Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Thomas Weiß

Entwurf und Implementierung einer Skriptsprache für die dynamische Verarbeitung von SQL

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Thomas Weiß

Entwurf und Implementierung einer Skriptsprache für die dynamische Verarbeitung von SQL

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. sc. pol. Wolfgang Gerken

Abgegeben am 19. August 2009

Thomas Weiß

Thema der Bachelorarbeit

Entwurf und Implementierung einer Skriptsprache für die dynamische Verarbeitung von SQL

Stichworte

SQL, Domänenspezifische Skriptsprache, Continuation-Passing-Style, Interpreter, Objektorientierte Programmierung, Funktionale Programmierung

Kurzzusammenfassung

SQL als Abfragesprache für relationale Datenbanksysteme sowie die existierenden Werkzeuge für den Umgang mit diesen, erfüllen die sich durch einige Anwendungsszenarien ergebenden Anforderungen nur teilweise. Dies trifft insbesondere auf Anwendungsszenarien zu, welche durch hohe Dynamik und Komplexität gekennzeichnet sind. In dieser Arbeit wird eine domänenspezifische Skriptsprache entwickelt, welche diese Szenarien explizit adressiert. Konzipiert wird diese als berechnungsvollständige, objektorientierte Sprache, um komplexe Verarbeitungen zu ermöglichen. Dynamische Prozesse sollen durch die Konzeption als interpretierte, dynamisch typisierte Skriptsprache unterstützt werden. Durch die Konzeption der Sprache als Obermenge von SQL wird neben der interaktiven Benutzung auch die Abwärtskompatibilität zu bestehenden SQL-Skripts ermöglicht. Neben der Sprache und deren Grammatik wird ein Interpreter für diese entworfen. Der Ansatz des Continuation-Passing-Style wird hierfür verfolgt. Betrachtet wird zudem eine prototypische Implementierung des Interpreters.

Thomas Weiß

Title of the paper

Design and Implementation of a Scripting Language for the Dynamic Processing of SQL

Keywords

SQL, Domain-specific Scripting Language, Continuation-Passing-Style, Interpreter, Object-oriented Programming, Functional Programming

Abstract

SQL as query language for relational database systems and existing tools for their use can only partially satisfy the requirements imposed on them by some usage scenarios. This is particularly the case for usage scenarios which are characterized by high dynamics and complexity. In this thesis a domain-specific scripting language will be developed to address such scenarios explicitly. This language will be designed as a turing complete, object oriented language to support complex processings. Dynamic processes are to be supported by designing the language as interpreted, dynamically typed scripting language. By the language's design as a superset of SQL, its interactive use as well as the backwards compatibility to existing SQL-scripts will be facilitated. In addition to the language and its grammar, a corresponding interpreter will be designed that follows the Continuation-Passing-Style. Furthermore, a prototypical implementation of this interpreter is observed.

„Es ist nicht von Bedeutung,
wie langsam du gehst,
solange du nicht stehen bleibst.“

Konfuzius

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

1 Einleitung

In der vorliegenden Bachelorarbeit wird die Entwicklung einer Programmiersprache betrachtet, deren besonderer Fokus auf der Interaktion mit Datenbanksystemen liegt. Die Schwerpunkte dabei sind der Entwurf der Sprache und einer geeigneten Grammatik welche die unterschiedlichen Besonderheiten verschiedener Datenbanksysteme berücksichtigt sowie der Entwurf einer Maschine, mit welcher Programme der Sprache ausgeführt werden können.

1.1 Motivation

Für die Interaktion mit Datenbanken bzw. Datenbanksystemen bestehen diverse Möglichkeiten, angefangen von speziellen Clientprogrammen über proprietäre Programmierschnittstellen für Programmiersprachen über relativ datenbankunabhängige Low-Level-Abstraktionsschichten bis hin zu vollwertigen Objekt-Relational-Mappern in unterschiedlichen Ausführungen. Alle diese Möglichkeiten haben jedoch Grenzen, welche den Einsatz für bestimmte Anwendungsgebiete unpraktisch erscheinen lassen.

So werden im Data-Mining-Umfeld häufig in Datenbanken abgelegte Daten turnusmäßig aufbereitet, verdichtet, validiert und transferiert, wobei diese Daten sich oft strukturell ändern können. Verarbeitungsroutinen die einen direkten Zugriff auf die zugrunde liegenden Datenbanken zulassen und zugleich flexibel, umfangreich sowie dynamisch austauschbar sind, können hier wertvolle Hilfen sein.

Ebenso ergibt sich bei der Entwicklung bzw. dem Prototyping von Datenbankanwendungen häufig die Notwendigkeit, interaktiv mit den jeweiligen Datenbanken zu arbeiten, Strukturen zu testen, zu ändern, Daten in neue Formate zu migrieren und zu konvertieren. Letztere Aufgaben stellen sich oft auch bei der Wartung von Anwendungen, insbesondere bei Legacy-Anwendungen.

Existierende Werkzeuge decken die sich aus diesen Einsatzszenarien ergebenden Anforderungen jeweils nur in Teilen ab. Die Ergänzung des Spektrums um ein Werkzeug, welches sich diesen Anforderungen umfassend annimmt, kann einen deutlichen Mehrwert bei der Arbeit mit Datenbanken darstellen. Die Entwicklung eines solchen Werkzeugs soll daher Gegenstand dieser Arbeit sein.

1.2 Übersicht

In Kapitel 2 werden die bereits angedeuteten Anforderungen konkretisiert und sich daraus ergebende Konsequenzen für die zu entwickelnde Sprache und Maschine analysiert.

Lösungsansätze für die Umsetzung der gegebenen Anforderungen an die Sprache werden in Kapitel 3 recherchiert. Ebenso werden in diesem Kapitel verschiedene Techniken der Implementierung von Interpretern untersucht. Die untersuchten Techniken werden hinsichtlich ihrer Eignung zur Lösung der vorliegenden Problemstellung bewertet und eine geeignete Technik ausgewählt.

Ein Konzept für die Realisierung der Sprache sowie des Interpreters wird in den Kapiteln 4 bzw. 5 erarbeitet. Der Fokus liegt dabei auf grundlegenden Designentscheidungen beim Entwurf der Grammatik und des Parsers der Sprache sowie der Maschine, welche Programme der Sprache ausführen soll, der Integration von Sprache und Maschine und schließlich dem Zusammenspiel von Interpreter und Hostsystem.

Kapitel 6 beschreibt die Realisierung des entwickelten Konzeptes.

Eine Zusammenfassung und Bewertung der erreichten Ergebnisse sowie ein Ausblick auf mögliche Verbesserungen und Weiterentwicklungen wird schließlich in Kapitel 7 gegeben.

Im Rahmen dieser Arbeit werden diverse Konzepte und Aspekte anhand jeweils relevanter Teile eines konkreten Anwendungsbeispiels verdeutlicht. Dessen vollständiger Code findet sich in Anhang B.

2 Anforderungsanalyse

Um konkrete Anforderungen an die zu entwickelnde Sprache formulieren zu können, werden zunächst die bereits in Abschnitt 1.1 angedeuteten Ziele präzisiert. In den folgenden Abschnitten werden daraus die Hauptanforderungen an die Sprache abgeleitet.

2.1 Ziele

Zum einen soll im Rahmen dieser Arbeit eine Programmiersprache entwickelt werden, welche die Arbeit mit Datenbanken in den folgenden Bereichen vereinfachen soll:

- Datenverarbeitung
  - Verwaltung auch komplexer Verarbeitungsroutinen
  - Verwaltung dynamischer Verarbeitungsroutinen
  - Verarbeitung von Datentransfers
- Anwendungsentwicklung
  - Prototyping
  - Testen

Desweiteren ist es von Bedeutung, dem Aufwand der Entwicklung der Programmiersprache einen möglichst hohen Nutzen gegenüberzustellen. Konkret können dafür die folgenden Ziele formuliert werden:

- Eine möglichst grosse Nutzerbasis soll erreicht werden.
- Der Umstieg auf die zu entwickelnde Programmiersprache soll möglichst leicht fallen.

Im Folgenden werden aus diesen Zielen Anforderungen an die Programmiersprache abgeleitet und dargelegt. Zudem werden bestehende Technologien hinsichtlich dieser Anforderungen betrachtet.

2.2 Hauptanforderungen

Als Hauptanforderungen werden im Rahmen dieser Arbeit Anforderungen gesehen, die die zu entwickelnde Programmiersprache im Ganzen beeinflussen. In späteren Abschnitten werden darüber hinaus weitere Anforderungen formuliert, welche nur Teilsysteme der Programmiersprache betreffen oder sich aus den aufgeführten Hauptanforderungen ergeben.

2.2.1 Verwaltung komplexer Verarbeitungsroutinen

Die Verwaltung komplexer Verarbeitungsroutinen erfordert leistungsfähige Werkzeuge. Diese sollten dem Anwender möglichst wenige Beschränkungen hinsichtlich der Art der lösbaren Problemklassen auferlegen. Eine Möglichkeit, dies zu gewährleisten ist, das Werkzeug turing-vollständig zu konstruieren. Damit kann sichergestellt werden, daß jedes von einem Computer berechenbare Problem mit Hilfe des Werkzeuges theoretisch gelöst werden kann (vgl. [Tur36]).

Da *SQL* auf den Prizipien der relationalen Algebra beruht und somit wie diese nicht turing-vollständig ist (vgl. [War07, Kap. 13.1]), kann mit *SQL* allein nicht jedes berechenbare Problem gelöst werden. Ein Beispiel hierfür wäre das Durchlaufen rekursiver Datenstrukturen.

Die zu entwickelnde Programmiersprache sollte also die Funktionalität von SQL um **Turing- oder Berechnungsvollständigkeit** erweitern.

2.2.2 Verwaltung dynamischer Verarbeitungsroutinen

Die Verwaltung dynamischer Verarbeitungsroutinen ist in Situationen notwendig in denen sich Anforderungen häufig ändern und die betreffenden Verarbeitungsroutinen entsprechend häufig an die Anforderungen angepaßt werden müssen. Um diese so einfach wie möglich zu gestalten, sollte der Aufwand für die Aktivierung einer Änderung möglichst klein gehalten werden. Für eine Programmiersprache bedeutet dies, daß diese idealerweise in der Lage sein sollte, Programme ohne vorherige Kompilierung auszuführen. Dies führt zur nächsten Anforderungen an die zu entwickelnde Programmiersprache. Diese sollte über einen Interpreter verfügen und **Programme dynamisch, ohne Kompilierung ausführen** können.

2.2.3 Verarbeitung von Datentransfers

Die Verarbeitung von Datentransfers ist erforderlich, wenn Daten zwischen unterschiedlichen Datenbanken bzw. -systemen ausgetauscht werden müssen. Um dies zu ermöglichen, muss das entsprechende Werkzeug zumindest in der Lage sein, mit mehreren Datenbanken kommunizieren zu können. Für die zu entwickelnde Programmiersprache bedeutet

dies, daß diese die Arbeit mit **mehreren Datenbankverbindungen innerhalb eines Programmes** unterstützen muss.

2.2.4 Prototyping und Testen

Prototyping und Testen erfordern rasche, unkomplizierte Entwicklung sowie die Möglichkeit, interaktiv arbeiten zu können. Im Falle eines Werkzeuges für die Arbeit mit Datenbanken bedeutet dies, daß SQL-Befehle möglichst direkt abgesetzt werden können sollten. Für Programmiersprachen im Allgemeinen heißt dies, daß diese dem Entwickler ermöglichen sollte, sich hauptsächlich auf die Lösung des konkreten Problems zu fokussieren. Die Notwendigkeit, sich auf die Anforderungen der Programmiersprache zu konzentrieren sollte dabei möglichst gering gehalten werden.

Für die zu entwickelnde Programmiersprache bedeutet dies, daß diese einen Modus für die **interaktive Arbeit** bereitstellen sollte, in dem SQL-Befehle möglichst direkt, also wie vom jeweiligen Datenbanksystem erwartet, abgesetzt werden können. Außerdem sollte die Sprache Eigenschaften moderner Skriptsprachen wie zum Beispiel dynamische Typisierung unterstützen. Abschnitt 2.4.1 wird hierauf näher eingehen.

2.2.5 Erreichen einer grossen Nutzerbasis

Um eine grosse Nutzerbasis zu erreichen, sollte die Programmiersprache möglichst viele unterschiedliche Datenbanksysteme unterstützen. Durch diese **Datenbankunabhängigkeit** soll ein möglichst grosses potentielles Einsatzgebiet für die Programmiersprache geschaffen werden. Im Rahmen dieser Arbeit wird die Datenbankunabhängigkeit exemplarisch anhand der Datenbanksysteme *Oracle (ab Version 10g)*, *MySQL (Version 5.0)* sowie *PostgreSQL (Version 8.3)* demonstriert und verifiziert.

2.2.6 Einfacher Umstieg

Ein einfacher Umstieg auf die zu entwickelnde Programmiersprache kann gewährleistet werden, indem diese möglichst abwärtskompatibel gestaltet wird. Das heißt, sie sollte in der Lage sein, ein möglichst grosses Spektrum bestehender SQL-Skripts verarbeiten zu können. Diese **Abwärtskompatibilität** soll sicherstellen, daß der Wechsel von bestehenden Werkzeugen zu der zu entwickelnden Programmiersprache möglichst reibungslos erfolgen kann. Bestehende Skripte sollten entsprechend nur dort angepasst werden müssen, wo erweiterte Funktionalität der Programmiersprache erforderlich ist.

Ebenso wichtig für das Erreichen dieses Zieles ist die bereits erwähnte **Datenbankunabhängigkeit**. In Umgebungen, in denen meherere unterschiedliche Datenbanksysteme im Einsatz sind, soll dadurch eine gewisse Konsolidierung der verwendeten Werkzeuge ermöglicht werden.

2.2.7 Zusammenfassung

Zusammenfassend lassen sich die folgenden Hauptanforderungen formulieren:

- Berechnungsvollständigkeit
- Dynamische Ausführung
- Interaktive Ausführung
- Unterstützung mehrerer Verbindungen
- Abwärtskompatibilität
- Datenbankunabhängigkeit

Die konkreten Auswirkungen dieser Anforderungen auf die zu entwickelnde Programmiersprache werden in den folgenden Kapiteln erarbeitet.

2.2.8 Abgrenzung zu bestehenden Technologien

Zunächst sollen jedoch bestehende Technologien hinsichtlich der bisher definierten Anforderungen betrachtet werden. Es handelt sich dabei jeweils um Technologien, welche bereits im Datenbankumfeld eingesetzt.

Datenbankabstraktionsschichten

Die genannten Anforderungen lassen darauf schließen, dass sowohl gängige O/R-Mapper (z.B. *Hibernate*[1]) als auch abstrahierende Programmierschnittstellen (z.B. *JDBC*[2]) für die vorgestellten Einsatzszenarien nicht geeignet sind. Diese arbeiten zwar datenbankunabhängig, sind jedoch für den interaktiven Einsatz sowie für Prototyping kaum hilfreich. Oft ist ein relevanter Konfigurations– bzw. Einrichtungsaufwand mit dem Einsatz von O/R-Mappern verbunden, was der Anforderung der schnellen und einfachen Entwicklung widerspricht. Ganz im Gegenteil würde sich die zu entwickelnde Sprache eher dazu eignen, derartige Technologien mit spezieller Unterstützung in diesen Szenarien einsetzbar zu machen. Dies soll jedoch nicht Gegenstand dieser Arbeit sein.

Im Falle der abstrahierenden Programmierschnittstellen ist ebenso meist ein nicht zu vernachlässigender Aufwand an Verwaltungsarbeit hinsichtlich der Datenbankverbindungen und anderer Ressourcen neben den eigentlichen SQL-Kommandos zu leisten. Entsprechend scheidet auch diese Technologie für die direkte Verwendung in den genannten Szenarien aus. Stattdessen bietet sich die Verwendung derartiger Programmierschnittstellen für die Implementierung der Skriptsprache an, um die geforderte Datenbankunabhängigkeit einfach gewährleisten zu können.

[1] `http://www.hibernate.org/`
[2] `http://java.sun.com/products/jdbc/`

Embedded SQL

Eine den aufgeführten Anforderungen sehr nahe kommende Technologie ist *Embedded SQL*, welches Teil von *ISO/IEC 9075:1999*[3] und folgender Standards ist. Dieses erlaubt die Einbettung von SQL-Befehlen in Programmen verschiedener Programmiersprachen. Dadurch kann SQL direkt mit der Mächtigkeit allgemeiner Programmiersprachen kombiniert werden. Gleichzeitig ist es so konzipiert, daß der Verwaltungsaufwand im Umgang mit der Datenbank stark reduziert wird.

Dieser Standard existiert jedoch nur für eine geringe Anzahl von Programmiersprachen (Ada, C, COBOL, Fortran, MUMPS, Pascal, PL/I [vgl. ISO99b, Kap. 16]) und dessen Umsetzung ist meist sehr datenbankspezifisch. So ist für jedes Datenbanksystem ein spezieller Compiler notwendig. Für PostgreSQL ist dies zum Beispiel *ECPG*, der jedoch ausschließlich die Sprachen C und C++ unterstützt (vgl. [Pos08, Kap. 32]). Die Firma Oracle liefert für seine Datenbanksysteme jeweils Compiler für C/C++ und COBOL (vgl. [Ora08b, Kap. 1]). Somit ist hier die Anforderung nach Datenbankunabhängigkeit nur in gewissem Maße gegeben, da jedes Programm für jede Zieldatenbank separat übersetzt werden muss. Die Interaktion mit Datenbanksystemen unterschiedlicher Hersteller in ein und demselben Programm gestaltet sich entsprechend eher schwierig. Außerdem ist durch den Einsatz von kompilierten Sprachen die Anforderung nach der dynamischen Ausführung nicht erfüllt.

Eine Möglichkeit, diesen Unzulänglichkeiten zu begegnen, wäre sicher die Implementierung eines Präprozessors für eine bestehende Skriptsprache. Dies würde jedoch erfordern, jeden SQL-Befehl syntaktisch vom übrigen Programm abzuheben, wie zum Beispiel mittels des `EXEC SQL`-Befehlspräfix in *Embedded SQL*. Ziel dieser Arbeit soll jedoch auch sein, bestehende SQL-Skripte möglichst unverändert ausführen zu können und auf erweiterte Funktionalität des Interpreters bei Bedarf zurückgreifen zu können.

Verschiedene Clientprogramme

Natürlich existieren diverse Clientprogramme, sowohl proprietär als auch datenbankunabhängig, welche in der Lage sind SQL-Skripte auszuführen. Als Beispiel proprietärer Clientprogramme sei das Programm *psql*[4] genannt, welches Bestandteil des *PostgreSQL* Datenbanksystems ist. Dieses Programm ist in der Lage, beliebige SQL-Befehle sowohl interaktiv als auch aus Skriptdateien direkt auszuführen und ermöglicht damit einfache, schnelle und dynamische Entwicklung. Es ist jedoch ausschließlich für den Einsatz im Zusammenspiel mit dem PostgreSQL-Datenbanksystem gedacht und entsprechend nicht datenbankunabhängig. Außerdem ist die einzige weitergehende Funktionalität die Unterstützung von Variablen und deren Substitution in SQL-Befehlen. Verzweigungen,

[3] Im Rahmen der vorliegenden Arbeit wird der Standard *ISO/IEC 9075:1999* als Referenz für die *Structured Query Language* verwendet, da dieser im Gegensatz zu aktuelleren Versionen im Internet frei verfügbar ist. Dieser Standard wird hier auch kurz als *SQL-99* bezeichnet.

[4] `http://www.postgresql.org/docs/8.3/static/app-psql.html`

Schleifen, etc. bietet das Programm jedoch nicht, wodurch sich eine gewisse Limitierung ergibt und die Anforderung nach der Bereitstellung von Elementen vollwertiger Programmiersprachen nicht erfüllt ist.

Ähnliches gilt für datenbankunabhängige Clientprogramme wie zum Beispiel *SQuirreL SQL*[5] und andere. Zumeist unterstützen diese Variablen für die Parametrisierung von SQL-Befehlen, darüber hinausgehende Funktionalität wird jedoch kaum geboten. Für anspruchsvollere Scripting-Aufgaben sind diese entsprechend ebensowenig geeignet.

Eine weitere Technologie, welche den Anforderungen sehr nahe kommt, ist *SQL*Plus* der Firma Oracle. Hierbei handelt es sich um einen SQL-Skript-Interpreter der den SQL-Sprachumfang um Befehle für den Umgang mit Variablen, Betriebssystemkommandos, Ausgabeformatierung und anderes erweitert. Außerdem unterstützt *SQL*Plus* die Einbettung von PL/SQL-Konstrukten, wodurch der Funktionsumfang vollwertiger Programmiersprachen erreicht wird. Dennoch erweist sich auch diese Technologie für die gegebenen Anforderungen als unzureichend, da *SQL*Plus* ausschließlich für Oracle-Datenbanksysteme zur Verfügung steht. Datenbankunabhängigkeit ist somit nicht gegeben. Durch die Verwendung von PL/SQL wäre der Einsatz jedoch ohnehin konzeptuell auf Datenbanksysteme beschränkt, die Stored Procedures unterstützen.

Gegenüberstellung

Tabelle 2.1 zeigt einen zusammenfassenden Vergleich bestehender Technologien mit der zu entwickelnden Sprache. Als Kriterien der Gegenüberstellung werden die bisher formulierten Hauptanforderungen herangezogen.

2.3 Anforderungen an die Syntax

Die bereits formulierten Hauptanforderungen haben konkrete Auswirkungen auf die Sprache. Ebenso ergeben sich diverse Rahmenbedingungen aus der Natur des zu entwickelnden Systems. Diese sollen in den folgenden Abschnitten genauer analysiert werden.

2.3.1 Syntax der Zielsprache

Die Forderung nach dem Erreichen einer grossen Nutzerbasis (Abschnitt 2.2.5) und einem einfach Umstieg (Abschnitt 2.2.6) hat auch Konsequenzen für die Syntax der zu entwickelnden Sprache. Das Erlernen einer neuen Programmiersprache stellt eine signifikante Investition auf der Seite des Benutzers dar. Im Hinblick auf die Zielsprache und die formulierten Anforderungen sollte der Lernaufwand möglichst gering ausfallen, um einen einfache Einstieg in die Programmierung mit der Zielsprache zu ermöglichen. Neben einer einfachen, leicht verständlichen Semantik der Sprache ist hierfür auch die Verwendung einer gängigen syntaktischen Struktur sinnvoll, da dies potentiellen Benutzern der

[5] `http://www.squirrelsql.org/`

	Abstraktionsschichten[a]		Embedded SQL	Gängige Clientprogramme[b]		Zielsprache
	Hibernate	JDBC		SQL*Plus	SQuirreL SQL	
Dynamische Ausführung	nein	nein	nein	ja	ja	ja
Interaktive Ausführung	nein	nein	nein	ja	ja	ja
Berechnungsvollständigkeit	—[c]	—[c]	ja	ja	nein	ja
Datenbankunabhängigkeit	ja	ja	teilweise[d]	nein	ja	ja
Mehrere Verbindungen	ja	ja	nein	nein	nein	ja
Abwärtskompatibilität	—[e]	—[e]	—[e]	nein	nein	ja

[a] *Hibernate* dient in dieser Betrachtung als Beispiel für ein O/R-Mapping Framework. *JDBC* dient als Beispiel für eine Low-Level Abstraktionsschicht.

[b] *SQL*Plus* wurde als Stellvertreter für die Riege der proprietären Clientprogramme gewählt. *SQuirreL SQL* wird als Beispiel der datenbankunabhängigen Clientprogramme betrachtet.

[c] Sowohl *Hibernate* als auch *JDBC* dienen nicht zur Ausführung von Programmen bzw. Skripten, daher ist das Kriterium der Berechnungsvollständigkeit hier nicht zutreffend.

[d] *Embedded SQL* ist insofern datenbankunabhängig, als daß die Sprache in [ISO99a] standardisiert ist, jedoch benötigen die Implementierungen der verschiedenen Hersteller jeweils unterschiedliche Compiler, weswegen hier nur von einer teilweisen Datenbankunabhängigkeit gesprochen werden kann.

[e] Weder *Hibernate*, noch *JDBC* und *Embedded SQL* dienen der Ausführung von SQL-Skripts weswegen das Kriterium der Abwärtskompatibilität im hier verwendeten Sinne nicht zutreffend ist.

Tabelle 2.1: Gegenüberstellung verschiedener Technologien

Sprache ermöglicht, auf vorhandenes Wissen zurückzugreifen. Der Begriff *gängig* kann natürlich in diesem Zusammenhang nicht konkret definiert werden. Abschnitt 4.1 geht hierauf genauer ein.

2.3.2 Einbettung der Structured Query Language

In erster Linie soll ein Interpreter für SQL-Skripte entwickelt werden. Entsprechend muß SQL eine Teilmenge der unterstützten Sprache sein.

Daraus ergeben sich einige Konsequenzen für die Syntax der Zielsprache. So unterstützt SQL zum Beispiel verschiedene syntaktische Formen, um Zeichenkettenliterale zu notieren. Diese können auch Zeichen enthalten, die in der Zielsprache eine besondere Bedeutung haben, wie zum Beispiel Befehlstrenner. Innerhalb einer SQL-Zeichenkette dürfen diese aber den Befehl nicht abschließen. Insofern ist es notwendig, daß die Zielsprache derartige syntaktische Eigenschaften der Structured Query Language kennt und berücksichtigt.

Im Folgenden wird die Structured Query Language hinsichtlich dessen untersucht und die sich daraus ergebenden Anforderungen an die Zielsprache aufgezeigt.

Dialekte

Aus der Forderung nach Datenbankunabhängigkeit (Abschnitt 2.2.5) ergibt sich die Notwendigkeit, verschiedene SQL-Dialekte möglichst flexibel zu unterstützen. Zu beachten ist hier, daß die Dialekte, die von den verschiedenen Datenbanksystemen unterstützt werden, jeweils den in *SQL-99* definierten Befehlsumfang proprietär erweitern oder auch nur teilweise unterstützen. Dies bedeutet, daß keine feste Menge an SQL-Schlüsselworten in der Sprache benutzt werden kann. Entsprechend muss entweder, je nach verwendeter Datenbank, die jeweils unterstützte Menge der Schlüsselworte dynamisch geladen werden oder die Sprache so gestaltet werden, dass eine Unabhängigkeit vom jeweiligen SQL-Dialekt besteht.

Befehle

Allen SQL-Befehlen gemein ist, unabhängig vom Dialekt, die syntaktische Grundstruktur: Beginn mit einem Wort[6] optional gefolgt von Whitespace, beliebigen Zeichen und Kommentaren oder Zeichenketten. Derartige Befehle müssen entsprechend in der Sprache als SQL-Befehle erkannt werden. Ebenso bietet diese Struktur Möglichkeiten der syntaktischen Abgrenzung von SQL-Befehlen und Skriptbefehlen.

[6] Bestehend aus beliebigen Zeichen des englischen Alphabets.

Befehlsfolgen

In *SQL-99* definiert ist zwar die Syntax der standardisierten SQL-Befehle, nicht jedoch die Trennung von mehreren aufeinanderfolgenden Befehlen. Neben dem Semikolon (;), welches von den meisten Clients (*mysql*, *psql*, *SQL*Plus*) als Befehlstrenner interpretiert wird, werden je nach Client weitere Sequenzen als Trenner unterstützt. Tabelle 2.2 zeigt eine Aufstellung der unterstützten Befehlstrenner.

	Oracle SQL*Plus	MySQL mysql	PostgreSQL psql
;	x	x	x
\g		x	x
\G		x	
/ (einziges Zeichen auf einer Zeile)	x		
(Leerzeile)	x		

Tabelle 2.2: Befehlstrennzeichen verschiedener SQL-Clients

Hinzu kommt die Tatsache, daß sowohl Oracle als auch MySQL *SQL procedure statements* (also den Rumpf des `CREATE PROCEDURE`- bzw. `CREATE FUNCTION`-Befehls – vgl. [Sun09, Kap. 12.8], [Ora08a, Kap. 14]) als Semikolon-separierte Befehlsliste interpretieren. Es ist also erforderlich, entweder diese Befehle in der Sprache speziell zu behandeln, wie dies zum Beispiel in *SQL*Plus* der Fall ist oder den verwendeten Befehlstrenner wählbar zu machen, wie im *mysql*-Client.

Kommentare

SQL-99 definiert sowohl die Doppelstrichsyntax (zwei Minuszeichen gefolgt von beliebig vielen beliebigen Zeichen gefolgt von einem Zeilenumbruch) als Syntax für einzeilige Kommentare als auch die aus Sprachen wie *C* oder *Java* bekannte `/* ... */`-Syntax für mehrzeilige Kommentare. *PostgreSQL* unterstützt im Unterschied zu *Oracle* und *MySQL* auch die Verschachtelung mehrzeiliger Kommentare wie vom Standard vorgesehen. Da auch Oracle mit *SQL*Plus* weitere Kommentarformen (über den `REMARK`-Befehl) bereitstellt sollte die Unterstützung verschiedener, wählbarer Kommentarsyntaxformen vorgesehen werden.

Zeichenketten

Die Syntax normaler Zeichenkettenliterale, die *SQL-99* definiert, wird von allen untersuchten Clients und Datenbanksystemen unterstützt und ist somit auch für die zu ent-

wickelnde Sprache unerläßlich. Dennoch gibt es Unterschiede in der Behandlung von Zeichenketten. So definiert *SQL-99* Bezeichnerzeichenketten[7] die von doppelten Anführungszeichen begrenzt werden. In *MySQL* werden derartige Zeichenketten standardmäßig ebenfalls als normale Zeichenketten interpretiert. Bezeichnerzeichenketten werden hingegen durch Backticks[8] repräsentiert (vgl. [Sun09, Kap. 8.1.1 u. Kap. 8.2]). *PostgreSQL* verhält sich dahingehend standardkonform, implementiert jedoch außerdem Variationen, die über den Standard hinausgehen. Gleiches gilt für *Oracle*, mit der Ausnahme, daß dort, anders als in *SQL-99*, Bezeichnerzeichenketten keine doppelten Anführungszeichen enthalten können (vgl. [Ora08c, Kap. 2]). Tabelle 2.3 zeigt eine Übersicht der verschiedenen Formen von Zeichenkettenliteralen.

Das korrekte Interpretieren von Zeichenketten ist elementar, da diese auch Zeichen enthalten können, welche in der Sprache eine besondere Bedeutung haben. So könnte ein Semikolon in einer Zeichenkette fälschlicherweise als Befehlstrenner interpretiert werden, wenn diese nicht als solche erkannt wird.

Da zum Zeitpunkt der Übersetzung eines Programmes der Sprache nicht notwendigerweise bereits eine Datenbankverbindung bestehen muß bzw. eine zum Zeitpunkt der Übersetzung bestehende Verbindung nicht identisch mit der zur Ausführungszeit bestehenden sein muß, kann daraus keine Information gewonnen werden aus der die konkrete, zu interpretierende Zeichenkettensyntax abgeleitet werden könnte.

Es muß also entweder eine klare Beschränkung der unterstützten Syntax geben – z.B. auf die in *SQL-99* definierte – oder in der Sprache Mittel zur Verfügung gestellt werden, mit deren Hilfe der Benutzer die Art des aktiven Datenbanksystems bzw. dessen verwendete Syntax explizit spezifizieren kann. Ersteres wäre allerdings der Abwärtskompatibilität nicht förderlich.

2.4 Anforderungen an die Semantik

Neben den in Abschnitt 2.3 behandelten Anforderungen an die Syntax ziehen die spezifizierten Hauptanforderungen ebenso Anforderungen an die Semantik der Zielsprache nach sich. Diese werden in den folgenden Abschnitten konkretisiert.

2.4.1 Semantik der Zielsprache

Da die Unterstützung von Prototyping eine zentrale Forderung an die Sprache ist (siehe Abschnitt 2.2.4), sollte diese explizit berücksichtigt werden. Eine Möglichkeit ist der Verzicht auf nicht zwingend notwendige Formalismen. Hierunter fällt zum Beispiel die Art der Typisierung in der Sprache. Statt statischer Typisierung, bei welcher der Typ

[7] *Delimited identifiers* — Zeichenketten, die statt normalen Texts Namen von Objekten einer Datenbank bezeichnen, vgl. [ISO99a, Kap. 5.4]

[8] Gemeint ist das Zeichen „‘“. Mangels einer griffigen, kurzen deutschen Bezeichnung für dieses Zeichen wird im Rahmen dieses Dokuments die englische Bezeichnung verwendet.

	Begrenzer	Escapes[a]	Backslash-Escapes[b]
SQL-99	' ... '	''	Nein
	"..."	""	Nein
Oracle	' ... '	''	Nein
	"..."	–	Nein
	q'X ... X'[c]	–	Nein
MySQL	' ... '	''	Ja[d]
	"..."	""	Ja[d]
	` ... `	``	Nein
PostgreSQL	' ... '	''	Ja[d]
	E' ... '	''	Ja
	"..."	""	Nein
	X ... X[e]	–	Nein

[a] Zeichensequenz, die innerhalb des jeweiligen Zeichenkettenliterals verwendet werden kann, um das Begrenzerzeichen zu erzeugen.

[b] Bei Unterstützung von Backslash-Escapes kann das jeweilige Begrenzerzeichen durch ein vorangestelltes Backslash-Zeichen erzeugt werden. Ein Backslash-Zeichen wiederum kann ebenfalls durch vorangestelltes Backslash-Zeichen erzeugt werden.

[c] *X* kann ein beliebiges Zeichen sein. Sofern X am Anfang der Zeichenkette aus einem der Zeichen <, (, { oder [besteht, muß dieses mit einem der Zeichen >,), } bzw.] am Ende der Zeichenkette korrespondieren.

[d] Sofern die entsprechende Einstellung aktiviert ist.

[e] *X* ist optional und kann eine beliebige Zeichenkette sein, die der Syntax von Bezeichnern in PostgreSQL entspricht.

Tabelle 2.3: Übersicht verschiedener Formen von Zeichenkettenliteralen

jeder Variablen und jeden Rückgabewertes explizit spezifiziert werden muß, sollte die Sprache **dynamische Typisierung** verwenden. Diese Art der Typisierung unterstützt eine zügige Entwicklung und ist bei Skriptsprachen weit verbreitet.

2.4.2 Beachtung der Semantik von SQL

Bei der Arbeit mit der Structured Query Language muß deren Semantik natürlich berücksichtigt werden. Dieser Abschnitt befasst sich mit relevanten Punkten derselben.

Befehlsklassen

SQL-99 teilt die verschiedenen SQL-Befehle unter anderem in unterschiedliche Klassen je nach deren Funktion ein (vgl. [ISO99a, Kap. 4.30]). Die Einteilung der Befehle in die folgenden Klassen ist für die vorliegende Arbeit von besonderer Relevanz:

Schemabefehle Hierbei handelt es sich um Befehle, welche dauerhafte Auswirkungen auf Datenbankschemata haben können. Derartige Befehle haben in der Regel keinen Rückgabewert.

Datenbefehle Dies sind Befehle, welche auf den Daten einer Datenbank arbeiten bzw. temporäre Auswirkungen auf Datenbankschemata haben. Also besonderer Fall gehört hierzu der `SELECT`-Befehl, welcher sich durch die Rückgabe von Daten in einer Ergebnismenge auszeichnet.

Daten verändernde Befehle Als Unterkategorie der Datenbefehle werden die *datenverändernden Befehle* geführt. Diese können einen dauerhaften Effekt auf die Daten einer Datenbank haben und umfassen die `INSERT`-, `UPDATE`- und `DELETE`-Befehle. Die Besonderheit dieser Befehle ist, daß sie statt einer Ergebnismenge die Anzahl der veränderten Datensätze als Ergebnis liefern. Zu `INSERT`-Befehlen können unter Umständen auch ein oder mehrere automatisch generierte Schlüssel zurückgegeben werden.

Transaktionsbefehle Bei diesen Befehlen handelt es sich um Befehle, mit deren Hilfe Transaktionen verwaltet werden können. Da die Verwaltung von Transaktionen meist von den jeweils verwendeten Programmierschnittstellen übernommen wird ist hier besonders auf eventuell auftretende Konflikte zu achten.

Die Besonderheiten der verschiedenen Befehle müssen in der zu entwickelnden Sprache berücksichtigt werden. Die Notwendigkeit hierzu ergibt sich aus den unterschiedlichen Ergebnissen bzw. speziellen Effekten der jeweiligen Befehle.

Dynamische SQL-Befehle

Dynamische SQL-Befehle sind Befehle, deren vollständiger Befehlstext erst zur Laufzeit bekannt ist. Für deren Realisierung gibt es zwei Möglichkeiten:

Einbettung von Variablenreferenzen in den Befehlstext. Der vollständige Befehlstext wird dabei zur Laufzeit von der Skriptsprache generiert. Hierfür ist es notwendig, eine möglichst eindeutige, konfliktfreie Syntax zu finden. Aufgrund der starken Variationen zwischen den verschiedenen SQL-Dialekten ist es nicht möglich Konflikte mit der SQL-Befehlssyntax aller Datenbanksysteme sicher auszuschließen. Ziel kann daher nur sein, das Konfliktpotential so gering wie möglich zu halten und für Konfliktfälle Möglichkeiten vorzusehen, diese durch explizite Anweisungen des Benutzers umgehbar zu machen.
Diese Art dynamischer SQL-Befehle ermöglicht, sämtliche Komponenten eines SQL-Befehls, z.B. auch Objektnamen, dynamisch zu generieren.

Parametrisierung von Befehlen mit Hilfe von *Prepared Statements*. Hierbei wird dem Datenbanksystem ein Befehl übermittelt, welcher anstelle konkreter Werte Platzhalter enthalten kann. Der Befehl kann dann vom Datenbankserver vorbereitet werden. Für die eigentliche – auch mehrmalige – Ausführung müssen schließlich nur noch die konkreten Werte der Parameter übermittelt werden.
SQL-99 definiert das Fragezeichen als Platzhaltersymbol in [ISO99a, Kap. 4.24], die im Folgenden auch als *positionale Parameter* bezeichnet werden. Daneben hat auch die nicht standardisierte Form der *benannten Parameter* recht weite Verbreitung gefunden. Dabei erhält jeder Parameter einen Namen und wird im Befehl durch einen vorangestellten Doppelpunkt markiert. Durch die fehlende Standardisierung ist jedoch auf etwaige Konflikte zu achten, etwa mit *PostgreSQL*s proprietärer Type-Casting-Syntax (spezifiziert in [Pos08, Kap. 4.2.8]).
Mit Hilfe dieser Art dynamischer SQL-Befehle können unter anderem große Performanzgewinne erzielt werden, wenn ein Befehl wiederholt mit unterschiedlichen Parametern ausgeführt werden soll, da der Befehl vom Datenbanksystem nur einmal interpretiert und analysiert werden muss. Allerdings erlaubt diese Art dynamischer Befehle nur die dynamische Spezifikation von Werten – Objektnamen und ähnliches können nicht übergeben werden.

Da beide Arten von dynamischen SQL-Befehlen unterschiedliche Vor- und Nachteile haben, sollten beide in der zu entwickelnden Sprache implementiert werden.

Batch-Ausführung

Viele Datenbanksysteme unterstützen die Batch-Ausführung mehrerer Befehle, bei der alle Befehle des Batchlaufs gemeinsam an den Datenbankserver gesendet werden und anschließend alle Ergebnisse gemeinsam zurück zum Client geschickt werden. Da hierdurch

in bestimmten Anwendungsfällen grosse Performanzgewinne erzielt werden können, ist eine direkte Untersützung empfehlenswert.

Zu unterscheiden sind hier Stapel von mehreren nicht-parametrisierten Befehlen und Stapel eines parametrisierten Befehls mit mehreren Wertepaaren. Beide Arten bedienen unterschiedliche Einsatzszenarien, sodaß beide Arten unterstützt werden sollten.

2.5 Anforderungen an die Maschine

Aus den bisher definierten Hauptanforderungen ergeben sich im Besonderen zwei Konsequenzen für die zu implementierende Maschine.

Zum einen betrifft die Forderung nach interaktiver Ausführung (Abschnitt 2.2.4) hauptsächlich die Maschine. Diese muß entsprechend in der Lage sein, einzelne Befehle individuell aber dennoch in einem gemeinsamen Kontext auszuführen.

Zum anderen ergeben sich Konsequenzen aus der Forderung nach Abwärtskompatibilität (Abschnitt 2.2.6), also der Forderung, bestehende SQL-Skripte ausführen zu können. So enthalten SQL-Skripte naturgemäß oft auch Daten in teilweise recht großem Umfang. Die zu implementierende Maschine sollte in der Lage sein auch diese Skripte auszuführen. Dies erfordert einen besonderen Umgang mit derartigen Skripten. Die gängige Vorgehensweise bei der Übersetzung von Programmen, Programme vollständig zu parsen, übersetzen und anschließend auszuführen dürfte bei sehr grossen Programmen zu einem unverhältnismäßig hohen Ressourcenverbrauch führen. Die Maschine sollte entsprechend die Möglichkeit bieten, Programme inkrementell zu übersetzen und auszuführen.

3 Grundlagen

In diesem Kapitel werden relevante Konzepte und Techniken für den Entwurf der zu entwickelnden Sprache sowie der zugehörigen Maschine untersucht. Die im vorigen Kapitel definierten Anforderungen an die Sprache werden dabei berücksichtigt.

Zunächst werden allgemeine Entwurfskonzepte für Programmiersprachen wie Programmierparadigmen vorgestellt, wobei auf das Konzept der objektorientierten Programmierung näher eingegangen wird. Anschließend werden verschiedene Ansätze von Typsystemen betrachtet. Die folgenden Abschnitte thematisieren die Verwendung von Namen in Programmiersprachen und stellen damit verbundene Konzepte und Techniken vor. Darauf folgt ein Abschnitt in welchem Funktionen behandelt werden und Konzepte für deren Verwendung und Implementierung untersucht werden. Schließlich wird der Vorgang der Übersetzung betrachtet, gefolgt von der Untersuchung der Ausführung von Programmen der Zielsprache und dafür relevanten Techniken.

3.1 Programmierparadigmen

Programmiersprachen im Allgemeinen können nach unterschiedlichsten Stilen bzw. Paradigmen entworfen werden. Je nachdem, welche Ziele bei der Entwicklung im Vordergrund stehen, kann dabei ein zentraler Stil verwendet oder eine Kombination verschiedener Stile verfolgt werden.

Für die zu entwickelnde Sprache kommen aufgrund der bereits vorgestellten Anforderungen mehrere Stile in Frage. Diese werden im Folgenden vorgestellt und deren besondere Relevanz erläutert.

3.1.1 Imperative Programmierung

Bei der imperativen Programmierung werden Programme als Folge von Anweisungen repräsentiert, die deren Zustand verändern. Den Gegenpol dazu stellt die deklarative Programmierung dar. Bei dieser handelt es sich um einen Stil, in welchem Programme unter anderem frei von Nebenwirkungen sind. Im Unterschied zur imperativen Programmierung beschreiben Programme im deklarativen Stil was berechnet werden soll, statt wie die Berechnung erfolgen soll. Gängige Konzepte wie globale Variablen, die gerade bei explorativer Verwendung von Programmiersprachen sehr häufig zum Einsatz kommen, werden allerdings in der deklarativen Programmierung nicht unterstützt. Diese Eigenschaft stellt daher eine gewisse Restriktion dar, die im Kontrast zur Forderung nach

einfachem Einstieg steht. Aufgrund dessen wird die Zielsprache imperative Programmierung unterstützen.

3.1.2 Prozedurale und Strukturierte Programmierung

Prozedurale Programmierung basiert auf dem Ansatz, Programmteile die aus mehreren Anweisungen bestehenden als Einheit zu betrachten und als Unterprogramm zu definieren. Diese Unterprogramme können von beliebigen Stellen eines Programmes aus aufgerufen werden, ohne die einzelnen Anweisungen jeweils duplizieren zu müssen. Dadurch wird eine einfachere Wiederverwendung von bestehendem Code ermöglicht und die Übersichtlichkeit und Wartungsfreundlichkeit erhöht.

Strukturierte Programmierung baut darauf auf und basiert auf der Verwendung von expliziten Kontrollstrukturen anstelle von Sprungbefehlen zur Modellierung des Kontrollflusses innerhalb eines Programms.

Beide Konzepte sind in der imperativen Programmierung seit geraumer Zeit etabliert und aufgrund ihrer Wichtigkeit bei der Erstellung von wartbaren Programmen, welche nach dem imperativen Paradigma modelliert sind, für die zu entwickelnde Sprache unerläßlich.

3.1.3 Objektorientierte Programmierung

Im ebenfalls etablierten, mittlerweile dominanten, objektorientierten Programmierstil werden Programme als Sammlung von Objekten modelliert, welche Daten mit den darauf möglichen Operationen vereinen. Eine Kollaboration der Objekte untereinander wird durch den Austausch von Nachrichten ermöglicht.

Da dieser Stil, aufgrund seiner Vorzüge bezüglich Modellierung, Wiederverwendung und Wartbarkeit, sehr weit verbreitet ist soll auch die zu entwickelnde Sprache diesen unterstützen.

3.1.4 Prototyp-basierte Programmierung

Die prototyp-basierte Programmierung ist ein Konzept der objektorientierten Programmierung. Im Unterschied zum klassenbasierten Ansatz erfolgt bei diesem die Modellierung von Objekten nicht mithilfe von Klassen, die deren Struktur definieren. Stattdessen werden Objekte erzeugt, indem bestehende Objekte kopiert werden. Deren Struktur wird entsprechend übernommen und kann anschließend modifiziert werden kann.

Da die prototyp-basierte Programmierung besonders schnelle Entwicklung ermöglicht, indem die Modellierung einer Objekthierarchie statt von abstrakten Konzepten von konkreten Beispielen ausgeht, eignet sich diese besonders für die mit der zu entwickelnden Sprache abzudeckenden Einsatzszenarien.

3.1.5 Funktionale Programmierung

Die funktionale Programmierung ist ein Paradigma in welchem Programme als Menge von mathematischen Funktionen modelliert werden. Nebenwirkungen durch Zustandsänderung von Variablen werden dabei vermieden. Stattdessen werden Werte an Funktionen übergeben. Diese wiederum liefern ein Ergebnis das an andere Funktionen übergeben werden kann.

Ein besonders attraktives Konzept funktionaler Programmierung sind Funktionen höherer Ordnung. Damit werden Funktionen bezeichnet, welchen Funktionen als Argumente übergeben werden können und welche Funktionen als Rückgabewerte liefern können. Besonders dieses Konzept ermöglicht teilweise eine im Vergleich zur rein imperativen Programmierung recht flexible und kompakte Modellierung von Programmen. So ermöglicht dieses Konzept zum Beispiel die Definition einer generischen Funktion mit welcher die Werte beliebiger Listen verändert werden können. Als Argumente kann diese Funktion dann eine Liste und eine weitere Funktion übernehmen, welche die konkrete Transformation eines Wertes der Liste definiert. In der rein imperativen Programmierung muß hier im Gegensatz dazu jeweils ein eigenständiges Schleifenkonstrukt verwendet werden.

Rein funktionale Programmierung ist für die Zielsprache aufgrund der nicht vorhandenen Nebenwirkungen ebenso nicht erstrebenswert wie die deklarative Programmierung. Eine Kombination aus imperativer und funktionaler Programmierung, insbesondere mit der Unterstützung von Funktionen höherer Ordnung wie sie bereits in diversen Skriptsprachen, wie zum Beispiel *Python* oder *JavaScript*, zum Einsatz kommt, verspricht jedoch für die gegebenen Einsatzszenarien besonders geeignet zu sein.

3.2 Objektorientierte Programmierung

Nach der kurzen Vorstellung der für die zu entwickelnde Sprache relevanten Programmierparadigmen geht der folgende Abschnitt näher auf das Konzept der objektorientierten Programmierung im Kontext der Zielsprache ein.

3.2.1 Prototyp-basierte Objektorientierung

Wie bereits in Abschnitt 3.1 beschrieben wird für die Zielsprache der prototyp-basierte Ansatz der Objektorientierung verfolgt. Dieser unterscheidet sich in einigen Bereichen grundlegend vom allgemein bekannten klassenbasierten Ansatz der Objektorientierung. Auf diese Unterschiede soll im Folgenden eingegangen werden.

3.2.2 Erzeugung von Objekten

Im Gegensatz zur klassenbasierten Objektorientierung gibt es bei der prototypbasierten Variante zwei verschiedene Möglichkeiten, Objekte zu erzeugen. Eine Möglichkeit der

Erzeugung ist das Klonen von bestehenden Objekten, eine andere Möglichkeit ist die Erzeugung aus dem Nichts, auch *Ex-Nihilo*-Erzeugung (vgl. [DMC92]) genannt.

Beide Varianten bieten unterschiedliche Vorteile. So kann durch das Klonen von Objekten das Verhalten eines Objektes auf eine neues Objekt übertragen werden. Ex-Nihilo-Erzeugung hingegen ermöglicht das Erzeugen von leeren Objekten, also Objekten ohne definiertes Verhalten oder spezifizierte Eigenschaften.

3.2.3 Vererbungskonzepte

In prototyp-basierten Systemen existieren ebenso zwei unterschiedliche Ansätze, Vererbung bzw. gemeinsames Verhalten mehrerer Objekte zu realisieren. Zum einen kann dies durch *Delegation* geschehen wie sie zum Beispiel [Lie86] beschreibt, zum anderen durch *Konkatenation*, auch als *Einbettung* bezeichnet (vgl. [Sav08]).

Bei der Delegation werden Objekte mit einer Referenz auf ein weiteres Objekt versehen, an welches Nachrichten delegiert werden, die das ursprüngliche Empfängerobjekt nicht versteht.

Bei der Konkatenation hingegen werden Objekte stattdessen vollständig kopiert. Dieser Ansatz hat jedoch den Nachteil, daß sowohl die effiziente Nutzung von Speicherplatz für Objekte als auch die Propagierung von Änderungen im Objektsystem schwierig ist. Ebenso schwierig ist es, im Kontext der Konkatenation Vererbungsbeziehungen aufzuheben (vgl. [Sav08]). Delegation teilt diese Nachteile nicht und ist daher der Ansatz der für die Zielsprache verfolgt werden soll.

Es gibt jedoch bei dem Ansatz der Delegation unterschiedliche Ausprägungen. So werden zum Beispiel in der Sprache *Self* Delegationsreferenzen explizit gesetzt und beim Klonen von Objekten diese vollständig kopiert (vgl. [US87]). Gemeinsames Verhalten mehrerer Objekte wird dabei über *Traits* genannte Objekte realisiert, welche ausschließlich für die Modellierung von Verhalten zuständig sind. Ein anderer in [Lie86] vorgestellter Ansatz realisiert Delegation auch implizit über die Klonierung. Hier werden Objekte nicht kopiert, sondern neue Objekte erzeugt und mit einer Delegationsreferenz auf das geklonte Objekt gesetzt. Dieses Verfahren wurde unter anderem in der Sprache *NewtonScript* eingesetzt. Auch in aktuellen Sprachen, wie zum Beispiel *IO*[9], findet es Verwendung, teilweise unter dem Namen *Differentielle Vererbung*. Da dieser Ansatz auf dem ohnehin zu verfolgenden Ansatz der Delegation basiert und eine noch höhere Flexibilität verspricht, soll auch dieser in der Zielsprache verfolgt werden.

3.2.4 Verwendungskonzepte

Der eigentliche Gedanke der bei der prototyp-basierten Objektorientierung verfolgt wird ist der, daß statt Klassen konkrete Beispielexemplare von Objekten verwendet werden,

[9] `http://www.iolanguage.com/`

die voll funktionsfähig sind. Anstelle des Erzeugens von Instanzen anhand von Klassen wird dabei das Beispielexemplar kopiert und die Kopie modifiziert.

Dennoch erlaubt die prototyp-basierte Objektorientierung auch eine eher an die klassenbasierte Objektorientierung angelehnte Modellierung (vgl. [Lie86], [Smi94], [Smi95]).

Da die klassenbasierte Objektorientierung das weiter verbreitete Konzept im Vergleich zur prototyp-basierten Objektorientierung ist, sollte die Zielsprache für die Verwendung nach diesem Prinzip besondere Unterstützung bieten.

3.2.5 Alles ist ein Objekt

Programmiersprachen wie *Smalltalk* oder *Self* haben gezeigt, daß der Ansatz, die Elemente einer Sprache möglichst uniform zu gestalten, diverse Vorteile mit sich bringt. Besonders die Modellierung jeglicher Elemente einer Sprache durch Objekte bringt ein sehr konsistentes Erscheinungsbild aber auch sehr hohe Flexibilität mit sich, da weite Teile des Systems beliebig manipuliert werden können (vgl. [GR02], [US87]).

Da diese Flexibilität gerade bei der explorativen Anwendung von Programmiersprachen sehr nützlich sein kann erscheint es sinnvoll, diesen Ansatz, soweit praktikabel, in der Zielsprache ebenso zu verfolgen.

Ein wesentliches Element in diesem Ansatz ist die bereits beschriebene Unterstützung von Funktionen höherer Ordnung. So ermöglichen erst diese im Zusammenspiel mit Polymorphie die Modellierung von Kontrollstrukturen durch einfache Objekte, was insbesondere dadurch erstrebenswert ist, daß diese Methode eine sehr hohe Flexibilität ermöglicht, indem die Sprache einfach um neue Kontrollstrukturen erweitert werden kann. Eine Mögliche Anwendung in der Zielsprache könnte zum Beispiel die Implementierung von Transaktionsblöcken sein, wie Listing 3.1 es demonstriert, wobei ein Verbindungsobjekt eine Methode bereit stellen könnte, die eine Funktion als Argument entgegennimmt, eine Transaktion startet, anschließend die übergebene Funktion ausführt und schließlich die Transaktion wieder beendet. Auf ähnliche Weise könnte die Zielsprache, um Unterstützung für Transaktionsblöcke für verteilte Transaktionen erweitert werden.

Die in Abschnitt 2.3.1 formulierte Forderung nach gängiger Syntax in der Zielsprache macht es jedoch erforderlich, einige Kontrollstrukturen auf herkömmliche Weise zu repräsentieren. So empfiehlt es sich, gängige Kontrollstrukturen wie `if`- oder `while`-Anweisungen durch eine jeweils spezielle Syntax zu repräsentieren. Die Implementierung dieser Kontrollstrukturen kann dennoch über den objektorientierten Ansatz erfolgen, indem derartige syntaktische Konstrukte während der Übersetzung eines Programms in das jeweilige semantische Äquivalent dieses Ansatzes transformiert werden. Syntaktische Kontrollstrukturen wären demnach nicht mehr als *syntaktischer Zucker* (engl. syntactic sugar), würden jedoch den Einstieg beim Umgang mit der Zielsprache sicher erleichtern. Listing 3.2 zeigt beide Varianten.

Ein weiteres zentrales Element in diesem Ansatz ist die Repräsentation von jeglichen Operationen durch das Senden von Nachrichten bzw. das Aufrufen von Methoden. Hier-

Listing 3.1: Transaktionsblock

```
Connection.inTransaction = fun (action) {
    this.beginTransaction();

    try {
        action();
    }
    catch (exception) {
        this.rollbackTransaction();
        return false;
    }

    this.commitTransaction();
    return true;
};

.Connection.inTransaction(fun () {
    -- execute some SQL statements
});
```

Listing 3.2: Kontrollstrukturen

```
-- Bedingte Verzweigung im imperativen Stil
if (someCondition) {
    doSomething();
}
else {
    doSomethingElse();
}

-- Bedingte Verzweigung im objektorientierten Stil
.(someCondition).ifThen(
    fun () { doSomething(); },
    fun () { doSomethingElse(); }
);
```

durch wird sowohl das Überladen von Operatoren als auch die Erweiterung der Sprache um neue Operatoren ermöglicht was auch zu einer hohen Flexibilität der Zielsprache führt. Listing 3.3 demonstriert das Überladen von Operatoren sowie die Definition eines neuen Operators.

3.3 Typisierung

Nach den grundlegenden Paradigmen ist die Art des verwendeten Typsystems eine weitere bestimmende Eigenschaft von Programmiersprachen. Typsysteme dienen dazu, Werten in einer Programmiersprache mit Typen zu versehen, welche wiederum bestimmen welche Operationen auf diesen Werten gültig sind. Auch bei den Typsystemen existieren

Listing 3.3: Überladen von Operatoren

```
1  -- Ueberladen des Operators + als Konkatenationsoperation
2  -- fuer String-Objekte
3  .String.+ = fun (aString) {
4      return this.concat(aString);
5  };
6
7  var s := "Hello " + "World!";
8
9  -- Definition des Operators | fuer SQL-Statement-Objekte
10 -- als Datentransferoperation.
11 .SQLStatement.| = fun (aStatement) {
12     while (this.hasMoreData()) {
13         var data := this.fetchData();
14         aStatement.executeWithData(data);
15     }
16 };
17
18 var srcStmt := ... a SELECT statement ...;
19 var dstStmt := ... an INSERT statement ...;
20
21 .srcStmt | dstStmt;
```

unterschiedliche Ansätze, von denen einige sich mehr für den angestrebten Einsatz der zu entwickelnden Sprache bei Prototyping und interaktiver Anwendung eignen als andere. Die geeigneten Varianten werden im folgenden kurz vorgestellt.

3.3.1 Dynamische Typisierung

Die dynamische Typisierung ist das Gegenstück zur statischen Typisierung. Bei statisch typisierten Programmiersprachen haben Variablen jeweils einen festen Typ und können nur Werte diesen Typs annehmen. Die Einhaltung dieses Grundsatzes wird dabei zum Zeitpunkt der Übersetzung eines Programmes geprüft. Eventuelle Verletzungen führen zum Scheitern des Übersetzungsvorgangs.

Im Unterschied dazu können Variablen bei der dynamischen Typisierung Werte verschiedener Typen annehmen. Eine Angabe des Typs von Variablen muss bei deren Deklaration entsprechend nicht erfolgen. Eventuell durchgeführte Typprüfungen werden anders als bei der statische Typisierung nicht zum Zeitpunkt der Übersetzung eines Programms sondern während dessen Ausführung, also zur Laufzeit, durchgeführt.

Diese Art der Typisierung findet sehr häufig bei Skriptsprachen Verwendung, da dadurch eine recht hohe Flexibilität sowie eine größere Kompaktheit des Programmcodes erreicht werden kann. Als Nachteil der dynamischen Typisierung gegenüber der statischen kann gesehen werden, daß weniger Korrektheitszusagen über den gegebenen Programmcode zur Übersetzungszeit getroffen werden können. Für Anwendungsgebiete wie Prototyping ist dieser Nachteil jedoch eher als gering einzustufen sodaß die Vorteile durch

die zügigere Entwicklung überwiegen und die dynamische Typisierung entsprechend die für die zu entwickelnde Sprache zu favorisierende Variante darstellt.

3.3.2 Starke Typisierung

Mit starker Typisierung ist in diesem Kontext die Vermeidung impliziter Typkonvertierungen gemeint. Bei dem Gegenstück, der schwachen Typisierung, können zum Beispiel Zeichenkettenwerte, bei der Verwendung in einem Kontext, welcher einen numerischen Wert erfordert, automatisch konvertiert werden. Dies kann stellenweise zu unbeabsichtigtem Verhalten führen und stellt daher eine Fehlerquelle dar. Durch einfache explizite Konvertierungsmechanismen können jedoch Vorteile der schwachen Typisierung relativ gut ausgeglichen werden, sodass die starkte Typisierung der schwachen in der zu entwickelnden Sprache vorgezogen wird.

3.3.3 Duck Typing

Bei Duck Typing handelt es sich um eine Form der dynamischen Typisierung bei objektorientierten Programmiersprachen. Hierbei wird die Gültigkeit der Verwendung von Objekten in einem bestimmten Kontext durch das Vorhandensein der jeweils benutzten Eigenschaften und Methoden bestimmt, nicht durch dessen Ableitung von einer bestimmten Klasse oder Implementierung einer bestimmten Schnittstelle.

Auch dieses Konzept führt zu einer hohen Flexibilität wie sie in den geplanten Einsatzgebieten der zu entwickelnden Sprache wünschenswert ist.

3.4 Gültigkeitsbereiche und Bindung

Ein zentraler Aspekt jeder Programmiersprache ist die Möglichkeit, Bezeichner, also Namen, zu Datenobjekten zuzuordnen (vgl. [AS96]). Die Art und Weise wie diese Zuordnung vorgenommen wird, stellt eine weitere grundlegende Eigenschaft von Programmiersprachen dar. Dieser Abschnitt widmet sich den relevanten Hintergründen und betrachtet die verschiedenen Möglichkeiten der Behandlung von Namen.

Gültigkeitsbereiche

Namen können, zumindest in imperativen Programmiersprachen, auf zwei verschiedene Arten verwendet werden – als Referenz und als Teil einer Deklaration.

Eine Referenz stellt dabei die Verwendung eines Namens als Repräsentation des zugeordneten Datenobjektes dar. Eine Deklaration wiederum stellt die Zuordnung von Informationen zu einem Namen dar. Diese Zuordnung erfolgt zur Übersetzungszeit. Man spricht dabei davon, daß ein Name von einer Deklaration gebunden wird (siehe [FW08]).

In den meisten Programmiersprachen ist die Verwendung desselben Namens in unterschiedlichen Deklarationen möglich. In diesem Zusammenhang legen Gültigkeitsbereichsregeln fest, welche Deklaration zu einer bestimmten Referenz eines Namens zugeordnet ist. Ein Gültigkeitsbereich einer Deklaration stellt dabei den Teil eines Programmes dar, auf den diese angewandt wird (vgl. [AEU99]).

Diese Zuordnung erfolgt zur Übersetzungszeit unter Verwendung von Symboltabellen. Diese werden in Abschnitt 5.2.4 genauer behandelt.

Zur Laufzeit kann ein Name, der von einer bestimmten Deklaration gebunden ist, auf verschiedene Datenobjekte verweisen. Dies ist zum Beispiel in Sprachen der Fall, die rekursive Funktionen unterstützen und somit auch in der Zielsprache. Listing 3.4 zeigt ein Beispiel einer rekursive Funktion.

Listing 3.4: Verschiedene Bindungen einer Deklaration durch rekursive Funktionsaufrufe

```
1  -- Eine rekursive Funktion, welche einen Artikel zum Beispiel aus
2  -- einer Datenbank laedt. Ein Artikel kann dabei aus mehreren
3  -- Teilen bestehen, die ebenfalls Artikel sind.
4  fun loadProduct(productID) {
5      var product := ... Artikeldaten laden ...;
6      var partIDs := ... Referenzierte Teile laden ...;
7
8      for (partID : partIDs) {
9          -- Rekursiver Aufruf von loadProduct(),
10         -- um referenzierte Teile als Artikel zu laden
11         var part := loadProduct(partID);
12         product.add(part);
13     }
14
15     return product;
16 }
```

Hieran wird deutlich, daß zum Beispiel der deklarierte Name *product* zur Laufzeit gleichzeitig auf verschiedene Datenobjekte verweisen kann, da durch die Rekursion mehrere Aktivierungen der Funktion *loadProduct* zur selben Zeit existieren können.

Die Abbildung von Namen auf ein Datenobjekt wird über eine Funktion beschrieben, die im Allgemeinen als Umgebung bezeichnet wird (vgl. [AEU99]). Die Zuordnung eines Datenobjektes zu einem bestimmten Namen wird wiederum als Bindung bezeichnet. Diese kann als dynamisches Gegenstück zur Deklaration gesehen werden.

Wie in [AEU99] beschrieben, werden Namen als Referenzen in zwei Kategorien eingeteilt. Namen, die im Gültigkeitsbereich einer Funktion vorkommen werden lokal zu dieser Funktion genannt, wenn die zugehörige Deklaration ebenfalls in dieser Funktion vorkommt. Findet sich die Deklaration außerhalb des Gültigkeitsbereiches dieser Funktion, wird von nicht-lokalen Namen gesprochen.

Listing 3.5 zeigt die Verwendung nicht-lokaler Namen. Die Namen *srcStmt* sowie

Listing 3.5: Verwendung nicht-lokaler Namen

```
1  -- Misst die Ausfuehrungszeit einer uebergenen Funktion.
2  fun stopTime(action) {
3      ... Uebergebene Funktion ausfuehren,
4          Zeit messen und zurueckgeben ...
5  }
6
7  -- Fuehrt einen Datentransfer zwischen zwei SQL-Befehlen aus
8  -- und misst die Ausfuehrungszeit mithilfe der Funktion stopTime.
9  fun timedTransfer(srcStmt, dstStmt) {
10     fun concreteTransfer() {
11         srcStmt | dstStmt;
12     }
13
14     return stopTime(concreteTransfer);
15 }
```

dstStmt sind dabei lokal zur Funktion *timedTransfer*, aber nicht-lokal zur Funktion *concreteTransfer*, da sie außerhalb von deren Gültigkeitsbereich deklariert wurden.

Bindung

Die Behandlung von Referenzen auf nicht-lokale Namen wird durch die Bindungsregeln einer Sprache festgelegt. Diese führen nun zu einem weiteren Kriterium anhand dessen Programmiersprachen klassifiziert werden. So wird zwischen der statischen bzw. lexikalischen Bindungsregel und der dynamischen Bindungsregel unterschieden.

Bei der statischen Bindungsregel wird die zu einem Namen zugehörige Deklaration einzig durch die lexikalische Struktur eines Programmes bestimmt. Dadurch ist es möglich Aussagen über Bindungen nur anhand des Programmtextes zu treffen.

Hingegen wird bei der dynamischen Bindungsregel die zu einem Namen zugehörige Deklaration anhand der jeweils zur Laufzeit aktuell vorhandenen Aktivierungen, also Funktionsaufrufe, bestimmt. Eine konkrete Aussage über Bindungen zu treffen kann dabei unter Umständen äußerst schwierig sein, da hierfür alle möglichen relevanten Ausführungspfade in Betracht gezogen werden müssen.

Listing 3.6 variiert das in Listing 3.4 gegebene Beispiel so, daß die Ausführung zu unterschiedlichen Ergebnissen führt, je nachdem, ob statische oder dynamische Bindungsregeln verwendet wird. Unter Verwendung dynamischer Bindung würde das Ergebnis dem des vorherigen Beispiels entsprechen, wohingegen bei statischer Bindung eine Endlosrekursion auftreten würde.

Die durch diese Bindungsregeln jeweils realisierte Semantik wird in Bezug zu den Gültigkeitsbereichen als statischer respektive als dynamischer Gültigkeitsbereich bezeichnet.

Da die Fähigkeit, Aussagen über Bindungen nur aufgrund der lexikalischen Struktur eines Programmes treffen zu können bei der Analyse und insbesondere der Fehleranalyse

Listing 3.6: Statische versus dynamische Bindung

```
1  var productID := 1;
2
3  fun loadProduct() {
4      println("Lade Artikel " + productID);
5
6      -- ... Artikel und Teile laden ...
7
8      for (partID : partIDs) {
9          var productID := partID;
10         loadProduct();
11     }
12 }
```

sehr von Vorteil ist, wird für die zu entwickelnde Sprache die Verwendung von statischen Gültigkeitsbereichen vorgezogen.

Im folgenden Abschnitt werden mögliche Techniken vorgestellt, mit welchen Umgebungen realisiert werden können, die der statischen Bindungsregel folgen und somit statische Gültigkeitsbereiche implementieren.

3.5 Umgebungen

Auch für das im vorigen Abschnitt vorgestellte Konzept der Umgebungen existieren diverse Möglichkeiten der Implementierung. Diese werden im Folgenden genauer betrachtet.

Umgebungen von Blöcken

Die einfachsten Manifestationen von Gültigkeitsbereichen in Programmiersprachen stellen Blöcke dar. Diese können in der Regel ineinander verschachtelt werden. Dadurch kann es vorkommen, daß eine Deklaration in einem Block B_1 sich auf denselben Namen bezieht wie eine Deklaration in einem umschließenden Block B_0. Wird dieser Name in einem weiteren Block B_2 referenziert, der in Block B_1 geschachtelt ist, kommt für Auffinden der zur Referenz gehörigen Deklaration eine Regel zum Tragen, die in [AEU99] als *am engsten umgebende Deklaration*-Regel genannt wird. Diese besagt sinngemäß, dass die zu einem Namen gehörende Deklaration sich im selben Block wie der Name befindet, falls eine solche existiert. Ansonsten befindet sich diese im engsten umschließenden Block des Namens, der eine entsprechende Deklaration enthält. Entsprechend kann es vorkommen, daß die Deklaration eines inneren Blockes die Deklaration eines äusseren Blockes versteckt. Listing 3.7 zeigt ein Beispiel. Die Deklaration des Namens *product* in Zeile 4 versteckt dabei die Deklaration desselben Namens in Zeile 1.

Da Blöcke ausschließlich an der Stelle von deren Deklaration aktiviert werden, können deren Umgebungen über einen einfachen Kellerspeicher (vgl. [AEU99]) oder, noch einfa-

Listing 3.7: Versteckte Deklarationen

```
1 var product := null;
2
3 fun loadProduct(productID) {
4     var product := ... Artikeldaten laden ...;
5     ...
6     return product;
7 }
8
9 .product = loadProduct(1);
```

cher, als Listen von Namen (vgl. [FW08] realisiert werden. Referenzen von Namen auf Deklarationen eines umschließenden Blockes werden dabei wie lokale Namen behandelt.

Umgebungen von Funktionen

Anders als mit Blöcken verhält es sich mit Funktionen. Da deren Aktivierung unabhängig von deren Deklaration erfolgen kann, müssen nicht-lokale Namen in deren Umgebung gesondert behandelt werden.

Für die Realisierung solcher Umgebungen gibt es unterschiedliche Ansätze. Da die Zielsprache auch Eigenschaften der funktionalen Programmierung (siehe Abschnitt 3.1.5) enthalten soll, wird diese unter anderem auch verschachtelte Funktionen unterstützen.

Daher müssen die in Frage kommenden Ansätze naturgemäß die Möglichkeit bieten, auf die Umgebung einer umschließenden Funktion zugreifen zu können. Eine Variante stellt das Hinzufügen einer entsprechenden Referenz zu jeder Umgebung dar. In [AEU99] wird in diesem Zusammenhang auch von Zugriffsverweisen gesprochen. Auch hier existieren unterschiedliche Möglichkeiten der konkreten Implementierung von Umgebungen. Einige Sprachen, wie zum Beispiel *JavaScript* oder *IO* verfolgen hier einen objektorientierten Ansatz, in dem Umgebungen durch normale Objekte der Sprache repräsentiert werden. Das Auffinden von Bindungen nicht-lokaler Namen geschieht dabei implizit über die Semantik die die jeweilige Sprache in Bezug zu Objekten definiert. Dieser Ansatz erscheint sehr elegant, dessen effiziente Implementierung stellt jedoch eine besondere Herausforderung dar. Für die Zielsprache empfiehlt sich dagegen ein Ansatz, der an sich effizient und gleichzeitig einfach zu implementieren ist. Diese Kriterien erfüllt zum Beispiel die *lexikalische Adressierung* (vgl. [AS96], [AEU99]), die im folgenden Abschnitt erläutert wird.

Lexikalische Adressierung

Bei der lexikalischen Adressierung werden Umgebungen durch Listen repräsentiert. Diese enthalten den Zugriffsverweis und die Bindungen der jeweiligen Umgebung. Namen werden dabei während Übersetzung durch lexikalische Adressen – zuerst von *de Brujin* in

[dB72] beschrieben und nach ihm auch *de Bruijn Indizes* genannt – ersetzt. Diese werden wiederum durch zwei numerische Indizes repräsentiert. Der erste Index gibt dabei die Umgebung an, der die Bindung eines Namens enthält, wobei der Index 0 der aktuellen Umgebung entspricht, der Index 1 der Umgebung des umschließenden Blockes usw. Der zweite Index entspricht der Position der Bindung in der entsprechenden Umgebung. Dabei entspricht der Index 0 der ersten Bindung in dieser Umgebung, der Index 1 der zweiten usw. Listing 3.8 zeigt das bereits in Listing 3.5 gegebene Beispiel, in welchem jeder Name mit der entsprechenden lexikalischen Adresse versehen wurde. Diese ist jeweils in spitzen Klammern hinter dem jeweiligen Namen notiert.

Listing 3.8: Lexikalische Adressen

```
1  fun stopTime<0,0>(action<0,0>) {
2      ...
3  }
4
5  fun timedTransfer<0,1>(srcStmt<0,0>, dstStmt<0,1>) {
6      fun concreteTransfer<0,2>() {
7          srcStmt<1,0> | dstStmt<1,1>;
8      }
9
10     return stopTime<1,0>(concreteTransfer<0,2>);
11 }
12
13 var srcStmt<0,2> := ...;
14 var dstStmt<0,3> := ...;
15
16 .timedTransfer<0,1>(srcStmt<0,2>, dstStmt<0,3>);
```

Die Eingangs behandelten Blöcke können auch als vereinfachte Form von Funktionen betrachtet werden. Die Vereinfachungen sind lediglich, daß diese keine Parameter enthalten können und daß wie bereits erwähnt deren Aktivierung nur an der Stelle von dessen Deklaration erfolgen kann. Daher ist es auch möglich, für deren Umgebungen die gleiche Implementierungstechnik wie für die von Funktionen zu verwenden. Der Einfachheit halber soll zunächst dieser Ansatz bei der Implementierung der Zielsprache verfolgt werden.

3.6 Speicherverwaltung

Für die Speicherverwaltung existieren verschiedene, unter anderem in [AEU99] beschriebene, Techniken die im Folgenden eruiert werden sollen.

Eine Technik ist die *Statische Speicherplatzzuweisung*. Dabei erfolgt die Bindung von Namen an einen Speicherplatz zur Übersetzungszeit. Diese Technik ist für die Zielsprache jedoch nicht geeignet, da hierbei zum Beispiel rekursive Funktionsaufrufe nicht erlaubt sind. Der Grund hierfür ist, daß bei dieser Technik eine bestimmte Bindung immer auf

denselben Speicherplatz verweist. Desweiteren erlaubt diese Technik keine dynamische Erzeugung von Datenstrukturen zur Laufzeit.

Eine weitere Technik ist die *Speicherplatzzuweisung im Keller*. Hier werden Aktivierungssegmente von Funktionen – also Speicherblöcke die bei der Ausführung von Funktionen benötigt werden, um damit verbundene Informationen zu speichern – in einem Kellerspeicher verwaltet. Die lokalen Variablen einer Funktion befinden sich dabei innerhalb von deren Aktivierungssegment. Diese Technik ist nicht geeignet, wenn Closures (siehe Abschnitt 3.7.3) unterstützt werden, da hier die lokalen Variablen einer Funktion nach Ende von deren Aktivierung gelöscht werden.

Alternativ bietet sich die *Speicherplatzzuweisung in dynamischem Speicher* an, wobei dynamischer Speicher einen Speicherbereich bezeichnet aus dem Teilbereiche angefordert und diese, im Unterschied zum Kellerspeicher, in beliebiger Reihenfolge wieder freigegeben werden können. Lokale Daten von Funktionen werden dabei im dynamischen Speicher abgelegt, sodaß diese auch nach Ende der Aktivierung einer Funktion erhalten bleiben können. Closures können daher bei dieser Strategie unterstützt werden. Deren Unterstützung macht auch die automatische Freigabe von nicht mehr verwendetem Speicher notwendig. Der Vorgang der automatischen Freigabe wird dabei gemeinhin als *Garbage Collection* bezeichnet.

Auch für die Realisierung der Garbage Collection gibt es verschiedene Techniken. Sofern jedoch das Hostsystem der Zielsprache bereits Unterstützung für Garbage Collection bietet kann auch diese verwendet werden, statt den Mechanismus selbst zu implementieren. Hierbei muß lediglich darauf geachtet werden, daß nicht mehr verwendete Umgebungen nicht länger referenziert werden. Entsprechend muß die Referenz auf die Umgebung einer Funktion aus deren Aktivierungssegment entfernt werden nachdem deren Aktivierung beendet wurde.

3.7 Funktionen

In den für die Zielsprache favorisierten Paradigmen der strukturierten und der funktionalen Programmierung sind Funktionen bzw. Prozeduren oder allgemein Unterprogramme ein zentrales Konzept. Im Folgenden werden verschiedene Aspekte dieses Konzeptes erörtert.

3.7.1 Parameterübergabe

Einer der wichtigsten Aspekte der Implementierung von Funktionen ist die Art der Parameterübergabe bzw. die Evaluierungsstrategie. Die wichtigsten Techniken, beschrieben unter anderem in [FW08], werden im Folgenden vorgestellt.

Call-by-value Bei der *Call-by-value*-Strategie werden die Ausdrücke, welche die Parameter repräsentieren, vor dem Aufruf der Funktion evaluiert und schließlich die Werte

der Ausdrücke an die, den Parametern der Funktion entsprechenden, Variablen gebunden. Meist wird dabei eine Kopie des Wertes erstellt. Eine Änderung des Wertes innerhalb der Funktion hat dabei keine Wirkung außerhalb der Funktion.

Call-by-reference Im Unterschied zu *Call-by-value* wird bei der *Call-by-reference*-Strategie von übergebenen Werten keine Kopie erstellt. Stattdessen wird eine Referenz auf den Wert übergeben. Dadurch ist es möglich, daß Zuweisungen an die entsprechenden Variablen auch außerhalb der jeweiligen Funktion eine Wirkung haben.

Call-by-name Diese Strategie stellt eine Form der *Lazy Evaluation* dar. Die Argumente einer Funktion werden dabei vor deren Aufruf gar nicht ausgewertet. Stattdessen erfolgt die Auswertung erst zu dem Zeitpunkt zu dem das jeweilige Argument verwendet wird.

Call-by-need Die *Call-by-need*-Strategie (vgl. [AFM+95], [MOW98]) stellt eine Verfeinerung der *Call-by-name*-Strategie dar. Bei dieser werden die als Argumente übergebenen Ausdrücke nicht bei jeder Verwendung neu ausgewertet. Einmal berechnete Werte werden stattdessen gespeichert und wiederverwendet, hier wird auch der Begriff *Memoization* benutzt.

Da die Zielsprache, wie bereits ausgeführt, den Ansatz *"Alles ist ein Objekt"* (siehe Abschnitt 3.2.5) verfolgen soll, sollte die Parameterübergabe per *Call-by-value* ausreichend sein, da bei der Übergabe von Objekten jeweils nur eine Referenz auf das entsprechende Objekt übergeben wird, sodaß eine Manipulation auch außerhalb einer Funktion einen Effekt hat. Damit lassen sich die Eigenschaften von *Call-by-reference*-Aufrufen auch in einer *Call-by-value*-Semantik modellieren. Da die *Call-by-reference*-Semantik im Vergleich zu *Call-by-value* eher selten benötigt wird, wird der nötige Umweg bei der Modellierung vertretbar sein. Gestützt wird diese Vermutung dadurch, daß gängige Sprache wie zum Beispiel *Java* oder *Python* ebenfalls ausschließlich die *Call-by-value*-Semantik implementieren.

Sowohl *Call-by-name* als auch *Call-by-need* stellen interessante Möglichkeiten dar. Beide haben jedoch im Kontext der Zielsprache gewisse Nachteile. So ist *Call-by-name* im Vergleich zu *Call-by-value* recht langsam, da Parameter einer Funktion einer Funktion bei jeder Verwendung evaluiert werden müssen. *Call-by-need* weist diesen Nachteil zwar nicht auf, jedoch haben beide Varianten den Nachteil, daß die Ausführungsreihenfolge von Ausdrücken nur sehr schwierig zu bestimmen ist. Das Vorhandensein von Nebenwirkungen wie in der Zielsprache der Fall kann dadurch sehr leicht zu unvorhergesehenem Verhalten führen. Die Verwendung von *Call-by-name* und *Call-by-need* ist daher letztlich nur in Sprachen sinnvoll, die frei von Nebenwirkungen sind.

3.7.2 Funktionen höherer Ordnung

Als Funktionen höherer Ordnung werden Funktionen bezeichnet, welche eine oder mehrere Funktionen als Argumente entgegennehmen können bzw. eine Funktion als Ergebnis liefern. Für rein funktionale Programmiersprachen ist dieses Konzept essentiell. Es stellt jedoch auch in Programmiersprachen, welche anderen Paradigmen folgen, eine äußerst nützliche Komponente dar (vgl. z.B. [Bak93]). Außerdem erfordert der in der Zielsprache verfolgte Ansatz *"Alles ist ein Objekt"* (siehe Abschnitt 3.2.5) die Unterstützung von Funktionen höherer Ordnung.

Listing 3.9: Iteration mit Funktionen höherer Ordnung

```
1  -- Definition der Iterationsfunktion. Der Parameter callback ist
2  -- eine Funktion, die fuer jeden Datensatz der Ergebnismenge des
3  -- SQL-Befehls aufgerufen wird.
4  .SQLStatement.each = fun (callback) {
5      while (this.hasMoreData()) {
6          callback(this.fetchData());
7      }
8  };
9
10 -- Verwendung der Iterationsfunktion mit Uebergabe einer anonymen
11 -- Funktion als Parameter.
12 .SQLStatement.| = fun (aStatement) {
13     this.each(fun (data) {
14         aStatement.executeWithData(data);
15     });
16 };
```

Listing 3.9 zeigt die Verwendung einer Funktion höherer Ordnung. Die in Listing 3.3 benutzte imperative Variante der Iteration über die Ergebnismenge eines SQL-Befehls wurde dabei durch die Iteration mithilfe einer Funktion höherer Ordnung ersetzt.

3.7.3 Closures

In einem System, welches Funktionen höherer Ordnung unterstützt, können diese verschachtelt auftreten. Dabei kann eine Funktion Variablen referenzieren, welche in einer umschließenden Funktion deklariert worden sind (siehe auch Abschnitt 3.4). Häufig wird hierbei von von *freien Variablen* gesprochen. Da Funktionen höherer Ordnung als Argumente oder Rückgabewerte von Funktionen dienen können, ist es möglich, daß eine innere Funktion f eine umschließende Funktion g verläßt. Der Aufruf von f kann dabei zu einem Zeitpunkt auftreten, zu welchem die Umgebung von g nicht mehr existiert. Die Referenzierung von freien Variablen innerhalb von f wäre also ungültig. Listing 3.10 zeigt ein Beispiel der Verwendung freier Variablen. Die Variable *productID* befindet sich dabei in der Umgebung der Funktion *createPartsLoader*, deren Aktivierung jedoch zum Zeitpunkt der Ausführung erzeugten Funktion bereits beendet ist.

Listing 3.10: Verwendung freier Variablen

```
1  -- Erzeugt eine Funktion, welche die Teile des
2  -- uebergebenen Artikels laedt.
3  fun createPartsLoader(productID) {
4      return fun () {
5          println("Loading parts of product " + productID);
6          var parts := ... Assoziierte Teile laden ...;
7          return parts;
8      };
9  }
10
11 -- Ladefunktion fue Artikel 1 erzeugen.
12 var loadParts := createPartsLoader(1);
13
14 -- Teile laden.
15 var parts := loadParts();
```

Dieses Problem wird üblicherweise durch die Verwendung von *Closures* adressiert (vgl. [FW08]). Es handelt sich dabei um Strukturen, welche zum einen eine Referenz auf eine Funktion und zum anderen die freien Variablen dieser Funktion enthalten.

Eine einfache Implementierung von Closures kann so aussehen, daß bei jeder Deklaration einer Funktion eine entsprechende Closure erzeugt wird, die neben der Funktionsreferenz eine Kopie der aktuellen Umgebung enthält. Diese Closure kann anschließend in der Umgebung registriert werden. Es gibt jedoch diverse Möglichkeiten diese Implementierung effizienter zu gestalten. So kann wie bereits erwähnt auf die Erzeugung von Closures gänzlich verzichtet werden, wenn die deklarierte Funktion die sie umschließende Funktion, falls vorhanden, nicht verlässt. Ebenso kann auf die Erzeugung verzichtet werden, wenn die Funktion keine freien Variablen enthält. Desweiteren muß die in der Closure gesicherte Umgebung nur die freien Variablen der zugehörigen Funktion enthalten. Weitere Optimierungsmöglichkeiten beschreibt [App07]. Diese sollen jedoch vorerst außer Acht gelassen werden.

3.8 Interpretationstechniken

In diesem Abschnitt werden ausgewählte Ansätze der Implementierung von Interpretern für Programmiersprachen dargestellt sowie deren Vor- und Nachteile untersucht. Schließlich werden die untersuchten Ansätze nach hier relevanten Kriterien bewertet.

3.8.1 Interpreter-Visitor-Pattern

In [GHJV95] werden zwei Entwurfsmuster, das Interpretermuster sowie das Besuchermuster, beschrieben. Das Interpretermuster stellt einen generischen Ansatz für die Implementierung von Interpretern dar. Dabei wird das zu interpretierende Programm durch

einen abstrakten Syntaxbaum dargestellt. Oft wird dieses Entwurfsmuster in Kombination mit dem Besuchermuster verwendet, in dem ein Besucher den Baum durchläuft und die entsprechende Aktion jedes besuchten Knotens ausführt. Der Kontrollfluss wird bei dieser Methode über Verzweigungen und Exceptions realisiert.

Vorteilhaft an dieser Methode ist die einfache Implementierung. Von Nachteil ist, daß hierbei die explizite Steuerung des Kontrollflusses schwierig beziehungsweise unübersichtlich ist.

3.8.2 Continuation-Passing-Interpreter

Der Continuation-Passing-Style (CPS) ist eine spezielle Notationsform für Programme, die unter anderem in [App07] beschrieben wird. In [FW08] wird dieser Stil für die Implementierung eines Interpreters verwendet. Dabei wird das zu interpretierende Programm als Folge von endrekursiven Funktionen dargestellt. Parameter dieser Funktionen sind jeweils ein Wert, mit welchem sie arbeiten sowie eine weitere Funktion, mit welcher die Berechnung fortgesetzt werden soll, die also den Kontrollkontext darstellt (Abschnitt 3.9 geht genauer hierauf ein). Der Kontrollfluss wird bei dieser Methode durch Verzweigung und direkte Manipulation des Kontrollkontexts realisiert.

Von Vorteil ist hier, daß auch diese Methode recht einfach zu implementieren ist. Außerdem ist der Kontrollfluss hier gut steuerbar. Zusätzlich ist auch die Tail-Call-Optimierung für interpretierte Programme leicht zu implementieren.

3.8.3 Stapelmaschine

Eine gängige Methode, Interpreter zu implementieren ist, diese als virtuelle Stapelmaschine zu implementieren. So nutzen mit der *Java Virtual Machine* (JVM) und der *.NET Virtual Machine* zwei sehr weit verbreitete Systeme diesen Ansatz (vgl. [Gou01]). Das zu interpretierende Programm wird dabei als Folge von Bytecodes repräsentiert, die den von der Maschine auszuführenden Befehlen entsprechen. Diese Befehle arbeiten auf einem oder mehreren Stapeln von Daten und Befehlen. Der Kontrollfluss wird hierbei durch Sprungbefehle realisiert (vgl. [Koo89]).

Vorteilhaft an dieser Methode ist, daß die Bytecode-Repräsentation eines Programmes gespeichert und zu einem späteren Zeitpunkt erneut aufgerufen werden kann, sodaß der Übersetzungsvorgang bei erneuter Ausführung ausbleiben kann. Desweiteren hat dieser Ansatz meist Geschwindigkeitsvorteile gegenüber den oben beschriebenen Methoden. Nachteilig ist jedoch die komplexere Implementierung durch eine höhere Abstraktion der Ausführungseinheit von der Eingabesprache und die zusätzlich notwendige Code-Generierung.

3.8.4 Registermaschine

Auch bei dieser Methode werden Programme als Folge von Bytecodes repräsentiert. Anstelle von Stapeln arbeiten die davon dargestellten Befehle jedoch auf einer definierten (maschinenspezifischen) Menge von Speicherplätzen, den Registern. Der Kontrollfluss erfolgt auch hier durch Sprungbefehle.

Vorteile dieser Methode sind die schnellere Ausführung von Programmen im Vergleich zur Stapelmaschine (vgl. [SGBE05]) durch die geringere Anzahl an für ein beliebiges Programm auszuführenden Maschinenbefehlen. Nachteilig ist, daß sich die Implementierung ein Wenig schwieriger gestaltet als bei Stapelmaschinen. Außerdem ist der für Registermaschinen generierte Code größer als der für Stapelmaschinen generierte (vgl. [DBGW03]).

3.8.5 Code-Generierung

Eine gänzlich andere Variante ist die, den Interpreter nicht selbst zu implementieren, sondern Code für einen bestehenden Interpreter bzw. eine bestehende Maschine zu generieren. Ziele der Code-Generierung könnten reale Maschinen sein, virtuelle Maschinen wie zum Beispiel die *Java Virtual Machine* oder andere Programmiersprachen.

Je nach Ziel der Code-Generierung hat diese Methode Vor- und Nachteile. Zu den Vorteile kann gehören, daß die Funktionalität bestehender Maschinen nutzbar ist und entsprechend keine eigene Implementierung notwendig ist. Zu den Nachteilen kann zählen, daß eine Beschränkung auf die Funktionalität der Ziel-Maschine besteht.

3.9 Continuation-Passing-Interpreter

Da die im vorigen Abschnitt vorgestellte Technik des Continuation-Passing-Interpreters, wie bereits erwähnt, mit einfacher Implementierung und recht hoher Flexibilität besticht, soll für die Maschine der Zielsprache zunächst diese Technik verfolgt werden.

Die folgenden Abschnitte gehen daher genauer auf diese Technik und damit verbundene Konzepte ein.

3.9.1 Continuation-Passing-Style

Der Continuation-Passing-Style (CPS) ist eine Notationsform in der Programme dargestellt werden können und ist eng mit dem λ-Kalklül verwandt. Im Continuation-Passing-Style erhält jede Funktion einen zusätzlichen Parameter, die Continuation. Diese beschreibt den Kontrollkontext in dem eine Funktion ausgeführt wird, also die Berechnungen, die nach dieser ausgeführt werden sollen. Eine Funktion im CPS gibt niemals einen berechneten Wert direkt zurück, es sei denn des handelt sich um die letzte Continuation in einer Befehlsfolge. Stattdessen übergibt sie diesen an die Continuation, welche genauso

fortfährt. Die CPS-Notation ist damit eine endrekursive Darstellung von Programmen. Die Konvertierung von Programmen in den CPS kann automatisch erfolgen. Wegen dieser Eigenschaft und wegen der klaren Eigenschaften, welche sich aus der direkten Verwandschaft mit dem λ-Kalkül ergeben, wird der CPS zumeist als Zwischencode bei der Übersetzung von Programmen verwendet. Ebenso findet dieser Stil aber auch Anwendung zum Beispiel bei der User-Interface-Entwicklung sowie bei der Web-Programmierung.

Listing 3.11 zeigt beispielhaft eine Implementierung der Fibonacci-Funktion im Continuation-Passing-Style (nach [FW08]).

Listing 3.11: Fibonacci-Funktion im Continuation-Passing-Style

```
1  function fib(n) {
2     cont = function (val) {
3         val
4     }
5
6     fibk(n, cont)
7  }
8
9  function fibk(n, cont) {
10    if (n < 2) {
11        cont(1)
12    }
13    else {
14        fibk(n - 1, function (val1) {
15            fibk(n - 2, function (val2) {
16                cont(val1 + val2);
17            }
18        }
19    }
20 }
```

3.9.2 Interpretation im Continuation-Passing-Style

Der Continuation-Passing-Interpreter ist ein Interpreter im Continuation-Passing-Style. Dieser besteht hauptsächlich aus zwei Funktionen. Eine Funktion, *eval*, evaluiert Ausdrücke. Eine weitere Funktion, *apply*, führt Continuations aus.

Die Funktion *eval* hat dabei folgende Signatur:

$$\texttt{eval} : Exp \times Cont \rightarrow Result$$

Der Pseudocode zu dieser Funktion lautet wie folgt:

```
1  function EVAL(exp, cont)
2      if exp is a simple expression then
3          value ← GETVALUEOF(exp)
4          return APPLY(cont, value)
5      else if exp contains sub-expressions then
6          if sub-expression's value should become main expression's value then
7              subExpression ← GETSUBEXPRESSION(exp)
8              return EVAL(subExpression, cont)
9          else
10             subExpression ← GETSUBEXPRESSION(exp)
11             extendedCont ← CREATECONTINUATION(subExpression, cont)
12             return EVAL(subExpression, extendedCont)
13         end if
14     end if
15 end function
```

Hierbei erzeugt die Funktion *createContinuation* eine dem Typ des übergebenen Ausdrucks entsprechende Continuation-Funktion. Je nach Art des Ausdrucks kann diese Funktion eine der folgenden Aktionen ausführen:

- einen wiederum enthaltenen Teilausdruck mittels *eval* auswerten
- sofern der übergebene Ausdruck keine Teilausdrücke enthält, diesen auswerten und den berechneted Wert mittels *apply* an die nächste Continuation übergeben
- wenn die Berechnung beendet ist, also keine weiteren Continuations vorhanden sind, die übergebenen Wert zurückgeben

Der final von der Continuation-Funktion zurückgegebene Wert stellt damit das Ergebnis des gesamten Programmes dar.

Die Funktion *apply* hat die folgende Signatur:

$$\texttt{apply} : Cont \times Value \rightarrow Result$$

Der Pseudocode für diese Funktion sieht wiederum wie folgt aus:

```
1  function APPLY(cont, value)
2      return CALLCONTINUATION(cont, value)
3  end function
```

Die Funktion *callContinuation* ruft hierbei nur die übergebene Continuation-Funktion mit dem ebenfalls übergebenen Wert auf.

Die Code-Beispiele verdeutlichen zwei wichtige Aspekte:

- In dieser Form wird das gesamte auszuführende Programm als Folge von endrekursiven Funktionsaufrufen dargestellt.
- Nimmt ein Ausdruck den Wert eines Teilausdrucks an, werden beide im gleichen Kontrollkontext evaluiert. Dadurch wird erreicht, daß zum Beispiel endrekursive Funktionsaufrufe im interpretierten Programm den Kontrollkontext nicht vergrößern. Diest stellt eine Form der Tail-Call-Optimierung dar.

Die Tatsache, daß das intepretierte Programm als Folge von endrekursiven Funktionsaufrufen dargestellt wird, hat für die Implementierung des Interpreters Konsequenzen. Unterstützt die Sprache in welcher dieser implementiert ist keine automatische Tail-Call-Optimierung muss diese vom Interpreter selbst vorgenommen werden. Die Tail-Call-Optimierung sorgt dafür, daß für endrekursive Funktionsaufrufe kein separates Aktivierungssegment angelegt wird. Wie bereits gezeigt, wird beim Continuation-Passing-Interpreter das gesamte interpretierte Programm als Folge von endrekursiven Funktionsaufrufen dargestellt. Das Unterlassen der Tail-Call-Optimierung würde entsprechend bedeuten, daß für jeden interpretierten Ausdruck ein Aktivierungssegment erzeugt würde, welches erst am Ende der gesamten Berechnung wieder entfernt werden würde. Die maximale Größe des für Aktivierungssegmente vorgesehenen Speichers wäre also ein stark limitierender Faktor für die mögliche Anzahl von interpretierten Ausdrücken. Für die manuelle Tail-Call-Optimierung schlägt [FW08] zwei Strategien vor, Trampolining und registerbasierte Verarbeitung, die in den folgenden Abschnitten vorgestellt werden sollen. In [Bak95] schlägt Baker eine weitere Methode vor, die ebenfalls kurz beschrieben wird.

Trampolining

Bei dieser Methode wird eine weitere Funktion, *trampoline*, eingeführt. Der Pseudocode für diese Funktion lautet wie folgt:

```
1 function TRAMPOLINE(bounce)
2     if bounce is a value then
3         return bounce
4     else if bounce is a function then
5         bounceResult ← CALLBOUNCE(bounce)
6         return TRAMPOLINE(bounceResult)
7     end if
8 end function
```

Das Argument *bounce* ist hierbei entweder eine parameterlose Funktion, die Bounce-Funktion, oder ein konkreter Wert.

Neben der Einführung der Trampoline-Funktion werden zusätzliche alle Continuation-Funktionen modifiziert, welche zu einer potentiell unbegrenzten Anzahl von Rekursionen führen könnten. Dies wären zum Beispiel Contiuation-Funktionen, die Funktionsaufrufe

in der interpretierten Sprache realisieren. Statt direkt mit der Berechnung fortzufahren erzeugen diese Continuations eine Bounce-Funktion und geben diese zurück. Dadurch liefern die Funktionen *eval* und *apply* ultimativ entweder ein Ergebnis, also einen konkreten Wert, oder eine Bounce-Funktion.

Eine Funktion welche ein Programm ausführt würde entsprechend wie folgt aussehen:

```
1 function evalProgram(programExpression)
2     valueOrBounce ← eval(programExpression)
3     return trampoline(valueOrBounce)
4 end function
```

In der obigen Form ist die Funktion *trampoline* natürlich noch immer eine endrekursive Funktion, weist also noch kein iteratives Verhalten auf. Dieses kann allerdings durch eine einfache Transformation geändert werden:

```
1 function trampoline(bounce)
2     while bounce is a function do
3         bounce ← callBounce(bounce)
4     end while
5     return bounce
6 end function
```

Damit weist die Funktion ein iteratives Verhalten auf. Der Interpreter führt somit keine unbegrenzten Rekursionen mehr aus.

Ein Nachteil dieser Methode ist, daß das Erzeugen von Bounce-Funktionen oder das Ausführen von Bounce-Funktionen mithilfe der Trampoline-Funktion unter Umständen, d.h. je nach Art der Ausführungsplattform kostspielig sein kann. Es gibt jedoch Ansätze diese Methode zu optimieren.

So wird in [SO01] ein Ansatz beschrieben, bei welchem jede Funktion um ein Argument, den Tail-Call-Counter, erweitert wird. In diesem werden die aktuell aufeinanderfolgend ausgeführten endrekursiven Funktionsaufrufe gezählt. Beim jedem endrekursiven Funktionsaufruf wird der Tail-Call-Counter um eins erhöht und übergeben. Bei nicht endrekursive Funktionsaufrufen wird der Wert 0 als Tail-Call-Counter übergeben. Jede Funktion wird außerdem so erweitert, daß diese zu Beginn den aktuellen Wert des Tail-Call-Counter prüft. Übersteigt dieser eine bestimmte Grenze, wird eine Bounce-Funktion erzeugt und zurückgegeben. Ansonsten wird die Funktion normal ausgeführt. Schließlich wird nach jedem endrekursiven Funktionsaufruf Code eingefügt, welcher prüft, ob der von der aufgerufenen Funktion zurückgegebene Wert eine Bounce-Funktion ist und der Wert des Tail-Call-Counter gleich 0 ist, dies also die oberste Funktion in der Aufrufhierarchie ist.

[SO01] beschreibt diesen Ansatz zwar im Kontext der Code-Generierung für die *Java Virtual Machine*, dieser wäre aber auch einfach auf den Continuation-Passing-Interpreter übertragbar.

Bakers Methode

Ein in [Bak95] beschriebener Ansatz macht sich die Eigenschaften des Contiuation-Passing-Style zu nutze. Ein Eingabe-Programm wird dabei in diesen Stil transformiert, sodaß alle Funktionsaufrufe endrekursiv sind. Während der Ausführung des Programms wird der Stapel, welcher die Aktivierungssegmente der Funktionen enthält, überwacht. Sobald dieser Stapel überläuft werden alle darin enthaltenen, noch in Benutzung befindlichen Daten, vom Stapel in einen Heap-Speicher übertragen. Anschließend wird der Stapel bis auf das initiale Aktivierungssegment geleert.

Wie jedoch in [SO01] gezeigt wird, ist diese Methode auf Maschinen wie der Java Virtual Machine, in welchen der Aufrufstapel nicht direkt manipuliert werden kann, weniger performant als der dort ebenfalls vorgestellte Ansatz zur Optimierung der Trampolining-Methode.

Registerbasierte Verarbeitung

Die Zweite in [FW08] beschriebene Methode, unbegrenzte Rekursion im Interpreter zu vermeiden, ist die Einführung von globalen Registern. Anstelle der Übergabe von Werten an beteiligte Funktionen werden diese in globale Register geschrieben und von dort abgerufen. Hierbei wird die Eigenschaft des Interpreters ausgenutzt, daß alle Funktionsaufrufe endrekursiv sind. Da die Übergabe der Werte nicht mehr über die Argumente der Funktionen *eval* und *apply* realisiert wird, werden diese Aufrufe in parameterlose umgewandelt. Damit sind diese Aufrufe jedoch immer noch endrekursiv. Dies läßt sich aber mit einer einfachen Änderung beheben. Der folgende Pseudocode zeigt diese:

```
 1 exp ← undefined
 2 cont ← undefined
 3 value ← undefined
 4
 5 function evalProgram(programExpression)
 6     exp ← programExpression
 7     next ← eval
 8     while end of program not reached do
 9         if next = eval then
10             EVAL
11         else if next = apply then
12             APPLY
13         end if
14     end while
15     return value
16 end function
```

```
17 function EVAL
18     if exp is a simple expression then
19         value ← GETVALUEOF(exp)
20         return apply
21     else if exp contains sub-expressions then
22         if sub-expression's value should become main expression's value then
23             exp ← GETSUBEXPRESSION(exp)
24             return eval
25         else
26             exp ← GETSUBEXPRESSION(exp)
27             cont ← CREATECONTINUATION(subExpression, cont)
28             return eval
29         end if
30     end if
31 end function
```

Es wird ersichtlich, daß hier eine vereinfachte Form des Trampolining verwendet wird. Diese Vereinfachung ist möglich, da nur zwei parameterlose Funktionen an diesem Mechanismus beteiligt sind und daher keine Bounce-Funktion erzeugt werden muß, um der als Trampoline-Funktion fungierenden Funktion *evalProgram* mitzuteilen, welche Aktion als Nächstes ausgeführt werden soll.

3.10 Zusammenfassung

Für die Entwicklung der Zielsprache wird eine Mischung aus imperativer, objektorientierter und teilweise auch funktionaler Programmierung favorisiert. Für die Objektorientierung soll der Prototyp-basierte Ansatz verfolgt werden. Entsprechend der Konzeption der Zielsprache als Skriptsprache wird diese als dynamisch typisierte Sprache entworfen. Um eine klare Semantik zu erhalten wird zusätzlich der Ansatz der starken Typisierung verfolgt. Auf Namen bzw. Variablen werden in der Zielsprache die gängigen statischen Bindungsregeln angewandt. Für die Realisierung von Umgebungen wird die Verwendung der lexikalischen Adressierung aufgrund von deren relativ einfacher Implementierung bei gleichzeitig recht hoher zu erwartender Geschwindigkeit favorisiert.

Die Implementierung der Maschine soll zunächst nach der Technik des Continuation-Passing-Interpreters erfolgen, da dieser Ansatz sowohl eine einfache Implementierbarkeit als auch eine gute Steuerbarkeit des Kontrollflusses aufweist. Konkret wird hierfür die Implementierung nach der registerbasierten Verarbeitung gewählt, da für diese das von den in Frage kommenden Varianten beste Verhältnis zwischen Ausführungsgeschwindigkeit und Implementierungsaufwand erwartet wird.

4 Konzeption der Sprache

In diesem Kapitel wird die grundlegende Syntax und Semantik der Zielsprache vorgestellt. Dabei werden die in den vorhergehenden Kapiteln erarbeiteten Anforderungen berücksichtigt und bereits evaluierte Konzepte verwendet, die zur Erfüllung dieser Anforderungen notwendig oder besonders geeignet sind. Die Beschreibung der Elemente der Zielsprache beschränkt sich dabei auf ein weitgehend abstraktes Niveau. Eine Konkretisierung erfolgt in der Referenz in Anhang A.

4.1 Grundstruktur der Syntax

Um der Forderung nach gängiger Syntax gerecht zu werden, orientiert sich die Zielsprache an anderen Sprachen, welche eine *C*-ähnliche Syntax verwenden. In diese Klasse fallen, neben der Sprache *C* selbst, *C++*, *Java*, *JavaScript*, *PHP* und andere. Diese Sprachen zählen zum Zeitpunkt des Schreibens der vorliegenden Arbeit zu den populärsten in Benutzung befindlichen Sprachen[10]. Von einer recht weit verbreiteten Vertrautheit mit der grundlegenden Syntax dieser Sprachen kann also ausgegangen werden.

Kennzeichnende Elemente dieser Art Syntax sind unter anderem die Verwendung von geschweiften Klammern, um Blöcke zu formen, die Trennung von Befehlen durch Semikolons und die Unterstützung von Kontrollstrukturen wie `if`-Anweisungen die in allen genannten Sprachen nahezu identisch sind. Die syntaktische Struktur der Zielsprache orientiert sich weitgehend an diesen Eigenschaften.

Die folgenden Abschnitte stellen die wichtigsten verwendeten syntaktischen und semantischen Elemente der Zielsprache genauer vor.

4.2 Ausdrücke

Ausdrücke stellen die Kernelemente der Zielsprache dar. Im Unterschied zu Befehlen (siehe Abschnitt 4.3) liefern diese immer einen Wert. Es werden vier verschiedene Formen von Ausdrücken verwendet:

[10] Gestützt wird diese Aussage zum Beispiel durch die auf `http://www.langpop.com/` veröffentlichten, diesbezüglichen Statistiken. Diese sind zwar nicht nach wissenschaftlichen Kriterien erstellt worden, reichen aber dennoch für eine grobe Einschätzung des Verbreitungsgrades der Sprachen.

- Literale
- Zuweisungen
- Funktions- bzw. Blockdefinitionen
- Funktions- bzw. Methodenaufrufe

Diese werden in späteren Abschnitten jeweils genauer beschrieben.

4.3 Befehle

Um wie gefordert einen imperativen sowie strukturierten Programmierstil (siehe Abschnitt 3.1) zu ermöglichen, werden neben Ausdrücken auch Befehle unterstützt. Dabei wird zwischen zwei Arten von Befehlen unterschieden:

Echte Befehle, diese liefern keinen Wert und haben in der Sprache keine gleichwertiges Gegenstück in Form eines Ausdrucks.

Preudobefehle, welche während der Übersetzung von Programmen der Sprache in eine jeweils semantisch äquivalente Ausdrucksform überführt werden.

Im Folgenden werden diese Befehlsarten genauer vorgestellt.

4.3.1 Echte Befehle

Wie bereits erwähnt liefern echte Befehle keinen Wert, wodurch sie sich grundlegend von Ausdücken unterscheiden. Diese Eigenschaft zieht nach sich, daß Befehle generell nicht Teil eines Ausdrucks sein können. Insbesondere können Befehle nicht in Zuweisungen verwendet werden.

Da jedoch die Zielsprache eher am objektorientierten als am strukturierten Programmierstil orientiert ist, existieren nur wenige echte Befehle. Konkret handelt es sich dabei um Variablendeklarationen und Gültigkeitsbereichsblöcke, also Befehle, welche Teil der eher am imperativen Paradigma ausgerichteten Implementierung der Unterstützung von Variablen sind (siehe dazu Abschnitt 4.5).

4.3.2 Pseudobefehle

Bereits durch die in der Zielsprache verfolgten Grundkonzepte der durchgängigen Objektorientierung sowie der Unterstützung von Closures ist es möglich, alle grundlegenden Kontrollstrukturen zu repräsentieren. Sprachen wie *Smalltalk* oder *Self* unterstützen diese Behauptung. Wie in Abschnitt 3.2.5 angedeutet, wird jedoch durch die Forderung nach gängiger Syntax sowie der Unterstützung imperativer und strukturierter Programmierung die Realisierung von befehlsartigen Kontrollstrukturen notwendig. Um den Kern

der Sprache möglichst einfach zu halten, werden diese Befehle zwar syntaktisch unterstützt, intern jedoch durch Ausdrücke repräsentiert. Dadurch kann deren Unterstützung auf den Parser beschränkt bleiben.

4.4 Literale

Die Zielsprache definiert vier verschiedene Arten von Literalen. Diese werden in den folgenden Abschnitten vorgestellt.

4.4.1 `null`-Literal

Das `null`-Literal repräsentiert den `null`-Wert. Dessen Semantik ist in der Zielsprache *kein Wert* bzw. *undefinierter Wert*.

4.4.2 Boolesche Literale

Die booleschen Literale `true` und `false` stehen für die entsprechenden booleschen Wahrheitswerte. Diese Werte werden in der Zielsprache durch Singleton-Objekte repräsentiert. Von diesen existiert entsprechend im System jeweils nur eine Instanz.

4.4.3 Numerische Literale

Numerische Literale repräsentieren ganzzahlige Werte oder Gleitkommawerte. Diese werden in der Zielsprache durch entsprechende Objekte repräsentiert, welche in Anhang A genauer beschrieben werden. Auf die Semantik numerischer Werte in der Zielsprache geht Abschnitt 5.3.5 genauer ein.

4.4.4 Zeichenkettenliterale

Die Zielsprache definiert zwei unterschiedliche Arten von Zeichenkettenliteralen. Zum einen fixe Literale, welche die ihnen entsprechenden Zeichenketten repräsentieren, zum anderen Literale, welche Variablenreferenzen enthalten können. Deren Wert wird dynamisch zur Laufzeit bestimmt. Syntaktisch unterscheiden sich diese Literale durch den jeweils verwendeten Begrenzer (siehe Anhang A). Listing 4.1 verdeutlicht den Unterschied anhand eines Beispiels.

4.5 Variablen

Als Ausnahme zum sonst durchgängig objektorientierten Ansatz der Zielsprache (siehe folgender Abschnitt), werden Variablen als separates Konzept unterstützt (siehe Abschnitt 3.5). Diese repräsentieren dabei den Speicherplatz für Referenzen auf Werte. Da,

Listing 4.1: Variablenreferenzen in Zeichenkettenliteralen

```
1 var reportsTable := 'stock_reports';
2
3 .println('Dropping table @{reportsTable}...');
4 .println("Dropping table @{reportsTable}...");
5
6 -- Ausgabe:
7 -- Dropping table @{reportsTable}...
8 -- Dropping table stock_reports...
```

wie in Abschnitt 3.3 angedeutet, die Zielsprache dynamische Typisierung verwendet, kann jede Variable eine Referenz auf jeden Wert, egal welchen Typs, annehmen.

Der Wert von Variablen kann durch Zuweisungen geändert werden. Dabei handelt es sich, wie in den meisten Programmiersprachen üblich, um einen Ausdruck, welcher eine linke und eine rechte Seite hat, wobei auf der rechten Seite ein beliebiger Ausdruck, auf der linken Seite eine Referenz auf einen Wert. Dies kann eine Variable sein, aber auch ein anderer Ausdruck (siehe hierzu Anhang A). Der Referenz der linken Seite wird dabei der Wert zugewiesen, den der Ausdruck der rechten Seite liefert.

Variablen müssen vor ihrer Verwendung deklariert werden. Dies kann sowohl explizit (oder statisch) als auch implizit (oder dynamisch) geschehen. Eine explizite Deklaration erfolgt entweder durch einen Deklarationsbefehl oder durch einen kombinierten Deklarations- und Zuweisungsausdruck. Ersteres ist ein echter Befehl, welcher keinen Wert liefert und daher nicht auf der rechten Seite einer Zuweisung verwendet werden kann. Letzters ist ein Ausdruck, sodaß diese Beschränkung hierfür nicht gilt. Listing 4.2 zeigt ein entsprechendes Beispiel.

Implizite Deklarationen erlauben es, Variablen zur Laufzeit zu deklarieren, statt zur Übersetzungszeit, wie es bei der expliziten Deklaration der Fall ist. Diese Funktionalität findet vor allem bei der Integration von Hostsystem und Zielsprache Verwendung. Zu beachten ist hier, daß auch bei impliziten Deklarationen statische Gültigkeitsbereiche verwendet werden. Dies ist also nicht zu verwechseln mit dynamischen Variablen im Kontext von dynamischen Gültigkeitsbereichen. Abschnitt 5.3.7 geht näher auf diese Thematik ein.

4.5.1 Gültigkeitsbereiche von Variablen

Wie in Abschnitt 3.5 angedeutet, werden in der Zielsprache statische Gültigkeitsbereiche verwendet. Diese werden durch Blöcke markiert, die verschachtelt werden können. Diese Blöcke können ein Teil von Pseudobefehlen und Ausdrücken sein, wie zum Beispiel in `if`-Anweisungen oder Funktionsdeklarationen. Zusätzlich definiert die Zielsprache einen weiteren echten Befehl, den Gültigkeitsbereichsblock, welcher ausschließlich dazu dient, einen separaten Gültigkeitsbereich zu definieren. Dies kann zum Beispiel nützlich sein,

Listing 4.2: Variablen - Deklaration und Zuweisung

```
1  fun loadProduct(productID) {
2          -- Deklaration der Variablen product.
3          var product;
4
5          -- Zuweisung eines Wertes an die Variable product.
6          product = ...;
7
8          -- Ebenfalls Zuweisung eines Wertes an die Variable
9          -- product. Verwendung des Schluesselwortes
10         -- var im Skriptbefehlsmodus optional.
11         var product = ...;
12
13         -- Deklaration der Variablen partIDs und gleichzeitige
14         -- Zuweisung eines Wertes.
15         var partIDs := ...;
16
17         -- Deklaration und Zuweisung an die Variable partCount
18         -- in Ausdrucksform.
19         if (partCount := partIDs.length > 0) {
20                 ...;
21         }
22
23         ...;
24 }
```

um zu verhindern, daß lokal benötigte Variablen global sichtbar werden[11]. Listing 4.3 zeigt einen entsprechenden Anwendungsfall.

Variablen sind ab dem Zeitpunkt ihrer Deklaration innerhalb des umschließenden Gültigkeitsbereichs verwendbar. Mehrfache Deklarationen von Variablen innerhalb desselben Gültigkeitsbereiches sind nicht möglich. In einem solchen Fall wird die Deklaration ignoriert und stattdessen die bereits deklarierte Variable verwendet. Diese Einschränkung gilt jedoch nicht für implizit deklarierte Variablen (siehe Abschnitt 5.3.7).

Im Gegensatz dazu kann jedoch eine Variable in einem verschachtelten Gültigkeitsbereich deklariert werden, deren Name einer Variablen entspricht, welche bereits in einem übergeordneten Gültigkeitsbereich deklariert worden ist. Dabei überlagert erstere die bereits deklarierte Variable, sodaß die Variable des übergeordneten Gültigkeitsbereiches nach der erneuten Deklaration nicht mehr erreichbar ist.

4.6 Funktionen und Block-Closures

Die Zielsprache unterstützt, wie in Abschnitt 3.7 formuliert, Funktionen höherer Ordnung sowie Closures. Dabei werden verschiedene Ausprägungen verwendet, die im Folgenden beschrieben werden. Ebenso werden Konzepte erläutert, die allen Varianten gemein sind.

[11] Also um *Namespace Pollution* zu verhindern.

Listing 4.3: Gültigkeitsbereichsblock

```
1  -- Deklaration der Variablen transfer im globalen
2  -- Gueltigkeitsbereich.
3  var transfer;
4
5  -- Oeffnen eines neuen Gueltigkeitsbereiches.
6  {
7          -- Deklarierte Funktion time im globalen
8          -- Gueltigkeitsbereich nicht sichtbar.
9          fun time(action) {
10                 ... Uebergebene Funktion ausfuehren und
11                     Zeit messen ...
12         }
13
14         -- Zuweisung der Transfer-Funktion an die globale Variable
15         -- transfer und damit Export der Funktion in den
16         -- globalen Gueltigkeitsbereich.
17         transfer = fun (srcStmt, dstStmt) {
18                 time(fun () {
19                         ... Eigentlichen Transfer durchfuehren ...
20                 });
21         }
22 }
23
24 -- Aufruf der Transfer-Funktion. Gueltig.
25 .transfer(...);
26
27 -- Aufruf der Funktion time. Ungueltig, da diese
28 -- Funktion im globalen Gueltigkeitsbereich nicht sichtbar ist.
29 .time(...);
```

4.6.1 Funktionen

Funktionen realisieren in der Zielsprache das Konzept der prozeduralen bzw. funktionalen Programmierung (siehe Abschnitt 3.1). Entsprechend enthalten Funktionen eine Menge von Befehlen bzw. Ausdrücken, welche beim Aufruf der jeweiligen Funktion evaluiert werden. Der Rückgabewert von Funktionen entspricht im Normalfall dem Wert des letzten evaluierten Ausdrucks der aufgerufenen Funktion. Zusätzlich unterstützt die Zielsprache einen weiteren echten Befehl, den Rücksprungbefehl. Mit dessen Hilfe kann die Ausführung der, den Rücksprungbefehl umschließenden, Funktion explizit beendet und ein Rückgabewert explizit spezifiziert werden.

Benannte Funktionen

Benannte Funktionen sind Funktionen deren Deklaration einen Namen enthält unter welchem diese jeweils referenziert werden können. Die Deklaration einer benannten Funktion ist semantisch äquivalent zu der Deklaration einer Variablen desselben Namens sowie der Zuweisung einer identischen anonymen Funktion an diese Variable. Durch das Konzept

benannter Funktionen soll in der Zielsprache strukturierte Programmierung (siehe Abschnitt 3.1) ermöglicht und der Zwang zur Verwendung des objektorientierten Programmierstils vermieden werden. Listing 4.4 zeigt Deklaration und Aufruf einer benannten Funktion.

Listing 4.4: Benannte Funktionen

```
1 fun time(action) {
2         ...
3 }
4
5 .time(...);
```

Anonyme Funktionen

Bei anonymen Funktionen handelt es sich um Funktionen, deren Deklaration keinen Namen enthält. Entsprechend können diese nur über den Wert, den deren Deklaration liefert, referenziert werden. Daher ist in der Zielsprache die Verwendung von anonymen Funktionsdeklarationen nur als Teil eines Ausdrucks möglich. Die Verwendung im Befehlskontext ist dementsprechend ungültig. Listing 4.5 zeigt die Deklaration einer anonymen Funktion. Das Beispiel ist semantisch äquivalent zu dem in Listing 4.4 gezeigten Beispiel.

Listing 4.5: Anonyme Funktionen

```
1 var time := fun (action) {
2         ...
3 };
4
5 .time(...);
```

4.6.2 Block-Closures

In der Zielsprache sind Block-Closures das semantische Gegenstück zu den aus der Sprache *Smalltalk* bekannten Blöcken. Im Kontext der Zielsprache werden diese als Block-Closures bezeichnet, um sie von anderweitig verwendeten Blöcken abzugrenzen.

Block-Closures unterscheiden sich von Funktionen durch die differierende Semantik des Rücksprungbefehls sowie des Ausführungskontexts (siehe Abschnitt 4.7.4). Im Gegensatz zu Funktionen beendet der Rücksprungbefehl in einer Block-Closure nicht nur die Ausführung dieser, sondern ebenso der Funktion innerhalb welcher die Block-Closure definiert wurde (im *Smalltalk*-Jargon werden diese als *Home-Methods* bezeichnet). Ebenso ist der Ausführungskontext einer Block-Closure immer der Ausführungskontext der definierenden Funktion.

Listing 4.6 verdeutlicht den Unterschied in der Semantik von Funktionen und Block-Closures. Dabei wird die in Listing 3.9 gezeigte funktionale Iterationsmethode verwendet indem dieser einmal eine Funktion sowie ein anderes Mal eine Block-Closure übergeben wird.

Listing 4.6: Rücksprungbefehl in Block-Closures

```
1  select count(*) from stock_report;
2  -- Ausgabe: 10
3
4  var count, stmt;
5
6  var count = 0;
7  var stmt = sql select * from stock_report;
8  .stmt.first = fun () {
9          -- Uebergabe einer anonymen Funktion.
10         this.each(fun (data) {
11                 count = count + 1;
12                 return; -- Ruecksprungbefehl
13         });
14 };
15 .stmt.first();
16
17 println(count);
18 -- Ausgabe: 10
19
20 var count = 0;
21 var stmt = sql select * from stock_report;
22 .stmt.first = fun () {
23         -- Uebergabe einer Block-Closure
24         this.each { data =>
25                 count = count + 1;
26                 return; -- Ruecksprungbefehl
27         };
28 };
29 .stmt.first();
30
31 .println(count);
32 -- Ausgabe: 1
```

4.6.3 Funktionsparameter

Funktionen sowie Block-Closures können beim Aufruf Parameter übergeben werden. Hierbei wird die in Abschnitt 3.7 favorisierte *Call-by-value*-Semantik verwendet. Die Anzahl der gültigen Parameter einer Funktion wird bei deren Deklaration definiert. Die Unterstützung einer variablen Anzahl von Parametern wird in der Zielsprache aus Gründen der Einfachheit vorerst nicht verfolgt.

4.7 Objekte

Aufgrund der Vorzüge des in Abschnitt 3.2.5 vorgestellten Ansatzes der durchgehenden Objektorientierung werden in der Zielsprache alle Werte durch Objekte repräsentiert. Gleiches gilt für Funktionen und Block-Closures. Außerdem wird der ebenfalls erwähnte Ansatz verfolgt, sämtliche Operationen durch Versand von Nachrichten an Objekte zu realisieren, unter anderem da hierdurch die Semantik der Sprache sehr leicht verändert werden kann. Die einzige Ausnahme davon stellen Zuweisungsoperationen an Variablen dar, die, wie bereits erläutert, generell eine Ausnahme vom objektorientierten Ansatz darstellen.

Das in der Zielsprache verwendete Objektmodell wird in den folgenden Abschnitten genauer erläutert.

4.7.1 Struktur von Objekten

Objekte werden in der Zielsprache durch eine Menge von eindeutigen Schlüsseln, im Folgenden *Slots*[12] genannt, repräsentiert, welchen beliebige Werte zugeordnet werden können. Jedes Objekt der Zielsprache kann hierbei sowohl als Schlüssel als auch als Wert fungieren.

Objekte können zur Laufzeit beliebig modifiziert werden. Die Zielsprache unterstützt das Hinzufügen, Entfernen und Modifizieren von Slots, sodaß eine hohe Dynamik erreicht wird. Das Hinzufügen von Slots erfolgt durch einfaches Zuweisen eines Wertes an einen noch nicht existierenden Slot eines Objektes.

4.7.2 Vererbung

Wie in Abschnitt 3.1 dargelegt, wird in der Zielsprache das Konzept der prototyp-basierten Objektorientierung verfolgt. Vererbung wird dabei durch das Konzept der Delegation (siehe Abschnitt 3.2.1) realisiert. Hierfür verwendet die Zielsprache ein gängiges Modell, in dem jedes Objekt einen speziellen Slot, den *Parent Slot* enthalten kann.

Auf die Implementierung von Mehrfachvererbung wird im Rahmen dieser Arbeit vorerst verzichtet. Diese könnte durch die Unterstützung meherer Parent Slots innerhalb eines Objektes leicht nachgerüstet werden. Da dies jedoch die Semantik der Zielsprache nicht entscheidend verändern würde, wird auf die konkrete Unterstützung der Mehrfachvererbung vorerst verzichtet.

4.7.3 Objekterzeugung

Die Zielsprache unterstützt zwei Arten der Objekterzeugung, die bereits in Abschnitt 3.2.1 vorgestellt wurden. Die *Ex-Nihilo*-Objekterzeugung wird dabei über eine spezielle,

[12] Beim Begriff *Slot* handelt es sich um eine in der prototyp-basierten Objektorientierung gängige Bezeichnung für eine Nachricht, die ein Objekt versteht.

als Objektliterale bezeichnete, Syntax ermöglicht. Diese orientiert sich an der aus *JavaScript* bekannten, aber nicht auf diese Sprache beschränkten *JavaScript Object Notation (JSON)*. Ziel dabei ist auch hier, einen einfachen Einstieg durch Verwendung gängiger Syntax zu gewährleisten.

Listing 4.7: Objekterzeugung mithilfe des Objektliterals

```
1 var anObject := {
2         aProperty: 'hello',
3         aMethod: fun () {
4                 return 'world';
5         }
6 };
```

Das Klonen von Objekten wird, wie in Abschnitt 3.2.1 vorgesehen, über das Konzept der *Differentiellen Vererbung* realisiert. Listing 4.8 verdeutlicht das Prinzip.

Listing 4.8: Klonierung mit differentieller Vererbung

```
1  var Stock := {
2          code: '123'
3  };
4
5  var stockValue := Stock.clone();
6  .stockValue.productID = 42;
7  .stockValue.quantity = 23;
8
9  .print("stockValue.code: " + stockValue.code);
10 .print("stockValue.productID: " + stockValue.productID);
11
12 -- Ausgabe:
13 -- stockValue.code: 123
14 -- stockValue.productID: 42
15
16 .Stock.code = '456';
17
18 .print("stockValue.code: " + stockValue.code);
19
20 -- Ausgabe:
21 -- stockValue.code: 456
```

Neben der Klonierung unterstützt die Zielsprache, mithilfe eines speziellen `new`-Operators, eine der Instanziierung bei der klassenbasierten Objektorientierung ähnliche Operation, um die Verwendung in klassenbasierter Semantik zu vereinfachen (siehe Abschnitt 3.2.1). Die Benutzung des `new`-Operators zeigt Listing 4.9.

Alle in der Zielsprache erzeugten Objekte besitzen ein spezielles Basisobjekt als Wurzel ihrer Objekthierarchie. Dieses implementiert grundlegende Funktionalitäten der Zielsprache. Modifikationen an diesem Basisobjekt wirken sich daher auf alle Objekte aus – auch auf solche, die mit Hilfe von Objektliteralen erzeugt wurden.

Listing 4.9: Klassenbasierte Objekterzeugung

```
1  var ValueObject := {
2          -- Init-Nachricht wird durch new-Operator gesendet.
3          -- Fungiert als Konstruktor.
4          init: fun (data) {
5                  data.eachSlot { name, value =>
6                          this{name} = value;
7                  };
8          }
9  };
10
11 var value := new ValueObject({ hello: 'world' });
12
13 .print(value.hello); -- Ausgabe: world
14 .print(ValueObject.hello); -- Ausgabe: null
```

4.7.4 Nachrichten und Ausführungskontext

Wie in allen objektorientierten Sprachen können auch in der Zielsprache Nachrichten an Objekte gesendet werden. Die Antwort besteht jeweils aus dem Wert, der demjenigen Slot im Empfängerobjekt zugeordnet ist, welcher der Nachricht entspricht. Sofern dieser Wert eine Funktion ist, kann diese evaluiert werden. Die Evaluierung erfolgt dabei im Kontext des Empfängerobjektes. Dieses kann innerhalb der Funktion über das auch in anderen Sprachen gängige `this`-Schlüsselwort referenziert werden. Ebenso unterstützt die Zielsprache das `super`-Schlüsselwort. Dieses repräsentiert dabei das Parent-Objekt des Objektes, in dem die ausgeführte Funktion gefunden wurde. Nachrichten, die an `super` gesendet werden, werden also an dieses Parent-Objekt gesendet, wobei der Ausführungskontext beibehalten wird und daher dem originalen Empfängerobjekt entspricht. Listing 4.10 demonstriert dieses Verhalten an einem Beispiel.

Der in Abschnitt 3.2.5 erläuterte Ansatz, alle Operationen in der Zielsprache durch das Senden von Nachrichten zu repräsentieren wird in der Zielsprache verfolgt. Um diesen Ansatz praktikabel zu gestalten wird zwischen normalen Nachrichten, binären Nachrichten und unären Nachrichten als Präfixoperatoren unterschieden. Auf die Letzteren geht der folgende Abschnitt genauer ein.

Binäre Nachrichten

Binäre Nachrichten sind ein unter anderem von *Smalltalk* her bekanntes Konzept, mit Hilfe dessen mathematische und andere Operationen in objektorientierter Semantik aber dennoch syntaktisch in ihrer gewohnten Form notiert werden können. Binär sind dabei Nachrichten, welche zwei Argumente besitzen, das Empfängerobjekt sowie ein weiteres beliebiges Objekt als Parameter. Anders als *Smalltalk* beachtet die Zielsprache jedoch,

Listing 4.10: Auführungskontext: *this* und *super*

```
var ValueObject := {
        init: fun (data) {
                data.eachSlot { name, value =>
                        -- this referenziert die aktuelle Instanz.
                        -- Initialisiert die Slots der Instanz mit
                        -- den Werten des uebergebenen Objektes.
                        this.setSlot(name, value);
                };
        }
};

var Product := {
        -- ValueObject als Parent-Objekt von Product definieren.
        parent: ValueObject,
        init: fun (data) {
                -- Nachricht init() an Parent-Objekt senden.
                super.init(data);
                -- Weiteren Slot in der aktuellen Instanz erzeugen.
                this.parts = null;
        }
};

var productData :=
        (sql select * from products where id = 1).first();
var product := new Product(productData);
```

wie zum Beispiel auch die Sprache *IO*[13], mathematische Vorrangregeln für eine bestimmte Menge von Operatoren. Operatoren, die nicht Teil dieser Menge sind, also zum Beispiel benutzerdefinierte Operatoren, werden von links nach rechts evaluiert.

Listing 4.11: Vorrangregeln bei binären Nachrichten

```
-- Zunaechst eigenen Multiplikationsoperator definieren.
Num.mul = Num.*;

-- Multiplikation mit Standardoperator.
-- Mathematische Vorrangregeln werden beachtet.
var i := 2 + 3 * 4; -- Ergibt: 14

-- Multiplikation mit eigenem Operator.
-- Evaluierung von links nach rechts.
var j := 2 + 3 mul 4; -- Ergibt: 20
```

Einen Sonderfall stellen die booleschen Operatoren *logisches UND* sowie *logisches ODER* dar. Diese Operatoren sind insofern speziell, als daß deren Parameter jeweils nur evaluiert werden sollte, wenn der Empfänger dem booleschen Wert *wahr* respektive *falsch* entspricht. Da die Zielsprache jedoch wie in Abschnitt 4.6.3 beschrieben die *Call-*

[13] http://www.iolanguage.com/

by-value-Strategie statt der *Call-by-name*-Strategie (siehe Abschnitt 3.7) implementiert, werden als Funktionsparameter verwendete Ausdrücke bereits vor dem Aufruf der jeweiligen Funktion evaluiert. Um dies zu verhindern und eine bedingte Evaluierung im Kontext der Operatoren zu ermöglichen, werden Ausdrücke, die als Parameter der betreffenden booleschen Operatoren verwendet werden, während der Übersetzung eines Programmes transparent in Block-Closures gekapselt. Dies ermöglicht dem Empfängerobjekt, die Evaluierung des Parameters zu steuern.

Unäre Nachrichten als Präfixoperatoren

Unäre Nachrichten, also Nachrichten mit nur einem Argument, dem Empfängerobjekt, werden in einigen Fällen ähnlich wie binäre Nachrichten besonders behandelt. Die Zielsprache definiert dabei eine bestimmte Menge unärer Operatoren, welche auf der linken Seite des Empfängerobjektes notiert werden, sodaß deren Syntax mit der entsprechenden mathematischen Notation korrespondiert.

4.7.5 Objekte als assoziative Arrays

Inspiriert von *JavaScript* werden Objekte in der Zielsprache grundlegend wie assoziative Arrays modelliert und entsprechend auch als solche verwendet. Dies ermöglicht sowohl eine einfache und leicht verständliche Semantik der Manipulation von Objekten als auch einen bequemen Umgang mit assoziativen Arrays. Allerdings stellt diese Modellierung auch einige besondere Anforderungen an die Realisierung von Objekten, die im Folgenden erläutert werden.

Schlüssel

Für eine hohe Flexibilität, empfiehlt es sich, in assoziativen Arrays eine möglichst geringe Beschränkung der als Schlüssel verwendbaren Werte zu erreichen. Daher können in der Zielsprache Slots, genau wie die darüber referenzierten Werte durch beliebige Objekte der Skriptsprache repräsentiert werden.

Iterationsverhalten

Bei der Verwendung eines Objektes als assoziatives Array muß sichergestellt werden, daß eine Iteration über die Schlüssel des assoziativen Arrays nur diejenigen Slots liefert, die explizit in dem betreffenden Objekt gesetzt wurden. Die Iteration darf also Slots von Parent-Objekten nicht berücksichtigen.

Problematisch wäre hier die Ableitung aller Objekte vom bereits erwähnten Basisobjekt, wenn dies über die Zuweisung des Parent Slot erfolgen würde. In *JavaScript* zum Beispiel wird dieses Problem umgangen, indem jeder Slot eines Objektes eine *enumerable* genannte Eigenschaft spezifiziert, die angibt, ob eine Iteration über den jeweiligen Slot

erfolgen kann. In der Zielsprache wird stattdessen das Konzept impliziter Parent Slots definiert. Diese legen dabei eine Vererbungsbeziehung zwischen zwei Objekten fest, ohne einen Parent Slot explizit zu spezifizieren. Dadurch kann intern die Vererbungsbeziehung eines Objektes von der expliziten Definition eines Slots entkoppelt werden, sodaß diese sich nicht auf das Iterationsverhalten desselben auswirkt.

Listing 4.12 demonstiert dieses Konzept an einem Beispiel.

Listing 4.12: Iterationsverhalten von Objekten

```
1  -- Datensatz laden. Liefert ein Objekt mit nur zwei Slots
2  -- „id“ und „name“.
3  var data := (sql select id, name from products).first();
4
5  -- Nachricht eachSlot() definiert im Basis-Objekt,
6  -- das als implizites Parent-Objekt referenziert ist.
7  .data.eachSlot { name, value =>
8          print("@{name}: @{value}");
9  };
10
11 -- Ausgabe (z.B.):
12 -- id: 1
13 -- name: Artikel1
14 --
15 -- Weder „parent“ noch „eachSlot“ erscheinen in der Ausgabe.
```

Schlüsselsuche

Bei der Verwendung eines Objektes als assoziatives Array, muß sichergestellt sein, daß beim Lesen eines Slots dieses Objektes ausschließlich Slots berücksichtigt werden, die explizit spezifiziert wurden. Es darf entsprechend nicht die normalerweise erfolgende Suche von Slots in den explizit oder implizit spezifizierten Parent-Objekten durchgeführt werden, wenn der gesuchte Slot nicht im betreffenden Objekt selbst existiert. Dafür stellt die Zielsprache zum einen entsprechende Funktionalität über ein bestimmtes vordefiniertes Objekt (siehe Anhang A) zur Verfügung, zum anderen wird ein Objekt bereitgestellt, welches beliebige Objekte kapseln kann und die geschilderte Semantik beim Zugriff auf Slots des gekapselten Objektes berücksichtigt. Beides wird in Listing 4.13 beispielhaft präsentiert.

4.8 Ausnahmebehandlung

Um die Entwicklung von fehlertoleranten Programmen sinnvoll zu unterstützten, verwendet die Zielsprache das gängige Konzept der strukturierten Ausnahmebehandlung (siehe z.B. [Cri89]). Hierzu stellt die Zielsprache den `try-catch-finally`-Pseudobefehl sowie

Listing 4.13: Schlüsselsuche unter Vermeidung von Parent-Objekten

```
1  -- Beispielhafte Implementierung des Dict-Objektes
2  var Dict := {
3          init: fun (anObject) {
4                  this.wrappedObject = anObject;
5          },
6          get: fun (key) {
7                  -- Wert des mit key spezifizierten Slots
8                  -- liefern, ohne Suche in Parent-Objekten.
9                  -- Liefert null, falls nicht gefunden.
10                 return Sys.explicitSlot(this.wrappedObject, key);
11         }
12 };
13
14 -- Definition eines beliebigen Objektes, das ein Parent-Objekt
15 -- referenziert.
16 var anObject := {
17         parent: {
18                 slotInParent: 'a parent slot value'
19         },
20         slotInObject: 'a slot value'
21 };
22
23 .print("anObject.slotInObject = " + anObject.get('slotInObject'));
24 .print("anObject.slotInParent = " + anObject.get('slotInParent'));
25
26 -- Ausgabe:
27 -- anObject.slotInObject = a slot value
28 -- anObject.slotInParent = null
```

den korrespondierenden `throw`-Pseudobefehl und entsprechende Ausdrücke zur Verfügung (siehe Anhang A). Ausnahmen können in der Zielsprache durch beliebige Objekte repräsentiert werden und müssen nicht wie in einigen anderen Programmiersprachen, wie zum Beispiel *Java*, Objekte eines bestimmten Typs sein.

4.9 SQL-Befehle

SQL-Befehle werden in der Zielsprache durch Objekte repräsentiert. Ihre Verwendung kann im Befehlskontext sowie im Kontext eines Ausdrucks erfolgen. Im ersten Fall wird der Befehl während der Übersetzung transparent in einen Ausdruck umgewandelt. Dieser entspricht dem Versand einer bestimmten Nachricht an das, den SQL-Befehl repräsentierende, Objekt und veranlasst dessen sofortige Ausführung. Die Antwort auf diese Nachricht ist je nach Art des SQL-Befehls (siehe Abschnitt 2.4.2) ein Objekt, welches die Ergebnismenge oder den *Update Count*[14] des SQL-Befehls repräsentiert.

Die Zielsprache unterstützt sowohl statische als auch dynamische SQL-Befehle in den

[14] Die Anzahl der durch den SQL-Befehl veränderten Datensätze.

Listing 4.14: SQL in Befehls- und Ausdrucksform

```
1 -- SQL in Befehlsform
2 select * from stock_report;
3 -- Ausgabe: Formatierte Ergebnistabelle
4
5 -- Analog in Ausdrucksform
6 .(select * from stock_report).do();
7 -- Ausgabe: Formatierte Ergebnistabelle
```

in Abschnitt 2.4.2 genannten Varianten. Die folgenden Abschnitte gehen genauer auf die unterschiedlichen Arten der SQL-Befehle ein.

4.9.1 Statische SQL-Befehle

Statische SQL-Befehle enthalten weder Variablenreferenzen noch eingebettete Parameter und sind entsprechend unveränderlich. Sie können daher sowohl im Befehls- als auch im Ausdruckskontext verwendet werden.

4.9.2 SQL-Befehle mit eingebetteten Variablenreferenzen

Es wird in der Zielsprache die Einbettung von Variablenreferenzen in SQL-Befehle unterstützt. Diese Referenzen müssen in einer speziellen Syntax notiert werden, die in Anhang A beschrieben wird. Die Referenzierung kann dabei zum einen innerhalb von Zeichenkettenliteralen des SQL-Befehls, zum anderen an beliebigen anderen Stellen des Befehls erfolgen.

Bei der Referenzierung innerhalb von Zeichenkettenliteralen wird der Wert der referenzierten Variablen an der entsprechenden Stelle des Literals eingesetzt. Sind im eingesetzten Wert Zeichen enthalten, welche innerhalb des betreffenden Literals eine spezielle Bedeutung haben, werden diese vor dem Einsetzen maskiert. Hierfür ist jedoch die korrekte Verwendung der Parse-Direktive notwendig, über welche der aktive SQL-Dialekt spezifiziert werden kann (siehe Abschnitt 4.11).

Beide Arten der Einbettung von Variablenreferenzen werden in Listing 4.15 gezeigt.

Listing 4.15: Einbettung von Variablenreferenzen in SQL-Befehle

```
1 -- Referenzierung im Befehlstext.
2 var reportsTable := 'stock_reports';
3 select * from @{reportsTable};
4
5 -- Referenzierung im Zeichenkettenliteral. Enthaltenes
6 -- Begrenzerzeichen wird bei Einsetzung automatisch maskiert.
7 var stockCode := "some'stock";
8 select * from stock_reports where stock_code = '@{stockCode}';
```

SQL-Befehle mit eingebetteten Variablenreferenzen können im Befehlskontext und im Kontext von Ausdrücken verwendet werden.

Nach dem Auflösen von Variablenreferenzen kann ein SQL-Befehl Parameter enthalten. Ist dies der Fall gelten die für parametrisierte SQL-Befehle definierten Beschränkungen, die im folgenden Abschnitt beschrieben werden.

4.9.3 Parametrisierte SQL-Befehle

Neben Variablenreferenzen können SQL-Befehle in der Zielsprache auch Parameter, wie in Abschnitt 2.4.2 dargelegt, enthalten. Diesen Parametern können über bestimmte Nachrichten, die die SQL-Befehlsobjekte verstehen, Werte zugewiesen werden. Listing 4.16 zeigt ein Beispiel. Da diese Zuweisungen vor dem Ausführen eines solchen SQL-Befehls erfolgen müssen, können Parameter bei Verwendung eines SQL-Befehls im Befehlskontext nicht ersetzt werden und werden entsprechend auch nicht als solche interpretiert.

Listing 4.16: Parametrisierte SQL-Befehle

```
1  -- Positionale Parameter
2  .ProductStock.loadStockQty = fun (product, stock) {
3          (sql select qty from product_stock
4                where product_id = ? and stock_id = ?)
5                  .with(product.id, stock.id)
6                  .each { data => return data['qty']; };
7          return null;
8  };
9
10 -- Benannte Parameter
11 .ProductStock.loadStockQty = fun (product, stock) {
12         (sql select qty from product_stock
13               where product_id = :productID and stock_id = :stockID)
14                 .withNamed({ productID: product.id, stockID: stock.id })
15                 .each { data => return data['qty']; };
16         return null;
17 };
```

4.10 Befehlsmodi

Um der Forderung nach Abwärtskompatibilität (Abschnitt 2.2.6) und Interaktivität (Abschnitt 2.2.4) gerecht zu werden, wird in der Zielsprache zwischen zwei verschiedenen Befehlsmodi unterschieden. Es handelt sich dabei zum einen um den SQL-Befehlsmodus, zum anderen um den Skriptbefehlsmodus. Diese verwenden jeweils unterschiedliche syntaktische Regeln. Im Folgenden werden diese Modi genauer erläutert sowie deren Notwendigkeit begründet.

4.10.1 SQL-Befehlsmodus

Die Forderung nach Abwärtskompatibilität bedingt die Fähigkeit der Zielsprache, bestehende SQL-Skripte ausführen zu können. Außerdem macht die Forderung nach Interaktivität die Fähigkeit der Zielsprache notwendig, SQL-Befehle möglichst direkt und einfach absetzen zu können. Der SQL-Befehlsmodus ist daher der Standardmodus.

Entsprechend der Syntax von SQL (siehe Abschnitt 2.3.2) werden in diesem Modus alle Befehle, die der Grundstruktur von SQL-Befehlen entsprechen, als solche interpretiert und direkt ausgeführt. Skriptbefehle, wie Variablendeklarationen oder Funktionsaufrufe, müssen in diesem Modus syntaktisch von SQL-Befehlen abgegrenzt werden. Dazu definiert die Zielsprache einen Präfix, welcher, dem SQL-Standard entsprechend, nicht der Beginn eines SQL-Befehls sein kann und damit ermöglicht, Skriptbefehle syntaktisch eindeutig von SQL-Befehlen abzugrenzen. Desweiteren definiert die Zielsprache einige Schlüsselworte, welche weder in *SQL-99*, noch in den explizit unterstützten Datenbanksystemen Verwendung als Beginn eines SQL-Befehls finden. Skriptbefehle, welche mit derartigen Schlüsselworten beginnen, werden in Zielsprache als ausreichend von SQL-Befehlen abgegrenzt angenommen und bedürfen auch im SQL-Befehlsmodus nicht der Abgrenzung durch den erwähnten Präfix.

Es besteht dennoch die Möglichkeit, daß nicht explizit unterstützte Datenbanksysteme bzw. SQL-Dialekte Befehle definieren, welche in Konflikt mit den definierten Schlüsselworten stehen. Dies wird jedoch zugunsten der Benutzbarkeit der Zielsprache in Kauf genommen, da derartige Konflikte ausschließlich eine Beeinträchtigung der Abwärtskompatibilität, nicht jedoch der Funktionalität der Zielsprache bedeuten. Desweiteren werden die betreffenden Schlüsselworte sorgfältig gewählt, um Konflikte zu minimieren.

Neben der impliziten Aktivierung des SQL-Befehlsmodus als Standardmodus bietet die Zielsprache auch die Möglichkeit, diesen Modus explizit, innerhalb des Skriptbefehlsmodus (siehe unten), zu aktivieren. Dazu wird eine spezielle Blocksyntax definiert (siehe Anhang A).

Listing 4.17 verdeutlicht die Syntax des SQL-Befehlsmodus an einem Beispiel.

4.10.2 Skriptbefehlsmodus

Die Zielsprache unterstützt einen Skriptbefehlsmodus, um eine akzeptable Benutzbarkeit für die Skriptprogrammierung zu gewährleisten. Im Unterschied zum SQL-Befehlsmodus müssen hier SQL-Befehle mit einem Präfix versehen werden, um als solche interpretiert zu werden. Alle Befehle ohne diesen Präfix werden als Skriptbefehle interpretiert. Der im vorigen Abschnitt erwähnte Präfix für Skriptbefehle ist daher nicht notwendig, kann aber dennoch verwendet werden. Gleiches gilt für den SQL-Befehlspräfix im SQL-Befehlsmodus.

Der Skriptbefehlsmodus wird in der Zielsprache innerhalb eines jeden Blockes mit Ausnahme des speziellen SQL-Befehlsblockes (siehe voriger Abschnitt) aktiviert. Zu diesen

Listing 4.17: SQL-Befehlsmodus

```
1  -- Skriptbefehl mit Schluesselwort im SQL-Befehlsmodus.
2  -- Kein Praefix notwendig.
3  if (tableExists('stock_report')) {
4      -- Explizite Aktivierung des SQL-Befehlsmodus mithilfe
5      -- des SQL-Befehlsblockes.
6      sql {
7          drop table stock_report;
8      }
9  }
10
11 -- SQL-Befehl im SQL-Befehlsmodus. Kein Praefix notwendig.
12 create table stock_report (...);
13
14 -- Funktionsaufruf. Punkt als Praefix leitet Ausdruck
15 -- im SQL-Befehlsmodus ein
16 .print('Created table stock_report');
```

Blöcken zählen zum Beispiel der Rumpf von Kontrollstrukturen wie `if`-Anweisungen oder `while`-Schleifen, Funktionen oder Block-Closures.

Aufgrund der speziellen Syntax für Skriptbefehle im SQL-Befehlsmodus ist dieses Verfahren ein guter Kompromiss aus Benutzbarkeit und Abwärtskompatibilität. Der Skriptbefehlsmodus kann nicht versehentlich aktiviert werden, stattdessen muß dies durch explizite Notation geschehen. Dadurch kann die Intention des Benutzers impliziert werden, innerhalb eines, den Skriptbefehlsmodus aktivierenden, Befehls weitere Skriptbefehle notieren zu wollen. Das gewählte Verfahren unterstützt diesen Anwendungsfall entsprechend.

Listing 4.18 zeigt die verschiedenen Elemente des Skriptbefehlsmodus an einem Beispiel.

Listing 4.18: Skriptbefehlsmodus

```
1  -- Block der If-Anweisung aktiviert Skriptbefehlsmodus implizit.
2  if (tableExists('stock_reports')) {
3      -- SQL-Befehl im Skriptbefehlsmodus. Praefix sql notwendig.
4      sql drop table stock_reports;
5
6      -- Ausdruck im Skriptbefehlsmodus.
7      -- Punkt als Praefix nicht notwendig, aber erlaubt.
8      .print('Dropped table stock_reports');
9  }
```

4.11 Parse-Direktiven

Aufgrund der unterschiedlichen syntaktischen Anforderungen der verschiedenen SQL-Dialekte, ist es, wie in Abschnitt 2.3.2 beschrieben, notwendig, SQL-Befehle situationsabhängig unterschiedlich zu parsen. Da Informationen über den aktiven Dialekt nicht zwingend zum Zeitpunkt der Übersetzung eines Skriptes zur Verfügung stehen, muß der Benutzer entsprechend die Möglichkeit haben, diese explizit zu spezifizieren.

Daher definiert die Zielsprache Parse-Direktiven, also spezielle Anweisungen mit welchen das Verhalten des Parsers der Sprache direkt beeinflusst werden kann. Dies betrifft insbesondere das Verhalten bezüglich von Kommentaren und Zeichenkettenliteralen innerhalb von SQL-Befehlen.

Um diese Direktiven im Parser einfach identifizieren zu können, werden diese syntaktisch durch einen weiteren Präfix abgegrenzt, der weder in Skript- noch in SQL-Befehlen Verwendung findet.

Ein Beispiel in dem eine Parse-Direktive verwendet wird, um verschiedene Zeichenkettensyntaxen zu aktivieren, zeigt Listing 4.19.

Listing 4.19: Variierung der Zeichenkettensyntax mithilfe von Parse-Direktiven

```
1  -- Zeichenkettensyntax nach SQL-Standard per Default aktiv.
2  select "stock_quantity" from "stock_reports"
3  where product_number = 'WEIRD''PRODUCT;';
4
5  -- Explizite Aktivierung der von MySQL unterstuetzten
6  -- Zeichenkettensyntax mithilfe der set-Parse-Direktive.
7  \set quotes=mysql;
8  select `stock_quantity` from `stock_reports`
9  where product_number = "WEIRD'PRODUCT;";
10
11 -- Umschalten zu der von Oracle unterstuetzten Syntax.
12 \set quotes=oracle;
13 select "stock_quantity" from "stock_reports"
14 where product_number = q'{WEIRD'PRODUCT;}';
```

5 Konzeption des Interpreters

Nachdem im letzten Kapitel syntaktische und semantische Aspekte der Sprache erörtert wurden, befaßt sich dieses Kapitel mit Übersetzung und Ausführung von Programmen der Sprache. Dabei werden involvierte Strukturen und Abläufe definiert sowie Konzepte für grundlegende Funktionalitäten der Zielsprache entwickelt.

5.1 Architektur

Der Interpreter besteht aus drei Komponenten, welche eine bestehende Plattform – im folgenden als Hostsystem bezeichnet – zur Ausführung verwenden. Bei diesen Komponenten handelt es sich um die folgenden:

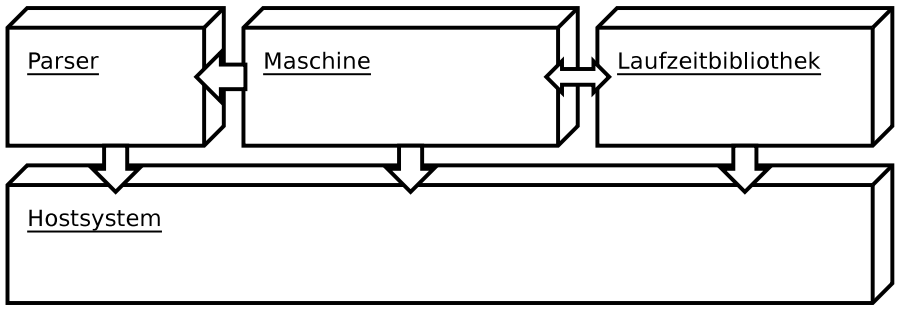
Übersetzer – Übernimmt die Transformation des Quellcodes eines Programmes in eine ausführbare Repräsentation.

Maschine – Interpretiert den vom Übersetzer erzeugten Code und führt diesen aus.

Laufzeitbibliothek – Stellt Programmen der Zielsprache grundlegende Funktionalität zur Verfügung und fungiert als Schnittstelle zwischen Zielsprache und Hostsystem.

In den folgenden Abschnitten wird zunächst der Übersetzer beschrieben. Anschließend werden die Maschine sowie für die Laufzeitbibliothek relevante Aspekte behandelt.

Abbildung 5.1: Architektur des Interpreters



5.2 Übersetzer

Wie im vorigen Abschnitt erwähnt, ist der Übersetzer verantwortlich für die Transformation des Quellcodes eines Programmes der Zielsprache in eine Darstellung, die von der Maschine interpretiert werden kann.

In den folgenden Abschnitten wird dessen Konzeption erläutert.

5.2.1 Werkzeuge

Die Entwicklung eines Übersetzers für eine einigermaßen komplexe Sprache, kann eine umfangreiche Aufgabe darstellen. Jedoch existiert ein breites Spektrum an Werkzeugen wie Parser-Generatoren, mit deren Hilfe diese Aufgabe erheblich vereinfacht werden kann. Für die Entwicklung des Übersetzers der Zielsprache wird der Parser-Generator *ANTLR*[15] verwendet. Dieser kann ausgehend von der Spezifikation einer Grammatik einer Sprache Parser für diese generieren, die das Verfahren des rekursiven Abstiegs (siehe z.B. [AEU99]) verwenden.

ANTLR wurde gewählt, da dieses Werkzeug im Vergleich zu anderen Parser-Generatoren relativ gut dokumentiert ist, unter anderem in [Par07], aktiv weiterentwickelt wird und ein umfangreiches Portfolio an Funktionalität bietet. So unterstützt ANTLR eine variable Größe des Lookahead innerhalb einer Grammatik, syntaktische Prädikate die eine beliebige Größe des Lookahead ermöglichen (eine auch LL(*) genannte Technik) sowie semantische Prädikate die es ermöglichen, den Parsevorgang zur Laufzeit zu beeinflussen. Zusätzlich ist ANTLR in der Lage, nicht nur abstrakte Syntaxbäume sondern auch Parser für diese zu generieren. Ein weiterer Vorteil ist die relativ grosse Anzahl an Zielsprachen für welche Parser generiert werden können. Zuletzt ist zu erwähnen, daß ein relativ großes Ökosystem an Entwicklungswerkzeugen existiert, die die Arbeit mit ANTLR erleichtern (z.B. *ANTLRWorks*[16]).

5.2.2 Phasen der Übersetzung

Die Übersetzung von Programmen kann in die folgenden verschiedenen Phasen (vgl. [AEU99]) unterteilt werden:

Lexikalische Analyse, bei der der Zeichenstrom des Quellprogramms in eine Folge von Symbolen zerlegt wird.

Syntaxanalyse, bei der die Symbole zu grammatikalischen Sätzen der Zielsprache zusammengefasst werden.

[15] `http://www.antlr.org/`
[16] `http://www.antlr.org/works`

Semantikanalyse, bei der das Quellprogramm auf semantische Fehler überprüft und Informationen gesammelt werden.

Zwischencodeerzeugung, bei der das Quellprogramm in eine für die folgende Phase geeignete Darstellung übersetzt wird.

Codeoptimierung, bei der der erzeugte Zwischencode modifiziert wird, um eine möglichst effiziente Ausführung des Programms zu erreichen.

Codeerzeugung, bei der eine Repräsentation des Quellprogramms erzeugt wird, die von einer bestimmten Maschine ausgeführt werden kann.

Die Phasen *Zwischencodeerzeugung* sowie *Codeoptimierung* werden für die Realisierung des Prototypen der Zielsprache nicht in Betracht gezogen, da das Hauptaugenmerk zunächst auf deren Funktionalität und weniger auf deren Effizienz liegt.

Da die Zielsprache diverse syntaktische Elemente definiert, deren Semantik identisch ist (siehe Abschnitt 4.3.2), wird die Übersetzung in mehreren Schritten durchgeführt. Im ersten Schritt erfolgt dabei die lexikalische sowie die Syntaxanalyse, bei welcher derartige syntaktische Strukturen in einheitliche Strukturen überführt werden. Die weiteren Phasen der Übersetzung erfolgen im nächsten Schritt. Bei dem zu entwickelnden Übersetzer handelt es sich daher um einen Multi-Pass-Übersetzer (vgl. [Bor90]) bzw. präziser formuliert um einen Two-Pass-Übersetzer, bei dem die Übersetzung in zwei Schritten durchgeführt wird.

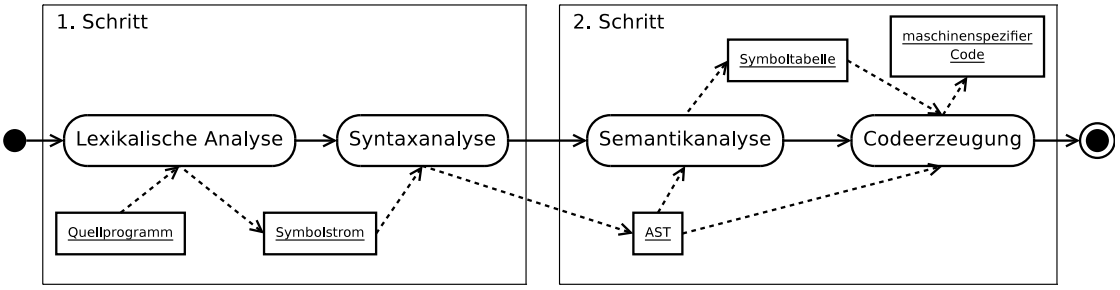
Im ersten Schritt, in dem lexikalische und Syntaxanalayse durchgeführt werden, wird aus einem gegebenen Quellprogramm ein abstrakter Syntaxbaum (AST) generiert. Im zweiten Schritt werden Semantikanalyse und Codeerzeugung durchgeführt. Während der Semantikanalyse wird der generierte AST analysiert und eine Symboltabelle aufgebaut. Der folgende Abschnitt geht hierauf genauer ein. Bei der Codeerzeugung werden anhand des AST Strukturen erzeugt, welche als Eingabe für die Maschine, also den eigentlichen Interpreter, dienen können.

Die Phase der Codeerzeugung ist für Maschinen, die nach dem Interpreter-Visitor-Pattern (siehe Abschnitt 3.8.1) konzipiert sind, nicht zwingend erforderlich. Stattdessen ist es mit derartigen Maschinen auch möglich, ASTs direkt zu interpretieren[17]. Gleiches gilt ebenso für den gewählten Ansatz des Continuation-Passing-Interpreters (siehe Abschnitt 3.8.2). Die Überführung des AST in eine maschinenspezifische Struktur führt jedoch zu einer besseren Trennung zwischen Parser und Maschine, da zum Beispiel nicht jede Änderung der Struktur des AST zu einer Anpassung der Maschine zwingen muss. Auf die Phase der Codeerzeugung wird entsprechend nicht verzichtet.

Der Vorgang der Übersetzung kann damit wie in Abbildung 5.2 dargestellt werden.

[17] Der Interpreter *MRI* (`http://en.wikipedia.org/wiki/Ruby_MRI`) für die Programmiersprache *Ruby* ist hierfür ein Beispiel.

Abbildung 5.2: Vorgang der Übersetzung



5.2.3 Komponenten des Übersetzers

Die einzelnen Phasen der Übersetzung werden von unterschiedlichen Komponenten durchgeführt. Die lexikalische Analyse wird vom *Scanner* bzw. *Lexer* durchgeführt. Die Syntaxanalyse wird vom *Parser* übernommen. Semantikanalyse und Codeerzeugung werden vom *Walker* durchgeführt, der einen Parser für abstrakte Syntaxbäume darstellt.

5.2.4 Objekte des Übersetzers

Der Übersetzer verwendet verschiedene Datenobjekte. Das Quellprogramm wird dabei von einem *Zeichenstrom* repräsentiert, aus welchem der Scanner einen *Symbolstrom* generiert. Sowohl der Zeichenstrom als auch der Symbolstrom sind Objekte, welche durch Standardimplementierungen von ANTLR bereitgestellt werden. Im Kontext der Zielsprache ergeben sich jedoch besondere Anforderungen an diese Objekte, die in einem folgenden Abschnitt, 5.2.5, beschrieben werden. Der vom Parser generierte *abstrakte Syntaxbaum* wird durch entsprechende, von ANTLR bereitgestellte Objekte, repräsentiert. Die *Symboltabelle*, die der Walker generiert wird spezifisch entsprechend der Anforderungen der Zielsprache konzipiert, worauf der folgende Unterabschnitt genauer eingeht. Auf den ebenfalls vom Walker erzeugten *maschinenspezifischen Code* wird in Abschnitt 5.3 eingegangen.

Symboltabelle

Symboltabellen werden von einem Übersetzer verwendet, um Gültigkeitsbereichs- und Bindungsinformationen von Namen zu verwalten (vgl. [AEU99]) und stellen damit eine grundlegende Komponente des Übersetzers dar.

Die vom Übersetzer der Zielsprache verwendete Symboltabelle wird während der Semantikanalyse aufgebaut, da hier das Quellprogramm bereits in einer normalisierten Form vorliegt, und daher der Aufwand für die Erstellung der Symboltabelle geringer ausfällt als dies während der Syntaxanalyse der Fall sein würde.

Im Quellprogramm werden dabei auftretende Namen unter Berücksichtigung des jeweiligen Gültigkeitsbereiches in der Symboltabelle registriert. Da Gültigkeitsbereiche das statische Pendant zu Umgebungen sind (siehe Abschnitt 3.4), erfolgt die Modellierung der Symboltabelle analog dazu, d.h. als Liste von Namen sowie einer Referenz auf die Symboltabelle des jeweils umschliessenden Gültigkeitsbereiches.

Aufgrund der Struktur der Zielsprache sind außer der Verwaltung von Variablennamen keine weiteren Anforderungen an die Symboltabellen präsent. Da die Sprache dynamisch typisiert ist, ist es zum Beispiel nicht notwendig Informationen über den Typ von Variablen zu sammeln.

Zur Verwaltung von Variablennamen gehört auch das Erzeugen der in Abschnitt 3.5 beschriebenen lexikalischen Adressen. Jedem im Quellprogramm auftretenden Variablennamen wird dabei eine Adresse zugeordnet über die diese zur Laufzeit referenziert werden kann.

Um die Verwendung von Namen vor deren Deklaration zu gestatten, was insbesondere im Falle von Funktionen von Nutzen ist, wäre es notwendig, die Verwaltung von Namen in mehreren Schritten während der Übersetzung vorzunehmen. Hierzu müssten im ersten Schritt nicht deklarierte Namen in die Symboltabelle eingetragen werden und als solche markiert werden. In einem weiteren Schritt müssten diese Namen um die entsprechenden Adressen ergänzt werden. Die Konzeption der Übersetzung als Multi-Pass-Übersetzung würde dies ermöglichen. Im Rahmen dieser Arbeit soll hierauf jedoch verzichtet werden, da hierfür auch eine spezielle Behandlung benannter Funktionen (siehe Abschnitt 4.6.1) erforderlich wäre und das aktuell verwendete Konzept die Deklaration von Funktionen vor deren Definition erlaubt. Eine Ersatzfunktionalität ist somit gegeben.

5.2.5 Besonderheiten

In diesem Abschnitt werden einige Besonderheiten bei der Übersetzung betrachtet, die sich aus der Zielsprache bzw. den Anforderungen an diese ergeben.

Inkrementelle Übersetzung

Die Forderung nach Abwärtskompatibilität macht es notwendig, daß die Zielsprache bestehende SQL-Skripts verarbeiten kann. Aufgrund der Natur von SQL als Datenbanksprache können diese auch große Mengen an Daten enthalten. Dies ist im Hinblick auf die Konzeption des Übersetzers als Multi-Pass-Übersetzer problematisch, da hier das Quellprogramm in eine Baumstruktur – den AST – überführt wird, die, hinsichtlich ihres Speicherbedarfes, ein Vielfaches der Größe des Quellprogrammes erreichen kann. Gleiches gilt für die Struktur, die während der Codeerzeugungsphase generiert wird.

Aus diesem Grund wird ein inkrementeller Übersetzungsmodus unterstützt. In diesem Modus werden die zwei Übersetzungsschritte (siehe Abbildung 5.2) nicht nacheinander für jeweils das gesamte Quellprogramm ausgeführt, sondern im Wechsel für jeweils einen

vollständigen Befehl. Durch diese Vorgehensweise kann der Speicherbedarf des Übersetzers erheblich reduziert werden, da jeweils nur ein Teil des Quellprogramms in den verschiedenen Zwischendarstellungen im Speicher vorgehalten werden muß.

Der Parser, der für die Syntaxanalyse zuständig ist, muß daher die Möglichkeit bieten, die Analyse für nur einen Teil eines Programms durchzuführen. Die von ANTLR generierten Parser verwenden die Methode des rekursiven Abstiegs. Entsprechend wird für jede Produktion einer Grammatik eine Funktion bzw. Methode generiert, die separat aufgerufen werden kann. Die Grammatik der Zielsprache kann daher so konstruiert werden, daß diese eine Produktion enthält, mit welcher genau ein Befehl analysiert wird. Diese Produktion wird Teil der Schnittstelle des generierten Parsers und kann daher direkt aufgerufen werden, sodaß hiermit die Voraussetzungen für die inkrementelle Übersetzung gegeben sind.

Ebenso ist jedoch auch die Unterstützung der Maschine notwendig, die in der Lage sein muß, verschiedene Befehle separat aber in einem gemeinsamen Kontext ausführen zu können. Diese Notwendigkeit besteht allerdings bereits durch die in Abschnitt 2.2.4 gestellten Forderung nach Unterstützung interaktiver Ausführung. Abschnitt 5.3.8 geht hierauf näher ein.

Kontextsensitivität der lexikalischen Analyse

Als Konsequenz der Forderung nach Datenbankunabhängigkeit werden, wie in Abschnitt 2.3.2 beschrieben, verschiedene syntaktische Variationen von Zeichenkettenliteralen und Kommentaren in Abhängigkeit vom gewählten SQL-Dialekt unterstützt. Mithilfe von Parse-Direktiven (siehe Abschnitt 4.11) kann dabei der SQL-Dialekt zur Übersetzungszeit gewechselt werden. Desweiteren unterscheidet die Sprache zwischen Zeichenkettenliteralen in Skript- und SQL-Befehlen, wobei die Syntax von Zeichenketteniteralen in Skriptbefehlen nicht vom gewählten SQL-Dialekt abhängig ist.

Dies hat zur Folge, daß während der lexikalischen Analyse für eine identische Eingabe je nach Kontext unterschiedliche Symbole generiert werden müssen, wobei der Kontext durch den Parser bestimmt wird. Listing 5.1 zeigt ein Beispiel. Dadurch kann die lexikalische Analyse nicht unabhängig von der syntaktischen Analyse durchgeführt werden, sondern muß stattdessen schrittweise, synchron mit dieser, vorgenommen werden.

Hieraus ergibt sich ein geringfügiges Problem bei der Verwendung von ANTLR. Dessen Lexer, der als Basis für automatisch generierte Lexer für spezifische Grammatiken dient, arbeitet unabhängig vom Parser und zerlegt eine Eingabe in einem einzigen Schritt in eine Folge von Symbolen. Daher muß zunächst dieser Basislexer durch einen passenderen Lexer ersetzt werden, der die oben genannten Anforderungen erfüllt. Dies ist jedoch ohne weiteres möglich.

Konkret bedeutet dies, daß der Lexer ein Symbol erst dann erzeugen darf, wenn es vom Parser angefordert wird, wobei die Notwendigkeit hierzu auch für die im vorigen Abschnitt beschriebene inkrementelle Übersetzung gegeben ist.

Listing 5.1: Kontextabhängigkeit der lexikalischen Analyse

```
1  -- Im SQL-92 Modus keine spezielle Behandlung von Backslash-Sequenzen
2  -- in Zeichenkettenliteralen.
3
4  \set quotes=sql92;
5
6  select '\'; -- Symbole: "select" "'\'" ";"
7
8  -- Im MySQL-Modus Behandlung von Backslash-Sequenzen zum Beispiel
9  -- zum Maskieren von Begrenzerzeichen in Zeichenkettenliteralen.
10
11 \set quotes=mysql;
12
13 select '\';'; -- Symbole: "select" "'\';'" ";"
```

5.3 Maschine

Die Maschine ist für die Ausführung von Programmen der Zielsprache zuständig. Deren Konzeption wird in den folgenden Abschnitten erörtert.

5.3.1 Hostsystem

Als Hostsystem für die Zielsprache bzw. den Interpreter soll die Java-Plattform dienen. Diese bietet mit der *Java Database Connectivity (JDBC)*[18] API eine umfangreiche Abstraktionsschicht, welche den Zugriff auf verschiedenste relationale Datenbanksysteme über eine einheitliche Programmierschnittstelle ermöglicht. Diese Schnittstelle ist insoweit modular aufgebaut, als daß die Unterstützung spezifischer Datenbanksysteme über austauschbare Treiber realisiert ist. Zum Zeitpunkt des Schreibens dieser Arbeit umfaßte die offizielle Treiberdatenbank[19], in welcher bekannte Treiber für verschiedene Datenbanksysteme registriert sind, 221 Einträge. Die Forderung nach Datenbankunabhängigkeit (siehe Abschnitt 2.2.5) kann damit also erfüllt werden.

Ein weiterer Vorteil von Java ist dessen Plattformunabhängigkeit. Durch diese kann der Einsatz der Zielsprache auf allen Plattformen ermöglicht werden, auf welchen Java-Programme lauffähig sind. Da Java bereits von der jeweils zugrundeliegenden Plattform abstrahiert, entsteht entsprechend bei der Entwicklung der Zielsprache kein signifikanter Mehraufwand für deren Portabilität. Dadurch kann auf einfache Art und Weise eine weite Verbreitung der Zielsprache ermöglicht werden, was sich positiv im Hinblick auf die Forderung nach dem Erreichen einer großen Nutzerbasis (siehe Abschnitt 2.2.5) auswirkt.

[18] http://java.sun.com/javase/technologies/database/index.jsp
[19] http://devapp.sun.com/product/jdbc/drivers

5.3.2 Grundlegender Aufbau

Die Konzeption der Maschine erfolgt auf Basis des in Abschnitt 3.9.2 beschriebenen registerbasierten Continuation-Passing-Interpreters. Hieraus ergibt sich eine recht einfache Struktur der Maschine. Die folgenden Objekte werden von dieser verwendet:

Befehl – Der auszuführende Befehl bzw. der zu evaluierende Ausdruck.

Wert – Der Wert den die Evaluierung des letzten Ausdrucks ergeben hat.

Continuation – Die Repräsentation der restlichen Berechnungen des ausgeführten Programms.

Fortsetzung – Ein Flag, welches angibt, ob im nächsten Berechnungsschritt ein Befehl oder eine Continuation verarbeitet werden soll.

Umgebung – Eine Repräsentation der im aktuellen Gültigkeitsbereich vorhandenen Bindungen.

Kontext – Ein Objekt, welches Informationen zum ausgeführten Programm speichert – u.a. Dateiname, globale Umgebung und Objekte.

Auf einige dieser Objekte gehen die folgenden Abschnitte näher ein.

Befehl

Befehlsobjekte werden während der Codeerzeugungsphase des Übersetzers gebildet und stellen die Eingabe der Maschine dar, die das zu interpretierende Programm repräsentiert. Diese Objekte reflektieren in hohem Maße die Struktur des Quellprogramms.

Wert

Jeder Wert in der Zielsprache ist ein Objekt. Auf deren Modellierung geht Abschnitt 5.3.4 genauer ein.

Continuation

Die Continuation wird in der Maschine als Stapel von Continuation-Objekten repräsentiert. Dies ermöglicht eine effiziente Manipulation der Continuation, da spezifische Continuation-Objekte direkt adressiert werden können, wie es zum Beispiel für die Implementierung der Block-Closures relevant ist.

Umgebung

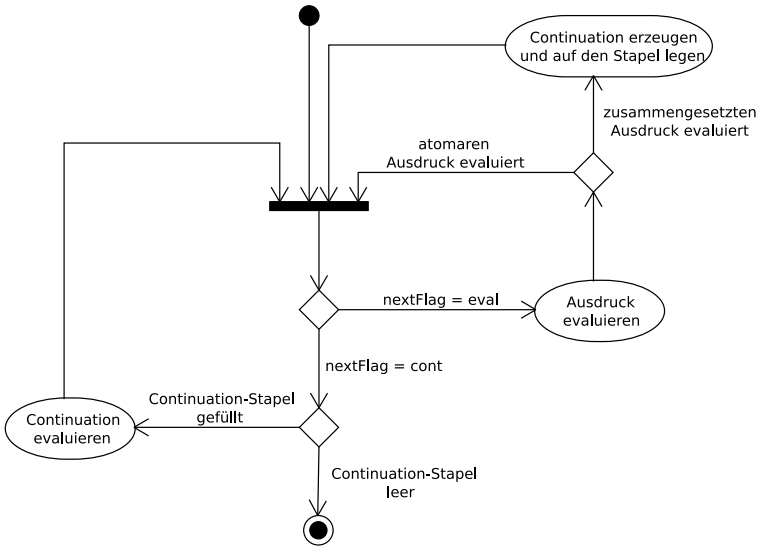
Die Umgebung wird, wie in Abschnitt 3.5 erläutert, unter Verwendung des Konzeptes der lexikalischen Adressierung modelliert. Die Struktur des Umgebungsobjektes entspricht daher der dort beschriebenen. Für einige der in den folgenden Abschnitten behandelten

Funktionalitäten ist jedoch die lexikalische Adressierung aufgrund ihrer statischen Natur unzureichend. Daher wird in der Zielsprache ein weiteres Konzept verwendet, das eine gewisse Dynamisierung zuläßt. Abschnitt 5.3.7 geht darauf genauer ein.

5.3.3 Ablauf der Interpretation

Der Ablauf der Interpretation entspricht grundlegend der in Abschnitt 3.9.2 beschriebenen registerbasierten Interpretation im Continuation-Passing-Style. Abbildung 5.3 zeigt eine schematisierte Darstellung dieses Ablaufs.

Abbildung 5.3: Ablauf der Interpretation



5.3.4 Repräsentation von Objekten der Zielsprache

Objekte der Zielsprache werden primär durch assoziative Arrays repräsentiert, die die Slots der Objekte beinhalten. Unterschieden wird hier zwischen Objekten, die von Programmen der Zielsprache definiert werden und Objekten, die die Zielsprache selbst definiert. Letztere, im Folgenden als *native Objekte* bezeichnet, implementieren Basisfunktionalitäten, welche von Programmen der Zielsprache verwendet werden können.

Über diese nativen Objekte wird ein Großteil der Funktionalität der Zielsprache realisiert. Dies umfaßt auch die Funktionalität von Kontrollstrukturen wie Verzweigungen und Ausnahmebehandlungen. Dabei kann größtenteils auf Funktionen der Hostsprache zurückgegriffen werden, wodurch die Komplexität der Maschine stark reduziert werden kann.

Neben der Bereitstellung von Basisfunktionalität kapseln native Objekte auch Werte und Strukturen der Hostsprache, wie zum Beispiel numerische Werte, Zeichenketten, Arrays usw., um diese den Programmen der Zielsprache zugänglich zu machen.

Interaktion nativer Objekte mit der Maschine

Die Implementierung von Kontrollstrukturen durch die nativen Objekte macht die Möglichkeit zur Interaktion zwischen diesen Objekten und der Maschine erforderlich. So muß es für native Objekte möglich sein, die Maschine zur Ausführung von Block-Closures oder Funktionen veranlassen zu können, um zum Beispiel bedingte Verzweigungen realisieren zu können. Hierfür stellt die Maschine eine entsprechende Schnittstelle, im folgenden auch als native Schnittstelle bezeichnet, bereit.

Instanziierung nativer Objekte

Bei der Instanziierung nativer Objekte ist zu beachten, daß diese in der Regel Werte der Hostsprache kapseln. Um zu verhindern, daß derartige Objekte in einem ungültigen Zustand existieren, werden die gekapselten Werte über deren Konstruktor initialisiert.

Hierfür stellen native Objekte eine enstsprechende Methode bereit, die im Zuge der `new`-Operation (siehe Abschnitt 4.7.3) aufgerufen wird. Deren Rückgabewert ist ein neues, initialisiertes, natives Objekt.

Dieses einfache Schema hat den Nachteil, daß es die Vererbung von nativen Objekten verhindert. Da hiervon jedoch die Vererbung unter nicht-nativen Objekten nicht betroffen ist, wird dies zunächst in Kauf genommen.

5.3.5 Repräsentation numerischer Werte

Da der Umgang mit numerischen Werten bei der Programmierung allgemein sowie bei der Arbeit mit Datenbanken im Besonderen von hoher Bedeutung ist, geht dieser Abschnitt gesondert auf die Repräsentation numerischer Werte in der Zielsprache ein.

Das Hostsystem Java bietet eine vielzahl numerischer Datentypen, wie Tabelle 5.1 zeigt. Um die Komplexität des Umgangs mit numerischen Werten in der Zielsprache zu verringern wäre ein transparentes Typsystem zumindest für ganzzahlige Typen, bei denen kein Informationsverlust auftritt, ideal. Dies könnte mithilfe automatischer Konvertierung numerischer Werte in den für deren Repräsentation notwendigen Typen geschehen. Hierfür wäre jedoch eine Overflowerkennung notwendig, durch welche festgestellt werden kann, ob das Ergebnis einer arithmetischen Operation im verwendeten Typen darstellbar ist. Diese wird jedoch für ganzzahlige Typen von Java nicht unterstützt, sodaß diese manuell durchgeführt werden müsste. Hierfür existieren Ansätze (z.B. [ADB08]) die jedoch zunächst nicht verfolgt werden sollen, im weiteren Verlauf der Entwicklung jedoch wieder aufgegriffen werden können.

	Bezeichnung	Speicherbedarf	Größe bzw. Präzision
Ganze Zahlen	byte	8 Bit	begrenzt
	short	16 Bit	begrenzt
	int	32 Bit	begrenzt
	long	64 Bit	begrenzt
	BigInteger	variabel	beliebig
Reelle Zahlen	float	32 Bit	begrenzt
	double	64 Bit	begrenzt
	BigDecimal	variabel	beliebig

Tabelle 5.1: Numerische Datentypen in Java

Um dennoch eine relativ geringe Komplexität im Umgang mit numerischen Werten zu ermöglichen, werden die in der Zielsprache verwendeten numerischen Typen auf jeweils einen Repräsentanten der beiden Zahlenbereiche mit begrenzter sowie beliebiger Präzision beschränkt.

Arithmetische Operationen auf Werten verschiedener numerischer Typen werden unterstützt. Dabei wird eine Angleichung der Typen der involvierten Werte vorgenommen, wobei diese jeweils derartig durchgeführt wird, daß kein Informationsverlust auftritt. In den Datentypen von Java ausgedrückt, wäre beispielsweise das Ergebnis einer Multiplikation eines *long*-Wertes mit einem *BigInteger*-Wert ein *BigDecimal*-Wert.

Da die Division von reellen Zahlen beliebiger Präzision Ergebnisse liefern kann, welche mit endlichem Speicherbedarf nicht darstellbar sind (zum Beispiel 1/3) wird in derartigen Situationen die Präzision auf einen definierten Wert begrenzt.

Ansonsten wird beim Umgang mit numerischen Werten die Semantik der Hostsprache beibehalten.

5.3.6 Einbindung externer Skripte

Um die Einbindung bestehender Skripte in ein Programm der Zielsprache zu ermöglichen, stellt die Maschine einen entsprechenden Mechanismus bereit. Dieser wird nicht für Zwecke der Modularisierung konzipiert, sondern ausschließlich, um eine einfache Integration bestehender Skripte zu gewährleisten. Das in der Anwendung simpelste Konzept dürfte hier das Konzept der Include-Files sein, bei denen die Einbindung eines externen Skriptes im Ergebnis äquivalent ist mit der Ersetzung der Einbindungsanweisung durch den Inhalt des externen Skriptes.

Für eine akzeptable Flexibilität dieses Mechanismus muß die Einbindung eines externen

Skriptes zur Laufzeit erfolgen, sodaß ein konkretes einzubindendes Skript dynamisch spezifiziert werden kann.

Durch diese Rahmenbedingungen ergeben sich die folgenden Konsequenzen:

1. Der Übersetzer muß die Möglichkeit bieten, die Übersetzung eines Programmes mit einer gegebenen Symboltabelle vorzunehmen. Diese Notwendigkeit besteht bereits durch die Semantik der Sprache, die spezifiziert, daß eine bestimmte Menge von Objekten standardmäßig zur Verfügung stehen.

2. Die Maschine muß in der Lage sein, zu jeder Umgebung, die korrespondierende Symboltabelle zu erhalten, um diese für die Übersetzung des einzubindenen Skriptes verwenden zu können.

3. Die Maschine muß eine Schnittstelle bieten, über welche die Ausführung eines übersetzten Skriptes veranlaßt werden kann. Die Notwendigkeit hierfür besteht ohnehin. Hinzu kommt nur, daß diese Schnittstelle auch nativen Objekten zugänglich gemacht werden muß.

4. Die Maschine muß eine Schnittstelle bieten, über welche die Umgebung manipuliert werden kann. Dies ist notwendig, um Variablen, welche im eingebundenen Skript deklariert wurden, dem einbindenden Skript zugänglich machen zu können. Dies ist ebenfalls für den in Abschnitt 5.4.1 beschriebenen Import von Modulen des Hostsystems notwendig.

5. Die Umgebung muß so konzipiert sein, daß diese, neben der statischen, auch dynamische Bindung unterstützt. Auch dies ist bereits für den Import von Modulen des Hostsystems notwendig. Abschnitt 5.3.7 geht genauer darauf ein.

Es zeigt sich, daß nur Punkt 2 speziell für die Einbindung externer Skripte notwendig ist und daher gesondert behandelt werden muß. Die restlichen Punkte werden jeweils an anderer Stelle behandelt.

Da Umgebung und Symboltabelle sich strukturell stark ähneln ist eine Übersetzung leicht möglich. Das Umgebungsobjekt wird daher mit einer entsprechende Methode versehen, welche diese vornimmt.

5.3.7 Unterstützung dynamischer Bindung

Als dynamische Bindungen werden hier Bindungen bezeichnet, die zur Laufzeit eines Programmes der Zielsprache erzeugt werden. Diese sind erforderlich, wenn die einem Namen zugehörige Bindung nicht zum Zeitpunkt der Übersetzung eines Programmes bekannt ist, wie es u.a. beim verwendeten Mechanismus für die Einbindung externer Skripte der Fall ist (Abschnitt 5.3.6).

Wird zum Beispiel ein Skript eingebunden, welches eine Variable deklariert, die zu einem späteren Zeitpunkt vom einbindenden Skript verwendet wird, so kann dieser Variablenreferenz während der Übersetzung des einbindenden Skriptes keine Deklaration zugeordnet werden. Durch die Deklaration der Variablen im eingebundenden Skript wird diese Referenz dennoch zur Laufzeit gültig. Die Konsequenz hieraus ist, daß die Gültigkeit von Variablenreferenzen generell nur zur Laufzeit geprüft werden kann. Listing 5.2 verdeutlicht dies an einem Beispiel.

Listing 5.2: Dynamische Bindung bei Einbindung externer Skripte

```
1  -- including.sql: Das einbindende Skript.
2
3  var greeting := 'Good morning.';
4
5  -- Einbinden des externen Skriptes.
6  .includeFile('included.sql');
7
8  -- Referenzierung der im eingebundenen Skript
9  -- deklarierten Variable "world".
10 -- Ausgabe: Hello World!
11 .print('Hello ' + world);
12
13 -- Referenzierung der in beiden Skripten deklarierten
14 -- Variablen "greeting".
15 -- Ausgabe: Good afternoon.
16 .print(greeting);
17
18 -- included.sql: Das eingebundene Skript.
19 var world := 'World!';
20 var greeting := 'Good afternoon.';
```

Durch die Verwendung lexikalischer Adressierung ergeben sich diesbezüglich jedoch zwei Probleme:

1. Lexikalische Adressen können nur für Variablen gebildet werden, deren Deklaration zur Übersetzungszeit bekannt ist. Entsprechend können diese nicht für das Auffinden der Bindung eines Namens herangezogen werden, wenn dessen Deklaration zur Laufzeit erfolgt.

2. Zur Laufzeit muß die dynamische Bindung zu einer Variablen verwendet werden auch wenn diese explizit deklariert wurde. Vorraussetzung hierfür ist, daß die dynamische Bindung einer Deklaration im Sinne der *am engsten umgebenden Deklaration*-Regel (siehe Abschnitt 3.5) entspricht.

Diese Probleme werden durch die Erweiterung des Konzeptes der Umgebungen um dynamische Umgebungen adressiert. Diese besitzen wie die statischen Umgebungen einen Zugriffsverweis auf deren Elternumgebung. Im Unterschied zu diesen verwenden die jedoch die Namen von Variablen statt deren lexikalischer Adressen für das Auffinden von

Bindungen wodurch Punkt 1 adressiert wird. Punkt 2 wird realisiert, indem die Suche nach der Bindung einer Variablen immer in der dynamischen Umgebung durchgeführt wird, es sei denn bei der Variablen handelt es sich um eine in der (statischen) Elternumgebung explizit deklarierte Variable deren Deklaration nach der Aktivierung der dynamischen Umgebung stattgefunden hat. Wurde eine entsprechende Bindung nicht in der dynamischen Umgebung gefunden, wird die Suche immer in der Elternumgebung fortgesetzt.

Zusätzlich wird sichergestellt, daß die Suche nach Bindungen von implizit deklarierten Variablen in statischen Umgebungen immer in der jeweiligen Elternumgebung durchgeführt wird. Dadurch wird erreicht, daß die Bindung gefunden werden kann, auch wenn es sich um eine nicht-lokale Variable handelt. Wird keine Bindung gefunden, ist also keine Elternumgebung mehr vorhanden in der die Suche fortgesetzt werden kann, kann die Maschine eine entsprechende Warnung generieren.

5.3.8 Interaktive Ausführung

Die interaktive Ausführung macht es erforderlich, daß die Maschine mehrere Befehle eines Programmes unabhängig voneinander, aber in einem gemeinsamen Kontext ausführen kann. In der Maschine der Zielsprache betrifft dies die Umgebung, die über die Ausführung verschiedener Befehle hinweg erhalten bleiben muß. Die Maschine gewährleistet dies, indem diese eine Kontextobjekt verwendet, welches die globale Umgebung eines Progammes referenziert. Dieses Kontextobjekt kann in verschiedenen Befehlsevaluierungen einer Maschineninstanz oder auch in verschiedenen Maschineninstanzen wiederverwendet werden.

5.4 Integration von Skriptsprache, Interpreter und Hostsystem

Die Hostsprache Java bietet eine umfangreiche Palette an Bibliotheken und Programmierschnittstellen für ein breites Spektrum von Anwendungsfällen. Eine gute Integration der Hostsprache in die Zielsprache ist daher von großem Vorteil, da hierdurch die bereits bestehenden Bibliotheken in der Zielsprache nutzbar gemacht werden können. Die Notwendigkeit, eigene Bibliotheken für die Zielsprache zu entwickeln kann dadurch stark reduziert werden.

Relevante Aspekte bei der Integration der Hostsprache sind dabei der Import von Bibliotheken und Klassen in den Namensraum der Zielsprache, das Erzeugen von Objekten der Hostsprache, das Aktivieren von Methoden dieser Objekte sowie die Abbildung von Typen der Hostsprache auf Typen der Zielsprache und umgekehrt.

Diese Aspekte werden in den folgenden Abschnitten betrachtet.

5.4.1 Import von Modulen der Hostsprache

Der Import von Modulen der Hostsprache erfolgt zur Laufzeit, da zum einen diese Module nicht zwingend zum Zeitpunkt der Übersetzung zur Verfügung stehen müssen, zum anderen da hierdurch eine gute Flexibilität erreicht werden kann, indem zu importierende Module dynamisch zur Laufzeit spezifiziert werden können.

Im Kontext der Hostsprache Java bezeichnen Module Klassen sowie Pakete (bzw. Packages), wobei der Import eines Paketes den Import aller darin befindlicher Klassen bewirkt.

Import von Klassen

Der Import von Klassen gestaltet sich recht einfach. Diese werden in der Zielsprache durch ein spezielles natives Objekt repräsentiert, welche die jeweiligen Java-Klassen kapseln. Der Import einer Klasse wird dabei durch eine von der Zielsprache bereitgestellten Methode realisiert, deren Rückgabewert die gekapselte Klasse ist. Dadurch können Namenskollisionen vermieden werden, indem die Bindung der Klasse durch eine Zuweisungsoperation an einen beliebigen Namen der Zielsprache erfolgen kann.

Listing 5.3: Import von Klassen

```
1  -- Import einer Klasse als Zuweisung.
2  var Date := importClass('java.util.Date');
3  -- Aequivalenter Import in Befehlsform wird unterstuetzt.
4  import java.util.Date;
5
6  -- Aliasing einer importierten Klasse durch Verwendung
7  -- beliebiger Variablennamen moeglich.
8  var MyDate := importClass('java.util.Date');
9  -- Aequivalenter Import in Befehlsform:
10 import java.util.Date as MyDate;
```

Import von Paketen

Der Import von Paketen ist ein etwas umfangreicherer Vorgang. Zum einen müssen hier mehrere Bindungen in einer Operation erzeugt werden – eine Bindung für jede Klasse eines Paketes – zum anderen sind die zu bindenden Namen weder zum Zeitpunkt der Übersetzung noch zum Zeitpunkt des Imports bekannt. Ersteres aufgrund der dynamischen Natur des Imports, Letzeres da systembedingt keine Möglichkeit existiert, über alle Klassen eines Paketes zu iterieren.[20]

[20] Java erlaubt die Definition eigener ClassLoader – Komponenten, welche für das Laden von kompilierten Klassen zuständig sind – und sieht dabei die Möglichkeit, Klassen über unterschiedliche Medien wie Netzwerkserver zu laden, explizit vor. Viele Medien unterstützen jedoch das Auflisten enthalte-

Dementsprechend kann der Import von Paketen nicht als Zuweisungsoperation modelliert werden. Stattdessen wird hier die dynamische Bindung (siehe Abschnitt 5.3.7) verwendet, indem der Vorgang des Imports durch die Installation einer dynamischen Umgebung modelliert wird.

Da die Klassen eines importierten Paketes wie bereits erwähnt zum Zeitpunkt des Imports nicht bekannt sind, wird hier eine spezielle Ausprägung der dynamischen Umgebung verwendet. Diese verwendet einen Namensauflöser, um Namen zu Werten zuzuordnen. Der Namensauflöser prüft dabei, ob ein Name dem einer Klasse im importierten Paket entspricht und bindet diese, gekapselt in ein Objekt der Zielsprache, an den jeweiligen Namen.

Listing 5.4: Import von Paketen

```
1 -- Import aller Klassen im Paket java.util.
2 .importPackage('java.util');
3 -- Aequivalenter Import in Befehlsform wird unterstuetzt.
4 import java.util.*;
5
6 -- Ausgabe der aktuellen Zeit mithilfe der importierten
7 -- Klasse java.util.Calendar.
8 .print('Now: ' + Calendar.instance.timeInMillis);
```

5.4.2 Objekterzeugung

In der Zielsprache kann die Erzeugung von Objekten der Hostsprache auf zwei Wegen erfolgen. Zum einen durch Instanziierung von Klassen der Hostsprache, also durch das Aufrufen von deren Konstruktor, zum anderen durch das Aufrufen von statischen Methoden der Klassen der Hostsprache, sofern diese einen Rückgabewert liefern.

Die Instanziierung kann dabei in der Zielsprache durch den Aufruf der `new`-Operation auf ein Objekt, welches eine Klasse des Hostsystems kapselt, veranlasst werden. Diese Objekte verfügen über einen nativen Konstruktor (siehe Abschnitt 5.3.4), der im Zuge dieser Operation aktiviert wird. Dieser native Konstruktor ermittelt einen zu den übergebenen Argumenten passenden Konstruktor der gekapselten Klasse über die von Java bereitgestellte *Reflection-API*, aktiviert diesen und liefert das erzeugte Objekt.

Der Aufruf von statischen Methoden der Klassen der Hostsprache erfolgt in der Zielsprache durch Senden einer Nachricht an das eine Klasse kapselnde Objekt. Die Antwort auf diese Nachricht ist ein Objekt der Zielsprache, welches die dem Namen der Nachricht entsprechenden Methoden kapselt. Dieses Objekt kann wie Funktions- und Block-Closure-Objekte aktiviert werden, wodurch eine den übergebenen Argumenten entspre-

ner Ressourcen nicht, sodaß die entsprechenden Schnittstellen dies ebenso nicht vorsehen. (siehe auch `http://java.sun.com/javase/6/docs/api/java/lang/ClassLoader.html`)

chende Methode ausgeführt und deren Rückgabewert zurückgeliefert wird. Der folgende Abschnitt geht genauer auf diese Thematik ein.

5.4.3 Aktivierung von Methoden der Hostsprache

Die Aktivierung von Methoden der Hostsprache erfolgt wie im vorigen Abschnitt beschrieben für statische und nicht-statische Methoden auf die gleiche Art und Weise durch Senden einer Nachricht an ein Objekt, dessen Antwort ein Methodenobjekt ist. Der Unterschied ist nur, daß das Objekt in dessen Kontext eine Methode aktiviert wird, einmal eine Klasse ist, einmal eine Instanz.

Die Hostsprache Java unterstützt das Überladen von Methoden. Das heißt sie werden über ihre Signatur eindeutig bestimmt, die aus dem Methodennamen und den Typen sowie der Anzahl ihrer Argument besteht (vgl. [GJSB00, Kap. 8.4.2]). Durch das alleinige Senden einer Nachricht an ein gekapseltes Java-Objekt kann daher die zugehörige Methode nicht eindeutig bestimmt werden. Entsprechend kann das als Antwort auf die Nachricht gelieferte Methodenobjekt daher mehrere Methoden desselben Namens kapseln. Die Auswahl der konkreten Methode erfolgt bei der Aktivierung des Methodenobjektes anhand der übergebenen Argumente.

Um die Interaktion mit Objekten der Hostsprache möglichst intuitiv zu gestalten, sollte bei diesem, *Overload Resolution* genannten, Vorgang idealerweise in jeder Situation die gleiche Auswahl getroffen werden, die von der Hostsprache getroffen werden würde. Dies ist jedoch aus den im Folgenden dargelegten Gründen nicht möglich.

Zum einen verwendet Java den Single-Dispatch-Mechanismus , um die zu einer Nachricht gehörige Methode zu bestimmen. Dabei wird zur Laufzeit nur der Typ des Empfängerobjektes verwendet, die Typen der Argumente hingegen zur Übersetzungszeit. Im Unterschied dazu muß die Zielsprache aufgrund ihrer untypisierten Natur den Multiple-Dispatch-Mechanismusimplementieren. Hierbei werden sowohl der Typ des Empfängerobjektes als auch die Typen der Argumente benutzt, um die zu aktivierende Methode zu bestimmen. Die Listings 5.5 und 5.6 verdeutlichen den Unterschied jeweils an einem Beispiel.

Hinzu kommt, daß eine Diskrepanz zwischen den von der Zielsprache verwendeten Typen und jenen der Hostsprache besteht. So bietet letztere mehrere verschiedene Typen für die Repräsentation von ganzzahligen Werten und Gleitkommazahlen. Die Zielsprache bietet hingegen nur jeweils einen Typen (siehe Abschnitt 5.3.5).

Entsprechend kann das Ziel des von der zu entwickelnden Sprache verwendete Dispatch-Mechanismus nur eine möglichst gute Approximation des von Java verwendeten Mechanismus darstellen. Die Qualität wird dabei bestimmt von der Funktion über welche die Typen der Zielsprache auf Typen der Hostsprache abgebildet werden.

Situationen, in denen keine Methode eindeutig qualifiziert werden kann, sind jedoch nicht vermeidbar. Daher bietet die Zielsprache auch die Möglichkeit, die gewünschte Methode explizit zu spezifizieren. Hierfür stellt das Objekt, in welchem Methoden in der

Listing 5.5: Single Dispatch in Java

```
1  class A {
2  }
3
4  class B extends A {
5  }
6
7  class Printer {
8          public void print(A a) {
9                  System.out.println("A");
10         }
11
12         public void print(B b) {
13                 System.out.println("B");
14         }
15 }
16
17 class Test {
18         public static void main(String... args) {
19                 A a = new A();
20                 B b = new B();
21                 A c = new B();
22
23                 Printer printer = new Printer();
24                 p.print(a); // Ausgabe: A
25                 p.print(b); // Ausgabe: B
26                 p.print(c); // Ausgabe: A
27         }
28 }
```

Zielsprache gekapselt werden, eine Funktion bereit, welche die Selektion einer spezifischen Methode durch explizite Angabe der gewünschten Signatur erlaubt. Listing 5.7 demonstriert dies an einem Beispiel.

Einen Sonderfall stellt hier der Wert `null` der Hostsprache dar. Diesem kann zur Laufzeit kein Typ zugeordnet werden, was dazu führen kann, daß eine Methode nicht eindeutig qualifiziert werden kann, wenn eines der Argumente den Wert `null` hat. Dieser kann jedoch nur aus zwei Quellen stammen – zum einen aus Eigenschaften der Objekte der Hostsprache, zum anderen aus Rückgabewerten von Methoden der Hostsprache. Mit Ausnahme von Methoden, die keinen Wert liefern (Rückgabewert `void`), kann daher immer ein dem `null`-Wert zugeordneter Typ abgeleitet werden. Daher wird dieser in der Zielsprache durch ein eigenes Objekt repräsentiert, welches den jeweiligen Typen speichert, sodaß dieser bei der Overload Resolution verwendet werden kann.

Listing 5.6: Multiple Dispatch in der Zielsprache

```
1  -- Import der in Listing 5.5 definierten Klassen.
2  import test.*;
3
4  var a := new A();
5  var b := new B();
6  var c := new B();
7
8  var printer := new Printer();
9  .p.print(a); -- Ausgabe: A
10 .p.print(b); -- Ausgabe: B
11 .p.print(c); -- Ausgabe: B
```

Listing 5.7: Manuelle Overload Resolution

```
1  -- Import der in Listing 5.5 definierten Klassen.
2  import test.*;
3
4  var a := A();
5  var b := B();
6
7  var printer := new Printer();
8  var print := printer.print;
9
10 .print.select('A').call(printer, a); -- Ausgabe: A
11 .print.select('B').call(printer, b); -- Ausgabe: A
12 .print.select('B').call(printer, b); -- Ausgabe: B
```

5.5 Integration von SQL und Skriptsprache

Naturgemäß ist die Integration von SQL und Skriptsprache für die zu entwickelnde Sprache einer der bedeutendsten Punkte. Nachdem die syntaktischen Aspekte bereits in Kapitel 4.3 behandelt wurden, widmet sich dieser Abschnitt nun den semantischen Aspekten.

5.5.1 Ressourcenverwaltung

Hauptziel bei der Integration von SQL und Skriptsprache ist es, die Notwendigkeit der damit verbundenen Verwaltung von Ressourcen auf ein Minimum zu reduzieren, um den Umgang mit der Zielsprache möglichst einfach zu gestalten sowie den interaktiven Einsatz sinnvoll zu unterstützten.

Die Problematik beim Umgang mit den meisten Ressourcen liegt weniger in deren Erzeugung sondern vielmehr in deren korrekter Freigabe nach der Verwendung. Um eine diesbezügliche Vereifachung zu schaffen empfiehlt sich insbesondere für Sprachen, welche Closures unterstützen, ein *Loan Pattern*[21] genanntes Entwurfsmuster. Mit diesem

[21] `http://scala.sygneca.com/patterns/loan`

kann sichergestellt werden, daß eine Ressource korrekt freigegeben wird, indem sich eine spezialisierte Funktion um Erzeugung und Freigabe der Ressource kümmert. Einer übergebenen Closure wird die erzeugte Ressource dabei zur Verfügung gestellt. Nach der Terminierung der Closure erfolgt in jedem Fall die Freigabe. Auf diese Weise kann verhindert werden, daß die Freigabe von Ressourcen versäumt wird.

Da sich dieses Entwurfsmuster bereits in diverse Sprachen wie zum Beispiel *Ruby*, *Scala* oder *Groovy* insbesondere auch im Umgang mit Datenbanken bewährt hat, wird es auch in der Zielsprache angewandt.

5.5.2 Datenbankressourcen

Bei der Arbeit mit SQL sind die folgenden Ressourcen involviert:

- Datenbankverbindungen (Connections)
- SQL-Befehlsobjekte (Statements)
- Ergebnismengen (ResultSets)

Diese werden in den folgenden Abschnitten genauer behandelt.

Datenbankverbindungen

Die Verwaltung von Datenbankverbindungen wird in der Zielsprache von einem speziellen Objekt, dem *Connection Manager* vorgenommen. Dieser erlaubt das Erzeugen von Datenbankverbindungen, welche in der Zielsprache durch Objekte repräsentiert werden, die jeweils ein *JDBC*-Connection-Objekt kapseln und weitgehenden Zugriff auf dessen Funktionalität gewähren. Zusätzlich wird erweiterte Funktionalität für einen einfacheren Umgang mit Datenbankverbindungen bereitgestellt (siehe dazu Anhang A).

Eine der Hauptanforderungen an die Zielsprache ist die Unterstützung der Arbeit mit mehreren Datenbankverbindungen gleichzeitig (siehe Abschnitt 2.2.3). Da die Zielsprache die Möglichkeit vorsieht, SQL-Befehle ohne die explizite Angabe der zu verwendenden Datenbankverbindung zu erzeugen, muß diese implizit erfolgen können. Hierfür wird das Konzept der aktiven Verbindung verwendet. Jeder erzeugte SQL-Befehl wird dabei in der jeweils aktiven Verbindung erzeugt.

Die Auswahl der aktiven Verbindung erfolgt auf drei Wegen:

1. Die erste, zur Laufzeit eines Skriptes erstellte Verbindung wird automatisch zur aktiven Verbindung.
2. Eine Verbindung kann über den *Connection Manager* global aktiviert werden.
3. Das Verbindungsobjekt implementiert hierfür den bereits beschriebenen Loan Pattern. Damit kann die entsprechende Verbindung innerhalb eines Blockes als aktiv

definiert werden. Beim Verlassen des Blockes wird die vorherige aktive Verbindung reaktiviert.

Um geöffnete Verbindungen automatisch beim Beenden eines ausgeführten Skriptes schließen zu können, werden diese bei ihrer Erzeugung immer im ConnectionManager registriert. Zum Ende der Ausführung wird dieser immer von der Maschine veranlaßt, alle offenen Verbindungen zu schließen.

SQL-Befehlsobjekte und Ergebnismengen

Der einfachste und wahrscheinlich häufigste Anwendungsfall beim Umgang mit SQL-Befehlen ist wahrscheinlich das einmalige Ausführen eines bestimmten, möglicherweise parametrisierten, SQL-Befehls und das anschließende Verarbeiten der Befehlsmenge. Um die Zielsprache für diesen Anwendungsfall zu optimieren wird die Ressourcenverwaltung von SQL-Befehlen und Ergebnismengen daraufhin ausgelegt.

Hierfür stellt das Befehlsobjekt Methoden zur Verfügung, welche die Ausführung des Befehls veranlassen, das Ergebnis des Befehls liefern und im Anschluß sowohl das Ergebnismengenobjekt, sofern vorhanden, als auch das Befehlsobjekt freigeben.

Für die unterschiedlichen Befehlsarten (siehe Abschnitt 2.4.2) werden hier unterschiedliche Methoden bereitgestellt. Für Datenbefehle zum Beispiel eine am Loan Pattern orientierte Methode für die Iteration über die Ergebnismenge.

Ein weiterer wichtiger Anwendungsfall ist das mehrfache Ausführen eines parametrisierten Befehls mit unterschiedlichen Parametern. Die Ressourcen derartiger Befehle dürfen natürlich nicht nach einer Ausführung freigegeben werden. Auch für diesen Anwendungsfall werden Methoden bereitgestellt, die nach dem Loan-Pattern arbeiten.

5.5.3 Transaktionen

Transaktionen werden in der Zielsprache ähnlich wie Ressourcen behandelt, indem diese über den Loan Pattern innerhalb eines Blockes aktiviert werden können. Methoden für die manuelle Transaktionsverwaltung werden ebenso bereitgestellt.

5.5.4 Batchausführung

Wie bereits in Abschnitt 2.4.2 erwähnt, existieren zwei Arten von Batchausführungen.

Stapel von nicht-parametrisierten Befehlen[22] werden wieder über den Loan Pattern durch das Verbindungsobjekt realisiert. Stapel eines parametrisierten Befehls werden hingegen über das SQL-Befehlsobjekt realisiert. Auch hier kommt der Loan Pattern zum Einsatz. Listing 5.8 demonstriert die beiden Varianten.

[22] Gemeint sind hier nicht-parametrisierte Befehle im Sinne von JDBC. Das heißt auch durch eingebettete Variablenreferenzen parametrisierte Befehle werden in diesem Kontext als nicht-parametrisiert angesehen.

Listing 5.8: Batchausführung von SQL-Befehlen

```
1  -- Stapel nicht-parametrisierter Befehle:
2
3  .ConnMgr.batch {=>
4          for (i : 1 .. 10) {
5                  sql insert into test values (@{i});
6          }
7  };
8
9  -- Stapel parametrisierter Befehle:
10
11 .(sql insert into test values (?)).batch { batch =>
12         1.to(10) { i => batch.add(i); };
13 };
```

5.5.5 Ausgabe der Ergebnisse von SQL-Befehlen

Bei der interaktiven Arbeit mit SQL-Interpretern ist generell die Präsentation der Ergebnisse ausgeführter SQL-Befehle unabdingbar. Entsprechende Möglichkeiten müssen daher auch in der Zielsprache vorgesehen werden.

Im Kontext der Zielsprache ist jedoch die Anzeige der Ergebnismenge eines jeden ausgeführten SQL-Befehls nicht sinnvoll ist. Ein Beispiel hierfür sind Befehle deren Ergebnismenge durch das ausführende Programm verwertet wird.

Als Kriterium für die Anzeige des Ergebnisses eines Befehls dient daher dessen Kontext. Für SQL-Befehle im Befehlskontext kann eindeutig definiert werden, daß deren Ergebnismenge vom Programm nicht verwertet wird, sodaß eine Anzeige während der interaktiven Ausführung erfolgen kann und soll. Bei SQL-Befehlen im Ausdruckskontext ist dies nicht der Fall, sodaß hier keine Anzeige der Ergebnisse erfolgt.

Da SQL-Befehle im Befehlskontext während der Übersetzung eines Programmes in einen Ausdruck umgewandelt werden, muß sichergestellt sein, daß dieser Ausdruck weiterhin als Befehl identifizierbar ist. Dies wird dadurch realisiert, daß der Ausdruck den Versand einer bestimmten Nachricht an das Erzeugte SQL-Befehlsobjekt darstellt. Auf diese Nachricht hin wird die Ausführung des SQL-Befehls sowie die anschließende Anzeige des Ergebnisses veranlaßt.

Um diese möglichst lose an die Maschine zu koppeln wird hier das Beobachtermuster verwendet. Hierfür stellt sie eine Schnittstelle bereit, über welche beliebige Beobachter registriert werden können. Die eigentliche Anzeige der Ergebnisse wird von diesen durchgeführt. Hieraus ergibt sich unter anderem auch der Vorteil, daß diese austauschbar sind und daher die Ausgabe der Ergebnisse in unterschiedlichen Formaten auf einfache Weise realisiert kann.

6 Realisierung

Dieses Kapitel beschreibt ausgewählte Aspekte einer prototypischen Implementierung der Zielsprache anhand des in den vorherigen Kapiteln entwickelten Konzeptes.

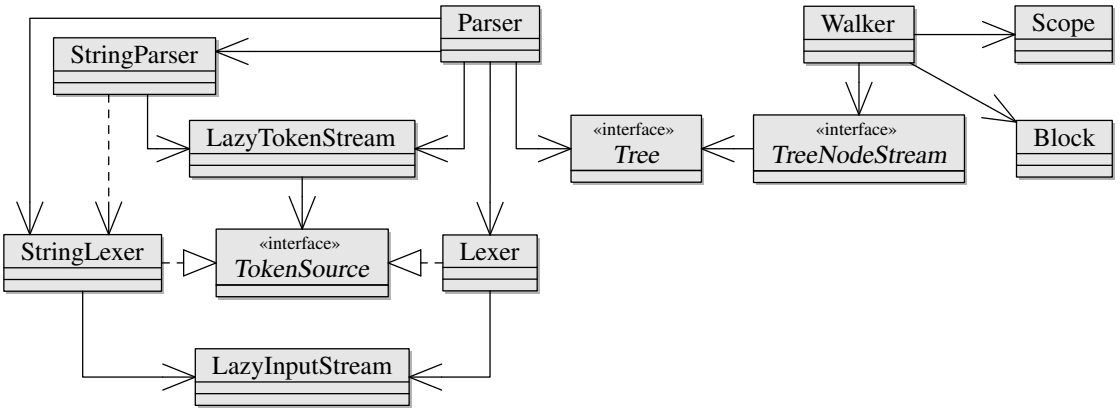
6.1 Ziel

Mit dem entwickelten Prototypen sollen die definierten Anforderungen grundlegend erfüllt werden. Ziel dieses Prototypen ist es, eine kritische Betrachtung des Gesamtkonzeptes auf Basis einer funktionierenden Implementierung zu ermöglichen. Desweiteren soll mit diesem die Evaluierung von gewählten Techniken im Kontext der Zielsprache ermöglicht werden, um die Richtung der weiteren Entwicklung aufzeigen zu können.

6.2 Übersetzer

Abbildung 6.1 zeigt ein grobes Klassendiagramm des Übersetzers. Aus Gründen der Übersichtlichkeit reflektiert dieses nicht jedes Detail des realisierten Prototypen sondern beschränkt sich auf dessen wesentlichen Elemente.

Abbildung 6.1: Übersetzer – Klassendiagramm



Wie in Abschnitt 5.2.3 beschrieben sind die Klassen *(String-)Lexer* bzw. *(String-)Parser* für die lexikalische respektive Syntaxanalyse verantwortlich. Der Zei-

chenstrom des Quellprogramms wird durch die Klasse *LazyInputStream* repräsentiert, welche der Lexer verwendet. Der Symbolstrom wird durch *LazyTokenStream* realisiert. Dieser liest einzelne Symbole von einer Symbolquelle, die durch den Lexer repräsentiert wird und liefert diese Symbole an den Parser. Die Lexer-Klassen implementieren jeweils das Interface *TokenSource*. Der vom Parser erzeugte abstrakte Syntaxbaum wird durch eine, das Interface *Tree* implementierende, Klasse realisiert, die wie das Interface durch die Laufzeitumgebung von ANTLR bereitgestellt wird. Gleiches gilt für das Interface *TokenSource*. Der Walker, repräsentiert durch die gleichnamige Klasse, realisiert wie erwähnt die Phasen der Semantikanalyse und der Codeerzeugung, wobei diese gleichzeitig durchgeführt werden. Bei der Semantikanalyse findet die Klasse *Scope* Verwendung, die die Symboltabelle repräsentiert. Das Ergebnis der Codeerzeugung ist ein Objekt der Klasse *Block*, welches eine Folge von Befehlen des Quellprogramms repräsentiert. Abschnitt 6.3 wird hierauf genauer eingehen.

Die grundlegende Struktur des Übersetzers ist damit in weiten Teilen auf das verwendete Werkzeug ANTLR zurückzuführen. Besonderheiten des Entwurfes werden in den folgenden Abschnitten erläutert.

6.2.1 Sonderbehandlung von Zeichenkettenliteralen

Aus dem gezeigten Klassendiagramm wird ersichtlich, daß zwei Lexer – *Lexer* und *StringLexer* – sowie zwei Parser – *Parser* und *StringParser* – am Vorgang der Übersetzung beteiligt sind. Wie in Abschnitt 5.2.3 beschrieben, realisieren diese die lexikalische sowie die Syntaxanalyse, also den ersten Schritt der Übersetzung. Parser und Lexer sind hierbei die Kombination von Klassen, welche von ANTLR aus einer gegebenen Grammatik generiert werden.

Im Falle der Zielsprache wurde im Prototypen deren Grammatik unterteilt in separate Grammatiken. Eine dieser Grammatiken beschreibt die Syntax der Zielsprache, die andere beschreibt die Syntax von Zeichenkettenliteralen.

Dieser Ansatz wurde gewählt da die Menge der unterstützten syntaktischen Formen von Zeichenkettenliteralen im Zusammenspiel mit der Unterstützung eingebetteter Variablenreferenzen zu einer relativ hohen Komplexität der Grammatik von Zeichenkettenliteralen führte. Von einer Trennung dieser Grammatik von der restlichen Grammatik der Zielsprache wurde eine Erhöhung der Übersichtlichkeit und damit auch der Wartbarkeit des Gesamtsystems erhofft.

Dem *Parser* sind hierbei nur die syntaktischen Elemente von Zeichenkettenliteralen bekannt, welche diese einleiten. Wird ein solches Element erkannt, wird die Kontrolle an den *StringParser* übergeben, welcher das Literal analysiert und in einen abstrakten Syntaxbaum überführt. Erreicht der *StringParser* das Ende eines Zeichenkettenliterals, übergibt er die Kontrolle zurück an den aufrufenden Parser, welcher schließlich den generierten AST in dessen AST einbindet.

Für diese Art der Realisierung ist die in Abschnitt 5.2.5 beschriebene inkrementelle

Übersetzung ebenfalls notwendig, da hier die lexikalische Analyse jeweils nur im Zuständigkeitsbereich des entsprechenden Parsers erfolgen darf. So darf beispielsweise der *StringLexer* die lexikalische Analyse der Eingabe nicht über das Ende eines Zeichenkettenliterals hinaus fortsetzen. Die diesbezüglichen Anpassungen werden in Abschnitt 6.2.3 beschrieben.

6.2.2 Realisierung von Parse-Direktiven

Die über Parse-Direktiven ermöglichte Aktivierung bzw. Deaktivierung der verschiedenen syntaktischen Formen von Zeichenkettenliteralen (siehe Abschnitt 4.11) wird im Prototypen mithilfe von semantischen Prädikaten realisiert.

Diese formulieren Bedingungen über welche der Parser bzw. Lexer angewiesen wird, Alternativen einer Produktion der Grammatik nur zu erwägen, falls die entsprechende Bedingung erfüllt ist. Die relevanten Produktionen der Grammatik der Zielsprache sind dabei mit Bedingungen versehen, die Bezug auf boolesche Variablen nehmen. Diese werden vom *Parser* im Zuge der Bearbeitung der entsprechenden Direktiven gesetzt, sodaß damit die Grammatik der Sprache während der Übersetzung beeinflusst werden kann.

6.2.3 Anpassungen im Bereich der lexikalischen Analyse

Wie in Abschnitt 5.2.5 beschrieben, bedingt die Syntax der Zielsprache, daß die lexikalische Analyse in Abhängigkeit vom Zustand des Parsers erfolgt. Zudem erfordert die Unterstützung der inkrementellen Übersetzung, daß die lexikalische Analyse nur so weit wie notwendig vorgenommen wird. Die entsprechende Notwendigkeit, diese nicht separat, sondern schrittweise durchzuführen, wenn der Parser ein Symbol vom Lexer anfordert, wird im Prototypen berücksichtigt. Dabei werden statt der von ANTLR bereitgestellten Standardimplementierungen der involvierten Komponenten eigene Implementierungen verwendet. Konkret sind dies die Klassen *LazyInputStream* sowie *LazyTokenStream*.

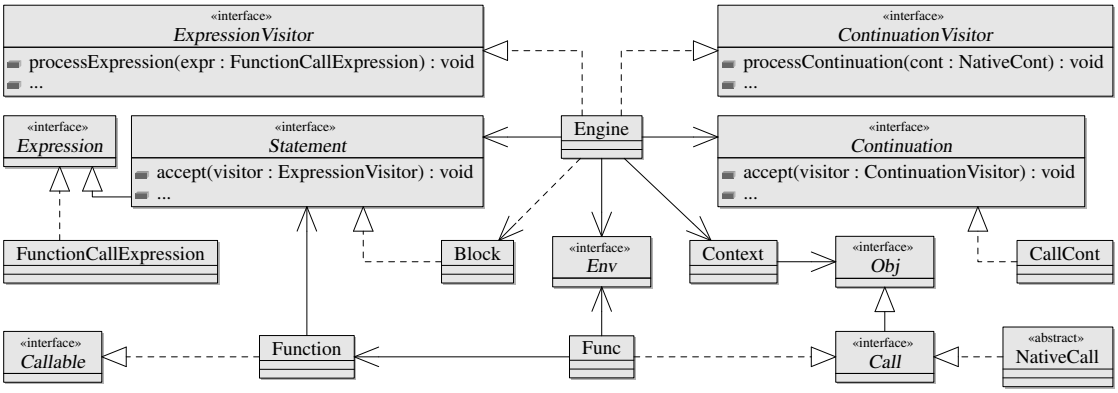
LazyInputStream implementiert das Interface *CharStream* und dient damit als Ersatz für den normalerweise verwendeten *ANTLRInputStream*. Dieser *LazyInputStream* wird von den Lexern als Zeichenstrom verwendet, der ein Zeichen erst dann vom Eingabestrom liest, wenn dieses vom Lexer angefordert wird. Diese Maßnahme ist nur für die inkrementelle Übersetzung relevant.

LazyTokenStream implementiert das Interface *TokenStream* welches bei ANTLR als Schnittstelle zwischen Parser und Lexer fungiert. Der normalerweise verwendete *CommonTokenStream* liest alle Symbole während der Initialisierung. Durch *LazyTokenStream* wird hingegen im Prototypen das verlangte Verhalten realisert, indem hier ein Symbol vom zugrundeliegenden Lexer nur im Rahmen der Lookahead- und Consume-Operationen angefordert wird, die der Parser bei der Navigation durch den Symbolstrom verwendet.

6.3 Maschine

Das in Abbildung 6.2 dargestellte Klassendiagramm zeigt die grundlegende Struktur der Maschine in einer aus Gründen der Übersichtlichkeit vereinfachten, unvollständigen Form. Die Klasse *Engine* repräsentiert dabei die Hauptkomponente, welche den eigentlichen Continuation-Passing-Interpreter implementiert. Weitere Komponenten sowie besonders relevante Aspekte der Implementierung werden in den folgenden Abschnitten erläutert.

Abbildung 6.2: Maschine – Klassendiagramm



6.3.1 Maschinencode

Die von der Maschine ausführbaren Befehlsobjekte (siehe Abschnitt 5.3.2) sind jeweils Instanzen von Klassen, die das Interface *Statement* implementieren, wobei als Eingabe für die Maschine Instanzen der Klasse *Block* dienen. Diese enthalten eine Liste von *Statement*-Objekten und reflektieren damit die grundlegende Struktur eines jeden Programmes der Zielsprache.

Diese *Statement*-Objekte werden von der Maschine evaluiert. Der bei der Evaluierung involvierte Code wird dabei an zentraler Stelle innerhalb der, die Maschine repräsentierenden, Klasse *Engine* definiert, wobei für jeden konkreten Befehl bzw. Ausdruck eine Methode existiert. Um dies auf sinnvolle Weise auch mit dem in Java verwendeten Single-Dispatch-Mechanismus (siehe auch Abschnitt 5.4.3) realisieren zu können, wird hierbei auf das Besuchermuster zurückgegriffen, womit ein dem Double-Dispatch-Mechanismus entsprechendes Verhalten erreicht werden kann. *Engine* fungiert dabei als Besucher, die verschiedenen konkreten *Statement*-Klassen als besuchte Elemente. Das gleiche Verfahren wird bei der Evaluierung von Continuation-Objekten verwendet.

Die Verwendung einer `switch`-Anweisung statt des Besuchermusters wurde in Betracht gezogen, jedoch aufgrund verschiedener Nachteile verworfen.

6.3.2 Aktivierungen

Da in der Zielsprache jeglicher Kontrollfluß durch Methodenaufrufe erfolgt, stellen Aktivierungen in der Maschine ein zentrales Thema dar, auf dessen Realisierung daher in diesem und dem folgenden Abschnitt näher eingegangen werden soll.

Die Machine kennt vier verschiedene Typen aktivierbarer Objekte:

1. Von Programmen der Zielsprache definierte aktivierbare Objekte:
   a) Funktionen
   b) Block-Closures
2. In der Hostsprache definierte aktivierbare Objekte:
   a) native Funktionen
   b) primitive Funktionen

Jeder dieser Typen wird in der Zielsprache durch ein eigenes Objekt repräsentiert, welches die interne Repräsentation des jeweiligen Typen kapselt.

6.3.3 Vorgang der Aktivierung

Aufrufe aktivierbarer Objekte in Programmen der Zielsprache werden allgemein durch Objekte der Klassen *FunctionCallExpression* bzw. *SlotCallExpression* repräsentiert, je nachdem ob der Aufruf einen Funktionsaufruf oder den Versand einer Nachricht darstellt.

Beide Objekte referenzieren dabei einen Ausdruck, der zu einem aktivierbaren Objekt evaluieren muß. Vor dessen Evaluierung wird ein Continuation-Objekt des Typs *CallCont* erzeugt über welches die Aktivierung nach erfolgter Evaluierung vorgenommen wird.

Die restlichen Schritte der Aktivierung sind jeweils spezifisch zum Typen des zu aktivierenden Objektes und werden in den folgenden Abschnitt beschrieben.

Aktivierung von Funktionen

Funktionsobjekte werden durch Evaluierung von Ausdrücken des Typs *FunctionDefinitionExpression* erzeugt. Dabei wird die aktuelle Umgebung gesichert und mit dem Funktionsobjekt assoziiert wodurch die Semantik der statischen Gültigkeitsbereiche realisiert wird.

Während der Aktivierung, im Rahmen der Evaluierung der *CallCont*, wird die aktuelle Umgebung gesichert sowie eine neue Umgebung erzeugt, die als Umgebung der Funktion fungiert. Die mit dem Funktionsobjekt assoziierte Umgebung wird dabei als Elternumgebung eingerichtet. Anschließend wird eine weitere Continuation vom Typ *CallArgCont* erzeugt, mit welcher die Aktivierung nach der Evaluierung der Funktionsargumente fortgesetzt wird.

Durch diese Continuation werden die evaluierten Funktionsargumente in der Umgebung der Funktion registriert und wiederum eine Continuation des Typs *FunRetCont* erzeugt. Diese sorgt nach Beendigung der Funktion für die Wiederherstellung der vor dem Aufruf der Funktion aktiven Umgebung. Anschließend wird die aktuelle Continuation als Closure-Home-Continuation in der Umgebung registriert, was für die Realisierung von Block-Closures (siehe folgender Abschnitt) von Bedeutung ist. Zuletzt wird die Evaluierung des Funktionsrumpfes veranlaßt.

Die Beendigung der Ausführung der aktivierten Funktion kann implizit nach der Evaluierung des letzten Befehls bzw. Ausdrucks des Funktionsrumpfes geschehen oder explizit durch Evaluierung des `return`-Befehls. In beiden Fällen wird die *FunRetCont* evaluiert. Bei Verwendung des `return`-Befehls wird hierfür zunächst die oberste auf dem Continuation-Stapel befindliche *FunRetCont* gesucht und alle weiter oben auf dem Stapel befindlichen Continuations verworfen.

Aktivierung von Block-Closures

Block-Closure-Objekte werden durch Evaluierung von Ausdrücken des Typs *BlockClosureExpression* erzeugt. Deren Aktivierung erfolgt größtenteils analog zur Aktivierung von Funktionsobjekten. Die Unterschiede beziehen sich auf die abweichende Semantik des `return`-Befehls und des Ausführungskontexts (siehe Abschnitt 4.6.2). Hier wird statt einer *FunRetCont*-Continuation eine des Typs *ClosRetCont* erzeugt, welche neben der gesicherten Umgebung auch mit dem erzeugten Block-Closure-Objekt assoziiert wird. Wird diese während der Evaluierung eines `return`-Befehls auf dem Continuation-Stapel gefunden, so wird die zu dem referenzierten Block-Closure-Objekt gehörende Closure-Home-Continuation über dessen gesicherte Umgebung geladen und durch Manipulation des Continuation-Stapels zur aktiven Continuation befördert. Dadurch wird der Rücksprung zu der, die Block-Closure definierenden, Funktion realisiert.

Aktivierung von nativen und primitiven Funktionen

Bei der Aktivierung von nativen Funktionen wird bereits während der Evaluierung der *CallCont*-Continuation ein anderer Pfad als bei Funktionen und Block-Closures eingeschlagen. Hier wird eine Continuation des Typs *NativeCont* erzeugt, die nach der Evaluierung der Argumente die gegebene native Funktion über eine Methode des von dieser implementierten Interfaces *NativeCall* aufruft.

Bei diesem Aufruf wird neben den Funktionsargumenten auch eine Referenz auf die ausführende Maschine übergeben, um der nativen Funktion die Interaktion mit der Maschine im Rahmen der nativen Schnittstelle (siehe Abschnitte 5.3.4 und 5.4.1) zu ermöglichen.

Primitive Funktionen werden durch eine Enumeration repräsentiert, die in der Maschine von einer einzelnen Methode evaluiert werden. In dieser wird mittels einer `switch`-Anweisungen zum zugehörigen Code verzweigt.

6.3.4 Ausführungskontext

Der Ausführungskontext von Funktionen (siehe Abschnitt 4.7.4) wird im Prototypen in der Umgebung registriert.

Das Schlüsselwort `this`, mit welchem der Ausführungskontext in Programmen der Zielsprache referenziert werden kann, wird hier von einem Ausdruck des Typs *ThisExpression* repräsentiert. Dessen Evaluierung liefert das registrierte Objekt. Das Senden einer Nachricht an `this` ist daher äquivalent zum Versand einer Nachricht an das Kontextobjekt.

Das Schlüsselwort `super` wird durch einen Ausdruck des Typs *SuperExpression* repräsentiert. Dieser wird zu dem Objekt evaluiert, welches den Parent des Objektes darstellt, in dem die aktuell ausgeführte Funktion gefunden wurde. Nachrichten die an `super` gesendet werden, also Funktionsaufrufe im `super`-Kontext werden vom Übersetzer als solche markiert, wodurch die Möglichkeit gegeben ist, diese in der Maschine gesondert zu behandeln. Diese Sonderbehandlung besteht darin, daß hier nicht das Empfängerobjekt einer Nachricht als Kontextobjekt in der Umgebung der Funktion registriert wird, sondern das Kontextobjekt der aktuellen Umgebung. Das Empfängerobjekt wird hingegen separat in der Umgebung registriert, wobei diese Referenz bei der Evaluierung der *SuperExpression* verwendet wird. Dadurch wird sichergestellt, daß Nachrichten an `super` jeweils an das nächsthöhere Objekt in der jeweiligen Objekthierarchie gesendet werden.

6.3.5 Native Schnittstelle

Die Interaktion von nativem Code mit der Maschine geschieht über die in Abschnitt 5.3.4 erwähnte native Schnittstelle. Über diese wird nativem Code unter anderem die Ausführung von aktivierbaren Objekten in der Machine ermöglicht. Die Ausführung kann dabei auf zwei Arten erfolgen. Zum einen kann die Ausführung veranlasst werden, ohne sie direkt vorzunehmen. Die Ausführung wird dabei erst nach Verlassen der nativen Funktion vorgenommen. Zum anderen kann die Ausführung direkt erfolgen, wobei das Ergebnis der Ausführung an den nativen Code zurückgegeben wird. Hierbei führt die Maschine eine schrittweise Evaluierung aus, die unterbrochen wird sobald die Ausführung des aktivierbaren Objektes beendet ist. Die Adresse der vor der Ausführung aktiven Continuation im Continuation-Stapel wird hierfür als Kriterium herangezogen.

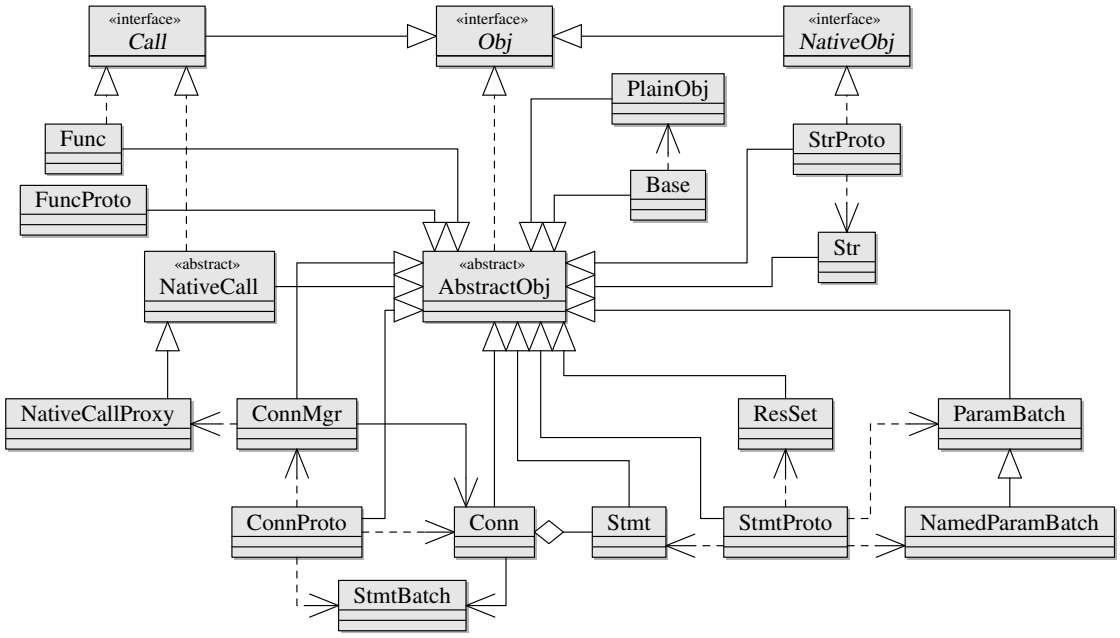
Die Semantik des `return`-Befehls in Bezug auf Block-Closures (siehe Abschnitt 4.6.2) macht besondere Vorkehrungen in der nativen Schnittstelle erforderlich. Da durch den `return`-Befehl innerhalb einer Block-Closure die Ausführung in der diese deklarierenden Funktion fortgesetzt wird, muß hier auch die Ausführung der aufrufenden nativen Funktion sofort beendet werden. Im Prototypen wird dieses Verhalten über das werfen einer speziellen Exception, der *ClosureTerminatedException* realisiert, die nach informeller Konvention von nativem Code nicht behandelt werden soll, sodaß dieser hierdurch abrupt beendet werden kann.

Ein ähnlicher Mechanismus wird für die Realisierung von Kontrollflusselementen für Schleifenkonstrukte verwendet.

6.4 Laufzeitumgebung

Die grundlegende Struktur der Laufzeitumgebung zeigt Abbildung 6.3. Es handelt sich auch hierbei um eine vereinfachte Darstellung. In den folgenden Abschnitten wird der für die Zielstellungen dieser Arbeit relevanteste Teil vorgestellt. Zunächst werden jedoch bei der Realisierung besonders wichtige Aspekte bezüglich der Implementierung nativer Objekte allgemein erläutert.

Abbildung 6.3: Laufzeitumgebung – Klassendiagramm



6.4.1 Repräsentation nativer Objekte

Die nativen Objekte der Zielsprache können grundlegend in zwei Kategorien eingeteilt werden. Dabei handelt es sich zum einen um Singleton-Objekte, von welchen nur eine Instanz im System existiert. Diese stellen hauptsächlich grundlegende Funktionalitäten für die Programme der Zielsprache zur Verfügung. Zum anderen gibt es beliebig instanziierbare Objekte, die hauptsächlich Werte kapseln. Letztere werden im Prototypen

modelliert durch zwei Klassen – eine Instanzklasse und eine Prototypklasse. Die Instanzklasse modelliert hierbei die Instanzen der Objekte und enthält die Eigenschaften, die deren Werte repräsentieren. Die Prototypklasse modelliert jeweils zu einer Instanzklasse gehörige implizite Parent-Objekt (siehe Abschnitt 4.7.5) und implementiert dessen Funktionalität.

Da die Ausführung mehrerer Instanzen der Maschine innerhalb eines Java-Prozesses, also innerhalb einer Instanz der *Java Virtual Machine*, möglich sein soll muß sichergestellt sein, daß die verschiedenen Instanzen sich nicht gegenseitig beeinflussen können. Die Ausführung eines Programmes in einer Instanz der Maschine darf also keine direkte Auswirkung in einem Programm zeigen, das in einer anderen Instanz der Maschine ausgeführt wird.

Die Instanzen der Prototypklassen können daher nicht mit der jeweiligen Instanzklasse assoziiert werden. Stattdessen werden die Instanzen während der Initialisierung von Instanzen der Maschine in einem dazu spezifischen Kontextobjekt registriert. Dieses Kontextobjekt wird bei der Instanziierung der Instanzklassen verwendet, um das zugehörige implizite Parent-Objekt zu laden.

6.4.2 SQL-Objekte

Da die Arbeit mit dem, für die Interaktion mit Datenbanken zuständigen, Teil der Laufzeitbibliothek naturgemäß den wichtigsten Teil der Zielsprache darstellt, werden in diesem Abschnitt die diesbezüglich relevanten Objekte vorgestellt. Der restliche Teil der Laufzeitbibliothek wird hier nicht näher betrachtet. Weitergehende Informationen dazu finden sich jedoch in Anhang A.

Die folgenden Objekte sind beim Umgang mit Datenbanken in der Zielsprache involviert:

ConnMgr	Der *Connection Manager* (siehe Abschnitt 5.5.2). Dieser ist zuständig für die Verwaltung von Datenbankverbindungen.
Conn	Repräsentation einer Datenbankverbindung.
Stmt	Repräsentation eines SQL-Befehls.
ResSet	Repräsentation einer Ergebnismenge.
StmtBatch	Hilfsobjekt für die Realisierung der Batch-Ausführung von nicht-parametrisierten SQL-Befehlen.
ParamBatch	Hilfsobjekt für die Realisierung der Batch-Ausführung von parametrisierten SQL-Befehlen.

In den folgenden Abschnitten werden die wichtigsten dieser Objekte und deren Interaktion im Prototypen genauer erläutert. Da zum Verständnis hierfür jedoch auch die Behandlung von SQL-Befehlen im Übersetzer und der Maschine wichtig ist, folgt zunächst eine kurze Beschreibung dieser.

Übersetzung und Interpretation von SQL-Befehlen

In der Syntax der Zielsprache werden SQL-Befehle analog zu Zeichenketten als Literale behandelt und vom Übersetzer in einen Ausdruck des Typs *SQLLiteralExpression* transformiert, welcher die einzelnen Bestandteile eines Befehls enthält. Diese Bestandteile können neben dem eigentlichen Befehlstext auch eingebettete Variablenreferenzen sowie Zeichkettenliterale mit eingebetteten Variablenreferenzen sein. Zusätzlich enthält dieser Informationen über den zur Übersetzung des Literals verwendeten Übersetzungsmodus, wie zum Beispiel die aktive Zeichenkettensyntax.

Im Zuge der Evaluierung dieses Ausdrucks durch die Maschine werden eingebettete Variablenreferenzen aufgelöst. Sofern diese in Zeichenkettenliteralen vorkommen wird hierbei auf korrekte Maskierung der jeweiligen Begrenzerzeichen geachtet. Das Ergebnis ist ein Objekt, *RawSQL*, welches den Befehlstext sowie den Übersetzungsmodus enthält.

Je nachdem, ob ein SQL-Literal im Ausdrucks- oder im Befehlskontext notiert ist, wird die generierte *SQLLiteralExpression* vom Übersetzer in einen Ausdruck eingebettet der äquivalent zu dem Aufruf `ConnMgr.createStatement(rawSQL)` bzw. `ConnMgr.executeStatement(rawSQL)` ist. Die Hintergründe hierzu werden im folgenden Abschnitt erläutert.

ConnMgr

Das *ConnMgr*-Objekt ermöglicht im Prototypen das Erzeugen von Datenbankverbindungen sowie die explizite globale Aktivierung von Datenbankverbindungen. Zudem dient es als Referenz auf die aktive Verbindung (siehe Abschnitt 5.5.2). Für die Realisierung wird hier das Stellvertretermuster verwendet. Dabei werden Nachrichten, die an das *ConnMgr*-Objekt gesendet und von diesem nicht verstanden werden an das *Conn*-Objekt der aktiven Verbindung weitergeleitet. Unter anderem wird hierdurch ermöglicht, daß SQL-Befehle immer in der jeweils aktiven Verbindung ausgeführt werden, indem der Übersetzer diese in Nachrichten übersetzt, die an das *ConnMgr*-Objekt statt eines spezifischen Verbindungsobjektes gesendet werden.

Conn

Das *Conn*-Objekt kapselt ein Objekt des Hostsystems des Typs *java.sql.Connection*, welches eine JDBC-Datenbankverbindung repräsentiert. *Conn*-Objekte werden vom *Connection Manager* erzeugt und sind für das Erzeugen und Ausführen von SQL-Befehlsobjekten – über die bereits erwähnten `createStatement`- und `executeStatement`-Methoden – zuständig.

Durch Platzierung dieser Zuständigkeit im *Conn*-Objekt wird auf einfache Weise die Möglichkeit geschaffen, alle SQL-Befehle im Kontext einer Verbindung betrachten zu können. Dies wiederum ermöglicht zum Beispiel eine einfache Realisierung der Batch-

Ausführung nicht-parametrisierter Befehle (siehe Abschnitt 2.4.2) mittels einer Methode des *Conn*-Objektes, die nach dem Loan-Pattern arbeitet.

Darüberhinaus stellt das *Conn*-Objekt weitere Methoden, zum Beispiel für die Arbeit mit Transaktionen, zur Verfügung.

Stmt

Das *Stmt*-Objekt repräsentiert ein SQL-Befehlsobjekt und kapselt sowohl den Befehlstext als auch das Befehlsobjekt des Hostsystems. Dieses kann dabei vom Typ *java.sql.Statement* oder vom Typ *java.sql.PreparedStatement* sein. Das heißt, daß das *Stmt*-Objekt sowohl vorbereitete als auch einfache Befehle kapselt.

Die Erzeugung des Host-Befehlsobjektes erfolgt mittels später Initialisierung, wobei die konkrete Art des erzeugten Befehlsobjektes durch den Kontext der Verwendung bestimmt wird. So wird zum Beispiel vom *Stmt*-Objekt eine Methode bereitgestellt, mit welcher Werte für benannte Parameter eines SQL-Befehls gesetzt werden können. Eine anschließende Ausführung führt zur erneuten syntaktischen Analyse des SQL-Befehls, um diesen auf benannte Parameter hin zu untersuchen sowie der Erzeugung eines Host-Befehlsobjektes des Typs *java.sql.PreparedStatement*, welches parametrisierbar ist.

Die Ausführung des gekapselten Befehlsobjektes kann auf zwei Wegen erfolgen. Zum einen implizit, zum Beispiel durch Aufruf der Methode mit welcher die Iteration über die Ergebnismenge veranlaßt wird. Hier werden allozierte Ressourcen automatisch im Rahmen des Loan-Pattern freigegeben. Zum anderen kann die Ausführung explizit, zum Beispiel über die `do`-Methode erfolgen. Ein Aufruf dieser Methode erfolgt auch durch die `executeStatement`-Methode des *Conn*-Objektes und entsprechend auch im Rahmen der Ausführung von SQL-Befehlen die in Programmen der Zielsprache im Befehlskontext verwendet werden.

Im Rahmen der `do`-Methode werden werden eventuell registrierte Beobachter (siehe Abschnitt 5.5.5) benachrichtigt. Dabei wird bei der Benachrichtigung sowohl der Befehlstext als auch die Ergebnismenge (siehe folgender Abschnitt) übergeben, um externen Komponenten deren Auswertung zu ermöglichen.

6.5 Qualitätsmanagement

In diesem Abschnitt sollen die während der Entwicklung des Prototypen der Sprache ergriffenen Maßnahmen zur Qualitätssicherung beschrieben werden.

Die Ziele dieser Maßnahmen lassen sich wie folgt formulieren:

1. Sicherstellen einer weitgehend korrekten Implementierung der Spezifikation der Sprache, wie sie in Kapitel 4 und Anhang A gegeben ist.
2. Erreichen einer zufriedenstellenden Qualität des Prototypen hinsichtlich dessen Implementierung sowie Entwicklungsfähigkeit.

Um diese Ziele zu erreichen wurden Maßnahmen der konstruktiven sowie der analytischen Qualitätssicherung ergriffen.

6.5.1 Maßnahmen der konstruktiven Qualitätssicherung

Als hauptsächliche Maßnahme der konstruktiven Qualitätssicherung kann die Entwicklung des Prototypen nach aktuellen Methoden des Software-Engineering angesehen werden. So erfolgte die Entwicklung zum Beispiel nach dem Prinzip der Objektorientierung. Es wurden gängige Entwurfsmuster angewendet wo diese sinnvoll einsetzbar waren. Desweiteren wurden gängige Entwurfs- und Architekturprinzipien, wie z.B. der Entwurf nach Zusändigkeiten oder die Lose Kopplung berücksichtigt.

Zusätzlich wurden Werkzeuge für die Entwicklung selbst, für das Konfigurationsmanagement (speziell Versionsmanagement) und die Planung des Projektfortschrittes eingesetzt.

6.5.2 Maßnahmen der analytischen Qualitätssicherung

Im Rahmen der analytischen Qualitätssicherung wurden sowohl statische als auch dynamische Analysen durchgeführt.

Statische Analysen

Statische Analysen wurden hauptsächlich mit Hilfe von Werkzeugen durchgeführt, die von der verwendeten Entwicklungsumgebung bereitgestellt wurden. Dabei handelt es sich im Wesentlichen um zwei verschiedene Arten:

Inspektionen Hierbei wird der Quellcode statisch auf bestimmte Kriterien hin untersucht und gegebenenfalls eine Warnung produziert. Kriterien sind zum Beispiel die redundante Initialisierung von Variablen oder Bedingungen, welche immer denselben Wert liefern. Die Inspektionen wurden automatisch laufend während der Entwicklung durchgeführt.

Abhängigkeitsanalysen Diese wurden genutzt, um problematische Abhängigkeiten zwischen den Komponenten des Prototypen zu identifizieren. Dabei handelt es sich um zyklische Abhängigkeiten zum einen zwischen Paketen, zum anderen zwischen Klassen. Diese Analysen wurden manuell in unregelmäßigen Abständen durchgeführt.

Insbesondere die Inspektionen konnten dabei gewinnbringend eingesetzt werden, da durch diese viele kleinere Fehler bereits während der Entwicklung automatisch erkannt und entsprechend korrigiert werden konnten.

Die Abhängigkeitsanalysen konnten in einigen Fällen genutzt werden, um die Struktur des Prototypen zu verbessern. Einige der erkannten zyklischen Abhängigkeiten konnten

jedoch aufgrund des begrenzten Zeitrahmens, der für die Entwicklung bestand, nicht mehr aufgelöst werden. Diese werden in Abschnitt 6.7 näher spezifiziert.

Dynamische Analysen

Dynamische Analysen wurden in Form von Tests durchgeführt. Da es sich hierbei um ein relativ umfangreiches Thema handelt geht der folgende Abschnitt separat darauf ein.

6.6 Tests

Als Teil der analytischen Qualitätssicherung wurden im Rahmen der Entwicklung des Prototypen Tests durchgeführt. Das Hauptziel dieser Tests war das Auffinden von Fehlern in der Umsetzung der Spezifikation der Sprache durch den Prototypen. Ein weiteres Ziel war das frühzeitige Erkennen von Fehlern, die durch Änderungen und Erweiterungen am Prototypen eingeführt wurden.

6.6.1 Konstruktion von Testfällen

Dem oben genannten Ziel des Testens folgend, wurden Testfälle hauptsächlich als Programme der Sprache formuliert, sodaß entsprechend die Umsetzung der Spezifikation der Sprache direkt und die Funktion des Prototypen eher indirekt getestet wurde. Dies hat den Vorteil, daß die Testfälle relativ unabhängig von der Implementierung des Prototypen sind und damit eine recht gute Wiederverwendbarkeit aufweisen.

6.6.2 Testarten

Es wurden Komponenten- und Schnittstellentests durchgeführt, wobei viele Testfälle je nach Blickwinkel als beides angesehen werden können. Desweiteren wurden Regressionstests durchgeführt.

Komponenten- und Schnittstellentests

Die Komponententests sind dabei unterteilt in drei verschiedene Klassen je nach Art der involvierten Komponenten:

1. Tests des Übersetzers
2. Tests der Maschine und der Laufzeitumgebung
3. Tests mit Zugriff auf Datenbanken

Dabei können die letzten beiden Punkte auch als Schnittstellentests angesehen werden, da das jeweilige Programm der Sprache, das den Testfall darstellt, einem Test der durch die Laufzeitumgebung der Sprache implementierten Schnittstelle entspricht.

Regressionstests

Die im vorigen Abschnitt erwähnten Komponenten- und Schnittstellentests wurden während der Entwicklung des Prototypen auch im Rahmen von Regressionstests verwendet. Dabei wurden diese Tests zumeist direkt nach einer durchgeführten Änderung oder Erweiterung ausgeführt. Zusätzlich wurden diese Tests auch vor jedem Einstellen geänderten Quellcodes in das für das Konfigurationsmanagement verwendete Versionsverwaltungssystem ausgeführt.

6.6.3 Werkzeuge

Für die Automatisierung der Testausführung, die insbesondere für die durchgeführten Regressionstests relevant ist, wurde das Framework *TestNG*[23] verwendet. Dieses zeichnet sich besonders durch die Möglichkeit der Definition von Abhängigkeiten zwischen den verschiedenen Testfällen aus, wodurch das Auftreten von Folgefehlern reduziert werden kann. So ist beispielsweise eine Abhängigkeit der Maschinen- bzw. Laufzeitumgebungstests von den Übersetzertests definiert. Ein erkannter Fehler in letzteren verhindert dadurch die Ausführung ersterer, die mit hoher Wahrscheinlichkeit Fehler erkennen würden, die nur als Folge des erkannten Fehlers im Übersetzer auftreten würden.

Für die Messung der Anweisungsüberdeckung der Tests wurde das Werkzeug *EMMA*[24] verwendet.

Beide Werkzeuge wurden von der verwendeten Entwicklungsumgebung unterstützt, sodaß die Arbeit mit diesen Werkzeugen über die entsprechend bereitgestellten Schnittstellen erfolgen konnte.

6.6.4 Testverfahren

Für die Auswahl von Testfällen wurden verschiedene Ansätze kombiniert, die in den folgenden Abschnitten vorgestellt werden.

Black-Box-Tests

Dem formulierten Ziel des Testens, Fehler in der Umsetzung der Spezifikation der Sprache zu finden, kommt das Verfahren des Black-Box-Testens entgegen. Dieses wurde häufig verwendet, um geeignete Testfälle bzgl. eines bestimmten Aspekts der Spezifikation zu finden. Dabei wurde teilweise mit Äquivalenzklassenbildung und Grenzwertermittlung gearbeitet. Einschränkend muß hierzu jedoch gesagt werden, daß diese aufgrund des

[23] `http://testng.org/`

[24] `http://emma.sf.net/` – Hierbei ist anzumerken, daß mit diesem Werkzeug nicht konkret die durchlaufenen Anweisungen, sondern vielmehr die durchlaufenen Quellcodezeilen gemessen werden können. Da diese jedoch weitgehend korrelieren sollten, wird dieser Unterschied in der weiteren Betrachtung außen vor gelassen.

engen Zeitrahmens eher selten konsequent und systematisch umgesetzt wurden. Häufiger wurden diese aus intuitivem Verständnis heraus identifiziert und Testfälle entsprechend ausgewählt. Für die weitere Entwicklung besteht in diesem Zusammenhang daher noch Potential zur Verbesserung.

White-Box-Tests

Bei der Ausführung der Test-Suite konnte mithilfe des oben genannten Werkzeugs die Anweisungsüberdeckung (C_0-Überdeckung) auf einfache Weise gemessen werden. Entsprechend konnte dieses Maß für die Konstruktion weiterer Testfälle im Rahmen des White-Box-Verfahrens herangezogen werden. Der in Abschnitt 6.6.1 beschriebene Ansatz zur Konstruktion von Testfällen hat sich diesbezüglich als positiv erwiesen, da dabei naturgemäß alle relevanten Komponenten zum Einsatz kommen und die Anweisungsüberdeckung bereits dadurch recht gute Werte erreichte.

Als weiterer positiver Effekt der Messung der Anweisungsüberdeckung neben der Ermittlung von noch nicht getestetem Code sei auch die Ermittlung von unbenutztem bzw. totem Code erwähnt, der in einigen Fällen erst hierdurch offentsichtlich wurde.

6.6.5 Ergebnisse

Da das Black-Box-Verfahren nicht durchgehend systematisch verwendet wurde, bleibt als Maß für die Testabdeckung hauptsächlich der Grad der Anweisungsüberdeckung. Dieser weist beim finalen Stand des Prototypen einen Wert von 70% auf, der Grad der getesteten Klassen des Prototypen entspricht 93%. Es besteht also Potential zur Verbesserung. Unter Berücksichtigung der Tatsache, daß die Zahl der notwendigen Testfälle bei Annäherung an 100% Anweisungsüberdeckung zunehmend steigt, kann der erreichte Wert jedoch durchaus als gutes Mittelmaß angesehen werden.

Während der Entwicklung des Prototypen wurde keine Statistik über gefundene und korrigierte Fehler geführt. Daher ist leider keine fundierte Schätzung bzgl. der wahrscheinlichen Fehlerhäufigkeit im Prototypen möglich. Unter der Annahme, daß der Prototyp die Spezifikation der Sprache vollständig implementiert kann jedoch aufgrund des Grades der Anweisungsüberdeckung davon ausgegangen werden, daß der Prototyp einen großen Teil der Spezifikation der Sprache korrekt implementiert. Dabei ist natürlich zu berücksichtigen, daß Tests nicht die Abwesenheit von Fehlern sondern nur die Anwesenheit von Fehlern nachweisen können, sodaß selbst bei vollständiger Anweisungsüberdeckung nicht davon ausgegangen werden könnte, daß die Spezifikation fehlerfrei implementiert ist. Ebenso können natürlich Fehler in der Spezifikation enthalten sein, die durch die Tests nicht aufgedeckt werden können.

6.7 Kritik

Im Prototypen konnten in dem zur Verfügung stehenden Zeitrahmen noch nicht alle gewünschten Funktionalitäten umgesetzt werden. Dies betrifft hauptsächlich die folgenden Punkte:

- Die Laufzeitumgebung ist nicht vollständig ausimplementiert. Zwar stehen alle relevanten Objekte zur Verfügung, diese bieten jedoch noch nicht jede wünschenswerte Funktion. So werden zum Beispiel hinsichtlich arithmetischer Operationen von den numerischen Objekte nur die vier Grundrechenarten unterstützt.
- Nicht alle Spezifika der verschiedenen SQL-Dialekte werden vom Übersetzer berücksichtigt. So werden zum Beispiel Backslash-Escape-Sequenzen in Zeichenkettenliteralen noch nicht unterstützt.
- Diverse Funktionalitäten der *JDBC*-Schnittstelle wurden noch nicht berücksichtigt. So wird zum Beispiel das Laden mehrere Ergebnismengen eines SQL-Befehls noch nicht im Prototypen abgebildet. Ebenso werden weitergehende Funktionalitäten wie der Umgang mit Stored Procedures, Savepoints, etc. noch nicht direkt unterstützt. Stattdessen muss hierfür direkt auf die Objekte der JDBC-Schnittstelle zurückgegriffen werden.
- Die Unterstützung von verteilten Transaktionen wurde noch nicht implementiert. Für die weitere Entwicklung empfiehlt sich hier der Einsatz des Frameworks *Atomikos TransactionsEssentials*[25] oder auch des *Bitronix Transaction Manager*[26].
- Die Implementierung der JSR-223-Spezifikation ist ebenso noch nicht erfolgt. In der weiteren Entwicklung ist hier der Einsatz einer Kompatibilitätsschicht geplant mit welcher die notwendigen Funktionalitäten bereitgestellt werden können.

Die relevantesten Punkte wurden jedoch beispielhaft implementiert, sodaß eine grundlegende Funktionalität des Prototypen gewährleistet ist.

Neben diesen, die Funktionalität betreffenden Punkten, weist der Prototyp auch einige qualitative Mängel in Form von zyklischen Abhängigkeiten auf. Dabei handelt es sich zum einen um Abhängigkeiten zwischen den verschiedenen numerische Objekten, zum anderen um die Abhängigkeit zwischen dem Interface *Context*, welches den Maschinenkontext repräsentiert und dem Interface *Obj*, welches die nativen Objekte repräsentiert. Der Maschinenkontext verwendet hier native Objekte, um Assoziationen zwischen diesen Objekten zu verwalten. Native Objekte wiederum verwenden den Maschinenkontext, um auf verschiedene globale Objekte zuzugreifen.

Desweiteren bestehen einige zyklische Abhängigkeiten zwischen Paketen, die auf Schnittstellenverletzungen in der Architektur hinweisen.

Im weiteren Verlauf der Entwicklung müssen diese problematischen Abhängigkeiten noch behandelt werden.

[25] `http://www.atomikos.com/Main/TransactionsEssentials`
[26] `http://docs.codehaus.org/display/BTM/Home`

7 Fazit

7.1 Zusammenfassung

Mit der prototypischen Implementierung von *EllaScript* (Ella Language for Language Amplification) wurde eine funktionsfähige Skriptsprache geschaffen, die mit der gewählten Kombination aus objektorientierter und imperativer Programmierung leistungsfähig und einfach zu bedienen ist. Aufgrund des Umfangs des Themas wurden bei der Konzeption hauptsächlich Lösungsansätze gewählt, die sich durch einfache Realisierbarkeit auszeichnen, wodurch bereits der Prototyp einen recht hohen Funktionsumfang aufweist. Die formulierten Anforderungen wurden daher weitgehend erfüllt, sodaß sich dieser für eine komplexe Interaktion mit einer Vielzahl von Datenbanksystemen eignet.

Durch die Konzeption als interpretierte Programmiersprache wurde die Berechnungsvollständigkeit sowie die Möglichkeit der dynamischen Ausführung erreicht. Die Abwärtskompatibilität wurde u.a. durch die Konzeption der Grammatik der Sprache als Obermenge von SQL erreicht. Hierdurch wurde ebenso ein einfacher interaktiver Umgang mit der Sprache ermöglicht. Durch die Nutzung von *JDBC* als Datenbankschnittstelle konnte eine gute Datenbankunabhängigkeit erreicht werden. Die Arbeit mit mehreren Datenbankverbindungen konnte durch die entsprechende Modellierung der Laufzeitbibliothek ermöglicht werden.

7.2 Ausblick

Für die weitere Entwicklung der Sprache empfiehlt sich zunächst die Erweiterung der Laufzeitbibliothek, um z.B. die Unterstützung der von *JDBC* bereitgestellten Funktionalität zu vervollständigen. In weiteren Schritten kann die Ausführungsgeschwindigkeit der Maschine adressiert werden, der bei der Entwicklung des Prototypen keine hohe Priorität beigemessen wurde. Neben der Einführung einer Codeoptimierungsphase im Übersetzer kann hier auch die Umsetzung einer anderen als der verwendeten CPS-Interpretationstechnik erwogen werden. Hinsichtlich der geplanten Unterstützung für dynamische Sprache in kommenden Versionen der *Java Virtual Machine* ist hierfür besonders der Ansatz der Code-Generierung (Abschnitt 3.8.5) interessant. Eine Ausweitung des Fokus der Sprachen auf zusätzliche SQL-ähnliche Abfragesprachen, wie z.B. der *Hibernate Query Language*[27], erscheint ebenfalls sinnvoll.

[27] Proprietäre Abfragesprache des Hibernate ORM-Frameworks (siehe Abschnitt 2.2.8).

A Sprachreferenz

In der folgenden Referenz wird versucht, eine möglichst vollständige, für die Verwendung der Sprache relevante, Beschreibung derselben zu geben.

A.1 Notation

Für die Beschreibung der Grammatik der Sprache wird eine abgewandelte Form der *BNF* verwendet, die sich in den folgenden Punkten von dieser unterscheidet:

- Terminalsymbole werden in einfache Anführungsstriche gefaßt.
- Anführungsstriche in Terminalsymbolen werden mit vorangestelltem Backslash notiert. Der Backslash selbst ebenso.
- Die Sonderzeichen Newline sowie Carriage Return können in Terminalsymbolen durch die Sequenzen `\n` bzw. `\r` dargestellt werden.
- Gruppierungen werden durch runde Klammern dargestellt.
- Als Quantoren werden `*` (beliebig oft), `+` (mindestens einmal) sowie `?` (optional) verwendet.
- Einige Sequenzen sind der Übersicht halber natürlichsprachig notiert. Diese werden *kursiv* dargestellt.

Zusätzlich zu diesen Modifikationen sind einige Nichtterminalsymbole mit tiefgestellten Auszeichnern versehen. Diese haben zwei verschiedene Bedeutungen. In der im folgenden Abschnitt beschriebenen Scannergrammatik schränken die Auszeichner den Kontext, in welchem das markierte Nichtterminalsymbol aktiv ist, ein. Die folgenden Auszeichner werden dabei verwendet:

normal Das markierte Nichtterminalsymbol ist im Kontext eines Skriptbefehls – im Unterschied zum SQL-Befehl – aktiv.

SQL Das markierte Nichtterminalsymbol ist im Kontext eines SQL-Befehls aktiv.

SQL(dir=value) Das markierte Nichtterminalsymbol ist nur im Kontext eines SQL-Befehls aktiv, wenn die mit *dir* bezeichnete Parse-Direktive (siehe Abschnitt A.8) den mit *value* bezeichneten Wert hat.

Bei der in den darauffolgenden Abschnitten dargestellten Parsergrammatik werden die Auszeichner für die Markierung von verwandten Produktionen verwendet, die sich nur in dem Aspekt unterscheiden, der mit dem Auszeichner angezeigt wird.

In der vorliegenden Referenz werden diverse syntaktische Konstrukte beschrieben, welche in äquivalente Konstrukte übersetzt werden oder durch entsprechende Konstrukte alternativ notiert werden können. Diese werden wie folgt präsentiert:

betrachtete Notation	*äquivalente oder alternative Notation*

Die jeweils betrachtete Notation wird dabei immer links, die Alternative rechts dargestellt.

A.2 Lexikalische Struktur

```
<input element> ::= <white space>
                  | <comment>
                  | <token>

<token> ::= <identifier>
          | <keyword>
          | <literal>
          | <punctuator>
```

Jedes Programm der Sprache besteht aus Whitespace, Kommentaren und Symbolen. Symbole sind Bezeichner, Schlüsselworte, Literale sowie Interpunktion.

A.2.1 Whitespace und Kommentare

Als Whitespace werden das Leerzeichen, Carriage Return, Line Feed sowie das Tabulatorzeichen behandelt.

Die Syntax für Kommentare entspricht der in [ISO99a, Kap. 5.2] definierten Syntax, wobei zu beachten ist, daß wie im Standard vorgegeben, mehrzeilige Kommentare verschachtelt werden können.

A.2.2 Bezeichner

```
<identifier> ::= ( <word char> | <identifier start> )
                 ( <word char> | <identifier special> | <digit> )*
               | '.' '.'+

<identifier start> ::= '~' | '^' | '&' | '_' | '|' | '$' | '@'

<identifier special> ::= <identifier start> | '!' | '?'

<word char> ::= 'a'..'z' | 'A'..'Z' | '_'

<digit> ::= '0'..'9'
```

Bezeichner bestehen aus einer Sequenz der angegebenen Sonderzeichen sowie den Zeichen des englischen Alphabets. Die Erweiterung auf ein sinnvolle Teilmenge des UTF-8-Zeichensatzes ist für eine spätere Fassung vorgesehen.

A.2.3 Schlüsselworte

```
<keyword> ::= 'as' | 'break' | 'catch' | 'continue' | 'else' | 'exit'
            | 'finally' | 'for' | 'fun' | 'if' | 'import' | 'include'
            | 'new' | 'return' | 'sql' | 'super' | 'this' | 'throw'
            | 'try' | 'var' | 'while'
```

Schlüsselworte bezeichnen reservierte Worte der Sprache. Diese können nicht als Bezeichner verwendet werden.

A.2.4 Literale

```
<literal> ::= <integer literal>
            | <floating-point literal>
            | <boolean literal>
            | <null literal>
            | <string literal>
```

Die verschiedenen Literale stellen konstante Werte dar. Die folgenden Abschnitte zeigen die jeweilige Grammatik der unterstützten Literale.

Ganzzahlige numerische Literale

```
<integer literal> ::= <digit>+
```

Gleitkommazahlenliterale

```
<floating-point literal> ::= <digit>+ '.' <digit>+ <exponent>?

<exponent> ::= ('e' | 'E') ('-' | '+')? <digit>+
```

Boolesche Literale

```
<boolean literal> ::= 'true' | 'false'
```

Zeichenkettenliterale

```
<string literal> ::= <plain single quoted string>_normal
                   | <double quoted string>
                   | <single quoted string>_SQL
                   | <backtick quoted string>_SQL(quotes=mysql)
                   | <qquoted string>_SQL(quotes=oracle)

<plain single quoted string> ::=
                            '\'' <plain single quoted string content>* '\''

<plain single quoted string content> ::= '\'' '\''
                                       | any input character except single quote

<double quoted string> ::= '"' <double quoted string content>* '"'

<double quoted string content> ::= '"' '"'
                                 | <literal at-sign>
                                 | <embedded variable reference>
                                 | any other input character

<literal at-sign> ::= '@' '{' '}'

<embedded variable reference> ::= '@' '{' <identifier> '}'
```

Die Produktion *single quoted string* erlaubt Variablenreferenzen innerhalb von Single-quoted Strings, die Bestandteil eines SQL-Befehls sind.

Die Syntax der Backtick-quoted Strings entspricht der von MySQL definierten Syntax. Backtick-quoted Strings sind nur innerhalb von SQL-Befehlen gültig, wenn die MySQL-Stringsyntax aktiviert ist.

Die Syntax der Q-Quoted Strings entspricht der von Oracle definierten Syntax. Analog zu den Backtick-quoted Strings sind diese nur innerhalb von SQL-Befehlen gültig, wenn die Oracle-Stringsyntax aktiviert ist.

A.2.5 Interpunktion

```
<punctuator> ::= '(' | ')' | '{' | '}' | '[' | ']'
               | ';' | '.' | ',' | '=' | '!' | '?' | ':'
               | '\\' | '=>' | ':=' | '&&' | '||' | '=='
               | '!=' | '===' | '!==' | '>' | '>=' | '<' | '<='
               | '+' | '-' | '*' | '/' | '%'
               | <sql slash sep>SQL(sep=slash)

<sql slash sep> ::= '\n' '/' '\r'? '\n'
```

Neben Whitespace und Kommentaren fungieren auch Interpunktionssymbole als Symboltrenner. Einen Sonderfall stellen hier die Zeichen `!` und `?` dar. Da Bezeichner zwar nicht mit diesen beginnen, sie aber dennoch enthalten dürfen, werden Bezeichnersymbole durch diese Zeichen nicht getrennt. Im Zweifelsfall sollten die betreffenden Symbole in diesen Fällen daher explizit durch Whitespace von vorangestellten Symbolen getrennt werden.

A.3 Struktur von Programmen

```
<script> ::= <statement>SQL*

<sql block> ::= 'sql' '{' <statement>SQL* '}'

<block> ::= '{' <statement>* '}'
```

Programme der Sprache sind grundlegend in Blöcken strukturiert, welche Befehle enthalten können. Da auch Blöcke Befehle darstellen, können diese beliebig verschachtelt werden.

Es wird zwischen zwei verschiedenen Arten von Blöcken unterschieden. Im SQL-Befehlsblock können SQL-Befehle ohne einen kennzeichnenden Präfix notiert werden. Ausdrücke im Befehlskontext müssen hingegen mit dem Punkt als Präfix notiert werden. In normalen Blöcken verhält es sich umgekehrt.

Jedes Programm der Sprache stellt einen impliziten SQL-Befehlsblock dar, sodaß auf der obersten Ebene dessen syntaktische Regeln gelten.

A.4 Namen und Bindung

Die Sprache verwendet statische Bindungsregeln. Diese werden in Abschnitt 3.4 genauer beschrieben.

A.5 Objekte

Die Sprache ist eine prototyp-basierte objektorientierte Sprache in der alle Werte durch Objekte repräsentiert werden. Abschnitt 4.7 erläutert die diesbezüglich verwendeten Konzepte im Detail.

A.6 Befehle

```
<statement>_SQL ::= <script statement>_nosep
                  | <script statement>
                  | '.' <expression statement>
                  | <sql statement>
                  | <parse directive>

<statement> ::= <script statement>_nosep
              | <script statement>
              | <expression statement>
              | <sql statement>_prefixed
              | <parse directive>

<script statement>_nosep ::= <block>
                  | <sql block>
                  | <function definition statement>
                  | <if statement>
                  | <try statement>
                  | <for statement>
                  | <while statement>

<script statement> ::= <assign statement> ';'
              | <throw statement> ';'
              | <break statement> ';'
              | <continue statement> ';'
              | <return statement> ';'
              | <exit statement> ';'
              | <import statement> ';'
              | <include statement> ';'
              | ';'

<expression statement> ::= <assign expression> ';'

<expression statement>_noSQL ::= <assign expression>_noSQL ';'
```

Befehle müssen in der Sprache durch einen Separator terminiert werden. Bei normalen Befehlen handelt es sich hierbei um das Semikolon. SQL-Befehle unterstützen auch andere Sequenzen, die durch Parse-Direktiven aktiviert werden können (siehe Abschnitt A.8).

Bei den unterstützten Befehlen handelt es sich teilweise um syntaktischen Zucker, der jeweils in eine entsprechende Ausdrucksform übersetzt wird. Diese wird bei der Erläuterung der einzelnen Befehle im Folgenden angegeben.

Neben normalen Befehlen können auch Ausdrücke im Befehlskontext verwendet werden.

A.6.1 Funktionsdefinition

```
<function definition statement> ::=
                        'fun' <identifier> '(' <identifier list>? ')' <block>

<identifier list> ::= <identifier> ( ',' <identifier> )*
```

Funktionsdefinitionen im Befehlskontext sind grundlegend äquivalent zu den entsprechende Definitionsausdrücken. Hier gilt jedoch die Einschränkung, daß anonyme Funktionsdeklarationen im Befehlskontext nicht zulässig sind.

Funktionsdefinitionen werden genauer in Abschnitt A.7.9 beschrieben.

A.6.2 return-Befehl

```
<return statement> ::= 'return' <expression>?
```

Der `return`-Befehl beendet die Ausführung der Funktion in welcher dieser notiert wurde explizit. Der optionale Ausdruck wird vor der Rückkehr evaluiert und dessen Wert von der Funktion zurückgegeben. Ausdrücke die Aktivierungen darstellen sind Gegenstand einer Tail-Call-Optimierung.

A.6.3 Zuweisungsbefehle

```
<assign statement> ::= 'var' <assignment>

<assignment> ::= <identifier>
                 ( <assign op> ( <expression>
                               | <expression>_noSQL <assignment rest>
                               )
                 | <assignment rest>
                 )

<assignment rest> ::= ( ',' <identifier> <assign op> <expression>_noSQL )*

<assign op> ::= ':=' | '='
```

Zuweisungsbefehle dienen sowohl der Deklaration von Variablen als auch der Zuweisung von Werten an Variablen.

Ein Zuweisungsbefehl welcher nur einen Bezeichner spezifiziert entspricht der Deklaration einer entsprechenden Variablen.

Eine Zuweisungs mittels des Deklarationsoperators `:=` resultiert in der Deklaration der entsprechenden Variablen und der gleichzeitigen Zuweisung des Wert den die Evaluierungs des Ausdrucks der rechten Seite liefert.

Die Deklaration einer Variablen, welche im aktuellen Gültigkeitsbereich bereits deklariert ist führt zu einer Warnung und wird ansonsten ignoriert.

Die Zuweisung eines Wertes an eine nicht deklarierte Variable führt zu einer Warnung und der impliziten Deklaration der Variablen.

Die Evaluierung der Teilausdrücke des Zuweisungsbefehls erfolgt von rechts nach links.

A.6.4 SQL-Befehle

```
<sql statement> ::= <sql literal> <sql statement separator>

<sql statement>_prefixed ::= <sql literal>_prefixed <sql statement separator>
```

SQL-Befehle entsprechen weitgehend den in Abschnitt A.7.9 beschriebenen SQL-Ausdrücken. Im Unterschied zu diesen ist jedoch das Ergebnis eines SQL-Befehls semantisch äquivalent zur Erzeugung eines Objektes des Typs *Stmt*, wie dies bei SQL-Ausdrücken der Fall ist, und zusätzlich dem Versand der `do`-Nachricht an dieses Objekt.

A.6.5 Pseudobefehle

Bei den folgenden Befehlen handelt es sich um syntaktische Konstrukte, welche in Funktionsaufrufe übersetzt werden. Die Beschreibung dieser Befehle beschränkt sich daher auf die Angabe der jeweiligen Grammatik sowie des bzw. der jeweils äquivalenten Funktionsaufrufe. Deren Verhaltensweise wird an anderer Stelle erläutert.

If-Anweisung

```
<if statement> ::= 'if' <paren expression> <block>
                   ( 'else' ( <if statement> | <block> ) )?
```

```
if (expression) {
    statement;
}
```

```
Sys.ifThen(expression, {=>
    statement;
});
```

```
if (expression) {
    statement1;
}
else {
    statement2;
}
```

```
Sys.ifThen(expression, {=>
    statement1;
}, {=>
    statement2;
});
```

Try-Anweisung

```
<try statement> ::= 'try' <block> <catch branch> | <finally branch>

<catch branch> ::= 'catch' '(' <identifier> ')' <block> <finally branch>?

<finally branch> ::= 'finally' <block>
```

```
try {
    statement1;
} catch (identifier) {
    statement2;
}
```

```
Sys.tryCatch({=>
    statement1;
}, { identifier =>
    statement2;
});
```

```
try {
    statement1;
} finally {
    statement2;
}
```

```
Sys.tryFinally({=>
    statement1;
}, {=>
    statement2;
});
```

```
try {
    statement1;
} catch (identifier) {
    statement2;
} finally {
    statement3;
}
```

```
Sys.tryCatchFinally({=>
    statement1;
}, { identifier =>
    statement2;
}, {=>
    statement3;
});
```

Throw-Anweisung

```
<throw statement> ::= 'throw' <expression>
```

```
throw expression;
```

```
Sys._throw(expression);
```

For-Anweisung

```
<for statement> ::= 'for' <identifier>?
                    '(' <identifier list> ':' <expression> ')' <block>
```

```
for (identifier : expression) {
    statement;
}
```

```
(expression).each { identifier =>
    statement;
};
```

```
for identifier1 (identifier2 : expression) {
    statement;
}
```

```
(expression).identifier1 { identifier2 =>
    statement;
};
```

While-Anweisung

```
<while statement> ::= 'while' <paren expression> <block>
```

```
while (expression) {
    statement
}
```

```
{=> expression } whileTrue: {=>
    statement
};
```

Break-Anweisung

```
<break statement> ::= 'break'
```

```
break;
```

```
Sys._break();
```

Continue-Anweisung

```
<continue statement> ::= 'continue'
```

```
continue;
```

```
Sys._continue();
```

Exit-Anweisung

```
<exit statment> ::= 'exit' <expression>?
```

```
exit;
```

```
Sys._exit();
```

Import-Anweisung

```
<import statement> ::= 'import' <java identifier> ( '.' <java identifier> )*
                       ( '.' '*' | 'as' <identifier> )?
```

Die Produktion *java identifier* erlaubt in der aktuellen Fassung der Sprache eine Teilmenge der Bezeichner, welche die Produktion *identifier* erlaubt. Daher werden momentan nicht alle in Java gültigen Bezeichner von dieser Regel erfaßt.

```
import pkg.Class;
```

```
var Class := new JClass('pkg.Class');
```

```
import pkg.Class as Alias;
```

```
var Alias := new JClass('pkg.Class');
```

```
import pkg.*;
```

```
Sys.importPackage('pkg');
```

Include-Anweisung

```
<include statement> ::= 'include' <expression>
```

```
include expression;
```

```
Sys.includeFile(expression);
```

A.7 Ausdrücke

Dieser Abschnitt widmet sich den verschiedenen Ausdrücken, welche von der Sprache unterstützt werden.

```
<paren expression> ::= '(' <expression> ')'

<expression> ::= <expression statement>
               | <function definition expression>
               | <object literal>
               | <sql expression>

<expression>noSQL ::= <expression statement>noSQL
                    | <function definition expression>
                    | <object literal>
```

Jeder Ausdruck kann mit runden Klammern umschlossen werden, wodurch die Reihenfolge der Evaluierung von Ausdrücken direkt beeinflusst werden kann.

Die syntaktischen Gegebenheiten der Sprache erfordern an einigen Stellen, daß SQL-Befehle im Ausdruckskontext explizit von Klammern umschlossen werden, um Mehrdeutigkeiten der Grammatik zu vermeiden. Diese Notwendigkeit wird durch die oben angegebenen beiden *expression*-Produktionen reflektiert.

A.7.1 Zuweisungen

```
<assign expression> ::= <identifier> ':=' <expression>
                      | <conditional expression> '=' <expression>

<assign expression>noSQL ::= <identifier> ':=' <expression>noSQL
                           | <conditional expression> ( '=' <expression>noSQL )?
```

Zuweisungen können sowohl mit dem Deklarationsoperator `:=` als auch mit dem eigentlichen Zuweisungsoperator `=` erfolgen. Die Evaluierung der Teilausdrücke wird von rechts nach links vorgenommen.

Zuweisungsausdrücke werden von rechts nach links evaluiert.

A.7.2 Ternärer Operator

```
<conditional expression> ::= <or condition>
                             ( '?' <conditional result>
                               ':' <conditional result> )

<conditional result> ::= <conditional expression>
                       | <object literal>
```

Der ternäre Operator ist dem aus vielen *C*-ähnlichen Programmiersprachen bekannten gleichnamigen Konstrukt nachempfunden. Dieser Ausdruck ist als syntaktischer Zucker realisiert und wird in eine entsprechende If-Anweisung übersetzt.

```
condition ? expression1 : expression2
```

```
Sys.ifThen(condition, {=>
  expression1;
}, {=>
  expression2;
})
```

A.7.3 Binäre Operatoren

```
<binary expression> ::= <additive expression>
                        ( <identifier> <additive expression> )?
```

Binäre Operatoren sind in der Sprache als syntaktische Sonderform des Nachrichtenversands modelliert, bei welcher die jeweilige Nachricht zwei Argumente hat. Die gesendete Nachricht entspricht dabei dem jeweiligen Operator.

Einige der Operatoren werden speziell behandelt, um Vorrangregeln der Operatoren abzubilden. Generell kann jedoch jede Nachricht, welche zwei Argumente hat als binärer Operator verstanden bzw. notiert werden.

A.7.4 Unäre Operatoren

```
<unary expression> ::= <unary not expression>
                     | <unary minus expression>
                     | <call expression>
```

Unäre Operatoren stellen wiederum eine syntaktische Sonderform des Nachrichtenversands für Nachrichten mit nur einem Argument dar. Hier werden nur bestimmte Operatoren unterstützt, welche ausschließlich als Präfixoperatoren fungieren.

A.7.5 Logische Operatoren

```
<or condition> ::= <and condition> ( '||' <and condition> )*

<and condition> ::= <comparison condition> ( '&&' <comparison condition> )*

<unary not expression> ::= '!' <unary expression>
```

Um bei den binären logischen Operatoren `&&` bzw. `||` (logisches Und bzw. logisches Oder) eine bedingte Evaluierung der rechten Seite in Abhängigkeit von der linken Seite zu ermöglichen erfolgt eine Übersetzung des Ausdrucks bei welcher die rechte Seite jeweils in eine Block-Closure gefaßt wird, welche nach Bedarf evaluiert werden kann.

*condition*1 && *condition*2	(*condition*1).&&({=> *condition*2; })

Der unäre logische Operator `!` wird bei der Übersetzung nicht speziell behandelt und führt daher zum Aufruf der entsprechenden Nachricht.

! *condition*1	(*condition*1).!()

A.7.6 Arithmetische Operatoren

```
<additive expression> ::= <multiplicative expression>
                          ( <additive operator>
                            <multiplicative expression> )?

<additive operator> ::= '+' | '-'

<multiplicative expression> ::= <unary expression>
                                ( <multiplicative operator>
                                  <unary expression> )?

<multiplicative operator> ::= '*' | '/' | '%'

<unary minus expression> ::= '-' <unary expression>
```

Bei den arithmetischen Operatoren sind Vorrangregeln entsprechend der mathematischen Vorrangregeln realisiert. Die binären arithmethischen Operatoren führen zum Versand einer gleichnamigen Nachricht. Der unäre Minusoperator wird in den Versand der Nachricht `neg` übersetzt, um diese vom binären Minusoperator abzugrenzen.

$-$ *expression*	(*expression*).neg();

A.7.7 Vergleichsoperatoren

```
<comparison condition> ::= <binary expression>
                           ( <comparison operator> <binary expression> )?

<comparison operator> ::= '==' | '!=' | '===' | '!==' | '<' | '<=' | '>' | '>='
```

Auch die Vergleichsoperatoren werden in einen Versand einer jeweils gleichnamigen Nachricht übersetzt.

*expression*1 $==$ *expression*2	(*expression*1).==(*expression*2)

A.7.8 Suffixoperatoren

```
<call expression> ::= <simple expression><call expression suffix>

<call expression suffix> ::= <slot suffix>
                           | <call suffix>
                           | <index suffix>
```

Suffixoperatoren stellen die Operatoren mit der höchsten Priorität bzgl. des Vorrangs dar.

Die folgenden Abschnitte betrachten die verschiedenen Suffixoperatoren im Einzelnen.

Slotoperator

```
<slot suffix> ::= '.' <identifier>
                | '{' <expression> '}'
```

Der Slotoperator kann auf beliebige Objekte angewendet werden und liefert den Wert des bezeichneten Slots in dem jeweiligen Objekt. Existiert kein entsprechender Slot wird der Wert `null` zurückgegeben.

Die Punkt-Notation ist eine syntaktische Kurzform für die Klammer-Notation bei welcher der Bezeichner anstelle eines entsprechenden Zeichkettenliterals verwendet werden kann.

```
expression.identifier
```

```
expression{'identifier'}
```

Aktivierungsoperator

```
<call suffix> ::= '(' <arguments list>? ')' <block closure>?
                | <block closure>

<arguments list> ::= <expression>noSQL ( ',' <expression>noSQL )*
```

Der Aktivierungsoperator kann auf ausführbare Objekte angewendet werden. Eine Anwendung auf andere Objekte führt zu einem Fehler. Ausführbare Objekte sind Funktionen, Block-Closures, native sowie primitive Funktionen, die an anderer Stelle beschrieben sind.

Dem ausführbaren Objekt können Argumente übergeben werden, die vor dessen Aktivierung evaluiert werden. Die Anzahl der übergebenen Argumente muß im Falle von Funktionen und Block-Closures der Anzahl der Argumente entsprechen die bei deren Definition angegeben wurden. Einige native bzw. primitive Funktionen unterstützen hingegen eine variable Anzahl von Argumenten.

Im Anschluß an den Aktivierungsoperator kann optional ein Block-Closure-Ausdruck notiert werden. Diese Notation entspricht semantisch einer Notation bei welcher der Block-Closure-Ausdruck als letztes Argument des Aktivierungsoperators angegeben wurde. Alternativ kann ein Block-Closure-Ausdruck direkt auf einen primären Ausdruck folgen – der Aktivierungsoperator wird in diesem Falle impliziert. Diese auch in anderen Sprachen, wie zum Beispiel *Groovy*, zu findende Notationsform ermöglicht syntaktische Konstrukte, welche Blockstrukturen nachahmen.

```
expression1(expression2) {=>
    statement;
}
```

```
expression1(expression2, {=>
    statement;
})
```

Folgt der Aktivierungsoperator auf einen Slotoperator-Ausdruck wird das Objekt, dessen Slot referenziert wird als Kontext der Aktivierung verwendet, wodurch dieses Objekt in der Aktivierung durch das `this`-Schlüsselwort referenziert werden kann.

Indexoperator

```
<index suffix> ::= '[' <expression> ']'
```

Der Indexoperator ist eine syntaktische Abkürzung, die in Abhängigkeit von der Position des Ausdrucks in eine `get`- oder eine `set`-Nachricht übersetzt werden. Ist der Ausdruck, auf welchen sich der Indexoperator bezieht, auf der linken Seite einer Zuweisung notiert, wird eine `set`-Nachricht gesendet, ansonsten eine `get`-Nachricht. Diese Nachrichten wurden so gewählt, um eine Anwendung des Indexoperators auf alle Java-Objekte zu ermöglichen, die das Interface *java.lang.List* implementieren.

*expression*1[*expression*2]	*expression*1.get(*expression*2)
*expression*1[*expression*2] = *expression*3	*expression*1.set(*expression*2, *expression*3)

A.7.9 Einfache Ausdrücke

```
<simple expression> ::= <paren expression>
                      | <block closure>
                      | <construction expression>
                      | <array literal>
                      | <string literal>
                      | <boolean literal>
                      | <null literal>
                      | <integer literal>
                      | <floating-point literal>
                      | <identifier>
                      | 'this'
                      | <super expression>

<object literal>

<sql expression>
```

Einfache Ausdrücke stellen die grundlegenden Komponenten dar aus welchen sich Ausdrücke der Sprache zusammensetzen. Sowohl das Objektliteral als auch der SQL-Ausdruck sind prinzipiell ebenfalls ein einfache Ausdrücke, die jedoch aufgrund verschiedener syntaktischer Gegebenheiten der Sprache nicht von der obigen Produktion *simple expression* referenziert werden, sondern stattdessen an verschiedenen anderen Stellen der Grammatik der Sprache eingebunden sind.

Die folgenden Abschnitte gehen auf die verschiedenen Arten der einfachen Ausdrücke ein.

Funktionsdefinition

```
<function definition expression> ::=
                    'fun' <identifier>? '(' <identifier list>? ')' <block>
```

Funktionsdefinitionen erzeugen Funktionen, die in der Sprache durch aktivierbare Objekte des Typs *Func* repräsentiert werden.

Es können sowohl anonyme als auch benannte Funktionen definiert werden. Benannte Funktionen teilen sich den gleichen Namensraum mit Variablen. Die Definition einer benannten Funktion entspricht daher semantisch der Deklaration einer gleichnamigen Variablen mit anschließender Zuweisung einer entsprechenden anonymen Funktion.

```
fun identifier () {
    statement;
}
```

```
var identifier := fun () {
    statement;
};
```

Funktionen können optional Argumente definieren, welche beim Aufruf übergeben werden müssen.

Funktionen formen lexikalische Closures, sodaß freie Variablen innherhalb der jeweiligen Funktion referenziert werden können, auch wenn der Gültigkeitsbereich der Aktivierung der Funktion nicht dem Gültigkeitsbereich von deren Deklaration entspricht.

Block-Closure

```
<block closure> ::= '{' <identifier list>? '=>' <block content>* '}'
```

Block-Closure-Ausdrücke erzeugen aktivierbare Objekte des Typs *Clos*.

`return`-Befehle innerhalb von Block-Closures beziehen sich auf die definierende Funktion und beenden daher sowohl die Ausführung der Block-Closure als auch der definierenden Funktion. Ist die Ausführung der definierenden Funktion bereits beendet führt der `return`-Befehl zum einem Fehler.

`this`- sowie `super`-Ausdrücke innerhalb von Block-Closures beziehen sich ebenfalls auf die definierende Funktion – liefern also jeweils das gleiche Objekt wie in der definierenden Funktion.

Auch Block-Closures formen wie Funktionen lexikalische Closures.

Objektliteral

```
<object literal> ::= '{' ( <object slot> ( ',' <object slot> )* )? ','* '}'

<object slot> ::= <identifier> ':' <expression>noSQL
                | <string literal> ':' <expression>noSQL
```

Objektliterale dienen der Erzeugung neuer Objekte mit den jeweils angegebenen Slots.

Arrayliteral

```
<array literal> ::= '[' ( <expression>noSQL
                          ( ',' <expression>noSQL )*
                        )?
                        ','*
                    ']'
```

Arrayliterale stellen eine syntaktische Kurzform für die Erzeugung von Objekten des Typs *Lst* dar.

```
[]
```

```
Lst.clone()
```

SQL-Ausdruck

```
<sql expression> ::= <sql literal>prefixed

<sql literal> ::= <sql literal>prefixed
                | <sql intro> <sql token>*

<sql literal>prefixed ::= 'sql' <sql token>*

<sql intro> ::= <keyword>
              | <word>
              | <literal at-sign>
              | <embedded variable reference>

<sql token> ::= <sql string literal>
              | <literal at-sign>
              | <embedded variable reference>
              | any other character except separator

<sql statement separator> ::= ';' | <sql slash sep>

<word> ::= <word char>+
```

SQL-Ausdrücke erzeugen Objekte des Typs *Stmt*, welche jeweils den entsprechenden SQL-Befehlstext kapseln. Diese Ausdrücke können eingebette Variablenreferenzen enthalten, die zur Laufzeit interpoliert werden. Desweiteren kann auch jedes Zeichenkettenliteral innerhalb des SQL-Ausdrucks Variablenreferenzen enthalten. Bei der Interpolation dieser Variablenreferenzen werden eventuell enthaltene Begrenzersymbole maskiert.

Konstruktionsausdruck

```
<construction expression> ::= 'new' <simple expression>
                              '(' <arguments list>? ')'
```

Mithilfe des Konstruktionsausdruckes können Objekte nach dem Schema der klassenbasierten Objektorientierung erzeugt werden. Je nach dem Wert den die Evaluierung des dem `new`-Operator übergebenen Ausdrucks liefert, werden verschiedene Aktionen ausgeführt.

Handelt es sich bei dem Wert um ein natives Objekt, welches einen nativen Konstruktor spezifiziert wird dieser mit den entsprechenden Argumenten aufgerufen.

Handelt es sich hingegen um ein benutzerdefiniertes Objekt bzw. um ein Objekt, welches keinen nativen Konstruktor spezifiziert so wird ein neues Objekt erzeugt, welches das übergebene Objekt als Parent-Objekt referenziert. Im Anschluß wird der `init`-Slot des erzeugten Objektes mit den übergebenen Argumenten aktiviert.

Zeichenkettenliterale

Zeichenkettenliterale erzeugen Objekte des Typs *Str*. In Zeichenkettenliteralen, welche doppelte Anführungsstriche als Begrenzer verwenden, werden eingebettete Variablenreferenzen zur Laufzeit interpoliert. Bei Zeichenkettenliteralen mit einfachen Anführungsstrichen als Begrenzern ist dies nicht der Fall.

Boolesche Literale

Die booleschen Literale `true` und `false` liefern Objekte des Typs *Bool*. Es handelt sich dabei um Singleton-Objekte, welche jeweils nur ein einziges Mal in der Umgebung des Programmes existieren.

Null-Literal

Das `null`-Literal liefert ein Objekt des Typs *Null*. Von diesem Objekt können mehrere Instanzen in der Umgebung existieren, da dieses mit Typinformationen versehen werden kann, was bei der Interaktion mit Objekten der Hostsprache sowie SQL-Befehlen relevant ist.

Numerische Literale

Ganzzahlige numerische Literale liefern Objekte des Typs *Num* mit dem entsprechenden Wert. Gleitkommaliterale liefern Objekte des Typs *Real*.

Bezeichnerausdrücke

Bezeichnerausdrücke referenzieren die gleichnamige Variable. Ist eine Variable nicht definiert so liefern diese Ausdrücke den Wert *null* und es wird eine Warnung ausgegeben.

`this`-Ausdruck

Das Schlüsselwort `this` liefert den Kontext, also ein Objekt, in welchem die aktive Funktion ausgeführt wurde.

`super`-Ausdruck

```
<super expression> ::= 'super' <slot suffix> ( <call suffix> )?
```

Der `super`-Ausdruck ermöglicht das Senden von Nachrichten bzw. Referenzieren von Slots des Parent-Objektes des Objektes, in welchem die aktive Funktion lokalisiert wurde.

A.8 Parse-Direktiven

```
<parse directive> ::= '\\' 'set' ( 'quotes' | 'sep' ) '=' ( <identifier> | <word> ) ';'
```

Parse-Direktiven erlauben die Beeinflussung des Übersetzungsvorgangs. Sie bestehen aus dem Namen der Direktive sowie einem Wert. Parse-Direktiven haben nur innerhalb des umschließenden Blocks eine Wirkung. Die unterstützten Direktiven werden im Folgenden erläutert.

A.8.1 `quotes`

Die `quotes`-Direktive ermöglicht die Beeinflussung der Zeichenkettensyntax innerhalb von SQL-Literalen. Die folgenden Werte sind gültig:

sql92 Aktiviert die in [ISO92] definierte Zeichenkettensyntax. Dies ist der Standardwert.

oracle Aktiviert die von der Oracle Datenbank Version 10g unterstützte Zeichenkettensyntax.

mysql Aktiviert die von der MySQL Datenbank Version 5.0 unterstützte Zeichkettensyntax.

Die Unterstütztung für die von der PostgreSQL Datenbank Version 8.2 unterstützte Zeichenkettensyntax ist teilweise vorbereitet, jedoch noch nicht vollständig funktionsfähig.

A.8.2 `sep`

Die `sep`-Direktive ermöglicht die Spezifierung des zu verwendenden Befehlstrenners. Dies bezieht sich ausschließlich auf SQL-Befehle. Die folgenden Varianten werden unterstützt:

semicolon Aktiviert das Semikolon als Befehlstrenner. Dies ist der Standardwert.

slash Aktiviert die in der Scannergrammatik mittels der Produktion *sql slash sep* definierte Syntax als Befehlstrenner.

Die Aktivierung einer Variante der Befehlstrenner führt automatisch zur Deaktivierung der anderen unterstützten Varianten.

A.9 Vordefinierte Objekte

In den folgenden Abschnitten werden vordefinierte Objekte der Sprache sowie deren unterstützte Eigenschaften und Operationen bzw. Nachrichten beschrieben.

Die Namen der vordefinierten Objekte wurden jeweils so gewählt, daß möglichst kein Namenskonflikt mit Objekten der Standardbibliothek von Java entsteht. Hierdurch soll eine bessere Integration mit der Hostsprache gewährleistet werden, da dies ermöglicht, Klassen der Standardbibliothek von Java zu importieren, ohne Konflikte zu verursachen.

A.9.1 Obj

Dieses Objekt stellt die Basis aller Objekte der Sprache dar.

===(object) Führt einen Identitätsvergleich nach der Semantik der Hostsprache aus. Liefert `true` falls das mit *object* bezeichnete Objekt identisch ist, ansonsten `false`.

!==(object) Führt einen Identitätsvergleich nach der Semantik der Hostsprache aus. Liefert `true` falls das mit *object* bezeichnete Objekt nicht identisch ist, ansonsten `false`

==(object) Führt eine Gleichheitsprüfung nach der Semantik der Hostsprache mittels der `equals()`-Methode des jeweligen nativen Objektes aus. Liefert `true` falls das mit *object* bezeichnete Objekt gleich ist, ansonsten `false`.

!=(object) Führt eine Gleichheitsprüfung nach der Semantik der Hostsprache mittels der `equals()`-Methode des jeweiligen nativen Objektes aus. Liefert `true` falls das mit *object* bezeichnete Objekt nicht gleich ist, ansonsten `false`.

&&(closure) Aktiviert die mit *closure* bezeichnete Block-Closure und liefert das Ergebnis von deren Evaluierung.

||(closure) Liefert `true`. Die mit *closure* bezeichnete Block-Closure wird nicht aktiviert.

!() Liefert `false`.

clone() Erzeugt eine Kopie des Objektes nach dem Schema der differentiellen Vererbung. Dabei wird ein neues Objekt erzeugt, dessen Parent Slot auf das Objekt verweist, dessen `clone()`-Slot aktiviert wurde. Anschließend wird der `cloneInit()`-Slot des erzeugten Objektes aktiviert.

each(closure) Iteriert über die Slots des Objektes, dessen `each()`-Slot aktiviert wurde und aktiviert die mit *closure* bezeichnete Block-Closure für jeden Slot. Dieser werden jeweils der aktuelle Slot und dessen Wert als Argument übergeben. Die Slots der Parent-Objekte werden nicht berücksichtigt.

eachSlot(closure) Alias für `each`.

hasSlot(slot) Prüft, ob das Objekt den angegebenen Slot enthält. Liefert entsprechend `true` oder `false`. Die Slots der Parent-Objekte werden nicht berücksichtigt.

not() Alias für `!`.

removeSlot(slot) Entfernt den angegebenen Slot vom betreffenden Objekt und liefert den damit assoziierten Wert. Enthält das Objekt keinen entsprechenden Slot wird `null` zurückgegeben. Die Slots der Parent-Objekte werden nicht berücksichtigt.

toDict() Erzeugt ein *Dict*-Objekt, welches das Objekt kapselt, dessen `toDict()`-Slot aktiviert wurde.

A.9.2 Null

Das *Null*-Objekt repräsentiert in der Sprache semantisch *keinen Wert* oder *unbekannten Wert*.

***Konstruktor* ()** Erzeugt ein neues *Null*-Objekt.

===(object) Liefert `true`, sofern das mit *object* bezeichnete Objekt ein *Null*-Objekt ist, ansonsten `false`.

!==(object) Liefert `true`, sofern das mit *object* bezeichnete Objekt kein *Null*-Objekt ist, ansonsten `false`.

==(object) Alias für `===`.

!=(object) Alias für `!==`.

&&(closure) Liefert `false`.

||(closure) Aktiviert die mit *closure* bezeichnete Block-Closure und liefert das Ergebnis von deren Evaluierung.

!() Liefert `true`.

not() Alias für `!`.

type(class) Erzeugt ein neues *Null*-Objekt und verwendet die mit *class* bezeichnete Java-Klasse als Type-Hint. *class* muss dabei ein Objekt des Typs *JClass* sein.

A.9.3 Lst

Das *Lst*-Objekt repräsentiert Arrays der Sprache.

***Konstruktor* ()** Erzeugt ein neues *Lst*-Objekt.

add(value) Erweitert das Array um ein Element und fügt das mit *value* bezeichnete Objekt am Ende ein.

clone() Erzeugt ein neues *Lst*-Objekt.

each(closure) Iteriert über die Elemente des Arrays und aktiviert die mit *closure* bezeichnete Block-Closure für jedes der iterierten Elemente. Erwartet *closure* ein Argument wird dieser der Wert des jeweiligen Elements übergeben. Erwartet die *closure* zwei Argumente wird dieser der Index und der Wert des jeweiligen Elements übergeben.

get(index) Liefert das Objekt, welches sich an der mit *index* bezeichneten Stelle des Arrays befindet. Wirft eine Exception, falls der *index* ungültig ist. *index* muß ein Objekt des Typs *Numeric* sein.

remove(index) Entfernt das an der mit *index* bezeichneten Stelle des Arrays befindliche Objekt. Das Array enthält im Anschluß an diese Operation ein Element weniger. Wirft eine Exception, falls der *index* ungültig ist. *index* muß ein Objekt des Typs *Numeric* sein.

set(index, value) Ersetzt das an der mit *index* bezeichneten Stelle des Arrays befindliche Objekt durch das mit *value* bezeichnete Objekt. Wirft eine Exception, falls der *index* ungültig ist. *index* muß ein Objekt des Typs *Numeric* sein.

size() Liefert die Anzahl der Elemente die das Array enthält.

A.9.4 Dict

Das *Dict*-Objekt kapselt ein beliebiges Objekt der Sprache und ermöglicht dessen konfliktfreie Verwendung als assoziatives Array.

***Konstruktor* (object)** Erzeugt ein neues *Dict*-Objekt welches das mit *object* bezeichnete Objekt kapselt.

each(closure) Iteriert über die Slots des gekapselten Objektes und aktiviert die mit *closure* bezeichnete Block-Closure für jeden iterierten Slot. Dieser werden sowohl der Slot als auch dessen Wert als Argument übergeben.

get(key) Liefert den Wert des mit *key* bezeichneten Slots des gekapselten Objektes. Parent-Objekte desselben werden nicht berücksichtigt.

has(key) Prüft, ob das gekapselte Objekt den mit *key* bezeichneten Slot enthält und liefert entsprechend `true` oder `false`.

remove(key) Enfernt den mit *key* bezeichneten Slot und liefert den damit assoziierten Wert oder `null` falls kein entsprechender Slot existiert.

set(key, value) Setzt den Wert des mit *key* bezeichneten Slots des gekapselten Objektes auf den mit *value* bezeichneten Wert.

A.9.5 Str

Das *Str*-Objekt repräsentiert eine Zeichenkette indem es ein Objekt des Typs *String* der Hostsprache kapselt.

***Konstruktor* (object)** Erzeugt ein neues *Str*-Objekt das die Stringrepräsentation des mit *object* bezeichneten Objektes darstellt. Hierfür wird in der aktuellen Fassung die `toString()`-Methode des jeweiligen nativen Objektes verwendet.

+(string) Erzeugt ein neues *Str*-Objekt. Das Ergebnis entspricht der Konkatenation des *Str*-Objektes, dessen +-Slot aktiviert wurde und der Stringrepräsentation des übergebenen Objektes.

A.9.6 Numeric

Numeric stellt die Basis aller numerische Objekte der Sprache dar. Hierbei wird weitgehend die von Java definierte Semantik realisiert. Insbesondere können Gleitkommazahlen des Typs *Real* den Wert *NaN* (Not a Number) annehmen. Alle arithmethischen Operationen, bei welchen mindestens einer der Operatoren *NaN* ist, liefern wieder *NaN*. Alle Vergleichsoperationen, bei welchen mindestens einer der Operatoren *NaN* ist, liefern `false`.

Im Rahmen arithmetischer Operationen der numerischen Objekte werden, sofern Objekte unterschiedlichen Typs involviert sind, Typkonvertierungen durchgeführt. Der Typ des Ergebnisses der Operationen ist jeweils der Typ, in welchen beide Operanden konvertiert werden können, ohne einen Informationsverlust zu verursachen.

>(number) Führt einen numerischen Größer-Als-Vergleich aus. Liefert entsprechend `true` oder `false`.

>=(number) Führt einen numerischen Größer-Gleich-Vergleich aus. Liefert entsprechend `true` oder `false`.

<(number) Führt einen numerischen Kleiner-Als-Vergleich aus. Liefert entsprechend `true` oder `false`.

<=(number) Führt einen numerischen Kleiner-Gleich-Vergleich aus. Liefert entsprechend `true` oder `false`.

==(object) Prüft auf numerische Gleichheit. Liefert entsprechend `true` oder `false`.

!=(object) Prüft auf numerische Ungleichheit. Liefert entsprechend `true` oder `false`.

===(object) Prüft auf numerische, sowie auf Typgleichheit. Liefert entsprechend `true` oder `false`.

!==(object) Prüft auf numerische, sowie auf Typungleichheit. Liefert entsprechend `true` oder `false`.

+(number) Führt eine arithmetische Addition aus.

-(number) Führt eine arithmetische Subtraktion aus.

***(number)** Führt eine arithmetische Multiplikation aus.

/(number) Führt eine arithmetische Division aus.

bigNumValue() Liefert eine Repräsentation des numerischen Wertes als *BigNum*-Objekt. Hierbei wird eventuell eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

bigRealValue() Liefert eine Repräsentation des numerischen Wertes als *Num*-Objekt.

isInfinity() Liefert `false`.

isNaN() Liefert `false`.

isNegativeInfinity() Liefert `false`.

isPositiveInfinity() Liefert `false`.

neg() Führt eine arithmetische Negation aus.

numValue() Liefert eine Repräsentation des numerischen Wertes als *Num*-Objekt. Hierbei wird eventuell eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

realValue() Liefert eine Repräsentation des numerischen Wertes als *Real*-Objekt. Hierbei wird eventuell eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

toDouble() Liefert ein Objekt des Typs *JObject*, welches den numerischen Wert, konvertiert in einen Wert des Java-Typs *Double*, kapselt. Hierbei wird eventuell eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

toByte() Liefert ein Objekt des Typs *JObject*, welches den numerischen Wert, konvertiert in einen Wert des Java-Typs *Byte*, kapselt. Hierbei wird eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

toFloat() Liefert ein Objekt des Typs *JObject*, welches den numerischen Wert, konvertiert in einen Wert des Java-Typs *Float*, kapselt. Hierbei wird eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

toInteger() Liefert ein Objekt des Typs *JObject*, welches den numerischen Wert, konvertiert in einen Wert des Java-Typs *Integer*, kapselt. Hierbei wird eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

toLong() Liefert ein Objekt des Typs *JObject*, welches den numerischen Wert, konvertiert in einen Wert des Java-Typs *Long*, kapselt. Hierbei wird eventuell eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

toShort() Liefert ein Objekt des Typs *JObject*, welches den numerischen Wert, konvertiert in einen Wert des Java-Typs *Short*, kapselt. Hierbei wird eine Konvertierung vorgenommen, die mit einem Informationsverlust verbunden sein kann.

A.9.7 Num

Das *Num*-Objekt kapselt numerische Werte des Typs `long` der Hostsprache Java.

***Konstruktor* (number)** Erzeugt ein numerisches Objekt. Das mit *number* bezeichnete numerische Objekt wird dabei eventuell konvertiert.

..(number) Erzeugt ein *Range*-Objekt, mit einem Startwert, der dem Wert des Empfängerobjektes entspricht und dem mit *number* bezeichnet Stopwert.

to(number, closure) Iteriert über den numerischen Bereich, beginnend bei dem Wert des Empfängerobjektes bis zu dem mit *number* bezeichneten numerischen Wert. Führt für jeden Iterationsschritt die mit *closure* bezeichnete Block-Closure aus. Dieser wird der aktuelle Wert als Argument übergeben.

valueOf(string) Liefert die numerische Repräsentation des mit *string* bezeichneten Objektes. Entspricht der Java-Methode `Long.valueOf(String)`.

A.9.8 BigNum

Das *BigNum*-Objekt kapselt numerische Werte des Typs `BigInteger` der Hostsprache Java.

***Konstruktor* (number)** Erzeugt ein numerisches Objekt. Das mit *number* bezeichnete numerische Objekt wird dabei eventuell konvertiert.

valueOf(string) Liefert die numerische Repräsentation des mit *string* bezeichneten Objektes. Entspricht der Java-Methode `BigInteger.<init>(String)`.

A.9.9 Real

Das *Real*-Objekt kapselt numerische Werte des Typs `double` der Hostsprache Java.

***Konstruktor* (number)** Erzeugt ein numerisches Objekt. Das mit *number* bezeichnete numerische Objekt wird dabei eventuell konvertiert.

isInfinity() Liefert `true` sonfern der gekapselte numerische Wert positiv oder negativ unendlich ist, sonst `false`.

isNaN() Liefert `true` sofern der gekapselte numerische Wert *NaN* entspricht, sonst `false`.

isNegativeInfinity() Liefert `true` sonfern der gekapselte numerische Wert negativ unendlich ist, sonst `false`.

isPositiveInfinity() Liefert `true` sonfern der gekapselte numerische Wert positiv unendlich ist, sonst `false`.

valueOf(string) Liefert die numerische Repräsentation des mit *string* bezeichneten Objektes. Entspricht der Java-Methode `Double.valueOf(String)`.

A.9.10 BigReal

Das *BigReal*-Objekt kapselt numerische Werte des Typs `BigDecimal` der Hostsprache Java.

***Konstruktor* (number)** Erzeugt ein numerisches Objekt. Das mit *number* bezeichnete numerische Objekt wird dabei eventuell konvertiert.

valueOf(string) Liefert die numerische Repräsentation des mit *string* bezeichneten Objektes. Entspricht der Java-Methode `BigDecimal.<init>(String)`.

A.9.11 Range

Das *Range*-Objekt repräsentiert den numerischen Bereich zwischen zwei *Num*-Werten. Diese sind Teil des Bereiches.

***Konstruktor* (numStart, numStop)** Erzeugt ein neues *Range*-Objekt.

each(closure) Iteriert über jeden Wert des Bereichs und aktiviert die mit *closure* bezeichnete Block-Closure für jeden Iterationsschritt. Der aktuelle Wert wird der Block-Closure jeweils übergeben.

start() Liefert den Startwert des Bereichs.

stop() Liefert den Stopwert des Bereichs.

A.9.12 Bool

Das Bool-Objekt repräsentiert die boolesche Wahrheitswerte.

***Konstruktor* (object)** Liefert `true` oder `false` entsprechend des mit *object* bezeichneten Objektes.

A.9.13 true

Das *true*-Objekt repräsentiert den entsprechenden booleschen Wahrheitswert.

&&(closure) Aktiviert die mit *closure* bezeichnete Block-Closure und liefert das Ergebnis von deren Evaluierung.

||(closure) Liefert `true`.

!() Liefert `false`.

not() Alias für `!`.

A.9.14 false

Das *false*-Objekt repräsentiert den entsprechenden booleschen Wahrheitswert.

&&(closure) Liefert `false`.

||(closure) Aktiviert die mit *closure* bezeichnete Block-Closure und liefert das Ergebnis von deren Evaluierung.

!() Liefert `true`.

not() Alias für `!`.

A.9.15 Func

Func-Objekte repräsentieren Funktionen der Sprache.

call(context, [argument, ...]) Aktiviert die Funktion mit dem übergebenen Kontext und den angegebenen Argumenten.

A.9.16 Clos

Clos-Objekte repräsentieren Block-Closures der Sprache.

whileTrue(closure) Aktiviert die repräsentierte Block-Closure solange diese den Wert `true` liefert und aktiviert in jedem Schritt die mit *closure* bezeichnete Block-Closure.

A.9.17 JArray

Das *JArray*-Objekt kapselt Arrays der Hostsprache.

***Konstruktor* (type, length)** Erzeugt ein *JArray*-Objekt, welches ein Java-Array des mit *type* bezeichneten Typs und der mit *length* bezeichneten Länge enthält.

each(closure) Iteriert über jedes Element des Arrays. Die mit *closure* bezeichnete Block-Closure wird für jeden Iterationsschritt aktiviert. Der Block-Closure werden der Index und der entsprechende Wert oder nur der jeweilige Wert übergeben, je nachdem ob diese zwei oder ein Argument erwartet.

get(index) Liefert das Element an der mit *index* bezeichneten Stelle. Falls *index* ungültig ist, wird eine Exception geworfen.

set(index, value) Weist der Position *index* des Arrays den Wert *value* zu. Falls *index* ungültig ist, wird eine Exception geworfen.

A.9.18 JClass

JClass kapselt Java-Objekte des Typs *Class*. Nachrichten, die an das Objekt gesendet werden, werden an das jeweilige *Class*-Objekt gesendet, sonfern es diese versteht. Hierbei wird die Java-Beans-Konvention weitgehend berücksichtigt.

***Konstruktor* (className)** Erzeugt ein neues *JClass*-Objekt, welches die mit *className* bezeichnete Klasse kapselt.

***Instanzkonstruktor* ([argument, ...])** Erzeugt ein neues Objekt, indem der Konstruktor der gekapselten Klasse aktiviert wird. Das Ergebnis ist ein Objekt des Typs *JObject*, welches das erzeugte Objekt kapselt.

A.9.19 JObject

Das *JObject*-Objekt kapselt beliebige Werte der Hostsprache. Nachrichten, die an das Objekt gesendet werden, werden an das gekapselte Objekt gesendet, sonfern es diese versteht. Hierbei wird die Java-Beans-Konvention weitgehend berücksichtigt.

A.9.20 JMethod

Das *JMethod*-Objekt ist ein aktivierbares Objekt, welches Methoden der Hostsprache kapselt. Objekte diesen Typs werden von den Objekten *JClass* bzw. *JObject* als Antwort auf Nachrichten erzeugt, welche Methoden der jeweils gekapselten Objekte sind.

Bei den gekapselten Methoden handelt es sich um eine oder mehrere gleichnamige Methoden, welche sich ansonsten in ihrer Signatur unterscheiden.

call(context, [argument, ...]) Aktiviert die gekapselte Methode, die anhand der übergebenen Argumente als am passendensten angenommen wird. Wird keine passende Methode gefunden wird eine Exception geworfen. Das mit *context* bezeichnete Objekt stellt das Objekt dar, dessen Methode aktiviert werden soll.

select([typeName, ...]) Erzeugt ein neues *JMethod*-Objekt, welches nur die Methoden enthält, die zu den mit *typeName* spezifizierten Typen passen. Diese werden als *Str*-Objekte erwartet, welche jeweils den Namen einer Java-Klasse oder eines primitiven Java-Typs enthalten. Klassennamen können mit oder ohne den zugehörigen Package-Namen notiert werden.

A.9.21 ConnMgr

Das *ConnMgr*-Objekt ist für die Verwaltung von Datenbankverbindungen verantwortlich. Abschnitt 6.4.2 beschreibt dieses genauer.

active Stellt die aktive Verbindung dar. `null`, falls keine Verbindung vorhanden ist.

activate(conn) Aktiviert die mit *conn* bezeichnete Verbindung. *conn* wird als Objekt des Typs *Conn* erwartet.

create(url, [[user, pass], [driver]]) Erzeugt eine neue Datenbankverbindung mit den angegebenen Parametern. Liefert ein Objekt des Typs *Conn*.

createFromProps(propertiesFilename, [[user, pass], [driver]]) Erzeugt eine neue Datenbankverbindung mit den angegebenen Parametern, die aus der mit *propertiesFilename* bezeichneten Java-Properties-Datei gelesen werden. Liefert ein Objekt des Typs *Conn*.

A.9.22 Conn

Das *Conn*-Objekt repräsentiert eine Datenbankverbindung. Abschnitt 6.4.2 beschreibt dieses genauer.

batch([batchSize], closure) Aktiviert die mit *closure* bezeichnete Block-Closure. Alle von dieser ausgeführten SQL-Befehle werden als Teil eines Befehlsstapels ausgeführt, dessen Größe mit dem optionalen Argument *batchSize* spezifiziert werden kann. Wirft eine Exception falls hierbei ein Fehler auftritt.

begin() Startet eine Transaktion. Ist bereits eine Transaktion aktiv hat dieser Aufruf keine Wirkung. Wirft eine Exception falls ein Fehler auftritt.

close() Schließt die Datenbankverbindung. Wirft eine Exception falls hierbei ein Fehler auftritt.

commit() Schreibt die in der Transaktion vorgenommenen Änderungen fest. Ist keine Transaktion aktiv ist dies ein Fehler. Tritt ein Fehler auf wird eine Exception geworfen.

do(closure) Aktiviert die mit *closure* bezeichnete Block-Closure. Die repräsentierte Verbindung wird für die Dauer der Ausführung der Block-Closure als aktive Verbindung gesetzt.

rollback() Verwirft alle in der Transaktion vorgenommenen Änderungen. Ist keine Transkation aktiv ist dies ein Fehler. Tritt ein Fehler auf wird eine Exception geworfen.

tx(closure) Aktiviert die mit *closure* bezeichnete Block-Closure innerhalb einer Transaktion. Ist bereits eine Transaktion der Verbindung aktiv wird in der aktuellen Fassung eine Exception geworfen. Terminiert die Block-Closure normal wird die Transaktion festgeschrieben. Wird von der Block-Closure jedoch eine Exception geworfen, wird die Transaktion zurückgesetzt. Treten Fehler auf wird eine Exception geworfen.

withPrepared(closure) Aktiviert die mit *closure* bezeichnete Block-Closure. Alle innerhalb dieser erzeugten SQL-Befehlsobjekte unterliegen nicht der automatischen Resourcenverwaltung und werden daher während der Ausführung der Block-Closure nicht automatisch geschlossen. Dies ermöglicht die mehrfache Verwendung parametrisierter Befehlsobjekte. Nach Beendigung der Block-Closure werden alle Resourcen geschlossen.

A.9.23 Stmt

Das *Stmt*-Objekt repräsentiert einen SQL-Befehl. Abschnitt 6.4.2 beschreibt dieses genauer.

Die beschriebenen Routinen werfen jeweils eine Exception, falls ein Fehler auftritt. Desweiteren schließen diese Routinen vor deren Beendigung alle verwendeten Ressourcen, sofern dies nicht anders beschrieben oder die Ressourcenverwaltung vom *ConnMgr*-Objekt bzw. im Rahmen der `withPrepared`-Routine vorgenommen wird.

batch([batchSize], closure) Erlaubt die Stapelausführung eines parametrisierten SQL-Befehls. Aktiviert die mit *closure* bezeichnete Block-Closure. Dieser wird ein Objekt des Typs *ParamBatch* als Argument übergeben.

batchNamed([batchSize], closure) Erlaubt die Stapelausführung eines parametrisierten SQL-Befehls mit benannten Parametern. Aktiviert die mit *closure* bezeichnete Block-Closure. Dieser wird ein Objekt des Typs *NamedParamBatch* als Argument übergeben.

do() Führt den SQL-Befehl aus und benachrichtigt eventuell registrierte Beobachter über das Ergebnis der Ausführung.

each(closure) Führt den SQL-Befehl aus und aktiviert die mit *closure* bezeichnete Block-Closure für jede Zeile der Ergebnismenge. Als Argument wird der Block-Closure ein Objekt des Typs *ResSet* übergeben, welches die Ergebnismenge repräsentiert.

eachKey(closure) Führt den SQL-Befehl aus und aktiviert die mit *closure* bezeichnete Block-Closure für jeden im Rahmen des SQL-Befehls automatisch generierten Schlüssel. Der Block-Closure wird dabei jeweils ein entsprechendes *ResSet*-Objekt übergeben.

exec() Führt den SQL-Befehl aus.

first() Führt den SQL-Befehl aus und liefert ein Objekt, welches der ersten Zeile der Ergebnismenge entspricht. Die Slots des erzeugten Objektes entsprechen den Spaltennamen denen der jeweilige Wert zugeordnet ist. Falls der SQL-Befehl keine Ergebnismenge liefert, wird eine Exception geworfen.

getQueryString() Liefert den SQL-Befehlstext als *Str*-Objekt.

key() Führt den SQL-Befehl aus und liefert den ersten durch diesen Befehl automatisch generierten Schlüssel.

with([parameter, ...]) Aktiviert die Parametrisierung des SQL-Befehls und weist den Parametern die übergebenen Werte zu. Die Aktivierung dieser Routine ist nur gültig, wenn das SQL-Befehlsobjekt noch nicht ausgeführt wurde oder es sich um ein parametrisiertes SQL-Befehlsobjekt mit positionalen Parametern handelt. Im Rahmen dieser Routine werden keine Datenbankressourcen erzeugt oder in Anspruch genommen.

withNamed(object) Aktiviert die Parametrisierung mit benannten Parametern des SQL-Befehls und setzt die, durch das mit *object* bezeichnete Objekt, spezifizierten Parameter. Die Slots des Objektes entsprechen dabei den Namen der Parameter. Die Aktivierung dieser Routine ist nur gültig, wenn das SQL-Befehlsobjekt noch nicht ausgeführt wurde oder es sich um ein parametrisiertes SQL-Befehlsobjekt mit benannten Parametern handelt. Im Rahmen dieser Routine werden keine Datenbankressourcen erzeugt oder in Anspruch genommen.

withPrepared(closure) Aktiviert die mit *closure* bezeichnete Block-Closure. Während der Ausführung dieser Block-Closure ist die automatische Resourcenverwaltung deaktiviert, sodass parametrisierte SQL-Befehle wiederholt ausgeführt werden können.

A.9.24 ResSet

Das *ResSet*-Objekt kapselt die Ergebnismenge eines SQL-Befehls. Alle beschriebenen Routinen werfen im Falle eines Fehlers eine Exception.

each(closure) Iteriert über die vorhandenen Spalten. Die mit *closure* bezeichnete Block-Closure wird für jeden Iterationsschritt aktiviert. Als Argumente werden dieser der Name der jeweiligen Spalte sowie der zugeordnete Wert übergeben.

get(column) Liefert den Wert der mit *column* bezeichneten Spalte. Diese kann hierbei sowohl numerisch oder über deren Namen spezifiziert werden.

insert(closure) Aktiviert die mit *closure* bezeichnete Block-Closure und übergibt dieser ein Objekt des Typs *UpdateHelper*. Mithilfe dieses Objektes können Zeilen in die Ergebnismenge und in die Datenbank eingefügt werden.

update(closure) Aktiviert die mit *closure* bezeichnete Block-Closure und übergibt dieser ein Objekt des Typs *UpdateHelper*. Mithilfe dieses Objektes können einzelne Werte der Ergebnismenge modifiziert werden.

A.9.25 UpdateHelper

Das *UpdateHelper*-Objekt stellt ein Hilfsobjekt dar, mit welchem Werte einer Ergebnismenge verändert oder hinzugefügt werden können.

set(column, value) Setzt den mit *value* bezeichneten Wert in der mit *column* bezeichneten Spalte. Tritt ein Fehler auf oder wird dies nicht unterstützt, wird eine Exception geworfen.

A.9.26 ParamBatch

ParamBatch stellt ein Hilfobjekt dar, welches im Rahmen der Stapelausführung parametrisierter SQL-Befehle verwendet wird. Die beschriebenen Routinen werfen im Fehlerfall eine Exception.

add([parameter, ...]) Fügt dem Stapel eine Menge der spezifizierten Parameter hinzu. Dabei wird der SQL-Befehl, auf welchen sich das *ParamBatch*-Objekt bezieht, mit den bisher definierten Mengen an Parametern ausgeführt, wenn die maximale Große des Stapels erreicht ist.

finish() Führt den SQL-Befehl mit den eventuell vorhandenen, noch nicht berücksichtigten Parametermengen aus.

A.9.27 NamedParamBatch

NamedParamBatch stellt ebenfalls ein Hilfsobjekt dar, welches im Rahmen der Stapelausführung parametrisierter SQL-Befehle mit benannten Parametern verwendet wird. Dieses Objekt ist von *ParamBatch* abgeleitet und implementiert eine entsprechende Semantik.

add(object) Fügt dem Stapel eine Menge der spezifizierten benannten Parameter hinzu (siehe auch `Stmt.withPrepared()`). Die weitere Semantik dieser Routine entspricht der gleichnamigen Routine des *ParamBatch*-Objektes.

A.9.28 Sys

Das *Sys*-Objekt implementiert diverse Basisfunktionalitäten der Sprache. Die Slots dieses Objektes werden in den globalen Namensraum exportiert, sodaß zum Beispiel ein Aufruf von `Sys.print()` identisch mit dem Aufruf `print()` ist.

_break() Beendet das nächste umschließende Schleifenkonstrukt.

_continue() Beendet die aktuelle Iteration des nächsten umschließenden Schleifenkonstrukts und veranlaßt eine weitere Iteration.

_exit([value]) Führt zur sofortigen Beendigung der Ausführung. Sofern vorhanden wird das mit *value* bezeichnete Objekt als Ergebnis der Evaluierung zurückgegeben.

explicitSlot(object, slot) Liefert den Wert des mit *slot* bezeichneten Slots in dem mit *object* bezeichneten Objekt. Dabei werden dessen Parent-Objekte nicht berücksichtigt.

ifThen(condition, trueBranch, [falseBranch]) Aktiviert die mit *trueBranch* bezeichnete Block-Closure, wenn der Wert des mit *condition* bezeichneten Arguments `true` ist. Ansonsten wird die mit *falseBranch* bezeichnete Block-Closure aktiviert, sofern diese übergeben wurde.

includeFile(filename) Evaluiert das mit *filename* bezeichnete Skript so als wäre dessen Inhalt anstelle des `includeFile`-Aufrufs notiert. Relative Pfade beziehen sich dabei auf das aktuell evaluierte Skript.

importPackage(packageName) Importiert das mit *packageName* bezeichnete Java-Package. Im Anschluß an den `importPackage`-Aufruf können die enthaltenen Klassen über deren Namen referenziert werden. Die importierten Klassen werden durch Objekte des Typs *JClass* repräsentiert.

loop(closure) Führt die übergebene Block-Closure wiederholt solange aus bis die Ausführung explizit beendet wird. Dies kann durch den Aufruf von `_break()` oder das Auftreten einer nicht behandelten Exception geschehen.

print([value, ...]) Gibt die Stringrepräsentation der übergebenen Argumente auf der Standardausgabe aus. Dabei wird in der aktuellen Fassung die `toString()`-Methode des jeweiligen nativen Objektes verwendet.

scriptName() Liefert den Namen des aktuell evaluierten Skripts als *Str*-Objekt

scriptResource() Liefert das Resourceobjekt des aktuell evaluierten Skripts als *JObject*-Objekt.

_throw(value) Wirft eine Exception. Das mit *value* bezeichnete Argument kann dabei ein beliebiges Objekt sein.

tryCatch(tryBlock, catchBlock) Aktiviert die mit *tryBlock* bezeichnete Block-Closure. Wird innerhalb dieser eine Exception geworfen, so wird die mit *catchBlock* bezeichnete Block-Closure aktiviert und dieser die aufgetretene Exception als Argument übergeben.

tryCatchFinally(tryBlock, catchBlock, finallyBlock) Aktiviert die mit *tryBlock* bezeichnete Block-Closure. Analog zu `tryCatch` wird die mit *catchBlock* bezeichnete Block-Closure aktiviert, falls eine Exception auftritt. Anschließend wird die mit *finallyBlock* bezeichnete Block-Closure aktiviert.

tryFinally(tryBlock, finallyBlock) Aktiviert die mit *tryBlock* bezeichnete Block-Closure. Nach der Beendigung von deren Ausführung wird die mit *finallyBlock* bezeichnete Block-Closure aktiviert.

B Anwendungsbeispiel

Das vorliegende Anwendungsbeispiel basiert auf einer Datenbank von Artikeln eines Shop-Systems. Dabei kann jeder Artikel aus mehreren Teilen bestehen, die wiederum durch andere Artikel repräsentiert werden. Es handelt sich also um eine rekursive Datenstruktur.

Neben den Artikel existieren verschiedene Lager, in denen jeweils die Bestände aller vorhandenen Artikel vermerkt sind.

Das Beispielskript lädt alle vorhandenen Artikel und berechnet für jeden den tatsächlich vorhandenen Bestand. Zu beachten ist dabei, daß zusammengesetzte Artikel keinen eigenen Bestand haben. Stattdessen wird dieser aus den Beständen der Teile des Artikels berechnet.

Anschließend wird aus den berechneten Beständen ein Bestandsreport generiert, dessen Daten in einer temporären Tabelle abgelegt werden.

Zuletzt wird der generierte Bestandsreport in eine separate Datenbank transferiert.

Listing B.1: Anwendungsbeispiel

```
1  -- Basis-Prototyp fuer den Datensatz einer Datenbanktabelle.
   var ValueObject := {
           init: fun (data) {
                   data.each { slot, value =>
5                          this{slot} = value;
                   };
           }
   };

10 -- Prototyp, der ein Lager repraesentiert.
   var Stock := {
           parent: ValueObject,
           load: fun (id) {
                   (sql select * from stocks where id = ?)
15                 .with(id)
                   .each { row =>
                           return new Stock(row);
                   };
                   return null;
20         },
           loadAll: fun () {
                   var stocks := [];
                   (sql select * from stocks).each { row =>
                           stocks.add(new Stock(row));
25                 };
                   return stocks;
           }
```

```
};

-- Prototyp eines Artikels aus der Datenbank. Artikel
-- koennen rekursiv verschachtelt sein, also andere
-- Artikel enthalten.
var Product := {
        parent: ValueObject,
        init: fun (data) {
                super.init(data);
                this.parts = null;
        },
        isComposite: fun () {
                return this.parts != null;
        },
        load: fun (id) {
                var data := (sql select * from products where id = ?)
                        .with(id)
                        .first();
                return data ? this._build(data) : null;
        },
        loadAll: fun () {
                var products := [];
                (sql select * from products).each { row =>
                        products.add(this._build(row));
                };
                return products;
        },
        _build: fun (data) {
                var product := new Product(data);
                var parts := [];
                (sql select part_id, part_qty
                       from product_parts
                      where product_id = ?)
                .with(product.id)
                .each { row =>
                        var part := this.load(row['part_id']);
                        part.qty = row['part_qty'];
                        parts.add(part);
                };
                this.parts = parts;
                return product;
        }
};

-- Prototyp des Lagerbestands eines bestimmten Artikels.
-- Verschachtelte Artikel werden gesondert behandelt, indem
-- sich deren Bestand aus dem jeweils geringsten Bestand
-- ihrer Teile bestimmt.
var ProductStock := {
        init: fun (product) {
                this.product = product;
                this.quantities = new Dict();
                for (stock : Stock.loadAll()) {
                        var stockQty := stock.clone();
                        stockQty.qty = this.loadStockQty(product, stock);
                        this.quantities[stock.code] = stockQty;
                }
        },
```

```
        loadStockQty: fun (product, stock) {
                if (!product.parts) {
                        (sql select qty
                                from product_stocks
                                where stock_id = ? and product_id = ?)
                        .with(product.id, stock.id)
                        .each { row =>
                                return row['qty'];
                        };
                        throw "Product " + product.id + " not found " +
                                "in stock " + stock.id;
                }

                var minStock = null;
                product.parts.each { part =>
                        var partStock := this.loadStockQty(part, stock)
                                        / part.qty;
                        minStock = minStock && minStock < partStock
                                ? minStock
                                : partStock;
                };

                if (!minStock) {
                        throw "Internal error: No parts " +
                                "in product " + product.id;
                }

                return minStock;
        }
};

-- Tabelle fuer den Bestandsreport vorbereiten. In dieser
-- werden die Bestaende aller Artikel, auch zusammengesetzter
-- Artikel abgelegt.
drop table stock_report;
create temporary table stock_report (
        id int not null primary key serial,
        product_number varchar(32) not null,
        composite boolean not null,
        stock_code varchar(32) not null,
        stock_quantity int not null
        unique (product_number)
);

-- Bestandsreporttabelle in einer Transaktion befuellen.
.ConnMgr.tx {=>
        (sql insert into stock_report
                (product_number, composite, stock_quantity)
                values (?, ?, ?))
        .batch(100) { batch =>
                Products.loadAll().each { product =>
                        var stock := new ProductStock(product);
                        stock.quantities.each { stockCode, stock =>
                                batch.add(
                                        product.number,
                                        product.isComposite(),
                                        stockCode,
                                        stock.qty
```

```
                                        );
                                };
                        };
                };
        };

        -- Stmt-Objekt, das als Prototyp fuer jedes SQL-Befehlsobjekt dient
        -- um Hilfsroutine fuer den Datentransfer erweitern.
        {
                -- Der Code ist in einen separaten Gueltigkeitsbereichsblock
                -- geschachtelt, sodass die Funktion time, die nur
                -- hier benoetigt wird, nicht im globalen Gueltigkeitsbereich
                -- sichtbar wird.

                -- Timer-Funktion, die die Ausfuehrungszeit einer uebergebenen
                -- Funktion misst.
                fun time(action) {
                        import java.util.Date;

                        var start := new Date();
                        action();
                        var end := new Date();

                        return (end.timeInMillis - start.timeInMillis) / 1000;
                }

                -- Stmt-Objekt erweitern.
                Stmt.| = fun (dstStmt) {
                        var recordCount := 0;

                        var xferTime := time {=>
                                dstStmt.batch(1000) { batch =>
                                        this.each { row =>
                                                batch.add(row);
                                                recordCount = recordCount + 1;
                                        };
                                };
                        };

                        var recordsPerSec := recordCount / xferTime;

                        print("Transferred @{recordCount} records " +
                                "in @{xferTime} seconds.");
                        print("That is @{recordsPerSec} records per second.");

                        return null;
                };
        }

        -- Quellbefehl fuer Datentransfer erzeugen.
        var srcStmt := sql select * from stock_report;

        -- Weitere Datenbankverbindung zur Reportdatenbank aufbauen.
        var connReport := ConnMgr.create('...');

        -- Folgenden Code in einer Umgebung ausfuehren, in welcher
        -- die Verbindung zur Reportdatenbank als aktive Verbindung
        -- eingerichtet ist. Alle darin erzeugten SQL-Befehle
```

```
    -- beziehen sich auf diese Verbindung.
    .connReport.do {=>
        -- Oracle-spezifische Zeichenkettensyntax aktivieren.
205     \set quotes=oracle;

        -- Zielbefehl fuer Datentransfer erzeugen.
        var dstStmt := sql insert into stock_reports
                            (product_number, composite,
210                         stock_code, stock_quantity)
                       values
                            (?, ?, ?, ?);

        ConnMgr.tx {=>
215         -- Zieltabelle vorbereite.
            sql truncate table stock_reports;

            -- Datentransfer durchfuehren.
            srcStmt | dstStmt;
220     };
    };
```

Literaturverzeichnis

[ADB08] D. Alhadidi, M. Debbabi, and P. Bhattacharya. New AspectJ Pointcuts for Integer Overflow and Underflow Detection. *Inf. Sec. J.: A Global Perspective*, 17(5-6):278–287, 2008.

[AEU99] Alfred V. Aho, Ravi Ethis, and Jeffrez D. Ullmann. *Compilerbau Teil 1.* Oldenbourg, Wien, 2 edition, 1999.

[AFM+95] Zena M. Ariola, Matthias Felleisen, John Maraist, Martin Odersky, and Philip Wadler. A Call-by-Need Lambda Calculus. In *Proceedings of 22nd Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 233–246, 1995.

[App07] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, NY, USA, 2007.

[AS96] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 1996.

[Bak93] Henry G. Baker. Iterators: signs of weakness in object-oriented languages. *SIGPLAN OOPS Mess.*, 4(3):18–25, 1993.

[Bak95] Henry G. Baker. CONS should not CONS its arguments, part II: Cheney on the M.T.A. *SIGPLAN Not.*, 30(9):17–20, 1995.

[Bor90] Richard Bornat. *Understanding and writing compilers: a do-it-yourself guide.* Macmillan Publishing Co., Inc., Indianapolis, IN, USA, 1990.

[Cri89] Flaviu Cristian. Exception handling. In *Dependability of Resilient Computers*, pages 68–97, 1989.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[DBGW03] Brian Davis, Andrew Beatty, David Gregg, and John Waldron. The Case for Virtual Register Machines. In *Interpreters, Virtual Machines and Emulators (IVME '03*, pages 41–49. ACM Press, 2003.

[DMC92] Christophe Dony, Jacques Malenfant, and Pierre Cointe. Prototype-based languages: from a new taxonomy to constructive proposals and their validation. *SIGPLAN Not.*, 27(10):201–217, 1992.

[FW08] Daniel P. Friedman and Mitchell Wand. *Essentials of Programming Languages.* MIT Press, Cambridge, MA, third edition, 2008.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison-Wesley Professional, 1995.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Gou01] K John Gough. Stacking them up: a comparison of virtual machines. *Aust. Comput. Sci. Commun.*, 23(4):55–61, 2001.

[GR02] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* 2002.

[ISO92] International Organization for Standardization, ISO. ISO/IEC 9075:1992, Database Language SQL, Second Informal Review Draft. `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt`, 1992.

[ISO99a] International Organization for Standardization, ISO. ISO/IEC 9075-2:1999, Database Language SQL – Part 2: Foundation (SQL/Foundation). `http://www.ncb.ernet.in/education/modules/dbms/SQL99/ansi-iso-9075-2-1999.pdf`, 1999.

[ISO99b] International Organization for Standardization, ISO. ISO/IEC 9075-5:1999, Database Language SQL — Part 5: Host Language Bindings (SQL/-Bindings). `http://www.ncb.ernet.in/education/modules/dbms/SQL99/ansi-iso-9075-5-1999.pdf`, 1999.

[Koo89] Philip J. Koopman, Jr. *Stack computers: the new wave.* Halsted Press, New York, NY, USA, 1989.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, New York, NY, USA, 1986. ACM Press.

[MOW98] John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8:275–317, 1998.

[Ora08a] Oracle Corporation. Oracle® Database PL/SQL Language Reference. `http://download.oracle.com/docs/cd/B28359_01/appdev.111/b28370/toc.htm`, 2008.

[Ora08b] Oracle Corporation. Oracle® Database Programmer's Guide to the Oracle Precompilers. `http://download.oracle.com/docs/cd/B28359_01/appdev.111/b31231/toc.htm`, 2008.

[Ora08c] Oracle Corporation. Oracle® Database SQL Language Reference. `http://download.oracle.com/docs/cd/B28359_01/server.111/b28286/toc.htm`, 2008.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, Raleigh, 2007.

[Pos08] The PostgreSQL Global Development Group. PostgreSQL 8.3.6 Documentation. `http://www.postgresql.org/docs/8.3/static/index.html`, 2008.

[Sav08] Anthony Savidis. An enhanced form of dynamic untyped object-based inheritance. *Journal of Object Technology*, 7(4):101–122, 2008.

[SGBE05] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: Stack versus registers. In *In Proc. ACM/USENIX Int. Conf. on Virtual Execution Environments*, pages 153–163. ACM Press, 2005.

[Smi94] Walter Smith. Class-based NewtonScript Programming. *PIE Developers*, 1994.

[Smi95] Walter R. Smith. Using a prototype-based language for user interface: the Newton project's experience. *SIGPLAN Not.*, 30(10):61–72, 1995.

[SO01] Michel Schinz and Martin Odersky. Tail call elimination on the Java Virtual Machine. In *ACM SIGPLAN BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability*, pages 155–168. Elsevier, 2001.

[Sun09] Sun Microsystems, Inc. MySQL 5.0 Reference Manual. `http://dev.mysql.com/doc/refman/5.0/en/index.html`, 2009.

[Tur36] Alan M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society. Series 2.*, 42:230–265, 1936. Online: `http://www.abelard.org/turpap2/tp2-ie.asp`.

[US87] David Ungar and Randall B. Smith. Self: the power of simplicity. *SIGPLAN Not.*, 22(12):227–242, 1987.

[War07] Daniel Warner. *Advanced SQL*. Franzis Verlag, 2007.

Alle Weblinks wurden am 19. August 2009 auf ihre Aktualität geprüft.

Glossar

Aktivierung	Im Rahmen dieser Arbeit verwendete Bezeichnung für den Vorgang des Aufrufs bzw. der Ausführung einer Funktion durch die Maschine., 25
Aktivierungssegment	Speicherbereich, welcher die mit einem Funktionsaufruf verbundenen Daten wie Argumente, lokale Variablen, u.a. enthält., 39
Bindung	Zuordnung eines Namens zu einem Wert., 26
Closure	Kombination aus einer Funktion höherer Ordnung und deren freien Variablen. Diese bleiben über die Lebensdauer der Closure gültig., 32
Continuation	Im Continuation-Passing-Style verwendete Bezeichnung für den Kontrollkontext, in dem eine Funktion ausgeführt wird. Entspricht der dieser Funktion folgenden Berechnung des betreffenden Programmes., 35
Continuation-Passing-Style	Notationsform für Programme, bei welcher diese als Folge von endrekursiven Funktionen dargestellt werden., 34
de Bruijn Index	siehe *Lexikalische Adresse*., 28
Differentielle Vererbung	Eine Technik, mit welcher in der prototyp-basierten Objektorientierung eine Vererbungsbeziehung zwischen zwei Objekten im Rahmen der Klonierungsoperation hergestellt wird, indem ein neues Objekt erzeugt und mit einer Delegationsreferenz auf das geklonte Objekt versehen wird., 20
Dynamische Bindung	Bezeichnet im Kontext der Zielsprache eine Bindung, welche zur Laufzeit eines Programmes erzeugt wird. Nicht zu verwechseln mit dynamischer Bindung im Kontext der objektorientierten Programmierung., 73

Ex-Nihilo	Art der Objekterzeugung bei der prototyp-basierten Objektorientierung. Objekte werden dabei *'aus dem Nichts'* als leere Objekte ohne Parent-Objekt erzeugt., 19
Gültigkeitsbereich	Bereich eines Programmes, auf welchen eine Deklaration eines Namens angewandt wird., 24
Hostsprache	Die Sprache, in welcher die Zielsprache implementiert ist. Java., 70
Hostsystem	Das Basissystem, welches die Arbeitsumgebung des Interpreters darstellt und die fur diesen relevante Basisfunktionalitat zur Verfügung stellt, 2
Impliziter Parent Slot	In der Zielsprache eine Delegationsreferenz, die vom System festgelegt ist und nicht über die Slots der Objekte reflektiert wird., 54
JVM	Java Virtual Machine – Virtuelle Stapelmaschine die der Ausführung von Java-Programmen bzw. -Bytecodes dient., 34
Lexikalische Adresse	Im Rahmen der lexikalischen Adressierung verwendete Adresse eines Namens in einer Umgebung. Diese kann zum Zeitpunkt der Übersetzung eines Programmes gebildet werden., 28
Lexikalische Adressierung	Implementierungstechnik für Umgebungen unter Verwendung von lexikalischen Adressen., 28
Multiple Dispatch	Dynamic-Dispatch-Mechanismus, mit welchem die Auswahl infrage kommender Methoden im Rahmen eines Methodenaufrufes anhand der zur Laufzeit bekannten Typen aller Argumente des Methodenaufrufs durchführt wird., 78
Native Schnittstelle	Die Schnittstelle der Maschine der Zielsprache, welche von nativen Objekten für die Interaktion mit der Maschine genutzt werden kann., 71
Natives Objekt	Ein Objekt der Zielsprache, welches in der Hostsprache implementiert ist., 70

Nebenwirkung	Im Kontext von Programmiersprachen werden Funktionen bzw. Prozeduren als mit Nebenwirkungen behaftet bezeichnet, wenn diese neben der Rückgabe eines Wertes den Zustand eines Computersystems verändern oder anderweitig mit der Ausenwelt interagieren. Beispiele von Nebenwirkungen sind die Veränderung von Variablen oder Ein-/Ausgabeoperationen., 17
Overload Resolution	Vorgang der Auswahl einer spezifischen Methode anhand der Typen der übergebenen Argumente., 78
Parent Slot	In der Zielsprache eine spezieller Slot von Objekten, welcher als Delegationsreferenz dient., 50
Parse Direktive	In der Zielsprache eine Anweisung, mit welcher das Verhalten des Parsers beeinflußt werden kann., 60
PL/SQL	*Procedural Language/SQL*, eine proprietäre Programmiersprache der Firma Oracle, die SQL um Eigenschaften prozeduraler und objektorientierter Programmiersprachen erweitert., 8
Pseudobefehl	In der Zielsprache Befehle, welche bei der Übersetzung in äquivalente Ausdrücke überführt werden., 43
Single Dispatch	Dynamic-Dispatch-Mechanismus, mit welchem die Auswahl infrage kommender Methoden im Rahmen eines Methodenaufrufes anhand des zur Laufzeit bekannten Typen eines Argumente des Methodenaufrufs durchführt wird., 78
Slot	Im Kontext der prototyp-basierten Objektorientierung eine Nachricht, die ein Objekt versteht., 50
syntactic sugar	siehe *Syntaktischer Zucker.*, 21
Syntaktischer Zucker	Syntaktische Konstrukte in Programmiersprachen, durch welche ein einfacherer Umgang mit diesen erreicht oder deren Ausdrucksstärke erhöht werden soll. Die Funktionalität der Sprache wird von diesen Konstrukten nicht beeinflußt, da jeweils äquivalente Konstrukte existieren., 21

Tail-Call-Optimierung	Behandlung von endrekursiven Funktionsaufrufen, durch welche beliebig tiefe Rekursionen mit konstantem Speicherbedarf ausgeführt werden können., 37
Umgebung	Funktion, die die Abbildung von Namen auf Werte beschreibt., 27

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 19. August 2009	
Ort, Datum	Unterschrift