UNIVERSITÄT OSNABRÜCK
Institut für Informatik

Bachelorthesis

hobbes
–
Ein Java-Interpreter in JavaScript

Johannes Emerich

Juni 2011

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachter: PD Dr. Helmar Gust

Gewidmet meinen lieben Eltern und dem gekrümmten Horizont.

Zusammenfassung

Skripten und Lehrmaterialien können auf einer ständig wachsenden Zahl von Endgeräten rezipiert werden. Die in Informatik-Lehrmaterialien enthaltenen Code-Beispiele benötigen jedoch in der Regel eine bestimmte Umgebung zur Ausführung, die aufgrund von Einschränkungen durch Hard- oder Software nur von einem geringen Teil der Endgeräte geboten werden kann. Ein großer und wachsender Teil der Endgeräte unterstützt jedoch die Anzeige von HTML und eingebettetem JavaScript. Durch das Schaffen einer geeigneten Umgebung im Browser mittels JavaScript kann so die direkte Ausführung von Beispielcode ermöglicht werden. In der Arbeit soll als ein Ansatz ein in JavaScript geschriebener Java-Interpreter vorgestellt werden.

Inhaltsverzeichnis

KAPITEL 1

EINFÜHRUNG

Während unmittelbar nach der Erfindung des Digitalcomputers Rechenzeit ein knappes Gut war und angehende Programmierer möglichst viele Konzepte bereits vor der Arbeit mit dem Rechner zu lernen hatten, wurde mit dem *personal computer* das Lernen am Rechner für immer mehr Menschen zugänglich. Durch Multi- und Hypermedia wurden so völlig neue Möglichkeiten interaktiven Lernens eröffnet. In der heutigen *post-PC era*[1] ist die Zahl der zum Medienkonsum geeigneten und verfügbaren Geräte weiter gewachsen. Insbesondere durch Tablets und E-Book-Reader werden durch neue Bedienkonzepte und einer besseren Eignung zum Lesen längerer Texte wiederum neue Ansätze möglich. Ein solcher ist der in dieser Arbeit beschriebene, der das interaktive Lernen von Programmiersprachen erleichtern soll. Da die Arbeit im Gedanken an die Einführungsveranstaltung *Informatik A* der Universität Osnabrück entstand, wird dabei eine Lösung für die dort verwendete Programmiersprache Java vorgestellt.

1.1 NEUARTIGE LESEGERÄTE ALS MOTIVATION

Mit der steigenden Zahl an, zu sinkenden Preisen erhältlichen, Lesegeräten, wächst auch die papierlose Rezeption von Lehrmaterialien und technischen Dokumentationen. Während die Attraktivität dieser Geräte sich zum Teil aus dem einfachen Bezug und der platzsparenden Vorhaltung einer großen Auswahl an Texten erklärt, bieten die neuen Technologien auch inhaltliche Chancen für Didaktiktexte. Einerseits kann man wohl mit einem verstärkten Einzug des inzwischen *altbewährten* Hypertext und einem Aufbruch strikt linearer Texte rechnen. Andererseits

[1] Begriff von Steve Jobs, siehe u.a. [McC11]

bietet die digitale Darbietung auch für lineares Material neue Möglichkeiten.

In fast allen Einführungen in Programmiersprachen weisen die Autoren auf die Notwendigkeit des praktischen Nachvollziehens der erklärten Beispiele hin. In der Realität sind die Bedingungen zur Befolgung des gut gemeinten Rates aber häufig nicht ideal. Zwischen dem Lesen im Buch und dem Ausführen des Beispiel-Codes steht in der Regel die Einrichtung der jeweiligen Laufzeit-Umgebung auf einem Rechner, sowie der Bezug des Quellcodes von CD oder Website, im schlimmsten Fall gar das manuelle Kopieren des Quellcodes aus dem Buch. Zwar könnte man zu Recht auf die relative Nichtigkeit dieser Hürden hinweisen. Es wird aber einleuchten, dass selbst diese kleinen Hindernisse zum Aufschub des Nachvollziehens führen können. Es erfordert dann einige Disziplin, die verschobene Aufgabe auch wirklich nach zu holen. Doch selbst bei bestem Willen schränkt das nötige Vorhandensein eines entsprechend eingerichteten Rechners die Wahl der Lernumgebung stark ein.

Die Verbreitung der neuen Lesegerättypen Tablet und E-Book-Reader legt nahe, die Tatsache, dass es sich bei Ihnen im Grunde um Computer handelt, zu nutzen, um die Kluft zwischen Text und praktischer Erfahrung zu verringern.

Die große Diversität auf dem Markt schafft dabei jedoch einige Herausforderungen. Man begegnet hier verschiedensten Prozessor-Architekturen, Betriebssystemen, und hardware- sowie herstellerbedingten Einschränkungen. Das bedeutet zum einen, dass in der Regel nicht damit zu rechnen ist, dass der übliche Nutzer, ob praktisch oder grundsätzlich, in der Lage ist, selbst eine entsprechende Umgebung auf dem Gerät einzurichten. Zum anderen gelten für Produzenten von Lehrmaterialien die gleichen Einschränkungen, die, so die Hersteller die Bereitstellung geeigneter Umgebungen nicht direkt legal oder technisch unterbinden, zumindest das Erstellen auf die jeweilige Architektur angepasster Umgebungen erforderlich machen.

Es wäre wünschenswert, wenn sich der Traum des *write once, run anywhere*[2] auch in dieser Gerätekategorie erfüllen könnte. Während diese Rolle auf dem PC in vielen Fällen von Suns/Oracles Java oder Macromedias/Adobes Flash gespielt wurde, haben gerade diese Technologen im Mobile-Computing-Markt – nicht zuletzt durch Apples gezielte Initiative[3] – einen zunehmend schweren Stand.

Durch die Ausrichtung des Marktes auf Internetkonsum und Webanwendungen gibt es aber weiterhin eine Umgebung, die den meisten Geräten gemein ist – den Browser. Möchte man hier Interaktion ermöglichen, ist (da Flash und Java

[2] Ein im Zusammenhang mit Java verwendeter Slogan, siehe [Wik10]

[3] siehe z.B. Jobs' offener Brief zum Thema Flash, [Job10]

auch hier wegfallen) die Wahl des Werkzeugs vorbestimmt. JavaScript ist die de facto einzige Sprache im Browser, da alle anderen Möglichkeiten, etwa Microsofts VBScript wiederum plattformspezifisch sind. Es ist daher zu untersuchen, ob sich JavaScript im Browser zur Einrichtung entsprechender Umgebungen eignet.

1.2 Zielsetzung und Aufbau

Ziel der Arbeit ist es, durch die Implementation eines Java-Interpreters in JavaScript die Machbarkeit der zuvor beschriebenen Idee zu prüfen und dabei einen Einblick in Möglichkeiten und Schwierigkeiten zu erhalten. Um den Rahmen einer Bachelorarbeit nicht zu sprengen, soll nur ein beschränkter Teil des Java-Sprachkerns unterstützt werden und die Interaktion mittels eines simulierten Terminals mit Text-Ein- und -Ausgabe stattfinden. Dabei geht es um Möglichkeiten, mit dem häufig noch als *toy language* geltenden JavaScript komplexe Software umzusetzen. Persönliches Ziel war weiterhin ein besseres Verständnis der Funktionsweise höherer Programmiersprachen und des Bezugs von Sprachspezifikation zu tatsächlicher Umsetzung.

Zunächst sollen zum besseren Verständnis der späteren Teile in Kapitel 2 die relevanten Technologien vorgestellt werden, die für die Thesis erarbeitet wurden. Das ist zum einen eine Beschreibung der betreffenden Sprachbestandteile von Java. Zum anderen ist es die Beschreibung JavaScripts als Sprache mit einer besonderen Geschichte und dadurch bedingten Besonderheiten. Es soll auch auf aktuelle Bemühungen eingegangen werden, die Vorteile dieser Besonderheiten zu nutzen und gleichzeitig mit ihren Nachteilen zurecht zu kommen. Darauf folgend soll eine kurze Erklärung der Konzepte des Compilerbaus mit `lex` und `yacc` folgen.

In Kapitel 4 wird der zugrundegelegte Entwicklungs-Ansatz mit einem verhaltensgeleiteten Verfahren beschrieben, der verwendete Parser-Generator *Jison* vorgestellt und schließlich die letztendliche Programmstruktur mit den Schritten Parsing, Kompilierung und Ausführung erläutert. Zum Abschluss des Kapitels wird eine beispielhafte Verwendung des Interpreters im Browser beschrieben.

In Kapitel 5 wird abschließend auf didaktische Hoffnungen und Befürchtungen eingegangen, der Erfolg des Projektes evaluiert und zuletzt ein Überblick über alternative Ansätze gegeben.

KAPITEL 2

TECHNISCHE GRUNDLAGEN

2.1 JAVA

Java ist eine stark typisierte, klassenbasiert-objektorientierte Programmiersprache, die sich syntaktisch an C und C++ anlehnt[1]. In den frühen 1990er Jahren wurde die Sprache von Sun Microsystems zunächst für *Embedded*-Anwendungen entworfen, entwickelte sich aber schnell zu einer vielseitig einsetzbaren Sprache für objektorientierte Systemprogrammierung. In üblichen Szenarien wird Java zunächst in Java-Bytecode übersetzt, welcher dann von einer virtuellen Maschine, der *Java Virtual Machine* interpretiert und ausgeführt wird. Es wird in diesem Zusammenhang zwischen *compile-time*-Fehlern, die während der Übersetzung in Bytecode auftreten, und *run-time*-Fehlern, die während der Ausführung des Bytecodes auftreten, unterschieden. Im Unterschied zu anderen Sprachen ist beim Kompilieren von Java durch die starke Typisierung eine weitgehende Programmanalyse möglich, die die Gefahr falscher Zuweisungs- oder sonstiger Operationen verringert. Durch ein abstrakteres Sprachniveau werden zusätzlich einige potentiell gefährliche Programmier-Fehler, wie sie beispielsweise in C++-Programmen auftreten können, grundsätzlich verhindert.

Die Übersetzung in die *Zwischensprache* Bytecode erlaubt eine erhöhte Portabilität der Java-Binärprogramme bei guter Performanz – welche allerdings zu einem erheblichen Teil durch Optimierungen und teilweise Kompilierung zu Maschinencode durch die virtuelle Maschine zur Laufzeit erreicht wird.

Insbesondere in den späten 90er Jahren wurde Java darüber hinaus eine beliebte Wahl für einführende Programmier-Kurse und konnte diese Rolle bis heute ver-

[1] Geschichte und Merkmale von Java nach [Wal10] und [GJSB05]

teidigen [Sch07], wobei sich in jüngerer Zeit eine Entwicklung hin zu Konzepten funktionaler Programmierung und damit eine Abwendung von Java abzuzeichnen scheint [BSS10].

In Javas Typensystem besitzt jede Variable und jeder Ausdruck bereits zum Zeitpunkt der Übersetzung einen bestimmten Typen, der die möglichen Zuweisungen, Operationen und Werte einschränkt. Javas Typen-Taxonomie wird in zwei Grundkategorien unterteilt, **primitive Typen** und **Referenz-Typen**. Variablen primitiven Types speichern einen tatsächlichen Wert primitiven Typs, während Variablen vom Referenz-Typ nur die Referenz zu einem entsprechenden Wert enthalten. Die Referenz-Typen sind die Klassen-, Interface- und Array-Typen, sowie der mit allen drei zuweisungskompatible null-Typ. Auf sie soll hier nicht weiter eingegangen werden, da sie nicht Teil der umgesetzten Sprachmerkmale sind.

Die primitiven Typen sind der boolesche Typ `boolean` sowie die numerischen Typen. Die einzigen beiden Werte von booleschem Typ sind die durch die Literale `true` und `false` notierten Wahrheitswerte. Die numerischen Typen sind die vorzeichenbehafteten Ganzzahltypen `byte`, `short`, `int` und `long`, der vorzeichenlose Ganzzahltyp `char`, sowie die Gleitkommatypen `float` und `double`. Der Typ `char` repräsentiert mittels der UTF16-Zeichentabelle ein Zeichen. Tabelle 2.1 gibt einen Überblick über die Wertebereiche der Ganzzahltypen.

Typ	Bits	Minimalwert	Maximalwert
`byte`	8	-128	127
`short`	16	-32768	32767
`int`	32	-2147483648	2147483647
`long`	64	-9223372036854775808	9223372036854775807
`char`	16	0 bzw. `'\u0000'`	65536 bzw. `'\uffff'`

Tabelle 2.1: Wertebereiche der Ganzzahltypen

Die Gleitkommatypen kodieren reelle Zahlen nach dem Standard IEEE 754. Dabei ist ein Bit für das Vorzeichen reserviert. Die Minimal- und Maximalwerte in Tabelle 2.2 stehen daher für die absolute Entfernung von der Null.

Typ	Bits	Minimalwert	Maximalwert
`float`	32	$1.40239846 \times 10^{-45}$	$3.40282347 \times 10^{38}$
`double`	64	$4.94065645841246544 \times 10^{-324}$	$1.79769313486231570 \times 10^{308}$

Tabelle 2.2: Wertebereiche der Gleitkommatypen

Werte der Gleitkommatypen können außerdem die vorzeichenbehaftete *Unendlichkeit*, die vorzeichenbehaftete Null, sowie der *Not-a-Number*-Wert, der das Ergebnis bestimmer nicht definierter mathematischer Operationen ist, sein.

Java besitzt außer den booleschen und numeralen eine Reihe weiterer Literale, von denen hier nur das String-Literal genannt sein soll. Jedes String-Literal ist vom Typ `String` und zwei identische String-Literale werden zur selben `String`-Instanz ausgewertet, da vor dem Erzeugen einer neuen Instanz zunächst eine systeminterne Tabelle auf bereits bestehende Instanzen der Zeichenkette untersucht wird. In diesem Zusammenhang ist es sinnvoll und notwendig, dass Javas String-Instanzen unveränderlich sind. Im Rahmen der vorliegenden Arbeit wurden String-Literale mit aufgenommen, dabei aber als eine Art primitiver Typ behandelt, da eine Implementation der Referenz-Typen nicht vorgesehen war.

Java zeichnet sich durch eine außergewöhnlich starke Klassenorientiertheit aus, aufgrund derer Anweisungen nur innerhalb von Methodendeklarationen notiert werden dürfen, welche wiederum nur im Körper einer Typendeklaration, also einer Klassen- oder Interfacedeklaration erscheinen dürfen. Eine Kompilationseinheit, das heißt eine Java-Quelldatei, darf darüber hinaus nur genau eine Typendeklaration enthalten. Weitere, sogenannte *innere Klassen*, dürfen aber im Inneren einer Klassendeklaration deklariert werden. Innere Klassen wie auch Interfaces sollen im weiteren keine Beachtung finden, da sich ihre Relevanz auf die objektorientierten Aspekte von Java beschränkt.

Es wird im weiteren davon ausgegangen, dass der Leser mit der grundsätzlichen Syntax von C-artigen Sprachen vertraut ist und die detaillierte Untersuchung einiger beispielhafter Konstrukte daher im Zusammenhang mit der Übersetzung in JavaScript stattfinden.

2.2 JavaScript

Innerhalb eines Jahres entwarf und implementierte Brendan Eich 1995 im Auftrag von Netscape eine Sprache zum Einsatz in Netscape's Webbrowser *Netscape Navigator*[2]. Zunächst unter dem Namen *LiveScript* geführt, war sie als Bindeglied zwischen HTML-Dokumenten und darin eingebundenen Java-Applets gedacht. In diesem Sinne und im Zuge der neuen Popularität von Java als Sprache, wurde sie dann am 4. Dezember 1995 von Netscape und Sun als *JavaScript* vorgestellt und als Teil der Version 2.0 des Netscape Navigator veröffentlicht.

[2]Geschichte von JavaScript nach [Sei09], [Cro08] und [Fla08]

2.2.1 Die Entstehung von JavaScript

Eich war angehalten, JavaScript in syntaktischer Ähnlichkeit zu Java zu entwerfen, den Sprachumfang aber nicht zu weit zu fassen, um zu vermeiden, dass Programmierer Browseranwendungen komplett in JavaScript zu verfassen, statt in der Hauptsache Java zu verwenden [Sei09]. Gleichzeitig nennt Eich Java allerdings als eine Sprache, die als abschreckendes Beispiel auf die konzeptuelle Entwicklung JavaScripts wirkte und unter anderem Anlass zur Wahl eines Modells prototypischer Vererbung war. Als Vorbild dieses Modells nennt Eich *Self*. Weitere Einflüsse waren der funktionale Ansatz des *LISP*-Dialekts *Scheme* sowie das Event-Modell von *HyperTalk*. Heraus kam eine dynamisch typisierte, prototypbasiert-objektorientierte, funktionale Programmiersprache.

Obwohl Microsoft mit *VBScript* eine eigene Scripting-Sprache für ihren Browser *Internet Explorer* besaßen, enthielt die 1996 erschienene Version 3.0 einen JavaScript-Nachbau unter dem Namen *JScript* [Cha01]. Über den über Jahre dauernden Konkurrenzkampf um die Vormachtstellung im Browsermarkt sei hier nur gesagt, dass sie das Arbeiten mit JavaScript durch teils inkomptabile APIs und Implementationen immens erschwerten und diese, gar nicht den Sprachkern betreffenden, Probleme mit zum schlechten Ruf der Sprache beigetragen haben. Recht früh sah man aber die Notwendigkeit einer gemeinsamen Grundlage und so wurde die 1996 begonnene Standardisierung des Sprachkerns im Juni 1997 abgeschlossen. Der Standard ECMA-262 mit dem Titel *ECMAScript: A general purpose, cross-platform programming language* benennt JavaScript offiziell zu *ECMAScript* um, in der Praxis hat sich diese Bezeichnung allerdings nicht durchgesetzt[3].

ECMAScript wird zunächst unabhängig vom Einsatz im Browser beschrieben und enthält nicht die BOM- (*Browser Object Model*) bzw. DOM-Erweiterungen (*Document Object Model*), die die Interaktion mit Webdokumenten ermöglichen. Stattdessen gilt der Browser lediglich als **eine** mögliche Umgebung, die im globalen Namensraum Objekte zur Browserinteraktion verfügbar macht und außerdem eine Möglichkeit zur Bindung von ECMAScript-Funktionen an Browserevents bietet. Hierzu und zur folgenden Beschreibung des Sprachkerns, siehe [ECM99], [Cro08] und [Fla08].

[3] Das 1997 veröffentlichte Standarddokument ist nicht mehr offiziell zu beziehen, aber auf dem Webserver der Mozilla Foundation archiviert: http://www.mozilla.org/js/language/E262.pdf Zur besseren Nachvollziehbarkeit soll als Quelle für die weitere Darstellung die Revision des Standards von 1999 gelten, [ECM99].

2.2.2 ECMAScript: Der Sprachkern

ECMAScript kennt die primitiven Typen *Undefined* mit einem einzigen Wert[4], *Null* mit dem einzigen Wert `null`, *Boolean* mit den Werten `true` und `false`, *Number* mit 64 Bit IEEE-754-Gleitkommazahlen als Werten und *String* mit Unicode-16-Zeichenketten als Werten.

Alle anderen Werte in ECMAScript sind Objekte. Ein Objekt ist dabei eine ungeordnete Sammlung von Eigenschaften, wobei jede Eigenschaft eine benannte Referenz auf einen anderen Wert ist. Man kann ECMAScript-Objekte also als spezielle Hash-Tabellen betrachten. In starkem Kontrast zu anderen objektorientierten Programmiersprachen, benötigt man zur Erzeugung eines Objekts keine bestimmte Klasse und kein Objekt ist Instanz irgendeiner Klasse. Objekte lassen sich stattdessen auf verschiedene Arten erzeugen. Da es sich bei ihnen um einfache Eigenschafts-Sammlungen handelt, lässt sich mittels des Objekt-Literals ein Objekt wie folgt definieren:

```
1 > var john = { name : "John", saiten : 6 };
2 > john.name
3 'John'
4 > john['name']
5 'John'
```

Listing 2.1: Beispiel eines Objekt-Literals

Dadurch erhält man ein Objekt mit der Eigenschaft `name` mit dem *String*-Wert `"John"` und der Eigenschaft `saiten` mit dem *Number*-Wert `6`. Die beiden in 2.1 gezeigten Arten, auf Objekteigenschaften Zugriff zu nehmen sind äquivalent. Die Punkt-Notation ist lediglich eine kompaktere Syntax für Eigenschaftsnamenkonstanten, falls diese die Form eines ECMAScript-Identifiers besitzen.

ECMAScript definiert ein Spektrum von eingebauten Objekten, darunter je eines in Korrespondenz zu jedem primitiven Typ, sowie *Array*, *Date*, *Function*, *Math*, *Object* und das *globale Objekt*. Mit einer Ausnahme sind diese Objekte unter einem entsprechenden Namen – beispielsweise das *Array*-Objekt unter dem Namen `Array` – im globalen Namensraum referenziert[5].

Die Ausnahme, das *globale Objekt*, ist im globalen Namensraum unter dem Namen `this` referenzierbar. Das deutet bereits auf die Tatsache hin, dass es sich beim globalen Namensraum um die Menge der Eigenschaftsnamen des globalen Objekts handelt. Da sich der Wert des Stichworts `this` mit dem Kontext, in dem es erscheint, ändern kann, ist das globale Objekt in der Regel noch unter einem

[4] im globalen Namensraum üblicherweise unter dem Namen `undefined` referenziert, sonst Wert jeder nicht initialisierten Variablen

[5] und dereferenzierbar

gesonderten Namen im globalen Namensraum referenziert – im Browser unter `window`.

Die Offenheit und Manipulierbarkeit des globalen Namensraums über das globale Objekt stellt eine große Problemquelle dar, zumal ECMAScript keine eigenen Namensräume für Codeblöcke besitzt. Das Konzept des globalen Objekts ist der Verwendung im Browser zu zu schulden, wo es eine einfache Lösung für die Kommunikation verschiedener in einem Dokument eingebundener Scripts bietet. Das schafft nicht nur sicherheitsrelevante, sondern schon allein softwaretechnische Probleme, da es die Modularisierung und Kapselung von Komponenten erschwert.

```
1 > var i = 0;
2 > while (i < 10) { var j = i++; }
3 > j
4 9
```

Listing 2.2: In einem Block deklarierte Variable *verschmutzt* globalen Namensraum

Der einzige Kontext, in dem ein gesonderter Namensraum vergeben wird, ist innerhalb von Funktionskörpern. Wie beschrieben, gibt es ein eingebautes *Function*-Objekt, welches zum Erzeugen von Funktionsobjekten genutzt werden kann. Üblicherweise wird dazu aber die ebenfalls verfügbare Literalschreibweise für Funktionen verwendet.

```
1 > function sauberBleiben (wert) { var lokal = wert; }
2 > sauberBleiben(1);
3 > lokal
4 ReferenceError: lokal is not defined
```

Listing 2.3: Beispiel eines Funktions-Literals mit lokalem Namensraum

Vergisst man bei der ersten Zuweisung zu einer Variablen allerdings das Stichwort `var`, verlässt der Name den lokalen Namensraum.

```
1 > function schlechtGemacht (wert) { lokal = wert; }
2 > schlechtGemacht(1);
3 > lokal
4 1
```

Listing 2.4: Beispiel eines Funktions-Literals mit Programmierfehler

Weiter unten werden Methoden beschrieben, mit diesen Schwächen umzugehen. Funktionen können ganz wie andere Werte als Eigenschaften von Objekten gespeichert werden. In diesem Zusammenhang spricht man von ihnen dann auch als *Methoden* des Objekts, deren Eigenschaft sie sind. Das Stichwort `this` verweist innerhalb einer Methode auf das zugehörige Objekt. Dabei kann eine Funktion durchaus Methode mehrerer Objekte sein.

```
> var john = { name : "John", saiten : 6 };
> var paul = { name : "Paul", saiten : 4 };
> var meetAndGreet = function () { return "Hello, I am " + this.name + ", my instrument has " + this.
    saiten + " strings." };
> john.meet = meetAndGreet;
> paul.greet = meetAndGreet;
> john.meet();
'Hello, I am John, my instrument has 6 strings.'
> paul.greet();
'Hello, I am Paul, my instrument has 4 strings.'
> meetAndGreet();
'Hello, I am undefined, my instrument has undefined strings.'
```

Listing 2.5: Eine Funktion als Methode der Objekte `john`, `paul`, und des globalen Objekts

Im Zusammenhang mit Funktionen erhält man auch eine weitere Möglichkeit zur Objekterzeugung. Dazu kann jede beliebige Funktion als Konstruktor eingesetzt werden, indem sie mit vorangestelltem Stichwort `new` aufgerufen wird.

```
> function Mitglied (name, saiten) { this.name = name; this.saiten = saiten; }
> new Mitglied("George", 6);
{ name: 'George', saiten: 6 }
```

Listing 2.6: Erzeugung eines Objekts mittels Konstruktor-Funktion

Es ist dabei reine Konvention, den Funktionsnamen von Konstruktoren groß zu schreiben. In diesem Zusammenhang verweist `this` auf das neu erstellte Objekt. Bisher scheint es, als könnte man den selben Effekt erzielen, indem man die Funktion `Mitglied` als Methode eines leeren Objekts aufruft. Beim konstruktivischen Gebrauch einer Funktion wird allerdings eine spezielle interne Eigenschaft des erzeugten Objektes gesetzt, die sich sonst nicht beliebig manipulieren lässt, die aber eine fundamentale Rolle in ECMAScripts Vererbungsmodell spielt.

Jedes native ECMAScript-Objekt `a` besitzt eine *Prototyp*-Eigenschaft, deren Wert ein anderes Objekt oder `null` ist. Ist der Wert ein Objekt, wird dieses dementsprechend als der *Prototyp von `a`* bezeichnet. Jedes Objekt erbt alle Eigenschaften seines Prototyps, so es sie nicht durch einen eigenen Wert unter selbem Namen verdeckt.

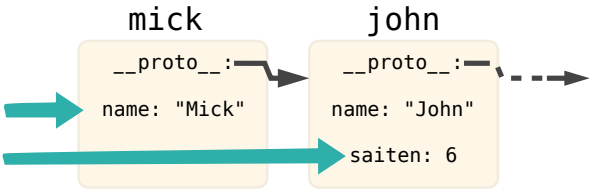

Abbildung 2.1: `mick` erbt Eigenschaft `saiten`, aber nicht `name` seines Prototyps

Da der Prototyp eines Objekts selbst einen Prototypen besitzen kann, spricht man auch von einer *prototype chain*, bei der Eigenschaften über mehrere Knoten

hinweg vererbt werden können.

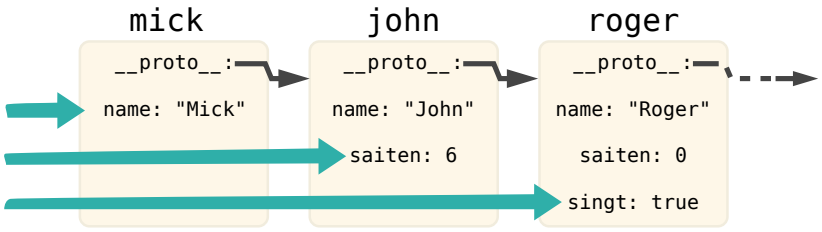

Abbildung 2.2: `mick` erbt über `john` die Eigenschaft `singt` von `roger`

Dass der Prototyp eines Objekts unter dem Namen `__proto__` tatsächlich zugänglich ist, ist eine Eigenheit einiger Implementationen und nicht Teil des Standards. Generell besitzt aber jedes Objekt eine entsprechende Eigenschaft.[6] Diese Vererbungsmethode kann dem Programmierer aber nur dienlich sein, wenn eine Möglichkeit besteht, diese Eigenschaft zu beeinflussen. Hier kommen die Konstruktor-Funktionen zum Tragen. Da auch Funktionen Objekte sind, können ihnen Eigenschaften zugewiesen werden. Beim Erzeugen eines Objekts mit Hilfe einer Konstruktor-Funktion wird dasjenige Objekt als Prototyp eingetragen, das unter der Eigenschaft `prototype` der Konstruktor-Funktion referenziert ist. Es ist daher sinnvoll, durch die Verwendung des Begriffs `__proto__` für den Prototypen eines Objekts und des Begriffs `prototype` für die *Prototyp*-Eigenschaft einer Konstruktor-Funktion, eine klare Terminologie zu schaffen.

Zu einer Funktion `A` wird `A.prototype` beim Erzeugen von `A` mit einem leeren Objekt initialisiert. Dieses kann manipuliert oder ersetzt werden, um den Prototypen der durch `A` erzeugten Objekte fest zu legen.

```
1 > function Mitglied (name, saiten) { this.name = name; this.saiten = saiten; }
2 > Mitglied.prototype.greet = function () { return "Hello, I am " + this.name + ", my instrument has " +
      this.saiten + " strings." };
3 > var pete = new Mitglied("Pete", 0);
4 > pete.greet();
5 'Hello, I am Pete, my instrument has 0 strings.'
```

Listing 2.7: Das neue Objekt `pete` besitzt über seinen Prototypen die Methode `greet`.

Funktionen sind in ECMAScript *first-class*-Objekte, können also als vollwertiges Objekt behandelt und zum Beispiel in Variablen gespeichert und als Funktionsparameter übergeben werden. Vor allem letzteres ist von enormer Bedeutung für die eventbasierte Programmierung.

[6] Im ECMAScript-Standard werden interne Eigenschaften wie die *Prototyp*-Eigenschaft durch eckige Klammern gekennzeichnet: *[[Prototype]]* Da die Verwendung der eckigen Klammern in Eigenschaftsnamen die Punkt-Notation (siehe Listing 2.1) als Zugriffsmöglichkeit ausschließen würde, wäre sie in Implementationen, die Zugriff auf interne Eigenschaften gestatten wollen, unvorteilhaft.

2.2.3 Moderner Einsatz

JavaScript[7] hätte vielleicht noch einige Zeit die Rolle der Scripting-Sprache für Kleinstinteraktionen und Spielereien in Websites gespielt, die es in den 1990er Jahren inne hatte, und wäre dann durch eine andere Technologie ersetzt worden. Mit dem Aufkommen von *Ajax*-Anwendungen *Ein ursprünglich für Asynchronous JavaScript And XML stehender Begriff aus dem Jahr 2005, wobei heute verschiedene weitere Datenformate außer XML zum Einsatz kommen. Details können [?] entnommen werden.* , also der asynchronen Kommunikation mit dem Webserver aus einem Webdokument heraus, veränderte sich die Rolle von JavaScript aber in kürzester Zeit zu einer tragenden. Viele Webanwendungen und damit ganze Geschäftsmodelle hingen nun mehr und mehr von JavaScript ab. Um mit der weiterhin mängelbehafteten Sprache und den Browserinkonsistenzen arbeiten zu können, waren und sind viele Mühen notwendig. Einige der Strategien seien im Folgenden kurz beschrieben.

Neuentwurf der Sprache

Innerhalb der letzten zehn Jahre gab es Bestrebungen, eine Revision des ECMA-Script-Standards zu verfassen. Diese vor allem auch von Brendan Eich getragene Initiative, sah unter anderem Klassen und Interfaces und eine feinere Typisierung der Sprache vor [ECM07]. Diese Bemühungen waren sehr umstritten, nicht zuletzt wegen der ungeklärten Frage, ob man die Browsernutzer innerhalb einer angemessenen Zeit zur Umstellung auf neue Clients würde bewegen können. Die Arbeit am sogenannten *ECMAScript 4* wurde nie zu Ende geführt und man hat sich mit der inzwischen veröffentlichten 5. Revision des Standards für eine behutsame Überarbeitung der Sprache entschieden, die näher an den ursprünglichen Konzepten geblieben ist [ECM09]. Viele moderne Browser setzen bereits Teile dieses Standards um. An einer neuen, tiefgreifenderen Revision der Sprache versucht man sich nun erneut unter dem Namen *Harmony* [Eic08].

Zähmung der Sprache

Bibliotheken Eine beliebte und sehr erfolgreiche Lösung für die inkonsistenen Browser-APIs stellen diverse JavaScript-Bibliotheken dar, die eine Abstraktionsschicht zwischen Browser und Webentwickler legen und damit eine konsistente und bequeme API zur Traversierung und Manipulation des Dokumentenbaums,

[7] Im folgenden Abschnitt geht es wieder um JavaScript als ganzes und der Begriff *ECMAScript* wird bis auf wenige Ausnahmen im Rest der Arbeit keine besondere Rolle spielen.

sowie für *Ajax*-Anfragen bereitstellen können. Viele dieser Bibliotheken gleichen nicht so sehr Schwächen des Sprachkerns aus. Beliebte Beispiele dieser Kategorie sind *jQuery*, *Prototype*, *MooTools* und *Dojo*.

BENUTZUNGSKONVENTIONEN Eine wichtige Rolle bei der Popularisierung konfliktvermeidender JavaScript-Praktiken spielt Douglas Crockford, der zahlreiche Aufsätze und ein Buch zu Themen rund um JavaScript verfasst hat. Crockford propagiert *subsetting* im Umgang mit Programmiersprachen, womit gemeint ist, dass der letzendliche Umfang einer Sprache nicht allein von ihren Designern, sondern auch von ihren Nutzern aktiv bestimmt werden sollte. Durch die Nichtverwendung einiger von JavaScripts problematischen Merkmalen soll so die Qualität der benutzten Sprache gehoben werden. Da JavaScripts stark funktionaler Charakter mächtige Möglichkeiten eröffnet, stehen dem Programmierer selbst bei Verzicht auf einige andere Bestandteile zumindest theoretisch bewiesenermaßen weiterhin alle Wege offen. Heute sind viele der Grundideen Crockfords weit verbreitet und viele weitere Akteure treiben die Benutzungsmuster voran. Auf die vielen Weisen einzugehen, in denen JavaScript heute verwendet wird, wäre ein interessantes Thema für sich. Hier soll nur kurz auf eine Methode zur Datenkapselung eingegangen werden, die auch in der vorliegenden Arbeit zum Einsatz kam.

MODUL-OBJEKTE Die Problematik des einen globalen Namensraumes vor allem bei der Kombinationen mehrerer JavaScript-Komponenten wurde schon genannt. Durch die Verwendung von Objekten zur Bündelung der Bestandteile einer Komponente, ist es möglich den globalen Namensraum mit nur einem Namen zu *belasten* und alle weiteren Programmbestandteile als Untereigenschaften des so benannten Objekts verfügbar zu machen.

```
1 > var Gruppe = {}; // erzeuge leeres Objekt
2 > Gruppe.mitglieder = []; // erzeuge Mitglieder-Array
3 > Gruppe.Mitglied = function (name, saiten) { Gruppe.mitglieder.push(this); this.name = name; this.
      saiten = saiten; };
4 > new Gruppe.Mitglied("Ringo", 0);
5 { name: 'Ringo', saiten: 0 }
6 > Gruppe.mitglieder
7 [ { name: 'Ringo', saiten: 0 } ]
```

Listing 2.8: Modulerzeugung mittels Objektpräfix

Durch die Verwendung des `Gruppe`-Objekts als Präfix wurde ein eigener Namensraum geschaffen. Allerdings hat dieses Methode eine Schwachstelle. Möchte man eine Komponente des Moduls ansprechen, muss man an vielen Stellen über das `Gruppe`-Objekt gehen. Man sieht das beispielhaft im Konstruktor

`Gruppe.Mitglied`, der das neue Objekt in die Liste `Gruppe.mitglieder` einträgt. Kommt es nun bei der Zusammenstellung mehrerer Module zu einem Konflikt der äußeren Namen, muss eines der Module an eventuell vielen Stellen im Programmcode angepasst werden. Abhilfe schafft das Erzeugen eines wirklichen Namensraums mit Hilfe einer Funktion.

```
1 var Gruppe = (function (Gruppe) {
2   Gruppe.mitglieder = []; // erzeuge Mitglieder-Array
3   Gruppe.Mitglied = function (name, saiten) { Gruppe.mitglieder.push(this); this.name = name; this.
        saiten = saiten; };
4
5   return Gruppe;
6 })({}); // uebergebe leeres Objekt
```

Listing 2.9: Modulerzeugung mittels flexiblem Objektpräfix

Eine Funktion mit dem formalen Parameter `Gruppe` wird definiert und sofort mit einem leeren Objekt als Argument ausgeführt. Die Komponenten werden dem übergebenen Objekt hinzugefügt und dieses zum Schluss als Funktionswert zurückgegeben. Kommt es in diesem Fall zu einem Namenskonflikt, reicht es aus, den Variablennamen `Gruppe` im äußeren Namensraum zu ändern. Im Inneren der Funktion ist `Gruppe` als formaler Parameter Teil des Funktionsnamensraumes und kann problemlos angesprochen werden. Diese und viele weitere Muster, samt ihren Vor- und Nachteilen, werden in [Cro08], Blogs und Fachartikeln vorgestellt und diskutiert.

COMMUNITY-STANDARDS Durch die zunehmende Verbreitung von JavaScript-Fähigkeiten und Praktiken und die durch den Aufbau des Web bedingte Notwendigkeit, Nutzereingaben auf Client **und** Server zu validieren, stieg in den vergangenen Jahren das Interesse an serverseitigen JavaScript-Anwendungen. Projekte wie *narwhal*[8] und *node.js*[9] machen Gebrauch von den effizienten JavaScript-Interpretern, die seitens der Browserhersteller in jüngeren Jahren als Open-Source-Projekte entwickelt wurden. Je nach Bedarf der einzelnen Projekte werden dabei Schnittstellen für Zugriff auf Dateisystem, Netzwerk und weitere Ressourcen hinzugefügt. Um nicht Geschichte zu wiederholen und die APIs der verschiedenen Implementationen divergieren zu lassen, bemüht man sich im Projekt *CommonJS*[10] um Quasistandards, die eine weitgehende Kompatibilität der Umgebungen ermöglichen sollen. Dem Wunsch nach einer einheitlichen Programmorganisation in Server und Client ist man indes nur teilweise näher gekommen. Das synchrone

[8] `http://narwhaljs.org/`
[9] `http://nodejs.org/`
[10] `http://www.commonjs.org/`

Laden von Modulen, wie es in serverseitigem JavaScript üblich ist, ist mit der asynchronen Funktionsweise des clientseitigen Ladens schlecht verträglich. Möchte man, wie es bei dieser Arbeit der Fall war, eine JavaScript-Anwendung in Browser und Kommandozeile verwendbar machen, muss der modular organisierte Quellcode für den Browser zuvor passend umstrukturiert werden.

Über die aktuellen Entwicklungen in diesem sehr dynamischen Feld ließe sich noch viel sagen. Es sollte aber klar geworden sein, dass JavaScript eine ungewöhnliche Sprache ist, die sich in bedachten Händen vielseitig einsetzen lässt. Durch die Investitionen der Browserhersteller gibt es inzwischen eine Auswahl sehr performanter Umgebungen, die den Browser als Applikationsplattform immer attraktiver und realistischer machen. Gleichzeitig gilt es, bestehende Browser-Installationen zu bedienen, die aus Generationen vor ECMAScript 5 und JavaScript-Bytecode-Interpretern stammen.

2.3 Compilerbau mit `lex` & `yacc`

It's a poor sort of memory that only works backwards.
—The White Queen in *Through the Looking-Glass*

Bei der Entwicklung eines Compilers ist es eine große Erleichterung, Teile der Arbeit an automatische Werkzeuge abgeben zu können. Ein seit vielen Jahren bekanntes und erprobtes Werkzeug, das aus geeigneten Angaben zu lexikalischer und grammatikalischer Syntax einer Sprache einen Parser generieren kann, ist das Duo `lex` & `yacc`. Zwar wird in dieser Arbeit der Parser-Generator *Jison* verwendet, eine JavaScript-Adaption des *GNU Bison*-Programms. Dieses ist allerdings selbst eine rückwärtskompatible Erweiterung des `lex`-`yacc`-Prinzips. In diesem Abschnitt soll ein Einblick in die grundlegende funktionelle Organisation aller genannter Programme gegeben werden.

2.3.1 Kontextfreie Sprachen

Beim Entwurf einer Programmiersprache ist es von einiger Bedeutung, die Syntax so zu gestalten, dass der einem Programmtext entsprechende Syntaxbaum in effizienter Weise erstellt werden kann. Ein sinnvolles Grundkriterium ist dafür die Beschränkung auf durch kontextfreie Grammatiken darstellbare Sprachen. Nicht alle Programmiersprachen fallen unter diese Kategorie. Durch das Verschieben einiger syntaktischer Entscheidungen auf einen Zeipunkt nach dem eigentlichen

Parsing, ist es möglich, auch für solche Sprachen Parsergeneratoren mit Hilfe von `yacc` und ähnlichen Werkzeugen zu erstellen. Der Autor von C++, Bjarne Stroustrup, beschreibt in [Str94] einige der zusätzlichen Schwierigkeiten, die es in diesem Fall zu lösen gilt.

Zwar ist Java syntaktisch an C angelehnt, die Sprache wurde aber so entworfen, dass man kontextfreie Grammatiken für ihre grammatikalische Syntax angeben kann [GJSB05, 9].

Eine kontextfreie Grammatik kann gegeben werden durch die Angabe vierer definierender Merkmale,

1. einer endlichen Menge von **Terminalsymbolen**, die das Alphabet der Sprache bilden,

2. einer endlichen Menge von **Nichtterminal-Symbolen**, die eine syntaktische Kategorie bezeichnen,

3. einer endlichen Menge von **Produktionen**, die durch ihren *Rumpf* angeben, zu welcher Folge von Terminal- und Nichtterminal-Symbolen man vom durch den *Kopf* gegebenen Nichtterminal-Symbol übergehen darf,

4. sowie zuletzt einem **Startsymbol** aus der Menge der Nichtterminal-Symbole, das die syntaktische Kategorie der Grammatik bezeichnet, die jeden gültigen Ausdruck der Sprache umfassen kann.

Ist $\mathbf{a}$ ein Terminalsymbol und sind $\mathbf{A}$ und $\mathbf{B}$ Nichtterminal-Symbole, kann eine kontextfreie Produktion beispielsweise als $\mathbf{A} \rightarrow \mathbf{a}\ \mathbf{B}$ notiert werden. $\mathbf{a}\ \mathbf{B} \rightarrow \mathbf{A}$ ist hingegen keine kontextfreie Produktion, da ihr Kopf nicht aus genau einem Nichtterminal-Symbol besteht. Für spätere Erklärungen des Parsings, werden zunächst noch einige termini technici benötigt, die ausführlich in [ALSU08] beschrieben werden.

ABLEITUNG Man erhält gültige Ausdrücke einer gegebenen kontextfreien Grammatik, durch eine *Ableitung* aus dem Startsymbol. Man notiert dazu eingangs das Startsymbol. Anschließend notiert man solange Zeichenketten, die man erzeugt, indem man in der zuletzt notierten Zeichenkette ein Nichtterminal-Symbol durch den Rumpf einer Produktion ersetzt, die das entsprechende Nichtterminal als Kopf enthält, bis die zuletzt notierte Zeichenkette nur noch Terminalsymbole enthält. Diese zuletzt notierte Zeichenkette ist ein gültiger Ausdruck der Sprache. Man spricht im besonderen von einer *Linksableitung*, wenn bei jedem Ersetzungsschritt

stets das äußerste linke Nichtterminal-Symbol gewählt wird. Wird hingegen stets das äußerste rechte Nichtterminal-Symbol ersetzt, spricht man von einer *Rechtsableitung*.

REDUKTION Beim Parsing einer Programmiersprache möchte man nicht nach Zufall gültige Programmtexte aus der Grammatik ableiten, sondern gegebene Programmtexte auf ihre syntaktische Struktur analysieren. Dazu ist es nötig von einer gegebenen Zeichenfolge auf einen Baum syntaktischer Kategorien zu schließen. Das Zusammenfassen einer Folge von Terminal- und Nichtterminal-Symbolen zu einem Nichtterminal-Symbol nennt man eine *Reduktion*. Dabei sind offensichtlich nur solche Reduktionen sinnvoll, die vom Rumpf einer Produktion der Grammatik zum zugehörigen Kopf übergehen. Eine *Linksreduktion* ist die Umkehrung einer Linksableitung, eine *Rechtsreduktion* entsprechend die Umkehrung einer Rechtsableitung.

2.3.2 FUNKTIONSWEISE DER PARSERGENERATOREN

Die Terminalsymbole einer kontextfreien Grammatik sind hinsichtlich der grammatikalischen Struktur atomar, können also nicht weiter zerlegt werden. Hinsichtlich der lexikalischen Struktur können sie allerdings wiederum selbst komplexe Ausdrücke sein. Bei Parsern unterscheidet man daher in der Regel zwei Aufgabenbereiche. Der **Lexer** übernimmt die Analyse der lexikalischen Struktur eines Programmtextes, teilt dem **Parser** also mit, um welches Nichtterminal es sich bei der aktuell betrachteten Eingabe handellt. Der Parser benutzt diese Information um die grammatikalische Struktur des Programmtextes zu analysieren.

Die Unix-Programme `lex` und `yacc` sind in entsprechender Weise aufgeteilt. Aus einer Datei, die Regeln zur Erkennung verschiedener Tokentypen – wobei ein Tokentyp einem Terminalsymbol in der Grammatik entspricht – enthält, erstellt `lex` einen Lexer. Dieser kann als Subroutine im Parser eingesetzt werden, der in ähnlicher Weise von `yacc` aus einer zweiten Datei, die die Produktionen der Grammatik enthält, erstellt wird. Die Darstellung der Spezifikationsformate folgt [ALSU08, Kapitel 3.5 und 4.9] und [Fre11].

`lex`-FORMAT UND LEXER

Eine `lex`-Lexer-Spezifikation besteht aus drei durch die Zeichen `%%` getrennten Abschnitten. Im **Deklarationsabschnitt** werden Variablen, Konstanten und benannte Muster zur späteren Verwendung vereinbart. Im **Übersetzungsregelab-**

schnitt werden die eigentlichen Muster zur Erkennung der Tokentypen notiert. Im dritten Abschnitt können zuletzt **Hilfsfunktionen** zur Verwendung im Übersetzungsregelabschnitt vereinbart werden. Listing 2.10 zeigt schematisch eine Lexer-Spezifikation, wie sie beim Erkennen einfacher Rechenoperationen zum Einsatz kommen könnte.

```
1 %{ /* optionale Konstantendeklaration */ %}
2 ziffer                        [0-9]
3 ganzzahl                      [1-9]{ziffer}*
4 %%
5 [ \t\n]+                      { /* Leerzeichen ignorieren */ }
6 {ganzzahl}                    { /* speichere Zahl und gib zurueck: Tokentyp ZAHL */ }
7 "+"                           { /* gib zurueck: Tokentyp PLUS */ }
8 %%
9 /* optionale Hilfsfunktionen */
```

Listing 2.10: Lexer-Spezifikation im `lex`-Format

Bei der Definition der Tokentyp-Muster kommen reguläre Ausdrücke zum Einsatz, die die lexikale Syntax des Tokentyps angeben und erkennen. Die im Deklarationsabschnitt benannten Muster können in anderen Musterdeklarationen oder Übersetzungsregeln verwendet werden, um eine bessere Programmstruktur und -lesbarkeit zu erzielen.

Der aus einer solchen Spezifikation erzeugte Lexer verwandelt einen Eingabestrom von Zeichen sukzessive in einen Ausgabestrom von Tokentypen. Dabei ruft der Parser eine Routine des Lexers auf, die diesen veranlasst, das nächste Token zu bestimmen. Dabei sucht der Lexer die längstmögliche Zeichenfolge, die von einer der Übersetzungsregeln erkannt wird. Existiert mindestens ein, aber kein eindeutig bestimmter längster Treffer, wird die zuerst deklarierte Übersetzungsregel angewandt. Dabei kann der Lexer den Text des Tokens oder eine verarbeitete Version davon in eine mit dem Parser geteilte Variable schreiben, um diesem zum Beispiel den Wert einer Zahl verfügbar zu machen. Zusätzlich kann eine mit dem Parser vereinbarte Konstante zurückgegeben werden, die den erkannten Tokentypen angibt. Alternativ kann der Lexer die Kontrolle noch nicht an den Parser zurück geben und ein weiteres Token erkennen – zum Beispiel um bedeutungslose Leerzeichen zu übergehen. Findet der Lexer keine passende Übersetzungsregel, wird die Suche mit einem Fehler beendet.

`yacc`-Format und Parser

Die Angabe einer `yacc`-Parser-Spezifikation erfolgt ebenfalls in drei Abschnitten. Im **Deklarationsabschnitt** kann die Deklaration von Hilfsvariablen untergebracht

und eine Liste der zu erwartenden Tokentypen gegeben werden. Im **Übersetzungsregelabschnitt** werden die Produktionen der Sprachgrammatik zusammen mit den zugehörigen *semantischen Aktionen* angegeben. Der dritte Abschnitt bietet wieder Platz für **Hilfsfunktionen**. Listing 2.11 knüpft an das `lex`-Beispiel 2.11 an und zeigt den Aufbau einer Parser-Spezifikation.

```
1  %{ /* optionale Hilfsvariablen */ %}
2  %token ZAHL /* vereinbare Tokentyp ZAHL */
3  %token PLUS /* vereinbare Tokentyp PLUS */
4  %%
5  addition                   /* Kopf */
6    : ausdruck PLUS ausdruck /* Rumpf */
7      { $$ = $1 + $3; }      /* semantische Aktion */
8    ;                        /* Ende der Uebersetzungsregel */
9  ausdruck
10   : addition               /* Rumpf 1 */
11     { $$ = $1; }
12   | ZAHL                   /* Rumpf 2 */
13     { $$ = $1; }
14   ;
15 %%
16 /* optionale Hilfsfunktionen */
```

Listing 2.11: Parser-Spezifikation im `yacc`-Format

Im Beispiel besitzt der Kopf `ausdruck` zwei Rümpfe. Das entspricht einer einfachen Kurzschreibweise zweier Produktionen mit identischem Kopf. Allgemein kann für zwei oder mehr Rümpfe die Übersetzungsregel notiert werden, indem die Rümpfe durch den *Pipe*-Charakter `|` getrennt werden.

In der semantischen Aktion eines Rumpfes wird im Regelfall der Wert des Kopfes, `$$`, aus den Werten `$1`, ..., `$n` des n-gliedrigen Rumpfes bestimmt.

Das deutet schon auf die Funktionsweise des durch `yacc` generierten Parsers hin, der das Parsing *bottom-up* durchführt, das heißt mit Terminalsymbolen beginnend durch Reduktion zu Nichtterminalen und schließlich zur Spitze, dem Startsymbol, gelangt. Dabei arbeitet der Parser als *Shift-Reduce*-Parser, der so lange syntaktische Einheiten und ihre Werte auf einen Stapel legt (*shift*), bis Teile des Stapels durch Reduktion zu einer neuen Einheit zusammengefasst werden können (*reduce*). Der Parser fragt dazu vom Lexer schrittweise neue Token ab, und entscheidet je nach Rückgabe, ob ein *shift*- oder *reduce*-Schritt notwendig ist. Aus Effizienzgründen wird dabei an die spezifizierte Grammatik die Anforderung gestellt, dass die Entscheidung zwischen *shift* und *reduce* alleine aufgrund der auf dem Stapel liegenden Einheiten und dem zuletzt gelesenen Token getroffen werden kann. Dabei wird der Programmtext von links nach rechts durchlaufen, weshalb man Parser diesen Typs auch als *LR(1)-Parser* klassifiziert: Die Einga-

be wird von **links** nach rechts durchlaufen, wobei **Rechtsreduktionen** anhand **eines** *Lookahead*-Tokens durchgeführt werden. Theoretisch soll dabei der ganze Eingabetext von links nach rechts gelesen und zum Ende auf das Startsymbol der Grammatik reduziert werden. In diesem Fall *akzeptiert* der Parser die Eingabe, wenn der Stapel nur das Startsymbol enthält und kein Eingabetext mehr übrig ist. In der Praxis bleibt es aber dem Autor der `yacc`-Spezifikation überlassen, auch unvollendete Parsing-Ergebnisse zurück zu geben und die Ausführung zu beenden. Falls anhand Stapel und aktuellem Token weder *shift* noch *reduce* oder *accept* möglich ist, wird der Kontrollfluss an eine Fehlerbehandlungsroutine übergeben.

KONFLIKTE Die Beschränkung auf Grammatiken, die von einem LR(1)-Parser erkannt werden können, hat zur Folge, dass einige kontextfreie Sprachen nicht oder nicht ohne weiteres von `yacc`-erzeugten Parsern verarbeitet werden können, wenn sich für sie keine konformen Grammatiken konstruieren lassen. Ungeachtet der letztendlichen Kategorie der Sprache, kann es bei Verwendung einer nicht LR(1)-konformen Grammatik zu zwei Arten von Konflikten kommen. Im Falle eines *Shift/Reduce*-Konflikts kann der Parser anhand von Stapel und aktuellem Token nicht entscheiden, ob er den Token auf den Stapel verschieben oder eine Reduktion vornehmen soll. `yacc` benutzt zur Konfliktlösung die Strategie, im Zweifel einen *shift* durchzuführen. Bei einem *Reduce/Reduce*-Konflikt kann anhand Stapel und aktuellem Token die Wahl zwischen zwei oder mehr möglichen Reduktionen nicht getroffen werden. Der Konflikt wird gelöst, indem anhand der in der Spezifikation zuerst erscheinenden Produktion reduziert wird.

VERWENDUNG In der Regel werden die semantischen Regeln dafür benutzt, einen *abstrakten Syntax-Baum* zu konstruieren, indem die Werte der Rumpfglieder als Kinder des Kopfes verankert werden. Bei Verwendung des Parsers als Komponente eines Compilersystems kann dann an passender Stelle, etwa bei Erreichen des Startsymbols, eine Referenz auf den Parsebaum an die aufrufende Komponente zurückgegeben werden. Die erzeugte Baumrepräsentation der Programmstruktur kann dann verwendet werden, um das Programm auf Fehler zu prüfen und nach etwaiger Optimierung die Übersetzung in die Zielsprache vorzunehmen.

KAPITEL 3

ANSATZ UND UMSETZUNG

Wie im Abschnit 2.1 zu Java beschrieben, werden Java-Programme normalerweise weder in Maschinencode übersetzt, noch direkt interpretiert, sondern in einen speziellen Bytecode übersetzt, welcher dann interpretiert wird. Bei der Konzeption der vorliegenden Arbeit stellte sich daher die Frage, welche Form eines Java-Programms im Browser ausgeführt werden soll. Es kamen drei Modelle in Frage.

1. Entwicklung einer virtuellen Maschine für Java-Bytecode in JavaScript
2. Direkte Übersetzung von Java-Quellcode in eine mittels JavaScript ausführbare Form
3. Entwicklung eines Java-zu-Java-Bytecode-Compilers und einer virtuellen Maschine für Java-Bytecode in JavaScript

Da Modell 1 eine vorherige Übersetzung der jeweiligen Quelldateien in Bytecode voraussetzt, wurde es von der Anforderung, dass die Programme vor der Ausführung manipulierbar sein sollten, ausgeschlossen[1]. Modell 2 erzeugt durch die fehlende *Virtual Machine* das Gefühl, man hätte es nicht *wirklich* mit Java zu tun und bietet weniger Möglichkeiten als eine JavaScript-Implementation der JVM. Dafür stellt das Bearbeiten der Java-Programme im Browser kein Problem dar. Modell 3 vereint zuletzt die positiven Aspekte der Modelle 1 und 2, ohne ihre Probleme zu teilen. Gleichzeitig ist es aber eine Umsetzung beider Modelle.

Da sowohl Modell 1, als Modell 2 den Umfang einer Bachelorarbeit bereits mehr als füllten, musste eine Entscheidung auf diese Auswahl beschränkt bleiben.

[1] Modell 1, und damit Modell 3, haben andere, sehr gewichtige Vorteile, auf die in der Alternativschau in 4.2 eingegangen wird.

Wegen des schweren Mangels des alleinstehenden Modell 1 fiel die Wahl dabei auf Modell 2.

Bei der vorliegenden Arbeit soll also Java-Quellcode in eine mittels JavaScript ausführbare Form übersetzt und in einer JavaScript-Laufzeitumgebung ausgeführt werden. Dabei soll der umzusetzende Umfang auf den Sprachkern beschränkt sein und die Nutzeraktion daher mittels Textein- und -ausgaben erfolgen. Dazu soll es möglich sein, im Browser eine kommandozeilenartige Oberfläche zu simulieren, wie sie typischerweise zur Ausführung einfacher Java-Programme verwendet wird. Da zum Einsatz in einer Lernumgebung vor allem das Verhalten als *echtes* Java im Vordergrund steht, soll das Ziel ein möglichst javaähnliches Verhalten sein, ohne, dass die Implementationsdetails überereinzustimmen haben.

Für eine entwicklerfreundliche Umgebung wird dabei das Vorhandensein von browserexternen JavaScript-Interpretern genutzt, die eine Entwicklung auf der Kommandozeile erlauben.

Mit den geklärten Parametern des Projekts kann auch die Namenswahl erklärt werden. Der von der Mozilla Foundation in Java geschriebene JavaScript-Interpreter *Rhino* ist nach David Flanagans JavaScript-Standardwerk *JavaScript – The Definitive Guide* benannt, auf dessen Vorderseite ein Rhinozeros abgebildet ist. Nach der selben Logik musste dieses Projekt nach dem Titeltier des Java-Standardwerks des selben Verlags benannt werden – einem Tiger. Da das Wappentier entsprechend der Projektstatuten freundlich und hilfsbereit sein sollte, hat sich der Autor gegen *Shirkan* und für *hobbes* entschieden.

3.1 Entwicklungsmethodik

The beetle king slammed down his fist,
"I didn't ask what it seems like, I asked what it is."
—The King Beetle in *The King Beetle On A Coconut Estate*

In der objektorientierten Programmierung spricht man von *Ducktyping*, wenn die Verwendbarkeit eines Objekts an einer bestimmten Stelle im Programm nicht anhand von expliziter Typenannotation, Klassenzugehörigkeit, oder ähnlichem, sondern anhand der Eigenschaften und Methoden des Objekts entschieden wird[2]. Das Verfahren folgt dem Gedanken, dass es bei der Verwendung eines Gegenstands vor allem darauf ankommt, ob sich der Gegenstand so verhält, wie es

[2]siehe dazu zum Beispiel das Glossar der *Python*-Dokumentation unter `http://docs.python.org/glossary.html#term-duck-typing`

im Kontext der Verwendung von Nöten ist. Worum es sich *essentiell* handelt, ist innerhalb des Kontextes irrelevant.

Bei der Entwicklung von hobbes wurde diese Sichtweise auf den Java-Interpreter als Gegenstand übertragen. Durch die Entscheidung gegen die Umsetzung als Bytecode-Interpreter entfernte sich das Projekt bereits stark von den Implementationsdetails üblicher Java-Runtimes. Da das Ziel des Projektes aber ein Einsatz in Lernumgebungen ist, der dem Benutzer die Möglichkeit geben soll, zu Erkunden, wie sich ein Java-Programm in der Ausführung verhält, erfüllt hobbes seinen Zweck, wenn mit ihm ausgeführte Java-Programme sich so verhalten, wie sie es in typischen Java-Runtimes tun. Da die wohl typischste Runtime die offizielle *Java Runtime Environment* ist, wurde diese zum konkreten Vorbild gewählt. Um dabei eine gedankliche Trennung zwischen Java und der von hobbes unterstützen Teilmenge zu erleichtern, wurde letztere in der Entwicklung unter dem Namen *Vava* geführt.

3.1.1 Entwicklungs- und Testumgebung

Um die vielen Verhaltensanforderungen, die an Javas Semantik gebunden sind, in kontrollierter Weise implementieren und prüfen zu können, wurde ein testgetriebener Entwicklungsansatz gewählt. Dazu wurden Teile der offiziellen Sprachspezifikation, [GJSB05], in eine Sammlung von *YAML*-Dateien[3] extrahiert und zu jeder prüfbaren Bedingung eine Java-Quelldatei als Testfall eingetragen. Wie in Listing 3.1 zu sehen, kann ein Spezifikationspunkt durch mehrere Testdateien geprüft werden. Das macht vor allem dann Sinn, wenn ein Test die Ausgabe bei erfolgreicher Ausführung und ein anderer die Ausgabe bei fehlgeschlagener Kompilierung prüfen soll.

```
1  suite: "Operators on Integral Values"
2  section: 2
3  specifications:
4
5    - description: "Vava provides a number of operators that act on integral values, including numerical
         comparison (which results in a value of type boolean), arithmetic operators, increment and
         decrement, bitwise logical and shift operators, and numeric cast."
6      test_files:
7        - 'IntegralOperators.java'
8        - 'IntegralOperatorsMisused.java'
9
10   - description: "Operands of certain unary and binary operators are subject to numeric promotion."
11
12   - description: "Any value of any integral type may be cast to or from any numeric type."
13     test_files:
```

[3] ein *XML*-ähnliches, aber lesefreundlicheres Datenformat: `http://yaml.org/`

```
14      - 'Casting.java'
```

Listing 3.1: Ausschnitt aus einer Spezifikation während der Entwicklung[4]

Um jederzeit einen Überblick über die erreichte Einhaltung der Spezifikation erhalten zu können, wurde eine Reihe von *Rake-Tasks*[5] geschrieben, die anhand der *YAML*-Spezifikationen automatische Tests durchführen können. Dabei können einerseits nur die innerhalb einer kurzen Zeitspanne zuletzt bearbeiteten Spezifikationen geprüft werden, um keine Wartezeit in den Entwicklungsprozess zu bringen. Vor allem kann aber eine farbig markierte Version der Spezifikation als HTML-Dokument erzeugt werden, die eine schnelle visuelle Übersicht über den aktuellen Stand und etwaige Regressionen ermöglicht. Dabei werden erfolgreich getestete Spezifikationspunkte grün eingefärbt, unerfolgreich getestete rot, ungetestete gelb.

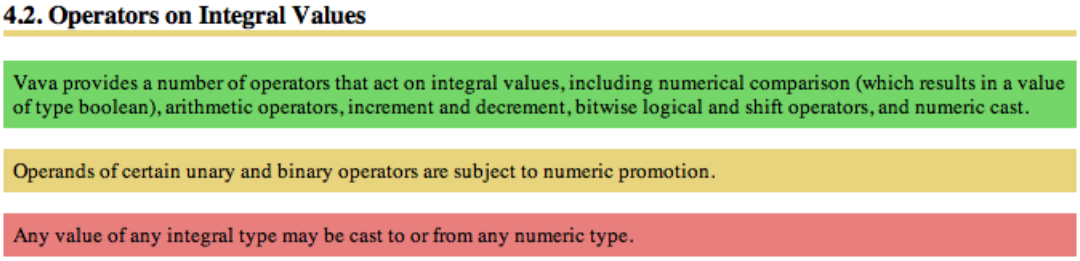

Abbildung 3.1: Farbig markierter Report während der Entwicklung

Der Erfolg oder Misserfolg eines einzelnen Tests wird durch die *Rake-Tasks* bestimmt, indem der Ausgabestring der Kompilierung und Ausführung des Test-Programms durch `javac` und `java` mit dem der Kompilierung und Ausführung durch `hobbes` verglichen wird. Nur bei gleicher Ausgabe wird der Test als Erfolg gewertet.

Diese Methode, die Entwicklung zu organisieren, erwies sich als ein sehr hilfreicher und zielführender Leitfaden in einem doch sehr umfassenden Projekt.

3.1.2 Duale Entwicklung

Um den automatischen Vergleich der Ausgabe von hobbes mit der von `javac` & `java` zu ermöglichen, war es notwendig, hobbes nicht nur in Browsern, sondern

[4] Spezifikationstext adaptiert aus [GJSB05]

[5] *Rake* steht für *Ruby Make* und erlaubt die Automatisierung von Aufgaben durch Ruby-Programme (`http://rake.rubyforge.org/`)

auch auf der Kommandozeile ausführen zu können. Aus den vielen zeitgenössischen Lösungen für browserexternes JavaScript wurde *node.js* ausgewählt, da es auf Google's JavaScript-Interpreter *V8* aufbauend eine performante Umgebung mit einigen nützlichen Standardmodulen für Ein- und Ausgabe bietet. Damit konnte gleichzeitig die Organisation des Programms mithilfe des dem *CommonJS*-Standard folgenden Modulsystems übersichtlicher gestaltet werden. Wegen der in Abschnitt 2.2.3 erwähnten Probleme bei der clientseitigen Implementation des *CommonJS*-Modul-Standards, musste für den Einsatz im Browser ein weiterer *Rake-Task* geschrieben werden, der aus den einzelnen JavaScript-Dateien, die in *node.js* durch das Laden als Modul verbunden wurden, eine einzige Quelldatei erzeugt. Um dabei, wie für *CommonJS*-Module vorgesehen, jedem Modul einen eigenen Namensraum zu geben, wird eine Methode verwendet, die der in Listing 2.9 demonstrierten ähnelt.

Zuletzt muss, abhängig von der Ausführungsumgebung, ein jeweils passendes Modul für Ein- und Ausgaben, also eine jeweilige Entsprechung von `java.lang.System` geladen werden. Da im Sprachkern keine Möglichkeit für entsprechende Abfragen vorgesehen ist, die Umgebungsdetektion aber in keiner Umgebung zu einem Fehler führen soll, ist es üblich, zu prüfen, ob der Name `require` eine Funktion referenziert und gegebenenfalls noch, ob diese gewisse Eigenschaften besitzt. In der Browser-Version von hobbes wird dazu auf höchster Ebene innerhalb des hobbes-eigenen Namensraumes die lokale Variable `require` mit einem beliebigen String initialisiert. Nach bester Einschätzung des Autors, ist derlei in heutigem JavaScript *common practice*.

Ziel der Modularisierung und dualen Entwicklung war auch eine erhöhte Portabilität und Kompatibilität. Inwieweit dieses Ziel erreicht wurde, ist Thema des Abschnitts 4.1.

3.2 Parser-Generierung

Durch das Auffinden zweier unermesslich wertvoller Hilfsmittel konnte ein erheblicher Teil der zur Entwicklung eines Compiler nötigen Arbeit durch die Anpassung und Verwendung bereits bestehender Grundlagen vollbracht werden. Es handelt sich dabei um eine LR(1)-Grammatik für Java, sowie die in JavaScript verfasste *GNU-Bison*-Adaption *Jison*. In den beiden folgenden Abschnitten werden die dabei anfallenden Aufgaben beschrieben.

3.2.1 JAVA-LR(1)-GRAMMATIK

Die in [GJSB05] präsentierte Grammatik ist vornehmlich für den menschlichen Leser gedacht und nicht als Eingabe für `yacc`-artige Parsergeneratoren geeignet. Glücklicherweise enthalten frühere Auflagen der Spezifikation ein zusätzliches Kapitel, mit einer entsprechend adaptierten Grammatik. In der online verfügbaren ersten Auflage findet sich in Kapitel 19 eine LALR(1)-Grammatik der Java-Sprache[6]. LALR-Grammatiken sind eine besondere From von LR-Grammatiken, die mit einem Verfahren des Parsergenerators kompatibel sind, eine kleinere Parsertabelle zu erzeugen, ohne dabei *Reduce/Reduce*-Konflikte zu erzeugen. Da *Jison* sowohl LALR(1)-, als LR(1)-Grammatiken verarbeiten kann, war die LALR-Form ob der kleineren Parsergrösse im Zusammenhang mit dem Einsatz im Web zwar sehr willkommen, aber nicht unabdinglich.

Diese Version der Grammatik musste lediglich aus der menschenlesbaren Form ins `yacc`-Format übersetzt werden. Um die lexikalische Syntax der Tokentypen zu beschreiben, wurden die als kontextfreie Grammatiken gegebenen Definitionen aus [GJSB05] in reguläre Ausdrücke übersetzt. Hierbei trat nur im Fall von mehrzeiligen Kommentaren der Fall ein, dass die zur Verfügung stehende reguläre Sprache nicht ausreichte, den Tokentypen korrekt zu charakterisieren. Mit der Verfügbarkeit zusätzlicher Informationsspeicher mittels *Jison* konnte dieses Problem aber leicht behoben werden.

3.2.2 *Jison*

Jison ist ein von Zach Carter als Open-Source-Projekt entwickelter Nachbau von *Bison* in JavaScript[7]. Dabei dient *Bison* zwar als Vorlage, aufgrund der speziellen Anforderungen und Möglichkeiten, die beispielsweise die in JavaScript bereits vorhandene Funktionalität für reguläre Ausdrücke mit sich bringt, weichen die Spezifikations-Formate der beiden Programme aber in einigen Punkten voneinander ab[8].

Als Beispiele für den Übertragungsprozess sollen im Bereich der Lexer-Spezifikation die Gleitkommazahlsyntax und die Mehrzeilen-Kommentar-Syntax benutzt werden. Im Bereich der Parser-Spezifikation dienen dazu die Produktionen der `return`-Anweisung. Die Behandlung eines überraschendererweise auftretenden *Shift/Reduce*-Konflikts bei `if-else`-Konstrukten schließt den Abschnitt ab.

[6] `http://java.sun.com/docs/books/jls/first_edition/html/19.doc.html`

[7] `http://zaach.github.com/jison/`

[8] Diese Abweichungen wurden unter anderem vom Autor dieser Arbeit im Wiki des Projekts dokumentiert und sollten bei Interesse dort nachgeschlagen werden. (`https://github.com/zaach/jison/wiki/DeviationsFromBison`)

Lexikalische Spezifikation

Gleitkommazahlen Abbildung 3.2 zeigt die lexikalische Syntax der Gleitkommazahlen, wobei kursiv gedruckte Bezeichner Nichtterminal-Symbole und normal gesetzte Bezeichner Terminal-Symbole darstellen. Die links stehenden und von einem Doppelpunkt gefolgten Nichtterminal-Bezeichner geben den Kopf einer Produktion an. Die eingerückt stehenden Folgen von Bezeichnern geben den Rumpf einer Produktion an, wobei jede Zeile einen eigenen Rumpf beschreibt. Zusätzlich markiert eine im Subskript notierte *opt*-Kennzeichnung ein Symbol als optional. Zur Ausschreibung eines Rumpfes mit optionalen Symbolen wird der Rumpf einmal mit und einmal ohne jedes optionale Symbol notiert. Hinter der für *FloatingPointLiteral* notierten Regel verbergen sich also insgesamt $(2 \times 2 \times 2) + (2 \times 2) + 2 + 2 = 16$ Produktionen.

FloatingPointLiteral:
Digits . $Digits_{opt}$ $ExponentPart_{opt}$ $FloatTypeSuffix_{opt}$
. *Digits* $ExponentPart_{opt}$ $FloatTypeSuffix_{opt}$
Digits ExponentPart $FloatTypeSuffix_{opt}$
Digits $ExponentPart_{opt}$ *FloatTypeSuffix*

ExponentPart:
ExponentIndicator SignedInteger

ExponentIndicator: one of
e E

SignedInteger:
$Sign_{opt}$ *Digits*

Sign: one of
\+ -

FloatTypeSuffix: one of
f F d D

Digits:
Digit
Digits Digit

Digit:
0
NonZeroDigit

NonZeroDigit: one of
1 2 3 4 5 6 7 8 9

Abbildung 3.2: Lexikale Syntax der Gleitkommazahlen als kontextfreie Grammatik[9]

Um den resultierenden regulären Ausdruck lesbarer zu machen, können die in Abschnitt 2.3.2 beschriebenen Musterdeklarationen im Kopfteil einer `lex`-Spezifikation verwendet werden. In Listing 3.2 sind die den Nichtterminal-Kategorien entsprechenden regulären Ausdrücke zu sehen.

```
1 D               [0-9]           /* Digit        */
2 NZ              [1-9]           /* NonZeroDigit */
3 Ds              ("0"|{NZ}{D}*)  /* Digits       */
4 EXPO            ([Ee][+-]?{Ds}) /* ExponentPart */
```

Listing 3.2: Deklarierung benannter Muster im Kopf der `lex`-Spezifikation

[9] aus [GJSB96]

Dabei kennzeichen doppelte Anführungszeichen in *Jisons* regulären Ausdrücken ein oder mehr literale Zeichen. Durch die Umschließung mit geschweiften Klammern kann durch seinen Namen Bezug auf ein zuvor definiertes Muster genommen werden. Die übrigen Bestandteile der Muster verhalten sich wie in gewöhnlichen regulären Ausdrücken.

Im Übersetzungsregelabschnitt folgt dann nur noch eine Zeile mit dem der syntaktischen Kategorie *FloatingPointLiteral* entsprechenden regulären Muster. Abweichend von den C-Parsergeneratoren, in deren Tradition *Jison* steht, wird zur Tokentypidentifikation ein String zurück gegeben, dem, mittels einer vom Parser vorgehaltenen Tabelle, ein Zahlwert zugeordnet ist.

```
1 ({Ds}"."{Ds}?{EXPO}?[fFdD]?|"."{Ds}{EXPO}?[fFdD]?|{Ds}{EXPO}[fFdD]?|{Ds}{EXPO}?[fFdD])/([^\w]|$) {
      return 'FLOATING_POINT_LITERAL'; }
```

Listing 3.3: Übersetzungsregel für Gleitkommazahlen

MEHRZEILEN-KOMMENTARE Mittels regulärer Ausdrücke ist es bei Nichtbeachtung von Zeilenumbrüchen in den meisten heutigen Systemen leicht möglich, mehrzeilige Kommentare durch das Muster `/\*.*?\*/` zu erfassen. In Umgangssprache übersetzt, beschreibt es eine Zeichenkette, die mit einem *Slash* (`/`), gefolgt von einem Asterisk (`*`), beginnt, anschließend beliebig viele Zeichen enthält, solange in diesen kein Asterisk gefolgt von einem *Slash* vorkommt, und schließlich mit einem Asterisk und einem *Slash* endet. Ermöglicht wird das durch den `*`-Quantor, der den vor ihm stehenden Ausdruck kein- oder mehrmal wiederholt und das dahinter stehende Fragezeichen, das verhindert, das dabei der im Muster folgende Teil mit erfasst wird. Man nennt `*?` die *non-greedy*-Variante von `*`.

In `lex`' Regulären Ausdrücken gibt es allerdings keine Möglichkeit, Quantoren *non-greedy* zu machen. Das hat den Effekt, dass ein entsprechendes Muster den gesamten Text, der zwischen dem Anfang des ersten Mehrzeilen-Kommentars und dem Ende des letzten Mehrzeilen-Kommentars steht, als Kommentar erkennen würde. Da dies inakzeptabel ist, Mehrzeilen-Kommentare aber erlaubt sein müssen, muss eine etwas komplexere Erkennung das einfache Muster ersetzen.

Dazu unterstützt *Jison* die von `flex`, einer Weiterentwicklung von `lex`, eingeführten *Startbedingungen*. Der Lexer kann dabei in bestimmte benannte Zustände versetzt werden. Die Namen dieser Zustände lassen sich dann als Präfix der Übersetzungsregeln verwenden, um diese als nur für den jeweiligen Zustand gültig zu markieren[10].

[10] Zur generellen Funktionsweise der Startbedingungen, siehe das `flex`-Manual: `http://dinosaur.compilertools.net/flex/flex_11.html`
Zur Benutzung in *Jison* siehe die Dokumentation: `http://zaach.github.com/jison/docs/#lexical-analysis`

Im Deklarationsabschnitt der Spezifikation wird eingangs der Zustand `comment` vereinbart:

```
%s              comment
```

Listing 3.4: Ankündigung des `comment`-Zustands

Mithilfe des zusätzlichen Zustands kann nun das problematische Muster ersetzt werden, wie Listing 3.5 zeigt. Dazu wird bei Erkennen des Musters `/*` der Lexer in den `comment`-Zustand versetzt und die Kontrolle nicht an den Parser zurückgegeben. Solange der Lexer in diesem Zustand verbleibt, wird zunächst geprüft, ob als nächstes Token `*/` erscheint. Ist das nicht der Fall, wird genau ein beliebiges Zeichen erkannt, dabei aber die Kontrolle nicht abgegeben und daher direkt nach dem nächsten Token gesucht. Dieser Vorgang wird solange wiederholt, bis entweder der Eingabestrom abbricht und ein Fehler verursacht wird, oder der `comment`-Zustand durch das Erscheinen von `*/` verlassen wird. Anschließend wird die Tokensuche im normalen Zustand fortgeführt.

```
1 "/*"                {this.begin('comment');}    /* Versetzung in comment-Zustand bei Slash-Asterisk */
2 <comment>"*/"       {this.popState();}          /* Verlassen des comment-Zustand bei Asterisk-Slash */
3 <comment>.          {/* skip comment content*/} /* Ignoriere ein Zeichen im comment-Zustand */
```

Listing 3.5: Spezialregeln für `comment`-Zustand

Grammatikalische Spezifikation

Return-Anweisung Zur Umsetzung der durch [GJSB96] gegebenen LALR(1)-Grammatik soll mit der *Return*-Anweisung zunächst nur ein sehr einfaches Beispiel gegeben werden. Wieder markiert Kursivsatz Nichtterminal-Bezeichner und der *opt*-Kennzeichner optionale Symbole. Der in Abbildung 3.3 dargestellte Grammatikauszug enthält also zwei Produktionen, wobei ein *ReturnStatement* entweder aus dem Token `return`, gefolgt von dem Token `;`, oder aus dem Token `return`, gefolgt von einem Ausdruck (*Expression*), gefolgt von dem Token `;` bestehen kann.

ReturnStatement:
return $Expression_{opt}$;

Abbildung 3.3: Produktionen für *ReturnStatement*[11]

Entsprechend benötigt man in der in Listing 3.6 dargestellten `yacc`-Umsetzung zwei Rümpfe mit semantischen Aktionen, wobei im Falle eines vorhandenen

[11] aus [GJSB96]

Ausdrucks dieser dem Konstruktor des `ReturnStatement`-Knotens übergeben wird. Es ist zu beachten, dass der Tokentyp `return` vom Lexer erkannt wird.

```
1 return_statement
2   : 'return' expression
3     { $$ = new yy.ReturnStatement($2, @$); }
4   | 'return'
5     { $$ = new yy.ReturnStatement(@$); }
6   ;
```

Listing 3.6: Übersetzungsregel für *Return*-Statement

Der `ReturnStatement`-Konstruktor ist einer von vielen Konstruktoren, die je einer syntaktischen Kategorie entsprechen und dem fertigen Parser vor dem Parsevorgang zur Verfügung gestellt werden. Außer etwaigen Kindknoten übergibt der Parser dem Konstruktor in den meisten Fällen als letztes Argument ein Objekt mit Informationen zur Position des syntaktischen Konstrukts im Programmtext.

HÄNGENDES ELSE Das Problem des hängenden *Else* (*dangling else*) ist in der Compilerbau-Literatur wohlbekannt und ein typisches Beispiel eines *Shift/Reduce*-Konflikts. Abbildung 3.4 zeigt ein aus [ALSU08, 336] adaptiertes Schema. Mit dieser Grammatik bieten sich dem Parser, bei einem Stapel aus einem if-Token, einem *Ausdruck*, einem then-Token und einer *Anweisung*, und einem aktuellen else-Eingabetoken zwei Möglichkeiten. Einerseits die Reduktion der auf dem Stapel befindlichen Eingabe zu einem *Statement*, andererseits die Verschiebung des else-Tokens auf den Stapel.

Anweisung:
if *Ausdruck* then *Anweisung* else *Anweisung*
if *Ausdruck* then *Anweisung*
AndereAnweisung

Abbildung 3.4: Mehrdeutige LR(1)-Grammatik

Durch das in 2.3.2 beschriebene Verfahren `yaccs` bei *Shift/Reduce*-Konflikten wird dieser Fall durch Verschieben des else-Tokens korrekt behandelt. Nun enthält aber die in [GJSB96] gegebene Grammatik eine in Abbildung 3.5 gezeigte Besonderheit, die dem Parser eine vergleichbare Lösung unmöglich macht.

Die syntaktischen Kategorien *Statement* und *StatementNoShortIf* sind dabei nicht disjunkt und gewisse Eingabefolgen können sowohl zu *Statement* als zu *StatementNoShortIf* reduziert werden. Hier verbirgt sich also die Gefahr eines

[12] aus [GJSB96]

IfThenStatement:
if (*Expression*) *Statement*

IfThenElseStatement:
if (*Expression*) *StatementNoShortIf* else *Statement*

Abbildung 3.5: Grammatik mit *Reduce/Reduce*-Konflikt[12]

Reduce/Reduce-Konflikts. Durch die übliche Wahl der in der Spezifikaton zuerst erscheinenden Produktion kann dieses Problem aber nicht gelöst werden, da die Wahl der richtigen Reduktion nicht unabhängig vom darauf folgenden Token ist. Jede Anordnung der Produktionen für *Statement* und *StatementNoShortIf* hat zur Folge, dass entweder manche *IfThenStatement*s oder manche *IfThenElseStatement*s nicht richtig erkannt werden, da durch die Reduktion die Entscheidung zwischen *IfThenStatement* und *IfThenElseStatement* bereits ohne Ansehen eines etwaigen else-Tokens getroffen wurde.

Der Autor sieht sich außerstande zu beurteilen, ob es sich hier um einen Fehler in [GJSB96], eine Kurzschreibweise, oder letztendlich doch nur um ein Missverständnis des Autors handelt. Das Problem jedenfalls konnte gelöst werden, indem die einrümpfigen Produktionen durch Produktionen ersetzt wurden, die für jede der Unterkategorien von *Statement* bzw. *StatementNoShortIf* einen Rumpf enthalten. Durch diese etwas längliche Unterteilung kann die Entscheidung bis zu einem etwaigen else-Token offen gelassen und dann anhand der üblichen *Shift/Reduce*-Strategie richtig getroffen werden. Listing 3.7 zeigt beispielhaft die Produktionen des *IfThenStatement* in der `yacc`-Spezifikation.

```
1  if_then_statement
2    : KEYWORD_IF LEFT_PAREN expression RIGHT_PAREN statement_without_trailing_substatement
3      { $$ = new yy.IfThen($3, $5, @$); }
4    | KEYWORD_IF LEFT_PAREN expression RIGHT_PAREN if_then_statement
5      { $$ = new yy.IfThen($3, $5, @$); }
6    | KEYWORD_IF LEFT_PAREN expression RIGHT_PAREN if_then_else_statement
7      { $$ = new yy.IfThen($3, $5, @$); }
8    | KEYWORD_IF LEFT_PAREN expression RIGHT_PAREN while_statement
9      { $$ = new yy.IfThen($3, $5, @$); }
10   | KEYWORD_IF LEFT_PAREN expression RIGHT_PAREN for_statement
11     { $$ = new yy.IfThen($3, $5, @$); }
12   ;
```

Listing 3.7: Lösung durch Umgehung zweier Nichtterminale

3.3 Programmstruktur

Seh'n Sie, Herr Doctor, manchmal hat man so 'nen Charakter, so 'ne Struktur ... Aber mit der Natur ist's was ander's, sehen Sie, mit der Natur, das ist so was, wie soll ich doch sagen – zum Beispiel –

—Woyzeck in *Woyzeck*

Die Präsentation von hobbes' Programmstruktur wird entlang des Ablaufs einer typischen Verwendung entwickelt. Als erstes erfolgt dabei der **Aufbau eines abstrakten Syntaxbaums** mit Hilfe des durch *Jison* generierten Parsers. Nach dem erfolgreichen Aufbau wird im **Kompilationsschritt** aus der Baumstruktur rekursiv JavaScript erzeugt. Zuletzt wird das Kompilat **in einer Umgebung mit eigenem Namensraum ausgeführt**. Abbildung 3.6 gibt einen Überblick über die Bestandteile von hobbes.

- `hobbes`
  - `compiler`
    - `ast_nodes.js` ..... Konstruktoren für Nichtterminal-Knoten
    - `parser.js` ..... Mit *Jison* erzeugter Parser
  - `extensions`
    - `es5.js` ..... ECMAScript-5-Erweiterungen für alte Umgebungen
    - `json.js` ..... *JSON*-Erweiterungen für alte Umgebungen
  - `stdlib`
    - `java`
    - `java.js` ..... Nachbauten wichtiger `java.lang`-Klassen
  - `utils`
    - `code_builder.js` ..... Hilfsfunktionen zur Code-Erzeugung
  - `vava`
    - `class.js` ..... *Vava*-Klassen
    - `entity.js` ..... Obertyp für Entities mit Datenkapselung
    - `method.js` ..... *Vava*-Methoden
    - `scope.js` ..... *Scope*-Implementation
    - `type.js` ..... Typensystem und primitive Typen
  - `compiler.js` ..... Compiler-Modul
  - `extensions.js` ..... Lädt Erweiterungen
  - `stdlib.js` ..... Standardibliothek-Modul
  - `utils.js` ..... Hilfsfunktions-Modul
  - `vava.js` ..... Laufzeitumgebungs-Modul
  - `web.js` ..... Browser-API-Modul
- `hobbes.js` ..... Haupt-Modul

Abbildung 3.6: Modulübersicht für hobbes

3.3.1 Parsing und Aufbau eines AST

In Abschnitt 3.2.2 waren bereits Beispiele der Aufrufe zu sehen, die, aus den Werten beim Parsing gefundener Token, Knoten vom Typ der zugehörigen syntaktischen Kategorie erzeugen. Die Konstruktorfunktionen dafür sind im Teilmodul `hobbes/compiler/ast_nodes` des Compiler-Moduls definiert und werden dem Parser vor dem Parsevorgang über dessen offen zugängliche Eigenschaft `yy` verfügbar gemacht.

Jeder Knoten des Baumes erbt mittels prototypischer Vererbung vom Grund-Knotentyp `ASTNode` und verfügt damit über Methoden, die der Erzeugung, Typprüfung und Kompilierung des Knotens und seiner Kindknoten dienen. Unter einem *Knotentyp* ist hier der Konstruktor eines Knotenobjektes zu verstehen. Jeder Knoten-Konstruktor trägt dabei unter der Eigenschaft `type` jedes neu erzeugten Objektes einen Typennamen wie *ASTNode* oder *ReturnStatement* ein.

Um die korrekte Funktionsweise der Vielzahl an Knoten kontrollieren zu können, wurde für jeden Knotentypen ein Unit-Test verfasst, der die vom Knoten-Konstruktor erzeugten Objekte auf die Einhaltung des `ASTNodeInterface` prüft und die Ausgabe der `compile`-Methode mit einem manuell erstellten Sollwert vergleicht. Ein Interface in diesem Sinn ist ein Objekt, das Methoden bereitstellt, um andere Objekte auf bestimmte Eigenschaften und deren Typen zu prüfen[13].

Als Beispiel zeigt Listing 3.8 den schon aus 3.2.2 bekannten `IfThen`-Konstruktor.

```
1  var IfThen = exports.IfThen = function (ifExpr, thenExpr) {
2    this.type = 'IfThen';
3    this.setLoc(arguments[arguments.length-1]);
4    this.children = [];
5    if (!ifExpr || !thenExpr) {
6      throw new TypeError('Expected condition and conditional.');
7    }
8    this.appendChild(ifExpr);
9    this.appendChild(thenExpr);
10 };
11
12 IfThen.inherits(ASTNode);
```

Listing 3.8: `IfThen`-Konstruktor

In beschriebener *bottom-up*-Manier wird so aus einem Java-Programmtext ein Baum aus JavaScript-Objekten erzeugt. Die Wurzel eines Syntaxbaumes ist in allen Fällen vom Typ `CompilationUnit` und wird als Ergebnis des Parsevorgangs zurück gegeben. Da jedes Knotenobjekt eine `toString`-Methode besitzt, die eine String-Darstellung des durch den Knoten gegebenen Teilbaumes erzeugen kann, ist im

[13] Diese und viele weitere Methoden zur strukturierten Entwicklung mit JavaScript werden in [HD07] beschrieben und können hier nicht im Einzelnen diskutiert werden.

Fall eines nach dem Parsen auftretenden Fehlers eine übersichtliche Darstellung des Syntaxbaums verfügbar. Listing 3.9 enthält den Programmtext, dessen durch hobbes erzeugte Syntaxbaum-Darstellung in Abbildung 3.7 gezeigt ist.

```
1 public class HalloWelt {
2   // Kommentare verschwinden
3   public static void main(String[] args) {
4     if (1 != 0) { System.out.println("Hallo Welt!"); }
5   }
6 }
```

Listing 3.9: HalloWelt-Programmtext

Abbildung 3.7: Ausgabe der `toString`-Methode des `CompilationUnit`-Knotens

3.3.2 Kompilation

Nach Anstoßen des Parsevorgangs übergibt der Compiler die Kontrolle an das Parser-Modul und wartet auf die Rückgabe des Syntaxbaumes. Tritt während des Parsens ein Fehler auf, wird das geworfene Fehler-Objekt im Compiler-Modul mit Informationen zur Fehlerquelle im Programmtext angereichert und anschließend neu geworfen. Es bleibt dem Nutzer des Moduls überlassen, wie auf den Fehler zu reagieren ist.

Bei erfolgter Rückgabe dagegen wird die `compile`-Methode des Wurzelknotens mit einem Options-Objekt als Argument aufgerufen, welches während des Kompilierens dem vertikalen Informationsaustausch innerhalb des Syntaxbaums dient. Um die Funktionsweise dieser Kommunikationseinheit verständlich zu machen, ist es zunächst notwendig, das Modul `hobbes/vava/scope` zu erklären.

SCOPE-CHAINING

Die durch den Konstruktor `Scope` erzeugten Objekte bieten unter Verwendung von JavaScripts *prototype chain* die Möglichkeit, hierarchische lexikalische Namensräume zu realisieren. Dazu besitzt jedes Scope-Objekt die Methode `__descend`, die wie in Listing 3.10 implementiert ist. Sie erzeugt eine neue Konstruktor-Funktion, deren `prototype`-Eigenschaft sie auf das Objekt, dessen Methode sie ist setzt. Das mit diesem Konstruktor erzeugte Objekt (`newScope`) hat dann das bestehende Scope-Objekt zum Prototypen. Zuletzt werden dem neuen Scope-Objekt noch optional übergebene Namen hinzugefügt.

```
1 Scope.prototype.__descend = function (namesValues) {
2   var Scoper = function () {};
3   Scoper.prototype = this;
4   var newScope = new Scoper();
5   return newScope.__add(namesValues);
6 };
```

Listing 3.10: `__descend`-Methode

Da ein damit neu erzeugtes Scope-Objekt alle Eigenschaften des ihn erzeugenden Scope-Objekts besitzt, an ihm vorgenommene Änderungen sich aber nicht auf seinen Erzeuger auswirken, ist so eine Lösung für lexikalische Namensräume geschaffen, wie sie in Javas Methoden und Blöcken verwendet werden. Das rechte Objekt in Abbildung 3.8 beispielsweise erbt die Eigenschaft `b` seines Erzeugers, überdeckt aber `a` und fügt `c` neu hinzu[14]. Zur Verwendung im weiteren Text soll der Terminus der *Vertikale* eines Scope-Objekts eingeführt werden, womit die Kette der Scope-Objekte gemeint ist, von denen das Scope-Objekt Eigenschaften erbt.

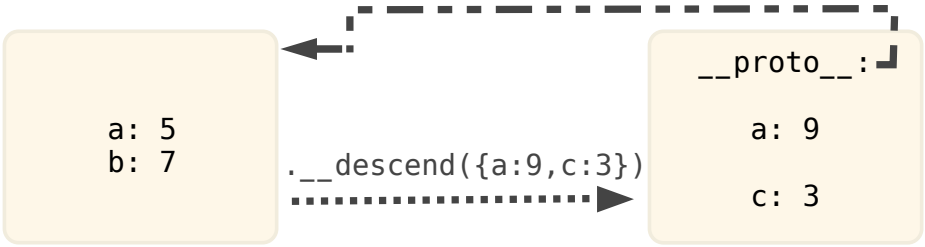

Abbildung 3.8: Scope-Vererbung

Scope-Objekte finden in hobbes an zwei Stellen Verwendung. Zum einen dienen sie zur Umsetzung von lexikalischen Namensräumen in der Ausführung der kompilierten Programme. Zum anderen dienen sie während des Kompilationsschrittes zur Buchführung über die mit Namen und Methodensignaturen verbundenen Java-Typen. Dazu nun weiter im Kompilationsprozess.

[14] Die `__proto__`-Eigenschaft des Erzeuger-Objekts ist dabei aus Gründen der Übersicht nicht eingezeichnet.

Rekursive Kompilierung

Das der `compile`-Methode der Wurzel übergebene Options-Objekt enthält zunächst nur die Eigenschaft `name`, die ein Scope-Objekt referenziert. Dieses enthält in seiner Vertikale mindestens die aus dem Modul `stdlib` geladenen Objekte des `java.lang`-Nachbaus. Sind bereits weitere Klassen geladen und kompiliert worden, wie es in der aktuellen Konfiguration von hobbes automatisch mit einem Teil des Java-Pakets `AlgoTools` geschieht[15], sind diese ebenfalls in der Vertikale des Scope-Objekts verfügbar.

Von der Wurzel absteigend wird nun unter Weitergabe des Options-Objekts der Befehl zur Kompilierung bis zu den Blättern des Syntaxbaumes propagiert. Aus dem Kompilat seiner Kindknoten erzeugt dabei jeder Knoten den seinem Typ entsprechenden JavaScript-Code. Als Beispiel für diesen Vorgang soll wieder der `IfThen`-Knotentyp dienen, dessen `compileNode`-Methode[16] in Listing 3.11 zu sehen ist.

```
1 IfThen.prototype.compileNode = function (opts) {
2   var blockOpts = opts.descendScope({indent: (opts.indent || 0) + 2});
3   var js = 'if (this.__env.BooleanValue.intern(true) === ';
4   js += this.children[0].compile(opts) + ') {\n';
5   js += this.children[1].compile(blockOpts);
6   return utils.indent(js + '\n}\n', opts.indent);
7 };
```

Listing 3.11: `compileNode`-Methode des `IfThen`-Tokentyps

Bei Knotentypen, deren Wert einen Java-Typ besitzt, wird während oder nach der Kompilierung der Java-Typ des Knotens aus den Werten seiner Kindknoten bestimmt, deren Java-Typ zu diesem Zeitpunkt stets bekannt ist. Die genauen Routinen zur Typ-Bestimmung sind dabei von Knotentyp zu Knotentyp sehr unterschiedlich. Als Beispiel soll lediglich die `compileTimeCheck`-Methode des Obertyps `BinaryOperatorNode` dienen, die in vielen Knotentypen für binäre Operatoren zum Einsatz kommt.

```
1 BinaryOperatorNode.prototype.compileTimeCheck = function (opts) {
2   if (!this.isApplicable())
3     opts.addError(
4       this.nonFatalError(
5         'operator ' + this.operator + ' cannot be applied to ' +
6         this.children[0].getVavaType() + ',' + this.children[1].getVavaType()
7       )
8     );
9   else
```

[15] Die `AlgoTools` werden in der Lehrveranstaltung *Informatik A* der Universität Osnabrück als Hilfsmittel unter anderem zur einfacheren Ein- und Ausgabe verwendet.

[16] `compileNode` wird neben anderen Methoden eines Knotens von `compile` aufgerufen

```
10      this.vavaType = this.constructor.table[this.children[0].getVavaType()][this.children[1].getVavaType
            ()];
11  };
```

Listing 3.12: `compileTimeCheck`-Methode des `BinaryOperatorNode`-Obertyps

Widerspricht der Wert einer der Kindknoten den semantischen Bedingungen eines Konstruktes, wird, wie ebenfalls in Listing 3.12 erkenntlich, ein Fehler im Options-Objekt vermerkt. Die Kompilierung bricht aber meist nicht sofort ab, sondern wird fortgesetzt, um potentiell weitere Fehler zu protokollieren. Im Fehlerfall werden nach dem Ende des Kompilierens die aufgetretenen Fehler gesammelt geworfen. Abbildung 3.9 zeigt die Ausgabe bei der versuchten Ausführung von Listing 3.13 mit hobbes.

```
1  public class Fehlerhaft {
2    public static void main(String[] args) {
3      int a = 5;
4      if (a || false) {
5        System.out.println(2 * true == false);
6      }
7    }
8  }
```

Listing 3.13: Fehlerhafter Java-Code

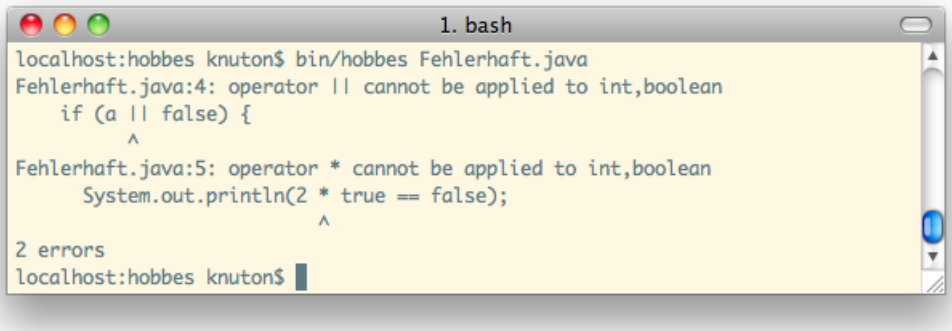

Abbildung 3.9: Fehlermeldung bei fehlerhaftem Java-Code

Verläuft die Kompilierung ohne Fehler, terminiert die `compile`-Methode des Wurzel-Knotens und gibt das fertige JavaScript-Kompilat zurück. Als Beispiel eines solchen Ergebnisses dient das bereits bekannte `HalloWelt`-Programm aus Listing 3.9. Sein Kompilat ist in Listing 3.14 zu sehen.

```
1  this.__add({
2    HalloWelt: new this.__env.VavaClass(
3      "HalloWelt",   // Klassen-Name
4      { fields: [],  // Klassen-Variablen
5        methods: [   // Klassen-Methoden
6          // Der Konstruktor erhaelt Methodenname, Rueckgabetyp, formale Parameter und Funktionskoerper
7          new this.__env.VavaMethod("main", "void", [{identifier:"args",vavaType:"String[]"}], function ()
               {
```

```
8           if (this.__env.BooleanValue.intern(true) === this.__env.BooleanValue.intern(this.__env.
                IntValue.intern(1).to("int") !== this.__env.IntValue.intern(0).to("int"))) {
9             this.System.out.send("println(String)", [new this.__env.StringValue("Hallo Welt!")]);
10          }
11        })
12      ]
13    },
14    this
15  )
16 });
17 return this["HalloWelt"];
```

Listing 3.14: Kompilat des `HalloWelt`-Programms[17]

Zur Verwandlung eines solchen Quellcode-Strings in ausführbaren Code, wird der String dem in JavaScript eingebauten `Function`-Konstruktor übergeben, der eine Funktion erzeugt, die den String zum Funktionskörper hat. Zuletzt wird die jeder Funktion eigenen `call`-Methode der Funktion aufgerufen, die es erlaubt, die Funktion als Methode eines der `call`-Methode übergebenen Objekts auszuführen. Hierbei wird ein Scope-Objekt übergeben, das bereits geladene Pakete und Klassen, sowie Konstruktoren und Funktionen der Laufzeitumgebung unter dem Präfix `__env` enthält.

Durch die Art des Funktionsaufrufs steht das Scope-Objekt im Inneren der Funktion unter dem `this`-Stichwort zur Verfügung. Nun werden mittels der im Modul `hobbes/vava` definierten Konstruktoren Objekte erzeugt, die Java-Entitäten verörpern und dabei nach jeweiligem Bedarf einen lokalen Namensraum aus dem Scope-Objekt erzeugen. Das dem Scope-Objekt hinzugefügte `VavaClass`-Objekt wird als Funktionswert zurückgegeben.

3.3.3 Ausführung

Das nun zur wiederholten Ausführung zur Verfügung stehende Programm behält auf den verschiedenen Hierarchiestufen einen jeweils eigenen Namensraum und kann durch Aufruf der `main`-Methode der Klasse ausgeführt werden. Besitzt ein zur Ausführung vorgesehenes `VavaClass`-Objekt keine entsprechende Methode, wird die Ausführung mit Hinweis auf die fehlende Methode abgebrochen.

Sämtliche in der Ausführung miteinander interagierenden Komponenten sind JavaScript-Objekte, die Klassen, Methoden, Variablen oder Werte repräsentieren und für ihren Typ spezifische Methoden besitzen. Dieser stark durch die Sprache *Ruby* inspirierte Ansatz kann hier nicht in vollem Umfang diskutiert werden. Für ein grundsätzliches Verständnis der Voränge sollte die Betrachtung des Kompilats

[17] zur besseren Lesbarkeit umformatiert

durch die sprechenden Methodennamen der betreffenden Objekte ausreichen. Für ein tiefgreifenderes Verständnis der Implementation sei auf selbige verwiesen.

Die über `java.lang.System` verfügbaren Ein- und Ausgabe-Schnittstellen, die beim ersten Ausführen des Kompilats in das Scope-Objekt eingefügt wurden, werden je nach Umgebung der Ausführung passend gewählt. Dementsprechend führt `bin/hobbes HalloWelt.java` zum gewünschten Ergebnis.

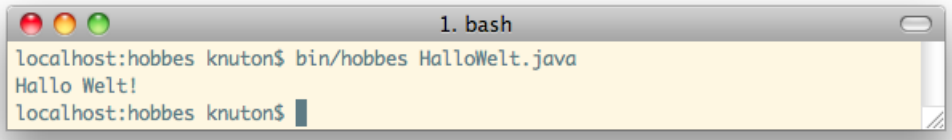

Abbildung 3.10: Ausgabe von `HalloWelt.java`

3.3.4 Verwendung in Websites

Zum Abschluss des Kapitels soll noch eine beispielhafte Verwendung von hobbes im Browser erklärt werden. Die im globalen Namensraum des Browsers unter `hobbes.web` verfügbare API ist momentan noch sehr schlank und bietet lediglich zwei Methoden, `setup` und `execute`, die unterschiedlich feine Konfigurationsmöglichkeiten offen lassen.

`hobbes.web.setup`

Bei der Anwendung der `setup`-Methode braucht nichts weiter getan zu werden, als die den ausführbaren Quellcode enthaltenden HTML-Elemente mit der HTML-Klasse `hobbesecutable` zu kennzeichnen, und nach dem Laden des Dokuments `hobbes.web.setup(outputElem)` auszuführen. `outputElem` ist dabei das Element oder die HTML-ID des Elements, das als Ausgabefeld ausgeführter Programme dienen soll. Die gekennzeichneten Quellcode-Träger werden dann automatisch erkannt und mit einem Button zur Ausführung versehen. Für diesen Einsatz ist Listing 3.15 ein Minimalbeispiel.

```
1  <!DOCTYPE html>
2  <html>
3    <head>
4      <title>hobbes &ndash; ein Java-Interpreter in JavaScript</title>
5      <script type="text/javascript" src="hobbes-web.min.js"></script>
6      <script type="text/javascript">
7        window.onload = function () { hobbes.web.setup('term'); };
8      </script>
9    </head>
10   <body>
```

```
11      <pre class="hobbesecutable">
12  public class HalloWelt {
13    public static void main(String[] args) {
14      if (1 != 0) { System.out.println("Hallo Welt!"); }
15    }
16  }
17      </pre>
18      <pre id="term"></pre>
19    </body>
20  </html>
```

Listing 3.15: Verwendung der `setup`-Methode

`hobbes.web.execute`

Die zuvor beschriebene Methode ist bequem und unkompliziert, lässt aber wenig Freiheit und ist daher nur für sehr einfache Einsätze geeignet. Durch die direkte Verwendung der `execute`-Methode kann der Anwender Quellcode-Träger und Ausgabefeld direkt übergeben – und so zum Beispiel jedem Programm ein eigenes Ausgabefeld zuweisen. Das Kompilieren und Ausführen wird von dieser Methode direkt angestoßen, der Anwender muss sich also selber um Kontrollelemente kümmern.

Kapitel 4

Rückblick und Ausblick

Nachdem nun Ziele, Konzept und Umsetzung vorgestellt wurden, soll in einer abschließenden Reflexion einerseits der Erfolg der Entwicklung evaluiert werden. Mit Blick auf die erkannten Stärken und Schwächen der Software soll dann auf mögliche Varianten und Alternativen eingegangen werden. Die Arbeit schließt mit einigen Gedanken zum Einsatz in der Didaktik.

4.1 Evaluation

I wanted everything, but I think that I only got most of it.
—Kurt Vile, *I Wanted Everything*

Das Hauptziel bei der Konzeption von hobbes war die Erschaffung einer vielseitig einsetzbaren und einfachen Lösung zur Ausführung von Beispiel-Programmen im Browser-Kontext, die eine möglichst große äußerliche Nähe zum Verhalten einer *echten* Java-Umgebung aufweist. Um die dazu nötige Flexibilität zu erreichen, wurde besonderer Wert auf einen modularen Aufbau mit wenig Minimalanforderungen gelegt.

Stärken

Wie in Abschnitt 3.3.4 beschrieben, ist es gelungen, eine leicht konfigurierbare Komponente für den Einsatz im Browser zu entwickeln. In durchgeführten Tests erfüllte hobbes seine Aufgabe zuletzt in den Browsern *Chrome* (mindestens ab Version 10), *Firefox* (mindestens ab Version 3.6) und *Internet Explorer* (ab Version 9). Die für Browser gepackte Version ist 349 Kilobyte groß, durch *minifying* mit Goo-

gle's *Closure Compiler*[1] kann die Dateigröße auf 223 Kilobyte verringert werden. Bei diesem Vorgang werden Kommentare entfernt, Variablen umbenannt und automatische Syntaxumformungen vorgenommen, um ein äquivalentes Programm mit weniger Zeichen zu generieren. Dieses Ergebnis ist durchaus für den Einsatz in Websites geeignet. Zum Vergleich hat die in vielen Websites eingebundene JavaScript-Bibliothek *jQuery*, die der Interaktion mit dem Dokumentenbaum eines Webdokuments dient, selbst bereits eine Größe von 90 Kilobyte[2].

Im Zusammenspiel mit einer Vielzahl an frei verfügbaren, in JavaScript geschriebenen, Quelltext-Editoren, ist die Einrichtung einer benutzerfreundlichen Oberfläche zur Bearbeitung und Ausführung von Java-Programmen im Browser durch das Verbinden der Komponenten mit geringem zusätzlichem Aufwand verbunden.

Die Ausgabe der Kompilierung und Ausführung von Java-Programmen mit hobbes entspricht in weiten Teilen glaubhaft der der offiziellen Umgebung, wie sich für die durch die Spezifikations-Tests abgedeckten Sprachmerkmale automatisch überprüfen lässt.

Durch hobbes' modularen Aufbau ist die Adaption für neue Umgebungen relativ unkompliziert und erfordert nicht viel mehr als die Bereitstellung einer passenden Implementation von `java.lang.System` und einer Routine zur Behandlung der potentiell von Parser oder Compiler geworfenen Fehler.

Schwächen

Die zuvor beschriebene Unterstützung der populären Browser ist erfreulich, einige weniger weit verbreitete Browsertypen werden aber ebenso wie einige ältere Browser-Versionen derzeit nicht unterstützt. Zu nennen wären hier *Safari*, *Opera* und ältere Versionen des *Internet Explorer*, die teilweise noch einen nicht unerheblichen Anteil der Installationen ausmachen. Die fehlende Unterstützung *Safari*s und *Opera*s ist aufgrund deren weiter Verbreitung auf mobilen Geräten allerdings ein klarer Schwachpunkt und wäre beim tatsächlichen Produktiveinsatz von hobbes dringend zu beheben.

Eine unangenehme Einschränkung der gewählten Implementationsweise ist die Abhängigkeit von durch die jeweiligen JavaScript-Umgebungen angebotenen Eingabeschnittstellen. ECMAScript definiert keinerlei Methoden zur Abfrage von Nutzerangaben und die Ausführung eines Programms findet rein seriell statt. Es ist daher nur auf eine Art möglich, auf Nutzereingaben zu warten, nämlich durch

[1] `http://code.google.com/closure/compiler/`

[2] `http://code.jquery.com/jquery-1.6.1.min.js`, gemessen am 30.05.2011

Warten auf die Rückgabe einer entsprechenden Eingabefunktion. Definiert eine Umgebung keine synchronen Eingabeschnittstellen, ist hobbes zwar lauffähig, kann aber nur nicht-interaktive Programme ausführen. Diese Einschränkung hat auch zur Folge, dass die Eingabe im Browser mittels der eingebauten Funktion `prompt` erfolgen muss, welche dem Nutzer einen Eingabedialog anzeigt und so aus der Nutzungsmetapher des simulierten Terminals fällt.

Optimierungen & Aufgaben

Die bei der Entwicklung von hobbes eingesetzte Testumgebung war eine wertvolle Hilfe, die aber noch weiter verbessert werden könnte. Gerade im Hinblick auf die fehlende Unterstützung einiger Browser, wäre es wünschenswert, dass die automatisierten Tests nicht nur in einer, sondern in mehreren JavaScript-Umgebungen ausgeführt würden. Einige in Browsern verwendete Umgebungen, wie etwa die in *Firefox* eingesetzte *SpiderMonkey-Engine*[3], sind ebenfalls außerhalb des Browsers verfügbar und sollten zur durchgehenden Kontrolle genutzt werden.

Die Typ-Prüfungen bei der Kompilierung entsprechen noch bei weitem nicht der feinen Analyse des offiziellen Compilers. An einigen Stellen, wie etwa bei der Ermittlung des Wahrheitswertes einfacher boolescher Ausdrücke oder der Unterscheidung zwischen problematischen und unproblematischen Zahl-Typ-Umwandlungen, könnten die Analyseschritte mit wenig konzeptueller und einiger manueller Arbeit nachgefügt werden.

Zuletzt wäre jede Erweiterung des unterstützten Sprachumfangs begrüßenswert, allem voran die Unterstützung von Javas Feldern (*arrays*), die eine wichtige Rolle bei der Vermittlung von Datenstrukturen spielen.

4.2 Alternativen

Angesichts der trotz des großen Entwicklungs-Aufwandes vorhandenen Einschränkungen stellt sich erneut die Frage nach alternativen Lösungen. Ohne dabei auf Java zu beharren, werden allgemein Ansätze für ähnliche Problemstellungen beschrieben, die tatsächlich gewählt wurden oder vorstellbar wären. Dabei werden zwei Kategorien konkurrierender Ansätze unterschieden. Eine attraktive dritte Möglichkeit besteht in der Automatisierung des in dieser Arbeit verwendeten Ansatzes.

[3] `https://developer.mozilla.org/en/SpiderMonkey`

AJAX-KOMMUNIKATION MIT SERVER

Eine weit weniger aufwändige und zugleich erfolgreich eingesetzte Variante ist die Verwendung JavaScripts zur Implementation eines Terminals im Browser, das der Kommunikation mit einer andernorts eingerichteten Laufzeitumgebung dient. Beispiele für dieses Modell sind ein *Ruby-REPL*[4] unter `tryruby.org` sowie ein System zur Ausführung von *Go*-Programmen[5]. Letzteres erlaubt dabei nur das Verfassen und Ausführen von Programmen, aber keine Interaktion während der Ausführung. Bei beiden Systemen werden die Nutzereingaben an ein Serversystem geschickt, dort ausgeführt und die entstehende Ausgabe zurück an den Browser des Nutzers übermittelt. Solche Lösungen verursachen wenig Entwicklungsaufwand in JavaScript, machen dafür aber eine ständig verfügbare Server-Infrastruktur notwendig, auf deren Sicherung und Skalierung geachtet werden muss. Ein weiterer Nachteil ist die Angewiesenheit des Nutzers auf eine verfügbare Internetverbindung zur Arbeit mit dem System.

VIRTUELLE MASCHINE

Der wohl größte Vorteil bei der Nachbildung einer virtuellen Maschine wie der *Java Virtual Machine* wäre die dadurch eröffnete Bandbreite an Programmiersprachen. Da für mehr und mehr Sprachen wie beispielsweise *Scala*[6], *Ruby*[7] und *Python*[8] Bytecode-Compiler existieren, würde durch eine JavaScript-*JVM* die Grundlage für gleich mehrere Einsatzfelder geschaffen. Durch die Möglichkeit, in eine virtuelle Maschine Möglichkeiten zur Unterbrechung und Fortsetzung der Ausführung zu integrieren, würde hier eine attraktive Lösung der in Abschnitt 4.1 beschriebenen Eingabeproblematik geboten, indem die Fortsetzungsroutine als Event-Verarbeitung nach erfolgter Nutzereingabe registriert und die Ausführung zwischenzeitlich unterbrochen werden könnte.

AUTOMATISIERUNG

Das Open-Source-Projekt *emscripten*[9] ist in der Lage, bereits bestehende Interpretersysteme, deren Quellen zum Bytecode der *LLVM*-Compilerkomponente[10]

[4] *Read-Eval-Print Loop*, eine interaktive Programmierumgebung [Wik11]

[5] *Go* ist eine Systemprogrammiersprache, `http://golang.org/`, das genannte System ist unter `http://golang.org/doc/playground.html` erreichbar.

[6] `http://www.scala-lang.org/`

[7] `http://www.jruby.org/`

[8] `http://www.jython.org/`

[9] `http://github.com/kripken/emscripten`

[10] `http://llvm.org/`

kompiliert werden können, in JavaScript zu überführen. Auf diese Weise wurden bereits Interpreter für *Python*[11] und *Lua*[12] erstellt. Mit einer Dateigröße von weit mehr als 2 Megabyte[13] ist das resultierende JavaScript-Kompilat des *Python*-Interpreters allerdings in einem sehr grenzwertigen Bereich.

4.3 Chancen und Gefahren des Lehreinsatzes

Operations of thought are like cavalry charges in a battle—they are strictly limited in number, they require fresh horses and must only be made at decisive moments.
—Alfred North Whitehead, *An Introduction to Mathematics*

Gerade beim Erlernen völlig neuer Wissensgebiete, kann dem Lernenden geholfen werden, indem die Freiheit geschaffen wird, sich auf den eigentlichen Lernstoff zu konzentrieren. Gegenüber der selbständigen Einrichtung einer ersten Programmierumgebung hat die Bereitstellung einer nutzerfreundlichen vorkonfigurierten Lösung den Vorteil, dass der Bereich der Probleme, die vom Lernenden zu bewältigen sind, auf den Bereich der intendierten Lerninhalte eingegrenzt werden kann.

Einführungsveranstaltungen der Informatik werden an deutschen Hochschulen nicht nur von Studenten der Informatik, sondern auch von Studenten anderer Fachbereiche besucht. Dadurch ist nicht davon auszugehen, dass der durchschnittliche Hörer mit dem Konzept eines Terminals vertraut ist. Der Browser hingegen gehört zur täglichen Erfahrungswelt heutiger Computer-Nutzer. Die Gefahr einer Überforderung unvorbereiteter Studenten, die das geistige Einlassen auf die eigentlichen Inhalte verhinden könnte, sollte möglichst gering gehalten werden. Durch die Verlagerung in eine vertraute Umgebung, könnten dem Anfänger Aufgaben abgenommen werden, die für ein Grundverständnis von algorithmischer Datenverarbeitung nicht notwendig sind und so die Anzahl gleichzeitig zu verarbeitender neuer Erfahrungen reduziert werden.

Andererseits könnte man konsistent argumentieren, dass zum Umfang der Lehrinhalte einer einführenden Informatikveranstaltung eben auch die Konfrontation mit den Details von Computersystemen gehört. Das Einrichten einer anfängerfreundlichen Umgebung sollte nicht den Effekt haben, die den Studenten insgesamt nähergebrachte Computerkompetenz zu verringern. Die Hemmungen im Umgang mit Computersystemen müssen für die erfolgreiche Arbeit mit

[11] `http://syntensity.com/static/python.html`
[12] `http://syntensity.com/static/lua.html`
[13] `http://syntensity.com/static/python.js`, gemessen am 31.05.2011

Computern abgebaut werden. Während also wünschenswert wäre, Studenten anderer Fachbereiche das Verständnis algorithmischer Methoden auf einem abstrakten Niveau zu vermitteln, ohne sie dabei unnötig mit dem Gebrauch von Programmierer-Werkzeug zu belasten, wäre es verhängnisvoll, dadurch die Entwicklung der Computerkompetenz der Hauptfachstudenten zu verringern.

Durch die zunehmend wichtige Rolle des Browsers und der wachsenden Verfügbarkeit von Lösungen, die der in dieser Arbeit vorgestellten ähneln, werden die hier gestellten Fragen von immer grßerer Relevanz sein. Es bleibt dem Lehrpersonal überlassen, den für die jeweiligen Anforderungen angebrachten Grad an technischen Details zu finden.

LITERATURVERZEICHNIS

[ALSU08] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, *Compiler. Prinzipien, Techniken und Werkzeuge*, 2 ed., Pearson Studium, München, 2008.

[BSS10] Randal E. Bryant, Klaus Sutner, and Mark J. Stehlik, *Introductory Computer Science Education at Carnegie Mellon University: A Deans' Perspective*, Tech. Report CMU-CS-10-140, Carnegie Mellon University, Pittsburgh, 2010, http://link.cs.cmu.edu/article.php?a=524.

[Cha01] Steve Champeon, *JavaScript: How Did We Get Here?*, 2001, http://oreilly.com/pub/a/javascript/2001/04/06/js_history.html, abgerufen am 20.05.2011.

[Cro08] Douglas Crockford, *JavaScript: The Good Parts*, O'Reilly, Sebastopol, 2008.

[ECM99] ECMA International, *Standard ECMA-262*, Tech. report, 1999, http://www.ecma-international.org/publications/standards/Ecma-262-arch.htm.

[ECM07] ———, *Proposed ECMAScript 4 Edition*, Tech. report, 2007, http://www.ecma-international.org/activities/Languages/Language\%20overview.pdf.

[ECM09] ———, *Standard ECMA-262, 5. Ed.*, Tech. report, 2009, http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[Eic08] Brendan Eich, *ECMAScript Harmony*, https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html, besucht am 20.05.2011, 2008.

[Fla08] David Flanagan, *JavaScript - the definitive guide*, O'Reilly, Sebastopol, 2008.

[Fre11] Free Software Foundation, *Bison Manual*, 2011, http://www.gnu.org/software/bison/manual/.

[GJSB96] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha, *The Java Language Specification*, 1 ed., Sun Microsystems, Mountain View, 1996.

[GJSB05] ———, *The Java Language Specification*, 3 ed., Addison-Wesley, Upper Saddle River, 2005.

[HD07] Ross Harmes and Dustin Diaz, *Pro JavaScript Design Patterns*, Apress, New York, 2007.

[Job10] Steve Jobs, *Thoughts On Flash*, 2010, http://www.apple.com/hotnews/thoughts-on-flash/, besucht am 20.05.2011.

[McC11] Harry McCracken, *The Post-PC Era Is Already Here*, 2011, http://www.time.com/time/business/article/0,8599,2058101,00.html, besucht am 20.05.2011.

[Sch07] Holger Schwichtenberg, *Programmiersprachen in Lehre und Praxis*, 2007, http://www.heise.de/ix/artikel/Bestandsaufnahme-506760.html, abgerufen am 20.05.2011.

[Sei09] Peter Seibel, *Coders at Work*, Apress, New York, 2009.

[Str94] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, Amsterdam, 1994.

[Wal10] Jim Waldo, *Java: The Good Parts*, O'Reilly Media, Sebastopol, 2010.

[Wik10] Wikipedia, *Write once, run anywhere — Wikipedia, The Free Encyclopedia*, 2010, http://en.wikipedia.org/w/index.php?title=Write_once,_run_anywhere&oldid=366363860, abgerufen am 20.05.2011.

[Wik11] ———, *Read-eval-print loop — Wikipedia, The Free Encyclopedia*, 2011, http://en.wikipedia.org/w/index.php?title=Read-eval-print_loop&oldid=431801072, besucht am 01.06.2011.

Erklärung

Hiermit versichere ich nach bestem Wissen und Gewissen, die eingereichte Bachelorthesis selbständig und unter Verwendung nur der angegeben Quellen und Hilfsmittel verfasst zu haben. Alle Stellen, die sinngemäß oder wörtlich der Arbeit anderer Autoren entnommen wurden, habe ich kenntlich gemacht. Kein Teil dieser Arbeit wurde bereits zur Erlangung eines Grades dieser oder einer anderen Universität eingereicht.

Berlin, 31. Mai 2011