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

1. Auflage 2013

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:

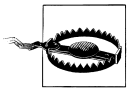
Kursivschrift
: Kennzeichnet URLs, Dateinamen, Dateinamen-Erweiterungen und Verzeichnis-/Ordnernamen. Ein Pfad im Dateisystem wird zum Beispiel als */Entwicklung/Anwendungen* erscheinen.

`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 കറി), 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.