variablen, stattdessen basierend auf Kombinatoren.

– Fortsetzung –

Im 2. Weltkrieg arbeitete Curry an der Programmierung des ENIAC, des ersten elektronischen Universalrechners. Von 1942-44 war er in der angewandten Physik bei Frankford Arsenal beschäftigt, ab 1944 dann beim Ballistic Research Laboratory (Aberdeen Proving Ground), das den Bau des ENIAC in Auftrag gegeben hatte. Beim Ballistic Reseach Laboratory war er in dem Komittee, das für die Anwendung und das Testen der Maschine zuständig war. Die Mitglieder der Gruppe waren ein Astronom, ein Zahlentheoretiker und der Logiker Curry.

Zusammen mit Willa Wyatt, einer der Programmiererinnen des ENIAC, schrieb Curry den Artikel *A study of inverse interpolation of the ENIAC* über die Interpolation bei der Berechnung von ballistischen Tabellen mittels des ENIAC. Im Rahmen der Studie bemerkte er bereits, dass die Organisation der Programmierung eine wichtige Rolle bei der Problemlösung spielte. Er beschrieb, dass das Hauptprogramm eine Zusammensetzung von Unterprogrammen war, die je nach Problemvariation ausgetauscht werden konnten.

Die Thesen von John von Neumann und H.H. Goldstine, in Kombination mit seiner eigenen Vorarbeit münden in Currys späteren Artikel *On the composition of programs for automatic computing* und hatten noch einige weitere Artikel zur Folge.

Durch seine theoretische Erfahrung auf dem Gebiet der Logik, und seiner praktische Erfahrung bei der Programmierung des ENIAC, trug Curry zur Entwicklung einer abstrakten Schreibweise zur Programmierung von Rechenmaschinen bei. Diese ist die Grundlage der Programmiersprachen, wie wir sie heute kennen.

Im Jahr 1936 gründete Curry die *Association for symbolic logic*. Nachdem er nach einigen Jahren deren Präsidentschaft aufgab, konnte er sich dem Schreiben einer umfassenden Arbeit über kombinatorische Logik widmen, in der er Beziehungen zum Lambda-Kalkül Kapitel 7 seines Kollegen Church zog. Er lehrte und forschte dabei weiter an der Pennsylvania State University. Nachdem er in Rente ging, nahm er im Alter von 66 Jahren eine Professorenstelle in Amsterdam als Wissenschaftsphilosoph und -historiker für Logik an, ging aber nach vier Jahren nach Pennsylvania zurück.

Das Buch *A short biography of Haskell B. Curry, in To H. B. Curry : essays on combinatory logic, lambda calculus and formalism (London-New York, 1980), vii-xi.* beschreibt die Currys als freundliches Paar, das gerne kocht.

> »Jeder, der die Currys kannte, weiß dass sie immer freundlich und hilfsbereit waren. Haskell war für seine Kollegen und Studenten immer mehr als nur eine Quelle wichtiger Ideen. Er war jederzeit bereit allen, die mit ihm reden wollten, zuzuhören, ihre Ideen zu diskutieren und hatte ein Talent Leute zu ermutigen neue Wege zu gehen und zu denken. Damit hat er mit Sicherheit einen wichtigen Beitrag zum anhaltenden Enthusiasmus geleistet. Insbesondere bei vielen, die heute in der kombinatorischen Logik arbeiten. Berühmt waren die Currys auch für ihre Gastfreundschaft, wo auch immer sie gerade gewohnt haben. Bei ihnen fanden immer viele Parties und andere lockere Zusammentreffen statt, und wir hegen die Vermutung, dass Virginias Kochkünste eine nicht zu unterschätzende Rolle beim Wachsen des Interesses an kombinatorischen Logik spielten.«

– Fortsetzung –

Die beiden funktionalen Sprachen Haskell und Curry und auch die Curryfizierung sind nach Haskell B. Curry benannt.

Zudem ist Curry der Schöpfer von Currys Paradox in der Logik, einem Paradox aus der Klasse der selbstbezüglichen Paradoxien. Das bekannteste selbstbezügliche Paradox ist vermutlich das Paradox des Epimenides, sinngemäß lautet dieses so: »Der Kreter Epimenides sagt, alle Kreter lügen«.

Currys Paradox ist sozusagen die Negation dieses Lügner-Paradoxes. Es gibt sowohl eine Formulierung in der Mengenlehre als auch in der Logik. Am einfachsten zu verstehen ist es allerdings in seiner Formulierung als *Gottesbeweis*: »Wenn dieser Satz wahr ist, dann existiert Gott.« Bertrand Russells Lösung der Russelschen Antinomie (»Menge aller Mengen, die sich nicht selbst als Element enthalten«) durch ein hierarchisches Typsystem (Typentheorie), um Selbstbezüge zu vermeiden, hilft uns auch bei Curry's Paradoxon aus der Patsche, und Gottes Existenz ist doch nicht bewiesen. ;-)

Außerdem ist Curry zusammen mit William Alvin Howard auch der Namensgeber für die Curry-Howard-Korrespondenz. Diese beschreibt die direkte Verbindung zwischen Computerprogrammen und logischen Beweisen (»proofs as programs«). Der Zusammenhang zwischen beidem fiel bei der Untersuchung der Typen der Kombinatoren in der kombinatorischen Logik auf. Typen können als logische Axiome gesehen werden. Heutzutage gibt es sogar Programmiersprachen, in denen sich Beweise als Programme schreiben lassen, etwa die Programmiersprache *Coq* (Kapitel 10 im Abschnitt »Abhängige Typen, Agda, Idris und Coq« auf Seite 177).

Quelle: *http://www-history.mcs.st-andrews.ac.uk/Biographies/Curry.html*

Quelle: *Haskell before Haskell. Curry's contribution to programming (1946–1950)*

Man sagt dementsprechend auch, eine Funktion wird gecurried oder curryfiziert. Vermutlich auch, weil niemand »geschönfinkelt« sagen möchte, obwohl auch Moses Schönfinkel an dem gleichen Thema geforscht hat.

Curryfizierung ist also die Umwandlung einer einzelnen Funktion mit `n` Argumenten in `n` Funktionen mit je einem einzelnen Argument. Wie sieht das für ein konkretes Beispiel aus? Angenommen, wir haben eine Funktion `f`, die drei Argumente addiert:

```
function f (x, y, z) { return x + y + z; }
```

Curryfiziert sieht diese Funktion so aus:

```
function fcurr (x) {
  return function (y) {
    return function (z) {
      return x + y + z;
    }
  }
}
```

Um die gesamte Funktionsanwendung für `f(x, y, z)` zu bekommen, schreiben wir:

```
fcurr(x)(y)(z);
```

Wenn man jedoch nur `fcurr(10)(20)` aufruft, dann bekommt man eine teilweise angewandte Funktion. Der Rückgabewert ist die Closure `function (z) { return 10 + 20 + z; }`, also eine anonyme Funktion mit gebundenen Variablen.

Jetzt ist es Zeit für noch ein anschaulicheres Beispiel. Wir recyclen unsere oben definierte plus-Funktion, curryfizieren sie und füllen das erste Argument ein.

```
// Zur Erinnerung:
// function plus (a, b) { return a + b; }

var plusTen = plus.curry(10); // definiere eine Funktion, die 10 + argument erzeugt
plusTen(20);
> 30
```

Die Funktion `curry` ist in JavaScript nicht implementiert, kann aber selbst geschrieben werden. Hier haben wir den Funktions-Prototypen erweitert, indem wir die Funktion `curry` unter dem Namen `Function.prototype.curry` definieren. Dadurch wird die Funktion `curry` zu *allen Funktionen* hinzugefügt.

```
Function.prototype.curry = function () {
  // wir merken uns f
  var f = this;
  if (arguments.length < 1)
    return f;
  var a = toArray(arguments);
  return function () {
    var b = toArray(arguments);
    return f.apply(this, a.concat(b));
  }
}
```

Die Funktion `toArray` ist eine Hilfsfunktion, welche die Argumente der Funktion, `arguments`, in ein echtes Array umwandelt. Die Variable `arguments` ist aufgrund des Designs der Sprache JavaScript kein vollständiges Array, sondern nur ein Array-Lookalike und muss umgewandelt werden, damit wir die `concat`-Funktion benutzen können, die nur auf echten Arrays definiert ist.

```
function toArray (xs) {
  return Array.prototype.slice.call(xs);
}
```

Die Funktion `apply` ist sehr ähnlich zu `call` (siehe Tipp »Die Funktion call ...« auf Seite 49), der Unterschied liegt in den Argumenten: `call` erwartet nach `this` die Argumente einzeln, wohingegen `apply` ein Array übergeben bekommt, und jedes `Array`-Element als einzelnes Argument benutzt.

Die Variable `arguments` ist in jeder Funktion verfügbar, und enthält die Argumente der Funktion. Diese Variable ist kein echtes Array, sondern nur ein Array-Lookalike (h*ttps://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/arguments*). Wenn wir alle `Array`-Funktionen benutzen wollen, müssen wir `arguments` in ein echtes Array konvertieren.

Die eigentliche Funktionalität der Funktion `curry` steckt in dem Ausdruck `f.apply(this, a.concat(b))`. Dieser Ausdruck enthält einen Taschenspielertrick, der die umgebende Closure und die Argumente der zurückgegebenen Funktion benutzt. In dem betrachteten Ausdruck werden `a` und `b` konkateniert und `f` wird auf das Ergebnis der Konkatenation angewendet. In unserem Beispiel ist `f` die Funktion `plus`, und `a` ein Array von Argumenten, mit denen wir `plus` aufgerufen haben, also `1`. Was aber ist genau `b`? Durch die Verwendung der Variable `arguments` innerhalb der Funktion, die wir zurückgeben, besteht `b` aus den Argumenten, die wir der fertig curryfizierten Funktion geben wollen. Wir erhalten also eine Funktion zurück, die `f` auf `f`s Originalargumente `a`, und die Argumente `b`, die die Funktion selbst noch bekommt, anwendet.

Quelle: *http://javascriptweblog.wordpress.com/2010/04/05/curry-cooking-up-tastier-functions/*

Mithilfe der Funktion `curry` können wir nun eine Funktion `plusOne` definieren, die nur noch ein Argument erwartet, und auf dieses die Zahl `1` addiert.

```
// Zur Erinnerung:
// function plus (a, b) { return a + b; }

var plusOne = plus.curry(1);
plusOne(10);
> 11
```

Dies ist schon mal praktisch, um die Funktion `plusOne` wiederzuverwenden. Besonders nützlich wird die Curryfizierung aber im Zusammenspiel mit Funktionen höherer Ordnung (Kapitel 2 im Abschnitt »Abstraktion von einer Funktion: Funktionen höherer Ordnung« auf Seite 25), wenn diese eine Funktion mit nur einem Argument erwarten, wie die Funktion `map` im Kapitel 4 (im Abschnitt »Parallele Berechnung auf Arrays mit Map« auf Seite 48).

Genau wie bei der binären Funktion `plus` können wir auch eine mehrstellige Funktion mit mehreren Argumenten füllen. Dies kann entweder durch wiederholte Anwendung von `curry` geschehen:

```
function sum3 (a, b, c) { return a + b + c; }
var sum3_1_1 = sum3.curry(1).curry(1);
sum3_1_1(2);
> 4
```

...oder durch das Einfüllen mehrerer Argumente.

```
var sum3_1_1 = sum3.curry(1, 1);
sum3_1_1(2);
> 4
```

Teilweise Funktionsanwendung

Wir haben sozusagen im Vorbeigehen zusammen mit der Curryfizierung auch noch ein weiteres Konzept kennengelernt: die teilweise oder auch partielle Funktionsanwendung (partial function application). Hierbei wird eine Funktion, die mehrere Argumente erwartet, nur mit einem Teil der Argumente gefüllt. Dadurch entsteht eine neue Funktion, die nur noch die restlichen, fehlenden Argumente erwartet. Nach diesem Prinzip sind wir bei der Definition der Funktionen `plusTen` und `plusOne` vorgegangen: Wir haben in die Funktion `plus`, die eigentlich zwei Argumente erwartet, bereits ein Argument »hineingeschoben«.

Die partielle Funktionsanwendung wird manchmal mit der Curryfizierung in einen Topf geworfen. Es handelt sich jedoch um zwei unterschiedliche Konzepte, die aber oft zusammen auftauchen, da die Curryfizierung die partielle Funktionsanwendung erst ermöglicht. Dies ist auch in unserer obigen Definition der Fall, wo beide Konzepte gleichzeitig angewendet werden. Eigentlich müsste unsere Funktion daher `curryAndPartial` heißen.

Besonders in der JavaScript-Community wird schlicht von Curryfizierung gesprochen, wenn eigentlich »Curryfizierung und partielle Funktionsanwendung« gemeint ist. Die beiden Konzepte lassen sich in JavaScript schwer trennen, da eine Funktion nicht das Gleiche ist wie der Wert, den sie zurückgibt, so wie es in rein funktionalen Sprachen der Fall ist (vgl. Rückgängig machen der Curryfizierung auf Seite 42).

Hineinschieben der Argumente für die teilweise Funktionsanwendung erfolgt in unserem Beispiel – und generell meistens von rechts. Mit Oliver Steeles Bibliothek *functional.js* kann man auch Argumente »zwischendurch« weglassen, indem man diese durch einen Unterstrich ersetzt, und bekommt eine partiell angewandte Funktion zurück:

```
var makeHaskellsName = sum3.partial("Haskell ", _, " Curry");
makeHaskellsName("B.");
> Haskell B. Curry
```

Quelle: *http://junctional.org/articles/partial-application/*

Automatische Curryfizierung

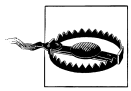
Einige Sprachen wie Haskell haben automatisches Currying eingebaut. Wird hier eine Funktion teilweise mit Argumenten gefüllt, dann wird sie automatisch curryfiziert, und wir erhalten eine Funktion, welche die restlichen Argumente erwartet. Füllen wir eine Funktion komplett mit Argumenten, dann wird sie ausgeführt.

In JavaScript müssen wir `curry` jedoch explizit aufrufen, wenn wir eine Funktion curryfizieren (eigentlich: curryfizieren und teilweise anwenden) wollen. Die automatische Curryfizierung für JavaScript wird von der externen Bibliothek *wu.js* bereitgestellt. Die Funktion `wu.autoCurry` entscheidet beim jeweiligen Aufruf anhand der übergebenen und der erwarteten Argumente der Funktion, ob die Funktion curryfiziert werden muss und tut es nur dann, wenn nicht alle Argumente angegeben wurden. (*http://javascript-weblog.wordpress.com/2010/06/14/dipping-into-wu-js-autocurry/*).

```
function sum3 (a, b, c) { return a + b + c; }
sum3curried = wu.autoCurry(sum3);

sum3curried(1);
> function () { ...  } // wartet noch auf 2 Argumente; currying wurde aufgerufen
sum3curried(1, 2);
> function () { ...  } // wartet noch auf 1 Argument; currying wurde aufgerufen
sum3curried(1, 2, 3);
> 6 // currying nicht nötig, sum3curried aufgerufen
sum3curried(1, 2, 3, 4);
> 6 // überflüssiges Argument verworfen
```

Die Anzahl der Argumente, die eine Funktion bei ihrer Ausführung erwartet, ist in JavaScript in der `length`-Eigenschaft der Funktion definiert. Diese wird von `wu.autoCurry` benutzt, um sicherzustellen, dass genügend Argumente vorhanden sind, um die Funktion aufzurufen. Bei einer variablen Anzahl von Argumenten kann `wu.autoCurry` die minimale Anzahl der Argumente mitgegeben werden, die für eine Ausführung der Funktion notwendig ist.



Spezialität des Hauses: Der logische Oder-Operator `||` gibt das linke Argument zurück, sofern es nicht zu `false` evaluiert. Es ist `false`, wenn es nicht übergeben und dadurch `undefined` ist Kapitel 8 im Abschnitt »Wahrheitswerte/Booleans« auf Seite 132. Auf diese Weise können Default-Argumente realisiert werden: Wird das Argument nicht übergeben, so wird der Default-Wert, der rechts vom Oder-Operator steht, genommen.

Definieren wir uns eine Funktion `sum3or4`, die drei oder vier Argumente aufsummiert. Wird das vierte Argument weggelassen, bekommt es den Default-Wert `0`.

```
function sum3or4 (a, b, c, d) {
  d = d || 0;
  return a + b + c + d;
}
var sum3or4curried = wu.autoCurry(sum3or4); //wie sum3curried
sum3or4curried(1, 2, 3, 4);
> 10
sum3or4curried(1, 2, 3);
> function () { .. } // wartet immer noch auf das optionale Argument!

sum3or4curried = wu.autoCurry(sum3or4, 3); // 3 Argumente reichen zur
Funktionsanwendung
sum3or4curried(1, 2, 3); // wird erfolgreich ohne das optionale vierte Argument
> 6
ausgeführt
```

Zu beachten ist natürlich, dass wir eine Funktion zurückbekommen, die weitere Argumente erwartet, wenn wir der curryfizierten Funktion zu wenig Argumente übergeben. Aber dies wäre in einer Programmiersprache mit automatischem Currying genauso der Fall und ist ja auch erwünscht.

Es bleibt noch die Frage, wie wir die Curryfizierung rückgängig machen können. Die Funktion `uncurry` transformiert `n` Funktionen mit einem Argument in eine Funktion mit `n` Argumenten. Dies ist in Sprachen wie Haskell besonders nützlich, da zusammengesetzte Datentypen dort oft als Tupel dargestellt werden. Haskell ist im Gegensatz zu JavaScript eine Sprache mit echter partieller Funktionsanwendung. In JavaScript können wir die partielle Funktionsanwendung nur durch Closures, die noch auf Argumente warten, nachbauen. Die Funktion `uncurry` kann in JavaScript nicht als echte inverse Funktion zu `curry` implementiert werden, denn anders als in Haskell ist in JavaScript ist die vollkommen mit Argumenten gefüllte Funktion ungleich dem Wert, den sie zurückgibt.

Quelle: *http://osteele.com/sources/javascript/functional/*

Implizite Argumente, Komposition und Point-Free-Style

Manchmal ist es auch sinnvoll, über eine curryfizierte Funktion vorübergehend ohne ihre Argumente nachzudenken. Mithilfe dieser Denkweise können wir Funktionen durch Funktionskompositionen oder Kombinatoren zu neuen Funktionen verbinden, ohne die Argumente zu deklarieren. Wir brauchen uns dann auch keine Namen für die Argumente auszudenken. Dies wird Point-Free-Style-Programmierung oder auch Tacit Programming genannt und führt häufig zu besonders eleganten Lösungen.

Um dieses Konzept anzuwenden, brauchen wir zunächst die Funktionskomposition. Diese konstruiert eine neue Funktion `z` aus den Eingabefunktionen `f` und `g`, die der Anwendung von `z(x) = (f ∘ g)(x) = (f(g(x)))` entspricht. Diese können wir wieder direkt auf dem Funktions-Prototyp definieren:

```
Function.prototype.c = function (g) {
  // wir merken uns f
  var f = this;
  // wir konstruieren Funktion z
  return function () {
    var args = Array.prototype.slice.call(arguments);
    // beim Aufruf stecken wir g's Rückgabe in einen Aufruf von f
    return f.call(this, g.apply(this, args));
  }
}

var greet = function (s) { return 'Hallo, ' + s; };
var exclaim = function (s) { return s + '!'; };
var excited_greeting = greet.c(exclaim);
excited_greeting('Haskell B. Curry');
> Hallo, Haskell B. Curry!
```

Wir können also, wie im Beispiel dargestellt, mehrere Funktionen durch Funktionskomposition miteinander verbinden. Damit werden die Argumente sozusagen unsichtbar von `exclaim` an `greet` durchgereicht. Der Programmcode bleibt dadurch sehr knapp und übersichtlich.

Was der Point-Free-Style mit Curryfizierung zu tun hat, wird am obigen Beispiel noch nicht deutlich. Alle verwendeten Funktionen sind einstellig, und es macht daher keinen Unterschied, ob wir sie curryfizieren oder nicht.

Betrachten wir jedoch unser Beispiel zur Curryfizierung von `plus`,

```
var plusOne = plus.curry(1);
```

so ist uns der Point-Free-Style sogar schon begegnet! Denn dieser Ausdruck ist eine Abkürzung für

```
function plusOne (x) { return plus.curry(1)(x); }
```

Durch die Zuweisung der neuen, curryfizierten Funktion zu der Variablen `plusOne` haben wir ein »unsichtbares« Argument `x` mitgeführt, und zwar das Argument, auf welches die curryfizierte Funktion noch wartet.

Argumente sind bei Variablenzuweisung von Funktionen implizit: `var plusOne = plus.curry(1)` entspricht `function plusOne (x) { return plus.curry(1)(x); }`

Natürlich können wir auch nicht-curryfizierten Funktionen Variablen zuweisen und so »unsichtbare« Argumente mitführen.

```
add = plus
add(1,1)
> 2
```

Auf der einen Seite gibt es Funktionen als First-Class-Citizens, man kann also Funktionen an Variablen zuweisen, wodurch implizite Argumente erzeugt werden. Auf der anderen Seite haben wir Currying und partielle Funktionsanwendung kennengelernt, womit wir Argumente einfüllen können. Daraus ergibt sich mithilfe der Kompositionsfunktion und anderer Funktions-Kombinatoren, die wir in Kapitel 5 und Kapitel 7 noch kennenlernen werden, der Point-Free-Style.

Quelle: *http://www.slideshare.net/drboolean/pointfree-functional-programming-in-javascript*

Quelle: *http://dailyjs.com/2012/09/14/functional-programming/*

Nachdem wir in unseren ersten Topf mit Curry hineingeschnuppert haben, erkennen wir, dass uns funktionale Techniken wie die Curryfizierung, partielle Funktionsanwendung und Funktionskomposition helfen können, mächtige Funktionen zu erschaffen, um Eingabedaten zu verarbeiten. Diese Funktionen sind wiederum aus kleineren Funktionen zusammengesetzt, die wir immer wiederverwenden.

Auf diese Weise erreichen wir einen modularen Aufbau, einen hohen Grad an Abstraktion, und wesentlich kürzeren Programmcode. Dadurch wird die Funktionalität klarer erkennbar und der Code ist leichter zu warten. Außerdem werden im Programmcode die Daten von den Funktionen entkoppelt.

Wir müssen somit nicht erst die Daten bekommen und in jedem Schritt mit den Daten und Variablen operieren, sondern schicken unsere Daten erst ganz zum Schluss durch die Funktionen, die wir uns zuvor aus kleineren, einfacheren Funktionen aufgebaut haben.

Es ist kein Zufall, dass wir mit einem grundlegendem vegetarischen Gericht wie Aloo Gobi begonnen haben, uns die Grundlagen der funktionalen Programmierung anzusehen. Die indische Küche ist typischerweise eine vegetarische Küche. Fast alle Menschen in Indien sind Vegetarier, zum Teil aus religiösen Gründen, größtenteils jedoch aus Armut. Dabei ist Gemüse in Indien mehr als nur eine Beilage. Manches Gemüse, wie die Aubergine, ist fast schon auf einer höheren Ordnung angesiedelt. Wie der aromatische Reichtum von Gemüse mit Hilfe von wenigen Standardfunktionen wie `map`, `reduce` und `filter` entfaltet werden kann, schauen wir uns im nächsten Kapitel an.

```
(λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s z))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

KAPITEL 4

Gemüse, Map, Reduce und Filter

Auberginen und die Funktionen höherer Ordnung

Das Wort Aubergine kommt aus dem Französischen. Anhand seiner Etymologie lässt sich der historische Weg dieses Gemüses nach Deutschland ziemlich direkt nachvollziehen. Aubergine stammt von dem Katalanischen »albergínia«, das vom Arabischen البادِنْجان (al-baðinjān) abstammt. Dieses ist dem Persischen بادنگان (bâdengân) entlehnt und auf das Sanskrit-Wort वातिगगम (ivātiga-gama) zurückführbar. Auberginen haben für Indien eine geradezu identitätsstiftende Bedeutung. In Indien liegen nicht nur die weltweit größten Anbaugebiete für Auberginen, sondern auch die unterschiedlichen Formen und die Vielfältigkeit der Geschmäcker sind so bedeutend, dass es im ersten Jahrzehnt des 21. Jahrhunderts Auberginen-Proteste gab, als Bauern eine Einschränkung der Vielfalt (Biodiversität) fürchteten.

Dies lag daran, dass der Konzern Monsanto 2005 Saatgut für die genetisch modifizierte Aubergine *Bt brinjal* verkaufen wollte. Vorangegangen war ein aufgrund erheblichen Drucks entstandener Vertrag zwischen der indischen Regierung und dem Konzern, der den Verkauf von genetisch modifizierten Pflanzen und Saatgut zunächst nur für diese Aubergine erlaubte. Damit traf Monsanto einen empfindlichen Nerv der indischen Bauernschaft. Die Aubergine, als Zutat aus dem Gemüsecurry nicht wegzudenken, ist eben mehr als nur irgendein Gemüse. Neben den Sorgen um die Verträglichkeit für den Menschen und dem Themenkomplex der Patente brachte vor allem die Standardisierung auf eine »Norm-Aubergine« die indischen Aktivisten in Rage, die bei ihren Protesten eindrucksvoll die ganze Vielfalt der Auberginen ausstellten.

Abbildung 4-1: Baingan Bharta (Foto von Matthew Mendoza)

Es ist möglicherweise gar nicht so weit hergeholt, den Konflikt zwischen Monsanto und den indischen Bauern um die *Bt brinjal* als einen Streit zwischen funktionalem und objektorientierten Denken aufzufassen. Denn in der herkömmlichen objektorientierten Softwareentwicklung wird genetisch modifiziert, vererbt und klassifiziert. Die indischen Auberginen sind auf den ersten Blick ein chaotisches Sammelsurium an Launen der Evolution, jede Auberginensorte hat dabei eine für sich alleine stehende Funktion. Trotzdem können alle über die gleichen Kochfunktionen komponiert werden. Die Funktionen, die auf die Auberginen oder ein anderes Gemüse angewendet werden, sind universell einsetzbar. Für ein Curry filtern, mappen und reducen wir Gemüse.

Weitere Funktionen höherer Ordnung: Map, Reduce, Filter und Kolleginnen

In diesem Kapitel lernen wir die Standard-Funktionen höherer Ordnung kennen: reduce, map und filter. Sie alle stehen in JavaScript seit der Version 1.6 (bzw. 1.8 für `reduce`) für Array-Datenstrukturen zur Verfügung (*https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array*) und lassen sich auch leicht selbst implementieren, wie wir sehen werden.

Diese Funktionen bilden einen nützlichen Werkzeugkasten, mit dem wir auf sequenziellen Datenstrukturen wie Arrays nach verschiedenen Mustern Berechnungen ausführen können. Die wichtigsten Muster lernen wir hier kennen:

Rezept: Baingan Bharta (Auberginen-Curry)

Für 2-3 Personen:

- 1 dicke, fette Aubergine
- 2 mittelgroße Tomaten
- 120g grüne Paprika, kleingeschnitten
- 1 grüne Chili
- ein kleines Stück Ingwer
- 3 EL Öl
- 1 TL Kumin
- 1 TL Korianderpulver
- 1 TL Kurkuma
- 1 TL rote Chilis
- Salz
- 1 TL Garam Masala
- 2 TL frischen, gehackten Koriander zum Garnieren

In einer mikrowellengeeigneten Form die Aubergine etwa 8-10 Minuten in der Mikrowelle kochen lassen, bis sie schön weich und durch ist. Die Aubergine abkühlen lassen, schälen und in kleine Stücke schneiden. Tomaten, Ingwer und grüne Chili mischen und pürieren. In einer Pfanne die Paprika bei mittlerer Hitze kurz anbraten und dann aus der Pfanne nehmen. Das Öl in der Pfanne gut erhitzen und einen Kuminsamen hineingeben. Wenn dieser zerplatzt, ist das Öl heiß genug. Nun die restlichen Kuminsamen hinzugeben. Wenn alle Samen geplatzt sind, das Tomatenpüree, Korianderpulver, Kurkuma, rote Chilis und etwas Salz in die Pfanne geben und gut durch kochen. Die Auberginenstückchen hinzufügen und unter Rühren zu einem Brei zerkochen. Diesen weitere 8-10 Minuten köcheln lassen. Paprika, frischen Koriander und Garam Masala zu der Aubergine geben und vorsichtig vermischen. Das fertige Bharta schmeckt mit Rotis oder Naan (indischem Brot, nicht zu verwechseln mit `NaN`-Brot). Guten Appetit!

- »paralleles« Arbeiten auf jeder Zelle, ohne die Struktur zu verändern (`map`)
- Extraktion einer Teilsequenz von Zellen und darin enthaltenen Lösungen anhand bestimmter Eigenschaften (`filter`)
- Strukturtransformation zu einer Gesamtlösung mittels einer Kombinationsfunktion (`reduce`)

Das map-reduce-Paradigma ist besonders in letzter Zeit sehr populär geworden, da es zur abstrakten Unterteilung in parallele (`map`) und kombinierende (`reduce`) Arbeitsabschnitte auch beim verteilten Rechnen auf einem Cluster oder in der »Cloud« benutzt werden kann.

Parallele Berechnung auf Arrays mit Map

Einige Funktionen führen auf jedem Element eines Arrays die gleiche Operation durch und geben ein neues Array mit den Ergebniswerten der Operation zurück.

Die folgende Funktion quadriert alle Werte eines Arrays:

```
var squares = function (xs) {
  var len = xs.length;
  var res = new Array(len);
  for (var i = 0; i < len; i++) {
    res[i] = xs[i] * xs[i];
  }
  return res;
};

var numbers = [1,2,3,4,5,6];
squares(numbers);
> [1, 4, 9, 16, 25, 36]
```

Nach dem gleichen Muster können wir auch aus unserem Array mit den Gewürzen aus Kapitel 2 im Abschnitt »Abstraktion von einem Codeblock: Funktionen« auf Seite 24 alle Gewürznamen extrahieren:

```
var getAllNames = function (xs) {
  var len = xs.length;
  var res = new Array(len);
  for (var i = 0; i < len; i++) {
    res[i] = xs[i].name;
  }
  return res;
};

var spices = [... s. Kapitel 2 ...]
getAllNames(spices);
> ["Nelken", "Zimt", "Schwarzer Pfeffer"]
```

Das Ausführen einer Funktion auf jedem Array-Element und die Rückgabe eines neuen Arrays mit den Resultaten verallgemeinern wir in der Funktion `map`.

Map iterativ

Die iterative Implementierung von map bekommt zwei Argumente, zum einen die Funktion `fun`, die auf jedem Array-Element ausgeführt wird, und zum anderen das Array `xs`. Zuerst überprüft `map`, ob `fun` tatsächlich eine Funktion ist, dann erzeugt es ein neues Array `res`, für das Resultat. Es folgt eine Schleife, in der die Funktion `fun` auf jedem Array-Element ausgeführt wird, und der Rückgabewert in dem Ergebnis-Array `res` gespeichert wird. Zum Schluss wird das Ergebnis `res` zurückgegeben.

typeof werden wir in Kapitel 8 im Abschnitt »Primitive Datentypen versus Objekte« auf Seite 135 erläutern. Hiermit wird sichergestellt, dass es sich bei fun um eine Funktion handelt – ansonsten gibt es eine Fehlermeldung.

Die Funktion call ist in JavaScript auf dem Funktionsprototyp definiert. Sie nimmt die Funktion und wendet sie auf die gegebenen Argumente an, ohne das Objekt, über das man zu der Funktion gekommen ist, implizit als Parameter zu haben. Auf die ähnliche Funktion apply sind wir schon in Kapitel 3 im Tipp »Die Funktion apply ...« auf Seite 38 eingegangen

```
// Abgewandelt aus Array.prototype.map:
var mymap = function(fun, xs) {
  if (typeof fun != "function")
    throw new TypeError();
  var len = xs.length;
  var res = new Array(len);
  for (var i = 0; i < len; i++) {
    res[i] = fun.call(xs, xs[i], i, xs);
  }
  return res;
}
```

Verwendung von Map

Unser Beispiel von oben, die Funktion squares, können wir nun mit Hilfe unserer Funktion mymap schreiben.

```
mymap(function (a) { return a * a; }, numbers);
> [1, 4, 9, 16, 25, 36]
```

Da es map in JavaScript bereits als vordefinierte Funktion für Arrays gibt, können wir das gleiche Ergebnis erhalten, indem wir

```
numbers.map(function (a) { return a * a });
```

schreiben. Noch abstrakter und im Aufruf kürzer ist es allerdings, squares als Funktion mit dem eingebauten oder selbstgeschriebenen map zu definieren:

```
function squares (xs) { return mymap(function (a) { return a * a; }, xs); }
squares(numbers);
> [1, 4, 9, 16, 25, 36]
```

Map und Curry

Stellen wir uns vor, dass wir ein Array von Zahlen haben, und mit map jede davon um eins erhöhen wollen. Dies würden wir aus dem Bauch heraus mit map in etwa so schreiben:

```
var arr = [1, 2, 3];
var incarr = mymap(+(1), arr); // gibt einen Fehler!
```

Hier gibt es, wie wir aus dem vorherigen Kapitel wissen, einige Dinge zu beachten: Die Funktion map erwartet eine »normale JavaScript-Funktion« in Präfix-Schreibweise, + wird

jedoch als Operator in der Infix-Schreibweise benutzt. Im vorangehenden Kapitel haben wir uns glücklicherweise die Funktion `plus` definiert, mit der wir den Operator `+` in die Präfix-Schreibweise umgewandelt haben. Die übergebene Funktion soll jeweils auf einem Element des Arrays operieren. Sie sollte also nur ein Argument benötigen, um angewendet werden zu können. Wir möchten aus `plus`, das zwei Argumente erwartet, eine Funktion machen, die nur noch ein Argument erwartet, indem wir das Argument 1 schon einfüllen. Aber wie soll das `plus(1)` funktionieren, wenn `plus` zwei Argumente erwartet? Wir brauchen unsere Funktion `curry` (Kapitel 3, Seite 38), die `plus` zum Teil mit Argumenten füllt. Dank dieser Überlegungen können wir `plusOne` konstruieren, das wir aus Kapitel 3, Seite 39 schon kennen.

```
var plusOne = plus.curry(1); // aus Kapitel 3
incarr = mymap(plusOne, arr);
> [2, 3, 4]
```

In Oliver Steeles Bibliothek functional.js geht sogar

```
incarr = map('+1');
```

Auch `squares` können wir mittels `curry` noch abstrakter beschreiben:

```
function square (a) { return a * a; }
squares = mymap.curry(square);
```

Durch das Currying der Funktion `map` kann man in diesem Fall das letzte Argument (Kapitel 3 im Abschnitt »Implizite Argumente, Komposition und Point-Free-Style« auf Seite 42) also das Array `xs`, weglassen und erhält eine extrem kurze und elegante Schreibweise. Die Funktion `squares` »mappt« die Funktion `square` auf ein Array von Eingaben.

Exkurs: Vorsicht bei parseInt in Verbindung mit Map

Auf Arrays von Zahlen arbeitet die Funktion `squares` ohne Probleme. Wenn Zahlen als Strings vorliegen, wie `["1", "2", "3", "4"]`, müssen wir sie zuerst als Zahlen einlesen. Da wir gerade `map` kennengelernt haben, wollen wir dazu mit `map` die Funktion `parseInt` aus Kapitel 2, auf Seite 26 auf jedes Array-Element anwenden. Aus dem Bauch heraus gehen wir so vor: `["1", "2", "3", "4"].map(parseInt);` Das Resultat der Anwendung ist jedoch nicht, wie wir erwarten, `[1, 2, 3, 4]`, sondern `[1, NaN, NaN, NaN]`.

Wir haben in Kapitel 2, auf Seite 26 gesehen, dass es Probleme geben kann, wenn `parseInt` ohne Basis aufgerufen wird, und genau das passiert hier. In diesem Beispiel spielt zudem die Art und Weise, mit der in JavaScript Argumente an Funktionen übergeben werden, eine Rolle. Denn die Anzahl der Argumente wird bei einem Funktionsaufruf nicht so streng gehandhabt wie in den meisten Programmiersprachen. Werden beim Funktionsaufruf weniger Argumente als die deklarierten übergeben, dann sind die restlichen `undefined`. Werden mehr Argumente übergeben, werden diese nicht an Namen gebunden. Die Variable `arguments` enthält sowohl die übergebenen Argumente, als auch ihre Anzahl in der Eigenschaft `length`. Zudem hat auch jede Funktion eine Eigenschaft `length`, welche die Anzahl der dekla-

rierten Argumente enthält. Mit diesen Informationen kann man beim Programmieren selbst überprüfen, ob genau so viele Argumente deklariert wie übergeben wurden (Kapitel 3 im Abschnitt »Automatische Curryfizierung« auf Seite 40). Die Funktion `parseInt` bekommt zwei Argumente: das Objekt, das als Zahl geparsed werden soll, und die Basis, die das Zahlensystem angibt, in dem die eingelesene Zahl interpretiert werden soll.

Die Funktion `map` übergibt der Funktion, die auf jedem Element des Arrays ausgeführt werden soll, nicht nur das jeweilige Array-Element, sondern auch den Index des Elements im Array und das gesamte Array. Die Funktion `parseInt` wird dadurch mit den Argumenten `("1", 0, ["1", "2", "3", "4"])`, `("2", 1, ["1", "2", "3", "4"])`, `("3", 2, ["1", "2", "3", "4"])` und `("4", 3, ["1", "2", "3", "4"])` aufgerufen. Es wird also jeweils fälschlicherweise der Array-Index als Basis an `parseInt` übergeben, was bei der Basis `0` zum Ergebnis `1` und sonst zum Ergebnis `NaN` führt.

Wir haben zwei Möglichkeiten, die Anwendung von `parseInt` durch Übergabe der Basis `10` zu reparieren. Entweder wir schreiben eine Wrapper-Funktion, die `parseInt` mit dem zusätzlichen Argument in eine neue Funktion verpackt, wie `parseIntBase10` in Kapitel 2. Oder wir benutzen eine partielle Funktionsanwendung mit der Funktion `curry` aus Kapitel 3. Allerdings schiebt man die Argumente von der rechten Seite in die curryfizierte Funktion hinein. Die partielle Funktionsanwendung füllt also zuerst das erste Argument, die Basis `10` wird als zweites Argument übergeben müssen. Hier hilft uns die Funktion höherer Ordnung `flip`, die die ersten beiden Argumente vertauscht: aus `koche("aubergine", 10)` wird durch Anwendung von `flip` `koche(10, "aubergine")`.

```
Function.prototype.flip = function () {
    var fn = this;
    return function () {
        var args = toArray(arguments);
        args = args.slice(1,2).concat(args.slice(0,1)).concat(args.slice(2));
        return fn.apply(this, args);
    };
}
```

Jetzt können wir das Array mit Hilfe von `flip` und `curry` konvertieren:

```
["1", "2", "3", "4"].map(parseInt.flip().curry(10));
```

An diesem Beispiel sieht man sehr gut die funktionale Problemlösungsstrategie: Aus kleineren, allgemeineren Funktionen wird eine Gesamtfunktion aufgebaut, die eine spezielle Aufgabe löst. Diese Gesamtfunktion wird dann auf die Eingabedaten angewendet. Daten und Funktionen sind also voneinander getrennt.

Wir haben gelernt, dass `map` die Berechnung auf jedem Element eines Arrays ausführt. Nun sind beliebige Funktionen denkbar, die nach diesem Muster auf Arrays angewendet werden, und sozusagen parallel in jeder Zelle die gleiche Berechnung ausführen. Aber auch andere Funktionen auf Arrays lassen sich zu einem Muster verallgemeinern. Im folgenden Abschnitt kombinieren wir die Elemente des Arrays durch paarweises Zusammenfügen zu einer Gesamtlösung. Dies erledigt die Funktion `reduce`.

Kombinieren von Lösungen mit Reduce

Motivation für Reduce: sum und unwords

Betrachten wir nun einmal die Funktion sum(xs), die eine Summe eines aus Zahlen bestehenden Arrays berechnet.

```
function sum (xs) {
  var x = 0;
  for (var i = 0; i < xs.length; i++) {
    x = x + xs[i];
  }
  return x;
}
```

So können wir die Gesamtzahl der Gäste berechnen, wenn wir 7 Kolleginnen, 4 Verwandte und 15 Nachbarinnen einladen.

```
sum([7, 4, 15]);
> 26
```

Eine andere Funktion, unwords(xs), fügt ein Array von Strings zu einem einzelnen String zusammen:

```
function unwords (xs) {
  var r = "";
  for (var i = 0; i < xs.length; i++) {
    r = r + xs[i];
  }
  return r;
}

unwords(["Kerala\n", "Tamil\n", "Nadu\n"]);
> "Kerala
  Tamil
  Nadu
"
```

Es fällt auf, dass sum und unwords nahezu identisch sind. Sie unterscheiden sich nur im Anfangswert der Variablen, die zurückgegeben wird. Bei sum ist der Anfangswert 0 und bei unwords "". Es gibt viele weitere Funktionen, die eine ähnliche Struktur haben und sich sich nur im Anfangswert und der Kompositionsfunktion unterscheiden. Da dies ein immer wiederkehrendes Muster ist, lohnt es sich hier zu verallgemeinern.

Die Idee ist, eine Funktion zu definieren, welche die Kompositionsfunktion (+ in beiden Fällen), den Anfangswert (0, bzw. ""), und das Array als Argumente übergeben bekommt. Diese Funktion nennen wir, wie auch in Lisp und Scheme, reduce, weil sie ein ganzes Array zu nur einem Wert reduziert. In Haskell und ML ist die gleiche Funktion unter dem Namen fold bekannt, weil sie ein Array faltet. Die Sprache C++ implementiert reduce in der Standard Template Library unter dem Namen accumulate in <numeric>.

Iteratives reduce

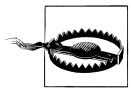
Die erste Implementierung von `reduce`, die wir uns ansehen wollen, ist eine iterative Variante.

```
function reduceIter (f, start, xs) {
  var coll = start;
  for (var i = 0; i < xs.length; i++) {
    coll = f(coll, xs[i]);
  }
  return coll;
}
```

Die Funktion reduceIter enthält eine Schleife, die auf jedem Array-Element die Funktion `f` ausführt. Die Funktion `f` bekommt zwei Argumente: das aus früheren Iterationen aufgebaute Zwischenergebnis (beim ersten Durchlauf den Anfangswert) und das aktuelle Array-Element. Das Zwischenergebnis wird in der Variablen `coll` gespeichert und nach dem letzten Schleifendurchlauf zurückgegeben.

Ein alternativer Weg, `reduce` zu implementieren, ist die Ersetzung der Schleife durch die Iterationsfunktion forEach. Auch diese Funktion ist auch eine Funktion höherer Ordnung. Sie erhält eine Funktion, und wendet diese der Reihe nach auf jedes Array-Element an.

```
function reduceFun (f, start, xs) {
  var coll = start;
  xs.forEach(function (x) { coll = f(coll, x); });
  return coll;
}
```



Spezialität des Hauses: Die Funktion `forEach` wendet genau wie `map` eine Funktion auf alle Arrayelemente an. Der Rückgabewert der Funktion wird dabei von `forEach` ignoriert, sie wird nur für die Nebenwirkungen ausgeführt. Die Funktion `map` gibt hingegen die Rückgabewerte für die Array-Elemente in einem neuen Array zurück.

Rekursives reduce

Im Paradigma, also der Denkweise der funktionalen Programmierung gibt es durch die Zustandslosigkeit keine Schleifen wie `for`, `while` und Freundinnen, um den Programmfluss zu steuern. Stattdessen gibt es die Rekursion: Die Funktion ruft sich selbst auf, um die Eingabe-Datenstruktur zu durchlaufen und/oder eine Ausgabe-Datenstruktur aufzubauen (siehe Kapitel 5). Funktional gedacht definieren wir also rekursiv:

```
function reduceRec (f, start, xs) {
  if (xs.length === 0) return start;
  return reduceRec(f, f(start, xs[0]), xs.slice(1));
}
```

Der Operator `===` ist in JavaScript (und PHP) der Identity-Operator, der prüft, ob zwei Werte gleich sind, ohne dabei eine Typkonversion durchzuführen. Dadurch ist er unter Umständen schneller als der `==`-Operator. Vor allem aber ist er sicherer, da in JavaScript die Typkonversion nach komplizierten Regeln erfolgt (siehe Kapitel 8 im Abschnitt »Falle 1: Automatische Typkonversion / Coercion« auf Seite 141). Die Operatoren `===` und `!==` sind ihren »evil twins« `==` und `!=` vorzuziehen, wie Douglas Crockford es in *JavaScript: The Good Parts* empfiehlt.

Die Funktion `slice`, die uns mit zwei Argumenten schon in Kapitel 1 begegnet ist, gibt mit nur einer Position als Argument den Rest eines Arrays ab der gegebenen Position aus, für `xs="Hallo"` wäre `xs.slice(1)` beispielsweise `"allo"`. Somit entspricht `xs.slice(1)` der Listen-Funktion `tail` in vielen anderen Sprachen, die das Ende einer Liste ab dem ersten Element zurückgibt. Dabei verändert `slice` das Original-Array nicht, sondern kopiert die entsprechenden Elemente oder Objektreferenzen in ein neues Array.

Vordefiniertes reduce auf Arrays

Die Funktion `reduce` gibt es seit der JavaScript-Version 1.8 bereits vordefiniert für Arrays. Dabei wird die Funktion auf dem Array-Objekt als Objekt-Methode ausgeführt, also in der Form `Array.reduce(f, start)`. Oben haben wir unser `reduce` jedoch nicht als Objekt-Methode definiert, sondern übergeben das Array `xs` als zusätzlichen Parameter. Beide Versionen haben ihre Berechtigung, puristische Vertreter des funktionalen Paradigmas argumentieren jedoch häufig, dass die Implementierung als Objekt-Methode funktionale und objektorientierte Ansätze unfein vermischt (*http://swizec.com/blog/javascripts-native-map-reduce-and-filter-are-wrong/swizec/1873*) – sozusagen ein bisschen Monsanto-Aubergine. Wir können beide Varianten verwenden, wenn wir den Aufruf entsprechend anpassen.

Das vordefinierte `reduce` bekommt den Startwert als letztes Argument. Er ist optional und kann weggelassen werden. Dann benutzt `reduce` das erste Element des Arrays. Unsere bisher definierten Varianten von `reduce` benötigen immer einen Anfangswert. Wollen wir trotzdem einen optionalem Anfangswert, um im Verhalten dem eingebaute reduce zu entsprechen, so müssen wir unser `reduce` mit einer Fallunterscheidung umgeben:

```
function reduceWrapper (f, xs, start) {
  if (start === undefined)
    return reduceRec(f, xs[0], xs.slice(1));
  else
    return reduceRec(f, start, xs);
}
```

Reduce im Einsatz: sum und unwords

Nun möchten wir endlich unsere Summenfunktion mithilfe von `reduce` schreiben. Das kann etwa so aussehen:

```
[7, 4, 15].reduce(plus, 0);
> 26
```

Mit O. Steeles Bibliothek *functional.js* geht sogar `sum = reduce('+', 0);`

Da wir wissen, dass `reduce` eine »normale« JavaScript-Funktion in Präfixschreibweise benötigt, haben wir wieder die Funktion `plus` verwendet, die wir schon im vorherigen Kapitel definiert haben.

Mittels `plus` und `reduce` können die Funktionen `sum` und `unwords` in jeweils einer Zeile definiert werden. Hier verwenden wir die von JavaScript bereitgestellte Funktion `reduce` aus dem Array-Prototyp. Diese wird auf dem Array `xs` aufgerufen, und bekommt die Funktion und den Startwert übergeben.

```
function shortsum (xs) { return xs.reduce(plus, 0); }
shortsum([7, 4, 15]);
> 26

function shortunwords (xs) { return xs.reduce(plus, ""); }
shortunwords(["Kerala\n", "Tamil\n", "Nadu\n"]);
> "Kerala
  Tamil
  Nadu
"
```

Reduce ist also eine allgemeine Funktion, die ein Array durch die Anwendung einer zweistelligen Funktion zu einem Wert reduziert. Wir haben gesehen, dass der Startwert bei der mitgelieferten Version auf Arrays auch weggelassen werden kann. Dies hat zur Folge, dass `reduce` dann mit den ersten beiden Werten des Arrays startet.

```
[7, 4, 15].reduce(plus);
> 26
```

Left- und Right-Reduce

In JavaScript ist das eingebaute `reduce` ein left-reduce. Das bedeutet, dass die erste, also innerste Anwendung der binären Funktion sich links befindet, wenn man den auszurechnenden Ausdruck mit Klammern schreiben würde (siehe auch *http://foldl.com*).

Den auszurechnenden Ausdruck, der durch `reduce` erzeugt wurde, kann man sich auch als Baum vorstellen:

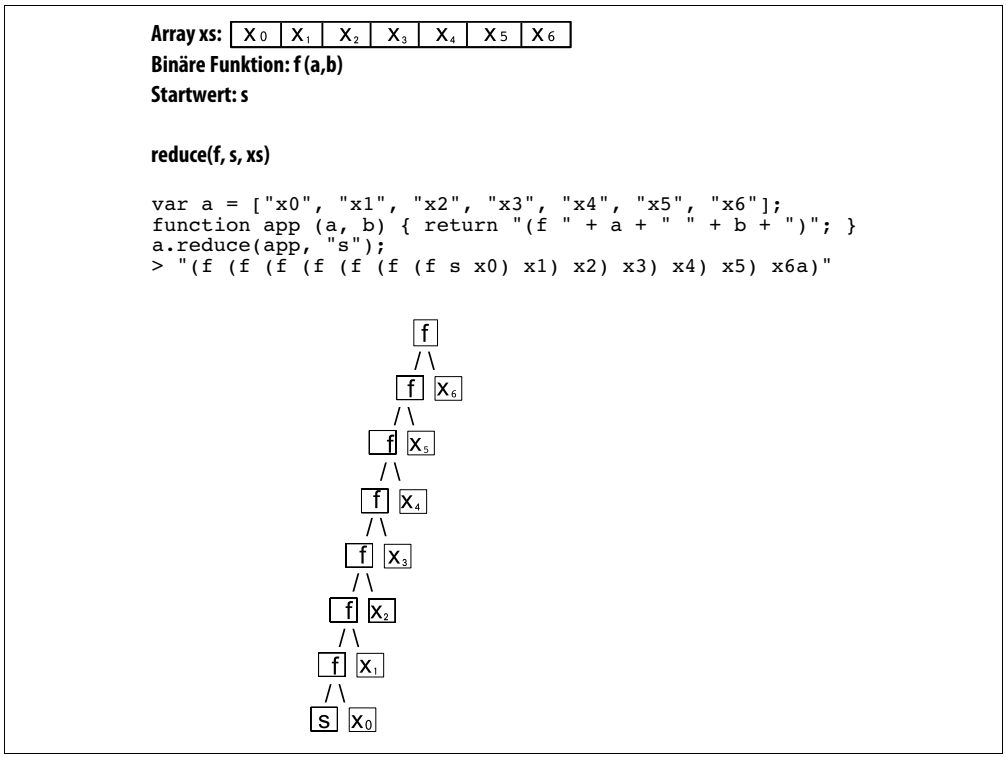


Abbildung 4-2: Der Baum der Funktionsanwendungen (abstrakter Syntaxbaum), der durch `reduce` *aus dem Array* `xs` *erzeugt wurde. Die Hilfsfunktion* `app` *gibt die Anwendung einer Funktion f mit Klammern aus. Beim Traversieren der Knoten kann der geklammerte Ausdruck abgelesen werden, der letztendlich ausgerechnet wird. Die Funktion* `f` *wird auf ihre Kindknoten als Argumente angewandt.*

Anders herum kann man sich natürlich auch den Baum aufbauen. Diese Funktion wird Right-Reduce genannt und ist ebenfalls seit der JavaScript-Version 1.8 unter dem Namen `Array.reduceRight` vorhanden.

Aber aufgepasst: Das Right-Reduce für Arrays entspricht nicht dem allgemein in der funktionalen Programmierung bekannten Right-Reduce (z.B. `foldr` in Haskell, s. auch *http://foldr.com/*). Sowohl bei `reduceRight` für Arrays in JavaScript, als auch bei `reduceRight` aus *underscore.js* und *wu.js* landet das Startelement trotzdem links außen, und der Baum wird genau wie beim normalen `reduce` auch linkslastig aufgebaut. Das Array wird dabei nur rückwärts eingelesen. Konzeptuell besteht also ein Unterschied zwischen der Version in JavaScript gegenüber der Version in Haskell und allen anderen Sprachen, sowie der Erläuterung in der Wikipedia.

Das in JavaScript implementierte `reduceRight` durchläuft das Array einfach nur rückwärts, ohne dabei einen rechtslastigen Baum aufzubauen. Wir schreiben uns daher unser eigenes `reduceRight`, das sowohl das Array rückwärts durchläuft, als auch einen rechtslastigen Baum aufbaut, so wie es in der Literatur üblicherweise beschrieben wird. Die

Array xs: | x_0 | x_1 | x_2 | x_3 | x_4 | x_5 | x_6 |

Binäre Funktion: f (a,b)

Startwert: s

reduceRight(f, s, xs)

```
var a = ["x0", "x1", "x2", "x3", "x4", "x5", "x6"];
function app (a, b) { return "(f " + a + " " + b + ")"; }
a.reduceRight(app, "s");
> "(f (f (f (f (f (f (f s x6) x5) x4) x3) x2) x1) x0)"
```

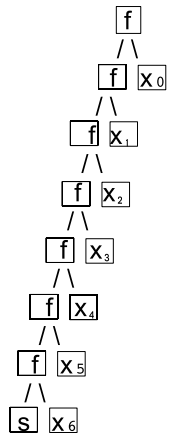


Abbildung 4-3: Der Baum der Funktionsanwendungen, der durch reduceRight in JavaScript erzeugt wurde.

Rechtslastigkeit erreichen wir, indem wir die jeweils vorhergehende Teillösung beim Schritt der binären Funktionsanwendung in das zweite Argument der binären Funktion stecken. Wenn wir die Teillösung in das erste Argument stecken würden, erhielten wir hingegen einen linkslastigen Baum, wie schon beim »normalen« `reduce`.

```
var myReduceRight = function (f, init, xs) {
  var acc = init;
  for (var i = xs.length - 1; i >= 0; i--) {
    acc = f(xs[i], acc);
  }
  return acc;
}
```

Testen wir unser `myReduceRight`, indem wir die Anwendung der übergebenen Funktion `f` mit Klammern ausgeben.

```
var a = ["x0", "x1", "x2", "x3", "x4", "x5", "x6"];
// app gibt eine Funktionsanwendung auf der Konsole aus, s. auch Abbildung
function app (a, b) { return "(f " + a + " " + b + ")"; }
myReduceRight(app, "s", a);
> "(f x0 (f x1 (f x2 (f x3 (f x4 (f x5 (f x6 s)))))))"
```

Array xs: | x_0 | x_1 | x_2 | x_3 | x_4 | x_5 | x_6 |

Binäre Funktion: f (a,b)

Startwert: s

foldr(f, s, xs)

```
Prelude> let xs = ["x0", "x1", "x2", "x3", "x4", "x5", "x6"]
Prelude> foldr (\x y -> "(f "++ x ++" "++ y ++")") "s" xs
"(f x0 (f x1 (f x2 (f x3 (f x4 (f x5 (f x6 s)))))))"x2) x1) x0)"
```

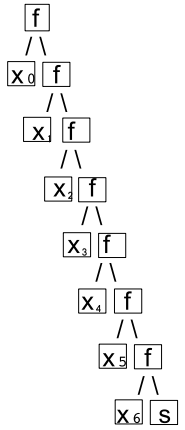


Abbildung 4-4: Bild zur Visualisierung von foldr in Haskell

Nach dem gleichen Schema stellt die Bibliothek *functional.js* unter dem Namen `foldr` eine Funktion bereit, die einen rechtslastigen Baum aufbaut, so wie wir es erwarten.

Fassen wir zusammen: `reduce` ist eine strukturelle Transformation, die ein Array oder eine Liste mittels einer binären Funktion, die zwei Argumente und einen Rückgabewert hat, zu einem einzelnen Wert reduziert. Dabei wird aus dem Array ein links- oder rechtslastiger Baum konstruiert, mit einer binären Funktion an den Verzweigungen, die auf ihre beiden Kindknoten angewendet wird. Die Elemente der Ursprungsliste können an den Blättern abgelesen werden. Bei der wiederholten Anwendung der binären Funktion, also der Auswertung des im Baum dargestellten Ausdrucks, wird aus den Werten des Ursprungs-Arrays letztendlich ein einzelner Wert.

Man muss betonen, dass `reduceRight` vor allem in Kombination mit der Auswertungsmethode *Lazy Evaluation* (faule Auswertung) besonders nützlich ist. Hierbei werden nur die Ausdrücke im Programm berechnet, die für die Rückgabe des Ausgabewertes unbedingt nötig sind. Wir beschreiben die Lazy Evaluation intensiver in Kapitel 7 im Abschnitt »Lazy Evaluation – die faule Auswertung« auf Seite 112. In der lazy ausgewerteten Sprache Haskell ist `foldr` interessant, da bei bestimmten Berechnungen die Rechts-Rekursion mit rekursivem Aufruf im rechten Argument der binären Funktion, die durch `foldr` aufgebaut wird, für das Ergebnis gar nicht benötigt wird (*http://en.wikipedia.org/wiki/Fold_%28higher-order_function%29#*

Evaluation_order_considerations). Auf diese Weise kann man sogar ohne Probleme auf endlosen Listen (Kapitel 5 im Abschnitt »Eine zirkuläre Liste« auf Seite 81) Berechnungen mit `foldr` ausführen!

Die Lazy Evaluation wird in JavaScript anders als in manchen funktionalen Sprachen nicht von der Sprache selbst unterstützt. Es gibt allerdings Bibliotheken, die die Lazy Evaluation in JavaScript ermöglichen (wu.js *http://fitzgen.github.com/wu.js/*). Bei der strikten Evaluation hat jedoch das normale `reduce` die Nase vorn, da man es in Form der Tail-Rekursion formulieren kann (Kapitel 5). Diese kann von einigen Compilern direkt in eine Schleife übersetzt und besonders effizient ausgeführt werden.

Auswählen von Lösungen mit Filter

Wir können also bereits eine Funktion mittels `map` auf jedes Element eines Arrays anwenden, um es zu transformieren. Die transformierten Elemente können wir dann mit `reduce` zu einem Endergebnis zusammenfügen. Eine weitere wichtige Funktion zum Transformieren eines Arrays ist das Filtern von Array-Elementen anhand einer bestimmten Eigenschaft. Dabei wird ein Ergebnis-Array aufgebaut, das nur aus den Elementen besteht, die diese bestimmte Eigenschaft besitzen. Diese Funktion wird `filter` genannt. Sie wird mit einem Array und einer Prädikatsfunktion gefüttert, die Auswahl-Eigenschaft beschreibt. Für jedes Array-Element gibt die Prädikatsfunktion entweder `true` oder `false` zurück, je nachdem ob das Element die gewählte Eigenschaft besitzt oder nicht. Die Funktion `filter` wählt nun alle Array-Elemente aus, die diese gewählte Eigenschaft besitzen.

In JavaScript ist die `filter`-Funktion auf Arrays bereits vordefiniert. Mithilfe von `filter` können wir herausfinden, welche Gewürze noch in ausreichender Menge vorrätig sind, um unser Gericht gut zu würzen. Für uns sind das mindestens 10 Gramm pro Gewürz.

```
function havePlenty (element) {
  return (parseInt(element.quantity_in_grams) >= 10);
}
```

Wie wir sehen, ist dies nur beim schwarzen Pfeffer der Fall.

```
spices.filter(havePlenty);
> [ { name: 'Schwarzer Pfeffer',
    state: 'gekörnt',
    quantity_in_grams: '11',
    active_compound: 'Piperine' } ]
```

Mit der Hilfsfunktion `not` können wir durch Funktionskomposition (Kapitel 3 im Abschnitt »Implizite Argumente, Komposition und Point-Free-Style« auf Seite 42) auch sehr elegant das Array der Gewürze konstruieren, die wir nachkaufen müssen.

```
function not (x) { return !x; }
spices.filter(not.c(havePlenty));
```

Wir finden heraus, dass Nelken und Zimt zur Neige gehen.

```
> [ { name: 'Nelken',
    state: 'gemahlen',
    quantity_in_grams: '2',
    active_compound: 'Eugenol' },
  { name: 'Zimt',
    state: 'Gemahlen',
    quantity_in_grams: '1',
    active_compound: 'Cinnamaldehyde' } ]
```

Zusammenhang zwischen Reduce und Map

Die Funktion `reduce` ist so mächtig, dass wir auf ihren Schultern unser ganzes Universum von Funktionen höherer Ordnung errichten können ;-). Wir können mittels `reduce` sowohl `map` als auch `filter` definieren.

Um map zu definieren, brauchen wir eine Funktion, die Array-Elemente verbindet, um diese `reduce` zu übergeben. Die Argumente sind das bisher aufgebaute Array und das aktuell zu bearbeitende Element, das Resultat ist das erweiterte Array. Auf diese Weise können wir wieder ein Array erhalten.

Diese Funktion wird `cons` genannt, kommt ursprünglich aus Lisp, ist aber auch in anderen funktionalen Programmiersprachen wie Haskell, den Lisp-Dialekten und Scheme zentral am Aufbau der unterliegenden Listen-Datenstrukturen beteiligt. Der Aufbau von Listen als rekursiver Datentyp, den wir im folgenden Kapitel 5 im Abschnitt »Strukturelle Rekursion auf Listen« auf Seite 77 kennen lernen, funktioniert nach dem gleichen Prinzip.

Gleichzeitig müssen wir aber auch die Funktion `f`, die auf jedes Element angewendet werden soll, beim Aufruf von `cons` anwenden. Wir brauchen hier also eine Spezialversion von `cons`, die wir in JavaScript folgendermaßen definieren:

```
function consApp (f) {
  return function (list, item) {
    list.push(f(item));
    return list;
  };
}
```

Und so sieht `map`, mit `reduce` ausgedrückt, aus:

```
function rmap (f, xs) {
  return reduceRec(consApp(f), [], xs);
}
```

Verallgemeinerung der Funktionen höherer Ordnung auf Arrays

Beim Gemüseschneiden in der Küche möchten wir verschiedene Aktivitäten verbinden: Die schlechten Stücke herausschneiden, das Gemüse in Scheiben schneiden und es dann in den Topf geben. Wir haben nur begrenzten Platz auf dem Schneidebrett, daher können wir nicht erst jedes Gemüse auf schlechte Stellen untersuchen, danach alle guten Stücke in Scheiben schneiden, und alles erst wenn es fertig geschnitten ist in den Topf geben. Besser ist es, während des Schneidens gleichzeitig die schlechten Stücke wegzuwerfen, und das geschnittene Gemüse in den Topf zu geben, sobald das Schneidebrett voll ist.

Ähnlich können wir auch mit `map`, `reduce` und `filter` verfahren. Zuerst schreiben wir eine Funktion `hom`, die `map`, `reduce` und `filter` kombiniert. Diese erleichtert uns, zu verstehen, dass die Funktionen ein gemeinsames Konzept teilen, wie wir es schon bei der Definition von `map` mit Hilfe von `reduce` erahnen konnten. Darüber hinaus können wir die in der Funktion `hom` kombinierten Funktionen auf großen Datenmengen ausführen, ohne die Daten wiederholt durchlaufen zu müssen. Die Funktion `hom` filtert das Array anhand eines Prädikates, wendet ein Mapping an und kombiniert die Ergebnisse mit Hilfe der Funktion `red` in ein Resultat.

In der Definition der Funktion `hom` können wir Aspekte von `map`, `reduce` und `filter` bei einem Vergleich mit ihren Implementierungen wiedererkennen:

```
function hom (xs, red, mapping, predicate, start) {
  var start = start || [];
  if (xs.length > 0) {
    var fst = xs[0];
    var rst = hom(xs.slice(1), red, mapping, predicate, start);
    if (predicate(fst))
      if (rst.length > 0)
        return red(mapping(fst), rst);
      else
        return [mapping(fst)];
    else
      return rst;
  } else
    return start;
}
```

Das Startelement ist wieder optional, wird es weggelassen, dann wird ein leeres Array verwendet. Die Funktion `hom` (`xs`, `red`, `mapping`, `predicate`, `start`) hat den gleichen Effekt wie `xs.filter(predicate).map(mapping).reduce(red, start)`.

Nun können wir wiederum `map`, `filter` und `reduce` als Spezialfälle von `hom` definieren. Wir brauchen dazu einige Hilfsfunktionen als Grundbausteine. Die Funktion `cons` erhält ein Element und einen Array, und fügt das Element vorne an das Array an. Dies ist sehr ähnlich zu dem `consApp` von oben, allerdings ohne die Anwendung von `f`.

```
function cons (x, xs) {
  xs.unshift(x); // vorn einfügen
  return xs;
}
```

Außerdem verwenden wir die Identitätsfunktion, die ihr Argument unverändert zurückgibt.

```
function id (x) { return x; }
```

Und wir benötigen die Funktion, die unabhängig vom Argument immer `true` zurückgibt:

```
function yes (x) { return true; }
```

Die Funktion `map` benutzt nun `hom` mit `cons` als Kombinationsfunktion, dem leeren Array als neutralem Element und der übergebenen Funktion `mapping`.

```
function mymap (fun, xs) {
  //das Prädikat gibt immer wahr zurück
  //das Startelement wurde weggelassen
  return hom(xs, cons, fun, yes);
}
```

In der Funktion `filter` findet kein `mapping` statt, da die Identitätsfunktion benutzt wird. Neben der Sequenz wird das Prädikat zum Filtern übergeben.

```
function filter (pred, xs) {
  //das Startelement wurde weggelassen
  return hom(xs, cons, id, pred);
}
```

Die Funktion `reduce` wendet die Kombinationsfunktion an.

```
function reduce (comb, start, xs) {
  return hom(xs, comb, id, yes, start);
}
```

Wir können nun die Namen der Gewürze aus Kapitel 2 im Abschnitt »Abstraktion von einem Codeblock: Funktionen« auf Seite 24 ausgeben, die schon gemahlen sind, und zu einem String kombinieren:

```
hom(spices, function (a, b) { return a + " und " + b },
            function (x) { return x["name"] },
            function (y) { return y["state"].toLowerCase() === "gemahlen" });
> 'Nelken und Zimt'
```

Wie wir in diesem Kapitel gesehen haben, können wir mit Hilfe von Funktionen höherer Ordnung unsere Berechnungen komplett ohne die Verwendung von imperativen Kontrollstrukturen wie Schleifen strukturieren. Wir haben die wichtigsten Vertreterinnen dieser Funktionen `reduce`, `map` und `filter` kennengelernt und diese schließlich mit der Funktion `hom` verallgemeinert. Im Zusammenspiel mit den funktionalen Konzepten aus dem vorherigen Kapitel, der Infix-Präfix-Konversion, Curryfizierung und Funktionskomposition bilden sie eine gute Grundausstattung für die funktionale Küche. Außerdem haben wir weitere nützliche Funktionen als Bausteine kennengelernt. Die Funktion `flip`,

die in einer binären Funktion das linke und rechte Argument austauscht, und dann noch `yes` und `not`. Es gibt noch unzählige weitere nützliche Funktionen, etwa `concatMap`, welches die Ergebnisse von `map` zugleich konkateniert. Hilfreiche Funktionen wie diese können nach den Beispielen in diesem Kapitel von Lesern und Leserinnen selbst implementiert werden.

```
(λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s
z)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

KAPITEL 5

Rekursion

Vom Problem zum Programm: Spezifikation, Ein- und Ausgabe

Noch bevor wir ein Programmierproblem durch den eigentlichen Programmcode lösen, sollten wir uns Gedanken machen, was unser Programm tun soll. Der Schritt vom Problem zum Programm ist die Spezifikation oder Problembeschreibung unseres Programms. Die Spezifikation ist der vom Problem ausgehende formalisierende Schritt vor der Modellierung aus Kapitel 2, Abschnitt »Modellierung: Theorie und Praxis« auf Seite 22, wobei es zwischen beiden starke Zusammenhänge gibt. Zur Spezifikation eines Problems machen wir uns kurz Notizen darüber oder zumindest Gedanken über die Form der Ein- und Ausgabe unseres Problems und was in groben Schritten zwischen Ein- und Ausgabe passieren muss.

Häufig wird in der funktionalen Programmierung die Eingabe in solchen Zwischenschritten systematisch verarbeitet oder eine Ausgabe entsprechend aufgebaut. Da wir aufgrund der Seiteneffektfreiheit keine Zuweisung und dadurch auch keine Schleifen zur Programmstrukturierung benutzen, also geschieht dieses systematische Durchlaufen der Datenstrukturen durch Rekursion durch wiederholte Funktionsaufrufe, wobei sich eine Funktion solange selber aufruft, bis ein Basisfall erreicht ist. Ein Zwischenschritt beim Durchlauf der Datenstruktur entspricht dann einem Funktionsaufruf.

Zerteilen eines Programmierproblems

Wie wir an einem einfachen Programmierbeispiel wie dem Rezeptor aus Kapitel 1 im Abschnitt »Rezeptor: Über hundert Curry-Gerichte, funktional und automatisch erstellt« auf Seite 14 sehen können, müssen wir für eine übersichtliche Lösung das Hauptproblem »Generieren eines Curry-Rezepts« in kleinere Teilprobleme wie »Auswahl des Gemüses« unterteilen.

Diese logische Unterteilung spiegelt sich auch in der Struktur des Programmcodes wider: Teilaufgaben werden in Codeblöcke eingeteilt, die durch geschweifte Klammern abgegrenzt sind. Meistens ist ein solcher Codeblock der Rumpf einer Funktion im Programmcode (siehe Kapitel 2 im Abschnitt »Abstraktion von einem Codeblock: Funktionen« auf Seite 24) und kann mit einem Funktionsnamen benannt und mit wechselnden Argumenten wiederholt ausgeführt werden.

Divide and Conquer

Das Zerteilen des gestellten Problems in Teilprobleme, um diese gesondert zu lösen und die Lösungen wieder zu einer Gesamtlösung zu kombinieren, ist ein genereller Lösungsansatz in der Programmierung. Dieser Ansatz wird auch *Teile und Herrsche* oder *Divide and Conquer* genannt, nach der lateinischen Redewendung *Divide et impera*, die durch Julius Cäsar bekannt wurde, aber schon in Sun Tzus Buch »Die Kunst des Krieges« erwähnt wird.

Mit Hilfe von Funktionen höherer Ordnung könnte man das Divide-And-Conquer-Schema ausdrücken als

- eine Funktion `istLoesbar(x)`, die sagt, ob ein Problem einfach lösbar ist,
- eine Funktion `loese(x)`, die ein einfaches Problem löst,
- eine Funktion `teile(x)`, die ein Problem in eine Liste von Problemen unterteilt und
- eine Funktion `herrsche(as)`, die eine Liste von Lösungen in eine Gesamtlösung überführt.

Das Schema geht dann ungefähr folgendermaßen:

```
var teileUndHerrscheRekursiv = function
  if (istLoesbar(x)) {
    return loese(x);
  } else {
    var xs = teile(x);
    return herrsche(xs.map(teileUndHerrscheRekursiv));
  }
}
```

Der Lösungsvorgang wird hierbei in zwei Fälle unterteilt, in den Basisfall, der ein Problem direkt löst, und in den rekursiven Fall, der das Problem weiter unterteilt und für jedes Teilproblem einen rekursiven Funktionsaufruf ausführt.

Wie die Probleme nun genau in diese Fälle unterteilt und dann gelöst werden, hängt von der Struktur des Problems ab. Die unterliegende Problemstruktur ist oft an die Struktur der Ein- und/oder Ausgabedaten gekoppelt.

Viele effiziente und klassische Algorithmen arbeiten nach dem Divide-And-Conquer-Schema, z.B. Sortieralgorithmen, Parser und der Algorithmus zur Berechnung der schnellen Fourier-Transformation. Diese Transformation bildet kontinuierliche Signale vom Zeit- auf den Frequenzraum ab und ist wichtig für das Kodieren von Signalen wie Musik- oder Videodaten.

Rekursive Funktionen

Rekursion begegnet vielen angehenden Programmierenden zunächst im Mathematikunterricht, als rekursive mathematische Funktion.

Ein bekanntes Beispiel ist die Fibonacci-Folge, eine unendliche Zahlenfolge, deren Glieder sich rekursiv aus den vorherigen Gliedern der Folge berechnen lassen.

Der Beginn der Fibonacci-Folge ist: 0, 1, 1, 2, 3, 5, 8, 13, ...

Die Angabe der Fibonacci-Folge ist durch eine Rekursionsvorschrift bedeutend einfacher als die Angabe durch eine Funktion in einer geschlossenen Form ohne Fallunterscheidung. Deshalb möchten wir sie im Computer rekursiv berechnen.

Die Fibonacci-Folge

Die geschlossene Form der Fibonacci-Funktion steht in Verbindung mit dem *goldenen Schnitt* und der *goldenen Zahl*. Konstruiert man ein Rechteck mit dem goldenen Schnitt als Seitenverhältnis und teilt dieses wiederum im goldenen Schnitt, und so weiter, erhält man in jedem Schritt eine fast komplette Zerlegung des Rechtecks in immer kleiner werdende Quadrate. Mit Hilfe von Viertelkreisen durch die entstehenden Quadrate kann man die Fibonacci-Spirale zeichnen. Diese approximiert die goldene Spirale, eine logarithmische Spirale mit dem goldenen Schnitt als Wachstumsfaktor.

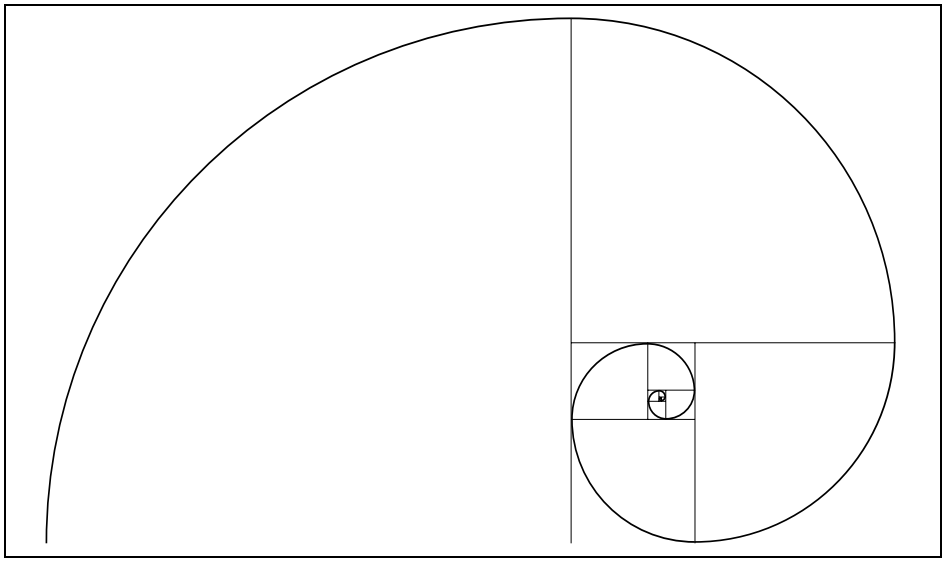


Abbildung 5-1: Die Fibonacci-Spirale

Die Fibonacci-Folge ist auch beim Curry-Kochen vertreten, denn diese Spirale liegt dem Wachstumsmuster vieler Pflanzen zugrunde. Die Kerne in der Sonnenblume sind in der Blüte nach ihr angeordnet, und auch in der Ananas und der Artischocke finden wir diese Spirale wieder.

Der umfassende Eintrag in *Sloane's Online Encyclopedia of Integer Sequences* (*http://oeis.org/A000045*) zeigt, dass die Fibonacci-Folge in vielen Zusammenhängen eine große Rolle spielt, und ist definitiv einen Besuch wert.

Die Fibonacci-Folge ist rekursiv definiert nach folgendem Schema:

```
f(n) = f(n - 1) + f(n - 2) für n >= 2
f(0) = 0
f(1) = 1
```

Anhand der rekursiven Funktionsvorschrift sehen wir schon: Das Problem, die n-te Fibonaccizahl zu berechnen, lässt sich zerlegen in einen rekursiven Fall für `n >= 2`, und zwei Basisfälle für `0` und `1`. Im Basisfall kann die Lösung ohne rekursiven Aufruf ausgerechnet, oder wie hier sogar direkt hingeschrieben werden. Im rekursiven Fall ruft die Funktion sich ein oder mehrere Male selbst auf und verwendet die so erhaltenen Teillösungen zur Berechnung einer Gesamtlösung. In unserem Beispiel werden die beiden Vorgänger-Zahlen (`n – 1`) und (`n – 2`) im Teile-Schritt als Teilprobleme rekursiv berechnet, und dann im Herrsche-Schritt in einer Summe kombiniert. Es wird also zur Lösung nach dem Divide-And-Conquer Schema vorgegangen.

Eine rekursive Berechnung einer Funktion hat zudem starke Analogien zum Beweisschema der strukturellen Induktion in der Mathematik, wenn man diese rückwärts betrachtet. ;-)

Zur rekursiven Berechnung der `n`-ten Fibonacci-Zahl im Computer können wir in der funktionalen Programmierung einfach die mathematische Funktionsdefinition hinschreiben.

Die Fallunterscheidung drücken wir im Programm durch eine Bedingung aus:

```
function fib (n) {
  if (n === 0) return 0;
  else if (n === 1) return 1;
  else return fib(n - 1) + fib(n - 2);
}
```

Die Berechnung folgt durch die Fallunterscheidung dem rekursiven Divide-And-Conquer Schema: das Problem wird für die `n`-te Fibonacci-Zahl, wenn `n >= 2` ist, rekursiv gelöst, indem die Lösungen für `n – 1` und `n – 2` kombiniert werden. Für die Zahlen `0` und `1` befinden wir uns in einem Basisfall und die Lösung wird direkt zurückgegeben.

Wir können die Fallunterscheidung auch durch einen `switch` ausdrücken oder – ganz knapp – mithilfe des ternären Operators.

Bei der rekursiven Berechnung spult der rekursive Fall die Berechnung Schritt für Schritt durch, bis die Basisfälle die Berechnung zum Stoppen bringen.

Deshalb ist es auch sehr wichtig, dass wir die Basisfälle nicht vergessen, denn sonst würde unsere Berechnung niemals enden! Wir blieben dann in einer Endlosschleife stecken, und das sogar ohne die Benutzung eines Schleifenkonstrukts.

Wir fassen zusammen, dass die Rekursion eine kurze und elegante Beschreibung bekannter Probleme ermöglicht und im rein funktionalen Paradigma ein wichtiges Mittel zur Steuerung des Kontrollflusses ist, da es in diesem Paradigma keine Zuweisung von Werten, und somit keine Schleifen als Kontrollstrukturen gibt, auch wenn uns Konstrukte

wie `map` und `reduce` aus dem vorherigen Kapitel, und Monaden, die wir in Kapitel 9 betrachten, ein ähnliches Vorgehen mit rein funktionalen Bordmitteln ermöglichen.

Eine rekursive Berechnung

Was geschieht nun genau bei der Berechnung einer rekursiven Funktion wie in unserem Beispiel? Dies können wir uns veranschaulichen, indem wir während der Berechnung auf die Konsole jeden Schritt ausgeben lassen.

```
function fib (n) {
  if (n === 0) {
    console.log("Basis 0");
    return 0;
  } else if (n === 1) {
    console.log("Basis 1");
    return 1;
  } else {
    var n_1 = fib(n - 1);
    var n_2 = fib(n - 2);
    console.log("Rekursion für " + n + ": " + n_1 + " + " + n_2 + " = " + (n_1 + n_2));
    return n_1 + n_2;
  }
}

fib(4);
> Basis 1
> Basis 0
> Rekursion für 2: 1 + 0 = 1
> Basis 1
> Rekursion für 3: 1 + 1 = 2
> Basis 1
> Basis 0
> Rekursion für 2: 1 + 0 = 1
> Rekursion für 4: 2 + 1 = 3
> 3
```

Die rekursiven Aufrufe verwenden wie geplant die beiden Vorgängerergebnisse und kombinieren diese. Dabei wird jeweils bis zum Basisfall zurückgehend alles berechnet.

Fettnäpfchen bei der Rekursion

Problem: Wiederholte Lösung von Problemen

Die Rekursion für den Fall `n = 2` wird bereits in dem obigen kleinen Rechenbeispiel mehrfach berechnet. Bei Berechnungen für größere `n` häufen sich die Wiederholungen. Um diese unnötigen Wiederholungen zu vermeiden, können wir die Technik der Memoisation benutzen und die bereits berechneten Werte für n in einem Array tabellieren.

Dies muss in JavaScript durch die Programmiererinnen, also uns, erledigt werden. Andere funktionale Sprachen erlauben Techniken einer »impliziten Memoisation«, oder

man könnte diese direkt implementieren. Wie das für Common Lisp und andere Sprachen geht, steht in Peter Norvigs Veröffentlichung »Techniques for Automatic Memoization with Applications to Context-Free Parsing,« Computational Linguistics, Vol. 17 No. 1, pp. 91–98, March 1991.

Lösung: Memoisation / Tabellierung

Zur Berechnung der n-ten Fibonacci-Zahl im Computer mit Hilfe von Memoisation merken wir uns die vorherigen Lösungen in einem Array. Wir wandeln die rekursive Berechnung also in eine iterative um.

```
function fastfib (n) {
  var fibs = new Array(); // Array für vorherige Lösungen
  fibs[0] = 0; // Basisfall 1
  fibs[1] = 1; // Basisfall 2
  for (var i = 2; i <= n; i++) { // Iteration bis zur Zahl n
       // Kombination der beiden vorherigen Lösungen zur neuen Lösung
       fibs[i] = fibs[i - 1] + fibs[i - 2];
  }
  return fibs[n];
}

for (var i = 0; i <= 20; i++) { console.log("fastfib(" + i + ") = " + fastfib(i)) };
```

In JavaScript können wir die ursprüngliche Rekursion auch in eine Rekursion mit einer Closure verwandeln. Die Rekursion mit Closure sieht fast genauso aus, berechnet aber nichts mehrfach. Durch die Kombination von dynamischen Scoping und Closures können wir schon berechnete Lösungen in einem Array tabellieren.

Diese Idee ist in der Funktion `fibclosure` umgesetzt, einer Funktion, die unmittelbar aufgerufen wird. In `fibclosure` wird zuerst ein Array `fibs` mit den Basisfällen initialisiert und dann eine Funktion zurückgegeben, die sich bei der Addition der beiden Vorgänger selbst aufruft. Die äußere Funktion wird verwendet, damit das Array `fibs` nicht global, sondern nur innerhalb der Funktion sichtbar ist. Dies ist ein beliebter Trick bei der Programmierung in JavaScript.

JavaScript-Besonderheit als Alternative: fibonacci mit closure

```
var fibclosure = (function () {
  var fibs = [0, 1]; // Lösungs-Array direkt mit Basisfällen
  return function (n) { // closure abhängig von vorherigen Lösungen
    if (fibs[n] === undefined) { // wenn noch nicht berechnet
      fibs[n] = fibclosure(n - 1) + fibclosure(n - 2);
    }
    return fibs[n];
  };
})();
for (var i = 0; i <= 20; i++) {
     console.log("fibclosure(" + i + ") = " + fibclosure(i));
}
```

Mit den obigen Lösungen verbrauchen wir `n` Array-Zellen um die `n`-te Fibonaccizahl zu berechnen. Aus der rekursiven Funktionsdefinition ist jedoch ersichtlich, dass wir jeweils nur die vorangehenden beiden Lösungen benötigen, um die n-te Lösung zu berechnen. Wir können also eine sparsamere Version schreiben, die mit nur zwei Array-Zellen zur Speicherung der vorherigen Lösungen auskommt.

Dann wird das Vorgehen allerdings etwas schwieriger zu lesen. Wir arbeiten diesmal mit `push` und `shift` auf einem zwei-zelligen Array zur Zwischenspeicherung. Neue Lösungen werden mit `push` hineingeschoben und alte, nicht mehr benötigte, mit `shift` herausgeschoben.

```
function fastfib2 (n) {
  var fibs = new Array();
  fibs.push(0); // Basisfall 1 hineinschieben
  fibs.push(1); // Basisfall 2 hineinschieben
  for (var i = 0; i < n; i++) {
    fibs.push(fibs[0] + fibs[1]); // hier haben wir kurz drei Zellen!
    fibs.shift(); // Lösung die nicht mehr gebraucht wird, herausschieben
  }
  return fibs[0];
}
```

Quelle: *http://en.literateprograms.org/Fibonacci_numbers_%28JavaScript%29*

Problem: Verbrauch von Call Frames

Ein weiteres Problem ist der »Verbrauch« von Call Frames auf dem Call Stack durch die rekursive Berechnungen.

Unsere Funktion `fib(n)` wird im rekursiven Fall zweimal aufgerufen. Nach Beendigung der Berechnung von `fib(n)` muss das Programm wissen, an welcher Stelle im Programmcode es zurückspringen muss, hinter den ersten oder den zweiten Aufruf?

Um solche Informationen vorzuhalten, benutzt jedes Programm einen Call Stack.

Der Call Stack, auch Funktions-Aufruf-Stack oder -Stapel, ist eine Datenstruktur, die für jedes Programm zu seiner Laufzeit im Speicher vorliegt, und Informationen über den Kontrollfluss bereithält. Für jeden Funktionsaufruf wird ein neuer Call Frame auf den Call Stack gelegt. Der Call Stack ist also ein Stapel von Call Frames.

Im Call Frame stehen Informationen zur Funktion, wie die Argumente, die lokalen Variablen und die Rücksprungadresse, der Programmzeile, in der es nach Ende der Funktion weitergeht.

In unserer rekursiven Fibonacci-Funktion `fib(n)` werden bei jedem Funktionsaufruf zwei Frames erzeugt, einer für den linken und einer für den rechten Summanden. Dies passiert solange, bis die rekursive Berechnung im Basisfall endet.

Jeder Call Frame enthält dabei alle Informationen, die zur Berechnung des Funktionsaufrufs benötigt werden. In unserem Fall also die richtigen Argumente.

Ein Call Frame kostet Speicher, und der korrespondierende Funktionsaufruf kostet Rechenzeit. Bei rekursiven Berechnungen stapelt sich für jeden Rekursionsschritt ein weiterer Call Frame auf dem Stack, bis der Basisfall erreicht ist und der Stapel nach und nach wieder abgebaut werden kann. Deshalb kann eine tiefe Rekursion unter Umständen mehr Speicher benötigen, als auf dem Computer verfügbar ist, so dass die Berechnung aus Speichermangel abgebrochen werden muss.

Lösung: Tail-Rekursion

Ein verbreitetes Konzept zum Einsparen von Call Frames ist die Tail-Rekursion, sofern diese vom Compiler oder Interpreter optimiert wird. Diese Optimierung wird in JavaScript bisher noch nicht gemacht, soll aber mit dem ECMAScript-Standard 6 umgesetzt werden.

Quelle: *http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls*

Wenn der rekursive Aufruf der eigenen Funktion zugleich auch der letzte Ausdruck in der Funktion ist, wird dies Tail-Rekursion (Heck-Rekursion, End-Rekursion) genannt. Diese Art von Rekursion kann besonders gut optimiert werden. Das Ergebnis der Funktion ist das Ergebnis des rekursiven Aufrufs. Dadurch kann für den rekursiven Aufruf der aktuelle Call Frame wiederverwendet werden und es muss kein neuer Call Frame angelegt und gefüllt werden.

Betrachten wir zur Abwechslung als Beispiel die Summenfunktion, die die Summe der natürlichen Zahlen von `1` bis einschließlich `x` berechnet:

```
function sum (x) {
  if (x === 1) return 1;
  return sum(x - 1) + x;
}
```

Leider ist die letzte Anweisung hier nicht der rekursive Aufruf der Summenfunktion `sum`, sondern die Addition des vorherigen Ergebnisses mit `x`. Daher wird das `x` auch nach dem rekursiven Aufruf benötigt, um den Rückgabewert zu berechnen.

Dies können wir besser sehen, wenn wir uns die Auswertungs-Reihenfolge anschauen. Hier ist `sum` so umgeschrieben, dass diese Reihenfolge klar zu erkennen ist:

```
function sum (x) {
  if (x === 1) return 1;
  var rc = x - 1;
  var resr = sum(rc);
  var res = resr + x;
  return res;
}
```

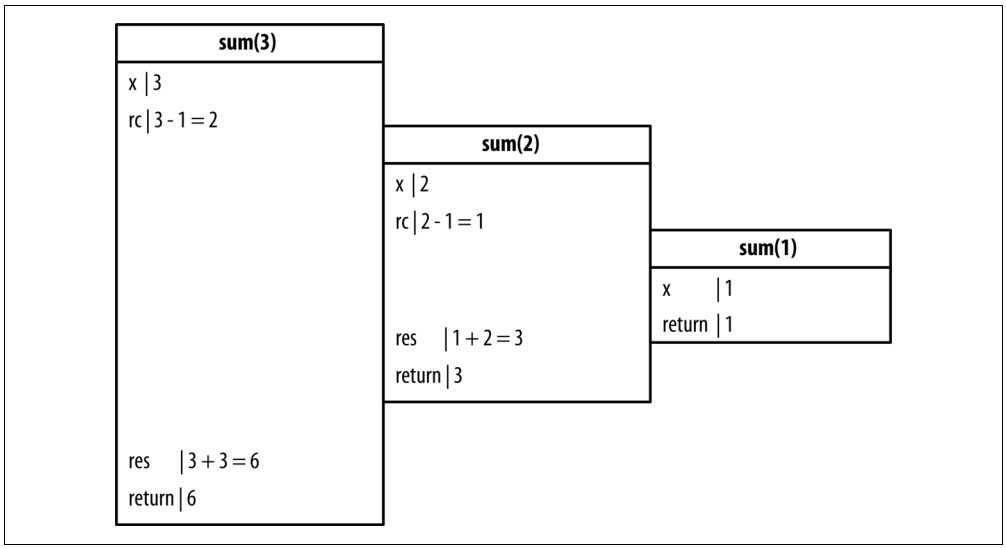


Abbildung 5-2: Der Aufruf von `sum(3)` *ruft rekursiv* `sum(2)` *auf, daher ist rechts von* `sum(3)` *die Box* `sum(2)`*. Die Variablen, die oben in jeder Box stehen, sind vor dem rekursiven Aufruf bekannt. Die Variablen und Werte, die unten stehen, sind erst nachdem der rekursive Aufruf zurückgekehrt ist, bekannt.*

Im Bild sehen wir den Funktionsaufruf mit den jeweiligen Variablen. Erst nachdem der innerste Aufruf `sum(1)` abgeschlossen ist, kann das Gesamtergebnis berechnet werden. Einige lokale Variablen werden auch nach dem Aufruf noch zur Berechnung benötigt, hier das `x`.

Wir können diese Funktion so umformulieren, dass der rekursive Aufruf erst zum Schluss kommt. Dies machen wir durch ein zusätzliches Argument `acc`, welches den aktuellen Zwischenwert enthält:

```
function sumTail (x, acc) {
  var acc = acc || 1;
  if (x === 1) return acc;
  return sumTail(x - 1, acc + x);
}
```

Wenn `acc` nicht als Argument übergeben wird, ist es `undefined` und wird auf den Startwert `1` gesetzt.

Im letzten Beispiel haben wir einen Trick benutzt, die akkumulative Rekursion. Wir können diesen Trick in der funktionalen Programmierung oft verwenden, um effizienter zum Ziel zu kommen. Wir führen dabei ein zusätzliches Argument in unserer rekursiven Funktion mit, den Akkumulator, der während der einzelnen Rekursionsschritte Information aufsammelt und an die tieferen Rekursionsschritte weitergibt. Dies erfordert eine zusätzliche Funktion, da beim »Starten« der akkumulativen Rekursion der richtige Startwert für den Akkumulator gesetzt werden muss. Erst dann kann die Version der Funktion mit dem Akkumulator, hier sumTail, aufgerufen werden. Bei der akkumulativen Rekursion ist beim Erreichen des Basisfalls bereits fast alle Arbeit getan, es muss nur noch die Basisfall-Lösung mit der Lösung im Akkumulator kombiniert werden.

Ohne einen Akkumulator betreiben wir den klassischen Fall, die Stack-Rekursion, bei der die Lösung erst auf dem Rückweg, beim Abbau des Rekursionsstacks berechnet werden kann.

Überlegen wir uns nun, in welcher Reihenfolge die Ausdrücke ausgewertet werden, kommen wir zu folgender Funktion:

```
function sumTail (x, acc) {
  var acc = acc || 1;
  if (x === 1) return acc;
  var arg1 = x - 1;
  var arg2 = acc + x;
  return sumTail(arg1, arg2);
}
```

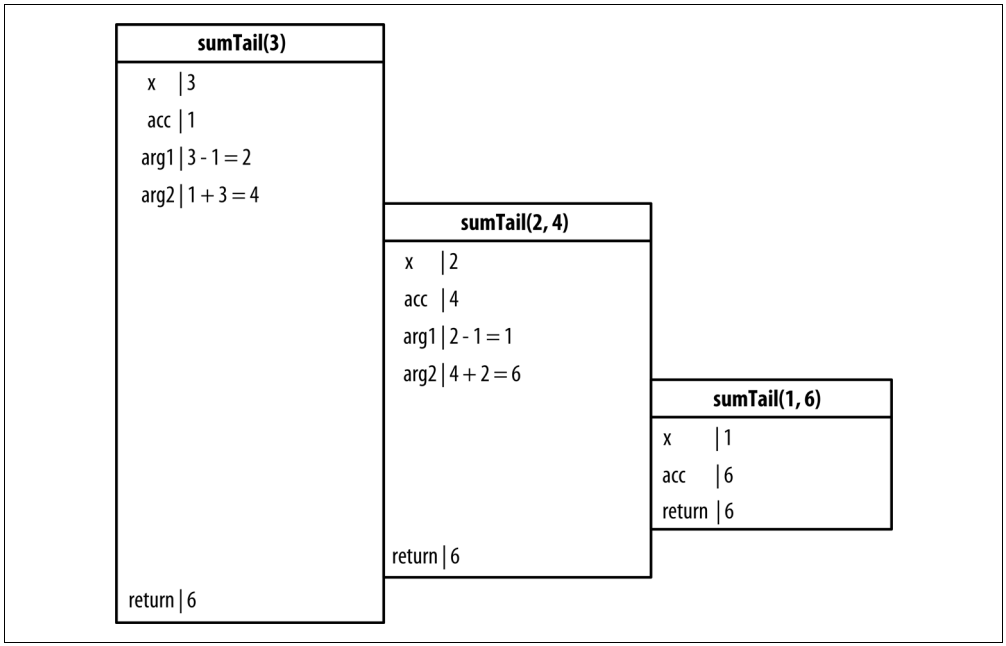


Abbildung 5-3: Der Stack beim Aufruf von `sumTail(3)`*, vgl. Abbildung 5-2. Die Funktion* `sumTail` *kann mit zwei Argumenten aufgerufen werden: der Zahl, deren Summe berechnet wird, und der Akkumulator, der die bisher berechnete Summe enthält. Der Akkumulator wird, wenn er nicht übergeben wird, mit 1 initialisiert. Alle lokalen Variablen stehen oben in der jeweiligen Box, da sie vor dem rekursiven Aufruf bekannt sind.*

Unsere tail-rekursive Summenfunktion hat als letzte Anweisung nur den rekursiven Aufruf von sich selbst. Nachdem dieser beendet ist, muss nur noch zum vorherigen Call Frame zurückgesprungen werden. Der aktuelle Call Frame kann wiederbenutzt werden, nur die Argumente müssen neu belegt werden. Der Rücksprung des rekursiven Aufrufs erfolgt an die gleiche Stelle wie der aktuelle Aufruf.

Ein Compiler oder Interpreter kann tail-rekursive Funktionen so optimieren, dass statt eines neuen Funktionsaufrufs die entsprechenden Argumente im letzten Call Frame neu belegt werden und dann an den Anfang der Funktion gesprungen wird.

Man kann sich diese Optimierung wie die Veränderung von Schleifenvariablen vorstellen. Jede tail-rekursive Funktion kann also direkt in eine Schleife umgewandelt werden!

Ausflug in die Realität der Tail-Call-Optimierung

Die Umgebung *node.js* optimiert tail-rekursive Aufrufe nicht, da in JavaScript vorwiegend imperativ statt funktional programmiert wird. Außerdem hat die Umgebung das Ziel, möglichst schnell zu sein, und zusätzliche Überprüfungen, etwa ob eine Funktion tail-rekursiv ist, erfordern mehr Rechenzeit.

Wenn wir unsere tail-rekursive Summenfunktion mit großen Zahlen aufrufen, bekommen wir einen Fehler, weil der Stack zu groß wird:

```
sumTail(100000);
> RangeError: Maximum call stack size exceeded
```

Wir müssen bisher die tail-rekursive Optimierung in JavaScript selbst durchführen. Eine beliebte Strategie ist, ein sogenanntes Trampolin zu benutzen. Ein Trampolin ist eine Schleife, die Aufrufe nacheinander ausführt. Anstatt dass eine Funktion sich selbst rekursiv aufruft, gibt sie den rekursiven Aufruf zurück, und das Trampolin ruft die Funktion auf.

```
function trampoline (pc) {
  while (typeof(pc) === 'function') {
    pc = pc();
  }
  return pc;
}
```

Wir schreiben unser factorialTail so um, dass es eine Closure zurückgibt, um die Auswertung zu verzögern:

```
function sumTail (x, acc) {
  var acc = acc || 1;
  if (x === 1) return acc;
  else return (function () { return sumTail(x - 1, acc + x); });
}
```

Unser Aufruf erfolgt nun über das Trampolin:

```
trampoline(function () { return sumTail(100000); });
> 5000050000
```

Mit Hilfe unseres Trampolins können wir also die Summe der ersten 100000 natürlichen Zahlen berechnen.

Alle JavaScript-Implementierungen haben ein solches Trampolin bereits eingebaut, die Event-Queue, die wir uns in Kapitel 6 näher anschauen.

– Fortsetzung –

Eine Alternative zum Trampolin ist die Umwandlung der rekursiven Funktion in eine Schleife. Diese Umwandlung ist generisch implementiert, beispielweise in `tailopt-js` (Link: *http://glat.info/pub/tailopt-js/)*. Mithilfe dieser Umwandlung können wir tail-rekursive JavaScript-Methoden schreiben und diese vor der Ausführung automatisch in Schleifen umwandeln lassen.

Auch der in Entwicklung befindliche neue JavaScript-Standard *ECMAScript 6 / Harmony* spezifiziert, dass tail-rekursive Optimierungen implementiert werden müssen. (Link: *http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls)*.

Im nächsten Kapitel lernen wir den Continuation-Passing-Style kennen, in dem Funktionen nicht zurückkehren, sondern direkt die nächste Funktion aufrufen. In Scheme programmiert man in diesem Stil. Gambit Scheme (Link: *http://dynamo.iro.umontreal.ca/wiki/index.php/Main_Page)*, das auch JavaScript als Zielsprache hat, probierte verschiedene Trampolin-Strategien aus, um eine möglichst hohe Geschwindigkeit zu erzielen. Weiterlesen kann man dazu in »Efficient Compilation of Tail Calls and Continuations to JavaScript« (*http://users-cs.au.dk/danvy/sfp12/papers/thivierge-feeley-paper-sfp12.pdf)*.

Viele rekursive Funktionen in der funktionalen Programmierung lassen sich tail-rekursiv implementieren, wie die Funktionen wie `map` und `reduce` aus Kapitel 4.

Rekursion über die Ein- und Ausgabedaten

Die Fibonacci-Funktion haben wir rekursiv berechnnet, weil sie sich in der Mathematik am einfachsten durch eine rekursive Definition beschreiben lässt. Die rekursive Berechnung ist aber auch dann eine gute Idee, wenn die Ein- oder Ausgabedaten einer Funktion selbst rekursiv aufgebaut sind. Indem die Rekursion sich an der Struktur des rekursiven Datentyps orientiert, wird ein solcher Eingabetyp verarbeitet oder ein Ausgabetyp aufgebaut. Beim rekursiven Durchlauf können die Eingabedaten verändert oder zu einer Gesamtlösung kombiniert werden. Oft sagt man auch, die rekursive Funktion akzeptiert oder parsed die Eingabedaten. Diese Sichtweise kommt aus der theoretischen Informatik, vom Konzept der Grammatik, die durch Regeln eine Sprache *generiert*, und dem dazugehörigen Parser, der die von der Grammatik erzeugten Sprache *akzeptiert*. In diesem Fall ist die »Sprache« ein (oder mehrere) rekursiv aufgebauter Datentyp einer bestimmten Form. Genauso kann die Funktion den Ausgabe-Datentyp auf rekursive Weise erzeugen. Dies entspricht dem Konzept der Grammatik als Regelwerk, das eine Sprache erzeugt.

Die Auffassung von Funktionen oder ganzen Programmen als Parser und Generatoren, die in Form einer zum Parser gehörigen Grammatik abstrakt formuliert werden, ist ein mächtiges Konzept in der funktionalen Programmierung. Mit Hilfe bestimmter Funktionen höherer Ordnung, sogenannten Parser-Kombinatoren, können aus einfachen Parsern, die nur einen einzelnen Buchstaben einlesen, komplexere Parser aufgebaut werden, die genau eine bestimmte Zeichenfolge akzeptieren und daraus eine Menge von Lösungen kombinieren.

Auf diese Weise kann ein komplexes Problem mithilfe der passenden Kombinatoren und einer beschreibenden Grammatik elegant formuliert und gelöst werden. Wir sehen ein Beispiel für eine solche Sprache in Kapitel 7 im Abschnitt »Partielle Auswertung« auf Seite 124, wenn wir über das Lambda-Kalkül sprechen. Doch zunächst wenden wir uns wieder der Rekursion auf Datentypen zu. Wir beginnen mit einem einfachen rekursiven Datentyp, der verketteten Liste.

Strukturelle Rekursion auf Listen

Angenommen, wir möchten eine verkettete Liste in JavaScript konstruieren. Moment, wozu brauchen wir Listen, wenn wir in JavaScript Arrays benutzen können? Listen sind die zentralen Datenstrukturen der funktionalen Programmiersprachen Lisp und Haskell. Wir schauen sie uns aber nicht nur deswegen an, sondern auch, weil sie einfache rekursive Datentypen sind, und deshalb gut zu verstehen. Es lohnt sich, das generelle Konzept einer rekursiven Datenstruktur und der strukturellen Rekursion zu erlernen, denn nach dem gleichen Konzept lassen sich in JavaScript auch komplexere rekursive Datentypen wie Bäume oder sogar unendliche Listen aufbauen und abarbeiten, wie wir sehen werden.

Eine verkettete Liste ist eine rekursiv definierte Datenstruktur, deren Elemente jeweils auf das Nachfolgeelement verweisen. Auf diese Weise könnten wir eine Liste von Arbeitsschritten in unserem Rezept darstellen. Wir beginnen mit der Definition des rekursiven Datentyps `LinkedList`, hier als eine objektorientierte Variante. Der Datentyp `LinkedList` enthält einen Wert `head` für jedes Listenelement und einen Verweis `tail` auf die Restliste.

```
function LinkedList (head, tail) {
  this._head = head; // Wert des Listenelements
  this._tail = tail; // Verweis auf die Restliste -> verkettete Liste
}
```

Dann können wir uns kombinierte Get- und Set-Methoden definieren um die beiden Eigenschaften `head` und `tail` für ein Element der verketteten Liste zu setzen oder abzufragen. Man könnte die Eigenschaften natürlich auch direkt manipulieren, aber wie wir aus der objektorientierten Programmierung wissen, kapselt ein Zugriff über Objekt-Methoden die Eigenschaften des Objektes, und bietet ein stabiles Interface nach außen, welches nicht von der konkreten Implementierung abhängig ist. Diese könnte also bei Bedarf angepasst werden, ohne dass diese Anpassung eine Veränderung der Schnittstelle zu anderen Programmteilen verursacht.

```
LinkedList.prototype.head = function () {
  if (arguments.length === 1)
    this._head = arguments[0];
  else
    return this._head;
}
```

```
LinkedList.prototype.tail = function () {
  if (arguments.length === 1)
    this._tail = arguments[0];
  else
    return this._tail;
}
```

Zum bequemen Erzeugen einer verketteten Liste aus einem Array schreiben wir uns die Hilfsfunktion `createLinkedListFromArray`.

```
function createLinkedListFromArray (xs) {
  if (xs.length === 0)
    return null;
  else
    return new LinkedList(xs[0], createLinkedListFromArray(xs.slice(1)));
}
```

Probieren wir das Ganze aus, indem wir eine Liste von Arbeitsschritten erzeugen und auf ihr mit Hilfe von `next()` entlangspazieren:

```
var list = createLinkedListFromArray(["0. Einkaufen",
                                       "1. Abmessen",
                                       "2. Zusammenrühren",
                                       "3. Kochen",
                                       "4. Abschmecken",
                                       "5. Essen"]);

list.head();
> "0. Einkaufen"
list.tail().head();
> "1. Abmessen"
```

Nun können wir diese Liste zusammengefasst ausgeben:

```
function unlines (list) {
  if (list === null)
    return "";
  else
    return list.head() + "\n" + unlines(list.tail());
}

unlines(list);
> "0. Einkaufen
1. Abmessen
2. Zusammenrühren
3. Kochen
4. Abschmecken
5. Essen
"
```

Dies ist ein klassisches Beispiel einer strukturellen Rekursion auf einer Liste. Wir führen eine Funktion auf einem Listenelement aus, und dann kombinieren wir die Lösung mit einem rekursiv berechneten Ergebnis aus der Restliste. Der Basisfall ist entweder die leere Liste, wie zuvor gesehen, oder die Liste, die das letzte Listenelement enthält, falls wir die-

ses gesondert behandeln wollen. Beispielsweise indem wir hinter dem letzten Element keinen Zeilenumbruch anhängen.

```
function unlines (list) {
  if (list.tail() === null)
    return list.head();
  else
    return list.head() + "\n" + unlines(list.tail());
}
```

Nach dem Schema der strukturellen Rekursion können wir nun diverse Funktionen definieren, die auf Listen operieren. Die folgende Funktion berechnet rekursiv die Länge der Liste.

```
function length (head) {
  if (head == null)
    return 0;
  else
    return 1 + length(head.next());
}

length(list);
> 6
```

In der theoretischen Informatik sind Funktionen, die nach diesem Schema aufgebaut sind, in die Klasse der primitiv-rekursiven Funktionen eingeordnet. Primitiv-rekursive Funktionen bestehen aus Projektionen auf Argumente wie etwa `head` oder `tail`, Konstanten, und Komposition dieser Elemente, zum Beispiel durch `+` oder `cons` (Kapitel 4 im Abschnitt »Verallgemeinerung der Funktionen höherer Ordnung auf Arrays« auf Seite 61). Da in der funktionalen Programmierung eine Konstante wie die Zahl `1` als null-stellige Funktion aufgefasst wird, die immer den Wert 1 zurück gibt, sind Konstanten hier auch Funktionen.

Quelle: *http://stackoverflow.com/questions/13323916/javascript-recursive-data-structure-definition*

Listen als algebraische Datentypen, Pattern Matching

Unter Rückgriff auf Sjoerd Visschers Bibliothek für algebraische Datentypen (Link: *http://w3future.com/weblog/stories/2008/06/16/adtinjs.xml* und Bram Steins Bibliothek für Pattern Matching (Link: *http://www.bramstein.com/projects/funcy/)* oder aber auch ihrer Zusammenfassung in (Link: *https://github.com/DrBoolean/Functional-Javascripts)* können wir die Listen-Datentypen auch als algebraische Datentypen modellieren. Ein algebraischer Datentyp ist ein Datentyp, der durch Kombination anderer Datentypen aufgebaut ist. Dieses Konzept wird vor allem in funktionalen Sprachen verwendet.

Man unterscheidet zwischen Produkt-Datentypen, deren Bausteine (Felder) unabhängig nebeneinander stehen, wie Tupeln oder Records, und Summen-Datentypen, wie Mengen oder Listen.

Ein gutes Beispiel für einen Summen-Datentyp ist die einfach verkettete Liste, wie wir sie im letzten Abschnitt benutzt haben. Diese hat zwei Varianten, sichtbar an den Konstruktoren und beim Aufbau der Liste in `createLinkedListFromArray`: die leere Liste `Nil` (null) und die rekursive Liste, die ein Element und eine Restliste bekommt (`LinkedList(value, next)`).

Mit der Bibliothek für algebraische Datentypen können wir den rekursiven Listentyp kurz und praktisch deklarieren:

```
List = Data(function (list, a) { return ({ Nil : {}, Cons: { head: a, tail: list } })});
```

Nun können wir auch schon eine Liste aufbauen. Durch die wiederholte Anwendung von `Cons` erhalten wir die Liste `[0, 1, 2]`, aufgebaut durch `Cons(0, Cons(1, Cons (2, Nil)))`. Unsere Funktion, die eine Liste aus einem Array aufbaut, sieht nun auch eleganter aus:

```
function createLinkedListFromArray (xs) {
  if (xs.length === 0)
    return Nil;
  else
    return Cons(xs[0], createLinkedListFromArray(xs.slice(1)));
}

var xs = [0, 1, 2];
createLinkedListFromArray(xs);
> { head: 0, tail: { head: 1, tail: { head: 2, tail: {} } } }
```

Wenn wir die Liste von Arbeitsschritten aus dem vorherigen Abschnitt erneut zur Hand nehmen, sehen wir, dass bei längeren Listen nur eine bestimmte Anzahl von Listenelementen ausgegeben wird.

```
var list = createLinkedListFromArray(["0. Einkaufen",
                                      "1. Abmessen",
                                      "2. Zusammenrühren",
                                      "3. Kochen",
                                      "4. Abschmecken",
                                      "5. Essen"]);
list;
> { head: '0. Einkaufen', tail: { head: '1. Abmessen', tail: { head: '2. Zusammenrühren',
 tail: [Object] } } }
```

Für unseren Listen-Datentyp ist der Vorteil eines algebraischen Datentyps vor allem die Schreibweise.

Eigentlich interessant ist aber, dass auf solchen Datentypen eine »Evaluation Algebra« (Auswertungs-Algebra) definiert werden kann, daher kommt auch der Name.

Die Algebra wertet die Datentypen je nach Konstruktor-Variante (also `Nil` oder `Cons` auf Listen) mit einer zugehörigen Algebrafunktion aus. Dank der algebraischen Datentypen können wir Daten und Evaluation getrennt betrachten. Um dies zu verstehen, stellen wir uns einen einfachen Taschenrechner mit Zahlen und Operationen vor. In der Datentypdeklaration wird die »Sprache« des Taschenrechners als Datentyp aus Werten und den Operationen plus und minus (etc.) aufgebaut, die alle Ausdrücke umfasst, die der Taschenrechner versteht.

In der Auswertungsfunktion wird dann anhand der Datentypen entschieden, wie die Ausdrücke ausgewertet werden. Bei einem Ausdruck für den Taschenrechner entspricht das der tatsächlichen Berechnung, also z.B. der Entscheidung, ob die Argumente eines Ausdrucks addiert oder subtrahiert werden, und dem Berechnen dieser Operation.

In unserer Funktion `createLinkedListFromArray` haben wir den Basisfall und den rekursiven Fall voneinander unterschieden, indem wir geschaut haben, wie lang das Eingabearray `xs` ist. Für diese Art der Fallunterscheidung anhand der Konstruktoren gibt es in vielen funktionalen Sprachen einen speziellen Mechanismus, das Pattern Matching. In unserem Fall wird auf das leere Array mit dem Basisfall reagiert, und ein Array mit mehr als einem Element wird nach dem rekursiven Schema verarbeitet.

Das Pattern Matching setzt auf dem Konzept der algebraischen Datentypen auf, denn die Fallunterscheidung wird basierend auf den verschiedenen Varianten des Konstruktors getroffen. So kann bei Listen direkt zwischen einer leeren Liste und einer Liste mit mindestens einem Element unterschieden werden, da diese mit verschiedenen Konstruktoren aufgebaut werden.

Programme mit Pattern Matching sind klarer verständlich als Fallunterscheidungen mit `case` oder `if`. Außerdem garantiert ein Matching auf alle Konstruktorvarianten beinahe schon eine korrekte Fallunterscheidung mit vollständigen und disjunkten Mustern, in der weder Fälle doppelt vorkommen, noch Fälle vergessen wurden.

Pattern Matching dieser Art ist in JavaScript mit der erwähnten Bibliothek von Bram Stein Link: *http://www.bramstein.com/projects/funcy/* möglich, jedoch ist diese bisher noch recht experimentell.

Eine zirkuläre Liste

Ausgehend von unserem rekursiv definierten Listen-Datentyp können wir auch eine zirkuläre Liste aufbauen, auf der wir immerwährend im Kreis entlang spazieren können.

Eine zirkuläre Liste ist die Vorstufe eines zirkulären Buffers, der zum Sammeln von Daten bei der asynchronen Kommunikation oder Ein- und Ausgabe benutzt werden kann. Wenn Daten in unregelmäßigen Abständen auflaufen, können diese in einem Ringbuffer zwischengespeichert werden, und werden von neueren Daten später automatisch überschrieben.

Hier benutzen wir Funktionen höherer Ordnung und Array-Tricks, um unsere verkettete zirkuläre Liste aufzubauen. Den rekursiven Listen-Datentyp definieren wir zur Abwechslung als Array. Das Muster ist jedoch dasselbe: eine rekursive Definition mit einem Wert und einem Verweis auf die Restliste.

```
function List (head, tail) {
  return function () { return [head, tail]; };
}
```

Die Funktion head gibt den Kopf der rekursiven Liste zurück, also den Wert des ersten Elements der Argumentliste list.

```
function head (list) {
  return list()[0]; // head
}
```

Die Funktion tail hingegen gibt den Rest der Argumentliste list zurück.

```
function tail (list) {
  return list()[1]; // tail
}
```

Dann benötigen wir noch eine Funktion, mit der wir eine zirkuläre Liste definieren können. Dazu brauchen wir erstmal einen Verweis auf den Kopf der Liste, wir erinnern uns, bei uns hat dieser den Namen head. Diesen Namen benötigen wir, um die Liste zu einem Kreis zu schließen. Dies machen wir, indem wir die Restliste wieder auf den Kopf der Liste zeigen lassen, nachdem wir alle gewünschten Elemente hinzugefügt haben.

Das Hinzufügen der gewünschten Elemente wird von der Funktion returnList übernommen. Diese baut aus dem Array xs von gewünschten Elementen mittels des Listenkonstruktors List und reduceRight eine rekursive Liste auf. Das letzte Element init (da reduceRight ja das Eingabe-Array rückwärts durchläuft) setzen wir auf den Kopf der Liste, this.head. Bei der Fertigstellung der Liste wird this.head auf den Kopf der Liste zeigen. Das bedeutet, der tail des letzten Elements ist wieder das erste Element.

Die Funktion tail liefert also für ein beliebiges Listenelement die richtige Antwort, nämlich die Restliste. Wir können uns demnach durch wiederholtes Aufrufen von tail auf der Liste fortbewegen.

Wir müssen aber auch noch sicherstellen, dass nicht nur tail, sondern auch head für alle Elemente funktioniert. Dazu fügen wir zu der konstruierten Liste, also zu dem Objekt returnList und nicht zu einem seiner Elemente, die neue Eigenschaft head hinzu, deren Wert wir explizit mittels value setzen. Eigenschaften auf Objekten können mittels defineProperty definiert werden. Dabei wird der Name der Eigenschaft und ein Property Descriptor übergeben. Ein Property Descriptor ist ein Hash mit Key-Value-Paaren. Bei uns enthält der Property Descriptor nur einen value für die Eigenschaft head. Den Wert der Eigenschaft head in der in returnList erstellten Liste setzen wir auf circle.apply. Dieses bewirkt, dass der endlose circle erst »ausgerechnet« wird, bevor wir den Wert des Elements mittels head auslesen. Da wir die Liste returnList in einer Funktion verpackt haben, erfolgt davor noch einen Aufruf von apply. Das Verpacken in »unausgerechnete« Funktionen und das Ausrechnen an der richtigen Stelle erlaubt uns den Umgang mit der sich unendlich wiederholenden Liste.

```
var makeCircle = function (xs) {
  var returnList = function () { return myReduceRight(List, this.head, xs); }
  return circle = returnList.apply(Object.defineProperty({}, 'head', {
     'value': function () { return circle.apply(this, arguments); } }
    ));
};
```

```
var lebenskreis = makeCircle(['Kochen', 'Essen', 'Schlafen']);

head(lebenskreis);
> "Kochen"
head(tail(lebenskreis));
> "Essen"
head(tail(tail(lebenskreis)));
> "Schlafen"
head(tail(tail(tail(lebenskreis))));
> "Kochen"
head(tail(tail(tail(tail(lebenskreis)))));
> "Essen"
```

Wie wir sehen, besteht der Kreis des Lebens für uns aus einem immerwährenden Zyklus von Kochen, Essen und Schlafen.

Auch auf dieser rekursiven Datenstruktur können wir mittels struktureller Rekursion entlanglaufen.

```
function taten (l, n) {
  if (n === 0) return "";
  else return head(l) + taten(tail(l), n - 1);
}
function tag () {
  return taten(lebenskreis, 3);
}
function woche () {
  return taten(lebenskreis, 3 * 7);
}

tag();
> "KochenEssenSchlafen"
woche();
> "KochenEssenSchlafenKochenEssenSchlafenKochenEssenSchlafenKochenEssenSchlafenKochen
EssenSchlafenKochenEssenSchlafenKochenEssenSchlafen"
```

Strukturelle Rekursion auf Bäumen

Eine andere häufig auftretende Datenstruktur ist die Baumstruktur, zum Beispiel als Struktur einer Webseite im Document-Object-Model.

Einen rekursiven Baum-Datentyp können wir auf eine ähnliche Weise wie bei den Listen gesehen rekursiv definieren. Er wird mit Hilfe eines Knoten-Datentyps konstruiert, welcher auf eine Liste von Kindknoten verweist, die selber wiederum Wurzelknoten von Unterbäumen sind.

```
function Tree (a, n) {
  this._children = a; // Liste von Kind-Unterbaeumen
  this._node = n; // Wert des Knotens
}
```

```
Tree.prototype.children = function () { //get und set children
  if (arguments.length !== 0)
    this._children = arguments;
  else
    return this._children;
}

Tree.prototype.node = function () { // get und set node
  if (arguments.length === 1)
    this._node = arguments[0];
  else
    return this._node;
}

var t = new Tree([new Tree(null, "A"), new Tree(null, "B")], "C");

t.children();
> [ { _children: null, _node: 'A' },
  { _children: null, _node: 'B' } ]

t.children()[0].node();
> "A"
```

Für einen Baum-Datentyp müssen wir bei strukturellen Rekursion folgende Fälle unterscheiden:

- Basisfall 1: Das Problem wird für den leeren Baum, oft Nil genannt, gelöst.
- Basisfall 2: Das Problem wird für ein Blatt gelöst.
- Rekursiver Fall: Das Problem wird für einen inneren Knoten gelöst. Dabei wird das Problem zunächst rekursiv für die Unterbäume gelöst, und dann zu einer Gesamtlösung kombiniert.

Nach diesem Schema können wir die Anzahl der Blätter eines Baumes berechnen:

```
function countLeaves (t) {
  // Basisfälle
  if (t === null || t.children() === null)
    return 1;
  // Rekursiver Fall: Summe aller Kindknoten
  else
    return shortsum(mymap(countLeaves, t.children()));
}

countLeaves(t);
> 2
```

Die Größe (`size`) eines Baumes ist die Anzahl aller Knoten des Baumes und kann nun leicht von der Leserin in einer Funktion berechnet werden. :)

Die Tiefe eines Baumes ist die Länge des längsten Pfades von der Wurzel bis zu einem Blatt. Um über alle bisher berechneten Pfade zu maximieren, müssen wir über die rekursiven Lösungen aller Kindknoten maximieren. Wir brauchen also eine Funktion, die das maximale Element eines Arrays liefert. Wir können dafür die in JavaScript vordefinierte Methode `max` aus `Math` benutzen. Da diese statisch ist, sollte man sie immer in Form von `Math.max.apply(null, array)` benutzen, und nicht als Objektmethode.

```
function arrayMax (xs) {
  return Math.max.apply(null, xs);
}
```

Damit können wir rekursiv die Tiefe ausrechnen. Für jede neue Verzweigung maximieren wir über die rekursiven Lösungen und addieren `1` hinzu.

```
function depth (t) {
  // Basisfälle
  if (t === null || t.children() === null)
    return 0;
  // Rekursiver Fall
  else
    // 1 plus Maximum über alle rekursiv berechneten Lösungen
    return 1 + arrayMax(mymap(depth, t.children()));
}

depth(t);
> 1
```

Mithilfe von Daten in Listen- und Baumform können wir viele klassische Programmierprobleme lösen. Eine Abwandlung der hier beschriebenen Baumstruktur mit nur zwei Kindern an jedem Knoten kann zum Aufbau eines binären Suchbaumes benutzt werden.

Ein allgemeines Rekursionsschema

Nach dem Vorbild der strukturellen Rekursion auf Listen und auf Bäumen kann man das Muster der strukturellen Rekursion auf beliebige rekursive Datentypen erweitern. Das jeweilige Schema orientiert sich dabei an der Struktur des Datentyps und durchläuft (traversiert) diese.

Es lohnt sich bei komplexeren Datenstrukturen auch manchmal das Durchlaufen der Struktur in einer eigenen Funktion zu definieren. So muss man für bestimmte Probleme nur überlegen, wie sie für einen Knoten gelöst werden, und durch die Kombination mit der Traversier-Funktion bekommt man die Gesamtlösung. In diesem Sinne könnte man zum Beispiel eine Funktion `map` für Bäume schreiben.

Eine andere Möglichkeit ist eine generalisierte Traversierfunktion für verschiedene Arten von Daten, mehr dazu in Kapitel 8 im Abschnitt »Polymorphismus« auf Seite 137.

Neben der strukturellen Rekursion, die von der Klasse der primitiv-rekursiven Funktionen gelöst wird, gibt es weitere Klassen rekursiver Funktionen. Abstrakter als die

strukturelle Rekursion ist die wohlfundierte Rekursion, die sich nicht an der Datenstruktur entlanghangelt, sondern nur verlangt, dass ein gestelltes Problem in jedem Schritt verkleinert wird (vgl. *Divide and Conquer* vom Anfang dieses Kapitels).

Es gibt auch noch weitere Tricks, wie die Verstärkung der Rekursion, bei der ein allgemeineres (verstärktes) Problem gelöst wird, so dass dadurch die eigentliche Problemlösung als Teilfall mitgelöst wird. So kann es bei einer rekursiven Berechnung auf Bäumen ratsam sein, das Problem gleich für Wälder, also Sequenzen von Bäumen, zu lösen. Dies muss man ohnehin tun, wenn man das Problem rekursiv für die Kindknoten lösen möchte.

Ein Baum kann als Spezialfall eines Waldes gesehen werden, da ein Wald als Sequenz von Bäumen definiert ist. Hat diese Sequenz die Länge 1, entspricht sie genau einem Baum. Die Lösung für einen Baum als Spezialfall eines Waldes ist dann sozusagen geschenkt, ähnlich einem Korollar in der Mathematik, aus dem Lateinischen für Zugabe oder Geschenk.

Obwohl die Rekursion als Konzept zum Verständnis der vorgestellten Lösungswege und der funktionalen Denkweise unbedingt erforderlich ist, werden wir sie im Folgenden nicht immer, und auch nicht immer direkt sichtbar verwenden. Denn wir haben ja bereits alternative Möglichkeiten kennengelernt, um Arbeit auf Datenstrukturen zu verrichten, wie zum Beispiel die mächtigen Funktionen `map` und `reduce`.

Rekursion ist also beim Kochen mit Funktionen eine beliebte und leckere Angelegenheit, aber weitaus weniger typisch als wir es uns vielleicht vorstellen. Ein bisschen ist Rekursion das Chicken Tikka Masala der funktionalen Programmierung – wenn man in Großbritannien »zum Inder« geht, ist Chicken Tikka Masala oft das erste Gericht, was einem in den Kopf kommt, aber in Wirklichkeit gibt es viel mehr zu entdecken.

Tatsächlich ist das Curry aus gegrillten marinierten Hähnchenfleischstücken (Chicken Tikka) in würzigen Tomatensoße eigentlich der englischen Küche zuzurechnen. Der Legende nach entstand es in den 1970er Jahren, als ein englischer Gast zu seinem Hähnchen im indischen Restaurant nach Bratensoße verlangte, worauf der Koch improvisierte. Im Jahr 2001 pries der damalige britische Außenminister Cook das Chicken Tikka Masala als Beispiel für gelungenen Multikulturalismus: »Chicken Tikka Masala ist nun ein wahres britisches Nationalgericht, nicht nur weil es das beliebteste ist, sondern da es perfekt abbildet, wie Großbritannien fremde Einflüsse aufnimmt und anpasst. Chicken Tikka ist ein indisches Gericht. Die Masala-Soße wurde hinzugefügt, um das Bedürfnis der Briten zu befriedigen, ihr Fleisch in Soße serviert zu bekommen. Multikulturalismus als positive Kraft für unsere Wirtschaft und Gesellschaft zu erkennen, wird bedeutende Auswirkungen für das Verständnis unserer Identität als Briten haben.«

Im nächsten Kapitel wenden wir uns der event-basierten Programmierung und einer echten Spezialität zu, den Continuations. Im Gegensatz zur Ablage von Programmschritten auf dem Stack, wie bei der Rekursion, wird hier völlig ohne Stack gearbeitet.

Rezept: Chicken Tikka Masala

Für 4 Personen:

- 675 g mariniertes Stücke Chicken Tikka (aus dem indischen Lebensmittelgeschäft)
- 2 TL Senfsamenöl
- 4 fein gehackte Knoblauchzehen
- 200 g fein gehackte Zwiebeln
- 1 Dose Fleischtomaten
- 1-4 fein gehackte Chilischoten
- 2 TL Korianderblätter, fein gehackt
- 50 mL Sahne
- 1 TL Essig, am besten Malzessig
- 2 TL Tandoori Masala Paste
- 1 TL Tomatenmark
- 1 TL Garam Masala

Öl in einer großen Pfanne oder Wok erhitzen, Knoblauch und Zwiebeln hinzufügen und glasig anbraten. Curry-Paste hinzufügen und gut mischen, eventuell mit Wasser oder Joghurt verdünnen. Die Hähnchenteile etwa zwei Minuten in der Mischung anbraten. Tomaten, Essig, Tomatenmark und Chilis hinzufügen und 5 Minuten köcheln lassen. Bei Bedarf noch Wasser zur Soße geben. Sahne, Garam Masala und Koriander hinzugeben und weiter köcheln lassen.

Mit Naan-Brot oder Curryreis servieren. Vor dem Essen mit frischem Koriander garnieren.

```
(λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
z))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

KAPITEL 6

Event-basierte Programmierung und Continuations

Oft beginnt das Kochen mit einem Rezept, einer Abstraktion der Ausführungsschritte zum Zubereiten einer Mahlzeit. Unser Wort Rezept kommt aus dem Lateinischen *recipere* (aufnehmen). Kochrezepte dürften zu den ältesten Schriftstücken überhaupt gehören, wobei sich die Geschichtswissenschaft noch nicht einig ist, was das älteste Rezept ist. Von einer Tontafel aus dem südlichen Babylonien stammt ein Rezept von etwa 1600 v. Chr. Ein chinesisches Rezept für Fischsalat stammt etwa aus dem Jahr 1330 v. Chr. und ist in dem vermutlich ersten Kochbuch enthalten, dem Yinshan Zhengyao, »Von der richtigen Zubereitung der Speisen und Getränke«.

Ein Rezept beinhaltet sowohl Daten, die Zutaten, als auch ein Programm, die Zubereitung. Diese Zubereitung besteht aus zeitlich voneinander abhängigen Zubereitungsschritten. Erst wenn die Gurken geschnitten sind, können sie in den Salat gegeben werden. Wenn der Reis fertig gekocht ist, wird das Gemüse hinzugegeben. Während der Mürbeteig geht, soll das Gemüse für die Quiche geschnitten werden.

Eine ähnliche zeitliche abhängige Reihenfolge gibt es auch in der Programmierung: Erst wenn der Server Daten geliefert hat, können diese auf dem Client verarbeitet werden. Im Programm müssen wir mit der Verzögerung zwischen der Anfrage des Clients und der Antwort des Servers umgehen. Der naive Ansatz, im Programm blockierend auf die Antwort zu warten, ist nicht wünschenswert, da das Programm mehrere Eingabekanäle haben kann. Und wer will schon ein Programm, das nicht reagiert, weil es auf Daten von einem Server wartet?

In diesem Kapitel stellen wir stellen zuerst das Konzept der event-basierten Programmierung vor. Speziell werden wir uns das im World Wide Web weit verbreitete Protokoll HyperText Transfer Protocol (HTTP) anschauen. Später im Kapitel werden wir dann die Fortsetzung, oder auch Continuation, kennenlernen. Doch zunächst zur event-basierten Programmierung.

Event-basierte Programmierung

Der Umgang mit asynchronen Ereignissen

Ein interaktives Programm reagiert auf Benutzer-Eingaben oder Netzwerkdaten, oder zyklisch nach Ablauf einer bestimmten Zeit. Diese unterschiedlichen Eingaben bezeichnen wir als Ereignisse oder Events. Komplexe Programme benutzen oft mehrere Eingabekanäle gleichzeitig, sie verarbeiten also Benutzereingaben, während sie auf Daten von einem Server warten. Um mit mehreren Eingabekanäle umzugehen gibt es verschiedene Programmierkonzepte.

Threads

Ein weit verbreitetes Konzept, um mehrere Eingabekanäle zu verarbeiten, ist in konventionellen objektorientierten Programmiersprachen Threads. Jeder Thread besitzt einen Stack und führt einen Codeblock, also ein Stück Programm, aus. Für jeden Eingabekanal wird ein separater Thread gestartet, der die ganze Zeit nichts anderes tut, als auf Eingaben über diesen Kanal zu warten. Die Programmierung mit Threads hat den Vorteil, dass der Code für jeden Thread übersichtlich und gekapselt ist. Ein großer Nachteil ist aber, dass Zugriffe auf gemeinsame Daten in den verschiedenen Threads synchronisiert werden müssen. Ansonsten kann es zu Schreib-Konflikten und inkonsistenten Daten kommen, wenn zwei Thread gleichzeitig dieselben Daten verändern. Die Synchronisierung der Threads sollte, damit die Threads möglichst wenig aufeinander warten, fein granular sein. Das Debuggen von Programmen, die mehrere Threads benutzen, ist nicht einfach und erquicklich, da alle Threads berücksichtigt werden müssen, und jeder Thread einen eigenen Stack hat. Außerdem kann jeder Thread auf alle Funktionen und Daten im Programm zugreifen.

Event-Loop

Ein alternatives Programmierkonzept, um mit vielen Eingabekanälen umzugehen, ist die sogenannte asynchrone, oder auch event-basierte Programmierung. Hier besteht das Hauptprogramm aus einer Schleife, dem sogenannten Event-Loop. Dieses schaut immer wieder nacheinander in allen Eingabekanälen, ob ein Event, etwa eine Eingabe oder ein Datenpaket angekommen ist. Jedes eintreffende Event verursacht den Aufruf seiner zugehörigen Callback-Funktion. Die event-basierte Programmierung hat den Vorteil, dass nur ein Stück Code zu jeder Zeit ausgeführt wird, und dadurch das Debuggen überschaubar ist. Die Umgebung node.js hat schon ein solches Event-Loop. Das Event-Loop funktioniert wie eine Queue, in die mittels `setTimeout(fun, delay)` eine Funktion `fun` zur verzögerten Ausführung eingereiht werden kann.

Probieren wir das Event-Loop einmal aus, indem wir die Variable `c` jede Sekunde inkrementieren:

```
var c = 0;
var count = function () {
  console.log("c ist ", c);
  c = c + 1;
  setTimeout(count, 1000);
};
count();

//Eine Sekunde später
> c ist 0

//Noch eine Sekunde später
> c ist 1
```

Beispiel: Rezept-Scraper

Wir schauen uns nun die event-basierte Programmierung an einem Beispiel Wikibooks zur Kommunikation über das HTTP-Protokoll an. Nehmen wir an, wir wollen in der Sammlung freier Bücher und Texte unter den Kochbüchern nachsehen, welche Rezepte aus der indischen Küche es dort gibt.

Wir benutzen in unserem Code zwei Zusatzbibliotheken, *request* und *cheerio*, die wir zunächst mit dem Node-Paketmanager npm im System installieren müssen (»`npm install -g request ; npm install -g cheerio`«).

Mit *request* wird die HTTP-Kommunikation zwischen Client und Server ein bisschen vereinfacht. Die Bibliothek *cheerio* bietet die Funktionalität, Webseiten als Bäume auf ihrem document object model (DOM), zu durchlaufen und einzelne Elemente zu extrahieren. Das Interface erinnert an die beliebte Bibliothek jQuery.

```
var request = require('request');
var cheerio = require('cheerio');
```

Wir definieren die anonyme Funktion `dataAvailable`, die drei Argumente bekommt – nämlich einen möglichen Fehler, zum Beispiel ein Verbindungsfehler, die Antwort des Servers und den Inhalt der Webseite. Wenn wir eine erfolgreiche Antwort vom Server bekommen haben, verwenden wir *cheerio*, um entsprechende Knoten mit Rezepten herauszufiltern. Die Knoten bearbeiten wir noch ein wenig, um diese in einer lesbaren Form auf die Konsole auszugeben:

```
var dataAvailable = function (error, response, body) {
  if (!error && response.statusCode === 200) {
    var html = cheerio.load(body);
    var links = html("div#mw-pages").find("a");
    links.map(function (x) { return links[x].attribs.title; }).
         forEach(function (x) { console.log(x.split(/\/ /)[1]); });
```

```
  }
};

request(
  {
    uri: 'http://de.wikibooks.org/wiki/Kategorie:Kochbuch/_' +
          encodeURIComponent('Indische_Küche'),
    headers: { 'User-Agent': 'Funktional programmieren lernen mit JavaScript'}
  },
  dataAvailable // Funktion
);
```

Die eigentliche Magie passiert im Aufruf von `request`, wo neben einem Objekt mit URI und User-Agent noch eine Funktion übergeben wird, nämlich die anonyme Funktion `dataAvailable`. In Worten ausgesprochen können wir uns die event-basierte Programmierung also so vorstellen: »Rufe eine Funktion auf und wenn Daten da sind dann mach etwas mit ihnen«. Tatsächlich passt diese Art zu programmieren perfekt zur Internet-Kommunikation, bei der nie ganz sicher ist, wann Daten zur Verfügung stehen, ob ein Server antwortet oder ein Mitbewohner vielleicht sogar den Stecker aus dem DSL-Router gezogen hat.

Die Ausgabe unseres Programms sieht so aus:

```
> Apfel-Chili-Chutney
> Biryani
> Gelbe Currypaste
> Kadala (Kichererbsen Masala)
> Kokosnuss-Chutney
> Kumro Chola
> Okras mit Pfeffer
> Pav Bhaji
> Raita
> Rote Beete mit Kokosnuss
> Xaucuti
```

Beispiel: Event-basiertes Kochen

Am Anfang dieses Kapitels haben wir ein Rezept dadurch charakterisiert, dass es aus Daten und Arbeitsschritten, die zeitlich voneinander abhängig sind, besteht. Wir stellen uns nun den Computer als eine Meisterköchin vor, die ein event-basiertes Programm ausführt. Da sie immer wieder die Eingabekanäle überprüft, kann sie flugs reagieren, wenn der Reistopf überkocht. Die Köchin nimmt unverzüglich den Topfdeckel ab und stellt den Reistopf auf eine kalte Herdplatte. Direkt danach führt die Köchin den vorherigen Arbeitsschritt fort. Jeder Arbeitsschritt erlaubt an bestimmten Stellen eine Unterbrechung, beim Schneiden von Gemüse beispielsweise nach jedem Schnitt. Diese Unterbrechung ist dazu da, um alle aktive Kanäle auf Eingaben zu überprüfen.

Um die Analogie fortzuführen, nehmen wir an, dass der Bruder der Köchin sein erstes eigenes Gericht zubereiten will. Der Bruder ist sehr vergesslich und braucht daher immer wieder Informationen über den nächsten Arbeitsschritt. Glücklicherweise gibt es einen

automatischen Rezept-Service via Telefon, wo bei Anruf das komplette Rezept vorgelesen wird. Hier der Code des Rezept-Service:

```
function rezeptservice () {
  console.log("Die folgenden Zutaten werden benötigt: ",
              "  Zimt",
              "  Kreuzkümmel",
              "  Koriander",
              "  Gewürznelke",
              "  Schwarze Pfefferkörner",
              "  Kokosnussmilch",
              "  Auberginen");
  console.log("Die Gewürze werden in einer Pfanne aromageröstet.");
  console.log("Die Auberginen werden in einer gesonderten Pfanne",
          "bei hoher Hitze angebraten.");
  console.log("Dann mit der Kokosnussmilch und den Gewürzen zusammen bei niedriger",
          "Hitze schmoren lassen, bis das Curry gut eingekocht und servierfertig ist.");
}
```

Dieser Rezept-Service ist schonmal ein guter Anfang, aber leider unpraktisch für den tatsächlichen Gebrauch, bei dem einige Arbeitsschritte zeitkritisch sind. Für den Bruder ist daher das Warten bis das Rezept vollständig vorgelesen wurde keine realistische Option. Der Rezept-Service hat allerdings keine Ahnung, in welchem Arbeitsschritt der Anrufer sich befindet, daher muss immer das komplette Rezept verlesen werden.

Eine Möglichkeit, dieses Problem zu umgehen, ist, dass der Bruder statt des automatisierten Rezept-Services die Meisterköchin anruft, um nach dem nächsten Schritt zu fragen. Sowohl der Bruder als auch die Köchin haben Kenntnis über den aktuellen Zubereitungsschritt des Gerichtes. Die Köchin ist aber beschäftigt und hat nicht immer Zeit und Lust, ihrem Bruder den nächsten Schritt am Telefon zu erklären.

Der Rezept-Service hat ähnliche Nachteile wie HTTP, das Protokoll, welches im World Wide Web benutzt wird. HTTP trägt keinerlei Zustandsinformationen mit sich. Eine beliebte Lösung, um Zustandsinformationen in der Internetkommunikation zu benutzen, ist, eine Sitzung auf dem Server zu eröffnen, in welcher temporäre Informationen über den Client gespeichert werden. Da Server nur begrenzt viel Speicher haben, müssen diese Sitzungen mit einem Verfallsdatum versehen werden. Wenn ein Client sich nach Ablauf seiner Zeit über weitere Schritte informieren will, ist dies nicht mehr möglich, da der Server die Information, in welchem Schritt der Client war, nicht mehr gespeichert hat.

Continuations

Eine alternative Lösung ist, dass der Rezept-Service nur einen Arbeitsschritt vorliest, und anschließend eine Telefonnummer durchgibt. Wenn diese Telefonnummer angerufen wird, wird der nächste Arbeitsschritt vorgelesen, wieder mit einer Telefonnummer im Anschluss daran.

Hier der Code des Rezept-Services als Continuation. Die Methode `rezeptContinuation` wird mit der angerufenen Telefonnummer aufgerufen, ermittelt durch die nicht implementierte Methode `angerufeneNummer`.

```
function rezeptContinuation (nummer, cont) {
  if (nummer == 0) {
      console.log("Die folgenden Zutaten werden benötigt: ",
                  "  Zimt",
                  "  Kreuzkümmel",
                  "  Koriander",
                  "  Gewürznelke",
                  "  Schwarze Pfefferkörner",
                  "  Kokosnussmilch",
                  "  Auberginen");
  } else if (nummer === 1) {
    console.log("Die Gewürze werden in einer Pfanne aromageröstet.");
  } else if (nummer === 2) {
    console.log("Die Auberginen werden in einer gesonderten Pfanne",
             "bei hoher Hitze angebraten.");
  } else if (nummer === 3) {
    console.log("Dann mit der Kokosnussmilch und den Gewürzen zusammen bei niedriger",
             "Hitze schmoren lassen, bis das Curry gut eingekocht",
             "und servierfertig ist.");
  }
  console.log("bitte ", (nummer + 1), " anrufen");
  cont(rezeptContinuation);
}

function warteAufAnruf (cont) {
  cont(angerufeneNummer(), warteAufAnruf);
}
```

Bei diesem Ansatz weiß der Bruder, in welchem Zustand sein Gericht ist, und der Rezept-Service weiß nichts über das konkrete, gerade kochende Gericht, hat aber eine eindeutige Telefonnummer für jeden Schritt und kann anhand dieser genau den gewünschten Arbeitsschritt zurückgeben. Diese Lösung entspricht einer Continuation: statt eines kompletten Programmes wird dieses Programm häppchenweise serviert, und wenn das Häppchen abgearbeitet ist, kann der nächste Schritt ausgeführt werden. Der nächste Schritt wird mit dem kompletten Zustand, also dem Gericht auf dem Herd und dem schon geschnittene Gemüse, kurzum dem gesamten Zustand der Küche, ausgeführt.

Das Konzept der Abstraktion von Ausführungsschritten erschliesst sich nicht so einfach wie die gewohnte Art über Programme zu denken. Gewöhnlich denken wir uns Programme als eine Folge von Ausführungsschritten, die in Unterroutinen springen können und dann wieder zurückkehren und den Stack im Programm weiter abarbeiten. Den Stack haben wir bereits in Kapitel 5 im Abschnitt »Fettnäpfchen bei der Rekursion« auf Seite 69, kennengelernt, als Datenstruktur, die für jedes Programm zur Laufzeit im Speicher vorhanden ist. Informationen über die aktive Funktion im Programm sind im Stack gespeichert, damit das Programm an die richtige Stelle zurückspringen kann, wenn die aktive Funktion ihre Berechnung beendet hat.

Dagegen wirken Continuations fast surreal. Eine Continuation enthält den gesamten Ausführungs-Stack in konkreter Form. Wenn sich der Zustand der Welt zwischen dem Erzeugen und Aufrufen der Continuation verändert hat, ist ihr Zustand auch nach dem Wiederaufruf der Continuation verändert.

Eine Continuation kehrt nie zurück, wir brauchen daher keinerlei impliziten Call Stack, sondern der Call Stack wird explizit als Argument übergeben. Mit diesem Argument wird die Berechnung fortgeführt, sobald die Funktion fertig abgearbeitet ist. Daher auch der Name »Fortsetzung« oder Continuation.

Während der Ausführung eines Programms befinden sich die Funktionsaufrufe auf einem Stack. Das unterste Element ist das Hauptprogramm, und jeder aktive Funktionsaufruf hat einen zugehörigen Call Frame. Wir stapeln also Ausführungsschritte immer mehr übereinander, um sie dann Schritt für Schritt wieder abzuarbeiten, siehe auch Kapitel 5 im Abschnitt »Problem: Verbrauch von Call Frames« auf Seite 71.

Wir können ganze Programme schreiben, die nur aus Continuations bestehen. Dieses Paradigma wird Continuation-Passing-Style genannt. Ein Programm in Continuation-Passing-Style braucht keinen impliziten Call Stack mehr, da es nie zurückkehrt. Ein Vorteil des Continuation-Passing-Styles ist, dass wir eine einzelne Continuation auf einen anderen Rechner übertragen können und die Ausführung dort fortsetzen können. Denn die Continuation enthält den zur weiteren Ausführung notwendigen Zustand. Zur Ausführung brauchen wir auch keinen Stack und können damit Speicherplatz sparen.

Beispiel: Fakultätsfunktion mit Continuations

Wir schauen uns wieder ein Beispiel aus der Mathematik an, die Fakultätsfunktion. Diese berechnet das Produkt der ersten `n` natürlichen Zahlen:

```
function factorial (n) {
  if (n === 1)
    return 1;
  else
    return factorial(n - 1) * n;
}
```

Wir können die Fakultätsfunktion auch als Continuation schreiben. Dazu fügen wir ein weiteres Argument hinzu, nämlich die Continuation `c`. Im Basisfall wird diese Continuation aufgerufen.

```
function factorial_cps (n, c) {
  if (n === 1)
    c(1);
  else
    factorial_cps(n - 1, function (x) { c(n * x); });
}
```

Wir benutzen die Variable `res`, der wir in der übergebenen Continuation das Ergebnis der Fakultätsberechnung zuweisen.

```
var res;
factorial_cps(6, function(x) { res = x; });
res;
> 720
```

Die Funktion `factorial_cps` ist nur teilweise in Continuation-Passing-Style, da die Operatoren `===`, – und * normale Funktionen sind. Wir können auch für diese Funktion Continuations schreiben.

```
function eq_cps (a, b, c) { c(a === b); }
function min_cps (a, b, c) { c(a - b); }
function mult_cps (a, b, c) { c(a * b); }

function factorial_real_cps (n, c) {
  var rec = function (m) {
    factorial_real_cps(m, function (f) { mult_cps(n, f, c); });
  };
  var cond = function (b) {
    if (b)
      c(1);
    else
      min_cps(n, 1, rec);
  };
  eq_cps(n, 1, cond);
}
```

In dieser Version können wir nun deutlich sehen, dass in jeder der definierten Funktionen alle Informationen für den weiteren Programmablauf enthalten sind. Wir können uns auch vorstellen, dass jede Funktion hier beliebig komplex werden kann und auf Eingabe warten kann.

Wir haben hier die Fakultätsfunktion manuell in den Continuation-Passing-Style umgeformt. Diese Umformung geschieht nach festgelegten Regeln, so dass sie auch von einem Computer vorgenommen werden kann.

Continuations in der Praxis

Die Programmiersprache Scheme benutzt den Continuation-Passing-Style. Wir haben in Kapitel 5 im Abschnitt »Lösung: Tail-Rekursion« auf Seite 72 eine Optimierung der Tail-Rekursion kennengelernt. Beim Continuation-Passing-Style besteht das gesamte Programm aus tail-calls. Die Optimierung der Tail-Rekursion kann somit beim Continuation-Passing-Style angewendet werden, um den Speicherplatz des Stacks zu minimieren.

Pyramidenförmiger Code, in dem eine Funktion mit einem Callback aufgerufen wird, der mit einem Callback aufgerufen wird, der mit einem Callback aufgerufen wird, und so weiter, ist im Continuation-Passing-Style.

Dieser pyramidenförmige Code tritt immer dann auf, wenn verschiedene Funktionen nacheinander auf Daten angewendet werden, und dabei jede Funktion asynchron arbeitet, und somit nicht das gesamte Programm blockieren soll. Stattdessen wird der Arbeitsschritt, der mit den fertig bearbeitenden Daten erfolgen soll, direkt übergeben. Ein Beispiel hierfür können wir in der Software eurlex.js (https://github.com/lobbyplag/eurlex-js/ – speziell `main` in *https://github.com/lobbyplag/eurlex-js/blob/master/eurlex.js*) finden, die Gesetzestexte der EU aus dem Internet herunterläd und analysiert.

Ein anderer Anwendungsbereich sind mehrseitige Web-Formulare. Statt temporäre Daten auf dem Server zu speichern, kann dem Client ein Programm in Continuation-Passing-Style übermittelt werden, das dem Benutzer alle Eingabemasken nacheinander zeigt, und erst nach dem vollständigen Ausfüllen dieser Formulare die Daten an den Server übermittelt. So kann der Benutzer lokal Änderungen im Formular machen, dabei vor und zurück navigieren, und trotzdem brauchen auf dem Server keinerlei Sitzungsdaten gespeichert zu werden. Wenn nur komplette Datensätze sinnvoll bearbeitet werden können, ist der Einsatz von Programmen im Continuation-Passing-Style hier sinnvoll.

Continuation-Passing-Style ist nur einer von mehreren möglichen Programmierstilen, um den Code, der sich mit asynchroner Kommunikation befasst, gut zu strukturieren. Monaden sind eine alternative Methode, wir besprechen sie in Kapitel 9. Um die »Callback-Hölle« zu vermeiden gibt es auch noch den Programmierstil der auf sogenannten Promises aufbaut, der aktuell wieder vermehrt diskutiert wird. Ein Promise macht ein Versprechen über die Art der Rückgabedaten eines asynchronen Funktionsaufrufs.

Link: *http://blog.jcoglan.com/2011/03/11/promises-are-the-monad-of-asynchronous-programming/*

Dadurch hat man ein versprochenes Objekt in der Hand und kann mit ihm im Programm weiterarbeiten. Aber erst nachdem die Daten durch den asynchronen Aufruf wirklich geliefert wurden, wird der weitere Programmcode tatsächlich ausgeführt (deferral). Callbacks haben gleich mehrere Nachteile. Ähnlich wie goto-Statements machen callbacks den Programmfluss oft unverständlich, sie können als eine Art come-from-Statements (wie in Intercal ;-)) gesehen werden. Callbacks sind außerdem eher imperative Programmelemente: Sie lassen sich nicht kombinieren, da ihre Argumente implizit sind, und die Rückgaben meist nicht weiter verwendet werden. Promises hingegen sind eher eine funktionale Lösung des Callback-Problems (*http://blog.jcoglan.com/2013/03/ 30/callbacks-are-imperative-promises-are-functional-nodes-biggest-missed-opportunity/)*.

Continuations überall

In fast allen JavaScript-Programmen können wir das Konzept der Continuation in Form der `return`-Anweisung finden. Hier ist das Rückkehrziel implizit – mit dem `return`

kehrt die Funktion zu der Umgebung zurück, die sie umschließt und vom Stack wird der vorherige Call Frame geholt. In fast allen Programmiersprachen gibt es diese sogenannten non-local exits, um die Ausführung der gerade aktiven Funktion zu beenden. Auch Exceptions, also Fehlerbehandlungen, können als Continuations aufgefasst werden.

Nun haben wir anonyme Funktionen, Closures und Continuations, kennengelernt. Vielleicht ist es an der Zeit, diese Begriffe ein bisschen zu sortieren. Eine anonyme Funktion ist ein Funktionsausdruck, bei dem der Name weggelassen wurde. Wenn diese Funktion Teile der Umgebung benutzt, wie lokale Variablen, wird dies Closure genannt. Eine Closure ist also eine Funktion, die zusätzlich eine Referenz auf die Umgebung, in der sie erstellt wurde, beinhaltet.

Continuations werden in JavaScript unter Verwendung von Closures implementiert. Eine Closure ist also ein Sprachelement, eine anonyme Funktion mit Umgebung, eine Continuation hingegen ist ein Konzept oder Design Pattern, bei dem das gesamte Restprogramm das noch ausgeführt werden soll übergeben wird.

Continuations sind also eher eine Spezialität, aber wenn wir auf sie achten, sehen wir sie plötzlich überall. In letzter Zeit gibt es bei türkischen Gemüsehändlern immer öfter die früher seltenen Okraschoten zu kaufen. Okras sind vielleicht ein bisschen gewöhnungsbedürftig, wie Continuations, – manche Menschen mögen ihre leicht schleimige Konsistenz nicht – aber wenn man sich davon nicht abschrecken lässt, ergeben sie ein herrliches vegetarisches Curry-Hauptgericht.

Im nächsten Kapitel geht es weniger speziell zu, was die Zutaten angeht: Wir wenden uns dem Grundnahrungsmittel der funktionalen Programmierung zu, dem Lambda-Kalkül.

Rezept: Bhindi Masala – Okra-Curry

Für 2-3 Personen.

- 200g Okra (Bhindi)
- 1 1/2 TL Öl
- 1/2 TL Kumin
- 1 TL Korianderpulver
- 1 TL Cayennepfeffer
- 1/2 TL Kurkuma
- 1/2 TL Salz
- Rote und grüne Paprikaschoten, fein zerteilt.

– Fortsetzung –

Die Okra waschen und abtrocknen. Feuchte Okra lässt sich schlecht schneiden. Anschließend in zwei Teile schneiden und beiseite legen.

Öl in einer Pfanne erhitzen. Mit einem Kuminsamen testen, ob das Öl heiss genug ist: der Kuminsamen sollte zerplatzen. Kumin vollständig in die Pfanne geben, mischen und die Okra dazu geben. Etwa eine Minute gut umrühren und anbraten. Abgedeckt etwa 3 Minuten kochen lassen.

Korianderpulver, Cayennepfeffer und Kurkuma hinzugeben.

Wenn die Okra weich ist, mit Salz abschmecken und die Paprikaschoten hinzugeben. Noch ein bisschen schmoren lassen und servieren.

```
(λ z → z (λ s → λ z → z) (λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s z))))))))))))))))))))))))))))))))))) (λ z → z (λ s → λ z → s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s
z))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

KAPITEL 7

Vom Lambda-Kalkül und Lammcurry

Reis und Bohnen sind Grundnahrungsmittel in vielen Kulturen. Sie sind ein minimalistisches Essen, machen satt, sind billig, und sind ausreichend um zu überleben. Auch das Lambda-Kalkül ist minimalistisch. Es ist ausreichend, um alle Programme zu programmieren, aber teilweise ist es schwer zu lesen, da es nur drei Arten von Ausdrücken gibt. Abwechslung beim Essen und beim Programmieren sind eine erfreuliche Bereicherung, und syntaktischer Zucker macht aus einer minimalistischen Kernsprache eine ausdrucksstarke Programmiersprache. Statt komplizierter Kombinationen unserer begrenzten Anzahl von Ausdrücken gibt es Schlüsselworte, die Programmierkonzepte wie Kontrollstrukturen oder Zahlen einführen. Theoretikerinnen lassen diesen Zucker gern weg, da sich die kleinere Kernsprache leichter erforschen lässt. In der Praxis kann man mit Hilfe des syntaktischen Zuckers allerdings kürzere und klarere Programme schreiben. In diesem Kapitel werfen wir einen Blick auf die Grundlagen, wir beschreiben und implementieren das Lambda-Kalkül.

Wie schon in Kapitel 3 beschrieben leben viele Menschen in Indien von Gemüse, Hülsenfrüchten und Reis. Die meisten indischen Gerichte bestehen aus nicht viel mehr. Aber gerade mit den richtigen Hülsenfrüchten, etwa den schwarzen Linsen, die in indischen Läden bei uns immer öfter unter dem Namen Urad Dal zu finden sind, lassen sich leckere Sattmacher kochen. Auch das Lambda-Kalkül hat nur wenige Grundzutaten. Dabei ist es dennoch ausreichend um alle Programme zu programmieren, mit dem Nachteil, dass es schnell schwer lesbar wird. Würden wir einfach alles vorhandene Gemüse in einen Topf schmeißen, käme nie ein schmackhaftes indisches Gericht heraus. Wir müssen also auswählen, welche Gemüse wir verwenden und welche Gewürze essentiell sind. Auch beim Lambda-Kalkül ist es wichtig die richtigen Gewürze zu finden, damit es mächtig und lecker genug wird um satt zu machen.

Rezept: Schwarze Butterlinsen (Urad Dal)

Für zwei Personen

- 150 g ganze, schwarze Linsen (Urad Dal)
- 1 Stück frischer Ingwer (etwa 5 cm)
- 2 Knoblauchzehen
- 1/2 TL Chilipulver
- 400 g passierte Tomaten (aus dem Tetrapack)
- 2-3 EL Butter
- 80 g Sahne
- 1 EL getrocknete Bockshornkleeblätter
- 1 1/2 TL Garam Masala
- Salz

Die Linsen in einer Schüssel mit reichlich kaltem Wasser mindestens 6 Stunden einweichen, besser über Nacht. Dann die Linsen in einem Sieb gut kalt abbrausen. Das Wasser sollte kaum mehr dunkel gefärbt sein. Ingwer und Knoblauch schälen und fein hacken. Beides mit Linsen, Chilipulver und 450 ml Wasser in einen großen Topf geben. Alles kurz aufkochen, dann 25-30 Minuten bei kleiner bis mittlerer Hitze zugedeckt köcheln lassen, bis die Linsen langsam weich werden und aufzuplatzen beginnen. Linsen in ein Sieb abgießen, dabei das Kochwasser auffangen. Dann die Linsen wieder zurück in den Topf schütten und Tomatenpüree, Butter und Sahne unterrühren und mit Bockshornkleeblättern, Garam Masala und Salz würzen. Den Deckel auflegen und die Linsen bei kleiner Hitze 1 ½ bis 2 Stunden ganz sanft köcheln lassen. Immer mal wieder nachschauen und umrühren. Sollte zu viel Flüssigkeit verkochen, etwas vom Linsenkochwasser angießen. Am Ende sollten die Linsen dicksämig und die Sauce dunkelrot sein.

Das Lambda-Kalkül ist eine minimalistische Sprache, die in den 1930ern Jahren entwickelt wurde, also komplett ohne Computer, mit Papier und Bleistift. Es kennt nur drei Arten von Ausdrücken: die Funktionsabstraktion, die Funktionsanwendung, und das Referenzieren einer Variablen.

Im Verlauf des Kapitels werden wir das Lambda-Kalkül als Programmiersprache in JavaScript implementieren. Das Lambda-Kalkül hilft uns, JavaScript und andere Programmiersprachen zu verstehen. Wir gehen zunächst auf die drei verschiedenen Arten von Ausdrücken ein, und dann auf die Auswertung der Ausdrücke. Später im Kapitel schauen wir weiter über den Tellerrand hinaus, indem wir uns verschiedenen Auswertungsstrategien zuwenden. Danach erweitern wir das Lambda-Kalkül um Kontrollstrukturen und die Kodierung von Daten. Zum Schluss des Kapitels entwickeln wir noch einen Parser für das Lambda-Kalkül.

Ausdrücke im Lambda-Kalkül

Im Lambda-Kalkül gibt es nur drei Arten von Ausdrücken. Die erste Art von Ausdruck, die *Funktionsabstraktion*, wird auch λ (Lambda) genannt, und entspricht einer anonymen Funktion mit genau einem Argument. Wir brauchen uns also im folgenden nicht zu erschrecken, wenn uns ein λ im Text begegnet. Es kann einfach als eine anonyme Funktion gelesen werden, wie wir sie schon aus JavaScript kennen. Die Beschränkung auf nur ein Argument ist auch kein Problem, denn Funktionen mit `n` Argumenten können wir durch Curryfizierung, wie in Kapitel 3 im Abschnitt »Curryfizierung und teilweise Anwendung von Funktionen« auf Seite 34 gezeigt, als `n` Funktionen mit jeweils einem Argument ausdrücken!

Die einfachste Funktion, die wir uns vorstellen können, macht sozusagen gar nichts. Sie ist die Identitätsfunktion (Kapitel 4 im Abschnitt »Verallgemeinerung der Funktionen höherer Ordnung auf Arrays« auf Seite 61), die das Argument, dass sie bekommt, unverändert zurückgibt. In der Schreibweise des Lambda-Kalküls notieren wir sie als: λ x → x
In JavaScript können wir sie wie folgt schreiben:

```
function (x) { return x; }
```

In der Schreibweise des Lambda-Kalküls verwenden wir also λ statt des Schlüsselwortes `function` in JavaScript. Danach folgt der Name des Argumentes. Der Pfeil → trennt das Argument vom Funktionsrumpf, welcher wiederum ein Ausdruck ist. In JavaScript wird der Funktionsrumpf von geschweiften Klammern umschlossen, und auf diese Weise vom Argument getrennt. Im Lambda-Kalkül wird der Wert des Funktionsrumpfes automatisch zurückgegeben, wir brauchen dort anders als in JavaScript kein `return`.

Der Ausdruck im Funktionsrumpf der Identitätsfunktion ist die Verwendung der Variable `x`, man sagt dazu *Variablenreferenz*. Die Variablenreferenz ist die zweite Art von Ausdrücken im Lambda-Kalkül. Referenzen können entweder auf gebundene, also schon belegte, oder auf freien Variablen verweisen. Hier zeigt die Referenz auf die gebundenen Variable `x`, die im Kopf der Funktionsabstraktion, also links vom Pfeil, durch λ x gebunden wurde.

Die dritte mögliche Art von Ausdruck im Lambda-Kalkül ist die *Funktionsanwendung*, die in JavaScript und anderen Programmiersprachen einem Funktionsaufruf entspricht. Ein Beispiel ist (λ x → x) y, die Identitätsfunktion auf y angewendet wird. Anders als in JavaScript benutzen wir keine Klammern, um die Argumente vom Funktionsnamen zu trennen. Stattdessen schreiben wir die Funktion und ihr Argument einfach hintereinander. Wir müssen Klammern einfügen, wenn nicht eindeutig ist, welches Argument zu welcher Funktion gehört. Dem Ausdruck λ x → x y entspricht mit explizit hingeschriebenen Klammern (λ x → (x y)). Die Funktionsanwendung bindet also stärker als die Funktionsabstraktion, und muss daher nicht extra eingeklammert werden. Die Funktionsanwendung x y x wäre mit expliziter Klammerung (x y) x. Zuerst wird die linke Funktionsanwendung x y ausgewertet. Wir sagen auch, dass die Funktionsanwendung nach links bindet oder auch linksassoziativ ist: Die am weitesten links stehende Funk-

tionsanwendung wird zuerst ausgewertet. In diesem Kapitel schreiben wir die Klammern in Beispielen explizit hin, um die Lesbarkeit zu verbessern.

Klammerregeln für Ausdrücke

Um die Ausgabe lesbar und eindeutig zu machen, benutzen wir folgende Klammerregeln:

- Eine Funktionsabstraktion erstreckt sich möglichst weit nach rechts, also hat λ x → x x den Funktionsrumpf x x, und nicht nur ein x
- Eine Funktionsanwendung als rechter Ausdruck in einer Funktionsanwendung wird eingeklammert, weil die Funktionsanwendung nach links bindet.

Für jede der drei Arten von Ausdrücken definieren wir in unserer Implementierung des Lambda-Kalküls in JavaScript jeweils eine Konstruktorfunktion und die Eigenschaft `toString` auf den Prototypen.

Die Funktion `toString` gibt die Ausdrücke rekursiv aus und beachtet dabei die gezeigten Klammerregeln.

Lambda-Ausdrücke in JavaScript

Unsere Funktionsabstraktion nennen wir `Lambda`, und zur Konstruktion einer Funktionsabstraktion wird ihr ein Variablenname `binder` und ein Funktionsrumpf `body` übergeben.

```
function Lambda (binder, body) {
  this.binder = binder;
  this.body = body;
}

Lambda.prototype.toString =
  function () {
    return "λ " + this.binder + " → " + this.body.toString();
  };
```

Die Funktionsanwendung nennen wir `Application`, und sie bekommt bei der Konstruktion die linke und die rechte Seite des Ausdrucks, also die Funktion und das Argument übergeben.

```
function Application (left, right) {
  this.left = left;
  this.right = right;
}

Application.prototype.toString =
  function () {
    var l = this.left.toString();
    var r = this.right.toString();
    if (this.left instanceof Lambda)
```

```
    l = "(" + l + ")";
  if (this.right instanceof Lambda || this.right instanceof Application)
    r = "(" + r + ")";
  return l + " " + r;
};
```

Die Variablenreferenz nennen wir `Variable`, und ihr Konstruktor bekommt den Bezeichner der Variable, die sie referenzieren soll, übergeben.

```
function Variable (name) {
  this.name = name;
}

Variable.prototype.toString =
  function () { return this.name; };
```

Nun können wir die Identitätsfunktion in unserem implementierten Lambda-Kalkül definieren:

```
var id = new Lambda("x", new Variable("x"));
id.toString();
> "λ x → x"
```

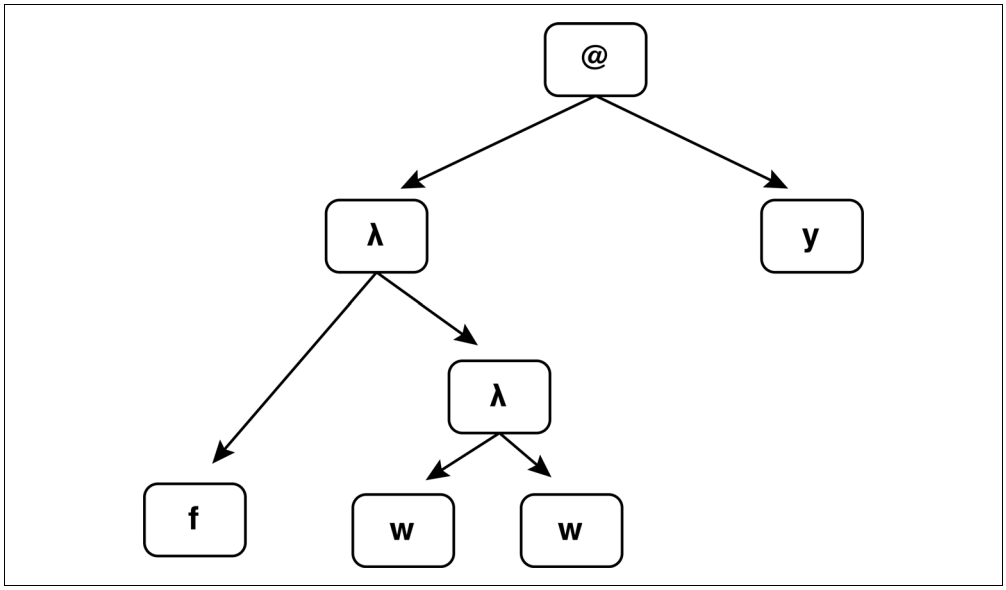


Abbildung 7-1: Abstrakter Syntaxbaum des Ausdrucks `(λ f → (λ w → w)) y` *im Lambda-Kalkül. Hier steht das Symbol* λ *für eine Funktionsabstraktion, dabei steht die Variable, die sie bindet, im linken Kindknoten und der Rumpf im rechten Kindknoten. Das Symbol* @ *bezeichnet eine Funktionsanwendung, die Funktion steht dabei im linken Kindknoten und ihr Argument im rechten Kindknoten.*

Wir können uns einen Ausdruck im Lambda-Kalkül als Baum vorstellen. Dieser Baum wird auch als abstrakter Syntaxbaum (Abstract Syntax Tree, AST) bezeichnet. Eine Variable entspricht einem Blatt. Die rekursiv definierte Funktionsanwendung und Funktionsabstraktion entsprechen im Baum inneren Knoten. Eine Funktionsabstraktion ist ein innerer Knoten mit zwei Kindknoten: der linke beinhaltet die gebundene Variable und der rechte den Funktionsrumpf. Eine Funktionsanwendung hat auch zwei Kindknoten, im linken einen Ausdruck, der auf den Ausdruck im rechten angewendet werden soll. Ein abstrakter Syntaxbaum enthält keine Klammern, da die Struktur des Baumes diese Information repräsentiert. Die Sichtweise als ein Baum ist insbesondere beim Parsen eines Ausdrucks wichtig, dem Vorgang, bei dem der Ausdruck in einen Baum verwandelt wird. An diesem Baum kann man sich dann bei seiner weiteren Verarbeitung, zum Beispiel der Auswertung des Ausdrucks, entlanghangeln.

Einbettung und domänenspezifische Sprachen

Bei der Einbettung des Lambda-Kalküls in JavaScript müssen wir in zwei Ebenen denken: Einerseits in JavaScript, in dem wir Objekte, Funktionen, Prototypen und Variablen zur Verfügung haben, und andererseits im Lambda-Kalkül, das nur λ, Funktionsanwendung und Variablenreferenz kennt. Wir können diese Welten nicht beliebig mischen. Die Variablen, die wir innerhalb des Lambda-Kalküls einführen, sind in JavaScript nicht sichtbar. Und JavaScript-Variablen und Ausdrücke gelten nicht in unserem Lambda-Kalkül. Wir werden JavaScript-Variablen benutzen, um unsere Lambda-Ausdrücke kurz und übersichtlich zu halten, wie wir es schon beim Ausdruck `id.toString()` gesehen haben, in dem die JavaScript-Variable `id` die Repräsentation der Identitätsfunktion im Lambda-Kalkül bindet.

Das Lambda-Kalkül ist ein Beispiel für eine in JavaScript eingebettete domänenspezifische Sprache (domain specific language). Eingebettet bedeutet, dass wir die Programmiersprache Lambda-Kalkül in JavaScript implementieren, und alle Operationen auf JavaScript-Operationen zurückführen. Eine domänenspezifische Sprache ist eine kleine Sprache, die das Lösen von Problemen eines speziellen Themenbereichs, der Anwendungsdomäne, durch passende Ausdrucksmöglichkeiten erleichtert. JavaScript besitzt mit den Regular Expressions schon von Haus aus eine domänenspezifische Sprache im Standardumfang.

Reduktion

Die Auswertung eines Ausdrucks im Lambda-Kalkül, auch Reduktion oder β-Reduktion, erfolgt strukturell rekursiv über alle Ausdrücke anhand von Auswertungsregeln. Wir implementieren die Auswertungsstrategie »applikative Ordnung«, weitere Auswertungsstrategien folgen später im Kapitel. Die Regeln für diese Auswertungs- oder auch Reduktionsstrategie sind:

- Eine Variablenreferenz kann nicht weiter reduziert werden.
- Eine Funktionsabstraktion kann nicht weiter reduziert werden, daher reduzieren wir den Funktionsrumpf (siehe auch Kapitel 7 im Abschnitt »Auswertungsstrategien und Normalformen« auf Seite 112).
- Bei einer Funktionsanwendung wird
  - zuerst die linke Seite und
  - dann die rechte Seite reduziert.
  - Falls die linke Seite eine Funktionsabstraktion ist, werden alle Referenzen zur gebundenen Variable durch die reduzierte rechte Seite ersetzt, und der verbleibende Funktionsrumpf wird weiter ausgewertet.

Reduzieren wir einen Ausdruck, ist das Ergebnis also entweder eine Variablenreferenz, oder eine Funktionsabstraktion, die nicht weiter reduziert werden kann, oder eine Funktionsanwendung, deren linker Ausdruck keine Funktionsabstraktion ist.

Jetzt implementieren wir die Reduktion in unserem Lambda-Kalkül. Dazu erweitern wir die Prototypen der Ausdrücke. Ganz im funktionalen Stil verändern wir keine Objekte, sondern erzeugen stattdessen neue. Dadurch können wir die an JavaScript-Variablen gebundenen Objekte, wie etwa die Identitätsfunktion von oben, mehrfach verwenden.

```
Lambda.prototype.reduce =
  function () { return new Lambda(this.binder, this.body.reduce()); };
Application.prototype.reduce =
  function () {
    var nr = this.right.reduce();
    var nl = this.left.reduce();
    if (nl instanceof Lambda) { // ist nl eine Funktionsabstraktion?
      var r = nl.body.substitute(nl.binder, nr);
      console.log(new Application(nl, nr).toString() + " ⇝ " + r.toString());
      var s = r.reduce();
      return s;
    } else
      return new Application(nl, nr);
  };
Variable.prototype.reduce = function () { return this; };
```

Überprüfen wir, ob die Auswertung der Identitätsfunktion funktioniert. Da diese bereits in ihrer vollständig reduzierten Form definiert ist, sollte keine weitere Reduktion stattfinden.

```
id.reduce().toString();
> "λ x → x"
```

Um es mit dem Autor des »Kleinen Prinzen«, Antoine de Saint-Exupéry, zu sagen: »Perfektion ist nicht dann erreicht, wenn es nicht mehr hinzuzufügen gibt, sondern wenn es nichts mehr gibt, was fortzunehmen ist.«

Substitution

Wie werten wir aus, wenn wir die Identitätsfunktion auf `y` anwenden? Um eine Funktionsanwendung wie `(λ x → x) y` reduzieren zu können, müssen wir das »ersetzen« oder substituieren (`substitute`) von Variablen definieren und implementieren. Die Substitutions-Funktion ist wieder strukturell rekursiv auf den drei verschiedenen Arten von Ausdrücken definiert. Sie bekommt den Bezeichner `o` (original) und den Ausdruck `n` (neu) übergeben und soll im aktuellen Ausdruck jedes auftauchende `o` durch den Ausdruck `n` ersetzen.

In unserem Beispiel `((λ x → x) y)` wird im Substitutionsschritt jedes im Funktionsrumpf auftauchende `x` durch ein `y` ersetzt. Der Funktionsrumpf besteht lediglich aus `x`, und wird somit zu `y`. Auf der Konsole wird jedoch nur der Reduktionsschritt, ohne den Zwischenschritt der Substitution ausgegeben: `(λ x → x) y` ⇝ `y`. Der geschnörkelte Pfeil ⇝ symbolisiert dabei den Reduktionsschritt, und die Ausgabe stammt aus der Funktionsanwendung von `reduce`.

Auf der Variablenreferenz prüft die Substitution, ob wir die zu ersetzende Variable referenzieren. Wenn ja, wird sie ersetzt indem der Ausdruck `n` zurückgegeben wird, ansonsten wird die Variablenreferenz zurückgegeben.

```
Variable.prototype.substitute =
  function (o, n) {
    if (this.name === o)
      return n;
    else
      return this;
  };
```

Auf der Funktionsanwendung delegiert die Ersetzung die Arbeit an die rechte und linke Seite, ersetzt in beiden Seiten rekursiv und gibt eine neue aus den Lösungen für beide Seiten aufgebaute Funktionsanwendung zurück.

```
Application.prototype.substitute =
  function (o, n) {
    return new Application(this.left.substitute(o, n), this.right.substitute(o, n));
  };
```

Bei der Funktionsabstraktion wird die Ersetzung an den Funktionsrumpf delegiert. Wenn die Funktionsabstraktion eine Variable bindet, die im einzusetzenden Ausdruck ebenfalls gebunden wird, gibt es sozusagen einen Namenskonflikt und die Variable wird im Funktionsrumpf verschattet (Kapitel 1 im Abschnitt »Shadowing« auf Seite 6). Im Ausdruck `(λ x → (λ x → (x x))) y` sind die beiden inneren Referenzen zur Variable `x` durch das innere `λ` gebunden, das äußere `x` ist also verschattet. Wenn wir hier die innere Variable umbenennen, erhalten wir den äquivalenten Ausdruck `(λ x → (λ x2 → (x2 x2))) y`.

Die Ersetzungsfunktion prüft außerdem, ob der zu ersetzende Bezeichner gleich dem in der Funktionsabstraktion gebundenen ist. Wenn diese gleich sind, braucht nichts ersetzt zu werden.

Ein ähnlicher Namenskonflikt entsteht, wenn der neue Ausdruck eine Variable referenziert, die in der Funktionsanwendung als Argument auftritt. Im Ausdruck (λ x → (λ y → (x y))) y dürfen wir das x bei der Substitution nicht einfach durch das äußere y ersetzen, da sonst das innere λ das äußere y bindet. Daher benennen wir das innere y in einen neuen, unbenutzten Bezeichner um und führen erst danach die Ersetzung im Funktionsrumpf durch.

In der Implementierung nutzen wir zum Lösen der Namenskonflikte zwei Hilfsfunktionen: die Funktion free, die die Menge der freien Variablen zurückgibt, und die Funktion gensym, die einen frischen Bezeichner erzeugt. Beide implementieren wir im Anschluss.

```
Lambda.prototype.substitute =
  function (o, n) {
    if (this.binder === o) //Verschattung
      return this;
    else if ((this.body.free().indexOf(n.name) !== -1) &&
             (n.free().indexOf(this.binder) !== -1)) { //free ist später implementiert
      //Bezeichner mit gleichem Namen, also zuerst einen frischen erzeugen
      var ns = gensym(); //gensym ist später implementiert
      //und die gebundene Variable zuerst ersetzen
      return new Lambda(ns, this.body.substitute(this.binder,
                                                 new Variable(ns)).substitute(o, n));
    } else
      //Normalfall: Delegation der Ersetzung an den Funktionsrumpf
      return new Lambda(this.binder, this.body.substitute(o, n));
  };
```

Die Funktion gensym benutzt #x und einen Zähler für frische Variablen. Wir selbst dürfen keine Variablennamen der Form #xn benutzen, damit der erzeugte Bezeichner unbenutzt und dadurch eindeutig bleibt.

```
var gensymcount = 0;
function gensym () {
  gensymcount = gensymcount + 1;
  return "#x" + gensymcount;
}
```

Die Menge der freien Variablen ist wieder strukturell rekursiv auf den Ausdrucksarten definiert:

- Bei Referenz einer Variablen x ist x ist frei.
- Die freien Variablen einer Funktionsabstraktion sind die freien Variablen des Funktionsrumpfes ohne die gebundene Variable der Abstraktion.
- Die freien Variablen einer Funktionsanwendung sind die freien Variablen der linken Seite vereinigt mit den freien Variablen der rechten Seite.

Da wir die Menge der freien Variablen als Array darstellen, brauchen wir noch eine Funktion, um ein Element aus dem Array zu entfernen:

```
function remove (ele, arr) {
  return arr.filter(function (a) { return a !== ele; });
}
```

Jetzt können wir die Funktion `free`, die die freien Variablen eines Ausdrucks zurückgibt, auf den Prototypen der drei Arten von Ausdrücken definieren:

```
Variable.prototype.free = function () { return [this.name]; };
Lambda.prototype.free =
  function () { return remove(this.binder, this.body.free()); };
Application.prototype.free =
  function () { return this.left.free().concat(this.right.free()); };
```

Zurück zu unseren drei Beispielen. Im ersten haben wir die Anwendung der Identitätsfunktion auf `y` behandelt.

```
((λ x → x) y)

new Application(id, new Variable("y")).reduce().toString();
> (λ x → x) y ⇝  y
> 'y'
```

Unser zweites Beispiel enthält eine Verschattung:

```
(λ x → (λ x → (x x))) y

new Application(
  new Lambda("x",
    new Lambda("x",
      new Application(new Variable("x"), new Variable("x")))),
  new Variable("y")).reduce().toString();
> (λ x → λ x → x x) y ⇝ λ x → x x
> 'λ x → x x'
```

Wie wir erwartet haben, wurden die Referenzen auf `x` nicht durch `y` ersetzt, da nur das äußere `x` ersetzt wurde, die Referenzen allerdings auf das innere `x` zeigen.

Und unser drittes Beispiel ist eine Funktionsanwendung, bei der ein Namenskonflikt zwischen gebundener und freier Variable entsteht und daher ein neuer, eindeutiger Bezeichner erzeugt werden muss:

```
(λ x → (λ y → (x y))) y

new Application(
  new Lambda("x",
    new Lambda("y",
      new Application(new Variable("x"), new Variable("y")))),
  new Variable("y")).reduce().toString();
> (λ x → λ y → x y) y ⇝ λ #x1 → y #x1
> 'λ #x1 → y #x1'
```

Im Reduktionsschritt wurde das `y` in der inneren Funktionsabstraktion in `#x1` umbenannt. Dadurch gibt es keinen Konflikt mehr zwischen der äußeren ungebundenen Referenz zur Variable `y` und der von der Funktionsabstraktion gebundenen Variable, nun `#x1`.

Wir haben soeben einen Reduzierer oder auch Auswerter für das Lambda-Kalkül implementiert. In den folgenden Abschnitten definieren wir vom Auswerter akzeptierten wohlgeformten Lambda-Ausdrücke und die Gleichheit von Ausdrücken. Im Anschluss daran stellen wir verschiedene Auswertungsstrategien vor. Danach betten wir Datentypen in unser Lambda-Kalkül ein, um es als richtige kleine Programmiersprache nutzen zu können.

Wohlgeformte Ausdrücke

Unser Auswerter erwartet wohlgeformte Ausdrücke – und wir können die Bedingungen für wohlgeformte Ausdrücke induktiv aufschreiben. Wenn ein Ausdruck den folgenden Bedingungen entspricht, ist er wohlgeformt.

- Eine Referenz zu einer Variablen ist ein wohlgeformter Ausdruck.
- Wenn der Ausdruck `e` wohlgeformt ist, und `x` eine Variable, dann ist die Funktionsabstraktion λ `x` → `e` wohlgeformt.
- Wenn der Ausdruck `e` und der Ausdruck `f` wohlgeformt sind, dann ist die Funktionsanwendung `e f` ein wohlgeformter Ausdruck

Der Ausdruck λ λ `x` → `x` ist nicht wohlgeformt, da direkt nach dem ersten λ ein Bezeichner erwartet wird. Erhält unser Auswerter einen nicht wohlgeformten Ausdruck, dann erzeugt er einen Fehler. Wenn wir die obigen Bedingungen bei der Konstruktion von Ausdrücken beachten, gehen wir sicher, einen wohlgeformten Ausdruck zu erzeugen.

Gleichheit von Ausdrücken – Äquivalenz

Zwei Ausdrücke sind intuitiv äquivalent, wenn sie unter allen Umständen das gleiche Ergebnis liefern. Die einfachste Form der Gleichheit liegt vor, wenn die Struktur der Ausdrücke identisch ist. Die Ausdrücke λ `x` → `x` und λ `y` → `y` sind gleich, da die Namen der gebundenen Variablen im Bezug auf das Ergebnis keine Rolle spielen. Diese Form der Gleichheit wird auch α-Äquivalenz genannt.

Es gibt jedoch auch noch andere Klassen der Gleichheit. Es kann beispielsweise in Betracht gezogen werden, ob die Ausdrücke vollständig reduziert sind.

Es lässt sich beweisen, dass die Äquivalenz zweier Lambda-Ausdrücke nicht entscheidbar ist. Bei einfachen Ausdrücken deren Form exakt gleich ist, ist es natürlich trotzdem feststellbar, wie wir eben gesehen haben.

Auswertungsstrategien und Normalformen

Wir unterscheiden zwei Gruppen von Auswertungsstrategien, streng (eifrig, strict) und verzögert (faul, lazy). Der entscheidende Unterschied liegt in der Auswertungsreihenfolge von Funktion und Argumenten. Bei der eifrigen Auswertung werden Argumente vor dem Funktionsrumpf ausgewertet, bei der faulen Auswertung wird zuerst der Funktionsrumpf ausgewertet und die Argumente nur bei Bedarf. »Eifrig« und »faul« werden bei Programmiersprachen weitaus weniger wertend benutzt als vielleicht bei der Leistung der Programmiererin am Arbeitsplatz – es handelt sich einfach um unterschiedliche Strategien, die beide ihre Berechtigung haben.

Lazy Evaluation – die faule Auswertung

Die Lazy Evaluation wertet nur Ausdrücke aus, die tatsächlich benötigt werden. So kann Rechenzeit eingespart werden, und sogar mit problematischen Werten gerechnet werden. Das folgende Programm zeigt diesen Vorteil. Der Wert des Arguments `x` wird nie verwendet, und somit muss die Division durch `0` beim Aufruf nicht ausgewertet werden.

```
function f (x) {
  if (false) x;
  else 10;
}

f(10 / 0);
```

Ein Nachteil der faulen Auswertung ist, dass man nicht immer wissen kann, wann ein Ausdruck ausgewertet ist. Falls die Auswertung Nebenwirkungen hat, ist nicht absehbar, wann diese eintreten.

Schauen wir uns verschiedene Vertreter der faulen Auswertung an.

Die erste Form der faulen Auswertung heißt *normale Ordnung* (normal order). Bei ihr wird von außen nach innen ausgewertet. Hier wird der Funktionsrumpf daher vor den Argumenten ausgewertet.

Die zweite Form der faulen Auswertung ist *call-by-name*. Bei ihr wird zuerst der Funktionsrumpf ausgewertet und dann die Argumente. Bei call-by-name werden dadurch bei einer Funktionsanwendung die noch nicht ausgewerteten Argumente im Funktionsrumpf ersetzt und müssen möglicherweise mehrfach ausgewertet werden. Im Gegensatz zur normal order wird bei call-by-name allerdings ein Funktionsrumpf, der nicht angewendet wird, nicht ausgewertet.

Die Auswertung *call-by-need* ist eine Optimierung von call-by-name. Im Gegensatz zu call-by-name wird ein Argument nur einmal ausgewertet und dieser ausgewertete Ausdruck wird für folgende Verwendungen zwischengespeichert.

Wir können die Auswertung unseres Lambda-Kalküls abändern, so dass es faul auswertet (call-by-name):

```
Lambda.prototype.reduce =
  function () { return this; };

Application.prototype.reduce =
  function () {
    var nl = this.left.reduce(); //reduktion der linken Seite
    if (nl instanceof Lambda) {
      var r = nl.body.substitute(nl.binder, this.right); //Ersetzung des Arguments durch
                                                         //das nicht ausgewertete Argument
      console.log(new Application(nl, this.right).toString() + "⟿" + r.toString
());
      var s = r.reduce();
      return s;
    } else
      return new Application(nl, this.right);
  };

Variable.prototype.reduce = function () { return this; };
```

Die rein funktionale Programmiersprache Haskell benutzt die faule Auswertungsstrategie call-by-need. Viele andere Programmiersprachen wie Java, C# oder auch JavaScript benutzen ebenfalls die faule Auswertung, allerdings nur für logische Ausdrücke bei Tests von bedingten Anweisungen und Schleifen. Dieser Spezialfall wird *short-circuit evaluation* oder Kurzschlussauswertung genannt.

Eager Evaluation – die eifrige Auswertung

Die meisten imperativen Programmiersprachen benutzen eine eifrige oder auch strikte Auswertungsstrategie. Die Reihenfolge der Auswertung ist in diesen Sprachen durch die Reihenfolge der Ausdrücke im Programm festgelegt, daher dürfen Compiler dieser Sprachen die Ausdrücke bei der Optimierung nicht ohne weiteres umordnen.

Die strikte Gruppe von Auswertungsstrategien hat auch mehrere Vertreter.

Bei der *applikativen Ordnung* (applicative order) werden zuerst die Argumente einer Funktionsanwendung von links nach rechts ausgewertet. Genau wie bei der normal order werden hier auch die Funktionsrümpfe reduziert, obwohl die Funktionsabstraktion nicht angewendet wird.

Eine andere strikte Auswertungsstrategie ist *call-by-value*, bei der zuerst die Argumente einer Funktionsanwendung ausgewertet werden, und danach der Funktionsrumpf. Der tatsächliche Wert des Argumentes wird an die Funktion übergeben. Die Programmiersprache C benutzt call-by-value.

Bei *call-by-reference* wird lediglich eine Referenz auf den Wert an die Funktion übergeben.

Viele verbreitete Programmiersprachen, wie JavaScript, C++, Java, Scala, C#, Lisp benutzen *call-by-sharing*, eine Kombination von call-by-value für die Übergabe von primitive Datentypen und call-by-reference für Objekt-Datentypen. Primitive Datentypen sind dadurch direkt verwendbar, und große Objekte werden nicht auf den Stack kopiert.

Wollen wir in unserem Lambda-Kalkül eine call-by-value-Auswertungsstrategie benutzen, sieht diese so aus:

```
Lambda.prototype.reduce =
  function () { return this; };

Application.prototype.reduce =
  function () {
    var nr = this.right.reduce(); //das Argument wird reduziert
    var nl = this.left.reduce(); //die Funktion wird reduziert
    if (nl instanceof Lambda) {
      var r = nl.body.substitute(nl.binder, nr); //das Argument wird ersetzt durch den
                                                 //Wert dessen
      console.log(new Application(nl, nr).toString() + "⇝" + r.toString());
      var s = r.reduce();
      return s;
    } else
      return new Application(nl, nr);
  };

Variable.prototype.reduce = function () { return this; };
```

Die Auswertungsstrategien faul und eifrig können jede bei Bedarf die jeweils andere nachahmen. Das sieht zwar nicht schön aus, ist aber manchmal hilfreich und wurde von John C. Reynolds 1972 in der Veröffentlichung »Definitional interpreters for higher-order programming languages« gezeigt. Die Idee ist, das Programm zuerst in Continuation-Passing-Style (Kapitel 6 im Abschnitt »Continuations« auf Seite 93) umzuformen.

Repräsentation von Daten im Lambda-Kalkül

Was wäre eine Programmiersprache ohne die Möglichkeit, Daten zu repräsentieren? Damit wir ganz puristisch im reinen Lambda-Kalkül bleiben, betten wir die Daten ins Lambda-Kalkül ein. Wir benutzen zur Einbettung nicht weiter reduzierbare Funktionsabstraktionen. Wir drücken in diesem Abschnitt Wahrheitswerte und natürliche Zahlen als Funktionsabstraktionen im Lambda-Kalkül aus.

Wahrheitswerte im Lambda-Kalkül: Church-Booleans

Ein Wahrheitswert kann einen von zwei Werten annehmen, wahr oder falsch. Daher brauchen wir die Möglichkeit, zwei verschiedene Werte zu repräsentieren. Da eine Funktionsabstraktion nur eine mögliche Rückgabe/Belegung hat, nämlich das Argument

selbst, brauchen wir mehr als eine. Wenn wir zwei Funktionsabstraktionen benutzen, haben wir zwei mögliche Rückgabewerte: entweder das erste oder das zweite Argument. Wahrheitswerte, die auf diese Weise kodiert sind, werden auch Church-Booleans genannt, nach ihrem Erfinder, dem Mathematiker Alonzo Church.

```
λ w → (λ f → w) benutzen wir als wahr und
λ w → (λ f → f) benutzen wir als falsch
```

In unserem Lambda-Kalkül sieht das so aus:

```
var wahr = new Lambda("w", new Lambda("f", new Variable("w")));
var falsch = new Lambda("w", new Lambda("f", new Variable("f")));
```

Die Belegung von `wahr` und `falsch` ist willkürlich, wir könnten sie auch umdrehen, dann müssen die nachfolgenden Operationen angepasst werden.

Negation

Im Folgenden definieren wir einige Operationen auf Wahrheitswerten, um mit den Wahrheitswerten arbeiten zu können. Die erste Operation ist die Negation, eine Funktion, die einen Wahrheitswert bekommt und das Gegenteil zurückgibt. Für `falsch` gibt sie `wahr`, und für `wahr` `falsch` zurück. Die Negation bekommt einen Wahrheitswert `p`, dieser ist wie gerade eben definiert eine zweistellige Funktion. Die Negation gibt wiederum eine zweistellige Funktion zurück, in der die Reihenfolge der Argumente vertauscht wurde (vgl. `flip` in Kapitel 4 auf Seite 50). Daher wenden wir zweimal das übergebene Argument `p` an, um die beiden Argumente der zweistelligen Rückgabefunktion zu konstruieren. Zuerst auf eine Funktion die das zweite Argument der zweistelligen Funktion zurückgibt, und dann auf eine Funktion, die das erste Argument der zweistelligen Funktion, zurückgibt. Beide Funktionen haben wir weiter oben schon definiert, es sind die Funktionen `falsch` und `wahr`. :)

```
λ p → p (λ w → (λ f → f)) (λ w → (λ f → w))
```

Oder auch:

```
λ p → p falsch wahr
```

```
var neg =
  new Lambda("p",
    new Application(new Application(new Variable("p"), falsch), wahr));
```

Wir testen die Negation, indem wir sie auf `falsch` anwenden und reduzieren. Wir erwarten `wahr` als Ergebnis.

```
new Application(neg, falsch).reduce().toString();
> (λ p → p (λ w → λ f → f) (λ w → λ f → w)) (λ w → λ f → f) ⇝
    (λ w → λ f → f) (λ w → λ f → f) (λ w → λ f → w)
> (λ w → λ f → f) (λ w → λ f → f) ⇝ λ f → f
> (λ f → f) (λ w → λ f → w) ⇝ λ w → λ f → w
> 'λ w → λ f → w'
```

Im ersten Reduktionsschritt wird das Argument `p`, hier `falsch`, ersetzt durch den Wert `falsch`. Im zweiten Reduktionsschritt wird die erste Funktionsanwendung ausgeführt. Hier wird das übergebene `falsch` auf die erste Funktion, die auch `falsch` ist, angewendet. Das Resultat ist die Identitätsfunktion. Im dritten Schritt wird die zweite Funktionsanwendung ausgeführt, die Identitätsfunktion wird auf `wahr` angewendet. Das Resultat ist `wahr`. Die Negation verhält sich wie gewünscht.

Wir stellen nun noch die logischen Operationen »und« und »oder« vor, die jeweils zwei Wahrheitswerte bekommen und einen zurückgeben. Die Operation »`und`« gibt nur dann `wahr` zurück, wenn beide Argumente `wahr` sind. Die Operation »`oder`« gibt `wahr` zurück, wenn mindestens eines der Argumente `wahr` ist.

Logisches Und

Wir schreiben »`und`« als Wahrheitstabelle: die Belegung des ersten Argument ist in der ersten Spalte dargestellt und die Belegung des zweiten Arguments befindet sich in der ersten Zeile. Das Ergebnis kann dann in der Tabelle abgelesen werden.

`"und"`	`wahr`	`falsch`
`wahr`	`wahr`	`falsch`
`falsch`	`falsch`	`falsch`

`Wahr und wahr` ergeben `wahr`, `wahr und falsch` ergeben `falsch`, `falsch und falsch` ergeben ebenfalls `falsch`.

Die Operation »`und`« bekommt zwei Wahrheitswerte als Argumente. Diese sind in unserer Kodierung zweistellige Funktionsabstraktionen. Der Ausdruck »`und`« mit Argumenten soll zu einem Wahrheitswert reduzieren, der ja eine zweistellige Funktionsabstraktion ist. Um eine zweistellige Funktionsabstraktion aus zwei zweistelligen Funktionsabstraktionen zu erhalten, müssen wir zwei Funktionsanwendungen ausführen.

Um das Ergebnis der Operation »`und`« aus den Argumenten zu konstruieren, wenden wir die beiden Argumente aufeinander an. Entscheidend ist dabei, in welcher Reihenfolge wir die Wahrheitswerte aufeinander anwenden. Unser Ziel ist, `wahr` zurückzugeben, wenn beide Argumente `wahr` sind, ansonsten `falsch`. `Wahr` gibt uns das erste Argument zurück. Im Ergebnis von »`und`« muss also die erste Funktionsanwendung vom ersten Argument `p` auf das zweite Argument `q` erfolgen. Auf das Ergebnis wenden wir in der zweiten Funktionsanwendung dann wieder das erste Argument `p` an.

Da »`und`« kommutativ ist, also `p und q` und `q und p` das gleiche Ergebnis liefern, können wir uns frei entscheiden, welches der Argumente wir zuerst anwenden. Es gibt also zwei mögliche Definitionen der Funktion »`und`« im Lambda-Kalkül.

```
λ p → (λ q → (p q p))
λ p → (λ q → (q p q))
```

Und auch in JavaScript

```
var und =
  new Lambda("p",
    new Lambda("q",
      new Application(
        new Application(new Variable("p"), new Variable("q")),
        new Variable("p"))));
var und2 =
  new Lambda("p",
    new Lambda("q",
      new Application(
        new Application(new Variable("q"), new Variable("p")),
        new Variable("q"))));
```

Wir probieren unser "und" aus:

```
new Application(new Application(und, falsch), falsch).reduce().toString();
> (λ p → λ q → p q p) (λ w → λ f → f) ⟿
    λ q → (λ w → λ f → f) q (λ w → λ f → f)
> (λ w → λ f → f) q ⟿ λ f → f
> (λ f → f) (λ w → λ f → f) ⟿ λ w → λ f → f
> (λ q → λ w → λ f → f) (λ w → λ f → f) ⟿ λ w → λ f → f
> 'λ w → λ f → f'
```

Im ersten Reduktionsschritt wird `falsch` für das Argument `p` eingesetzt. Im zweiten wird der resultierende Funktionsrumpf vereinfacht und die Identitätsfunktion ist das Ergebnis. Im dritten Reduktionsschritt wird die Identitätsfunktion auf `falsch` angewendet, was wiederum `falsch` ergibt. Im letzten Schritt wird für `q` `falsch` eingesetzt. Im Funktionsrumpf taucht `q` aber nicht mehr auf, weshalb das Endergebnis auch `falsch` als Rückgabewert liefert. Unser »`und`« funktioniert also richtig, weil das Ergebnis hier `falsch` ist. :)

Nun probieren wir unser »`und2`« aus, auch wieder auf `falsch` und `falsch`:

```
new Application(new Application(und2, falsch), falsch).reduce().toString();
> (λ p → λ q → q p q) (λ w → λ f → f) ⟿ λ q → q (λ w → λ f → f) q
> (λ q → q (λ w → λ f → f) q) (λ w → λ f → f) ⟿
    (λ w → λ f → f) (λ w → λ f → f) (λ w → λ f → f)
> (λ w → λ f → f) (λ w → λ f → f) ⟿ λ f → f
> (λ f → f) (λ w → λ f → f) ⟿ λ w → λ f → f
> 'λ w → λ f → f'
```

Auch »`und2`« funktioniert wie in der Tabelle. Bei den Reduktionsschritten für »`und2`« wird zuerst `p` eingesetzt und dann `q`. Erst dann wird im inneren Funktionsrumpf weiter reduziert, da die erste Funktionsanwendung das `q` benötigt.

Logisches Oder

Wir schauen uns nun das logische »`oder`« an. Dies hat die Wahrheitstabelle:

"oder"	wahr	falsch
wahr	wahr	wahr
falsch	wahr	falsch

`Wahr oder wahr` ergeben `wahr`, `wahr oder falsch` ergeben `wahr`, `falsch oder falsch` ergeben `falsch`.

Bei der Operation »`oder`« gelten die gleichen Überlegungen wie bei »`und`«: Sie bekommt zwei Wahrheitswerte als Argumente und gibt einen Wahrheitswert zurück, allerdings kommt bei »`oder`« auch dann `wahr` heraus, wenn eins der Argumente `wahr` ist. Wir wenden daher als erstes den ersten Wahrheitswert `p` auf sich selbst an, und den zweiten Wahrheitswert `q` auf das Ergebnis dieser Anwendung. Auch hier haben wir wieder zwei mögliche Definitionen, da `p oder q` und `q oder p` zum gleichen Ergebnis kommen.

```
λ p → (λ q → (p p q))
λ p → (λ q → (q q p))
```

In JavaScript:

```
var oder =
  new Lambda("p",
    new Lambda("q",
      new Application(
        new Application(new Variable("p"), new Variable("p")),
        new Variable("q"))));
var oder2 =
  new Lambda("p",
    new Lambda("q",
      new Application(
        new Application(new Variable("q"), new Variable("q")),
        new Variable("p"))));
```

Bedingte Anweisung

Nun haben wir Wahrheitswerte und die Boolschen Operationen Negation, »`und`« und »`oder`« kennengelernt, die auf Wahrheitswerten operieren. Wir können mit Hilfe der Wahrheitswerte auch die bedingte Anweisung, die bekannte `if`-Bedingung, im Lambda-Kalkül ausdrücken. Die `if`-Anweisung hat drei Ausdrücke als Argumente. Das erste Argument `p` wird auch Test oder Prädikat genannt und reduziert zu einem Wahrheitswert. Wenn dieser `wahr` ist, soll die `if`-Anweisung den Ausdruck im zweiten Argument `c` auswerten, die sogenannte Konsequenz. Ist der Test `falsch`, wird der im dritten Argument `a` übergebene Ausdruck, die Alternative, ausgewertet. Dies erreichen wir, indem das erste Argument `p` auf das zweite und dritte angewendet wird. Wenn `p wahr` ergibt, wird das zweite Argument `c` ausgewertet, da `wahr` per Definition die erste Abstraktion zurückgibt. Wenn `p falsch` ergibt, wird das dritte Argument `a` ausgewertet, da `falsch` per Definition die zweite Abstraktion zurückgibt.

```
λ p → (λ c → (λ a → (p c a)))

var iff =
  new Lambda("p",
    new Lambda("c",
      new Lambda("a",
        new Application(
```

```
            new Application(new Variable("p"), new Variable("c")),
            new Variable("a")))));
```

Wir probieren unsere bedingte Anweisung aus, indem wir `iff (wahr) wahr else falsch` reduzieren, wobei wir als Ergebnis `wahr` erwarten:

```
new Application(
  new Application(
    new Application(iff, wahr), wahr), falsch).reduce().toString();
> 'λ w → λ f → w'
```

Wir haben nun die Wahrheitswerte und auch die bedingte Anweisung kennengelernt und im Lambda-Kalkül implementiert. Doch auf Dauer ist es etwas fade, mit diesem ungewürzten Gemüse der Boolschen Logik zu programmieren. Daher betten wir auch die natürlichen Zahlen in unser Lambda-Kalkül ein.

Zahlen im Lambda-Kalkül: Church-Numerale

Natürliche Zahlen können wir ähnlich wie primitive Rekursion (Kapitel 5 im Abschnitt »Listen als algebraische Datentypen, Pattern Matching« auf Seite 79) definieren: Entweder eine Zahl ist 0, das ist der Basisfall, oder die Zahl ist die Nachfolgerin einer natürlichen Zahl. Diese Art der Definition nennt sich rekursiv. Wenn wir die natürliche Zahlen auf diese Weise auffassen, brauchen wir bei ihrer Kodierung als Funktionsabstraktion im Lambda-Kalkül nur zwei Argumente, ähnlich wie bei den Wahrheitswerten. Gibt die Funktionsabstraktion das erste Argument zurück, repräsentiert sie die 0, gibt sie die zweite zurück, die Nachfolgerin. Alonzo Church hat auch die Kodierung für die natürlichen Zahlen im Lambda-Kalkül erfunden, daher werden sie Church-Numerale genannt.

Wir haben also zwei verschiedene Ausdrücke für Zahlen: `zero`, auch `z` oder 0, und `succ`, auch `s`, die Nachfolgerin. Diese reichen aus, um beliebige natürliche Zahlen darzustellen:

```
0 = zero
1 = succ(zero)
2 = succ(succ(zero))
3 = succ(succ(succ(zero)))
```

Die 0 definieren wir im Lambda-Kalkül analog zu `falsch`, allerdings benutzen wir hier die Bezeichner `s` statt `w` und `z` statt `f`. Die Ausdrücke 0 und `falsch` sind nicht auseinanderzuhalten; sie sind α-äquivalent. Anhand des Kontexts, wenn wir also beispielsweise `succ` auf den Ausdruck anwenden, wissen wir ob 0 oder `falsch` gemeint ist. Das `s` ist unsere Nachfolger-Funktion. Wenn wir `s` `n`-mal auf 0 anwenden, erhalten wir die Zahl `n`.

```
0 : λ s → (λ z → z)
1 : λ s → (λ z → (s z))
2 : λ s → (λ z → (s (s z)))
var zero =
  new Lambda("s", new Lambda("z", new Variable("z")));
var one =
  new Lambda("s", new Lambda("z",
    new Application(new Variable("s"), new Variable("z"))));
var two =
  new Lambda("s", new Lambda("z",
```

```
    new Application(new Variable("s"),
      new Application(new Variable("s"), new Variable("z")))));
```

Auch auf den natürlichen Zahlen schauen wir uns einige Operationen zum praktischen Umgang an, zuerst die Inkrementierung um eins. Die Inkrementierungsfunktion bekommt eine natürliche Zahl, und soll die nachfolgende Zahl zurückgeben. In unserer Implementierung des Lambda-Kalküls muss also `s` angewendet werden.

```
λ n → (λ s → (λ z → (s (n s z))))

var inkrement =
  new Lambda("n",
    new Lambda("s",
      new Lambda("z",
        new Application(
          new Variable("s"),
          new Application(
            new Application(new Variable("n"), new Variable("s")),
            new Variable("z"))))));
```

Wir testen unsere Inkrementierungsfunktion, indem wir sie auf die Zahl 0 anwenden, also 0 um eins erhöhen. Als Ergebnis erwarten wir die Zahl 1:

```
new Application(inkrement, zero).reduce().toString();
> (λ n → λ s → λ z → s (n s z)) (λ s → λ z → z) ⇝
    λ s → λ z → s ((λ s → λ z → z) s z)
> (λ s → λ z → z) s ⇝ λ z → z
> (λ z → z) z ⇝ z
> 'λ s → λ z → s z'
```

Die Additionen von zwei Zahlen `m` und `n` können wir nun auf zwei Wegen definieren. Entweder wir wenden die Inkrement-Funktion `m` mal an:

```
λ m → (λ n → (m inkrement n))

var plusink =
  new Lambda("m",
    new Lambda("n",
      new Application(new Variable("m"),
        new Application(inkrement, new Variable("n")))));
```

Oder wir wenden `s` `m`-mal auf `n` an, die Komposition von `n`-mal `s` mit `m`-mal `s` ergibt `(n + m)` mal `s`:

```
λ m → (λ n → (λ s → (λ z → (m s (n s z)))))

var plus =
  new Lambda("m",
    new Lambda("n",
      new Lambda("s",
        new Lambda("z",
          new Application(
            new Application(new Variable("m"), new Variable("s")),
            new Application(
              new Application(new Variable("n"), new Variable("s")),
              new Variable("z")))))));
```

Die anderen mathematischen Operationen auf natürlichen Zahlen können wir nach dem gleichen Schema definieren.

Um natürliche Zahlen einfacher zwischen ihrer Repräsentation im Lambda-Kalkül und in JavaScript konvertieren zu können, definieren wir zwei Funktionen. Die Funktion `toChurch` wandelt eine JavaScript-Zahl in ein Church-Numeral im Lambda-Kalkül um, und die Funktion `toJS` funktioniert genau in die andere Richtung. Wir definieren `toChurch` rekursiv, und fügen bei jedem seiner Aufrufe eine weitere Funktionsanwendung von `s` hinzu.

```
function toChurch (n) {
  if (n === 0)
    return zero;
  else {
    var rec = toChurch(n - 1);
    //rec ist unser Ausdruck bis (n - 1)
    //wir fuegen ein s hinzu: λ s → λ z → s (rec s) z
    return new Lambda("s",
             new Lambda("z",
               new Application(new Variable("s"),
                 new Application(new Application(rec, new Variable("s")),
                                 new Variable("z"))))).reduce();
  }
}
```

Ein Church-Numeral `n` hat die Form `λ s → (λ z → term)`. Es besteht also aus zwei Funktionsabstraktionen, dem successor `s` und der zero `z`. Im inneren Funktionsrumpf `term` befinden sich `n` Funktionsanwendungen von `s`. Um das Church-Numeral in eine JavaScript-Zahl umzuwandeln, zählen wir die Anzahl der Funktionsanwendungen von `s`. Hierzu verwenden wir den JavaScript-Operator `instanceof` (Kapitel 8 im Abschnitt »Primitive Datentypen versus Objekte« auf Seite 135), der zwei Argumente bekommt, einen zu prüfenden Wert und einen Konstruktor. Er gibt nur dann wahr zurück, wenn das Objekt eine Instanz des Konstruktors ist. In der Funktion `toJS` stellen wir zuerst sicher, dass das Argument `church` mit zwei Funktionsabstraktionen beginnt. Dann rufen wir die Hilfsfunktion `countNestedApplications` auf, die einen Ausdruck und den Namen der successor-Funktion bekommt. Die Hilfsfunktion `countNestedApplications` zählt die Anzahl der Funktionsanwendungen durch rekursives Durchlaufen.

```
function toJS (church) {
  if (church instanceof Lambda) { // ist church eine Funktionsabstraktion?
    var name = church.binder;
    if (church.body instanceof Lambda) { // ist church.body eine Funktionsabstraktion?
      return countNestedApplications(church.body.body, name);
    }
  }
}

function countNestedApplications (expression, binder) {
  if ((expression instanceof Application) && expression.left.name === binder)
    return 1 + countNestedApplications(expression.right, binder);
```

```
  else
    return 0;
}
```

Wir haben Church-Numerale und einige Rechenoperationen auf ihnen implementiert. Auch haben wir Funktionen geschrieben, die Zahlen aus JavaScript in das Lambda-Kalkül übersetzen, und umgekehrt.

Bisher haben wir das Kodieren von Wahrheitswerten und natürlichen Zahlen durch Funktionsabstraktionen kennengelernt. Nach dem gleichen Schema können wir beliebige Daten repräsentieren. Ein Paar kann als dreistellige Funktion dargestellt werden. Es speichert zwei Werte und soll nicht komplett ausgewertet werden, daher müssen wir es in eine weitere Abstraktion einbetten. Mit Hilfe eines Paars können wir auch eine Liste repräsentieren: diese ist entweder leer (`nil`, `false`, 0) oder besteht aus einem Paar: der Kopf der Liste und dem Rest (Kapitel 5 im Abschnitt »Strukturelle Rekursion auf Listen« auf Seite 77). Mit Hilfe der verschiedenen Kodierungen können wir beliebigen Text im Lambda-Kalkül repräsentieren, wie über das gesamte Buch verteilt zu sehen ist.

Rekursion im Lambda-Kalkül: Fixpunktkombinatoren

Wir haben in Kapitel 5 die Rekursion kennengelernt. Auch im Lambda-Kalkül können wir Rekursion beschreiben. Da wir im Lambda-Kalkül keine globalen Variablen haben, kann sich eine Funktion nicht rekursiv aufrufen, da sie ihren eigenen Namen nicht referenzieren kann. Wir bedienen uns deshalb den Fixpunkten aus der Mathematik. Ein Fixpunkt `y` einer Funktion `f` ist dann erreicht, wenn `f` angewendet auf einen Wert `y` als Ergebnis das gleiche `y` liefert. Stellen wir uns die Funktion `f x = x * x` vor. Die Werte `0` und `1` sind hier Fixpunkte als `f 0 = 0` und `f 1 = 1`.

Ein Fixpunktkombinator ist eine Funktion höherer Ordnung, die den Fixpunkt einer Funktion berechnet. Ein bekannter Fixpunktkombinator ist der Y-Kombinator. Dieser Kombinator bekommt eine Funktion `f` als Argument und berechnet den Fixpunkt dieser Funktion. Der Y-Kombinator hat also die Spezifikation `Y = λ f → f (Y f)` Die direkte Implementierung dieser Spezifikation des Y-Kombinators in reinem JavaScript ist:

```
function Y (f) {
  return f(Y(f));
}
```

Und wenn wir, nur mal vorübergehend angenommen, Y als globale Variable im Lambda-Kalkül hätten:

```
Y = λ f → (f (Y f))
```

Diese Funktion würde allerdings aufgrund des rekursiven Aufrufs nicht terminieren. Zur Abhilfe umgeben wir den Aufruf mit einer Closure:

```
Y = λ f → (f (λ x → (Y f) x))
```

Das interessante ist nun, dass wir den rekursiven Aufruf von Y komplett entfernen können, indem wir den Ausdruck umformen:

```
Y = λ f → (λx → f (x x)) (λx → f (x x))
```

Wir wenden eine beliebige Funktion g auf den Y-Kombinator an:

```
Y g = (λ f → (λ x → f (x x)) (λ x → f (x x))) g ⟿
// β-Reduktion der Funktionsanwendung auf g
(λ x → g (x x)) (λ x → g (x x)) ⟿
// β-Reduktion der Funktionsanwendung links auf das Argument rechts
g ((λ x → g (x x)) (λ x → g (x x))) ⟿
//Einsetzen der Definition von Y
g (Y g)
```

Wir sehen durch die kurze Reduktionssequenz, dass der Y-Kombinator eine Fixpunktberechnung darstellt. Wenn wir weiter reduzieren, bekommen wir `g (g (Y g))`, `g (g (g (Y g)))` und so weiter.

Mit Hilfe des Y-Kombinators können wir also jede beliebige rekursive Funktion implementieren. Wir erhalten durch den Fixpunktkombinator eine Rekursion, ohne rekursive Funktionen zu implementieren. Und das alles im Lambda-Kalkül ohne Zusätze! Auf die drei grundlegenden Arten von Ausdrücken beschränkt können wir mit ein paar Tricks alle gängigen Programmierkonstrukte, also Daten, bedingte Anweisungen, und Rekursion im Lambda-Kalkül ausdrücken. Wir werfen nun noch einen Blick auf die partielle Auswertung.

Partielle Auswertung

Analog zur partiellen Funktionsanwendung in JavaScript in Kapitel 3 im Abschnitt »Teilweise Funktionsanwendung« auf Seite 40, können wir auch im Lambda-Kalkül partiell auswerten. Die Kombination aus partieller Funktionsanwendung und eifriger Auswertung, sozusagen eine sehr eifrige Auswertung, tritt dann in Aktion, wenn wir `plus` schon einmal mit der Zahl 1 füllen:

```
var inc = new Application(plus, one).reduce();
> (λ m → λ n → λ s → λ z → m s (n s z)) (λ s → λ z → s z) ⟿
    λ n → λ s → λ z → (λ s → λ z → s z) s (n s z)
> (λ s → λ z → s z) s ⟿ λ z → s z
> (λ z → s z) (n s z) ⟿ s (n s z)

inc.toString();
> 'λ n → λ s → λ z → s (n s z)'
```

Voilà, nun haben wir genau unsere Inkrementierungsfunktion von oben erhalten.

Die partielle Auswertung wird als Optimierung in Compilern verwendet.

Bei logischen Operationen erleichtert die partielle Auswertung das Verständnis:

```
new Application(und, falsch).reduce().toString();
> 'λ q → λ w → λ f → f'
```

```
new Application(oder, wahr).reduce().toString();
> 'λ q → λ w → λ f → w'
```

Hier sehen wir, dass die Funktionen jeweils ein weiteres Argument `q` erwarten, das aber keinen Einfluss auf das Ergebnis hat. Das ist intuitiv auch klar, weil die Funktionen mit `falsch` und `wahr` vorgefüllt schon das komplette Ergebnis kennt. In der Wahrheitstabelle ist das Ergebnis unabhängig vom letzten Argument.

Die mit `falsch` teilweise gefüllte Operation »`oder`« gibt das fehlende Argument zurück. Also erhalten wir die Identitätsfunktion als Ergebnis der partiellen Anwendung von »`oder`« auf `falsch`.

```
new Application(oder, falsch).reduce().toString();
> 'λ q → q'
```

Ein Parser für Lambda-Ausdrücke

Statt immer wieder mühsam die Konstruktoren für Funktionsabstraktionen, Variablen und Funktionsanwendungen von Hand zu schreiben, können wir diese Aufgabe von einem Parser erledigen lassen. In Kapitel 5 im Abschnitt »Rekursion über die Ein- und Ausgabedaten« auf Seite 76 haben wir erwähnt, dass Programme, die eine strukturierte Eingabe verarbeiten, als Parser aufgefasst werden können. Eine solche strukturierte Eingabe ist auch die Sprache »Lambda-Kalkül«.

Was ist nun die unterliegende Grammatik der Sprache?

Eine Grammatik besteht aus einer Menge von Regeln zur Ableitung, einem Startsymbol bei dem die Ableitung beginnt (hier `S`), einer Menge von Nichtterminalsymbolen die noch weiter abgeleitet werden können (hier `S`), und einer Menge von Terminalsymbolen die fertig abgeleitet sind (hier das 'λ', ' → ', '(', ')', und die Menge der Variablennamen, hier `a` genannt). Die Terminalsymbole sind die Symbole unserer Sprache, die am Ende ausgegeben werden. Unsere Grammatik hat nur eine einzige Regel. Diese Regel hat drei Alternativen, welche durch | voneinander getrennt werden, und den drei Arten von Ausdrücken entsprechen. Eine Regel ersetzt bei ihrer Anwendung das Nichtterminalsymbol auf der linken Seite von `::=` durch eine der Alternativen auf der rechten Seite:

```
S ::=  a            (Variable)
     | (S S)        (Funktionsanwendung)
     | λ a → S      (Funktionsabstraktion)
```

Die Grammatik ist ein Generator, sie generiert alle möglichen Ausdrücke. Ausgehend von Startsymbol `S` werden die Ableitungsregeln in allen ihren Alternativen angewendet, um Ausdrücke zu erzeugen, indem Nichtterminalsymbole (hier `S`) weiter ersetzt werden.

– Fortsetzung –

Da bei der zweiten Alternative, der Funktionsanwendung, nicht klar ist, welche Seite des Ausdrucks (S S) zuerst weiter ersetzt werden soll, formen wir sie so um, dass sie nur in der rechten Seite rekursiv ist. Dies erreichen wir durch eine zusätzliche Regel, ein weiteres Nichtterminalsymbol A und ein weiteres Terminalsymbol ε, das in einer Grammatik was den leeren String repräsentiert.

```
A ::=  a A     (Variable)
     | (S) A   (Funktionsanwendung)
     | ε
S ::=  A
     | λ a → S (Funktionsabstraktion)
```

Zu jeder Grammatik gibt es einen zugehörigen Parser, der genau die durch die Grammatikregeln ableitbare Sprache akzeptiert. Der Parser baut für jeden wohlgeformten Ausdruck, den er akzeptiert, einen abstrakten Syntaxbaum auf (vgl. Abbildung 7-1 auf Seite 105).

Wir bauen nun einen Parser für das Lambda-Kalkül, indem wir die obige Grammatik und Parser-Kombinatoren verwenden. Parser-Kombinatoren sind spezielle Funktionen höherer Ordnung, mit denen wir aus Terminal- und Nichtterminal-Parsern analog zur Grammatikstruktur einen Parser aufbauen können. Ein Parser ist eine Funktion, die festgelegte Zeichenfolgen akzeptiert. Ein Terminalparser ist dabei ein Parser, der genau ein Terminalsymbol akzeptiert, z.B. einen Variablennamen a, eine Klammer '(' oder ')', oder ein 'λ' oder einen '→'. Die Parser-Kombinatoren entsprechen genau der Art, auf die aus Terminal- und Nichtterminalsymbolen Grammatikregeln konstruiert werden.

Chris Double hat für JavaScript eine Bibliothek entwickelt, die verschiedene Parser-Kombinatoren bereitstellt (Link *https://github.com/doublec/jsparse)*, welche wir nun zum Parsen des Lambda-Kalküls benutzen.

Der Terminal-Parser `token(s)` akzeptiert genau den String `s`, der ihm als Argument übergeben wird.

Wir benutzen den Kombinator choice, um wie das Alternativsymbol | alternative Auswahl zwischen den rechten seiten einer Regel zu treffen. Wir benutzen den Kombinator `repeat1` um sicherzustellen, dass die rekursive Regel mindestens einmal ausgeführt wird. Dies ist nötig, damit nicht einfach der leere String ε akzeptiert wird. Wir benutzen den Kombinator `sequence`, um mehrere Parser hintereinander auszuführen. Dies ist nötig, wenn mehrere Symbole in der rechten Seite einer Regel nebeneinander stehen. Der Kombinator `whitespace` erzeugt einen Parser, der alle Leerzeichen vom Beginn der Eingabe abknabbert. Der Kombinator `wsequence` kombiniert `whitespace` und `sequence` jeder Parser in der Sequenz entfernt Leerzeichen. Der Kombinator `expect` wandelt einen ihm übergebenen Parser so um, dass der Parser zwar immer noch das gleiche akzeptiert, das Ergebnis allerdings verwirft, und nicht in den AST einfügt. Doch wie sieht der Parser für Variablennamen aus? Wir erlauben Buchstaben von a bis z, ausgedrückt durch den Parser.

– Fortsetzung –

```
var variable = whitespace(range("a", "z")); //ein Buchstabe in Kleinschreibung
```

Unser kompletter Parser für das Lambda-Kalkül entspricht der Struktur der Grammatik:

```
var a = action(variable, function (ast) { return new Variable(ast); });
var A = action(
 repeat1(choice(a, action(wsequence(expect(token("(")), S, expect(token(")"))),
                          function (ast)  { return ast[0]; }))),
 function (ast) {
  return ast.reduce(function (a, b)
   { return new Application(a, b);   });
 });
var S = choice(
 A,
 action(wsequence(expect(token("λ")), variable, expect(token("→")), S),
       function (ast) { return new Lambda(ast[0], ast[1]); }));
```

Der Parser akzeptiert jeden wohlgeformten Ausdruck, und baut aus ihm automatisch einen abstrakten Syntaxbaum auf. Der abstrakte Syntaxbaum ist wie wir oben gesehen haben, der Baum eines Lambda-Ausdrucks mit den Konstruktoren und Terminalsymbolen. Wir probieren unseren Parser aus. Das Ergebnis des Parsers ist ein Objekt, das in der Eigenschaft `ast` den abstrakten Syntaxbaum enthält.

```
S(ps("λ x → x")).ast.toString();
> "λ x → x"
S(ps("(λ x → x) y")).ast.reduce().toString();
> (λ x → x) y ⟿ y
> "y"
```

Der vom Parser erzeugte abstrakte Syntaxbaum entspricht dem Baum des Ausdrucks, wie wir ihn vorher umständlich von Hand eingegeben haben. Der Auswerter funktioniert also immernoch genau wie vorher auf dem abstrakten Syntaxbaum, nur müssen wir weniger schreiben.

Fazit zum Kapitel – ist noch Lamm da?

In diesem Kapitel haben wir das Lambda-Kalkül kennengelernt und in JavaScript implementiert. Das Lambda-Kalkül ist die Kernsprache sämtlicher funktionaler Programmierung, und die Regeln des Lambda-Kalküls zur Auswertung von Ausdrücken gelten in allen Programmiersprachen. Durch Rückführung auf diese Kernsprache mittels Curryfizierung und Kodierung lassen sich Programmeigenschaften leichter beweisen. Das Lambda-Kalkül ist zudem mächtig genug, um Daten wie Wahrheitswerte und natürliche Zahlen, und außerdem Kontrollstrukturen wie bedingte Anweisungen oder Rekursion in ihm auszudrücken.

Die unterschiedlichen Auswertungsstrategien wie faul (lazy, non-strict) oder eifrig (eager, strict) führen zu verschiedene Programmiersprachen. Haskell als rein funktionale Sprache, die faul ausgewertet wird, hat eine andere Semantik als das ebenfalls rein funktionale, aber eifrig ausgewertete ML.

JavaScript besitzt durch Kontrollstrukturen und vordefinierten Funktionen viel syntaktischen Zucker, also Sprachelemente, die das Programmieren erleichten. Lässt man diesen weg, ist JavaScript im Kern ein call-by-value Lambda-Kalkül, das primitive Datentypen integriert und als Erweiterung das Prototypen-System hat. Das in diesem Kapitel vorgestellte Lambda-Kalkül ist untypisiert, aber es gibt als Erweiterung das typisierte Lambda-Kalkül, in welchem jeder Variablen ein Typ zugewiesen wird. Im nächsten Kapitel lernen wir mehr über Typen.

Nach dem eher sparsamen Lambda-Kalkül ist es vielleicht Zeit für etwas Deftiges. Ist noch Lamm da? Dann wäre ein Lamm-Curry nicht schlecht!

Rezept: Lamm-Curry

Für zwei Personen:

- 600 g Lammkeule
- 3 Tomaten
- 1 Zwiebel
- 1 daumengroßes Stück Ingwer
- 1 Limette
- 200 ml Kokosmilch
- 1 TL Senfsaat
- 1/2 TL Kurkuma
- 1/2 TL Chilipulver
- 1 Chilischote
- 2 Limettenblätter
- neutrales Öl
- Salz

Lammfleisch in mundgerechte Stücke schneiden, Tomaten waschen und grob zerschneiden. Ingwer schälen und klein schneiden. Zwiebel schälen, halbieren und würfeln. Von zwei Limettenblättern die dicke Blattrippe entfernen und den Rest in dünne Streifen schneiden. Chilischote waschen und fein zerschneiden.

Das Lammfleisch in einer großen Pfanne oder einem Wok mit etwas Öl bei großer Hitze gut anbraten. Das braun gebratene Lamm salzen und in einer Schüssel beiseite stellen.

Bei mittlerer Hitze in der Pfanne Senfsaat, die frische Chilischote, Limettenblätter und Ingwer einige Minuten anbraten. Sobald die Senfkörner aufploppen, die Zwiebelwürfel dazugeben und unter mehrmaligem rühren langsam anbraten, damit sie glasig und weich werden. Chilipulver und Kurkuma streuen und umrühren. Dann alles mit den Tomatenwürfeln ablöschen und einige Minuten so vor sich hin köcheln lassen.

Anschließend ein wenig Wasser, Kokosmilch sowie das vorbereitete Lammfleisch mit in die Pfanne geben, umrühren und das Curry 20 Minuten köcheln lassen. Sobald das Lamm zart und die Sauce eingedickt ist, mit Limetten und Salz abschmecken.

Wir sind schon sehr tief in die Theorie der funktionalen Programmierung eingedrungen. Das Lambda-Kalkül dürfte für die meisten Programmierer und Programmiererinnen nur selten im Arbeitsalltag relevant werden. Nichtsdestotrotz hilft uns eine eigene Implementierung dieser Sprache beim Verständnis dafür, was JavaScript im Innersten zusammenhält. Um unser untypisiertes Lambda-Kalkül um die Datentypen von JavaScript zu ergänzen, sollten wir uns zunächst mit der Theorie hinter den Typen beschäftigen. Darum soll es im nächsten Kapitel gehen.

```
(λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
z)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

KAPITEL 8

Typen

Typisierung und Typüberprüfung

Beim Kochen kann durch die Prüfung der Zutatenliste eines Rezeptes vorher geschaut werden, ob genügend Nährstoffe (Eiweiß, Fett, Zucker, Vitamine) im fertigen Gericht sein werden. Diese Prüfung gibt uns vielleicht einen groben Anhaltspunkt, ob die Grundzutaten sinnvoll gewählt sind, und ob man von dem Gericht satt wird. Ob es aber wirklich gelingt, wissen wir zu diesem Zeitpunkt noch nicht. Die Prüfung ist außerdem nicht zwingend erforderlich, man kann auch einfach loskochen und schauen, was dabei herauskommt.

Ähnlich ist es beim Programmieren. Schon bevor der Code ausgeführt wird, kann durch Typisierung und Typüberprüfung herausgefunden werden, ob die für die richtige Funktion nötigen Eckdaten wie die Typen von Argumenten und Rückgabe stimmen. Typen beinhalten also Informationen über die Daten eines Programmes zur Compilezeit, wenn die konkreten Daten noch nicht da sind. Dynamisch typisierte Sprachen halten zur Laufzeit Typinformationen bereit. Es gibt eine Vielfalt verschiedener Typsysteme und Erweiterungen, die verschieden »strengen« Überprüfungen entsprechen.

Die beiden Hauptkriterien, nach denen sich die Typsysteme von Programmiersprachen unterscheiden lassen, sind dynamisch im Gegensatz zu statisch und stark im Gegensatz zu schwach typisiert. Wie in Kapitel 1 im Abschnitt »Typisierung« auf Seite 12 schon angedeutet, ist JavaScript eine dynamisch und schwach typisierte Sprache.

JavaScript ist dynamisch typisiert

In dynamisch typisierten Sprachen existieren die Typinformationen zur Laufzeit. Erst wenn es notwendig ist, zum Beispiel bei einer Zuweisung oder bei einer Referenzierung, wird auf die Typinformation eines Objekts oder Wertes geschaut. Variablen haben den Typ des Objektes oder Wertes, das sie referenzieren, aber von sich aus zunächst keinen speziellen Typ. Derselben Variable können nacheinander Werte oder Objekte unterschiedlicher Typen zugewiesen werden.

```
var a = "Garam Masala"; //a ist ein String
a = 23; //a ist eine Zahl
```

Wird nun eine Zahl benutzt, wo eigentlich eine Funktion erwartet wird, erhalten wir einen Typfehler, wie hier in `reduce` aus Kapitel 4 im Abschnitt »Vordefiniertes reduce auf Arrays« auf Seite 54 zu sehen:

```
[1, 2, 3].reduce(10);
> TypeError: 10 is not a function
```

Wird hingegen eine Funktion übergeben, wo eine Zahl erwartet wird, bekommen wir »not a number« (`NaN`):

```
1 - function (a) { return a; };
> NaN
```

Java und Haskell hingegen sind statisch typisierte Sprachen. Für jede Variable muss zur Compile-Zeit ihr Typ bekannt sein. Dazu müssen die Typen der Variablen oder Funktionen im Programmcode als Typdeklaration hingeschrieben werden. Die Typen können im Compiler geprüft werden, was eine frühere Fehlererkennung erlaubt.

Dynamisch typisierte Programmiersprachen wie JavaScript und seine Vorbilder Scheme und Self haben den Vorteil, dass man in ihnen schneller ein Programm entwickeln kann, da man die Typen nicht explizit hinzuschreiben braucht. Auch Änderungen in Programmen sind einfacher, da man sich bei der Änderung keine Gedanken um die Typen des restlichen Programms machen muss, das gerade nicht ausgeführt wird. In einer statisch typisierten Sprache müsste man in diesem Fall erst die Typen des gesamten Programms anpassen, bevor man es testen kann.

JavaScript ist schwach typisiert

Ausserdem ist JavaScript schwach (auch weak oder loosely) typisiert. Das bedeutet, dass JavaScript auf eigene Faust die Werte der Variablen in angemessene Typen konvertiert, je nachdem welcher Typ im jeweiligen Kontext der Variable erforderlich ist.

```
var nNeighbors = 4;
var nColleagues = 6;
nNeighbors + nColleagues + ' Portionen';
> 10 Portionen

'Portionen: ' + nNeighbors + nColleagues;
> Portionen: 46
```

Es ist ein grosser Unterschied, ob wir für 10 oder 46 Gäste kochen! Im Absatz »Falle 2: Überladene Operatoren« auf Seite 142 schauen wir genau, was hier passiert.

Vorsicht beim Umgang mit Typen in JavaScript

Obwohl JavaScript dynamisch und schwach typisiert ist, bedeutet das nicht, dass wir beim Programmieren nicht über die Typen unserer Variablen nachzudenken brauchen.

Ein häufiger Fehler bei der Web-Programmierung mit JavaScript tritt beim Auslesen von Eingaben in HTML-Formularen auf. Liest man die Value-Eigenschaft eines HTML-Formulars aus, in das der Benutzer eine Zahl eintragen soll, und addiert auf den ausgelesenen Wert eine andere Zahl, wie `42`, bekommt man ein unerwartetes Ergebnis. Die Value-Eigenschaft eines Formulars ist ein String (obwohl der String die gewünschte Zahl enthält). Addieren wir auf diesen String nun eine andere Zahl, wie `42`, so wird die Zahl `42` bei der Addition durch die automatische Typkonversion in einen String `"42"` umgewandelt, wie oben im Beispiel gesehen. Dieser String wird dann an den String aus dem Formular angehängt, anstatt die gewünschte Summe zweier Zahlen zu produzieren.

Dieses Vorgehen wird durch den janusköpfigen Operator `+` verursacht, der einerseits Zahlen addieren und andererseits auch Strings konkatenieren, also aneinanderreihen kann. Welche von beiden Versionen zur Anwendung kommt, hängt vom Kontext ab, und die Addition wird nur durchgeführt, wenn beide Argumente auch wirklich Zahlen sind.

Im Folgenden stellen wir zunächst die Datentypen der Sprache JavaScript vor, und gehen dann auf die Tücken der automatische Typkonversion (Coercion) und des Polymorphismus ein, wofür der Operator `+` mit seiner Doppelfunktion ein Beispiel ist.

Primitive Datentypen

Da JavaScript dynamische Datentypen hat, kann die gleiche Variable je nach Kontext an einen unterschiedlichen Datentyp gebunden sein. Mit dem `typeof`-Operator können wir herausfinden, welchen Typ eine Variable hat.

```
var x; // x ist undefiniert
typeof(x);
> 'undefined'

var x = 23; // x ist eine Zahl
typeof(x);
> 'number'

var x = "Kurkuma"; // x ist ein String
typeof(x);
> 'string'
```

Wir sehen, dass die Variable `x` je nach zugewiesenem Wert verschiedene Typen wie `number` oder `string` annimmt. Diese Typen sind in die Sprache eingebaute grundlegende Typen und werden primitive Datentypen genannt. Es gibt einige primitive Typen in JavaScript, die wir im folgenden kurz vorstellen.

Strings

Ein String ist ein Datentyp, der eine Zeichenfolge wie zum Beispiel »Garam Masala« enthält. Das kann jeder beliebige Text zwischen Anführungszeichen (Quotes) sein. Man kann einfache oder doppelte Anführungszeichen (Single- oder Double-Quotes) benutzen.

```
var spice = "Garam Masala";
var spice = 'Schwarzer Pfeffer';
typeof(spice);
> 'string'
```

Man kann innerhalb eines Strings Anführungszeichen verwenden, solange sie nicht die Anführungszeichen sind, mit denen der String selbst markiert ist. Ansonsten kann man sie auch – wie in einigen anderen Programmiersprachen – mit einem vorangestellten Backslash `'\'` markieren, um zu verhindern, dass sie als Endezeichen des Strings interpretiert werden.

```
var answer = "So geht's durchaus";
var answer = "Sein Name ist 'Haskell B. Curry'";
var answer = 'Sein Name ist "Brendan Eich"';
var answer = "Sein Name ist \"Haskell B. Curry\"";
```

Zahlen

JavaScript hat nur einen Datentyp für Zahlen. Man kann Zahlen in JavaScript mit oder ohne Dezimalpunkt schreiben. Alle Zahlen in JavaScript werden als 64-Bit-Fließkommazahlen auf dem Computer repräsentiert, wie es im IEEE-754 Standard festgelegt ist.

```
var portions = 34.00; // mit Dezimalpunkt. Keine halben Portionen! :D
var portions = 34; // ohne Dezimalpunkt.
typeof(portions);
> 'number'
```

Sehr große oder sehr kleine Zahlen können in wissenschaftlicher Notation geschrieben werden.

```
var avogadro = 6e23; // 6 000 000 000 000 000 000 000 00 mol^-1
var molar_mass = 1e-3; // 0.001 kg/mol
```

Wahrheitswerte/Booleans

Booleans können zwei Werte haben: `true` (wahr) und `false` (falsch).

```
var hungry = true;
var sated = false;
typeof(sated);
> 'boolean'
```

Wir kennen Wahrheitswerte auch aus dem Lambda-Kalkül in Kapitel 7. Dort haben wir falsch, 0, und die leere Liste identisch im Lambda-Kalkül definiert. Auch in JavaScript gibt es mehrere Repräsentationen für den Wahrheitswert `false`. Alle Zahlen sind wahr,

ausser der Zahl `0` und `NaN`. Ausserdem sind `null` und `undefined` falsch. Auch der leere String `""` ist falsch, sowie das leere Array `[]`. Und natürlich auch der Wahrheitswert `false`. Alle anderen Werte sind wahr.

Arrays

Mit dem folgenden Programmcode können wir ein Array `veggies` erzeugen.

```
var veggies = new Array();
veggies[0] = "Möhre";
veggies[1] = "Broccoli";
veggies[2] = "Paprika";
```

Oder in Kurzform (condensed array):

```
var veggies = new Array("Möhre", "Broccoli", "Paprika");
```

Oder direkt ausgeschrieben (literal array):

```
var veggies = ["Möhre", "Broccoli", "Paprika"]
```

Array-Elemente werden mit Null beginnend durchgezählt, das erste Element ist also `[0]`, das zweite `[1]` und so weiter.

Interessanterweise gibt uns der Operator `typeof` für unser Array `veggies` den Typ `object` zurück. Arrays sind also immer Objekte in JavaScript.

```
typeof(veggies);
> 'object'
```

Objekte

Ein Objekt wird von geschweiften Klammern eingeschlossen. Innerhalb dieser Klammern werden die Eigenschaften des Objekts als Paare von Namen und Werten definiert (`name:value`). Die verschiedenen Eigenschaften werden dabei durch Kommas getrennt. So wie die Gewürz-Objekte, die wir in Kapitel 2 im Abschnitt »Abstraktion von einem Codeblock: Funktionen« auf Seite 24 definiert haben.

```
var spice = {
   "name": "Zimt",
   "state": "gemahlen",
   "quantity_in_grams": "1",
   "active_compound": "Cinnamaldehyde"
};
```

Man kann nun auf die folgenden zwei Arten auf die Objekteigenschaften zugreifen:

```
var spicename = spice.name;
var spicename = spice["name"];
```

Objekteigenschaften haben je nach Wert selbst eigene Datentypen und verhalten sich wie Variablen, nur dass sie zum Objekt gehören.

```
typeof(spice.name);
> 'string'
```

Undefined und Null

`Undefined` kann sich in JavaScript auf den Typ als auch auf den Wert der Spezialvariable `undefined` beziehen. Alle Variablen, denen kein Wert zugewiesen wurde, haben den Wert `undefined`. Wenn wir durch `if (x === undefined)` überprüfen, ob eine Variable `undefined` ist, vergleichen wir `x` mit dem Wert der Variablen `undefined`. Wir können auch mit `if (typeof(y) === typeof(undefined))` überprüfen, ob eine Variable `defined` ist, dann vergleichen wir den Typ der Variablen `undefined` mit dem Typ von `y`.

In einigen JavaScript-Implementierungen kann der Variablen `undefined` auch ein anderer Wert zugewiesen werden. Dies sollte man allerdings vermeiden. Der `undefined`-Mechanismus ist auch aus anderen Skriptsprachen wie Perl (`undef`) bekannt. Variablen in JavaScript können zurückgesetzt oder geleert werden, indem man ihren Wert auf `null` setzt. Wir können auch Variablen wieder undefinieren, indem wir ihnen `undefined` zuweisen.

```
veggies = null;
typeof(veggies);
> 'object' // nicht undefined! nur "null"
```

Dabei ist zu beachten, dass `null` im Gegensatz zu `undefined` ein Schlüsselwort ist, und nicht überschrieben werden kann. Dies kann man ausprobieren, indem man versucht, `null` etwas zuzuweisen, dann bekommt man einen Fehler:

```
null = 42
> "ReferenceError: Invalid left-hand side in assignment"
```

Der Variablen `undefined` können wir auch keinen neuen Wert zuweisen, aber der Versuch einer Zuweisung führt zu keinem Fehler:

```
undefined = 42;
> "42"
```

Dieser Unterschied erklärt, warum ein Vergleich mit `null` schneller sein kann, als ein Vergleich mit `undefined`. Bei einem Test mit dem Operator `==` auf `undefined` wird nicht nur ein Wert verglichen, sondern eine Typkonversion durchgeführt (vgl. Kapitel 8 im Abschnitt »Falle 1: Automatische Typkonversion / Coercion« auf Seite 141).

Deklarieren von Variablen, Erzeugen von Objektinstanzen mit new

Wenn man mit var eine neue Variable deklariert, kann man ihren Typ festlegen, indem man ihr direkt einen Wert zuweist. Neue Instanzen eines Objekts werden mit dem Schlüsselwort new erzeugt.

```
var veggiename = new String;
var portions = new Number;
var hungry = new Boolean;
var veggies = new Array;
var spice = new Object;
```

Wenn wir als Wert eine Instanz von z.B. String, Number, oder Boolean, die es auch als primitive Datentypen gibt, mit new erzeugen, erzeugen wir jeweils ein neues Objekt und die Werte sind alle vom Typ object. Genauer gesagt passiert unter der Haube noch etwas, wenn wir mit new ein Objekt erzeugen: Zusätzlich wird auch die Konstruktor-Funktion ausgeführt.

```
typeof(veggiename);
> 'object'
typeof(portions);
> 'object'
typeof(hungry);
> 'object'
typeof(veggies);
> 'object'
typeof(spice);
> 'object'
```

Jedes JavaScript-Objekt kann entsprechend dem objektorientierten Paradigma eigene Properties (Eigenschaften) und Methods (Methoden/Objektfunktionen) haben. Dabei werden die auf einem Wert verfügbaren Eigenschaften und Methoden auf dem Prototyp des Objektes definiert, so dass auf den Objekten, die sich hinter veggiename, portions, hungry, veggies und spice verbergen, trotz gleichem Typ object jeweils unterschiedliche Methoden verfügbar sind. Zum Beispiel sind die Funktionen map, reduce und filter, die wir aus Kapitel 4 im Abschnitt »Weitere Funktionen höherer Ordnung: Map, Reduce, Filter und Kolleginnen« auf Seite 46 kennen, nur auf Arrays verfügbar.

```
veggies.map(id);
> []
portions.map(id);
> TypeError: Object [object Number] has no method 'map'
```

Primitive Datentypen versus Objekte

Entgegen anderslautender Behauptungen ist in JavaScript jedoch nicht »alles vom Typ Objekt«. Link: *http://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/*. Es gibt die oben genannten primitiven Datentypen, und diese sind keine Objekte, wenn sie nicht mit new erzeugt wurden.

- Strings
- Zahlen
- Wahrheitswerte/Boolean
- Undefined
- Null

Außerdem haben Funktionen noch eine Sonderstellung, und der `typeof`-Operator erkennt diese und gibt `function` zurück. Alle anderen Daten sind in der Tat vom Typ `object`.

Zur Erklärung muss erwähnt werden, dass auch Strings, Zahlen und Booleans sich mitunter wie Objekte verhalten. Z.B. kann man auf die Länge eines Strings wie auf eine Objekt-Eigenschaft zuzugreifen:

```
"Kurkuma".length;
> 7
```

JavaScript ist hier sehr entgegenkommend und erzeugt ein temporäres Wrapper-Objekt (Verpackung/Hülle), das ermöglicht, auf Eigenschaften des primitiven Datentyps wie auf die eines Objektes zuzugreifen.

Der Operator `typeof` hilft uns also, die verschiedenen primitiven Datentypen, Funktionen und Objekte zu unterscheiden. Alle Objekte haben jedoch den Typ `object`, `typeof` hilft uns also nicht weiter, sie feiner zu unterscheiden. Wie können wir den Prototyp eines Objekts, der in objektorientierten Programmiersprachen wie Java der Klasse des Objekts entspräche, herausfinden?

In diesem Fall helfen uns `constructor` und `instanceof`. Eine Eigenschaft auf allen Objekt-Prototypen ist `constructor`, sie gibt die Konstruktor-Funktion des jeweiligen Objekts zurück. Beispielsweise gibt `{}.constructor` die Objekt-Konstruktorfunktion zurück, und `[].constructor` gibt die Array-Konstruktorfunktion zurück. Genauso bekommt man bei einem selbst definierten Objekt auch eine selbstgeschriebene Konstruktorfunktion zurück.

Dann gibt es noch den `instanceof`-Operator, der zwei Argumente bekommt, einen zu prüfenden Wert (z.B. eine Variable) und einen Konstruktor. Er gibt `true` zurück, wenn das zu prüfende Objekt eine Instanz des Konstruktors ist. Dieser kann auch bei komplexen Fällen von Vererbung Klarheit verschaffen, beispielsweise ist ein `spice` auch immer ein Objekt, weil `Object` eine Oberklasse ist.

```
spice instanceof Object;
> true
```

Quelle: *http://skilldrick.co.uk/2011/09/understanding-typeof-instanceof-and-constructor-in-javascript/*

Verwandt ist auch der Operator `isPrototypeOf`, mit dem man überprüfen kann, ob ein Objekt in der Kette von Prototypen (Kapitel 1 im Abschnitt »Prototypenbasiertes Objekt-

system« auf Seite 9) eines anderen Objekts vorkommt. Der Prototype selbst kann im Gegensatz zur Konstruktorfunktion nicht überschrieben werden. Seit JavaScript 1.8.1 gibt es noch `getPrototypeOf`. Link: *http:// stackoverflow.com/questions/5972991/why-do-we-need-the-isprototypeof-at-all*

Polymorphismus

Das Wort Polymorphismus kommt aus dem griechischen (πολυμορφισμός Polymorphismos) und bedeutet Vielgestaltigkeit. Es ist ein Konzept, das ermöglicht, die gleiche Funktion auf verschiedenen Typen oder Objekten aufzurufen, und dabei für jedes in seiner eigenen Weise zu reagieren. Verschiedene Programmiersprachen-Communities benutzen das Wort Polymorphismus für jeweils unterschiedliche Konzepte.

Frühe Programmiersprachen wie Pascal erlauben nur einen Typ für jede Funktion und werden daher monomorph genannt. Die Identitätsfunktion, die uns in Kapitel 4 im Abschnitt »Verallgemeinerung der Funktionen höherer Ordnung auf Arrays« auf Seite 61 begegnet ist, muss in Pascal für jeden Typ neu implementiert werden. Die Identitätsfunktion auf Zahlen und die Identitätsfunktion auf Strings sind in monomorphen Sprachen zwei verschiedene Funktionen.

Einen anderen Ansatz verfolgen polymorphe Programmiersprachen, in denen eine Funktion Argumente und Rückgabe von verschiedenem Typ haben kann.

Arten von Polymorphismus

Wir halten uns an die Unterteilung des Polymorphismus von Luca Cardelli und Peter Wegner in mehrere Klassen. Dabei wird zunächst zwischen dem universellen und den Ad-hoc-Polymorphismus unterschieden, die jeweils nochmal unterteilt werden.

Beim universellen Polymorphismus kann eine Funktion gleich für mehrere Ein- und Ausgabetypen definiert werden. Einige Funktionen sind dabei nicht komplett universell, sondern setzen bestimmte Eigenschaften bei Typen voraus. Sortierfunktionen können zum Beispiel nur auf Objekten arbeiten, die vergleichbar sind. Eine Funktion kann *universell für mehrere Ein- und Ausgabetypen* definiert werden.

Neben dem universellen Polymorphismus gibt es den Ad-hoc-Polymorphismus. Hier wird polymorphes Verhalten umgesetzt, indem verschiedene, auf die jeweiligen Typen angepasste Funktionen unter dem gleichen Namen implementiert werden. Durch den *überladenen, also mehrfach definierten Funktionsnamen* sieht es so aus, als ob die gleiche Funktion auf verschiedenen Typen aufgerufen werden kann.

Universeller Polymorphismus

Den universellen Polymorphismus unterteilen wir weiter in den parametrischen und den Subtypen-Polymorphismus.

Im parametrischen Polymorphismus abstrahieren wir von Typen durch die Einführung von Typvariablen im Funktionstyp. Die Typvariablen werden bei der Typprüfung durch die passenden konkreten Typen ersetzt. Bei Verwendung von Typvariablen sind die Funktionstypen automatisch »so abstrakt wie möglich». Die Identitätsfunktion, deren Implementierung ja völlig unabhängig vom konkreten Typ ist, hat unter Verwendung einer Typvariable `a` den Funktionstyp »von a nach a«, auch `a -> a` geschrieben. Wird die Identitätsfunktion auf Zahlen angewendet, dann wird für die Variable `a` der Typ Zahl eingesetzt, und Argument und Rückgabe der Funktion sind vom Typ Zahl.

In der funktionalen Sprache Haskell können Typvariablen durch Typkontexte auf Typklassen, und damit auf bestimmte Eigenschaften, eingeschränkt werden. Durch den Funktionstyp mit Typkontext `Num a => a -> a` wird der Funktionstyp `a -> a` auf die Typklasse `Num` für Zahlen eingeschränkt. Eine Typklasse wie Num fasst Typen zusammen, die die gleichen Funktionen implementieren, in diesem Fall +, –, * und `negate` für `Num`.

Auch objektorientierte Sprachen wie C++, Java und C# unterstützen parametrischen Polymorphismus durch Templates oder Generics.

Subtypen-Polymorphismus ist Polymorphismus durch Vererbung ohne Spezialisierung. Die Klasse `Paprika` als eine Spezialisierung der allgemeineren Klasse `Gemüse` erbt alle Eigenschaften und Methoden der Oberklasse `Gemüse`. Die Methode `schnippeln` ist allgemein auf der Klasse `Gemüse` implementiert. Wird nun die Methode `schnippeln` auf einem Objekt vom Typ `Paprika` aufgerufen, wird die Methode `schnippeln` aus der Elternklasse ausgeführt. `Möhren`, `Paprika` und `Gurken` werden alle durch ein und dieselbe Methode geschnippelt. Da für alle Subtypen der Klasse `Gemüse` die gleiche in der Superklasse `Gemüse` implementierte Methode ausgeführt wird, bezeichnen wir dies als Subtypen-Polymorphismus.

Ad-hoc-Polymorphismus

Die zweite Polymorphismus-Gruppe, den Ad-Hoc-Polymorphismus, unterteilen wir in Überladung, Überschreiben und Duck Typing. Die Auswahl der Funktion anhand der Typen der Argumente (Überladung) oder aber des Typs (Überschreiben) oder Namens (Duck Typing, Name Typing) des aufrufenden Objekts ist im Bezug auf das Sprachdesign die einfachste Umsetzung des Polymorphismus und wird deshalb Ad-Hoc-Polymorphismus genannt.

Überladung ist die Definition mehrerer Funktionen gleichen Namens, die sich jedoch in ihrer Argument- oder Rückgabetypen unterscheiden. Dadurch wird die Bedeutung eines Funktionsnamens oder Operators überladen. Beim Aufruf einer überladenen Funktion wird anhand der Typen die passende Implementierung ausgewählt.

Sowohl Überschreiben als auch Name Typing sind Polymorphismus-Konzepte im objektorientierten Paradigma. Überschreibung ist die Vererbung mit Spezialisierung, wenn wir also die Methode `schnippeln` auf der `Kartoffel` anders implementieren als auf der `Gurke`. Dabei verdeckt die Implementierung der Methode `schnippeln` in der speziali-

sierten Klasse `Gurke` die Methode gleichen Namens in der Oberklasse, sie überschreibt diese Methode. Die spezialisierte Methode wird anhand des Typs des aufrufenden Objekts ausgewählt.

Beim Duck Typing wird im Programm durch die Abfrage einer Eigenschaft des Objektes (z.B. dem Namen) entschieden, welche Methode aufgerufen wird. Wenn ein Objekt aussieht wie eine Ente, schwimmt wie eine Ente und schnattert wie eine Ente, dann kann man auch denken, es ist tatsächlich eine Ente, obwohl sie kein sichtbares Etikett hat, welches sie als Ente bezeichnet. Das Wort Duck Typing kommt von dem nicht ganz ernst gemeinten Ententest in der Logik, eine Form der Induktion, die möglicherweise schon auf Wilhelm von Ockham zurückgeht, der mit dem Rasiermesser.

Die vorgestellten Wege wie Vererbung, Überladung und Duck Typing ermöglichen alle, beim Programmieren die gleiche Codezeile auf Variablen verschiedener Typen zu benutzen. Sie sind somit unterschiedliche Verkörperungen des allgemeinen Konzepts Polymorphismus.

JavaScript: Polymorphismus durch Prototypen

JavaScript ist, wie wir gesehen haben, eine dynamisch typisierte Sprache, die auf prototypischer Vererbung basiert. Ein JavaScript-Beispiel für Polymorphismus durch Vererbung, wie wir ihn in objektorientierten Sprachen wie Java kennen, könnte folgendermaßen aussehen:

```
function Gewuerz () {
  throw new Error("Abstrakte Klasse");
}

Gewuerz.prototype.printWirkstoff = function () {
  throw new Error ("Nicht implementiert");
};

function Pfeffer () {
  // Konstruktor
}

Pfeffer.prototype = Object.create(Gewuerz.prototype);
Pfeffer.prototype.printWirkstoff = function () {
  // Implementierung für Pfeffer
};

function Zimt () {
  // Konstruktor
}

Zimt.prototype = Object.create(Gewuerz.prototype);
Zimt.prototype.printWirkstoff = function () {
  // Implementierung für Zimt
};
```

Dann können wir in unserem Programm `printWirkstoff` für verschiedene Gewürze verwenden, nachdem wir überprüft haben ob wir auch tatsächlich ein Objekt vom Typ `Gewuerz` vorliegen haben.

```
var obj = new Zimt();

if (obj instanceof Gewuerz) {
    // mit einem Gewürz etwas machen, z.B. printWirkstoff() aufrufen
}
```

Oder wir prüfen mittels »Duck Typing«, ob `printWirkstoff` aufgerufen werden kann. Duck Typing geht davon aus, dass alles, was eine Methode quaken besitzt, eine Ente ist. Das Schlüsselwort `in` gibt `true` zurück, wenn das Objekt `obj` eine Eigenschaft (Property) mit dem gegebenen Namen `printWirkstoff` enthält. Dies ist in JavaScript etwas leichter umzusetzen als beim statischen Typing, bei dem nur ein Objekt einer Kindklasse die gleiche Methode erben und überschreiben, also eine Ente sein kann.

```
if ("printWirkstoff" in obj)
    obj.printWirkstoff();
```

oder

```
if (obj.printWirkstoff)
    obj.printWirkstoff();
```

oder noch genauer, inklusive Überprüfung ob es sich um eine Funktion handelt:

```
if (typeof obj.printWirkstoff === "function")
    obj.printWirkstoff();
```

Man könnte außerdem `Gewuerz` als Objekt ohne Konstruktor definieren, so würde es konzeptuell eher einer abstrakten Klasse entsprechen. Hierbei benutzen wir die Objekt-Prototypen, um die Implementierung nachzuliefern, also die abstrakte Klasse zu implementieren.

```
var Gewuerz = {
    printWirkstoff : function () {
        throw new Error ("Not implemented");
    }
}

function Pfeffer () {
  // Konstruktor
}

Pfeffer.prototype = Object.create(Gewuerz);
Pfeffer.prototype.printWirkstoff = function () {
  // Implementierung für Pfeffer
};

function Zimt() {
  // Konstruktor
}
```

```
Zimt.prototype = Object.create(Gewuerz);
Zimt.prototype.printWirkstoff = function () {
  // Implementierung für Zimt
};
```

Dabei muss man beachten, dass nach einer Änderung des Konstruktors durch den Programmierer der Operator `instanceof` nicht mehr funktioniert. Deshalb sind wir in so einem Fall auf Duck Typing angewiesen, oder können in neueren Browsern stattdessen `isPrototypeOf`, wie im Abschnitt »Primitive Datentypen versus Objekte« auf Seite 135 erklärt, benutzen:

```
if (Gewuerz.isPrototypeOf(obj)) {
    // mit dem Gewürz etwas machen
}
```

Außerdem muss man aufpassen, denn `Object.create` gibt es nicht in Browsern, die die ES5 Spezifikation nicht implementieren. In diesem Spezialfall kann man ein `polyfill` benutzen.

Link: *https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/create*

Quelle: *http://stackoverflow.com/questions/9850892/should-i-use-polymorphism-in-javascript*

Quellen der Verwirrung durch das Typsystem

Falle 1: Automatische Typkonversion / Coercion

Mit der Typkonversion oder Coercion in JavaScript verhält es sich in der Praxis ähnlich wie mit dem automatischen Einfügen von Semikolons: Ursprünglich als Bequemlichkeit gedacht, führt sie mitunter zur Missverständnissen und Verwirrung. Auch zur Typkonversion gibt es eine lebhafte Diskussion: Ist sie ein unerwartetes, aus dem Hinterhalt zuschlagendes »cast« und soll man versuchen, sie zu vermeiden, um klareren Code zu schreiben? Oder ist sie ein Feature, und kann man sie geschickt für sich nutzen?

Die Hauptquelle der Verwirrung ist die Typkonversion, oder auch Coercion, bei den Vergleichsoperatoren `==` und `!=`, besonders wenn diese Operatoren in bedingten Anweisungen benutzt werden. Eine weitere Quelle der Verwirrung ist die Überladung von Operatoren wie `+`, welcher sich auf verschiedenen Kombinationen von Eingabetypen unterschiedlich verhält und einen unterschiedlichen Rückgabetyp hat. Doch zunächst zur Coercion.

Coercion passiert in JavaScript automatisch bei der Anwendung der Operatoren `==` und auch `!=`, und die Regeln sind im ECMAScript-Standard festgelegt. Wir schauen uns die Regeln aus der ECMAScript 5-Spezifikation an.

Ein Vergleich `x == y`, in dem `x` und `y` Werte sind, ergibt `true` oder `false`.

Der Vergleich wird folgendermaßen durchgeführt, in den Klammern zeigen wir passende Beispiele für (`x == y`):

```
Wenn Typ(x) == Typ(y), dann
 - wenn Typ(x) Undefined ist => true (undefined == undefined)
 - wenn Typ(x) Null ist => true (null == null)
```

```
- wenn Typ(x) Number ist
  1) wenn x NaN ist => false (NaN == ?)
  2) wenn y NaN ist => false (? == NaN)
  3) wenn x die gleiche Zahl wie y ist => true (23 == 23)
  4) wenn x +0 und y -0 ist => true (+0 == -0)
  5) wenn x -0 und y +0 ist => true (-0 == +0)
  6) sonst => false (23 == 1)
- wenn Typ(x) ein String ist
  1) und x und y genau die gleiche Folge von Buchstaben => true ("Curry" == "Curry")
  2) sonst => false ("Curry" == "Currz")
- wenn Typ(x) ein Boolean ist
  1) und x und y beide true oder beide false => true (true == true)
  2) sonst => false (false == true)
- wenn Typ(x) ein Objekt ist
  1) wenn x und y auf des gleiche Objekt verweisen => true (var xs =[], dann xs == xs)
  2) sonst => false (var xs = [], dann ist (xs == []) false und ([] == []) auch false)
Wenn x null und y undefined ist => true (null == undefined)
Wenn x undefined und y null ist => true (undefined == null)
Wenn Typ(x) Number ist und Typ(y) String ist => vergleiche x == ToNumber(y) (2 == "2")
Wenn Typ(x) String ist und Typ(y) Number ist => vergleiche ToNumber(x) == y ("2" == 2)
Wenn Typ(x) Boolean ist => vergleiche ToNumber(x) == y ((false == 0) und (true == 1),
aber nicht (true == 2))
Wenn Typ(y) Boolean ist => vergleiche x == ToNumber(y) (s.o.)
Wenn Typ(x) String oder Number ist, und Typ(y) Object ist => vergleiche x == toPrimitive
(y)
sonst => false
```

Neben diesen Regeln müssen wir noch beachten, wie die Konversionen `ToNumber` und `ToPrimitve` funktionieren.

Quelle: *http://webreflection.blogspot.ca/2010/10/javascript-coercion-demystified.html*

Das sind eine ganze Menge Coercion-Regeln, und sie sind komplex, und dadurch schwer zu verinnerlichen. Die Vergleichsoperatoren `==` und auch `!=` sind mit Vorsicht zu genießen, da wir bei ihrer Anwendung die Coercion-Regeln beachten müssen. Nur wenn wir uns sicher sind, von welchem Typ die Objekte sind, die wir vergleichen, können wir die Operatoren `==` und auch `!=` verwenden.

Daher ist es oft ratsam, den Operator `===` (bzw `!==`) zu benutzen, der ohne Coercion direkt vergleicht und dabei überprüft ob die Typen passen. Er macht ein Programm in vielen Fällen eindeutiger lesbar und schneller. Um den Operator `==` besser zu verstehen, kann man ihn anhand dieser Spezifikation selbst mit Hilfe von `===` implementieren.

Link: *http://www.scottlogic.co.uk/2010/10/implementing-eqeq-in-javascript-using-eqeqeq/*

Falle 2: Überladene Operatoren

Eine weitere Quelle der Verwirrung bezogen auf Typen in JavaScript ist, dass die arithmetischen Operatoren überladen und auf Argumente von vielerlei Typ anwendbar sind (Kapitel 8 im Abschnitt »Polymorphismus« auf Seite 137). Überladung in statisch typisierten Sprachen ist gut verständlich, da für jede Variable ihr Typ zur Compilezeit bekannt ist, und somit auch

klar ist, welche Version einer überladenen Funktion angewandt wird. In JavaScript sind durch die dynamische Typisierung die Typen der Variablen jedoch erst zur Laufzeit bekannt. Daher ist zu dem Zeitpunkt, zu dem man ein Programm schreibt, oft noch nicht klar, welche Version eines überladenen Operators zum Zuge kommt.

Fast alle arithmetischen Operatoren in JavaScript konvertieren ihre Argumente in den Typ `Number`. Nur der Operator + ist ein Sonderfall. Er ist überladen und kann je nach Eingabetyp entweder Zahlen addieren oder Strings konkatenieren, wie wir zu Anfang dieses Kapitels am Beispiel des HTML-Formulars schon gesehen haben.

Die Überladung erlaubt uns, zur Laufzeit zu entscheiden ob wir summieren oder konkatenieren wollen, dies ist aber nicht besonders nützlich, da so ein Fall selten vorkommt (vgl. jedoch `sum` und `unwords` aus Kapitel 4 im Abschnitt »Motivation für Reduce: sum und unwords« auf Seite 52). Vielmehr erschwert sie uns, beim Programmieren sicher zu sein, ob denn nun addiert oder konkateniert wird. Der Schlüssel zur Entscheidung liegt im Wissen, welche Typen vorliegen. Ist ein String im Spiel, wird in Strings umgewandelt und konkateniert. Im Fall von mehreren Operationen spielt auch die Klammerung, also die Assoziativität, eine Rolle. Betrachten wir die Portionen-Berechnung vom Beginn des Kapitels:

```
var nNeighbors = 4;
var nColleagues = 6;
nNeighbors + nColleagues + ' Portionen';
> 10 Portionen

'Portionen: ' + nNeighbors + nColleagues;
> Portionen: 46
```

Im ersten Fall werden zunächst die zwei Zahlen `nNeighbors` und `nColleagues` addiert, und dann wird Ergebniszahl `10` mit dem String `'Portionen'` kombiniert. Dazu wird die Ergebniszahl `10` in den String `'10'` umgewandelt und die Strings werden dann konkateniert. Im zweiten Fall wird zunächst der String `'Portionen: '` mit der Zahl `nNeighbors` kombiniert. Es wird also die Zahl `nNeighbors` in den String `'4'` umgewandelt, und die beiden Strings werden konkateniert. Der Ergebnis-String `'Portionen: 4'` wird dann im folgenden Schritt mit der Zahl `6` kombiniert. Die Zahl `6` wird in den String `'6'` umgewandelt, und mit dem vorherigen Ergebnis-String konkateniert.

Quelle: *http://unixpapa.com/js/convert.html*

Stärkere Typen für JavaScript

Wie wir in den vorangehenden Abschnitten gesehen haben, ist JavaScript gar nicht so »typenlos« wie in der Community oft behauptet wird, und besitzt durchaus einige Werkzeuge, um zu unterscheiden, mit was für Datentypen und Objekten wir es gerade zu tun haben. Außerdem erlaubt es durch seine Prototypen auch objektorientierte Ansätze und Polymorphismus. Das Problem ist vielmehr, dass die Sprache uns durch die automatische Typkonversion anhand des Kontextes nicht zur Typkorrektheit verpflichtet, und damit ein starker Kontrollmechanismus verloren geht.

Viele funktionale Programmiersprachen verfolgen hier den gegenteiligen Ansatz. Haskell ist eine stark typisierte (strongly typed) Sprache, da die Typen einer Variable in Haskell nicht umgewandelt werden (keine Coercion). Ausserdem ist Haskell statisch typisiert: Für jeden Ausdruck kann immer der passende Typ inferiert werden. Bei fehlender Typinformation ist diese vom Programmierer als Typdeklaration mitzuliefern. Wenn man die Typdeklaration explizit hinschreibt, kann man sehr gut sehen, welche Ein- und Ausgabetypen jede vorliegende Funktion hat. Dies vereinfacht das Verständnis komplexerer Programme enorm. Somit dient das Typkonzept neben der Fehlererkennung ausserdem sowohl der Dokumentation als auch der Abstraktion, da man beim Programmieren direkt »in Typen« denken kann, ohne sich um die Implementierung im Detail zu kümmern. Es gibt aber trotzdem die Möglichkeit, Funktionen auf unterschiedlichen Typen benutzbar zu machen, indem man Typklassen und Typpolymorphismus benutzt, wie im Abschnitt »Polymorphismus« auf Seite 137.

Typen dienen als Kontrollmechanismus, da die Typprüfung nicht zusammenpassende Datentypen erkennt. Bei statischen Typen findet die Typprüfung zur Compilezeit statt. Dabei wird rekursiv über Programm, Funktionen und Ausdrücke vorgegangen. Die Arbeit passiert auf dem abstrakten Syntaxbaum, der beim Parsen des Programms durch den Compiler oder Interpreter aufgebaut wurde. Der Typchecker verwendet die expliziten Typannotationen, wie Variablentypen und Funktionstypen und die impliziten Typinformationen, die aus dem Kontext erschlossen werden können. Wenn die Typen nicht zusammenpassen gibt es einen Typfehler.

```
[1, 2, 3].reduce(10);
> TypeError: 10 is not a function
```

Starke und statische Typen können also bei der Erkennung von Fehlern helfen, und erleichtern das Nachdenken über das Programm. Denn zum einen helfen sie bei der Dokumentation, etwa durch Beschreiben der Ein- und Ausgabe einer Funktion, und zum anderen bei der Abstraktion, da statt in der konkreten Implementierung in abstrakteren Datentypen gedacht werden kann. Es gibt mehrere Ansätze, das Typkonzept in JavaScript stärker als Kontrollmechanismus zu nutzen.

Klassenbasiertes Objektsystem und Polymorphismus mit Joose

Wenn man sich ein klassenbasiertes Objektsystem mit Polymorphismus für JavaScript wünscht, kann man das Framework Joose (Link: *http://joose.it/*) benutzen. Es bringt statt der Prototypen richtige Klassen für JavaScript mit, und das mit jeder Menge syntaktischem Zucker für Klassendeklaration, Konstruktion von Objekten und Vererbung. Durch die Klassenhierarchie bekommen wir objektorientierten Polymorphismus, wie wir ihn aus Sprachen wie Java kennen. Dabei kann der in Joose geschriebene Code direkt von modernen JavaScript-Laufzeitumgebungen ausgeführt werden. Ansonsten liegt der Schwerpunkt von Joose jedoch nicht auf den Typen, sondern eher auf dem objektorientierten Paradigma.

Spezifikation von Typen für Variablen mit TypeScript

Eine beliebte Möglichkeit, um das Typkonzept in JavaScript zu verstärken, ist TypeScript aus dem Hause Microsoft. TypeScript ist eine auf JavaScript basierende Sprache (Kapitel 10 im Abschnitt »TypeScript« auf Seite 175), um Typen erweitert, die in normales JavaScript übersetzt werden kann. Sie erlaubt die Spezifikation von Typen für Variablen, und ermöglicht dadurch statisches Typechecking. Die Angabe der Typen ist optional, und fehlende Typen werden durch Typinferenz ergänzt.

Link: *http://www.typescriptlang.org/*

ECMAScript 4

JavaScript ist auch als ECMAScript bekannt, die aktuelle Version ist ECMAScript 5. Im Jahre 2007 wurde ECMAScript 4 als Nachfolger des bis dahin benutzten ECMAScript 3 vorgestellt. ECMAScript 4 erweitert ECMAScript 3 um Typen, um robuster zu funktionieren. Mit Hilfe dieses Typsystem wird es ermöglicht, ein JavaScript Programm ohne Typen graduell zu einem Programm mit Typen zu entwickeln. Dies wird durch Typannotationen erreicht. Leider war ECMAScript 4 zu komplex für JavaScript-Implementatoren, und somit wurde es nie implementiert und benutzt. Einen guten Einblick bietet die Veröffentlichung »Evolutionary Programming and Gradual Typing in ECMAScript 4«, verfübar unter http://www.ecmascript.org/es4/spec/evolutionary-programming-tutorial.pdf

Alternative mit Typinferenz und Typchecking: Rust

Ausserdem kann in diesem Zusammenhang noch Rust (Link: *http://www.rust-lang.org/*) erwähnt werden, eine ebenfalls praktisch orientierte Multiparadigmen-Sprache, die noch in der Entwicklung ist. Rust ist eine Sprache für Client- und Server-Webprogrammierung. Dabei ist Rust von Beginn an für große Projekte und auch serverseitigen Code gedacht. Die Sprache ist angelehnt an C++, mit zusätzlichem Fokus auf Parallelisierung und automatischer Speicherverwaltung. Rust wurde beim Design ein Typsystem verpasst, welches Typinferenz für mit `let` deklarierte Variablen erlaubt. Außerdem unterstützt das Typsystem von Rust Typklassen, und dadurch wird das schreiben polymorpher Funktionen ähnlich wie in Haskell stark erleichtert.

Fazit zu JavaScripts Typsystem

Es ist einerseits schade, dass JavaScript diese Vorteile durch sein flexibles Typsystem aufgibt, andererseits ist die dadurch entstehende Freiheit beim Programmieren ja auch angenehm, da man die Typen nicht hinzuschreiben braucht und die automatische Typkonversion vieles einfacher macht, wenn man ihre Tücken verstanden hat. Es ist ein bisschen Geschmackssache, was einem besser gefällt.

Im Allgemeinen sind Programme in stark typisierten Programmiersprachen robuster. Die Deklaration der Typen einer Funktion ist in vielen funktionalen Sprachen der erste Schritt der algorithmischen Problemlösung, man spricht dann von typ-getriebener Programmierung. Wenn die Typen erstmal stimmen, schreibt sich das Programm fast von selbst. Besonders beim Entwickeln komplexer funktionaler Abstraktionen wie Funktionen höherer Ordnung (Kapitel 2 im Abschnitt »Abstraktion von einem Codeblock: Funktionen« auf Seite 24) oder der Abstraktion eines Problems in eine Monade (Kapitel 9) hilft es uns sehr, wenn wir erst einmal die Typen der Funktionen, ihre Typsignaturen, entwickeln. Für stark typisierte Programmiersprachen gilt die Aussage »Well-typed programs cannot go wrong«. Wenn ein Programm typkorrekt ist, kann es in viele Arten von Fehlerzuständen zur Laufzeit nicht mehr kommen, da diese zu Typfehlern geführt hätten.

Allerdings werden Programme in schwach typisierten Programmiersprachen meist mit mehr Tests versehen, und mit diesen werden nicht nur Typen, sondern auch das Verhalten getestet.

In der indischen Gesellschaft gab es bis vor einigen Jahrzehnten ebenfalls ein sehr strenges Typensystem: die Kastengesellschaft. Während strikte Typen und Trennung im Ingenieursbereich praktisch und hilfreich sind, bergen sie in einer Gesellschaft Potenzial für Konflikte und Ungerechtigkeit. Auch wenn das Kastensystem offiziell lange abgeschafft ist, halten sich Trennungen und Vorschriften, etwa bei der Heirat bis heute. Auch wenn in den Großstädten allein aus praktischen Gründen nicht mehr darauf geachtet werden kann, dass keine Mitglieder unterschiedlicher Kasten zusammen an einem Tisch essen – bei den meisten Imbissen würde das zu logistischen Problemen führen – wird die Oberschicht und die Angehörigen vieler »White-Collar-Berufe« immer noch oft von der Kaste der Bramahnen gestellt. Eng verbunden mit dem Kastensystem sind diverse Speisevorschriften. Das Chicken-Curry, das wir beim Inder an der Ecke begeistert essen, ist im strengen Typsystem eigentlich nur den Bramahnen vorbehalten, die als einzige neben Eiern und Milchprodukten auch Hühnerfleisch essen dürfen. Das folgende Chicken-Curry wäre noch vor einiger Zeit in Indien etwas für eine kleine Minderheit gewesen.

Im nächsten Kapitel wenden wir uns einem Konzept für Fortgeschrittene zu, der Monade. Ganz so grundlegend wie beim Hühnchencurry bleibt es hier nicht, aber es bleibt delikat.

Rezept: Chicken-Curry

Für 4 Personen

- 500 g Hähnchenbrustfilet
- 4 Zwiebeln
- 400 ml Kokosmilch
- 50 g Kokosraspel
- 2 EL Öl
- 1/2 TL Chiliflocken
- 2 TL Kurkuma
- 2 TL Korianderpulver
- 2 TL Kreuzkümmel, gemahlen
- 2 TL Currypulver
- 2 TL Zitronensaft

Hähnchenfilets waschen und schneiden. Die Filets dann in jeweils ca. 12 Stücke teilen, so dass die Stücke mundgerecht sind, eher etwas kleiner als zu groß.

Die Zwiebeln schälen und in Ringe schneiden.

Eine Pfanne ohne Fett erhitzen. Die Kokosraspeln darin ohne Fett rösten bis sie duften, ohne dass sie braun werden. Beiseite stellen.

Das Öl in einem Topf oder einem Wok erhitzen, dann die Zwiebeln zugeben und glasig dünsten lassen, dann die Kokosraspeln, Kurkuma und Curry zugeben, kurz mit anrösten lassen.

Dann die Kokosmilch zugießen, verrühren und mit Korianderpulver und Kreuzkümmel sowie den Chiliflocken würzen. Aufkochen lassen und dann Zitronensaft zugeben.

Das Hähnchen in die Sauce legen und ca. 20-30 Minuten mit halb offenem Deckel köcheln lassen, so dass die Sauce schön sämig wird, dabei immer wieder umrühren.

```
(λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
z)))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
```

KAPITEL 9

Kochen als Monade

Beim Kochen ist die zeitliche Abfolge der Arbeitsschritte wichtig, da die Arbeitsschritte aufeinander aufbauen. Im Rezept für Auberginen-Curry (Kapitel 4 im Abschnitt »Rezept: Baingan Bharta (Auberginen-Curry)« auf Seite 47) müssen wir zuerst den Ingwer mit den Tomaten pürieren, bevor wir das Ingwer-Tomaten-Püree verwenden können. Dieses brauchen wir dann für den nachfolgenden Schritt. Ein Rezept beschreibt zum einen die Zutaten und zum anderen in welcher Reihenfolge und wie genau die Zutaten verarbeitet werden.

Im Paradigma der funktionalen Programmierung entsprechen die Zutaten den Datenstrukturen und die Zubereitungsschritte den Funktionen, wie wir schon in Kapitel 6 festgestellt haben. Alle Zubereitungsschritte sind nur von ihren Argumenten abhängig. Es ist nicht festgelegt, in welcher Reihenfolge sie genau ausgeführt werden. Sobald sie ihre Argumente bekommen, legen sie los. Möchte man nun eine bestimmte Abfolge von Arbeitsschritten erzwingen, greift man auf einen Trick zurück. Man verkettet die Arbeitsschritte so, dass sie jeweils vom Ergebnis des vorherigen Schrittes abhängen. Das ganze verpackt man dann noch schön ordentlich in einen Behälter, eine sogenannte Monade.

Bevor wir über diese praktischen Behälter sprechen, schauen wir uns zunächst an, wie man Funktionen verketten kann. Funktionen oder auch beliebige Ausdrücke, welche wir ja mit anonymen Funktionen in Funktionen verwandeln können, lassen sich in JavaScript beliebig hintereinander ausführen. Im Programmcode werden die Ausdrücke, die wir in zeitlicher Abfolge ausführen möchten, durch Zeilenumbrüche oder Semikolons getrennt. In der funktionalen Programmierung haben wir diesen »Luxus« nicht, da das prozedurale Konzept der zeitlichen Abfolge auf Seiteneffekten aufbaut, die es in der rein funktionalen Programmierung nicht gibt. Wir müssen uns – wenn wir der reinen Lehre der funktionalen Programmierung auch in JavaScript folgen wollen – sozusagen ein Semikolon erst programmieren, um Ausdrücke in zeitlicher Abfolge ausführen zu können. Dies wollen wir nun versuchen.

Ein programmierbares Semikolon

Wie wir wissen, können wir in JavaScript mehrere Ausdrücke hintereinander ausführen, indem wir sie mit einem Semikolon abtrennen. Je nach Blickwinkel könnten wir allerdings auch sagen, wir verbinden die Ausdrücke mit einem Semikolon. In der folgenden Funktion passieren zwei Dinge nacheinander und werden durch ein Semikolon getrennt:

```
function foo (bar) {
  console.log(bar); return bar.length
}

foo("hallo");
> hallo
> 5
```

Was genau macht die Funktionalität des Semikolons hier aus? Zunächst wird der übergebene String ausgegeben. Danach wird die Länge des Strings zurückgegeben. Diese beiden Schritte trennen wir in JavaScript und vielen anderen Sprachen durch ein Semikolon ab. Versuchen wir nun, eine Funktion zu schreiben, die sich genau wie ein Semikolon verhält. Wir kapseln die Ausdrücke in anonyme Funktionen, die wir unmittelbar aufrufen. Dazu umgeben wir zunächst die beiden Ausdrücke mit anonymen Funktionen, damit wir später beliebige Funktionen an ihrer Stelle benutzen können. Wir müssen außerdem noch ein zusätzliches `return` einfügen, damit die Rückgabe weiterhin funktioniert.

Umgeben mit anonymen Funktionen:

```
function foo (bar) {
  (function () { console.log(bar); })();
  return (function () { return bar.length; })();
}
```

Wir wollen eine Gesamtfunktion bauen, die das Semikolon ersetzt. Damit unsere Gesamtfunktion sich später rekursiv verschachteln lässt, wozu wir weiter unten kommen, umschließen wir auch die beiden Ausdrücke, die wir verknüpfen wollen, mit einer anonymen Funktion, so dass der Rückgabewert von `foo` eine Funktion ist.

```
function foo (bar) {
  return function () {
    (function () { console.log(bar); })();
    return (function () { return bar.length; })(); }();
}
```

Um nun in `foo` von den konkreten Ausdrücken, die wir verketten möchten, zu abstrahieren, wollen wir die beiden Funktionsausdrücke, die verkettet werden sollen, als Argumente übergeben. Wir beginnen mit der ersten Funktion, die den Eingabewert auf die Konsole ausgibt:

```
var a = function (xs) { console.log(xs); };

function foo (bar, fun_a) {
  return function () {
```

```
    (function (xs) { fun_a.call(this, xs); })(bar);
    return (function () { return bar.length; })(); }();
}
```

Aus Kapitel 3 im Tipp »typeof ...« auf Seite 49 wissen wir bereits: Wollen wir eine Funktion, wie `fun_a`, explizit mit bestimmten Parametern aufrufen, so können wir entweder `call` oder `apply` benutzen. Sowohl `call` als auch `apply` bekommen `this` als erstes Argument, unterscheiden sich aber in den weiteren Argumenten, die sie erwarten. Die Funktion `call` bekommt neben `this` direkt die Argumente für den Funktionsaufruf als weitere Argumente übergeben, `apply` jedoch erwartet die Argumente für den Funktionsaufruf in Form eines Arrays. Da wir nur ein Argument `xs` haben, benutzen wir `call` mit den Argumenten `this` und `xs`.

```
foo("hello", a);
> hello
> 5
```

Als nächstes schreiben wir unsere Verkettungsfunktion `foo` so um, dass sie auch den zweiten Funktionsausdruck, der für die Rückgabe verantwortlich ist, als Argument empfangen kann.

```
var b = function (ys) { return ys.length; };

function foo (bar, fun_a, fun_b) {
  return function () {
    (function (xs) { fun_a.call(this, xs); })(bar);
    return (function (ys) { return fun_b.call(this, ys); })(bar); }();
}

foo("hello", a, b);
> hello
> 5
```

Wir können also die Ausdrücke `a` und `b` verketten, und erreichen damit den gleichen Effekt wie ein Semikolon: Der erste Ausdruck wird ausgeführt, und das Ergebnis verworfen. Dann wird der zweite Ausdruck ausgeführt und dessen Ergebnis mittels `return` zurückgegeben. Die Funktion `foo` ist hierbei `mbind`, das sogenannte monadische bind geworden. Wir benennen deshalb `foo` in `mbind` um.

```
var mbind = foo;
```

Die Funktion `mbind` erinnert an die Funktionskomposition `(f ∘ g)(x) = (f(g(x)))` aus Kapitel 3 im Abschnitt »Implizite Argumente, Komposition und Point-Free-Style« auf Seite 42, nur war es dort so, dass die Rückgabe der Funktion `g` an die Funktion `f` weitergereicht wurde. Dies ist bei der Funktion `mbind` anders, denn sie verwirft ja das Ergebnis der ersten Berechnung, und führt diese also nicht aufgrund des Rückgabewertes, sondern nur wegen der Seiteneffekte aus. In Kapitel 4 haben wir ein ähnliches Funktionspaar kennengelernt: `map` und `forEach` wenden beide eine Funktion auf jedem Array-Element an. Dabei gibt `map` ein Array der Resultate zurück, aber `forEach` gibt nichts zurück, sondern wendet die Funktion nur wegen ihrer Seiteneffekte an (Kapitel 4, »Spezialität des Hauses«, Seite 53).

Die Verkettung mit `mbind` funktioniert auch für beliebig viele Ausdrücke. Nehmen wir eine dritte Funktion `c` hinzu, die ihr Argument mit Ausrufezeichen versehen ausgibt:

```
var c = function (zs) { console.log(zs + "!"); };
```

Bei der Verkettung mehrerer Funktionen müssen bereits zusammengesetzte Funktionen in eine anonyme Funktion verpackt werden, damit `mbind` auf ihnen noch einmal `call` aufrufen kann. Diese anonyme Funktion werden wir aber später im Kapitel noch los.

```
mbind("Dies funktioniert", a, function () { return mbind("wie ein Semikolon", c, b); });
> Dies funktioniert
> wie ein Semikolon!
> 17

// Umbruch für bessere Lesbarkeit bei der Analyse, macht so in node.js jedoch Probleme
mbind("Dies funktioniert", a, function () { return
// Ausführen von a mit "Dies funktioniert"; Verwerfen des Ergebnisses
mbind("wie ein Semikolon", c,
// Ausführen von c mit "wie ein Semikolon"; Verwerfen des Ergebnisses
      b); });
// Ausführen von b, Rückgabe des Ergebnisses
```

Wir sehen, dass zuerst `a` auf sein Argument angewandt und das Ergebnis verworfen wird, dann wird `c` auf sein Argument angewandt und das Ergebnis verworfen. Schließlich wird `b` aufgerufen, welches für das zurückgegebene Ergebnis der Kette von Berechnungen verantwortlich ist.

Dieses Vorgehen entspricht genau unserem Ziel, wir haben uns mit `mbind` eine Funktion programmiert, die den Effekt eines Semikolons in einer prozeduralen Programmiersprache hat! Und dazu haben wir nur Methoden der funktionalen Programmierung benutzt, auch wenn das aus unserer Perspektive als JavaScript-Programmierer nicht direkt offensichtlich ist. Unsere `mbind`-Funktion könnten wir nämlich genauso in einer rein funktionalen Programmiersprache, wo es keine Semikolons gibt, hinschreiben, denn die verkettete Ausführung wird allein durch die Implementierung der Funktion mbind gewährleistet.

Soeben haben wir den Kern unserer ersten Monade kennengelernt! Dies ist noch keine Monade im traditionellen Sinn, aber wenn wir sie noch etwas anpassen, werden wir später sehen, dass es sich um ein mathematisches Objekt aus der Kategorientheorie handelt, für das die sogenannten Monaden-Gesetze gültig sind (Kapitel 9 im Abschnitt »Die gefürchteten Monaden-Gesetze« auf Seite 166). Was macht unsere Monade? Sie kombiniert zwei Dinge zu einer Sequenz. Unter der Haube machen das alle Monaden so.

Monaden als Behälter: Dinge, Kochtöpfe und Monaden

Moment mal, alle Monaden kombinieren Dinge zu Sequenzen? Was für Dinge? Stellen wir uns vor, ein Ding wäre in diesem Fall ein Behälter, etwa ein Kochtopf. Man beginnt mit Topf eins, und sobald man das Ergebnis aus Topf eins hat, kann man Kochtopf zwei »durchführen«.

Dies ist eine andere Sichtweise auf das, was wir oben bereits programmiert haben: Monaden lassen sich mit `mbind` kombinieren, so dass man nach dem Durchlaufen der Sequenz von Monaden das Gesamtergebnis zurückgegeben bekommt.

Wie sieht `mbind` nun aus, wenn wir uns vorstellen, dass wir Behälter oder Kochtöpfe zu einer Sequenz verknüpfen möchten? Definieren wir uns einen Kochtopf mit Inhalt. Der Inhalt des Topfes, `contents`, ist in diesem Fall der Einfachheit halber eine Zahl. Und der Rechenschritt, `step`, erhöht die Zahl im Topf um eins.

```
function Pot (contents) { this.contents = contents; }
var a = new Pot(1);
var b = new Pot(2);
```

Das können wir auch ausdrücken als:

```
function step (i) { return new Pot(i + 1); }
a = new Pot(1);
b = step(a.contents);
```

Wir beginnen mit einem Kochtopf, und benutzen dann einen Wert aus dem Inneren des Kochtopfes, um einen neuen Topf mit dem richtigen Inhalt zu konstruieren. Dies ist ein Muster, das in der Programmierung oft vorkommt. Wir haben einen Wert, und benutzen ihn, um einen neuen Wert zu berechnen. Würde man nur die Berechnung hinschreiben, hätte man:

```
function step (i) { return i + 1; }

a = 1;
b = step(a);
```

Der Unterschied zwischen beiden Versionen ist, dass die erste Version die Berechnung in einen Kochtopf verpackt, und die zweite Version nackte Werte zum Berechnen verwendet.

Nehmen wir an, dass wir einen guten Grund haben, alles in einen Kochtopf zu verpacken. Die Gründe können vielfältig sein, aber der Hauptgrund ist, dass wir innerhalb des Kochtopfs noch zusätzliche Logik verstecken können, die automatisch etwas mit dem Inhalt im Kochtopf macht. Zum Beispiel können wir einfach alle Zutaten im Topf aufsummieren. Der Kochtopf merkt sich dann die Summe der Zahlen, die in ihm gelagert werden und aktualisiert die Summe in jedem Schritt. Ebenso wäre es an dieser Stelle möglich, im Topf eine Liste von Zutaten immer sortiert zu halten.

Die Frage ist nun, ob wir noch einen besseren Weg entwickeln können, um von Topf `a` nach `b` zu gelangen? Wir möchten das Monaden-Muster als ein generelles Werkzeug formulieren.

Unser Ziel ist eine Funktion, die den Wert aus einem Kochtopf herausnimmt, und dann die nächste Funktion auf dessen Inhalt (Wert) aufruft und schliesslich den Wert dieses Aufrufs zurückgibt. Der Rückgabewert wird wiederum ein Kochtopf sein, aber ein neu erzeugter mit anderem Inhalt.

Ganz im Stil der objektorientierten Programmierung in JavaScript definieren wir eine Methode auf dem Kochtopf-Prototyp:

```
Pot.prototype.mbind = function (f) { return f.call(this, this.contents); };
```

Wenn wir also einen Kochtopf haben, können wir seinen Wert herausnehmen, und damit einen neuen Kochtopf berechnen, alles bequem in einem einzigen Schritt:

```
var a = new Pot(1);
> { contents: 1 }

var b = a.mbind(step);
> { contents: 2 }

var c = b.mbind(step);
> { contents: 3 }

c.contents;
> 3
```

Es fällt auf, dass diese Version viel sauberer als unsere erste Version ist, obwohl das gleiche berechnet wird.

Das Monaden-Muster

Jedes Mal, wenn man eine Berechnung mit einem Objekt beginnt, dieses auseinandernimmt, und wieder ein Objekt vom gleichen Typ berechnet, hat man eine Monade. Wer nun denkt »aber das ist doch in fast allen meinen Programmen so«, liegt richtig. Monaden sind überall!

Um zu verstehen, warum das so ist, schauen wir uns noch einmal unsere Kochtopf-Monade an. Was macht sie genau zur Monade?

Beginnen wir mit Schritt eins der Berechnung.

```
a = new Pot(1);
```

Wir können also einen Wert in ein Kochtopf verpacken. Ein objektorientierter Entwickler sagt sofort, aha, ein Konstruktor! Auf Monaden sagt man dazu allerdings 'die `unit`-Funktion'. In Haskell sagt man `return`, das ist in JavaScript nicht so gut, da `return` ja bereits ein vordefiniertes Schlüsselwort ist. Wie auch immer wir es nennen, es ist eine Funktion vom Typ `A -> Kochtopf mit Inhalt A`, also eine Funktion, die irgendeinen Wert mit dem Typ `A` nimmt, und ihn in einen neuen Kochtopf verpackt.

Die weiteren Schritte unserer Berechnung haben alle die gleiche Form.

```
b = a.mbind(step);
```

Außerdem haben wir die ausgefuchste Funktion `mbind`, die in unseren Kochtopf `a` hineinschaut. Der Funktion `mbind` übergeben wir die Funktion `step`, die den Wert aus Kochtopf `a` verwendet, um einen neuen Kochtopf `b` zu konstruieren, der das Ergebnis zurückgibt. In Scala wird diese Funktion `flatMap` genannt, und in Haskell `>>=` oder `bind`. Der Name ist egal, wichtig ist, dass `bind` zwei Kochtöpfe miteinander zu einer Sequenz verknüpft, »wir beginnen mit einem Kochtopf und benutzen seinen Inhalt (Wert), um den nächsten Kochtopf zu berechnen.«

Bind auf Monaden: Wir beginnen mit einem Objekt, und benutzen seine Eigenschaften, um das nächste Objekt zu berechnen.

In Kurzschreibweise besteht das Monaden-Muster also aus zwei Funktionen, die jeweils eine Monade zurückgeben.

```
1) function unit(value)
2) function mbind(monad, function(value))
```

Und das war es auch schon. Wieso gibt es also so viel Gerede über Monaden? Warum nennen wir die Methode nicht einfach »benutze ein Ding um ein anderes zu berechnen«-Muster? Erstens ist das etwas wortreich, und zweitens wurden Monaden zuerst von Mathematikern beschrieben. Monaden bieten uns einen praktischen Einstieg in das ansonsten eher theoretische mathematische Gebiet der Kategorientheorie. Aus dieser stammt das Konzept der Monaden und ihrer allgemeineren Verwandten, den (applikativen) Funktoren. Im Programmier-Jargon gesprochen, ermöglicht uns ein Funktor, eine normale Funktion, die auf normalen Typen wie zum Beispiel Zahlen operiert, auf ausgefuchsteren Typen zu benutzen. Dies wird von der Funktion `fmap` erledigt, die »eine Funktion über einen Typ mappt«, also die Umwandlung der normalen Funktion in eine Funktion auf den ausgefuchsteren Typen vornimmt. Mit dem Hintergrundwissen aus der Kategorientheorie ist jede Monade auch ein Funktor, und Monaden können nicht nur wie hier gezeigt mit `bind` und `return` konstruiert werden, sondern auch mit `return`, `join` und `fmap`, wobei diese jedoch Operationen auf Funktoren sind. Bei der praktischen Verwendung des Monaden-Musters ist dies unwichtig. Es ist aber eher weiterführende Information für Enthusiasten.

Monaden links, Monaden rechts, Monaden überall

Bisher haben wir nur zwei Beispiele für Monaden gesehen: die sequenzielle Ausführung von Funktionen, und die Kochtopf-Monade, die einen Zustand im Inneren des Topfes hütet. Sie wird auch die State-Monade genannt. Da Monaden aber überall auftreten, schauen wir uns weitere Beispiele an. Unter Umständen kommt uns hierbei unsere (prozedurale) JavaScript-Brille in die Quere.

IO-Monade

In JavaScript arbeitet jede Zeile eines Programms bereits automatisch in der IO-Monade. Diese Monade wird in rein funktionalen Sprachen wie Haskell oder Miranda unbedingt benötigt, wenn Ein- und Ausgabe in einem Programm erfolgen soll. Denn die zur Ein- und Ausgabe nötigen Seiteneffekte sind in der IO-Monade gekapselt und nur in dieser überhaupt möglich. Dieser Umstand erklärt auch, warum in rein funktionalen Sprachen Monaden eine so grosse Bedeutung haben.

Man kann in Haskell selbst entscheiden, ob man innerhalb der IO-Monade arbeitet, dabei Seiteneffekte riskiert, aber Ein- und Ausgabe bekommt, oder ob man lieber rein funktional programmiert und nicht von Seiteneffekten überrascht werden kann, aber auf Ein- und Ausgabe verzichten muss. In JavaScript gibt es diese Unterscheidung nicht, da es als Multi-Paradigmen-Sprache aufgrund seiner prozeduralen Anteile nicht rein funktional arbeitet. Außerdem wurde JavaScript ursprünglich entwickelt, dazu die HTML-Ausgaben des Browsers zu manipulieren, da wäre eine Programmierung ohne Ein- und Ausgabe schwierig bis sinnlos. Die IO-Monade wird also in einer »unreinen« Sprache wie JavaScript gar nicht benötigt, weshalb wir sie hier nicht betrachten.

Wir können aber einzelne Funktionen im rein funktionalen Stil schreiben, wie etwa unser `mbind` von oben. Es kann daher nützlich sein, sich auch in JavaScript auf Seiteneffekt-Freiheit zu beschränken, um klarere Lösungen zu formulieren. Andere klassische Monaden sind deshalb für JavaScript-Programmierer durchaus interessant.

Writer-Monade

Die Writer-Monade kann zur automatischen Debug-Ausgabe verwendet werden. Wir erinnern uns, Monaden ermöglichen mittels `mbind` eine besondere Variante der Funktionskomposition. Angenommen, wir haben eine Funktion `sine`, die den Sinus einer Zahl unter Rückgriff auf die Bibliothek Math berechnet, und eine andere Funktion `cube` kubiert eine Zahl.

```
var sine = function (x) { return Math.sin(x); };
var cube = function (x) { return x * x * x; };
```

Beide Funktionen bekommen eine Zahl als Argument und geben eine Zahl zurück. Wie wir aus Kapitel 3 im Abschnitt »Implizite Argumente, Komposition und Point-Free-Style« auf Seite 42 wissen, können wir Funktionen in einer Kette nacheinander anwenden, wenn die Rückgabe der ersten Funktion zum Argument der zweiten Funktion passt.

```
var sineCubed = cube(sine(x));
```

Wir haben in Kapitel 3 im Abschnitt »Implizite Argumente, Komposition und Point-Free-Style« auf Seite 42 bereits eine Funktion zur Funktionskomposition `z(x) = (f ○ g)(x) = (f(g(x)))` entwickelt. Stark vereinfacht und nur für ein Argument definiert könnten wir sie auch so schreiben:

```
var compose = function (f, g) {
  return function (x) {
    return f(g(x));
  };
};
```

Nun können wir mit dieser Funktion die Funktionen `sine` und `cube` zusammensetzen.

```
var sineOfCube = compose(sine, cube);
sineOfCube(23);
> 0.3786593486919014
```

Wenn wir nun beim Programmieren beide Funktionen debuggen möchten, und ihre Aufrufe in der JavaScript-Konsole festhalten wollen, könnten wir dies so machen:

```
var cube = function (x) {
  console.log('cube was called.');
  return x * x * x;
};
```

Aber in einer rein funktionalen Programmiersprache dürfen wir so gar nicht vorgehen, denn eine Ausgabe per `console.log` ist weder Argument noch Rückgabe der umgebenden Funktion, sondern ein Seiteneffekt. Wollen wir die Information über den Aufruf trotzdem festhalten, müssen wir sie als Teil des Rückgabewerts übergeben. Wir modifizieren deshalb unsere Funktionen, so dass sie ein Paar zurückgeben, bei uns repräsentiert als ein Array mit 2 Elementen: Dem Ergebnis, und der zusätzlichen Debug-Information.

```
var sine = function (x) {
  return [Math.sin(x), 'sine wurde aufgerufen.'];
};

var cube = function (x) {
  return [x * x * x, 'cube wurde aufgerufen.'];
};
```

Leider lassen sich die Funktionen nun aber nicht mehr kombinieren!

```
cube(3);
> [ 27, 'cube wurde aufgerufen.' ]
compose(sine, cube)(3);
> [ NaN, 'sine wurde aufgerufen.' ]
```

Gleich zwei Fehler kommen hier zusammen: Die Funktion `sine` versucht fälschlicherweise den Sinus eines Arrays zu berechnen, und dabei kommt `NaN` heraus. Außerdem verlieren wir die Debug-Information aus dem Aufruf von `cube`. Wir erwarten jedoch, dass die Funktionskomposition uns das Ergebnis von `sine(cube(x))` zurückgibt, und dazu einen String, der die Information enthält, dass `cube` und `sine` aufgerufen wurden.

Die Ausgabe unserer Komposition von `sine` und `cube`, `compose(sine, cube)(3)`, soll so aussehen:

```
[ 0.956375928404503, 'cube wurde aufgerufen. sine wurde aufgerufen.' ].
```

Da der Rückgabetyp von `cube` ein Array ist, aber der Argumenttyp von `sine` eine Zahl, müssen wir unsere `compose`-Funktion für »Funktionen mit Debug-Information« noch etwas anpassen. Sie soll in jedem Schritt den Rückgabetyp der jeweiligen Funktion auseinandernehmen, und wieder sinnvoll für die nächste zusammensetzen.

```
var composeDebuggable = function (f, g) {
  return function (x) {
    var gx = g(x),      //cube(3) -> [ 27, 'cube wurde aufgerufen.' ]
        y  = gx[0],     //27
        s  = gx[1],     //'cube wurde aufgerufen.'
        fy = f(y),      //sine(27) -> [ 0.956375928404503, 'sine wurde aufgerufen.']
        z  = fy[0],     //0.956375928404503
        t  = fy[1];     //'sine wurde aufgerufen.'
    return [z, s + " " + t];
  };
};

composeDebuggable(sine, cube)(3);
> [ 0.956375928404503, 'cube wurde aufgerufen. sine wurde aufgerufen.' ]
```

Die Funktion `composeDebuggable` setzt zwei Funktionen zusammen, die jeweils eine Zahl als Argument bekommen, und ein Zahlen-String-Paar zurückgeben. Man sagt auch, beide Funktionen haben den Typ `Number -> (Number, String)`. Dabei entsteht wiederum ein Funktion vom gleichen Typ. Die zusammengesetzte Funktion bekommt ebenfalls eine Nummer als Argument und gibt ein `(Number, String)`-Paar zurück. Das bedeutet, sie kann wiederum mit anderen Funktionen dieses Typs rekursiv mit `composeDebuggable` zusammengesetzt werden.

Vor dem Hinzufügen der Debug-Information hatten unsere Funktionen hingegen den Typ `Number -> Number`, sie bekamen eine Zahl und gaben eine Zahl zurück. Funktionen mit gleichem Ein- und Ausgabetyp sind direkt mit der normalen Funktionskomposition kombinierbar.

Was wäre nun, wenn wir der Funktion `cube` mit Debug-Information auch einen gleichen Ein- und Ausgabetyp bescheren könnten? Dann könnten wir auch hier die normale Komposition verwenden. Heisst das, wir müssen `sine` und `cube` von Hand anpassen oder gibt es einen Trick? Wir könnten doch eine Funktion schreiben, die dies erledigt. Wie wir sehen, ist dies die `mbind`-Funktion unserer Monade.

```
var mbind = function (f) {
  return function (tuple) {
    var x  = tuple[0],
        s  = tuple[1],
        fx = f(x),
        y  = fx[0],
        t  = fx[1];
    return [y, s + " " + t];
  };
};
```

Benutzen wir `mbind`, um unseren Funktionen die richtigen Typen zu verpassen, so dass sie mit `compose` zusammengesetzt werden können.

```
compose(mbind(sine), mbind(cube))([3, '']);
> [ 0.956375928404503, 'cube wurde aufgerufen. sine wurde aufgerufen.' ]
```

Das ist aber noch etwas umständlich, denn alle Funktionen arbeiten nun mit (Number, String)-Paaren als Argument. Wir wollen aber eigentlich nur die Zahl übergeben und die Debug-Information soll automatisch weitergereicht werden. Deshalb brauchen wir eine weitere Funktion, welche die Werte vom Typ Number in die richtigen Typen (Number, String), auf denen die Monade intern arbeitet, konvertiert. Es ist die Funktion unit, die wir auch schon aus anderen Monaden kennen. In diesem Fall konstruiert sie ein Paar aus einer gegebenen Zahl und einem neuen, leeren String.

```
var unit = function (x) { return [x, '']; };

var f = compose(mbind(sine), mbind(cube));
f(unit(3));
> [ 0.956375928404503, 'cube wurde aufgerufen. sine wurde aufgerufen.' ]

// oder auch
compose(f, unit)(3);
> [ 0.956375928404503, 'cube wurde aufgerufen. sine wurde aufgerufen.' ]
```

Mit der Funktion unit können wir auch jede andere Funktion, die auf einer Zahl arbeitet, in eine Funktion mit Debug-Information konvertieren.

```
var round = function (x) { return Math.round(x); };

var roundDebug = function (x) { return unit(round(x)); };
```

Diese Umwandlung von einer »normalen« Funktion in eine Funktion mit Debug-Information lässt sich wiederum in eine Funktion namens lift abstrahieren. Die Funktion lift bekommt eine Funktion die auf Zahlen arbeitet, und gibt eine Funktion zurück, die eine Zahl als Eingabe bekommt, und ein (Number, String)-Paar zurückgibt.

```
var lift = function (f) {
  return function (x) {
    return unit(f(x));
  };
};

// vereinfacht:
var lift = function (f) { return compose(unit, f); };
```

Probieren wir dies mit unseren Funktionen aus:

```
var round = function (x) { return Math.round(x); };

var roundDebug = lift(round);

var f = compose(mbind(roundDebug), mbind(sine));

f(unit(27));
> [ 1, 'sine wurde aufgerufen.' ]
```

Wir sehen, dass es drei Arten der Abstraktion gibt, die beim Bau von Monaden immer wieder auftreten: Die Funktion lift wandelt eine normale Funktion in eine monadische Funktion um (liftet eine Funktion in die Monade). Die Funktion bind wandelt eine monadische Funktion in eine Form um, die sich mit compose zusammensetzen lässt. Die

Funktion `unit` wandelt einen einfachen Wert durch Verpacken in einen Container in das Format um, auf dem die Monade intern arbeitet (liftet einen Wert in die Monade).

Die eben beschriebene Monade wird auch Writer-Monade genannt.

Quelle: *http://blog.jcoglan.com/2011/03/05/translation-from-haskell-to-javascript-of-selected-portions-of-the-best-introduction-to-monads-ive-ever-read/*

List-Monade

Ein Problem, was vielen schon begegnet ist, ist die Entscheidung, ob eine Funktion ein einzelnes Element als Eingabe akzeptieren soll, oder aber eine Liste von Argumenten. Die Fallunterscheidung erfolgt meistens nach dem immer gleichen Muster, mit einer `for`-Schleife um die Funktion. Leider beeinflusst diese Änderung die Art, nach der man solche Funktionen hinterher zusammen setzen kann. Stellen wir uns vor, wir haben eine Funktion, die ein Element des DOM-Trees eines HTML-Dokuments bekommt, und alle Kinder von diesem Element in Form eines Arrays zurückgibt. Die Typsignatur dieser Funktion besagt, dass sie ein einzelnes HTML-Element bekommt, und ein Array von HTML-Elementen zurückgibt. Um die Funktionen dieses Abschnitts interaktiv auf einer Webseite auszuprobieren, benutzen wir die JavaScript-Developer-Konsole im Web-Browser Chrome und besuchen *http://mylittlewiki.org/wiki/G1_Ponies*.

```
var children = function (node) {
  var children = node.childNodes
  var xs = [];
  for (var i = 0; i < children.length; i++) {
    xs[i] = children[i];
  }
  return xs;
};

var heading = document.getElementsByTagName('h4')[0];
children(heading);
> [ #text, <span class="mw-headline" id="Collector_Ponies">…</span> ]
```

Angenommen, wir möchten die Enkel der Überschrift herausfinden, also die Kinder der Kinder der Überschrift. Frisch drauf losgeschrieben, scheint dies eine gute Lösung zu sein:

```
var grandchildren = compose(children, children);
```

Aber die Funktion `children` hat keinen gleichen Ein- und Ausgabetyp, also können wir sie nicht normal mit `compose` verketten. Würden wir die Funktion `grandchildren` von Hand schreiben, könnte sie so aussehen:

```
var grandchildren = function (node) {
  var output = [];
  var kids = children(node);
  for (var i = 0; i < kids.length; i++) {
```

```
    output = output.concat(children(kids[i]));
  }
  return output;
};
```

Wir sammeln alle Kinder der Kinder des Eingabeknotens auf, und verketten dabei die Ergebnislisten, um eine flache Liste zu bekommen, die alle Enkel enthält. Aber diese Schreibweise ist nicht besonders elegant, und enthält eine Menge umständlichen Code, der nur daher kommt, dass wir auf Listen arbeiten, und nichts mit dem Problem zu tun hat, das wir lösen möchten. Könnten wir einfach nur zwei listenverarbeitende Funktionen mit `compose` zusammensetzen, wäre unser Problem gelöst.

Erinnern wir uns an die vorherigen Beispiele, dann wird klar, wir brauchen eine Funktion `mbind`, die die Funktion `children` in eine Form bringt, die sich mit `compose` zusammensetzen lässt. Außerdem brauchen wir eine Funktion `unit`, die das ursprüngliche Argument, die Überschrift, in den richtigen Typ für die Monade verwandelt.

Das Problem ist, dass unsere Funktion ein HTML-Element als Argument bekommt, aber eine Liste von Elementen zurückgibt, also sollte unsere Umwandlung ein einzelnes Element in eine Liste verwandeln können und umgekehrt. Ganz genau genommen arbeiten wir hier auf Arrays und nicht auf Listen, aber dieser Unterschied spielt für das Vorgehen keine Rolle. Auch, dass in unserem Beispiel die Listenelemente HTML-Elemente sind, ist unwichtig. Wir können uns eine Liste von Elementen mit variablem Typ vorstellen (vgl. Kapitel 5 im Abschnitt »Listen als algebraische Datentypen, Pattern Matching« auf Seite 79, Kapitel 8 im Abschnitt »Polymorphismus« auf Seite 137). Die Funktion `unit` nimmt ein Element, und gibt eine Liste zurück die das Element enthält, und die Funktion `bind` nimmt eine Funktion, die ein Element bekommt und eine Liste von Elementen ausgibt, und wandelt diese in eine Funktion mit einer Liste als Ein- und Ausgabe um.

```
var unit = function (x) { return [x]; };

var bind = function (f) {
  return function (list) {
    var output = [];
    for (var i = 0, n = list.length; i < n; i++) {
      output = output.concat(f(list[i]));
    }
    return output;
  };
};
```

Nun können wir die Funktion `children` mit `compose` zweimal zusammengesetzt ausführen, um die Enkel zu erhalten:

```
var grandchildren = compose(bind(children), bind(children));

grandchildren(unit(heading));
> [<a href="/wiki/Category:Collector_Ponies" title="Category:Collector Ponies">Collector
Ponies</a>]
```

Soeben haben wir eine Listen-Monade entwickelt. Mit dieser Monade können wir Funktionen verketten, die ein einzelnes Element in eine Liste von Elementen überführen.

Quelle: *http://blog.jcoglan.com/2011/03/05/translation-from-haskell-to-javascript-of-selected-portions-of-the-best-introduction-to-monads-ive-ever-read/*

Abstrakte Datentypen als Monaden

Generell lassen sich abstrakte Datentypen gut als Monade realisieren, so dass die Implementierung versteckt wird, nur das Interface nach außen sichtbar ist, und dadurch bestimmte Eigenschaften für die Daten garantiert werden können. Auf diese Weise kann man etwa eine immer geordnete Liste implementieren, oder einen »immutable« Stack, bei dem die Werte auf dem Stack nicht verändert werden können (Link: *http://igstan.ro/posts/2011-05-02-understanding-monads-with-javascript.html*).

Auch die in Haskell klassische Monade `Maybe` ist interessant, denn sie erlaubt, Daten auf »nicht vorhanden« oder `null` zu setzen, wobei dies auch explizit durch zwei Varianten im Datentyp repräsentiert wird. Die Konstruktor-Variante `Just a` repräsentiert Daten von einem beliebigen Typ `a`, und die Variante `Nothing` repräsentiert die »leeren« Daten. Diese Information kann bei der Typprüfung einfliessen und ermöglicht auch Pattern Matching auf den Daten, also eine Fallunterscheidung anhand der Konstruktor-Varianten.

Die beiden Konstruktoren `Nothing` und `Just a` stellen wir in JavaScript als Funktionen dar, die sofort ausgeführt werden. Diesen Trick haben wir ja bereits mehrfach benutzt.

```
var Nothing = (function () {
    return {
        mbind: function () { return Nothing; },
        toString: function () { return "Nothing"; }
    };
})();

var Just = (function () {
    return function (a) {
        return {
            mbind: function (f) { return f(a); },
            toString: function () { return "Just " + a; }
        };
    };
})();
```

Nun können wir Variablen vom Typ `Maybe` erzeugen. Die `unit`-Funktion haben wir wie im Kochtopf-Beispiel weggelassen, denn sie verpackt ja einfach nur einen Wert in den monadischen Typ `Maybe`, was wir direkt mit dem Konstruktor erledigen können.

```
Just(1);
> { mbind: [Function], toString: [Function] }

Just(1).toString();
> "Just 1"
```

Und auch eine Funktion `f`, die einen Wert vom Typ `Maybe` zurückgibt. Sie könnte zum Beispiel einen Wert inkrementieren, aber nur, solange dieser kleiner als drei ist.

```
var f = function (x) {
  return x < 3 ? Just(x + 1) : Nothing;
};

Just(1).mbind(f);
> Object {mbind: function, toString: function}

Just(1).mbind(f).toString();
> "Just 2"

Just(3).mbind(f).toString();
> "Nothing"
```

Auf diese Weise lässt sich eine verkettete Berechnung realisieren, die in einem Fehlerfall `Nothing` zurückgibt, und im normalen Fall das Ergebnis der Kette von Berechnungen.

Bibliotheken als Monaden

Mit genügend Abstraktionsvermögen kann auch die JavaScript-Bibliothek jQuery als Beispiel einer monadischen Bibliothek gesehen werden. Die `unit`-Funktion ist in diesem Fall der Operator `$`, er liftet sein Argument in die jQuery-Monade. Die `bind`-Funktion wird durch die Kette von Methodenaufrufen auf dem gelifteten Wert repräsentiert, in der jeder Aufruf einen anderen Wert zurückgibt, der wiederum in der jQuery-Monade verpackt ist.

Zusammenhang zwischen Continuation-Passing-Style und Monaden

Programmieren mit Monaden erinnert stark an Progammieren im Continuation-Passing-Style (Kapitel 6 im Abschnitt »Continuations« auf Seite 93). In gewissem Sinne sind beide äquivalent: Continuation-Passing-Style ist ein Spezialfall einer Monade, und jede Monade kann in Continuation-Passing-Style verwandelt werden, indem man den Typ der Antwort verändert. Um das alles noch interessanter zu machen ist eine Continuation in JavaScript auch ein Spezialfall einer Closure. Alles baut sich aus den gleichen Grundbausteinen auf. Aber der monadische Ansatz bringt uns laut der klassischen Literatur zusätzliche Einsichten und feiner dosierbare Kontrolle. Dieser Unterschied liegt unter anderem an der »Kontrolle« der Typen durch die Monade. (Quellen: The essence of functional programming und *https://cs. uwaterloo.ca/~david/cs442/monads.pdf*)

Genauer gesagt liegt beim Continuation-Passing-Style die Kontrolle beim Aufrufer der Funktionen, wogegen beim Programmieren mit Monaden die Kontrolle in der Definition der Funktion `mbind` steckt. Sie liegt also bei der Autorin der Monade, nicht beim Nutzer der Monade. Um diesen Unterschied zu verstehen, schauen wir uns einmal beide Wege

für die Maybe-Monade an. Stellen wir uns vor, wir wollen in der Maybe-Monade addieren, und haben zwei Funktionen vom Typ Maybe Number. Wenn wir auf Maybe-Typen addieren, können verschiedene Fälle auftreten.

```
// Addition zweier echter Werte Just x und Just y ergibt Just (x + y)
// In Zeile 1 und 2 sehen wir mbind, in Zeile 3 unit
var addM1 = Just(5).mbind(function (x) {
  return Just(4).mbind(function (y) {
    return Just(x + y); });
});

// Addition mit Nothing ist nicht möglich und ergibt Nothing
var addM2 = Just(5).mbind(function (x) {
  return Nothing.mbind(function (y) {
    return Nothing; });
});

addM1.toString();
> 'Just 9'
addM2.toString();
> "Nothing"
```

Die Berechnung endet also, sobald wir auf Nothing stossen. Versuchen wir, dieses monadische Verhalten im Continuation-Passing-Style auszudrücken.

```
function addC (x, y, k) { return k(x + y); }
```

Die Funktion id ist die Identitätsfunktion, die wir schon aus Kapitel 4 im Abschnitt »Verallgemeinerung der Funktionen höherer Ordnung auf Arrays« auf Seite 61 kennen, und die ihr Argument unverändert zurückgibt.

```
addC(5, 4, id);
> 9
```

Die Variante im Continuation-Passing-Style lässt sich ähnlich wie eine Monade verkettet aufrufen.

```
addC(1, 2, function (x) {
  return addC(x, 4, function (y) {
  return addC(y, 5, function (z) {
    return z; }); }); });
> 12
```

Kurz gesagt, auf Monaden schreiben wir

```
f.mbind(function (a) { return k(a); } );
```

und im Continuation-Passing-Style

```
function (c) { return f(function (a) { return k(a, c); } ); }
```

was beinahe gleich aussieht.

Der Unterschied besteht darin, dass jeder Monaden-Typ auch als abstrakter Datentyp gesehen werden kann, und dadurch kann das Verhalten besonders gut kontrolliert werden.

So könnte bei der Addition mit Hilfe der `Maybe`-Monade im Fehlerfall ein Abbruch erfolgen, der die aktuelle Continuation ignoriert und den Fehlertyp `Nothing` zurückgibt. Auf Monaden können wir beim Design einen solchen Fehlerfall einbauen, mit Continuations geht das nicht so einfach.

Wir können diese Fehlerbehandlung jedoch auch für Continuations implementieren, indem wir den Continuation-Passing-Style als abstrakten Datentyp umformulieren. Dies erreichen wir, indem wir den Continuation-Passing-Style in einer Monade ausdrücken. Damit haben wir die Monade gefunden, die auch »die Mutter aller Monaden« genannt wird – die Continuation-Passing-Style-Monade.

Das Lösen von Problemen mit Monaden statt Continuation-Passing-Style ermöglicht uns auch einen abstrakteren Blickwinkel: Wir konzentrieren uns genau darauf, welche Operationen zur Lösung nötig sind, welche Gesetze diese erfüllen müssen, und wie man die in verschiedenen Monaden gekapselten Eigenschaften kombinieren kann.

Beide Programmiertechniken bieten eine elegante Lösung für den Umgang mit verschachtelten Callback-Funktionen, die bei asynchroner Client-Server-Kommunikation in der Web-Programmierung auftreten können. Ob wir lieber im Continuation-Passing-Style programmieren, oder den Umgang mit den Daten von der Problemlösungsstrategie abtrennen, indem wir eine Monade benutzen, ist letztendlich Geschmackssache.

Quelle: *https://cs.uwaterloo.ca/~david/cs442/monads.pdf*

Quelle: »Die Continuation-Monade ist die Mutter aller Monaden«. *http://blog.sigfpe.com/2008/12/mother-of-all-monads.html*

Warum das Ganze?

Lohnt es sich überhaupt, Monaden in JavaScript zu studieren? Sind sie nicht in erster Linie eine Krücke, um in rein funktionalen Sprachen prozedural programmieren zu können und eine bestimmte Ausführungs-Abfolge zu erzwingen? Das können wir in JavaScript doch ohnehin schon. Dieser Einwand ist verständlich, trotzdem hat die Beschäftigung mit Monaden weit mehr Vorteile.

Nach der Betrachtung mehererer Monaden wissen wir nun, wie sie funktionieren. Wir könnten auch sagen, wir haben das Monaden-Design-Pattern mit `bind`, `unit` und `lift` kennengelernt, und verstanden, wie es benutzt wird, um Funktionen verkettet in Sequenz ausführen zu können. Außerdem haben wir gesehen, dass dieses Muster in vielerlei Gestalt auftaucht. Aber was haben wir von diesem Wissen?

Einleuchtend sind die theoretische Vorteile, die wir erlangen wenn wir das Muster »Monade« erkennen, benennen und verwenden lernen. Mustererkennung und richtige Terminologie lassen uns besser über unsere Arbeit sprechen. Wir können bei weitergehender Beschäftigung mehr Zusammenhänge erkennen, etwa wie das Konzept der Monade auf dem Konzept des (applikativen) Funktors aufbaut, und verstehen neben den Zusammenhängen zur Mathematik außerdem mehr über die Funktionsweise prozeduraler und funktionaler Programmiersprachen.

Das Wissen um Monaden bringt uns auch praktische Vorteile. Wir können Funktionen direkt auf »beliebigen« Monaden definieren. Damit können wir sie auf allen Instanzen von Monaden wiederverwenden. Die Funktion `sequence` z.B. ist eine Funktion, die eine Liste oder ein Array von monadischen Berechnungen bekommt, und diese der Reihe nach ausführt. Sie gibt eine Liste von Ergebnissen zurück. Diese Funktion können wir abstrakt für alle Arten von Monaden definieren. Genauso können wir ein `map` oder `reduce` definieren, das auf Monaden arbeitet, oder herkömmliche Funktionen auf in Monaden verpackten Werten rechnen lassen, indem wir sie mit einer Funktion `lift` dahingehend anpassen (man sagt die Funktion wird in die Monade »geliftet«). Mehrere Monaden lassen sich mittels Monaden-Transformern verbinden.

Neben den Funktionen für alle Monaden gibt es auch Anwendungsfälle bei denen eine maßgeschneiderte Monade sinnvoll ist. So können wir innerhalb von Monaden verpackt wiederkehrende Aufgaben erledigen, wie das Traversieren oder Backtracken auf Datentypen. Außerdem können wir durch Kapselung innerhalb der Monade bestimmte Eigenschaften für die Daten sicherstellen. Zum Beispiel, dass ein Array von Daten immer sortiert bleibt, oder, dass bestimmte Daten nicht verändert werden können (»immutable« sind).

Generell ist die Datenkapselung in Verbindung mit wiederkehrenden, voneinander abhängenden Rechenschritten ein guter Grund, Monaden zu verwenden, da dadurch globale Variablen vermieden werden können. Auch hilft das Monaden-Design-Pattern, unfreiwillige Komplexität zu vermeiden, die entsteht, wenn Code in einer Funktion nicht direkt mit dem zu lösenden Problem zu tun hat, sondern dazu dient, Datentypen miteinander zu verbinden und passend zu machen. Wenn man diesen wiederkehrenden Code erkennt und aus der Funktion herauslösen kann, kann dies das Programm viel klarer machen. Das Verständnis von Monaden mag in der Tat nicht zwingend notwendig sein, um in JavaScript programmieren zu können, es ermächtigt uns aber in vielen Fällen, eleganteren Code zu schreiben.

Es gibt unterschiedliche Meinungen dazu, ob Monaden relevant sind oder nicht. Diese spannende Diskussion wird auch an mehreren Stellen im Netz geführt (z.B. *http://cdsmith.wordpress.com/2012/04/18/why-do-monads-matter/* oder *http://gbracha.blogspot.ca/2011/01/maybe-monads-might-not-matter.html*). Wir sind der Meinung, Wissen um Monaden schadet auf keinen Fall.

Die gefürchteten Monaden-Gesetze

Juchu, wir haben das Kapitel überstanden, ohne bisher über die Monaden-Gesetze zu sprechen, die bei den anderen gefühlt 63528 Abhandlungen über Monaden immer so verwirren. (Link: *http://www.haskell.org/haskellwiki/Monad_tutorials_timeline*) Moment mal. Nun, da wir das Konzept der Monaden besprochen haben, können wir die Gesetze direkt für unsere Monade hinschreiben. Tatsächlich sind diese Gesetze sehr intuitiv und wir sind ohnehin schon die ganze Zeit davon ausgegangen, dass sie gelten. Die Gesetze legen fest, wie sich `mbind` (Komposition) und `unit` (Konstruktor) unter bestimmten Bedingungen verhalten. Wir können uns diese Gesetze ähnlich dem Kommutativ- und Assozia-

tivgesetz der Addition vorstellen. Sie bilden die mathematische Grundlage dafür, dass die Operationen auf der Monade funktionieren.

```
1) mbind(unit(value), f) ==== f(value)
2) mbind(monad, unit) ==== monad
```

Die ersten beiden Gesetze sind die Identitäts-Gesetze, und besagen, dass die Funktion `unit` ein einfacher Konstruktor im Bezug auf die Funktion `mbind` ist. Wenn der Operator `mbind` den Wert in der Monade auseinandernimmt und ihn an sein Funktionsargument übergibt, hat dieser Wert genau den Typ, den auch `unit` in die Monade hineingetan hätte. Anders herum betrachtet, wenn der Funktionsparameter von `mbind` einen Wert nimmt und ihn in die Monade hineinsteckt, wir also auf der Monade einen Rechenschritt ausführen, ist der Wert in der Monade vom gleichen Typ wie vor dem Rechenschritt.

```
3) mbind(mbind(monad, f), g) ====
   mbind(monad, function (value) { return mbind(f(value), g) })
```

Das dritte Gesetz ist zwar schwer in Worten zu beschreiben, aber sehr einleuchtend, ein bisschen wie die Abseitsregel im Fußball. Wenn wir eine Funktion mit einer Funktion `f` mittels `mbind` verknüpfen, und wir danach das Ergebnis mittels `mbind` mit einer zweiten Funktion `g` verknüpfen, ist das das gleiche, wie wenn wir die erste Funktion `f` auf den Wert in der Monade anwenden, und danach `mbind` mit der zweiten Funktion `g` auf dem Ergebnis anwenden. Dieses Gesetz hat Ähnlichkeit mit dem Assoziativgesetz.

Am dritten Gesetz kann man sehen, wie sich mit `mbind` Funktionen verketten lassen. Man kann man ableiten, dass verschachtelte Anwendungen von `mbind` »geflattened«, also flach gemacht werden können. Der Programmcode wird durch diese Umwandlung sehr sequentiell, was wir ja auch erreichen wollen.

Weblinks und weiterführendes Material

Im Netz finden sich mindestens so viele Erklärungstexte über Monaden, wie Menschen, die sich mit Monaden beschäftigen. Wir haben einige Quellen gesammelt, besonders der Vortrag von Douglas Crockford kommt ganz ohne Vorwissen über rein funktionale Programmiersprachen, Typen oder Kategorientheorie aus, und begnügt sich wie wir ganz allein mit JavaScript. Noch dazu gibt er Tipps, wie man Monaden in JavaScript am besten umsetzt, auch ohne einen Infix-Operator für `bind` wie in Haskell, und ohne ein Makro für eine Monade, wie wir es in Lisp schreiben könnten. Stattdessen benutzen wir Objektmethoden und Funktionen. Wenn die Monaden nach dieser Erklärung immer noch mysteriös erscheinen, lohnt es sich, das ganze praktisch anzugehen und verschiedene Monaden selbst in JavaScript zu implementieren.

Quelle übersetzt und angepasst für JavaScript: *http://www.codecommit.com/blog/ruby/monads-are-not-metaphors*

Klassische akademische Quelle zum tieferen Nachfoschen: »Comprehending Monads«, Philip Wadler, 1992

Maybe-Monade mit Infix-Operator-Emulation *http://www.techtangents.com/emulating-infix-operator-syntax-in-javascript/*

Crockford's Vortrag Monads and Gonads *https://www.youtube.com/watch?v=b0EF0VTs9Dc*

Start mit der Kategorientheorie *http://johnbender.us/2012/02/29/faster-javascript-through-category-theory/*

Durch das monadische `bind` an frische Milch entsteht in der indischen Küche Käse, der dann wieder an ein Curry gebunden werden kann. Der indische Frischkäse Paneer ist in gewisser Weise eine Monade von der heiligen Kuh, im Hinduismus »die Mutter des Lebens« und sozusagen die Continuation-Monade des Daseins. Paneer schmeckt nicht nur ausgesprochen lecker, sondern ist auch eine hervorragende Grundlage für vegetarische Currygerichte aller Art.

Rezept: Paneer selbstgemacht

Für 4 Personen:

- 2l Milch
- 60-70 ml Zitronensaft

Zitronensaft mit etwas warmen Wasser verdünnen. Die Milch in einem schweren Topf oder einer passenden Monade zum Kochen bringen und unter ständigem Umrühren darauf achten, dass sie nicht anbrennt. Sobald die Milch kocht den Zitronensaft nach und nach hineintröpfeln. Wenn sich flockige Käsekrümel absetzen die Hitze herunterdrehen. Wenn sich das Fett der Milch abgesetzt hat, wird es abgeschöpft und in einen Topf gegeben, der mit einem Küchen- oder Mulltuch ausgeschlagen ist. Die Käsebröckchen gut einschlagen im Tuch, abbrausen und kräftig auswringen und drücken. Durch das Herauspressen der Flüssigkeit wird die Zitronensäure weniger. Den gepressten Käse im Tuch mit einer schweren Pfanne oder Konservendose beschweren und über Nacht etwa 6 Stunden (am besten im Kühlschrank) weiter zusammenpressen lassen. Paneer schmeckt in sahnigen Currys in Würfel geschnitten köstlich, eigentlich immer dann, wenn wir auf Fleisch verzichten wollen.

Selbstgemachter Käse oder selbstimplementierte Monaden mögen vielleicht am Anfang ein bisschen Respekt einflößen, aber so wild ist die Sache dann doch wieder nicht – und bei der Zubereitung werden einige Zusammenhänge über grundlegende Zutaten klar. Was bleibt, ist der Nachtisch und die ganzen leckeren Reste, denen wir uns im nächsten Kapitel zuwenden.

```
(λ z → z (λ s → λ z → s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
(s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s (s
z))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))))
(λ z → z (λ w → λ f → w) (λ w → λ f → w))))))))))))
```

KAPITEL 10

Nachtisch und Resteessen

In diesem Buch haben wir neben der funktionalen Programmierung in JavaScript auch Rezepte aus der indische Küche probiert. Nun wollen wir auf Weltreise gehen und erleben, wie in anderen Sprachen funktional programmiert werden kann. Wir starten dabei auf bekanntem JavaScript-Terrain und erweitern unseren funktionalen Sprachschatz durch Bibliotheken; unsere Reise führt uns dann zu Programmiersprachen, die nach JavaScript übersetzt werden. Wir werden abschließend einige weiter entfernte funktionale Sprachen besuchen. Jede Sprache bringt dabei ihre eigene Kultur mit sich.

Auch das Kochen unterscheidet sich je nach Region und Kultur in der Art und Tradition, nach der Speisen und Getränke ausgewählt und zubereitet werden. Bevor wir den indischen Subkontinent verlassen, wollen wir das Beste nicht vergessen, den Nachtisch.

Lassi (Hindi: लस्सी lassī; Urdu لسی) ist ein Mixgetränk aus der Punjab-Region Indiens und Pakistans, das Joghurt, Wasser, Früchte und Gewürze enthält. Am bekanntesten ist das Mango-Lassi, aber es gibt auch Varianten mit anderen Früchten, Minze, Rosenwasser oder sogar salziges Lassi. Durch den Fettgehalt mildert Lassi die Schärfe des Essens und es ist kalt serviert eine Erfrischung bei heissem Wetter.

Rezept: Mango-Lassi

Für 1 Person

- 150 g Joghurt
- 200 ml Wasser
- 200 g Mango-Fruchtfleisch
- 1 Prise Kardamom (gemahlen)
- 1 St Nelke (gemahlen)

– Fortsetzung –

- 1 Prise Zimt
- 1 Prise frisch gemahlenen Ingwer
- 1 EL brauner Zucker
- etwas Zitronensaft
- etwas gestoßenes Eis

Die Mango in kleine Würfel schneiden und gemeinsam mit den anderen Zutaten im Mixer pürieren. Auf gestoßenem Eis servieren oder vor dem Servieren noch mal in den Kühlschrank stellen.

Die im Buch besprochenen Techniken der funktionalen Programmierung führen durch Abstraktion und Vermeidung von Seiteneffekten zu lesbarem, kurzen und modularem Programmcode. Da wir sie in JavaScript erarbeitet haben, gehen wir zusammenfassend noch einmal auf die Besonderheiten dieser Programmiersprache ein.

Eigenheiten der Sprache JavaScript

Die Sprache JavaScript hat einige Besonderheiten, in denen sie sich von anderen Sprachen unterscheidet. Dies macht JavaScript zu einer Sprache, die sich oft überraschend verhält. Deshalb bezeichnet Douglas Crockford JavaScript auch als »die am meisten missverstandene Programmiersprache der Welt«.

Trotzdem ist JavaScript verbreitet, und sowohl für Programmiereinsteiger als auch für erfahrene Programmierer gut geeignet da es als Multi-Paradigmen-Sprache verschiedene Programmierstile erlaubt. Wir haben in diesem Buch ganz klassisch in JavaScript programmiert. Mit mehr syntaktischem Zucker oder gar ganz anderen Sprachen wäre die Umsetzung der Beispiele natürlich auch möglich.

Dynamischer Scope

Eine erste Liste der häufigsten Überraschungen haben wir in Kapitel 1 im Abschnitt »Eigenschaften und Eigenheiten von JavaScript« auf Seite 4 gesehen. Einige der Unwegsamkeiten können durch achtsames Design von Programmen umschifft werden, beispielsweise kann das dynamische Scoping durch Erzeugen einer Closure, die sofort ohne Argument appliziert wird (siehe Kapitel 5 im Abschnitt »Lösung: Memoisation / Tabellierung« auf Seite 70) ausgetrickst werden.

Prototypenbasiertes Objektsystem

JavaScript hat ein prototypenbasiertes Objektsystem. Prototypen werden in nicht vielen anderen Programmiersprachen verwendet. Für Programmierer, die sich in einem klassenbasierten Objektsystem besser zurechtfinden, gibt es externen Bibliotheken wie Joose

(Kapitel 8 im Abschnitt »Klassenbasiertes Objektsystem und Polymorphismus mit Joose« auf Seite 144). Bibliotheken, die für JavaScript klassenbasierte Objektsysteme implementieren, gibt es wie Sand am Meer. Die Objektsysteme verschiedener Bibliotheken sind leider oft nicht kompatibel und die Bibliotheken können dann nicht gemeinsam benutzt werden.

Coercion und der überladene Operator +

Weitere Überraschungen birgt die automatische Typkonversion (Kapitel 8 im Abschnitt »Falle 1: Automatische Typkonversion / Coercion« auf Seite 141) beim Vergleich zweier Werte. Bei dieser werden die Typen von Objekten automatisch umgewandelt, wie sie gerade benötigt werden.Deshalb ist es nötig, die Regeln der Konversion genau zu kennen, oder die Operatoren `==` und `!=`, welche ihre Operanden automatisch konvertieren, zu vermeiden und `===` (und `!==`) zu verwenden.

Probleme durch den überladenen Operator + haben wir schon in Kapitel 8 im Abschnitt »Falle 2: Überladene Operatoren« auf Seite 142 gesehen. Im Beispiel aus Kapitel 8 erhalten wir für den Ausdruck (`4 + 6 + ' Portionen'`) das Ergebnis `'10 Portionen'`. Aber für den Ausdruck (`'Portionen: ' + 4 + 6`) erhalten wir den String `'Portionen: 46'`. Da sich erst zur Laufzeit anhand der Typen der Argumente entscheidet, ob addiert oder konkateniert wird, muss man beim Programmieren überlegen, welche Operation ausgeführt wird. Bei mehreren Anwendungen von + spielt auch die Reihenfolge eine Rolle. Abhilfe kann eine explizite Typkonversion bieten, beispielsweise mit `parseInt`. Bei dieser Funktion sollte man wie in Kapitel 2 im Abschnitt »Abstraktion von einer Funktion: Funktionen höherer Ordnung« auf Seite 25 und Kapitel 4 im Abschnitt »Exkurs: Vorsicht bei ... « auf Seite 50 beschrieben darauf achten, die Basis immer richtig anzugeben.

Argumentübergabe

Auch die Art, mit der Argumente in JavaScript übergeben werden, kann überraschend für Programmierer anderer Sprachen wie C oder Java sein. In diesen Sprachen muss ein Programmierer mit Hilfe von `...` explizit definieren, dass eine Methode eine beliebige Anzahl von Argumenten bekommen kann. Man sagt, diese Methoden haben eine variable Arität oder Stelligkeit. In JavaScript hat jede Methode eine variable Stelligkeit. Die Deklaration der Argumente ist lediglich syntaktischer Zucker zum Binden dieser Argumente. Hier muss man beim Programmieren besonders aufpassen, oder explizit überprüfen, ob die Anzahl der tatsächlich übergebenen Argumente mit der Anzahl der deklarierten Argumente übereinstimmt. Diese kann durch einen Vergleich wie `this.length === arguments.length` innerhalb eines Funktionsrumpfes geschehen. Dabei enthält die Eigenschaft `length` einer jeden Funktion die Anzahl der deklarierten Argumente, und die Spezialvariable `arguments` ein Array-ähnliches Objekt der übergebenen Argumente. In Kapitel 4 haben wir erklärt, welche Überraschungen die Übergabe von Argumenten an Funktionen bereithalten kann.

Semikolons

Ein großer Streitpunkt unter JavaScript-Entwicklerinnen sind die Semikolons. Es gibt das automatische Einfügen von Semikolons (Kapitel 1 im Abschnitt »Semikolons« auf Seite 5). Wie bei der Coercion müssen entweder die Regeln gelernt und gekannt werden, oder aber nach jeder Anweisung ein explizites Semikolon getippt werden. Interessanterweise haben auch andere Programmiersprachen wie Scala einen ähnlichen Mechanismus, der Semikolons einfügt; Haskell und Python benutzen die Einrückung (also die Leerzeichen) zur Strukturierung des Programms in Blöcke. Viele Programmiersprachen erlauben, bei Schleifen und bedingten Anweisungen die geschweiften Klammern wegzulassen, wenn nur eine Anweisung im Rumpf steht. Hier ist, wie bei der Coercion, besondere Aufmerksamkeit erforderlich.

Konzepte der funktionalen Programmierung

Rekursion, algebraische Datentypen, Pattern Matching, Funktionen höherer Ordnung

JavaScript ist nicht in erster Linie als funktionale Sprache konzipiert worden. Einige fortgeschrittene Konzepte fehlen daher im Sprachumfang. In diesem Buch haben wir Techniken funktionaler Programmierung beschrieben, die wir bei der alltäglichen Arbeit mit JavaScript verwenden können, ganz ohne externe Bibliotheken. Zu diesen Techniken gehören Funktionen höherer Ordnung (Kapitel 3), Listenbeschreibungen (List comprehensions) als Kurzform für Kombinationen von map und filter (Kapitel 4), Rekursion (Kapitel 5 im Abschnitt »Listen als algebraische Datentypen, Pattern Matching« auf Seite 79). Weitere Techniken wie algebraische Datentypen und Pattern Matching (Kapitel 5 im Abschnitt »Listen als algebraische Datentypen, Pattern Matching« auf Seite 79) sind gute gedankliche Konzepte, und aus rein funktionalen Programmiersprachen nicht mehr wegzudenken. In JavaScript empfehlen wir, für diese Konzepte externe Bibliotheken zu benutzen, um JavaScript ein wenig funktionaler zu machen.

Continuation-Passing-Style

Ein ganz eigener Programmierstil ist der Continuation-Passing-Style, den wir in Kapitel 6 im Abschnitt »Continuations« auf Seite 93 kennengelernt haben. Im Continuation-Passing-Style kehren Funktionen nicht zurück, sondern bekommen eine Funktion übergeben, die sie am Ende mit ihrem Ergebnis aufrufen. Continuations vereinfachen die Struktur von Programmen, die asynchron arbeiten. Diese Art der Kommunikation ist bei der Web-Programmierung in JavaScript ein alltägliches Problem. Die asynchrone Programmierung mit Continuations ermöglicht das Schreiben von Programmen, die nur geringe Datenmengen mit dem Server austauschen, keine Sessions benutzen, und während der Kommunikation weiterhin in der Lage sind, mit dem Benutzer zu interagieren. Der nötige Zustand auf dem Server soll ja möglichst gering gehalten werden, um die Ser-

verressourcen zu schonen und die Option zu bieten, bei Abbruch der Netzwerkverbindung weiterhin die Anwendung interaktiv nutzen zu können.

Lazy Evaluation

Zum Sparen von Datenvolumen und Rechenzeit bietet sich auch die faule Auswertung oder Lazy Evaluation an, am besten mit wu.js garniert. Hier werden Ausdrücke nicht unmittelbar ausgewertet, sondern nur wenn sie tatsächlich benötigt werden. Die faule Auswertung haben wir uns während der Beschreibung des Lambda-Kalküls (Kapitel 7 im Abschnitt »Lazy Evaluation – die faule Auswertung« auf Seite 112) angeschaut.

Monaden

Um ein Programm möglichst abstrakt und funktional zu halten, und trotzdem die Möglichkeit zur Ausgabe von Debug-Information zu haben, haben wir in Kapitel 9 auf Seite 156 die Writer-Monade kennengelernt. Mit der Technik der Monaden bleibt das ursprüngliche Programm unverändert, wir schreiben Programme, die Programme als Eingabe und Ausgabe haben. Durch diesen Trick können wir Programme um zusätzliche in Monaden gekapselte Eigenschaften erweitern, ohne die Programmstruktur zu verändern.

Funktionale Bibliotheken

Im Verlauf des Buches sind uns einige funktionale Bibliotheken für JavaScript begegnet, die funktionale Konzepte und praktische Funktionen wie `flip` und `curry` bereitstellen.

Die Bibliotheken *underscore.js* und *functional.js* sind die bekanntesten JavaScript-Bibliotheken für funktionale Programmierung. Dabei ist *underscore.js* eine Funktionssammlung, die aus der Bibliothek Prototype extrahiert wurde. Prototype wurde früher zusammen mit *script.aculo.us* bei Ruby on Rails mitgeliefert, war also in einem der ersten Web-2.0-Frameworks. Funktionen wie `map` sind mittlerweile Teil des regulären Sprachstandards von JavaScript, man kann also auch ganz ohne Bibliotheken funktional programmieren, wie in diesem Buch.

Die Bibliothek *underscore.js* ist immer noch ein guter Einstieg in die funktionale Programmierung. Oliver Steeles Bibliothek *functional.js* ist mächtiger, aber auch viel unkonventieller in der Implementation. Zum Beispiel lassen sich dort Lambda-Ausdrücke als Strings hinschreiben, die dann von *functional.js* interpretiert werden. Die Bibliothek *wu.js* bereichert JavaScript um automatische Curryfizierung (Kapitel 3 im Abschnitt »Automatische Curryfizierung« auf Seite 40) und Lazy Evaluation (Kapitel 7 im Abschnitt »Lazy Evaluation – die faule Auswertung« auf Seite 112).

Eine umfangreiche Sammlung funktionaler Konzepte für JavaScript findet sich in der Bibliothek *Functional-JavaScripts*. Diese umfasst in etwa den in Haskell und seinem Prelude mitgelieferten Funktionsumfang: automatische Curryfizierung (Kapitel 3 im Abschnitt »Automatische Curryfizierung« auf Seite 40), algebraische Datentypen (Kapitel 5 im Abschnitt »Listen als algebraische Datentypen, Pattern Matching« auf Seite 79), Pattern

Matching (Kapitel 5 im Abschnitt »Listen als algebraische Datentypen, Pattern Matching« auf Seite 79), Monaden (Kapitel 9) und vordefinierte Funktionen aus dem Prelude.

Links:

https://github.com/osteele/functional-javascript/

http://documentcloud.github.com/underscore/

https://github.com/fitzgen/wu.js

https://github.com/DrBoolean/Functional-Javascripts

Sprachen, die JavaScript verbessern wollen

Es gibt einige Programmiersprachen, die auf JavaScript aufbauen, und dabei mehr Klarheit versprechen. Klarheit kann hier zweierlei bedeuten. Knapperen und dadurch klareren Programmcode durch mehr syntaktischen Zucker, oder Klarheit im Bezug auf die Typen (Kapitel 8), durch Annotieren von Variablen mit Typinformation und die dadurch ermöglichte statische Überprüfung.

Low Level Virtual Machine (LLVM) ist ein Compilerframework, das sich zum Ziel gesetzt hat, Compiler für verschiedene Quell- und Zielsprachen zu vereinheitlichen. Es wurde eine Zwischensprache festgelegt, so dass der Teil des Compilers, der die Quellsprache einliest, und der Teil, der die Zielsprache erzeugt, beliebig kombiniert werden können. Durch diese Entwicklung ist für jede Quellsprache von LLVM eine Übersetzung nach JavaScript möglich. Dadurch lässt sich nun jedes Programm nach JavaScript übersetzen, wenn die Quellsprache von LLVM akzeptiert wird, so dass eine grosse Menge bestehender Programme nun auch im Browser ausgeführt werden könnten.

Doch zurück zum Thema. Wir werden in verschiedene Küchen schauen, denn in unseren Augen ist jede der vorgestellten Programmiersprachen die nach JavaScript übersetzt werden die Reise wert, sie sich anzuschauen.

CoffeeScript

http://coffeescript.org/ Die vermutlich beliebteste Programmiersprache, die zu JavaScript-Code übersetzt wird, ist CoffeeScript. CoffeeScript wurde 2009 entwickelt, damals war der Compiler in Ruby geschrieben. Im Webframework Ruby on Rails is CoffeeScript seit der Version 3.1 von Rails im Lieferumfang dabei, wodurch die Sprache eine gewisse Verbreitung gefunden hat. Die Idee von CoffeeScript ist, JavaScript lesbarer und kürzer zu machen, es fügt also syntaktischen Zucker zu JavaScript hinzu. CoffeeScript wurde von Ruby und Python inspiriert, lehnt sich aber auch an Haskell an. Es gibt erweiterte Listenbeschreibungen (Kapitel 4) und Pattern Matching (Kapitel 5). CoffeeScript ist mittlerweile sogar in CoffeeScript selbst implementiert, und ein Programm in CoffeeScript ist in etwa ein Drittel kürzer als ein gleiches Programm in JavaScript. Die Performance zur Laufzeit ist nicht meßbar geringer als die von JavaScript. Es gibt aber auch einige Schattenseiten von Coffee-

Script, zum einen ist CoffeeScript sehr sensibel was Leerzeichen angeht. Ein Tab wird als einzelnes Leerzeichen gezählt. Die Fehlermeldungen des CoffeeScript-Compilers sind sehr rudimentär und nicht immer macht es Spaß, Code zu debuggen, da hier eine weitere Abstraktionsebene in JavaScript eingebracht wird, die sich im kompilierten Code oft nicht nachvollziehen lässt. Eine lesenswerte Kritik an CoffeeScript auf funktionaler Sicht findet sich unter http://ruoyusun.com/2013/03/17/my-take-on-coffeescript.html

LiveScript

http://livescript.net/ LiveScript war ulkigerweise der Originalname von JavaScript in Netscape 2.0 beta, ist aber auch eine Programmiersprache, die auf JavaScript aufsetzt. LiveScript bietet einige Syntaxerweiterungen, um funktionales Programmieren in JavaScript zu vereinfachen. Es gibt dort vereinfachte List comprehensions, Currying, Komposition und Pattern Matching. Es ist sehr ähnlich zu CoffeeScript. Die Bibliothek Prelude.ls ist eine in LiveScript geschriebene funktionale Bibliothek für JavaScript, die vom Prelude der Sprache Haskell inspiriert ist.

Dart

http://www.dartlang.org/ Die Programmiersprache Dart wurde von Google im Jahre 2011 entwickelt, um JavaScript zu ersetzen. Sie bietet ein klassenbasiertes Objektsystem, gewohnt aus Programmiersprachen wie Java oder C#. In Dart sind statische Typannotationen möglich, die zur Dokumentation und für Entwicklungswerkzeuge benutzt werden. Die Typannotationen werden in Laufzeitüberprüfungen umgewandelt, statt in JavaScript ständig den Operator `typeof` verwenden zu müssen, bietet Dart eine prägnante Syntax für Typannotationen. Dart ist angetreten, schneller als JavaScript zu sein. Kritikpunkte an Dart sind, dass sie die Syntax von JavaScript durch eine Syntax, die der von Java noch näher ist, ersetzt hat. Ein weiterer Kritikpunkt ist, dass das Typsystem anders als andere, daher schwer zu lernen, und nur optional ist. Auch die Nähe zu Google als Hersteller von Webbrowsern ist ein großer Kritikpunkt. Der Erfinder von JavaScript, Brendan Eich, sagte zu Dart, dass er glaubt, dass andere Browserhersteller wie Firefox, Opera, Apple und Microsoft Dart wegen der proprietären Herkunft niemals integrieren werden.

TypeScript

http://www.typescriptlang.org/ Auch Microsoft hat 2012 die Programmiersprache TypeScript veröffentlicht, die wir schon kurz in Kapitel 8 im Abschnitt »Spezifikation von Typen für Variablen mit TypeScript« auf Seite 145 vorgestellt haben. Die Motivation für Microsoft hinter TypeScript sind große Webanwendungen, mit denen Microsoft umgehen muss. TypeScript bietet ein klassenbasiertes Objektsystem und ein Modulsystem. Diese Features sind teilweise in ECMAScript 6 vorgesehen. Microsoft hat auch ein graduelles optionales Typsystem entwickelt. Ein sehr interessanter Aspekt bei TypeScript sind sogenannte Header-Files, ähnlich wie bei C und C++. In diesen Header-Files können Typsignaturen für Bibliotheken, die in reinem JavaScript geschrieben sind, bereitgestellt werden. Somit können bestehende

JavaScript-Bibliotheken wiederverwendet werden, und durch die zusätzlichen Informationen können statisch mehr Fehlerfälle gefunden und gemeldet werden. TypeScript ist mittlerweile auch in Microsofts Entwicklungsumgebung Visual Studio integriert. Die Kritik an TypeScript ist, ähnlich wie bei Dart, dass sie proprietär ist – also von einem Webbrowser-Hersteller entwickelt wird. Anders als bei Dart wird allerdings keine eigene Laufzeitumgebung verwendet, sondern der TypeScript-Code wird in JavaScript-Code übersetzt.

ClojureScript

https://github.com/clojure/clojurescript Die Programmiersprache Clojure ist ein Lisp-Dialekt, der in der Java Laufzeitumgebung ausgeführt wird. Dadurch können Bibliotheken, die in Java implementiert sind, direkt benutzt werden. Clojure hat viele Features von Common Lisp, wie Metaprogrammierung durch Makros, eine klare Trennung zwischen Methoden und Datenstrukturen, Funktionen höherer Ordnung. ClojureScript ist eine Teilmenge von Clojure, die speziell für JavaScript getuned ist. Die Sprache ClojureScript ist weniger mächtig als Clojure, ist aber speziell auf JavaScript angepasst und hat daher eine sehr gute Performance.

Roy

http://roy.brianmckenna.org/ Die Forschungssprache Roy compiled auch zu JavaScript-Code. Roy implementiert vor allem rein funktionale Features von Haskell wie ein Typsystem mit Typeninferenz, eine knappe Syntax für Monaden, Pattern Matching, Metaprogrammierung zur Compilezeit, eine genauere Syntax, in der die Leerzeichen relevant sind, ähnlich wie Haskell, Python und CoffeeScript. Die Motivation des Erfinders von Roy ist, dass es verdammt schwer sein kann, korrekten JavaScript Code zu schreiben.

ECMAScript 6

Seit 2008 ist ein neue ECMAScript Standard in der Entwicklung, der ECMAScript 6 oder auch Harmony genannt wird. Dieser Standard ist ähnlich wie ECMAScript 4 (Kapitel 8 im Abschnitt »ECMAScript 4« auf Seite 145) und wird ein klassenbasiertes Objektsystem beinhalten. Auch andere Feature wie die Optimierung von Tail-Rekursion (Kapitel 5), Namespaces und Packages sollen in ECMAScript 6 Einzug halten.

Die Welt der funktionalen Sprachen abseits von JavaScript

Uns sind in diesem Buch immer wieder andere Programmiersprachen begegnet, bei denen funktionale Programmierung näher liegt als bei JavaScript, weil sie dafür gemacht wurden. Es lohnt sich aus unserer Sicht, zwei Vertreterinnen genauer anzusehen. Mit einem Blick auf Lisp-Dialekte oder auch auf Haskell wird man auf keinen Fall dümmer. Dabei wollen wir nicht verschweigen, dass hier auch unsere Sympathien liegen: Während eine Autorin dieses Buches

in ihrer Doktorarbeit mit einer Haskell-Lösung Probleme der Bioinformatik löste, arbeitete ein anderer Autor dieses Buches an einem freien Compiler für den Lisp-Dialekt Dylan.

Scheme und Lisp

Scheme ist eine sehr kleine Programmiersprache, ähnlich wie das Lambda-Kalkül aus Kapitel 7 und wird daher an vielen Universitäten in der Lehre eingesetzt. Es ist sogar so klein, dass man es in 48 Stunden selbst implementieren kann, zum Beispiel auch in JavaScript. Link: *https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_ 48_Hours*.

Scheme benutzt eine Präfix-Notation, der Ausdruck `2 + 3` in JavaScript wird in Scheme als `(+ 2 3)` geschrieben. Ein solcher Ausdruck wird S-Expression genannt, und entspricht direkt einem abstrakten Syntaxbaum. Scheme hat kein Klassensystem und ist dynamisch typisiert, es gibt aber auch Dialekte, die Scheme um diese Eigenschaften erweitern. Eine grundlegende Eigenschaft von Scheme sind Makros, spezielle Funktionen, die schon zur Compilezeit ausgewertet werden. Diese Makros bekommen S-Expressions als Eingabe und geben wieder S-Expressions aus. Makros ermöglichen Metaprogrammierung, da sie erlauben Programme zu schreiben, die Programme verändern. Ein Makro nimmt dabei eine Umformung am abstrakten Syntaxbaum vor.

Auf Scheme baut die Sprache Common Lisp auf, die zusätzlich das klassenbasierte Objektsystem CLOS und andere Erweiterungen mitbringt.

Haskell

Die Programmiersprache Haskell ist eine stark typisierte sprache, die lazy ausgewertet wird. Sie ist nach Haskell B. Curry benannt, den wir vor allem aus Kapitel 3 im Abschnitt »Curryfizierung und teilweise Anwendung von Funktionen« auf Seite 34 kennen. Auch in Haskell sind Funktionen First-Class-Citizens, es gibt aber noch weitere funktionale Konzepte wie algebraische Datentypen, Pattern Matching, Listenbeschreibungen und automatische Curryfizierung. Das Typsystem von Haskell unterstützt eine statische Typprüfung, Typklassen und Typpolymorphismus. Fehlende Typinformationen werden durch die Hindley-Milner-Typinferenz automatisch abgeleitet. Da Haskell eine rein funktionale Sprache (purely functional) ist, sind Funktionen wie in der Mathematik aufzufassen. Für die gleiche Eingabe liefert eine Funktion stets die gleiche Ausgabe, ohne Seiteneffekte. In Haskell gibt es daher nur Variablenbindung und keine Zuweisung. Seiteneffekte werden in Haskell durch Monaden umgesetzt. Bekanntere Projekte, die in Haskell implementiert sind, sind der Windowmanager xmonad (Link: *http://www.xmonad.org/*) und auch das Betriebssystem house (Link: *http://programatica.cs.pdx.edu/House/*).

Abhängige Typen, Agda, Idris und Coq

Typsysteme in rein funktionalen Sprachen haben eine klare Trennung zwischen Werten und Typen. Der Typ ist die vorläufige Information, die über einen Wert zur Compilezeit bekannt ist. Werte hingegen sind erst zur Laufzeit bekannt. Um Werte zur Compilezeit bes-

ser unterscheiden zu können, möchte man zusätzliche Informationen zum Typ hinzufügen. Abhängige Typen sind strenger, da sie Informationen, die vorher nur im Wert vorhanden waren, in den Typ mit aufnehmen. In Haskell schreiben wir für die Konkatenation zweier Listen den Funktionstyp `List a -> List a -> List a`, die Funktion bekommt also zwei Listen mit Elementen vom Typ `a` als Argumente und gibt auch eine solche Liste zurück. In Idris, einer Programmiersprache mit abhängigen Typen, können wir den Typ der Konkatenationsfunktion noch genauer spezifizieren, indem wir den Listentyp zusätzlich von der Länge abhängig machen. Wir schreiben dann `List a n -> List a m -> List a (n + m)` für den Funktionstyp. Es wird bereits durch den Typ festgelegt, dass die Ergebnisliste so lang wie die Summe der Längen der Eingabelisten sein muss. Die Verantwortung der richtigen Ergebnislänge wurde also von der Implementierung in den Typ verlagert. So lassen sich noch mehr Fehler zur Compilezeit ausräumen. Die Compiler von Agda und Idris, zwei Sprachen mit abhängigen Typen, haben auch JavaScript als Zielsprache.

Die Curry-Howard-Korrespondenz haben wir schon in Kapitel 3 im Abschnitt »Curryfizierung und teilweise Anwendung von Funktionen« auf Seite 34 erwähnt. Sie beschreibt den Zusammenhang zwischen Logik und dem Lambda-Kalkül. Hier ist die Logik gemeint, in der Beweise konstruiert werden. Im Jahr 1969 entdeckte Curry, dass das Lambda-Kalkül und diese Beweis-Logik, die unabhängig voneinander entwickelt wurden, einander entsprechen. Ein Beweis für ein Theorem in der Logik kann als ein Programm aufgefasst werden. Wir erinnern uns an den Slogan aus Kapitel 3, »proofs as programs«.

Aufbauend auf der Idee der Curry-Howard-Korrespondenz werden abhängige Typsysteme verwendet, um Beweise interaktiv auf Computern zu führen. Ein Beispiel für ein interaktives Beweissystem ist der Theorem-Beweiser Coq. Coq benutzt abhängige Typen, um einen Theorem-Ausdruck in einen Typ zu übersetzen. Wenn die Typprüfung in Coq erfolgreich ist, ist das Theorem bewiesen.

Unsere Reise durch die aromenreiche Welt der funktionalen Programmierung endet hier, aber sie muss längst nicht zu Ende sein. Mit den vorgestellten Konzepten wie Rekursion, Map-Reduce, Continuations, dem Lambda-Kalkül und fortgeschrittenen Ideen wie Typen und Monaden sollte das Programmieren nie wieder so sein wie bisher. Viele Programmiererinnen und Programmierer machen es sich zum persönlichen Ziel, jedes Jahr eine neue Programmiersprache zu lernen. Nicht, um mit der neu gelernten Sprache alles ersetzen zu können – das dürfte ohnehin schwer fallen, wenn im Job JavaScript vorgegeben ist. Aber warum nicht mal in einen Lisp-Dialekt wie Scheme oder Dylan hineinschauen, um geschickter und wendiger mit Closures umgehen zu können? Warum nicht mal dem Geheimnis der Monaden mit Haskell auf den Grund gehen, um sich dann wieder jQuery zuzuwenden? Und was spricht dagegen, sich von neueren Sprachen wie Agda und Idris in Sachen Typen inspirieren zu lassen oder interaktiv mit dem Theorembeweiser Coq zu spielen? Ein Anfang ist gemacht und wir hoffen, dass das Programmieren in JavaScript nach der Lektüre dieses Buches eine neue Welt offen steht. Namaste und guten Appetit!

Link- und Literaturliste

Kapitel 1: Hinein ins Vergnügen

Links

http://nodecopter.com/ – The NodeCopter – Programming flying robots with node.js

http://www.haskell.org/haskellwiki/Closure – Closure – HaskellWiki

http://nodejs.org/ – Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications

https://npmjs.org/ – Node Packaged Modules

http://www.currybuch.de/ – Funktionales Programmieren in JavaScript

https://xkcd.com/221/ – xkcd: Random Number

Literatur

Douglas Crockford »JavaScript: The Good Parts – Unearthing the Excellence in JavaScript« O'Reilly Media / Yahoo Press, May 2008

Kapitel 2: Abstraktion

Links

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/parseInt – Referenz zu parseInt – JavaScript | Mozilla Developers Network

http://stackoverflow.com/questions/979256/how-to-sort-an-array-of-javascript-objects – How to sort an array of javascript objects?

Kapitel 3: Ein Topf mit Curry

Links

http://commons.wikimedia.org/wiki/File:Aloo_gobi.jpg – Foto Aloo Gobi

http://www.techtangents.com/emulating-infix-operator-syntax-in-javascript/ – Emulating infix operator syntax in javascript | techtangents

http://web.archive.org/web/20090406105627/http://blogger.xs4all.nl/peterned/archive/2009/04/01/462517.aspx – Overloading operators in javascript | peterned

http://www-history.mcs.st-andrews.ac.uk/Biographies/Curry.html – Haskell Brooks Curry Biographie

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/arguments – Referenz zu arguments - JavaScript | Mozilla Develpers Network

http://javascriptweblog.wordpress.com/2010/04/05/curry-cooking-up-tastier-functions – Curry: cooking up tastier functions | Angus Croll

http://junctional.org/articles/partial-application/ – Partial application | Brian Lonsdorf

http://javascriptweblog.wordpress.com/2010/06/14/dipping-into-wu-js-autocurry/ – dipping into wu.js: autoCurry | Angus Croll

http://osteele.com/sources/javascript/functional/ – Functional Javascript | Oliver Steele

http://www.slideshare.net/drboolean/pointfree-functional-programming-in-javascript – Pointfree functional programming in javascript | drboolean

http://dailyjs.com/2012/09/14/functional-programming/ – Functional Programming in JavaScript | Nathaniel Smith

Literatur

Moses Schönfinkel »Über die Bausteine der mathematischen Logik« in Mathematische Annalen 92, pp. 305–316, 1924 Translated by Stefan Bauer-Mengelberg as »On the building blocks of mathematical logic« in Jean van Heijenoort, 1967. A Source Book in Mathematical Logic, 1879–1931. Harvard Univ. Press: 355–66.

Haskell Brooks Curry, Willa A. Wyatt »A study of inverse interpolation of the Eniac.« in Technical Report 615, Ballistic Research Laboratories, Aberdeen Proving Ground, Maryland (1946)

Haskell Brooks Curry »On the composition of programs for automatic computing.« in Technical Report 9805, Naval Ordnance Laboratory (1949)

Liesbeth De Mol, Maarten Bullynck, Martin Carlé »Haskell before Haskell: Curry's Contribution to Programming (1946-1950)« in Conference on Computability in Europe - CIE, pp. 108-117, 2010 »A short biography of Haskell B Curry« in To H B Curry : essays on combinatory logic, lambda calculus and formalism" (London-New York, 1980), vii-xi.

Kapitel 4: Gemüse, Map, Reduce und Filter

Links

http://upload.wikimedia.org/wikipedia/commons/b/b7/BT_Brinjal_Protest_Bangalore_India.JPG – Foto der Proteste gegen die Brinjal Aubergine

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Array – Referenz zu Array - Mozilla Developer Network

http://swizec.com/blog/javascripts-native-map-reduce-and-filter-are-wrong/swizec/1873 – JavaScript's native map reduce and filter are wrong | Swizec Teller

http://foldl.com – Interaktive Erklärung der Funktion fold left

http://foldr.com – Interaktive Erklärung der Funktion fold right

http://en.wikipedia.org/wiki/Fold_%28higher-order_function%29#Evaluation_order_considerations – Evaluation order considerations

http://fitzgen.github.com/wu.js – A lazy, functional Javascript library that ain't nuthin' ta f*ck wit. | Nick Fitzgerald

Kapitel 5: Rekursion

Links

https://en.wikipedia.org/wiki/File:Fibonacci_spiral.svg – Zeichnung der Fibonacci-Spirale

http://oeis.org/A000045 – Die Fibonacci-Zahlenfolge in der Online Encyclopedia of Integer Sequences

http://en.literateprograms.org/Fibonacci_numbers_%28JavaScript%29 – Fibonacci-Zahlen in JavaScript

http://wiki.ecmascript.org/doku.php?id=harmony:proper_tail_calls – Proper Tail Calls in ECMAScript 6

http://glat.info/pub/tailopt-js/ – On-the-fly tail call optimization in Javascript without trampoline | Guillaume Lathoud

http://dynamo.iro.umontreal.ca/wiki/index.php/Main_Page – Gambit Scheme

http://users-cs.au.dk/danvy/sfp12/papers/thivierge-feeley-paper-sfp12.pdf (siehe auch Literatur)

http://stackoverflow.com/questions/13323916/javascript-recursive-data-structure-definition – Javascript recursive data structure definition

http://w3future.com/weblog/stories/2008/06/16/adtinjs.xml – Algebraische Datentypen

http://www.bramstein.com/projects/funcy/ – Functional Pattern Matching in JavaScript | Bram Stein

https://github.com/DrBoolean/Functional-Javascripts – Lots of functional libraries all modified to work with web/titanium/node environments. | drboolean

Literatur

Peter Norvig »Techniques for Automatic Memoization with Applications to Context-Free Parsing,« in Computational Linguistics, Vol. 17 No. 1, pp. 91–98, March 1991

Eric Thivierge, Marc Feeley »Efficient Compilation of Tail Calls and Continuations to JavaScript« in Scheme and Functional Programming 2012 (siehe auch Links)

Kapitel 6: Event-basierte Programmierung und Continuations

Links

https://github.com/lobbyplag/eurlex-js/ – Retrieve documents from EUR-Lex and convert them to usable data

https://github.com/lobbyplag/eurlex-js/blob/master/eurlex.js – Retrieve documents from EUR-Lex and convert them to usable data, Implementation

http://blog.jcoglan.com/2011/03/11/promises-are-the-monad-of-asynchronous-programming/ – Promises are the monad of asynchronous programming | James Coglan

http://blog.jcoglan.com/2013/03/30/callbacks-are-imperative-promises-are-functional-nodes-biggest-missed-opportunity/ – Callbacks are imperative, promises are functional: Node's biggest missed opportunity | James Coglan

Kapitel 7: Vom Lambda-Kalkül und Lammcurry

Links

https://github.com/doublec/jsparse – JavaScript Parser Combinator Library | Chris Double

Literatur

John Reynolds »Definitional interpreters for higher-order programming languages« in Proceedings of the ACM Annual Conference. Boston, Massachusetts. pp. 717–740, 1972

Graham Hutton »Higher-order functions for parsing.« in Journal of functional programming 2, no. 3 (1992): 323-343.

Kapitel 8: Typen

Links

http://javascriptweblog.wordpress.com/2010/09/27/the-secret-life-of-javascript-primitives/ – The Secret Life of JavaScript Primitives | Angus Croll

http://skilldrick.co.uk/2011/09/understanding-typeof-instanceof-and-constructor-in-javascript/ – Understanding typeof, instanceof and constructor in JavaScript | Skilldrick

http://stackoverflow.com/questions/5972991/why-do-we-need-the-isprototypeof-at-all – Why do we need the isPrototypeOf at all?

https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/create – Referenz zu Object.create | Mozilla Deverloper Network

http://stackoverflow.com/questions/9850892/should-i-use-polymorphism-in-javascript – Should I use polymorphism in javascript?

http://webreflection.blogspot.ca/2010/10/javascript-coercion-demystified.html – JavaScript Coercion Demystified | Andrea Giammarchi

http://www.scottlogic.co.uk/2010/10/implementing-eqeq-in-javascript-using-eqeqeq/ – Implementing eqeq in JavaScript using eqeqeq | Scott Logic

http://unixpapa.com/js/convert.html – Javascript Madness: Treacherous Type Conversions | Jan Wolter

http://joose.it/ – Joose: advanced meta-class system for JavaScript

http://www.typescriptlang.org/ – TypeScript is a language for application-scale JavaScript development

http://www.ecmascript.org/es4/spec/evolutionary-programming-tutorial.pdf (siehe auch Literatur)

http://www.rust-lang.org/ – Rust: a safe, concurrent, practical language

Literatur

Luca Cardelli, Peter Wegner »On Understanding Types, Data Abstraction, and Polymorphism« in ACM Computer Surv. (ACM) 28: 150 – 1985

Lars Hansen »Evolutionary Programming and Gradual Typing in ECMAScript 4« in 2007 (siehe auch Links)

Kapitel 9: Kochen als Monade

Links

http://blog.jcoglan.com/2011/03/05/translation-from-haskell-to-javascript-of-selected-portions-of-the-best-introduction-to-monads-ive-ever-read/ – Translation from Haskell to JavaScript of selected portions of the best introduction to monads I've ever read | James Coglan

http://mylittlewiki.org/wiki/G1_Ponies – My Little Wiki: Generation 1 My Little Ponies

http://igstan.ro/posts/2011-05-02-understanding-monads-with-javascript.html – Understanding Monads With JavaScript | Ionu G. Stan

https://cs.uwaterloo.ca/~david/cs442/monads.pdf (siehe auch Literatur)

http://blog.sigfpe.com/2008/12/mother-of-all-monads.html – The Mother of all Monads | Dan Piponi

http://cdsmith.wordpress.com/2012/04/18/why-do-monads-matter/ – Why Do Monads Matter? | cdsmith

http://gbracha.blogspot.ca/2011/01/maybe-monads-might-not-matter.html – Maybe Monads Might Not Matter | Gilad Bracha

http://www.haskell.org/haskellwiki/Monad_tutorials_timeline – Monad tutorials timeline | HaskellWiki

http://www.codecommit.com/blog/ruby/monads-are-not-metaphors – Monads Are Not Metaphors | Daniel Spiewak

http://www.techtangents.com/emulating-infix-operator-syntax-in-javascript/ – Emulating infix operator syntax in javascript | techtangents

https://www.youtube.com/watch?v=b0EF0VTs9Dc – Monads and Gonads | Douglas Crockford

http://johnbender.us/2012/02/29/faster-javascript-through-category-theory/ – Faster JavaScript Through Category Theory | John Bender

Literatur

Philip Wadler »The essence of functional programming« in POPL '92 Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (siehe auch Links)

Philip Wadler »Comprehending monads« in Mathematical Structures in Computer Science, Special issue of selected papers from 6'th Conference on Lisp and Functional Programming, 2:461–493, 1992.

Kapitel 10: Nachtisch und Resteessen

Links

https://github.com/osteele/functional-javascript/ – Functional is a library for functional programming in JavaScript | Oliver Steele

http://documentcloud.github.com/underscore/ – Underscore is a utility-belt library for JavaScript that provides a lot of the functional programming support that you would expect in Prototype.js (or Ruby), but without extending any of the built-in JavaScript objects.

https://github.com/fitzgen/wu.js – A lazy, functional Javascript library that ain't nuthin' ta f*ck wit. | Nick Fitzgerald

https://github.com/DrBoolean/Functional-Javascripts – Lots of functional libraries all modified to work with web/titanium/node environments. | drboolean

http://coffeescript.org/ – CoffeeScript is a little language that compiles into JavaScript. Underneath that awkward Java-esque patina, JavaScript has always had a gorgeous heart. CoffeeScript is an attempt to expose the good parts of JavaScript in a simple way.

http://ruoyusun.com/2013/03/17/my-take-on-coffeescript.html – My Take on CoffeeScript | Ruoyu Sun

http://livescript.net/ – LiveScript is a language which compiles to JavaScript. It has a straightforward mapping to JavaScript and allows you to write expressive code devoid of repetitive boilerplate.

http://www.dartlang.org/ – Dart brings structure to web app engineering with a new language, libraries, and tools

http://www.typescriptlang.org/ – TypeScript is a language for application-scale JavaScript development

https://github.com/clojure/clojurescript – ClojureScript is a new compiler for Clojure that targets JavaScript. It is designed to emit JavaScript code which is compatible with the advanced compilation mode of the Google Closure optimizing compiler

http://roy.brianmckenna.org/ – Roy is an experimental programming language that targets JavaScript. It tries to meld JavaScript semantics with some features common in static functional languages.

https://en.wikibooks.org/wiki/Write_Yourself_a_Scheme_in_48_Hours – Write Yourself a Scheme in 48 Hours - A Haskell Tutorial | Jonathan Tang

http://www.xmonad.org/ – *xmonad* – the tiling window manager that rocks

http://programatica.cs.pdx.edu/House/ – Haskell User's Operating System and Environment

Index

G

H

I

J

K

L

M

N

O

P

R

W

Y

Z

Über die Autorin und die Autoren

Hannes Mehnert arbeitet in verschiedenen Bereichen der Programmiersprachenforschung: von Compileroptimierungen in Dylan und deren Visualisierungen, über einen TCP/IP Stack inklusive einer domain-specific language für Binärprotokolle bis hin zu einer Erweiterung des Editors Emacs zur interaktiven Entwicklung von Idris-Code. Er widmet sich auch funktionalen Korrektheitsbeweisen von Java-Code in dem Theorem-Beweiser Coq und entwickelt dazu ein Eclipse-Plugin, das eine Entwicklungsumgebung für Coq bietet und das Beweisen von Java-Programmen mit der Entwicklung dieser in Eclipse integriert. In seiner Freizeit ist Hannes nicht nur Hacker, sondern auch ein leidenschaftlicher Barista - wenn er eine E61 Brühgruppe vorfindet; benutzt am liebsten das Betriebssystem FreeBSD, fährt und repariert gern sein Liegerad und backt sein Sauerteigbrot selbst.

Jens Ohlig arbeitet als Software-Entwickler bei Wikimedia Deutschland und lebt in Berlin und Bonn, aber hauptsächlich im Internet. Er ist sowohl von gutem Essen als auch freier Software begeistert und arbeitet mit wachsender Begeisterung mit JavaScript und Node.JS

Unter den Schöpfungen von **Stefanie Schirmer** findet sich von Holzskulpturen über twitternde Multiplayer-Skateboards bis hin zu Wearable-Spielen fast alles, was sich selber machen lässt. Am liebsten programmiert sie dabei funktional, so wie in ihrem Werdegang als Informatikerin. Zunächst bei ihrem Studium in Bielefeld, wo sie für ihr Diplom in naturwissenschaftlicher Informatik einen Typchecker in Haskell entwickelte, und erste Begegnungen mit eigenen Kombinatoren und Monaden hatte. Im folgenden Doktorandenstudium in Bioinformatik entwickelte sie Baumvergleichsmetriken zum Strukturvergleich von RNA-Molekülen und setzte den resultierenden Haskell-Code in C++ um. Diese Interessen brachten sie sowohl 2012 zur Hackerschool nach New York, als auch zu ihrer momentanen Forschungsstätte, der Université de Montréal in Kanada, wo sie in der Krebsforschung arbeitet.

Kolophon

Das Tier auf dem Einband von *Das Curry-Buch -- Funktional programmieren mit JavaScript* ist der Fleckenlinsang (*Prionodon pardicolor*). Er ist ein in Südostasien lebendes Raubtier aus der Familie der Linsangs (*Prionodontidae*). Das Verbreitungsgebiet der Fleckenlinsangs erstreckt sich von Nepal und dem nordöstlichen Indien über das südliche China bis nach Vietnam, Laos und Kambodscha sowie in den Norden Thailands. Ihr Lebensraum sind in erster Linie Regenwälder, sie kommen aber auch in Bambuswäldern, Galeriewäldern und teilweise im angrenzenden Grasland vor. Fleckenlinsangs sind überwiegend nachtaktiv. Sie können ausgezeichnet klettern und halten sich häufig auf den Bäumen auf. Sie schlafen auch dort, kommen aber bei der Nahrungssuche immer wieder auf den Boden. Vermutlich leben sie einzelgängerisch. Sie sind Fleischfresser. Ihre Nahrung besteht aus Nagetieren, Fröschen, Schlangen und kleinen Vögeln, Berichten zufolge fressen sie auch Aas. Ein- oder zweimal im Jahr bringt das Weibchen zwischen Februar und August meist zwei Jungtiere zu sich. Diese werden in einer Baumhöhle großgezogen.

Das Cover dieses Buchs wurde von Michael Oreal gestaltet. Als Textschrift verwenden wir die Linotype Birka, die Überschriftenschrift ist die Adobe Myriad Condensed und die Nichtproportionalschrift für Codes ist LucasFont's TheSan Mono Condensed.