Konzepte für die Einbettung von XQuery in eine Wirtssprache (Java)

Andreas Bühmann

27. April 2003

Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr. Dr. Theo Härder

Diplomarbeit

Konzepte für die Einbettung von XQuery in eine Wirtssprache (Java)

Andreas Bühmann*

27. April 2003

Betreut von Marcus Flehmig†

*buehmann@informatik.uni-kl.de
†flehmig@informatik.uni-kl.de

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Kaiserslautern, den 27. April 2003

Andreas Bühmann

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Arbeit mit XML-Daten erfolgt bisher oft dokumentorientiert und auf relativ tiefer Ebene. Selbst bei datenorientierter Sichtweise auf XML, also bei Interesse an den im XML-Dokument gespeicherten Daten, nicht etwas seiner Dokumentstruktur oder Reihenfolge, werden typischerweise bestimmte Dokumente ausgewählt und am Stück verarbeitet, z. B. mit verschiedenen Methoden eingelesen. Eine Berücksichtigung von verschiedenen Datentypen in XML-Dokumenten erfolgt zum Teil nur durch explizite Umwandlung von Zeichenketten innerhalb der Anwendung. Zur Auswahl und teilweise sogar Ausschnittsbildung von XML-Dokumenten stehen Pfadausdrücke aus XPath bereit. Eine darüber hinausgehende Unterstützung der Arbeit mit großen Datenbanken in XML-Form fehlt; es existiert keine Anfragesprache, die Inhalte mehrerer Dokumente verknüpfen, umstrukturieren und in getypter Form zugänglich machen kann.

Diese Lücke füllt die Sprache XQuery, die sich zur Zeit auf dem Weg zur Standardisierung durch das World-Wide-Web-Consortium befindet. Um sinnvoll eingesetzt werden zu können, muss eine Anfragesprache begleitet werden von einer oder mehreren Anwendungsprogrammierschnittstellen. Konzepte für eine solche sollen im Folgenden erarbeitet werden am Beispiel der bei der XML-Verarbeitung vorherrschenden Programmiersprache Java. Dabei wird ein Ansatz zur Einbettung von XQuery in Java im Mittelpunkt stehen.

1.1 Aufbau

Kapitel 2 ruft die benötigte Begriffswelt rund um XML in Erinnerung und wiederholt die wichtigsten Techniken, die zur Zeit für den Zugriff auf und die Verarbeitung von „XML-Daten“ existieren. Dabei wird neben den einführend genannten Beschränkungen auch deutlich, wie sich der Begriff der XML-Daten schrittweise entwickelt hat. Das Kapitel schließt mit einer kurzen Betrachtung der Kopplung von Datenbank- und Programmiersprachen im Allgemeinen.

Kapitel 3 führt die Sprache XQuery sowie das ihr zugrunde liegende Datenmo-

dell ein. Die Darstellung von XQuery erfolgt dabei entlang der Benutzersyntax und setzt keine Vorkenntnisse von XQuery voraus.

Kapitel 4 nähert sich der Kopplung von XQuery und Java mit dem Konzept der aufeinander aufbauenden Anfragen. Dieses erlaubt unter anderem eine anwendungsbezogene Sichtenbildung sowie parametrisierbare Anfragen.

Kapitel 5 kümmert sich dann um Realisierungsansätze für die vorgeschlagene Kopplung, beginnend bei der syntaktischen Erweiterung von Java bis hin zu Überlegungen für eine Umsetzung der Semantik. Dabei tritt eine Schwierigkeit bei der Materialisierung von Zwischenergebnissen zu Tage, die zur Zeit eine Implementierung aufbauend auf frei verfügbaren XQuery-Implementierungen verhindert.

Kapitel 6 fasst schließlich die Ergebnisse dieser Arbeit zusammen und führt alle diejenigen Punkte auf, an denen weiter gearbeitet werden kann.

2 Grundlagen

2.1 XML und seine Verarbeitung

2.1.1 Die X-Begriffe

Bei der Beschäftigung mit Themen im Umfeld von XML wird man schnell von einer ganzen Palette von Konzepten und Technologien konfrontiert, die jeweils einen bestimmten Teilbereich des Umgangs mit XML abdecken. Fast jede dieser Technologien hat ein „X“ im Namen, und die meisten sind durch das World-Wide-Web-Consortium (W3C) spezifiziert.

Auch im Folgenden wird auf solche X-Begriffe und durch sie bezeichnete Technologien immer wieder Bezug genommen. Deswegen soll dieser Abschnitt einen knappen Überblick über die für diese Arbeit relevanten Spezifikationen sowie ihre Zusammenhänge geben.

XML

Grundlage aller anderen Technologien ist *XML* (W3C 2000b), eine Metasprache zur flexiblen und bei Bedarf tief geschachtelten Strukturierung von Textdokumenten, vor allem durch so genannte Elemente und Attribute, die innerhalb eines Dokuments Metadaten transportieren. Durch Festlegung von Elementtypen und der Beschreibung ihrer Beziehungen, d. h. Reihenfolge und Art der Schachtelung, können Anwendungssprachen aus XML instanziiert werden, von denen auch mehrere in einem Dokument verwendet werden können. Um dabei Namenskonflikten aus dem Weg zu gehen, wurde das Konzept der *Namensräume* (Namespaces) in XML eingeführt (W3C 1999a). Das bedeutet nicht viel mehr als eine Aufteilung von XML-Namen in zwei Komponenten, einen Namensraum-Bezeichner (eine URI) und einen lokalen Namen. Ein solcher zweiteiliger Name heißt *qualifizierter Name* (*QName*).

XML Infoset

Während in der XML-Spezifikation nur eine Syntax für XML-Dokumente spezifiziert wird, beschreibt das *XML Information Set*, kurz *Infoset* (W3C 2001a), abstrakt die Menge an Informationen, die in einem XML-Dokument enthalten ist, das als Zeichenkette vorliegt[1]. Diese Informationen sind reihenfolgeerhaltend in einer Baumstruktur mit verschiedenen Knotentypen (Information Items) angeordnet, z. B. Dokument-, Element- oder Text-Knoten. Dabei haben die eigentlichen Daten in Text- und Attribut-Knoten keinen Typ[2]; alles ist eine Zeichenkette.

Das Infoset abstrahiert vollständig von der physischen Struktur eines XML-Dokuments. Dessen Aufteilung in verschiedene einander referenzierende Teile (Entities) ist nicht sichtbar; beim Übergang zum Infoset werden diese vollständig expandiert.[3] Ebenfalls irrelevant für das Infoset eines Dokuments sind z. B. die Reihenfolge von Attributen eines Elements oder die Darstellung von Zeicheninhalt (direkt, via Zeichenreferenz oder in einem CDATA-Abschnitt).

Die Infoset-Spezifikation stellt ein grundlegendes Datenmodell für XML dar. Andere Datenmodelle erweitern meist das Infoset um weitere Eigenschaften und Informationen, so auch das Datenmodell zu XQuery (vgl. später Abschnitt 3.2 auf Seite 14).

XML Schema

Die Typisierung der Inhalte von XML-Dokumenten ermöglicht die Schemasprache *XML Schema* (W3C 2001c, d, e). Sie bietet eine Reihe von eingebauten Basistypen; von diesen lassen sich durch Erweiterung oder Beschränkung benutzerdefinierte einfache Typen ableiten. Zur Beschreibung des erlaubten Inhalts von Elementen können neben einfachen auch so genannte komplexe Typen eingesetzt werden. Das sind solche Typen, die eine innere Struktur aufweisen und aus anderen Typen zusammengesetzt sind. Zur Strukturierung stehen reguläre Ausdrücke zur Verfügung, also prinzipiell die Mächtigkeit regulärer (formaler) Sprachen.

[1] Man spricht auch von serialisierter Form oder einer *Serialisierung* des XML-Dokuments. Mit dem Begriff XML-Dokument wird seit der Einführung des Infoset zunehmend diese abstrakte Form assoziiert.

[2] Genau genommen sind in der Dokumenttyp-Definition *DTD* gewisse Unterscheidungen in verschiedene Arten von Zeichenketten möglich. Als ein Typsystem kann man das aber kaum bezeichnen.

[3] Nur falls die Expansion nicht möglich ist, wird dies im Infoset vermerkt.

Auch einfache Integritätsbedingungen sind formulierbar wie etwa die Eindeutigkeit eines Schlüssels.

Typ- und Element-Deklarationen eines Schemas können einen QName erhalten, um aus anderen Dokumenten und Schemata referenziert werden zu können, ansonsten sind sie anonym oder nur im definierten Kontext gültig.

XML Schema definiert neben der Schemasprache auch einen Mechanismus der *Validierung* von Instanzdokumenten gegen Schemabeschreibungen. Bei diesem Prozess wird abstrakt das Infoset eines Dokument erweitert, indem Anmerkungen zur Gültigkeit und im Falle der Gültigkeit zum Typ von Knoten eingefügt werden. Dieses angereicherte Infoset wird als *Post Schema Validation Infoset (PSVI)* bezeichnet.

XPath

Die Auswahl von Teilen eines XML-Dokuments erleichtert die Sprache *XPath* (W3C 1999b) mit Hilfe von *Pfadausdrücken* (vgl. auch Abschnitt 3.4.2 auf Seite 29). Dabei werden in mehreren Schritten entlang verschiedener Achsen in der Baumstruktur Knoten aufgrund ihres Namens, ihres Inhalts oder ihrer Beziehung zu anderen Knoten selektiert.

XQuery

Die Erweiterung des einfachen XPath zu einer vollwertigen Anfragesprache, die auch die Verknüpfung mehrerer Datenquellen und sogar Umstrukturierungen zulässt, nennt sich *XQuery* und wird in Kapitel 3 auf Seite 11 detailliert vorgestellt. XQuery und die neue Version 2.0 von XPath (W3C 2002a) haben eine große Schnittmenge und werden momentan zusammen durch das W3C entwickelt und standardisiert.

2.1.2 Verarbeitung von XML

In der Vergangenheit haben sich verschiedene Möglichkeiten zum Zugriff auf und zur Verarbeitung von XML-Daten entwickelt. Dabei wurde der Begriff XML oder XML-Dokument schrittweise erweitert und abstrahiert, ausgehend von einer spezifischen Syntax über eine Menge von Textinformationen in Baumstruktur bis hin zu einer abstrakten und komplex getypten Datenstruktur.

SAX

Die primitivste Standardschnittstelle zum Zugriff auf XML-Daten stellt *SAX* dar, das *Simple API for XML* (SAX 2002). SAX orientiert sich eng an der ursprünglichen Definition von XML als spezielle Sprache (vgl. Abschnitt 2.1.1) und bietet nicht mehr als eine ereignisbasierte Schnittstelle zu einem Parser dieser Sprache: Beim linearen Lesen eines kompletten XML-Dokuments werden dessen Grundbausteine als Ereignisse an die Anwendung gemeldet. Typische Beispiele für solche Ereignisse sind Start und Ende eines Dokuments oder Elements oder das Auftreten von Zeicheninhalt.

Auf der Ebene von SAX gibt es (fast) keine Typen, jeder Inhalt eines Elements oder Attributs ist eine Zeichenkette. Schemainformationen werden nur optional dazu benutzt, Verletzungen dieses Schemas zu erkennen und das Parsen daraufhin nicht erfolgreich zu beenden.

DOM

Das *Document Object Model (DOM)* definiert eine abstrakte Menge von generischen objektorientierten Schnittstellen, mit denen in baumstrukturierten Dokumenten navigierend gearbeitet werden kann (z. B. W3C 2000a). Zu solchen Dokumenten zählen z. B. XML- aber auch HTML-Dokumente. Im XML-Fall bietet DOM quasi eine Schnittstelle zum Zugriff auf das Infoset eines XML-Dokuments. DOM erlaubt auch schreibende Zugriffe auf Dokumente sowie den Aufbau neuer Dokumente.

Für die Schnittstellen von DOM liegen zahlreiche Bindungen an konkrete Programmiersprachen vor.

Gern genannt wird der extrem erhöhte Hauptspeicherbedarf bei der Verwendung von DOM im Gegensatz zu SAX. Das ist jedoch zunächst keine Eigenschaft von DOM an sich, sondern der bestehenden Implementierungen, die der Verarbeitungsflexibilität und der Einfachheit halber immer komplette Dokumente einlesen und als Baumstruktur im Hauptspeicher halten. Allerdings wird DOM auch bei geschickter Realisierung durch die Nichtvorhersagbarkeit des Zugriffs (es ist möglich, kreuz und quer in der Baumstruktur eines Dokuments herumzuspringen) sicher immer einen höheren Ressourcenbedarf haben als SAX.

Die Mächtigkeit von DOM ist mit der Zeit in so genannten Ebenen (Level) gewachsen. In der Skizze zu DOM Level 3 (W3C 2003) ist auch der Zugriff auf Typinformationen aus dem PSVI eines XML-Dokuments möglich.

XML Data Binding

Eine recht neue Entwicklung auf dem Gebiet der XML-Verarbeitung stellen so genannte *Data-Binding*-Frameworks wie Castor XML (Castor 2003) oder JAXB (JAXB 2003) dar. Diese konzentrieren sich auf datenorientierte XML-Anwendungen, bei denen die im Dokument enthaltenen Daten im Vordergrund stehen. Oft spielt dabei Reihenfolge keine Rolle und die betrachteten Dokumente haben (als Daten-Container) eine rechte homogene Struktur. (Sosnoski 2002; Steinberg und Thinking 2002)

Die Idee des Data Binding ist die Überführung eines XML-Dokuments in einen Graphen von anwendungsnäheren Objekten als z. B. bei DOM.[4] Ein Graph statt eines Baumes entsteht durch die Abbildung von wertbasierten Beziehungen im XML-Dokument auf Assoziationen oder konkreter Referenzen zwischen Objekten. Objekte können dabei kleine Teilbäume aus dem XML-Dokument zusammenfassen, die in der Anwendungswelt ein zusammenhängendes Konzept beschreiben. Zum Beispiel könnte ein Person-Element mit Geburtsdatum-Attribut und Namen als Kindelement als ein Objekt mit entsprechenden Attributen dargestellt werden. Zur Beschreibung der Abbildung eines XML-Schemas auf eine Klassenstruktur steht bei Castor eine Abbildungssprache zur Verfügung; allerdings kennt das Framework auch Standardabbildungen, die sich an Namenskorrelationen zwischen XML-Elementen und -Attributen und Klassen und Methoden orientieren.

Durch Verwendung eines Data-Binding-Frameworks wird ein gewisser Grad an Datenunabhängigkeit erreicht, denn beispielsweise ist es bei geeigneter Abbildung für den Anwendungsprogrammierer egal, ob bestimmte Attribute im XML-Schema als (XML-)Attribut oder als Kindelement dargestellt oder wie Beziehungen modelliert sind.

Änderungen werden vom Konzept her nur insoweit unterstützt, dass ein modifizierter (oder neu aufgebauter) Objekt-Graph in ein *neues* XML-Dokument überführt wird. Ein Propagieren nur der geänderten Teile in das Ursprungsdokument ist (zumindest bei Castor) nicht vorgesehen.

[4] Diese Operation wird im Englischen mit der Begriffskonstruktion „unmarshal“ beschrieben, die umgekehrte Transformation von einem Objekt-Graphen in ein XML-Dokument mit „marshal“ (engl. für „anordnen“ oder auch „(in Linie) antreten lassen“; also im Wortsinn ähnlich zu „Serialisierung“).

2.2 Kopplung von Datenbank- und Programmiersprachen

Bevor in den folgenden Kapiteln konkret auf XQuery und seine Kopplung mit Java eingegangen wird, soll in diesem Abschnitt ein kurzer Blick auf allgemeine Freiheitsgrade bei der Kopplung einer Datenbanksprache mit einer Programmiersprache geworfen werden.

Datenbanksprachen (DBL) als Sprachen, die zur Selektion und Manipulation von Daten entworfen wurden, sind meist nicht als so mächtig konzipiert, dass sie universell einsetzbar sind. Im Allgemeinen ist es nicht möglich, in einer Datenbanksprache komplette Anwendungsprogramme zu formulieren. Eine Datenbanksprache wird eher als Zusatzsprache zu einer beliebigen Anwendungsprogrammiersprache (PL) gesehen und ist deshalb meist mit einem eigenen Datenmodell und Typsystem versehen. Diese von den Strukturen einer konkreten Programmiersprache, aber auch von der Speicherung der Daten an sich entkoppelte Stellung der Datenbanksprachen bewirkt eine Erhöhung der Datenunabhängigkeit.

Zur Erstellung von Anwendungen ist dementsprechend die Kopplung einer Datenbanksprache mit der gewählten Programmiersprache (Wirtssprache) notwendig. Es ist also eine Anwendungsprogrammierschnittstelle (API) zu entwerfen, die aus der PL heraus die Verwendung der DBL ermöglicht.

Solche APIs können nach der Tiefe der Integration der DBL in die PL klassifiziert werden (Härder u. a. 2000):

Aufrufschnittstellen benutzen die Mittel der reinen Wirtssprache zur Realisierung der Kopplung, d. h. die Übergabe von Anfragen in DBL und die Entgegennahme der Ergebnisse erfolgt über Funktionsaufrufe in entsprechende Laufzeitbibliotheken. Anfragen werden dabei meist als einfache Stringparameter übergeben.

Spracherweiterungen modifizieren die Wirtssprache W zu einer Sprache W'. Je nach Tiefe der Einbindung von DBL in PL spricht man auch von Einbettung oder Integration. Zur Übersetzung von Programmen in W' ist entweder ein Präcompiler von Nöten, der die DBL-Anteile analysiert und das Programm nach W übersetzt, oder ein direkter Compiler für die integrierte Sprache W'. Beide Varianten sind natürlich äquivalent, ein Präcompiler wird aber bei nicht zu tiefer Einbettung einfacher zu realisieren sein.

Ein Beispiel für eine Spracherweiterung bei der Nutzung eines Präcompilers stellt SQLJ dar (ISO 2002b, a; Saake und Sattler 2000), das SQL in Java einbettet. Dabei werden SQL-Anweisungen mit dem Präfix `#sql` markiert und in Java-Programme eingebettet. Der Datenaustausch zwischen SQL und Java wird dann über Wirtsvariablen in den SQL-Anweisungen sowie die Bindung von Anfrageergebnissen an speziell zu deklarierende Iteratoren realisiert.

Eine andere Möglichkeit der Variation in der Anbindung einer DBL ist die Wahl der so genannten Bindezeitpunkte. Ein Bindezeitpunkt markiert die Stelle im Lebensweg einer DBL-Anweisung, an dem sie unter Einbeziehung weiterer Information transformiert und der Ausführung einen Schritt näher gebracht wird. Dabei wird sie an diese zusätzlichen Informationen gebunden, also von ihnen abhängig.

Eine frühzeitige Analyse und Transformation ermöglicht eine frühe Fehlererkennung und verschiebt den zu leistenden Aufwand zeitlich nach vorn. Besonders günstig ist diese Verschiebung, wenn die Grenze zwischen Lauf- und Compilezeit überschritten wird, da dann der Aufwand nur einmal und nicht mehrfach zu leisten ist. Allerdings ist das Ergebnis einer solchen Transformation nur so lange gültig und verwendbar, wie sich die dazu benutzten Voraussetzungen nicht ändern.

Dieses allgemeine Prinzip sorgt im Falle der DBL bei früher Bindung für eine höhere Datenabhängigkeit, aber auch höhere Effizienz. Späte Bindung ist aufwändiger, aber führt weniger Abhängigkeiten ein.

3 XQuery

3.1 Einführung

XQuery wird als Anfragesprache für XML zur Zeit vom World-Wide-Web-Consortium entwickelt. XML wird dabei als Sprache verstanden, die den Inhalt vieler Datenquellen darstellen kann, angefangen bei semi-strukturierten und strukturierten Dokumenten bis hin zu relationalen oder objektorientierten Datenbanken. Entsprechend universell soll XQuery einsetzbar sein. Die Spezifikation von XQuery liegt zur Zeit als Working Draft vor (W3C 2002c).

XQuery ist eine funktionale Sprache. Dementsprechend besteht eine XQuery-Anfrage im Wesentlichen aus ineinander geschachtelten Ausdrücken und Unterausdrücken. Es gibt keine Anweisungen, keine änderbaren Variablen, und die Auswertung eines Ausdrucks hat keine Seiteneffekte. Ausdrücke sind beliebig komponierbar: Überall, wo ein Wert erwartet wird, kann ein Ausdruck stehen.

XQuery ist außerdem streng getypt. Die eben genannte Komponierbarkeit ist also dadurch eingeschränkt, dass der Typ eines Unterdrucks mit dem erwarteten Typ verträglich sein muss. Der Typ eines Ausdrucks lässt sich in XQuery durch eine so genannte Typinferenz aus den Typen seiner Unterausdrücke ableiten.

Der Inhalt dieses Kapitels basiert zu großen Teilen auf den aktuellen Working Drafts zu XQuery und seinem Datenmodell (W3C 2002c, d). Weitere Einführungen in XQuery, teilweise etwas veraltet, was den Entwicklungsstand der Sprache betrifft, sind von Chamberlin (2002) und Chase (2002) verfügbar.

3.1.1 Einfache Beispiele

Um einen ersten intuitiven Eindruck von XQuery zu bekommen, betrachten wir den Ausdruck in Listing 3.1 auf der nächsten Seite. Er enthält als Unterausdrücke viele typische Bestandteile von XQuery: Über die Zeilen 3 bis 8 erstreckt sich ein so genannter FLWOR-Ausdruck (nach den Schlüsselwörtern `for`, `let`, `where`, `order by` und `return`; zwei davon fehlen hier). Er ist vergleichbar mit dem bekannten Select-From-Where aus SQL.

Listing 3.1: Ein XQuery-Ausdruck: Erstelle eine Liste der Bücher, die bei Addison-Wesley nach 1991 erschienen sind, und gib dabei zu jedem Buch Jahr und Titel an. (aus W3C 2002b)

```
1  <bib>
2  {
3    for $b in document("http://www.bn.com/bib.xml")/bib/book
4    where $b/publisher = "Addison−Wesley" and $b/@year > 1991
5    return
6      <book year="{ $b/@year }">
7        { $b/title }
8      </book>
9  }
10 </bib>
```

Listing 3.2: Ein XQuery-Ausdruck ohne FLWOR: Wie viele Bücher sind verfügbar?

```
count(document("http://www.bn.com/bib.xml")/bib/book)
```

Die in den Zeilen 3, 4, 6 und 7 auftretenden Ausdrücke, die Schrägstriche enthalten (ausgenommen die URL in Zeile 3), heißen Pfadausdrücke und selektieren Knoten in der Baumstruktur des Eingabedokuments. Solche Knoten oder auch Folgen von Knoten können an Variable gebunden werden, deren Name mit einem Dollarzeichen beginnt.

Die XML-Tags in den Zeilen 1 und 10 und Zeilen 6 und 8 schließlich konstruieren Teile der Ergebnisstruktur: Ein `bib`-Element, darin eine Folge von `book`-Elementen, deren Inhalte und Attribute jeweils über weitere (beliebige) XQuery-Ausdrücke in geschweiften Klammern bestimmt werden.

Alle diese Arten von Ausdrücken sind optional, auch der FLWOR-Ausdruck ist nur ein Ausdrucksmittel unter vielen bei der Formulierung von Anfragen. Dass Anfragen auch ohne ihn auskommen können, zeigt Listing 3.2; die dort gezeigte Kombination aus einer Aggregatfunktion und einem Pfadausdruck ist ein vollständiger XQuery-Ausdruck. Alle zur Verfügung stehenden Ausdrücke sind

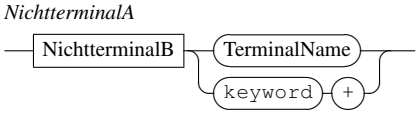
in Abschnitt 3.4 auf Seite 26 dargestellt. Vorher sind für das Verständnis allerdings Blicke auf das Datenmodell, das XQuery zugrunde liegt, sowie allgemeine Grundlagen zur Verarbeitung von XQuery notwendig.

XQuery-Anfragen können Prologe vorgeschaltet werden, in denen bestimmte Einstellungen für die Auswertung eines Anfrage untergebracht werden können. Da jede XQuery-Anfrage abgesehen von diesem optionalen Prolog ein Ausdruck ist, verwenden wir im Folgenden die Begriffe Ausdruck und Anfrage synonym, Anfrage dabei eher für Ausdrücke, die nicht Unterausdruck eines anderen Ausdrucks sind.

3.1.2 Syntaxdiagramme

Beginnend bei Abschnitt 3.3.4 auf Seite 23 sind in die Darstellung von XQuery Syntaxdiagramme eingeflochten, um die Benutzersyntax anschaulich zu machen, ohne dass zu jedem Ausdruckstyp ein Anfragebeispiel gebracht werden muss.

Die Syntaxdiagramme stellen Produktionen der Grammatik dar und sind von links nach rechts zu lesen. Als Titel tragen sie den Namen des Nichtterminalsymbols, das in dieser Produktion erklärt wird. Rechteckige Symbole stellen weitere referenzierte Nichtterminalsymbole dar, abgerundete Rechtecke Terminalsymbole. Lässt sich ein Nichtterminalsymbol durch eine konstante Zeichenkette darstellen (beispielsweise bei Schlüsselwörtern oder Operatoren), ist diese in nichtproportionaler Schrift angegeben, ansonsten steht ein Name innerhalb des Symbols.



Die dargestellte Grammatik basiert auf der offiziellen, wurde aber an vielen Stellen um technische Feinheiten bereinigt, um sie für den menschlichen Leser übersichtlicher zu machen. So wurden explizit aufgeführter Whitespace und Varianten von Produktionen, die nur zur Elimination von Mehrdeutigkeiten existierten, entfernt. Außerdem wurden einige kleinere Produktionen durch Einsetzen aufgelöst. Die Grammatik in diesem Kapitel sollte deshalb nie Grundlage für eine Implementierung sein.

Die englischen Namen der Produktionen wurden beibehalten, um eine parallele Arbeit mit der Originalgrammatik zu erleichtern. Die den deutschen Begriffen

entsprechenden Namen sind bei Verweisen auf Syntaxdiagramme angeführt, sofern auf mehr als eine Produktion verwiesen wird oder die Zuordnung nicht unmittelbar klar ist. Ein Index am Ende dieser Arbeit erleichtert das Navigieren in der Grammatik.

3.2 Datenmodell

3.2.1 Einführung

XQuery 1.0 teilt sein Datenmodell mit XPath 2.0 und XSLT 2.0. Das Datenmodell, das XQuery zu Grunde liegt (W3C 2002d), dient zwei Zwecken. Zum einen definiert es die Informationen in der Eingabe für einen XQuery- oder XSLT-Prozessor, zum anderen gibt das Datenmodell alle Werte an, die ein Ausdruck in XQuery 1.0, XPath 2.0 oder XSLT 2.0 haben kann. Das Datenmodell ist gerade so definiert, dass diese drei Sprachen bezüglich des Datenmodells abgeschlossen sind.

Immer, wenn im Folgenden einfach nur von *dem* Datenmodell gesprochen wird, ist das hier vorgestellte Datenmodell von XQuery gemeint.

Das Datenmodell baut auf dem Modell des XML Information Set (W3C 2001a) auf, ergänzt dieses aber um Typinformationen aus XML-Schema und verallgemeinert es dahingehend, dass es auch Kollektionen von Dokumenten darstellen kann.

Das Datenmodell beschreibt nur, welche Informationen aus Dokumenten verfügbar sind. Es gibt keine Vorgaben, wie diese Informationen repräsentiert oder in einer konkreten Programmiersprache angesprochen werden können.

Ein Wert innerhalb des Datenmodells ist eine Sequenz. Eine Sequenz ist eine geordnete Folge von null oder mehr Items. Ein Item[1] ist entweder ein atomarer Wert oder ein Knoten. Ein atomarer Wert ist ein Wert aus dem Wertebereich eines atomaren Typs aus der XML-Schema-Spezifikation (W3C 2001c, d, e). Ein Knoten gehört zu einer von sieben Arten von Knoten (Abschnitt 3.2.3 auf Seite 17).

[1] Eine gute deutsche Übersetzung für die Bestandteile einer Sequenz fehlt noch. Dummerweise ist der Begriff „Element“ im Zusammenhang mit XML schon vergeben.

3.2.2 Konzepte

Knotenidentität

Wie allen Datenmodellen zu XML liegt auch diesem eine Baumstruktur aus Knoten zu Grunde. Zusätzlich existiert auf der Menge der Knoten ein Konzept der Identität. Jeder Knoten ist einzigartig; zwei Knoten lassen sich auch dann unterscheiden, wenn sie den gleichen Wert besitzen.

Dokumentreihenfolge

Eine so genannte Dokumentreihenfolge (Document Order) ist zunächst auf der Menge aller Knoten eines bestimmten Dokuments definiert. Die Dokumentreihenfolge ist eine totale Ordnung, wobei die Ordnung in bestimmten (kleinen) Gruppen von Elementen implementierungsabhängig sein kann. Für die Dokumentreihenfolge gilt:

- Der Dokument-Knoten ist der erste Knoten.
- Die relative Ordnung zweier Geschwisterknoten (mit der Ausnahme von Attribut- oder Namensraum-Knoten) entspricht ihrer Reihenfolge in einer XML-Repräsentation. Ein solcher Knoten k_1 tritt genau dann vor einem Knoten k_2 auf, wenn sich der Anfang von k_1 im XML-Dokument vor dem Anfang von k_2 befindet.
- Elementknoten stehen vor ihren Kindknoten, und diese vor allen nachfolgenden Geschwisterknoten des Elementknotens.
- Namensraum-Knoten sind direkt nach ihrem zugehörigen Elementknoten eingeordnet. Die relative Ordnung mehrerer Namensraum-Knoten an einem Elementknoten ist implementierungsabhängig, aber stabil. Stabil bedeutet hier, dass sich eine Implementierung zwar bei der relativen Ordnung zweier Namensraum-Knoten Wahlfreiheit hat, nach einer Wahl aber immer die gleiche Ordnung annehmen muss.
- Attributknoten folgen direkt den Namensraum-Knoten des Elementknotens, zu dem sie gehören. Ihre relative Ordnung untereinander ist wieder implementierungsabhängig, aber stabil.

Das Konzept der Dokumentreihenfolge wird aus praktischen Gründen auf die Menge der Knoten in allen Dokumenten ausgedehnt: Die Reihenfolge zwei Knoten, die aus verschiedenen Dokumenten stammen, wird durch die Implementierung bestimmt, muss aber konsistent sein. Das bedeutet, das alle Knoten eines Dokuments entweder vor oder nach allen Knoten eines anderen Dokuments eingeordnet werden müssen.

XML-Schemata und das XML Information Set

Das Datenmodell ist aufbauend auf dem PSVI (Abschnitt 2.1.1 auf Seite 3) definiert. Ein Schema ist nicht unbedingt notwendig; das Datenmodell kann auch mit nur DTD-gültigen oder nur wohlgeformten XML-Dokumenten umgehen. Allerdings muss jedes Dokument der Namensraum-Spezifikation (W3C 1999a) entsprechen. Darüber hinaus kann das Datenmodell Dokumentfragmente und Listen von Fragmenten oder Dokumenten darstellen, ebenso wie atomare Werte, die keine Knoten sind.

Eine Rücktransformation aus dem Datenmodell in das XML Information Set existiert. Auf diesem Wege kann eine Serialisierung einer Instanz des Datenmodells in ein XML-Fragment erfolgen.

Typen im Datenmodell

Das Datenmodell unterstützt benamte Typen (im Gegensatz zu strukturellen Typen, die anhand Struktur identifiziert werden, nicht anhand ihres Namens). Das können die eingebauten Typen von XML-Schema sein (W3C 2001e), aber auch Typen, die darauf aufbauend vom Benutzer in einem Schema definiert werden. Erfasst werden dabei nur Typen, die in einem Schema global definiert sind, sich also eindeutig durch ein Paar aus Namensraum und lokalem Namen identifizieren lassen. Lokal definierte Typen werden durch die Universaltypen `xs:anyType` oder `xs:anySimpleType` repräsentiert.

Element- und Attributknoten sowie atomare Werte sind im Datenmodell mit Typinformationen versehen. Die Typinformation sichert zu, das das betreffende Item eine gültige Instanz dieses Typs ist. Die Typinformation kann auch ganz fehlen, wenn der Typ eines Items unbekannt ist.

Getypter Wert und Stringwert

Der Inhalt eines Text-, Attribut- oder Element-Knotens kann auf zwei verschiedenen Weisen aufgefasst werden: Als ein simpler Stringwert oder als getypter Wert.

Der Stringwert eines Knotens lehnt sich an die Vorstellung der ursprünglichen XML-Spezifikation (W3C 2000b) an, noch bevor strengere Typisierungen im Rahmen von XML-Schema eingeführt wurden. Er ergibt sich aus dem Inhalt eines Knotens; beispielsweise werden bei Elementen hierzu alle Text-Nachfolgerknoten in Dokumentreihenfolge verschmolzen.

Der getypte Wert eines Knotens ist eine Sequenz aus atomaren Werten. Diese werden anhand des Typs eines Knotens aus seinem Stringwert ermittelt, ähnlich wie bei der Schemavalidierung.

3.2.3 Knoten

Das Datenmodell unterscheidet sieben Arten von Knoten und definiert genau, welche Informationen mit ihnen assoziiert sind. Diese Informationen werden als Eigenschaften der Knoten angegeben. Als Werte dieser Eigenschaften können andere Knoten auftauchen. Auf diese Weise wird die Baumstruktur des Datenmodells realisiert.

Abbildung 3.1 auf der nächsten Seite gibt einen Überblick über die verschiedenen Knotentypen, ihre Eigenschaften und Beziehungen in UML-Darstellung. Nicht dargestellt ist die Dokumentreihenfolge: Jede Menge von Knoten, die an einer $1 : n$-Beziehung teilnimmt, ist geordnet.

3.2.4 Atomare Werte

Ein atomarer Wert besteht aus zwei Teilen, einem atomaren Typ und einem Wert aus dessen Wertebereich. Ein atomarer Typ ist ein primitiver Typ oder ein davon durch Einschränkung abgeleiteter Typ. Primitive Typen umfassen verschiedene Zahl-, Zeichenketten-, Namens- und Zeittypen.

3.2.5 Sequenzen

Eine Sequenz ist eine geordnete Liste aus null oder mehr so genannten Items. Ein Item kann ein Knoten sein oder ein atomarer Wert. Wenn ein Knoten zu einer Sequenz hinzugefügt wird, behält er seine Identititität. Ein bestimmter Knoten kann

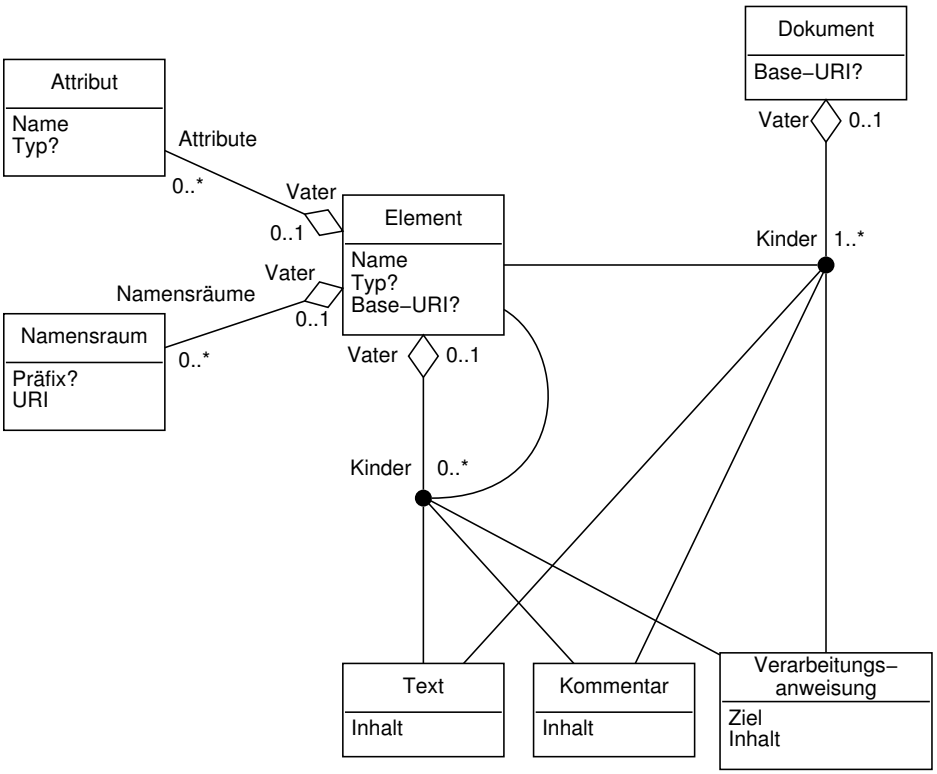


Abbildung 3.1: Versuch der Darstellung der Knotentypen des Datenmodells in UML-Notation. Fragezeichen zeigen an, dass eine Eigenschaft auch leer sein kann.

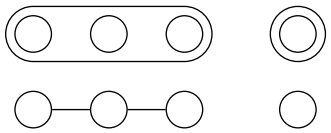


Abbildung 3.2: Zwei mögliche Vorstellungen von Sequenzen: Oben Items in einem Container, unten wie Atome in einem Molekül verbundene Items. Nur letztere verträgt sich mit der Identität von einzelnem Item und Sequenz, die nur dieses Item enthält (rechts unten).

also in mehr als einer Sequenz vorkommen. Sequenzen sind immer flach in dem Sinne, dass sie keine anderen Sequenzen als Items enthalten können. Fasst man zum Beispiel den atomaren Wert 1, die Sequenz $(2,3)$ und die leere Sequenz $()$ in dieser Reihenfolge zu einer neuen Sequenz zusammen, ergibt sich $(1,2,3)$, nicht $(1,(2,3),())$.

Zwischen einem Item und einer Sequenz, die nur dieses Item enthält, wird im Datenmodell nicht unterschieden. Ein einzelnes Item ist also immer auch eine Sequenz; eine Sequenz der Länge eins kann immer als ein einfaches Item aufgefasst werden.

Man mag in den letzten zwei Absätzen ein Paradoxon entdeckt haben: Einerseits kann eine Sequenz *s* keine anderen Sequenzen als Items enthalten, andererseits ist jedes Item *i* in *s* identisch zu einer Sequenz, die nur *i* enthält; also enthält *s* doch Sequenzen. Auch die Tatsache, dass jedes Item sich als Sequenz quasi selbst enthält, mutet auf den ersten Blick seltsam an.

Diese Widersprüche entstehen jedoch nur, wenn man sich eine Sequenz als eigenständigen Container oder als eine Hülle um eine Menge von Items vorstellt. Betrachtet man allerdings Items in einer Sequenz wie Atome, die zu einem Molekül verbunden sind, löst sich das Paradoxon auf: Natürlich ist ein einzelnes Atom auch ein Molekül. (Abbildung 3.2)

Für Sequenzen gibt es keinen Identitätsbegriff. Zwei Sequenzen sind gleich, wenn sie die gleichen Items in der gleichen Reihenfolge enthalten.

3.3 Grundlagen

3.3.1 Ausdruckskontext

Das Ergebnis eines XQuery-Ausdrucks kann außer von den Ergebnissen seiner Teilausdrücke von zusätzlichen Informationen abhängen. Diese Informationen werden im so genannten Ausdruckskontext zusammengefasst. Der Ausdruckskontext zerfällt in zwei Bereiche, den statischen Kontext und den Auswertungskontext.

Statischer Kontext

Der statische Kontext eines Ausdrucks besteht aus denjenigen Informationen, die bei seiner statischen Analyse zur Verfügung stehen. Sie beeinflussen also die Entscheidung, ob ein Ausdruck einen statischen Fehler enthält.

Hauptquelle der Informationen für den statischen Kontext ist der Anfrageprolog, aber auch die Umgebung außerhalb einer XQuery-Anfrage kann Kontextinformationen vorgeben. Im Einzelnen besteht der statische Kontext aus folgenden Teilen:

- Namensräume
- Sichtbare Schemadefinitionen
- Sichtbare Variablen
- Sichtbare Funktionen
- Collations (Informationen zum Vergleich und zur Sortierung von Zeichenketten nach den Regeln einer natürlichen Sprache)
- Base-URI

Auswertungskontext

Der Auswertungskontext enthält alle Informationen, die bei der Auswertung eines Ausdrucks verfügbar sind. Dies sind alle Informationen aus dem statischen Kontext und zusätzlich folgende:

Kontext-Item: Das Item, das gerade verarbeitet wird, heißt Kontext-Item, auch Kontextknoten, falls es ein Knoten ist. Das Kontext-Item liefert der Ausdruck `.` (Punkt).

Kontextposition Die Position des Kontext-Items in der gerade verarbeiteten Sequenz heißt Kontextposition. (Das erste Item in einer Sequenz hat die Position 1. Die Kontextposition ist immer kleiner oder gleich der Kontextgröße.)

Kontextgröße Die Kontextgröße ist die Länge der gerade bearbeiteten Sequenz, also die Anzahl der enthaltenen Items. Der Ausdruck `last()` liefert die Kontextgröße.

Diese ersten drei Komponenten des Auswertungskontextes werden zusammenfassend auch als Fokus eines Ausdrucks bezeichnet und geben an, relativ zu welchen Daten ein Ausdruck ausgewertet wird. (Zum Beispiel sind relative Pfadausdrücke immer am Kontextknoten verankert.)

Die übrigen Komponenten des Auswertungskontextes sind:

Dynamische Variablen Dies ist eine Menge von Tripeln aus Variablenname, Typ und Wert. Die Variablennamen entsprechen denen der sichtbaren Variablen im statischen Kontext. Jeder Variablenname ist mit einem Wert und dessen dynamischen Typ assoziiert. Der so zugeordnete dynamische Typ kann spezieller sein als der statische Typ, der dem Variablennamen zugeordnet ist.

Typen und Werte von Variablen können auch aus der Umgebung einer Anfrage stammen.

Datum, Zeit, Zeitzone Auf den Zeitpunkt, an dem ein Ausdruck ausgewertet wird, kann mit Hilfe der drei Funktionen `fn:current-date` (Datum), `fn:current-time` (Zeit) und `fn:current-dateTime` (Datum und Zeit) Bezug genommen werden.

Eingabesequenz Die Eingabesequenz stellt eine Art implizite Eingabe dar, wird in beliebiger Art und Weise durch die Implementierung befüllt und kann mittels der Funktion `fn:input` angesprochen werden.

3.3.2 Eingabefunktionen

XQuery bietet drei Funktionen an, mit denen man auf Daten außerhalb der Anfrage Bezug nehmen kann:

- In ihrer einfachsten Form nimmt die Funktion `fn:document` die URI eines XML-Dokuments entgegen und gibt den Dokument-Knoten dieses Dokuments nach Umwandlung in das Datenmodell zurück. Man kann mit der Funktion auch mehrere Dokumente auf einmal oder nur Fragmente von diesen auswählen.
- Die Funktion `fn:collection` funktioniert ähnlich, nur dass sie sich mit einer URI auf eine beliebige Sequenz aus Knoten bezieht.
- Ein Aufruf der Funktion `fn:input` ohne Parameter liefert die Eingabesequenz aus dem Auswertungskontext (Abschnitt 3.3.1 auf Seite 20).

3.3.3 Verarbeitung von Ausdrücken

Grundlage für die Verarbeitung von Ausdrücken ist das Datenmodell. Jeder Ausdruck hat bestimmte Instanzen des Datenmodells als Eingabe und als Ausgabe. Für Dokumente, auf denen ein XQuery-Ausdruck arbeitet, bedeutet dies, dass sie zunächst in das Datenmodell überführt werden müssen. Konzeptionell ist im Working Draft (W3C 2002c) dafür folgendes Vorgehen gegeben:

- Parsen mit einem XML-Parser
- Validierung gegen XML-Schemata; dabei entsteht eine Struktur, die Informationen über das Dokument und über die Validität und dynamische Typen seiner Bestandteile enthält, das schon bekannte Post-Schema Validation Infoset (PSVI). Diese Darstellung ist schon sehr verwandt mit derjenigen im Datenmodell.
- Transformation des PSVI in das Datenmodell. (W3C 2002d)

In der Realität können Instanzen des Datenmodells auch auf andere Art und Weise entstehen. Beispielsweise können sie direkt aus relationalen Datenbanken erzeugt werden. Dabei ist es nicht notwendig, die Instanzen des Datenmodells vor der Anfrageverarbeitung zu materialisieren.

3.3.4 Typen in XQuery

XQuery ist eine streng getypte Sprache mit einem Typsystem basierend auf XML-Schema. Ein Typfehler tritt immer dann auf, wenn der Typ eines Werts sich nicht mit dem Kontext verträgt, in dem der Wert auftritt.

Typprüfung

XQuery unterscheidet zwei Phasen bei der Verarbeitung eines Ausdrucks und macht damit die klassische Unterscheidung zwischen Compilezeit und Laufzeit: XQuery nennt die beiden Phasen Analysephase und Auswertungsphase.

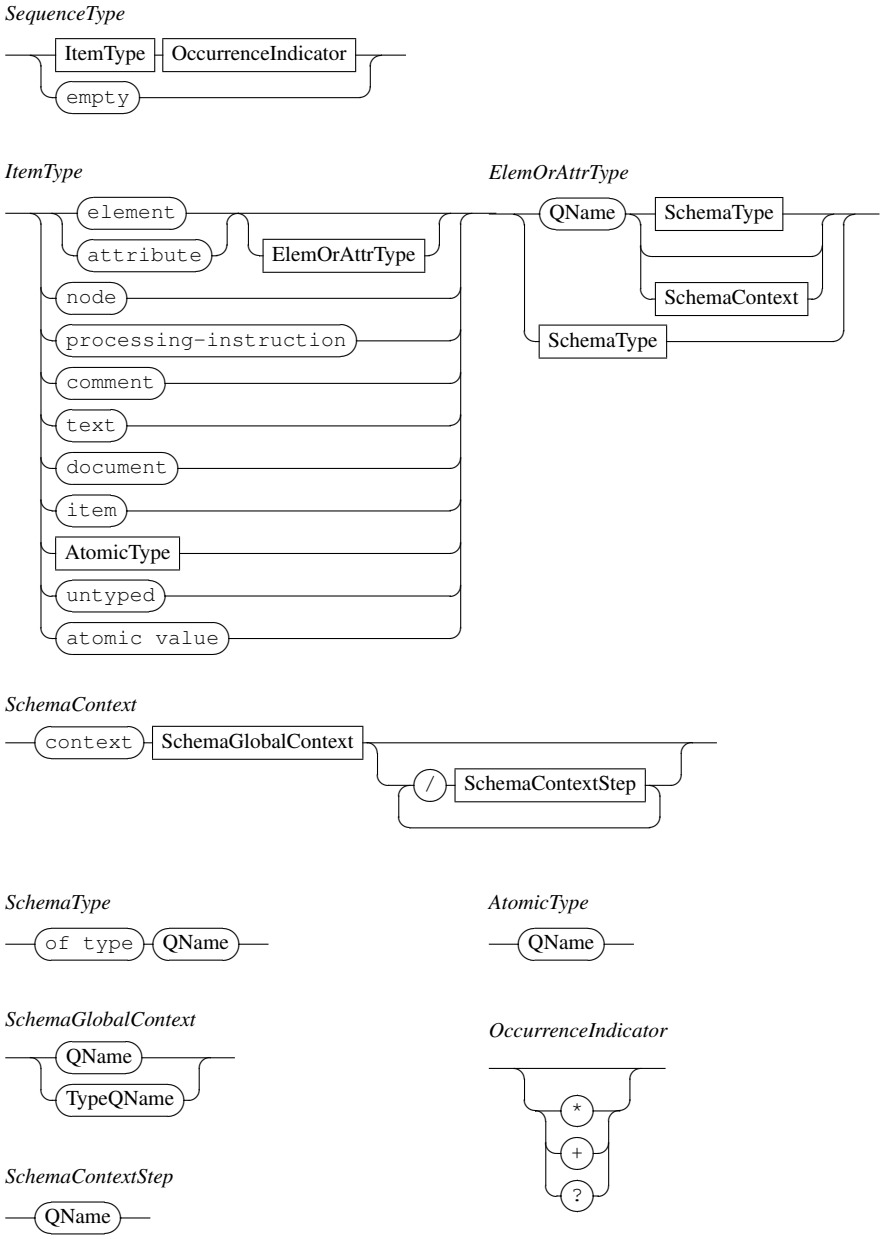
Die Analysephase betrachtet nur den XQuery-Ausdruck selbst und den statischen Kontext. Ziel der Typprüfung in dieser Phase ist es, früh Fehler zu finden und schließlich einen Typ für das Ergebnis der Anfrage abzuleiten. Dies wird erreicht, indem anhand von Regeln von den Typen der untergeordneten Teilausdrücke auf die Typen der übergeordneten Ausdrücken geschlossen wird (W3C 2002e). Dabei wird jeder Teilausdruck mit einem statischen Typ versehen. Der statische Typ ist dabei ein einzelner benamter Typ oder eine strukturelle Beschreibung.

In der Auswertungsphase werden zum ersten Mal die eigentlichen Daten, gegen die die Anfrage gerichtet ist, betrachtet. Zusätzlich hat der Auswertungskontext einen Einfluss auf das Ergebnis. Jeder Wert, der im Rahmen der Auswertung berechnet wird, ist mit einem dynamischen Typ assoziiert. Dieser dynamische Typ kann spezieller sein als der während der Analysephase für diesen Ausdruck ermittelte statische Typ.

Sequenztypen

Um sich in einem Ausdruck auf einen XQuery-Typ zu beziehen, steht eine einfache Syntax zur Verfügung (Syntax 1 auf der nächsten Seite; SequenceType). Zu einem solchen Sequenztyp gehört eine grobe Angabe über die Länge der Sequenz (`0`, `1`, `0..1`, `1..*` oder `0..*`) sowie eine Art speziellste Verallgemeinerung der Typen aller in der Sequenz auftretenden Items. Der Inhalt einer Sequenz wird also auf Benutzerebene immer nur durch einen einzigen Typ zusammengefasst.

Der Prozess des so genannten Sequenztyp-Matching dient dazu, für eine konkrete Sequenz und einen Sequenztyp, gegeben in der obigen Syntax, festzustellen, ob die Sequenz zu dem Typ gehört. Dabei muss geprüft werden, ob die Längen-



Syntax 1: Sequenztypen

beschränkung eingehalten wird und ob alle Items der Sequenz zum angegebenen Item-Typ (ItemType) gehören.

Die meisten der Item-Typen erklären sich von selbst. Besonderes Augenmerk ist allein auf Elemente und Attribute zu legen. Man kann hier den Typ auf verschiedene Weisen einschränken, indem man folgende Attribute angibt:

- Typ aus einem Schema
- Name des Elements oder Attributs (auch definiert in einem Schema): Ein schemaglobales Element oder Attribut wird dabei eindeutig durch einen QName identifiziert. Lokale Elemente oder Attribute können durch Angabe des Kontextes, in dem sie auftreten, eindeutig gemacht werden. Dieser Kontext ist einfach ein Pfad, ausgehend von einem globalen Element oder Attribut.

Typumwandlungen

Bei einigen Ausdrücke ist es zugelassen, dass die Operanden nicht genau dem erwarteten Typ entsprechen. In diesen Fällen können automatische Typumwandlungen stattfinden wie zum Beispiel die Extraktion atomarer Werte oder die Umwandlung zwischen verschiedenen numerischen Typen. Im Folgenden werden zwei häufig vorkommende Umwandlungen beschrieben.

Der Prozess der so genannten *Atomisierung* kommt zum Einsatz, wenn ein Ausdruck eine Sequenz aus atomaren Werten als Operand verlangt. Dabei werden Knoten, die als getypten Wert eine Sequenz aus atomaren Werten haben, durch diese ersetzt. Treten andere Knoten auf, scheitert die Atomisierung.

In Kontexten, in denen ein boolescher Wert erwartet wird, haben auch nichtboolesche Werte einen so genannten *effektiven booleschen Wert*. Folgende Werte entsprechen dem booleschen Wert *falsch*: Die leere Sequenz, der leere String, ein numerischer Wert gleich Null und der Fließkommawert NaN. Der effektive boolesche Wert aller anderen Sequenzen ist *wahr*.

3.3.5 Bindung von Variablen

Bestimmte Ausdrücke führen Variablen ein und binden sie an Werte.

Variablen sind in allen Unterausdrücken des Ausdrucks sichtbar, in dem sie gebunden werden. Davon ausgenommen sind nur der Ausdruck, der den Wert

Syntax 2: Ausdrücke und primäre Ausdrücke

bestimmt, an den die Variable gebunden wird, und alle Ausdrücke, die in der Anfrage vor der Variablenbindung stehen.

Ist in einem Ausdruck eine Variable aus einem umgebenden Ausdruck sichtbar, so kann diese durch eine neue Variable gleichen Namens überdeckt werden.

3.4 Ausdrücke

Wie bereits betont wurde, sind Ausdrücke in ihren vielfältigen Ausprägungen die Grundbausteine, aus dem XQuery-Anfragen zusammengesetzt werden können. Dieser Abschnitt gibt einen vollständigen Überblick über diese Ausprägungen. Einige der für XQuery charakteristischen Ausdruckstypen werden dabei detaillierter betrachtet, während viele andere schon zum Standardrepertoire von Anfrage- oder Programmiersprachen gehören.

Die Darstellung beginnt bei einfachen Ausdrücken und arbeitet sich zu komplexeren vor. (Syntax 2; Expr)

3.4.1 Primäre Ausdrücke

Die einfachsten Ausdrücke, die XQuery kennt, haben keine Unterausdrücke: Literale und Variablen. Zu den primären Ausdrücken (Syntax 2; PrimaryExpr) zählen weiterhin Funktionsaufrufe und geklammerte Ausdrücke.

Literale

Es existieren Schreibweisen für literale Zahlen und Zeichenketten, nämlich für die Typen `xs:string`, `xs:integer`, `xs:decimal` und `xs:double` (Syn-

Syntax 3: Literale

tax 3, Literal). Werte anderer Typen können mit Hilfe von Konstruktoren erzeugt werden: `xs:date("2001-08-25")`.

Variablen

Eine Variable steht immer für den Wert, der im Auswertungskontext an ihren Namen gebunden wurde.

Verschiedene Ausdrücke in XQuery können Variablen binden. Im einzelnen sind dies FLWOR-, quantifizierte und Typeswitch-Ausdrücke. Außerdem werden bei Funktionsaufrufen Werte an die formalen Parameter einer Funktion gebunden. Schließlich können Variablenbindungen auch aus der Umgebung einer Anfrage stammen.[2]

Variablen in XQuery sind keine Variablen im prozeduralen Sinn: Nach ihrer Bindung an einen Wert sind weitere Zuweisungen nicht möglich, allerdings auch nicht nötig. Variablen dienen nur dazu, bestimmte Werte für deren weitere Verwendung in Unterausdrücken zugänglich zu halten.

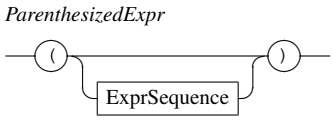
Variablen werden syntaktisch immer durch ein vorangestelltes Dollarzeichen `$` markiert. Dies ist vor allem wichtig, um Variablennamen von einfachen Schritten in Pfadausdrücken zu unterscheiden.

Geklammerte Ausdrücke

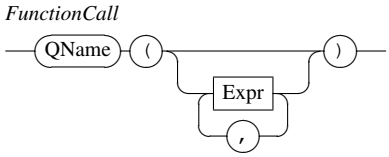
Mit der Klammerung von Ausdrücken kann man wie in anderen Sprachen die Auswertungsreihenfolge beeinflussen, die sich ohne Klammern aus der Präzedenz der Operatoren ergibt (Syntax 4 auf der nächsten Seite; ParenthesizedExpr).

Mit einem leeren Klammernpaar wird die leere Sequenz dargestellt.

[2] Dieser Umstand spielt später eine tragende Rolle bei der Einbettung von XQuery in Java.



Syntax 4: Geklammerte Ausdrücke



Syntax 5: Funktionsaufrufe

Funktionsaufrufe

Funktionsaufrufe sehen aus wie in anderen Sprachen (Syntax 5; FunctionCall). Es ist bei XQuery lediglich zu beachten, dass sich Kommata auf oberster Ebene in der Argumentliste unterscheiden von solchen, die Sequenzen konstruieren. Im Beispiel

```
f ((1,(2,3)),4)
```

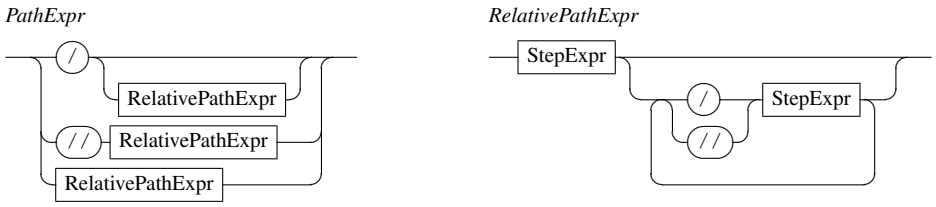
hat die Funktion `f` also zwei Argumente, nämlich die Sequenzen $(1,2,3)$ und (4), nicht etwa nur die Sequenz $(1,2,3,4)$.

Kommentare

Kommentare in XQuery sind keine Ausdrücke, sondern werden schon bei der lexikalischen Analyse entfernt.

{−− Kommentar −−}

Nicht zu verwechseln ist diese Art von Kommentaren mit Kommentaren in XML-Notation, die entsprechende Kommentar-Knoten im Datenmodell konstruieren und so auch Bestandteil des Ergebnisses einer Anfrage sein können. (Abschnitt 3.4.7 auf Seite 43)



Syntax 6: Pfadausdrücke

3.4.2 Pfadausdrücke

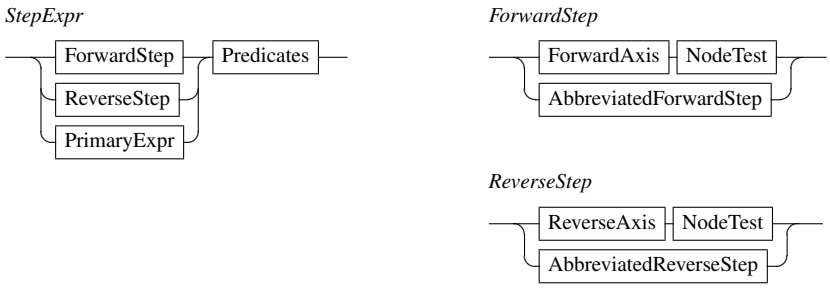
Pfadausdrücke sind neben FLWOR-Ausdrücken (Abschnitt 3.4.8 auf Seite 43) ein charakteristischer Bestandteil von XQuery. Sie dienen dazu, innerhalb der Baumstruktur von XML-Fragmenten bestimmte Knoten zu selektieren. In ihrer einfachsten Form erinnern sie syntaktisch an Pfade in einem Dateisystem, und auch ihre Semantik stimmt weitgehend mit diesen überein. Sie werden jedoch mächtiger durch zusätzlich einschränkende Prädikate, und die Tatsache, dass Elemente viele Kindelemente des gleichen Namens haben können, macht sie auch etwas schwieriger zu verstehen.

Jeder Pfadausdruck besteht aus einer Reihe von Schritten, die ausgehend von einer Verankerung entlang verschiedener so genannter Achsen navigieren, um die interessierenden Knoten zu finden (Syntax 6; PathExpr).

Für gewöhnlich wird ein Pfadausdruck am aktuellen Kontextknoten verankert und operiert relativ zu dieser Position. Auch absolute Pfadausdrücke sind möglich; sie sind durch einen führenden Schrägstrich am Dokument-Knoten des Dokuments verankert, in dem sich der aktuelle Kontextknoten befindet.

Schritte

Ein Schritt (Syntax 7 auf der nächsten Seite; StepExpr) ermittelt ausgehend von einer Ursprungssequenz (explizite Verankerung oder Ergebnis des vorhergehenden Schrittes) anhand einer Achse, einem Knotentest und weiteren Prädikaten eine neue Sequenz, indem er konzeptionell wie folgt vorgeht: Jedes Item aus der Ursprungssequenz wird nacheinander Kontext-Item. Von dort ausgehend werden alle Knoten entlang der angegebenen Achse betrachtet, die den Knotentest erfüllen. Diese Kandidaten-Sequenz wird nacheinander durch alle vorhandenen Prädi-



Syntax 7: Schritte

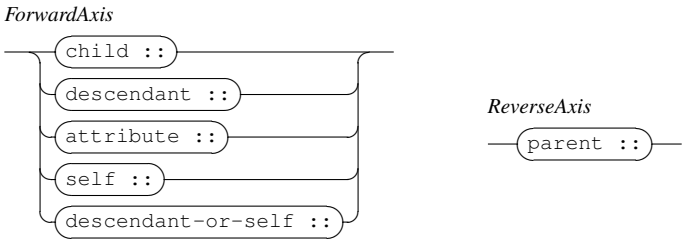
Tabelle 3.1: Achsen für Pfadausdrücke

Achse	enthaltene Knoten
child	Kinder des Kontextknotens
descendant	Nachfahren des Kontextknotens (aber niemals Attribut- oder Namensraum-Knoten)
parent	Vater des Kontextknotens
attribute	Attribute des Kontextknotens, falls er ein Element ist, sonst leer
self	nur der Kontextknoten selbst
descendant-or-self	Kontextknoten und seine Nachfahren

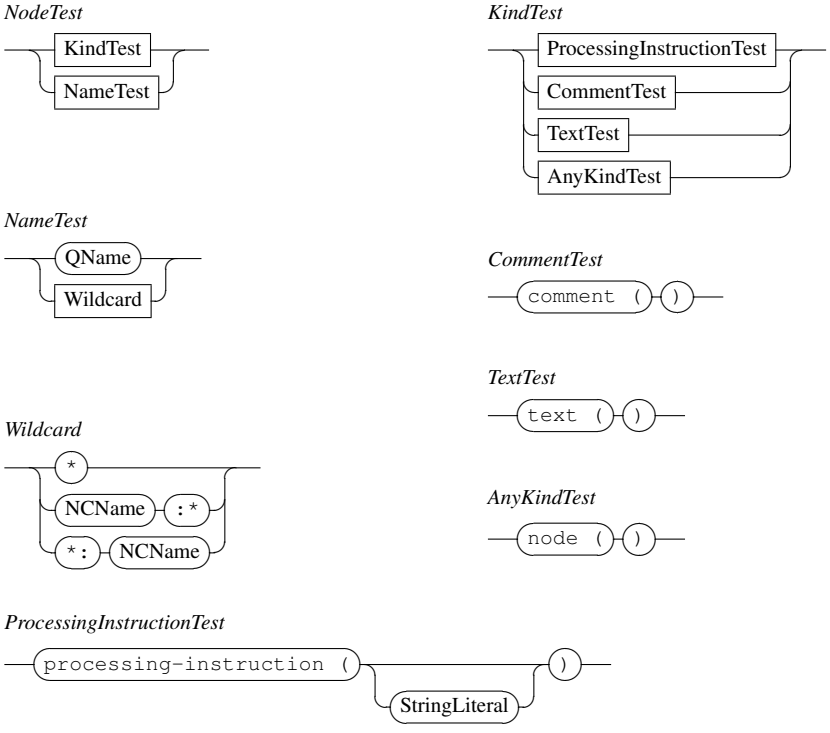
kate gefiltert. Alle auf diese Weise ermittelten Sequenzen werden dann vereinigt und in Dokumentreihenfolge sortiert.

XQuery unterstützt nur eine Auswahl der in XPath spezifizierten Achsen (Syntax 8 auf der nächsten Seite; ForwardAxis, ReverseAxis). Sie sind in Tabelle 3.1 zusammen mit den in ihnen enthaltenen Knoten aufgeführt.

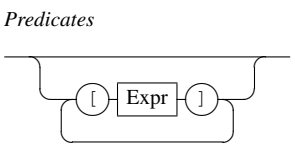
Ein Knotentest selektiert Knoten einer gegebenen Achse anhand ihres Typs oder ihres Namens (Syntax 9 auf der nächsten Seite; NodeTest). Die zweite Form wird deutlich häufiger gebraucht. Auf der Attribut-Achse werden alle Attributknoten auf Übereinstimmung des Namens getestet, auf allen anderen Achsen nur Elementknoten. Der Stern kann als Wildcard verwendet werden, allerdings nur für den ganzen Namen oder einen seiner beiden Bestandteile (Namensraum oder lokaler Name).



Syntax 8: Achsen



Syntax 9: Knotentests



Syntax 10: Prädikate

Prädikate

Ein Prädikat filtert eine beliebige Sequenz, indem nur die Items erhalten werden, für die das Prädikat zu *wahr* ausgewertet wird (Syntax 10; Predicates).

Ein Ausdruck, der eine Zahl ergibt, hat als Prädikat eine besondere Bedeutung: Das Prädikat ist genau dann *wahr*, wenn die Kontextposition mit der Zahl übereinstimmt. Dies ermöglicht die einfache Notation

```
child::person [4]
```

für das vierte `person`-Kindelement des Kontextknotens.

Bei allen anderen Ausdrücken ist zu beachten, dass auch aus Sequenzen oder Strings implizit ein effektiver boolescher Wert ermittelt werden kann. Damit sind kompakte Existenzbedingungen formulierbar; so liefert etwa

```
child::employee [ secretary ]
```

alle `employee`-Kinder des Kontextknotens, die selbst ein `secretary`-Kind haben.

Ausführliche Syntax

Für die Darstellung von Pfadausdrücken existieren zwei Schreibweisen, eine ausführliche, bei der explizit jede benutzte Achse mit ihrem vollständigen Namen referenziert wird, und eine abkürzende, bei der häufig benötigte Bestandteile auf wenige Zeichen zusammengezogen werden. Diese abkürzende Schreibweise wird als nächstes vorgestellt; es folgen zunächst einige einfache Beispiele für Pfadausdrücke in ausführlicher Syntax.

Der Pfadausdruck	selektiert
`child::para`	die `para`-Kindelemente des Kontextknotens
`child::*`	alle Kindelemente des Kontextknotens
`child::text()`	alle Text-Kindknoten des Kontextknotens
`attribute::*`	alle Attribute des Kontextknotens
`descendant-or-self::para`	die `para`-Element-Nachfolger des Kontextknotens und auch den Kontextknoten selbst, falls er ein `para`-Element ist
`child::*/child::para`	alle „Enkelkinder“ des Kontextknotens, die `para`-Elemente sind
`/`	die Wurzel der Knotenhierarchie, in der sich der Kontextknoten befindet
`/descendant::para`	alle `para`-Elemente im selben Dokument wie der Kontextknoten
`child::para[fn:position() = 1]`	das erste `para`-Kind des Kontextknotens

Abkürzende Syntax

Folgende abkürzende Schreibweisen (Syntax 11 auf der nächsten Seite; AbbreviatedForwardStep, AbbreviatedReverseStep) sind in Pfadausdrücken erlaubt:

1. Die Kind-Achse ist die Standard-Achse. Ein `child::` kann also weggelassen werden.
2. `attribute::` zur Benutzung der Attribut-Achse kann zu `@` verkürzt werden.[3]
3. `//` steht für `/descandant-or-self::node()/`. Hiermit wird also eine beliebig lange Zwischenfolge an Schritten ausgedrückt.
4. Ein Schritt `.` steht für das Kontext-Item.
5. Der Schritt `..` kürzt `parent::node()` ab. Die letzten beiden Schreibweisen sind aus Dateisystemen bekannt.

Die ersten der folgenden Beispiele sind die Abkürzungen der entsprechenden Beispiele in ausführlicher Syntax.

[3] `@`, gesprochen „at“, erinnert an „attribute“.

Syntax 11: Abgekürzte Schritte

Der Pfadausdruck	selektiert
`para`	die `para`-Kindelemente des Kontextknotens
`*`	alle Kindelemente des Kontextknotens
`text()`	alle Text-Kindknoten des Kontextknotens
`@*`	alle Attribute des Kontextknotens
`.//para`	die `para`-Element-Nachfolger des Kontextknotens und auch den Kontextknoten selbst, falls er ein `para`-Element ist
`*/para`	alle „Enkelkinder“ des Kontextknotens, die `para`-Elemente sind
`para[1]`	das erste `para`-Kind des Kontextknotens
`../@lang`	das `lang`-Attribut des Vaters des Kontextknotens

3.4.3 Sequenzausdrücke

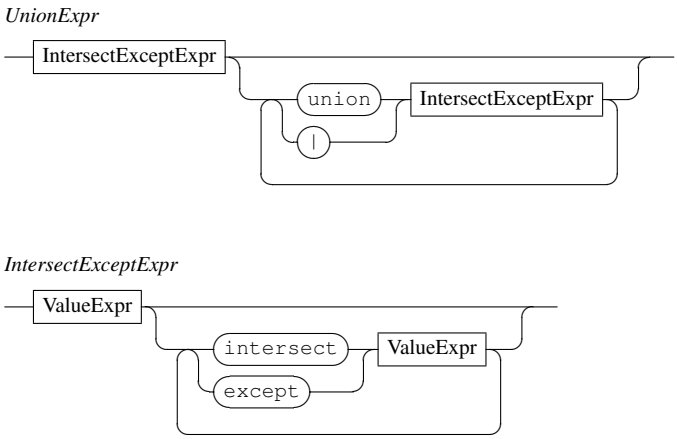
Sequenzen sind der zentrale Datentyp in XQuery. Natürlich enthält XQuery mehrere Operatoren, um Sequenzen zu konstruieren und zu verknüpfen.

Konstruktion von Sequenzen

Eine Möglichkeit zur Konstruktion von Sequenzen bietet der Komma-Operator (Syntax 12 auf der nächsten Seite; ExprSequence). Er verbindet einfach seine beiden Operanden zu einer Sequenz. Normalerweise muss man die durch Kommata getrennte Folge von Ausdrücken zusätzlich in runde Klammern einschließen, um sie wieder als Ausdruck verwenden zu können. Nur auf oberster Ebene in einer Anfrage oder innerhalb eines Element-, Attribut-, oder Dokument-Konstruktors können solche Sequenzen ohne (runde) Klammern stehen.

Der `to`-Operator (RangeExpr) erzeugt eine Sequenz von ganzen Zahlen zwischen zwei ganzzahligen Grenzen. Dabei kann die vordere Grenze auch größer sein als die hintere; es wird dann heruntergezählt.

Syntax 12: Komma- und `to`-Operator



Syntax 13: Mengenoperationen: Vereinigung, Schnitt und Differenz

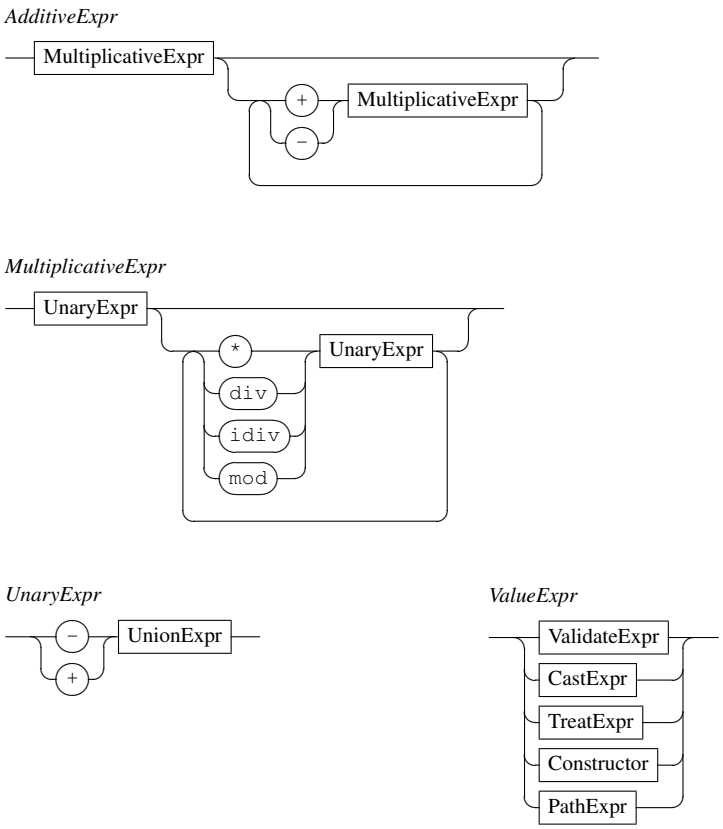
Verknüpfung von Sequenzen

Die Mengenoperationen zur Bildung von Vereinigung, Schnitt und Differenz stützen sich auf die Knotenidentität und liefern ihr Ergebnis immer in Dokumentreihenfolge (Syntax 13).

3.4.4 Arithmetische Ausdrücke

Zur Formulierung arithmetischer Ausdrücke stehen die üblichen Operatoren bereit (Syntax 14 auf der nächsten Seite). Eine Ungewohntheit ergibt sich dadurch, dass der Schrägstrich den Pfadausdrücken (Abschnitt 3.4.2 auf Seite 29) vorbehalten ist: Der Divisionsoperator heißt `div`, der Operator der ganzzahligen Division, sonst unter diesem Namen bekannt, deswegen `idiv`.

Vorsicht ist auch beim Minuszeichen geboten: Es kann in XML-Bezeichnern

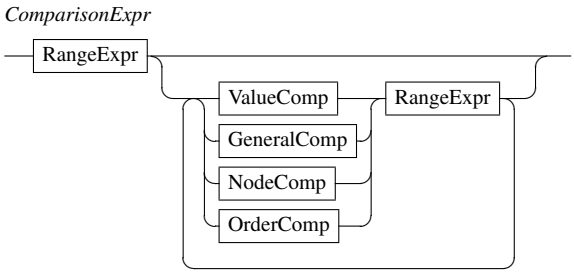


Syntax 14: Arithmetische Ausdrücke

vorkommen, also auch in Variablen- und Funktionsnamen. Soll ein Minuszeichen zwischen zwei Bezeichnern stehen, muss es durch Whitespace abgetrennt werden.

3.4.5 Vergleichsausdrücke

Zum Vergleich von zwei Werten kennt XQuery vier verschiedene Arten von Vergleichsausdrücken (Syntax 15 auf der nächsten Seite; ComparisonExpr). Man sollte besonders auf den Bedeutungsunterschied zwischen Wertvergleichen und allgemeinen Vergleichen achten. Knoten- sowie Reihenfolgevergleiche erklären



Syntax 15: Vergleichsausdrücke

Syntax 16: Wert- und allgemeine Vergleiche

sich dagegen fast von selbst.

Wertvergleiche

Die aus anderen Sprachen bekannten Operatoren zum Vergleich einfacher Werte wie „gleich“, „ungleich“, „kleiner“ und andere gibt es auch in XQuery, nur dass sie hier nicht die gewohnten symbolischen Namen =, `!=` und < tragen, sondern Buchstabenkürzel wie `eq`, `ne` und `lt` (ValueComp). Zur Verwirrung kann zunächst beitragen, dass die Operatoren mit den symbolischen Namen ebenfalls existieren und in machen Fällen sogar das gleiche Verhalten zeigen wie ihre Buchstaben-Pendants (Syntax 16; GeneralComp).

Syntax 17: Knoten- und Reihenfolgevergleiche

Allgemeine Vergleiche

Zu jedem Operator, der einfache Werte vergleicht, gibt es einen so genannten allgemeinen Operator, der implizit eine Existenzquantifizierung vornimmt (Syntax 16 auf der vorherigen Seite; GeneralComp). Diese Operatoren erwarten nämlich zwei Sequenzen von Werten zum Vergleich und liefern *wahr*, wenn ein Paar von Werten aus den beiden Sequenzen existiert, für das der entsprechende Wertvergleich *wahr* ergibt.

So ist es möglich, dass der auf den ersten Blick widersprüchlich wirkende Ausdruck

$a < *$b* **and** *$b* < *$a*

wahr ist, beispielsweise bei folgender Belegung der Variablen:

let *$a* := (1,2,3)
let *$b* := (2,3,4)
return (*$a* < *$b* **and** *$b* < *$a*)

Knoten- und Reihenfolgevergleiche

Die Operatoren `is` und `isnot` vergleichen zwei Knoten anhand ihrer Identität, die Operatoren << und >> anhand der Dokumentreihenfolge (Syntax 17; NodeComp, OrderComp).

3.4.6 Logische Ausdrücke

Im Gegensatz zu den aussagenlogischen Operatoren `and` und `or` (Syntax 18 auf der nächsten Seite) ist `not` als XQuery-Funktion realisiert. Es existiert zu XQuery eine ganze Standardbibliothek von Funktionen, auf die in dieser Einführung nicht weiter eingegangen wird (W3C 2002f). In dieser Bibliothek sind z. B. auch Aggregatfunktionen oder Funktionen zur Manipulation von Texten untergebracht.

Syntax 18: Logische Ausdrücke: Oder und Und

3.4.7 Konstruktoren

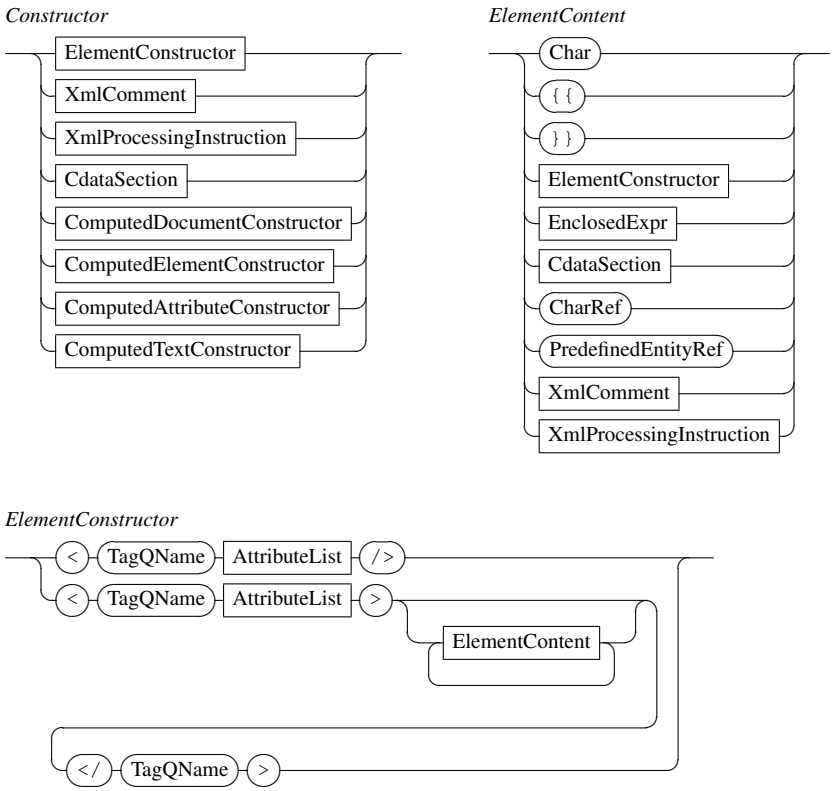
Konstruktoren haben den Zweck, neue XML-Strukturen im Rahmen einer Anfrage zu erzeugen, um so Strukturtransformationen durchzuführen. Einfache Konstruktoren sehen so aus, wie die XML-Strukturen, die sie erzeugen, in einem gewöhnlichen XML-Dokument aussehen würden (Syntax 19 auf der nächsten Seite; Constructor). Speziellere Formen erlauben die Berechnung von Element- oder Attribut-Namen zur Laufzeit.

Elementkonstruktoren

Ein Elementkonstruktor erzeugt ein Element des angegebenen Namens. Der Inhalt eines Elementkonstruktors wird im Normalfall einfach als literaler Text interpretiert. Ausnahmen sind weitere eingebettete Elementkonstruktoren und Ausdrücke, die in geschweifte Klammern eingeschlossen sind. Die Ergebnisse dieser Ausdrücke werden direkt in den Inhalt des Elements übernommen. Zusammengenommen sehen solche Konstruktor-Ausdrücke aus wie Muster für XML-Fragmente mit variablen Teilen (Listing 3.3 auf Seite 42).

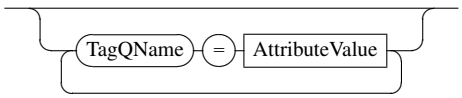
Berechnete Konstruktoren

Berechnete Konstruktoren ermöglichen die Erzeugung von Text- und Dokument-Knoten und vor allem von Element- und Attributknoten, deren Namen nicht konstant sind. Zu diesem Zweck kann ein eigener Ausdruck zur Berechnung des Namens angegeben werden. Gemeinsam ist allen berechneten Konstruktoren eine in geschweiften Klammern eingeschlossene Liste von Ausdrücken, die den Inhalt des zu konstruierenden Elements angibt (Syntax 21 auf Seite 42).

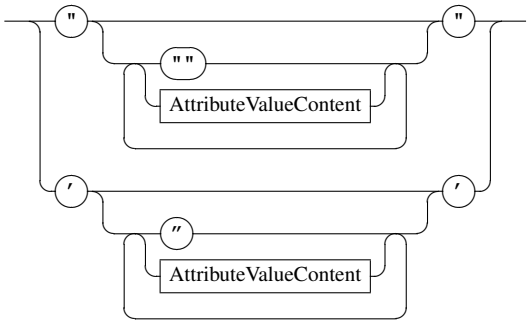


Syntax 19: Konstruktoren

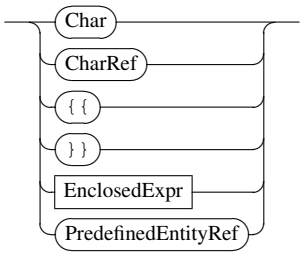
AttributeList



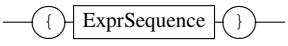
AttributeValue



AttributeValueContent



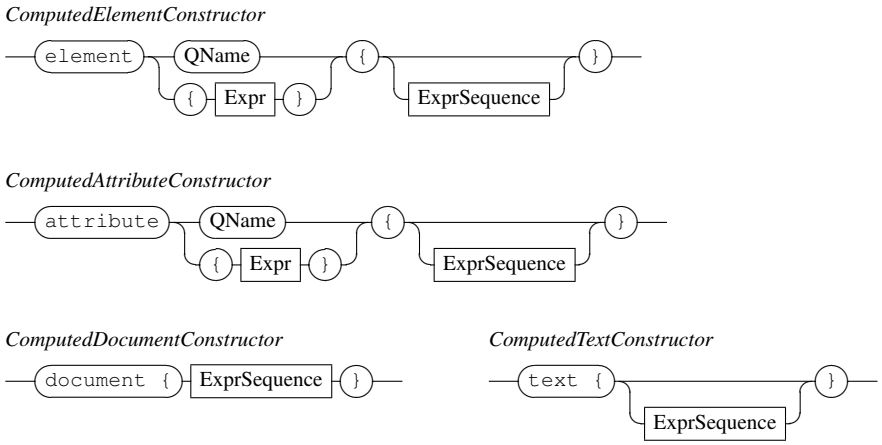
EnclosedExpr



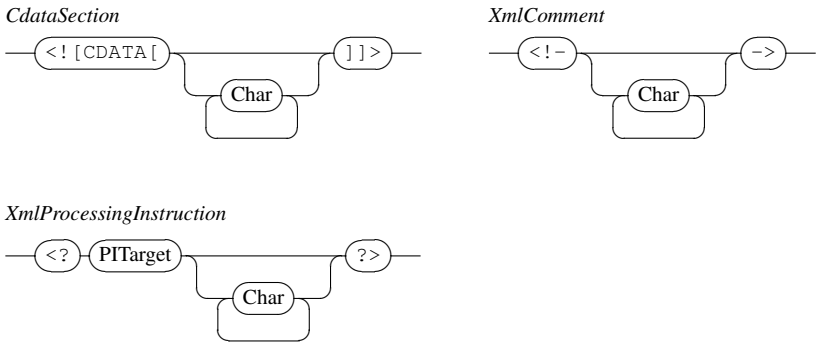
Syntax 20: Konstruktoren (Forts.)

Listing 3.3: Elementkonstruktoren als Muster für XML-Fragmente

```
<summary>
  <count>{ count(//book) }</count>
  <titles>
  {
    for $b in //book
    return <title published="{ $b/year }">{ $b/shorttitle }</title>
  }
  </titles>
</summary>
```



Syntax 21: Konstruktoren mit berechneten Namen, außerdem Konstruktoren für Dokument- und Text-Knoten



Syntax 22: CDATA-Abschnitte, Verarbeitungsanweisungen und Kommentare

Andere Konstruktoren

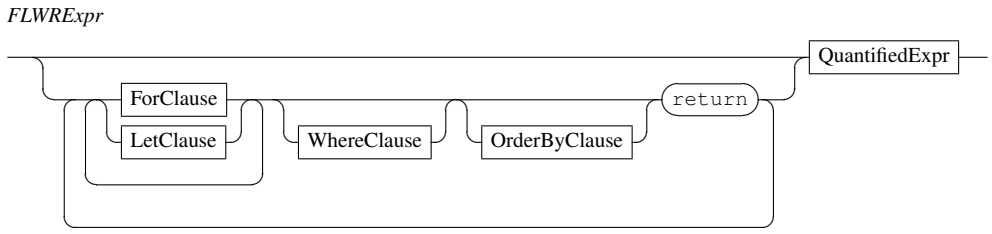
CDATA-Abschnitte, Verarbeitungsanweisungen und Kommentare können mit der aus XML-Dokumenten bekannten Notation verwendet werden (Syntax 22). Kommentare sind hier Bestandteile des Anfrageergebnisses und nicht zu verwechseln mit XQuery-Kommentaren.

3.4.8 FLWOR-Ausdrücke

Vergleichbar mit dem zentralen Konstrukt „select, from, where“ in SQL sind in XQuery die so genannten FLWOR-Ausdrücke (Syntax 23; FLWRExpr[4]). Sie sind vor allem bei der Formulierung von Joins und bei Strukturveränderungen hilfreich. Der Begriff FLWOR wird gerne wie das englische Wort „flower“ ausgesprochen und ist eine Abkürzung der beteiligten Schlüsselwörter `for`, `let`, `where`, `order by` und `return`.

Die `for`- und `let`-Klauseln in einem FLWOR-Ausdruck erzeugen einen so genannten Tupelstrom, d. h. eine Liste aus Tupeln gebundener Variablen. Die `where`-Klausel filtert diesen Strom und die `order-by`-Klausel kann ihn in eine andere Reihenfolge bringen. Schließlich wird die `return`-Klausel für jedes Tupel ausgewertet. Die entstehenden Sequenzen werden in der Reihenfolge des Tupelstroms zu einer Sequenz, dem Ergebnis des FLWOR-Ausdrucks, zusammengesetzt.

[4]Der Name der Produktion stammt aus einer Zeit, als es in FLWOR noch kein `order by` gab.



Syntax 23: FLWOR-Ausdrücke

Listing 3.4: Struktur eines einfachen FLWOR-Ausdrucks

```
for $x_1 in $X_1
 let $x_2 := $X_2
for $x_3 in $X_3
 ...
where p($x_1, $x_2, $x_3 , ...)
return f($x_1, $x_2, $x_3 , ...)
```

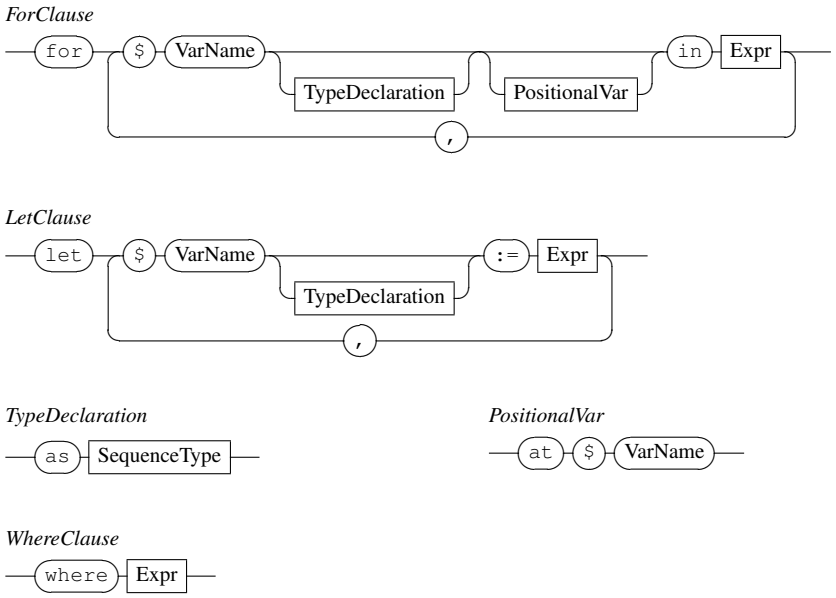
Das Ergebnis eines FLWOR-Ausdrucks wie in Listing 3.4 kann man vergleichen mit folgender mathematischen Darstellung der Erzeugung einer Ergebnismenge R aus anderen Mengen $X_1, X_2, X_3, \ldots$ unter Zuhilfenahme eines Prädikats p und einer Funktion f:

$$R = \{ f(t) \mid p(t), t \in X_1 \times \{X_2\} \times X_3 \times \ldots \}$$

Der Operator $\times$ steht dabei für das Kreuzprodukt und $\{X_2\}$ deutet an, dass nicht die Elemente der Menge X_2 betrachtet werden, sondern die Menge als Ganzes. Allerdings arbeitet ein FLWOR-Ausdruck nicht auf Mengen, sondern auf geordneten Sequenzen, so dass zusätzlich noch die Erzeugung einer Ordnung auf R aus den Ordnungen auf $X_1, X_2, X_3, \ldots$ eine Rolle spielt.

For- und Let-Klausel

Eine `for`-Klausel iteriert durch eine Sequenz und bindet jedes Item nacheinander an die gleiche Variable (Syntax 24 auf der nächsten Seite; ForClause). Eine `let`-Klausel (LetClause) demgegenüber bindet eine Variable an eine gesamte Sequenz. Beide Typen tragen gemeinsam zum Entstehen des Tupelstroms bei. Die

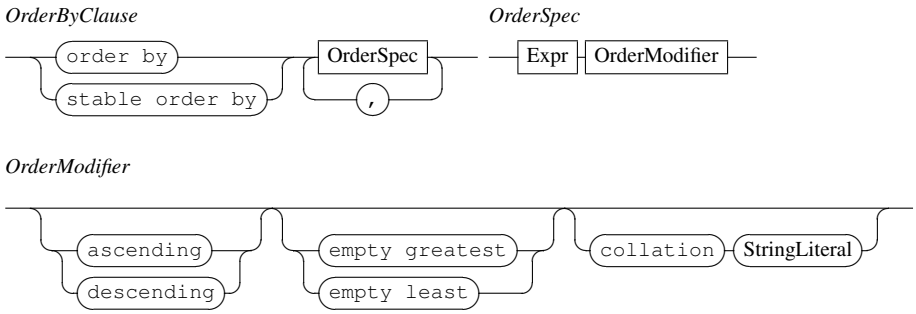


Syntax 24: For-, Let- und Where-Klausel

Reihenfolge von (möglicherweise mehreren) `for`- und `let`-Klauseln ist dabei signifikant: Ausdrücke in nachfolgenden Klauseln können sich auf Variablen aus vorhergehenden beziehen und so von diesen abhängen.

Die erzeugten Tupel bestehen aus Bindungen für alle in den Klauseln auftretenden Variablen; ihre Reihenfolge im Tupelstrom lässt sich am einfachsten erklären, indem man die `for`- und `let`-Klauseln in einem FLWOR-Ausdruck vorübergehend wie ihre gleichnamigen Pendants in prozeduralen Programmiersprachen versteht, also `let` als Zuweisung und jede `for`-Klausel als for-Schleife, die Items einer Sequenz von vorne nach hinten durchläuft und dabei den Block der nachfolgenden Klauseln ausführt. Die Reihenfolge der Variablenbelegungen, die bei der Ausführung dieser geschachtelten Schleifen entstehen würde, ist genau die des erzeugten Tupelstroms.

Durch zusätzliche Positionsvariablen (PositionalVar) kann in folgenden Klauseln auf den Stand der „Iteration“ in `for`-Klauseln zugegriffen werden.



Syntax 25: Order-By-Klausel

Where-Klausel

Ein Tupel des Tupelstroms kann die `where`-Klausel genau dann passieren, wenn der effektive boolesche Wert der Bedingung *wahr* ist (Syntax 24 auf der vorherigen Seite; WhereClause).

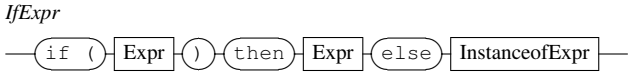
Order-By- und Return-Klausel

Die `order-by`-Klausel ermöglicht eine Umsortierung des Tupelstroms. Man kann beliebige Ausdrücke auf den Tupeln formulieren, nach deren Werten sortiert wird (Syntax 25; OrderByClause).

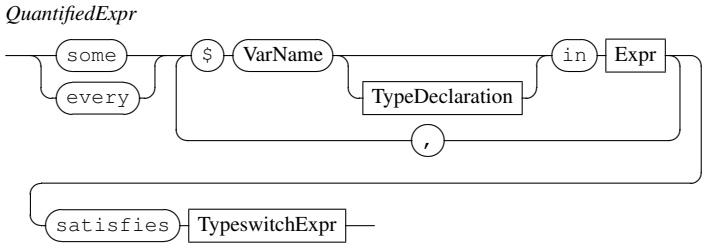
Für jedes Tupel im Tupelstrom wird schließlich die `return`-Klausel ausgewertet, die aus den Tupeln mit Hilfe beliebiger Ausdrücke einen Teil des Ergebnisses erzeugt. Die Ergebnisse der `return`-Klausel für alle Tupel werden in der Reihenfolge des Tupelstroms zu einer Sequenz verschmolzen.

3.4.9 Ungeordnete Ausdrücke

Eine Sequenz ist geordnet und jedes Ergebnis eines XQuery-Ausdrucks ist eine Sequenz. Wie gerade bei den FLWOR-Ausdrücken oder schon bei den Pfadausdrücken deutlich geworden ist, ist die Reihenfolge einer solchen Ergebnissequenz im Allgemeinen wohldefiniert. Für Fälle, in denen die Reihenfolge eines Ergebnisses keine Rolle spielt, existiert die Funktion `unordered`. Sie liefert konzeptionell eine gegebene Sequenz in einer beliebigen anderen Reihenfolge zurück.



Syntax 26: Bedingte Ausdrücke



Syntax 27: Quantifizierte Ausdrücke

Der Einsatz dieser Funktion ist als Hinweis an die Anfrageverarbeitung zu verstehen, der sich eventuell weiter reichende Optimierungsmöglichkeiten bieten, sobald sie nicht mehr auf die Reihenfolge des Ergebnisses achten muss. Bei SQL ist dies genau anders herum: Ergebnisse müssen explizit sortiert werden, sonst ist die Reihenfolge beliebig.

3.4.10 Bedingte Ausdrücke

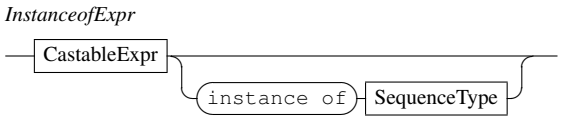
Bedingte Ausdrücke sind hinreichend bekannt. Als Besonderheit in XQuery ist lediglich anzumerken, dass der `else`-Zweig immer vorhanden sein muss (Syntax 26).

3.4.11 Quantifizierte Ausdrücke

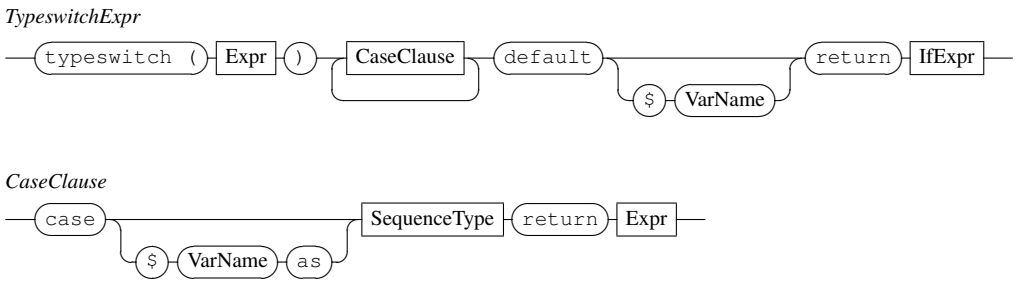
Existenz- und Allquantifizierung (Syntax 27; QuantifiedExpr) beziehen sich immer auf das kartesische Produkt der Mengen von Items aus verschiedenen Sequenzen, ähnlich wie bei `for`-Klauseln in FLWOR-Ausdrücken.

3.4.12 Ausdrücke auf Sequenztypen

Eine Reihe von Ausdrücken nimmt Bezug auf Typen in der Sequenztyp-Notation aus Abschnitt 3.3.4 auf Seite 23.



Syntax 28: Instance Of



Syntax 29: Typeswitch

Instance Of

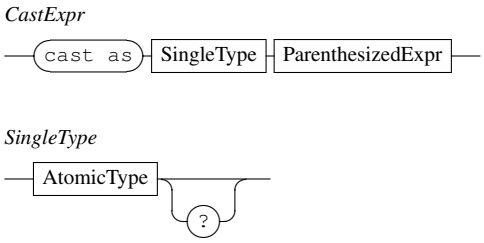
Java-Programmierer kennen einen Operator ähnlichen Namens und gleichen Verhaltens: instanceof. Der XQuery-Operator `instance of` (Syntax 28) liefert *wahr*, falls der Wert seines ersten Operanden zum Typ passt, der durch den zweiten Operanden gegeben ist. Diese Entscheidung verläuft nach den Regeln des Sequenztyp-Matching (Abschnitt 3.3.4).

Typeswitch

Analog zur switch-Anweisung in C++ oder Java, die tief geschachtelte if-then-else-Ketten zur wertbasierten Verzweigung zu vermeiden hilft, gibt es in XQuery einen `typeswitch`-Ausdruck zum Test eines Ausdrucks gegen eine Reihe von Typen (Syntax 29; TypeswitchExpr).

Cast

Von Zeit zu Zeit kann es notwendig sein, einen Wert in einen anderen Datentyp zu überführen. Dieses Casting (Syntax 30 auf der nächsten Seite; CastExpr) betrifft



Syntax 30: Cast

Syntax 31: Castable

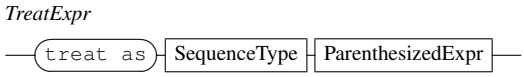
nur atomare Werte und ist nur für bestimmte Kombinationen aus Wert und Typ erfolgreich.

Castable

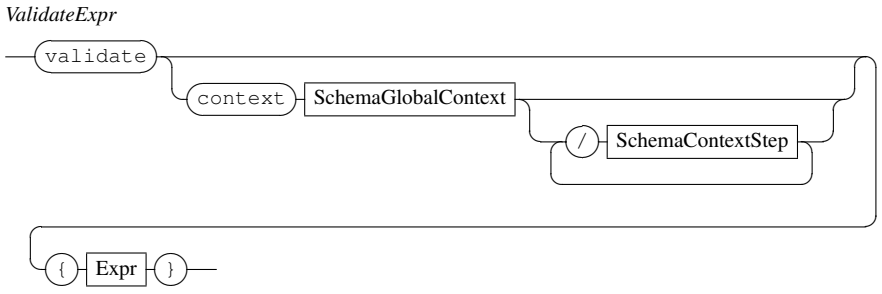
Um Fehler zu umgehen, die aus einer falschen Kombination von Wert und Typ bei einem Casting hervorgehen, lässt sich der Erfolg dieser Typumwandlung vorab mit dem Operator `castable as` testen (Syntax 31).

Treat

Im Gegensatz zu `cast` ändert `treat` (Syntax 32 auf der nächsten Seite) nicht den Wert oder dynamischen Typ seines Operanden, sondern nur den statischen Typ. Das hat den Sinn, zuzusichern, dass ein Ausdruck zur Laufzeit einen spezifischeren Typ haben wird, als aus der statischen Typanalyse allein vom System zu ermitteln ist. Ein Laufzeitfehler tritt auf, falls diese Zusicherung nicht eingehalten werden sollte.



Syntax 32: Treat



Syntax 33: Validierung

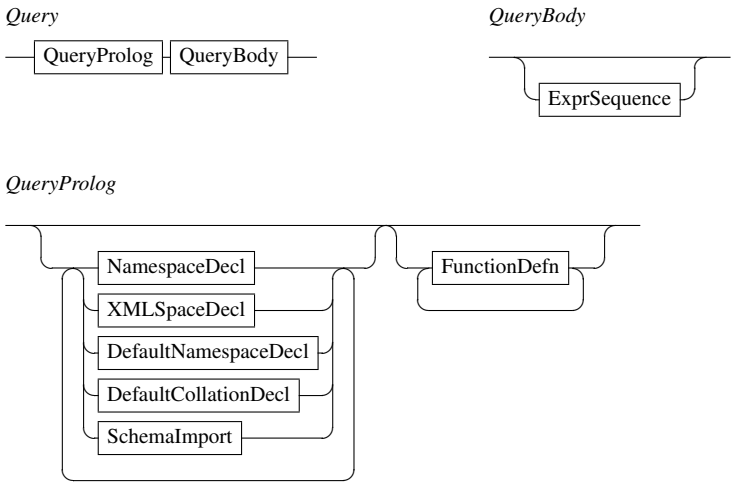
3.4.13 Validierungsausdrücke

Ein Validierungsausdruck (Syntax 33) validiert sein Argument gegen die sichtbaren Schemadefinitionen (Abschnitt 3.3.1 auf Seite 20). Die Element- und Attributknoten werden dabei durch validierte Versionen mit neuer Identität ersetzt.

Die Angabe eines Kontextes für die Validierung ist notwendig, wenn ein nicht schemaglobales Element zu validieren ist.

3.5 Der Anfrageprolog

Vor der eigentlichen XQuery-Anfrage (QueryBody) können in einem so genannten Prolog (QueryProlog) bestimmte Deklarationen und Definitionen untergebracht werden, die den Kontext für die Anfrage einrichten (Syntax 34 auf der nächsten Seite). Hier können Namensräume deklariert, Funktionen definiert und Standardeinstellungen getroffen werden. So gehören zu einer Anfrage mehrere Standard-Namensräume und eine voreingestellte Collation sowie eine Strategie zur Behandlung von Whitespace. Außerdem können im Prolog Schemainformationen importiert werden.



Syntax 34: Struktur von XQuery-Anfragen im Großen

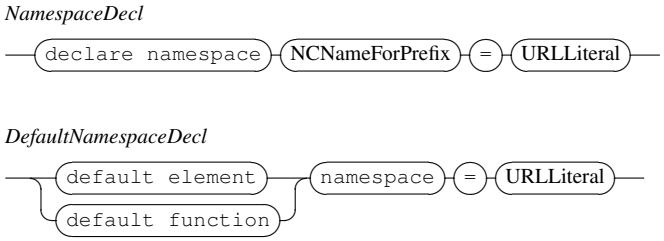
3.5.1 Deklaration von Namensräumen

Eine Namensraum-Deklaration bindet ein Präfix an die einen Namensraum identifizierende URI (Syntax 35 auf der nächsten Seite; (Default)NamespaceDecl). Diese Bindungen werden in den statischen Kontext (Abschnitt 3.3.1 auf Seite 20) übernommen. Von vornherein sind schon vier häufig benötigte Namensraumpräfixe definiert, nämlich `xml` für den XML-Namensraum, `xs` und `xsi` für die Namensräume von XML-Schema und Instanzdokumenten und `fn` für den Namensraum der Funktionsbibliothek von XQuery (W3C 2002f).

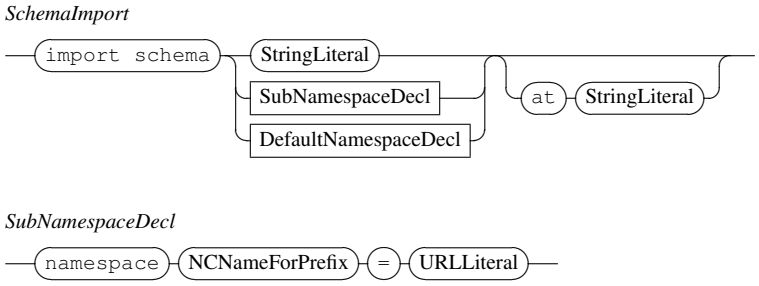
Bei Elementkonstruktoren können darüber hinaus, wie aus XML-Dokumenten gewohnt, `xmlns`-Attribute für Namensraumdeklarationen verwendet werden.

3.5.2 Schema-Importe

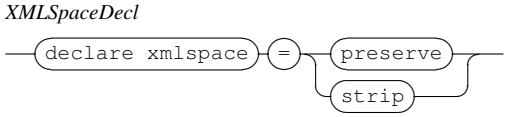
Ein Schema-Import (Syntax 36 auf der nächsten Seite; SchemaImport) macht einem XQuery-Prozessor die Typendefinitionen sowie Element- und Attribute-Deklarationen aus einem XML-Schema bekannt. Die so importierten Schemata bilden die Basis für erweiterte Typprüfungen während der Analysephase.



Syntax 35: Deklaration von Namensräumen



Syntax 36: Schema-Importe



Syntax 37: Xmlspace: Behandlung von Whitespace

Syntax 38: Standard-Collation

3.5.3 Xmlspace-Deklarationen

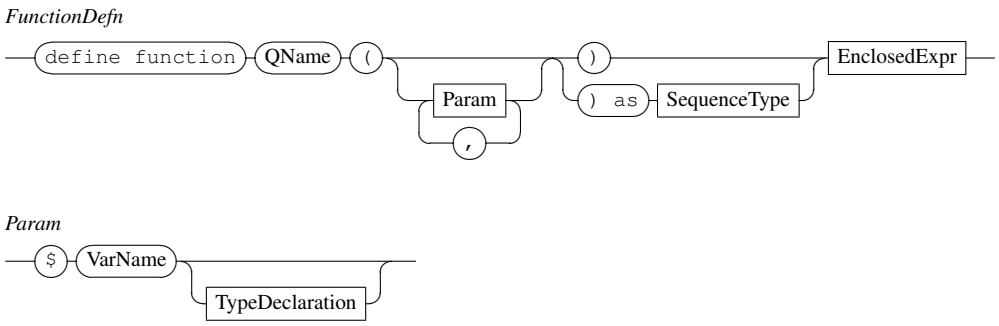
Die Art und Weise der Behandlung von Whitespace an bestimmten Positionen in Element- und Attribut-Konstruktoren kann mit einer Xmlspace-Deklaration eingestellt werden (Syntax 37; XMLSpaceDecl). Die Voreinstellung (strip) erleichtert die übersichtliche Formatierung von XQuery-Ausdrücken, ohne jeden Whitespace auch in das Anfrageergebnis zu übernehmen.

3.5.4 Standard-Collation

Die im Prolog angegebene Collation (Syntax 38; DefaultCollationDecl) wird zugrunde gelegt, wenn bei entsprechenden Funktionen oder Operatoren, die Texte verarbeiten, keine explizite Einstellung vorgenommen wird.

3.5.5 Funktionsdefinitionen

XQuery erlaubt dem Benutzer die Definition eigener Funktionen zur Verwendung in XQuery-Ausdrücken (Syntax 39 auf der nächsten Seite; FunctionDefn). Parameter und Ergebnis können (müssen aber nicht) mit Sequenztypen getypt werden. Auch rekursive Funktionen sind erlaubt. Zum Beispiel bestimmt folgende Funktion die Tiefe eines Elementbaumes:



Syntax 39: Funktionsdefinitionen

```
define function depth($e as element) as xs:integer
{
  {-- An empty element has depth 1 --}
  {-- Otherwise, add 1 to max depth of children --}
  if (empty($e/*)) then 1
  else max(for $c in $e/* return depth($c)) + 1
}
```

4 XQuery und Java

Erst jetzt, wo uns die Sprache XQuery aus dem letzten Kapitel im Detail bekannt ist, können wir daran denken, einen Vorschlag für eine Kopplung dieser Sprache mit Java zu entwerfen. Die Kenntnis von Java wird vorausgesetzt.

Hinter dem Streben nach einer engen Kopplung von XQuery und Java steckt der Wunsch, XQuery in der einfachst möglichen Art und Weise in einem Anwendungsprogramm nutzbar zu machen. Idealerweise sollte ein Entwickler nicht viel mehr als die beiden Sprachen Java und XQuery zu beherrschen brauchen, um sie gewinnbringend ohne viel Aufwand nebeneinander einsetzen zu können: Java für die prozedurale Verarbeitung, XQuery für den deklarativen Zugriff auf benötigte Daten. Außerdem sollte er sich in keine Abhängigkeiten bezüglich Hersteller der Anfrageverarbeitung, Aufenthaltsort der Daten oder Formaten für den Transport begeben müssen. Ein enge Kopplung von XQuery und Java verspricht dem Entwickler bessere Prüfungen auf Fehler und kompaktere und übersichtlichere Programme, was insgesamt seine Arbeit effizienter machen kann.

4.1 Vorbemerkungen

Eine Kopplung von XQuery und Java kann aus mindestens zwei verschiedenen Perspektiven beschrieben werden:

1. Bei der Sicht eines Benutzers dieser Kopplung steht im Vordergrund, wie die Anwendungsprogrammierschnittstelle aussieht, also welche logischen Konzepte ihr zugrunde liegen, wie ein Verarbeitungsmodell aussieht, welche Operationen möglich sind und welche Semantik sie haben, aber auch ganz grundlegend die Syntax der Kopplung: Wie werden Java-Programme formuliert, die den zu entwickelnden Kopplungsmechanismus nutzen?

2. Die zweite Perspektive betrifft die Realisierung der dem Benutzer angebotenen Konzepte. Wichtig ist hier vor allem eine sinnvolle Umsetzung, wobei diese Sinnhaftigkeit im Umfeld von Datenbanksystemen und Anfra-

gesprachen vor allem in Performance gemessen wird: Wie können die vom Benutzer gestellten Anfragen möglichst effizient ausgewertet werden?

Die Reihenfolge der Entwicklung dieser beiden Sichten scheint auf den ersten Blick gegeben: Natürlich muss erst definiert werden, wie die Syntax und Semantik der Kopplung aussehen sollen, bevor an Konzepte zu ihrer Realisierung gedacht werden kann. Anderseits ist es möglich, dass bestimmte Benutzeranforderungen verworfen werden (müssen), weil ihre generelle Realisierbarkeit in Frage steht. Nichtsdestotrotz werden wir im Folgenden die Kopplung in dieser Reihenfolge entwickeln; beim ersten Schritt weisen wir jedoch immer dann auf Möglichkeiten der Einschränkung oder Variation hin, falls sich schon andeuten sollte, dass der entsprechende Punkt Schwierigkeiten bei einer Realisierung bereiten könnte.

4.2 Leitfragen

Um die abstrakte Vorstellung von einer Kopplung von XQuery und Java etwas einzuengen, geben wir mit den folgenden Fragen eine Zielrichtung für die Entwicklung einer Kopplung vor.

Welche Funktionalität benötigt eine Kopplung von XQuery und Java? Welche Möglichkeiten erwartet ein Benutzer von ihr? Als Grundfunktionalität muss mindestens das Folgende möglich sein:

- Angabe und Ausführung von XQuery-Anfragen, um die den Benutzer interessierenden Daten zu beschreiben.
- Weiterverarbeitung der Anfrageergebnisse, also auch deren Übernahme in die Java-Welt.

Darüber hinaus wären die Möglichkeit zur Parametrisierung von Anfragen oder die Verwendung von Java-Funktionen in XQuery-Anfragen nützlich.

Welche Punkte sollten außerdem noch bei einer Kopplung von XQuery und Java berücksichtigt werden? Die wichtigsten Stichpunkte sind die folgenden:

- Syntaktische Verbindung von Java und XQuery
- Programmiermodell: Wie werden Anfragen gestellt und wie kann mit deren Ergebnissen verfahren werden?

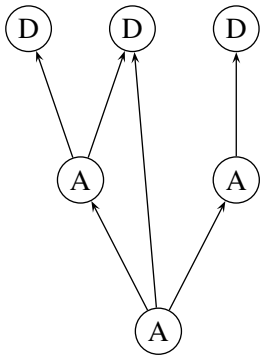


Abbildung 4.1: Anfragen (A) sowohl gegen ursprüngliche Dokumente (oder allgemeiner Datenmodell-Instanzen) (D) als auch gegen andere Anfragen.

- Laufzeit gegenüber Compilezeit: Was kann zur Erhöhung der Sicherheit und Robustheit zur Compilezeit geprüft werden (vor allem im Hinblick auf Typen)? Welche Informationen können zur Vorab-Optimierung herangezogen werden?
- Nahtlosigkeit: Das direkte Aufeinandertreffen nicht zueinander passender Konzepte ist möglichst zu vermeiden.

4.3 Grundidee

Am Anfang dieses Kopplungsvorschlages steht die Idee, die Abgeschlossenheit von XQuery bezüglich seines Datenmodells auch bei der Anbindung an Java auszunutzen. Sowohl Eingabe als auch Ausgabe eines XQuery-Ausdrucks sind Instanzen des Datenmodells, also ist es möglich, weitere Anfragen gegen Anfrageergebnisse zuzulassen. Graphisch dargestellt ergibt das eine Struktur aus Datenquellen und Anfragen wie in Abbildung 4.1 (Anfragegraph). Jeder Knoten stellt dabei logisch eine Instanz des Datenmodells dar, die entweder direkt gegeben oder durch eine Anfrage definiert ist.

Welchen Nutzen bringt eine solche stufenweise Anfrageformulierung? Als erstes vermeidet man einen unnötigen Bruch in der Verarbeitung: Warum sollte man nach einer ersten Anfrage gezwungen sein, über den Ergebnisbaum mit Java-Methoden zu navigieren, wenn man neben dem Ergebnis als Ganzem etwa noch

Teile daraus betrachten möchte? XQuery selbst ist für diese Aufgabe viel besser geeignet, weil es deklarative und damit optimierbare Verarbeitungsschritte zulässt und außerdem übersichtlicher dargestellt werden kann.

Als zweiter Vorteil können gleichartige Teile mehrerer Anfragen in eine vorgeschaltete Anfrage ausgelagert werden. Man kann dies als Definition von anwendungsspezifischen Sichten auffassen, wobei der Sichtbegriff hier ähnlich wie bei SQL zu verstehen ist. Nachfolgende Anfragen werden dadurch einfacher. Zusammenhänge zwischen Anfragen werden explizit gemacht. Die Redundanz in der Menge der formulierten Anfragen verringert sich; dadurch wird auch deren Wartbarkeit erhöht.
Listing 4.1 auf der nächsten Seite zeigt aufeinander aufbauende Anfragen in Aktion. Das Beispiel lehnt sich an die beiden Anfragebeispiele zu Beginn von Kapitel 3 an und enthält innerhalb des Java-Codes in eckige Klammern eingeschlossene XQuery-Ausdrücke, die sich über Variablennamen auf andere Anfragen beziehen. Die genaue Syntax und auch die anderen Konstrukte, die man entdecken kann, werden wir später behandeln; hier genügt ein intuitives Verständnis. Um dieses zu unterstützen, skizziert Abbildung 4.2 auf Seite 60 den Anfragegraph zu diesem Beispiel, also die Menge aller Anfragen und Datenquellen und ihre Beziehung zueinander.

4.4 Konkretisierung

Wie das einführende Beispiel schon angedeutet hat, gehen wir im Folgenden davon aus, dass eine Art Präprozessor für unseren Kopplungsvorschlag existieren wird, der es ermöglicht, schon zur Compilezeit Transformationen und Optimierungen vorzunehmen. Damit sind wir nicht allein auf eine Aufrufschnittstelle (CLI) beschränkt, sondern können auch eine syntaktische Einbettung von XQuery in Java zulassen.

Jede XQuery-Anfrage ist ein Ausdruck, der einen Wert aus der Sicht des Datenmodells, also eine Sequenz, zurückliefert. Anfragen können deshalb sinnvollerweise an den gleichen Stellen in einem Java-Programm auftauchen wie Java-Ausdrücke und liefern dort bei ihrer Auswertung eine Instanz des Datenmodells. Um diesen Wert in folgenden Anfragen verwenden zu können, ist eine Bindung an einen Namen erforderlich. Hierfür ist der übliche Mechanismus der Zuweisung eines Objekts an eine Java-Variable geeignet. Diese Variablen können dann in weiteren Anfragen benutzt werden.

Listing 4.1: Aufeinander aufbauende Anfragen

```
// We are interested in books.
element book* allBooks =
  [document("http://www.bn.com/bib.xml")/bib/book];

// Get some input from the user.
String publisher = "Addison-Wesley";
int threshold = 1991;

// Create a list of his favorite books.
item favorite = [
  <bib>
  {
    for $b in $allBooks
    where $b/publisher = $publisher
      and $b/@year < $threshold
    return
      <book year="{ $b/@year }">
        { $b/title }
      </book>
  }
  </bib>
];

// Pass this list to a method and extract some information.
Mail offer = new Mail("joe@user.com");
offer.attach(favorite);
offer.setSubject([count($favorite/book)]
   + " exciting books: "
   + [$favorite/book[1]/title] + " ...");
offer.send();

// Some iteration reusing $allBooks.
for $b as element book in $allBooks[@year = 2003]
do {
  System.out.println([$b/@isbn]);

  // type error, read does not accept books:
  //  public void read(element mind+ m)
  this.read($b);
}
```

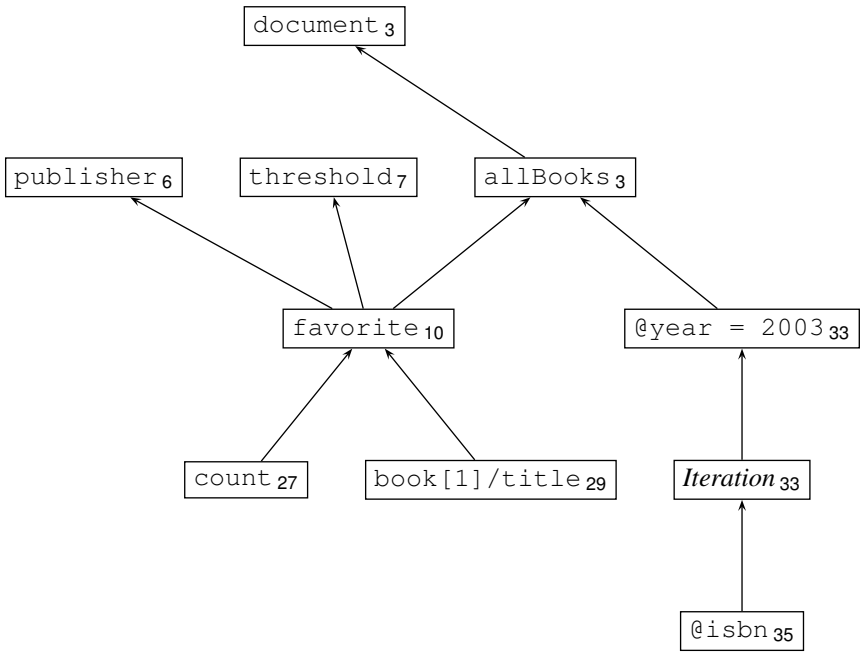


Abbildung 4.2: Anfragegraph zu Listing 4.1. Die Knoten sind mit einem Stichwort oder dem zugeordneten Variablennamen markiert; tiefgestellte Ziffern referenzieren die Zeilennummern, in denen die Anfrage oder die Datenquelle beginnt.

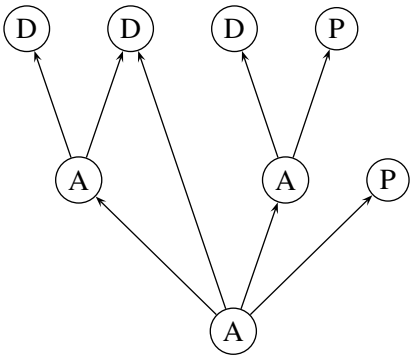


Abbildung 4.3: Parameter (P) als spezielle Datenmodell-Instanzen (vgl. Abbildung 4.1 auf Seite 57)

4.4.1 Parametrisierung als Spezialfall

Andere Schnittstellen zwischen Datenbank- und Programmiersprachen kennen parametrisierte Anfragen. In JDBC gibt es z. B. die Möglichkeit, Fragezeichen in Anfragen zu verwenden, um Parameter zu markieren. Parameter sind meist kleine, einfache Teile von Anfragen, die je nach Sichtweise ausgetauscht oder mit verschiedenen Werten befüllt werden können. Diese Parameter haben oft einfache Typen wie Strings oder Zahlen. Hinter dem Konzept der Parametrisierung steht die Idee, Gemeinsamkeiten zwischen Anfragen auszunutzen, um dem Benutzer die Anwendung vieler ähnlicher Anfragen zu erleichtern und vor allem um besser optimieren zu können. So kann in vielen Fällen für eine Anfrage mit Parametern schon eine Auswertungsstrategie festgelegt werden, bevor konkrete Parameterwerte bekannt sind.

Aus der Perspektive von XQuery sind Parameter nur eine spezielle Art von Ausdrücken und werden wie alle anderen zu einer Instanz des Datenmodells ausgewertet. Damit ist aber die Verwendung von Parametern einfach ein Spezialfall des Konzepts der aufeinander aufbauenden Anfragen: Ein Parameter einer Anfrage ist einfach eine zusätzliche Datenquelle, gegen die sich die Anfrage richtet. Abbildung 4.3 zeigt dieses Verhältnis von Parametern und anderen Anfragen in einem gegenüber Abschnitt 4.3 erweiterten Anfragegraph.

4.4.2 Sichten auf die Daten

Für die Weiterverarbeitung der Anfrageergebnisse bleibt die Frage, in welcher Form diese Ergebnisse vorliegen und für den Benutzer zugänglich sind.

Natürlich kann das Datenmodell von XQuery nicht vor dem Benutzer versteckt werden. Weitere Anfragen richten sich gegen diese Struktur, also muss sich der Benutzer zur Formulierung solcher Anfragen des Datenmodells und der verwendeten Schemata darauf bewusst sein. Wie der Benutzer konkret auf dieser Sicht arbeiten kann, wird in Abschnitt 4.5 betrachtet.

Aber es ist denkbar, auch andere Sichten auf die Instanzen des Datenmodells anzubieten, etwa auch zur Übernahme von Daten in bestehende Anwendungen. Hierzu wäre die Implementierung einer DOM-Sicht geeignet. Auch eine Abbildung auf Graphen komplexerer und eher anwendungsorientierter Objekte (im Sinne des Data-Binding-Konzepts, beispielsweise wie bei Castor (Castor 2003)) könnte die Weiterverwendung der Daten in bestimmten Fällen vereinfachen.

Diese Sichten sind allerdings als zweitrangig gegenüber der Datenmodell-Sicht anzusehen, da XQuery-Anfragen nur gegen letztere formulierbar sind. Typischerweise steht also eine Anfrage oder ein Anfragegraph am Anfang der Verarbeitung, die eine Instanz des Datenmodells liefert. Über diese können dann die anderen Sichten gelegt werden. Interessant ist noch die Frage, inwieweit ein flexibles Wechseln zwischen den Sichten möglich ist, also auch zurück zur Datenmodell-Sicht für weitere Anfragen, nachdem schon Operationen auf einer anderen Sicht ausgeführt wurden.

4.5 Zugang des Java-Codes zu den Anfrageergebnissen

Wir betrachten hier die grundlegende Sichtweise auf die Anfrageergebnisse, nämlich in Form des Datenmodells. Jedes Anfrageergebnis ist eine Sequenz, bestehend aus atomaren Werten oder aus Knoten, die normalerweise in größere Baumstrukturen aus Knoten eingebettet sind.

Zur Weiterverarbeitung solcher Anfrageergebnisse sind also im Wesentlichen drei Zugriffsmöglichkeiten aus Java notwendig:

1. Zugriff auf die einzelnen Items einer Sequenz
2. Auswahl von Teilstrukturen in Bäumen
3. Übernahme von atomaren Werten in entsprechende Java-Werte

4.5.1 Items aus einer Sequenz

Die Auswahl eines bestimmten Item aus einer Sequenz ist prinzipiell mit XQuery-Mitteln möglich. Um beispielsweise das siebte Item aus der Sequenz `$seq` zu erhalten, kann der einfache Ausdruck `$seq[7]` verwendet werden. Isoliert wird solch ein Ausdruck allerdings kaum vorkommen: Warum sollte man eine ganze Reihe von Ergebnissen ermitteln, wenn man nur am siebten interessiert ist?

Typisch ist hingegen eine Iteration über das Anfrageergebnis, bei der jedes Item der Reihe nach betrachtet wird.

Naiv lässt sich eine solche Iteration analog zu einer Iteration über ein Array formulieren:

```
for (int i = 1; i <= [count($seq)]; i++) {
  item t = [$seq[$i]];
  // do something with t
}
```

Allerdings muss zur Auswertung der Abbruchbedingung die Länge der Sequenz vorab bekannt sein. Dies ist ungünstig, weil so eine mit der Anfrageauswertung Schritt haltende Verarbeitung nicht möglich ist.

Aber sogar, wenn die Länge der Sequenz zunächst unbekannt ist oder aus Effizienzgründen unbekannt bleiben soll, ist mit XQuery-Mitteln eine Iteration möglich. Diese lehnt sich an das typische Iteratorvorgehen an, das dem Durchlauf einer verketteten Liste entspricht: Beginne beim ersten Item und nehme so lange das nächste, bis keine Items mehr übrig sind.

```
int i = 1;
item? t = [$seq[$i]];
while ([not(empty($t))]) {
  // do something with t
  i++;
  t = [$seq[$i]];
}
```

Für solche einfachen und häufig vorkommenden sequentiellen Durchläufe könnte man allerdings auch eine vereinfachte und an die for-Klausel angelehnte Syntax anbieten:

```
for $p as item at $i in $seq
do { /* something (in Java) with p */ }
```

Dies hat den Vorteil, dass die Iterationsabsicht dem System explizit bekannt gemacht wird und es so bessere Optimierungsmaßnahmen ergreifen kann. Die vorhergehenden Iterationsansätze stellen sich demgegenüber aus der Sicht des Systems als eine Reihe von potentiell unabhängigen Punktanfragen gegen die Sequenz dar. (Ein Zusammenhang ist höchstens durch aufwändige Analyse der Java-Schleifen zu ermitteln.)

Alle diese Überlegungen zu Iterationen betreffen nur für den Fall, dass jedes einzelne Item einer Sequenz mit Java-Mitteln weiterbearbeitet werden soll. Natürlich bleibt eine mengenorientierte XQuery-Anfrage auf der ganzen Sequenz weiterhin möglich.

4.5.2 Teilstrukturen aus einem Baum

Für die Aufgabe, aus einem baumartigen Anfrageergebnis Teile zu extrahieren, ist XQuery geradezu prädestiniert. Für diesen Zweck lassen sich also gerade Anfragen einsetzen, die auf anderen Anfragen aufbauen. Dabei kann eine „Anfrage“ hier oft so ein simpler Ausdruck wie `$person/name` sein.

4.5.3 Atomare Werte nach Java

Durch Iteration über Sequenzen oder Herunterbrechen auf immer kleinere Teilbäume wird bei atomaren Werten schließlich der Punkt erreicht, an dem ein echter Übergang zu Java-Datentypen unumgänglich ist.

Hierfür kann man geeignete Casting-Regeln definieren, so dass Zuweisungen von atomaren XQuery-Werten an entsprechend getypte Java-Variablen oder ihre Verwendung in Java-Ausdrücken möglich werden.

Dabei ist zu bedenken, dass Werte in XQuery immer nur als Sequenzen vorliegen, eine Sequenz mit nur einem Item aber diesem Item entspricht. Außerdem sollten Typumwandlungen, die in XQuery möglich sind, auch bei der Abbildung von atomaren XQuery- in primitive Java-Typen beachtet werden (vgl. Abschnitt 3.3.4 auf Seite 25).

Auch die umgekehrte Richtung sollte möglich sein, also der Einsatz von primitiven Java-Werten als entsprechend getypte XQuery-Sequenzen, zum Beispiel zur Parametrisierung von Anfragen.

Listing 4.2: Isolierter und gewöhnlicher Prolog

```
void func(node n) {
   declare namespace bc = "uri1";
   import schema namespace sn = "uri2";

   item* queryA = [$n/bc:name];
   item* queryB = [
 declare namespace de = "uri3"
      $n/de:thing[@bc:color]
   ];
}
```

4.6 Isolierte Prologe

Die Informationen im Prolog einer XQuery-Anfrage geben durch Namensräume, Funktionsdefinitionen und ähnliches (Abschnitt 3.5 auf Seite 50) eine Umgebung für die Anfrage an sich vor. Zur Zeit existiert für den XQuery-Prolog noch kein Modularisierungskonzept, so dass in jeder Anfrage die möglicherweise gleichen Informationen wiederholt werden müssen.

Die XQuery-Arbeitsgruppe hat den Bedarf für eine Modularisierung vor allem für Funktionsbibliotheken erkannt, diskutiert aber noch über deren Realisierung (W3C 2002c, Issues). Inhalte dieser Diskussion oder gar erste Lösungsansätze wurden noch nicht veröffentlicht. Bis zu einer Klärung führen wir für die Einbettung von XQuery in Java eine eigene Möglichkeit ein, um gleiche Prologteile aus mehreren Anfragen herauszuziehen.

Eine einfache Möglichkeit hierzu bieten isolierte Prologe, also Prologe, die im Java-Programm platziert werden und in einem bestimmten Gültigkeitsbereich alle weiteren XQuery-Ausdrücke beeinflussen. Dieser Gültigkeitsbereich sollte sich praktischerweise an die Gültigkeitsbereiche von anderen Java-Deklarationen wie Variablen anlehnen, die sich an der Klassen- und Block-Struktur orientieren. Das Attribut „isoliert“ soll darauf hinweisen, dass diese Prologe nicht unmittelbar Teil einer XQuery-Anfrage sind, sondern entfernt von den durch sie beeinflussten XQuery-Ausdrücken (den Körpern der Anfragen; vgl. Abschnitt 3.5 auf Seite 50) auftauchen.
In Listing 4.2 sind ein isolierter und ein echter Prolog einander gegenübergestellt.

Der isolierte Prolog (Namensraum-Deklaration und Schemaimport) in den ersten Zeilen hat Gültigkeit bis zum Ende des umgebenden Blockes, beeinflusst also beide XQuery-Anfragen. Die zweite Anfrage bringt zusätzlich noch ihren eigenen Prolog für eine lokale Namensraum-Deklaration mit.

5 Realisierungsansätze

Im Raum steht nach dem letzten Kapitel das Konzept der aufeinander aufbauenden Anfragen als Grundlage einer Einbettung von XQuery in Java. Im Folgenden werden einige Aspekte der konkreten Umsetzung dieser Einbettung betrachtet. Wir beginnen mit Überlegungen zu einer schönen, aber auch zweckmäßigen Syntax; die Ergebnisse sind teilweise schon aus einigen Beispielen bekannt. Es folgt dann die Umsetzung der Typüberprüfung und schließlich die Auswertung aufeinander aufbauender Anfragen an sich.

5.1 Syntax der Einbettung

Sichtbarster Bestandteil einer Spracheinbettung ist ihre Syntax. Für folgende Situationen bei der Benutzung der Kopplung von XQuery und Java muss eine Syntax bereitgestellt werden:

- Ein XQuery-Ausdruck wird anstelle eines Java-Ausdrucks verwendet.
- Eine Java-Variable wird mit einem Sequenztyp deklariert.
- Eine Java-Variable wird in einem XQuery-Ausdruck verwendet. Das betrifft Variablen mit einem Sequenztyp (aufeinander aufbauende Anfragen), aber auch andere Java-Typen, die implizit in einen Sequenztyp überführt werden können (Parametrisierung).
- Ein isolierter Prolog wird deklariert. Er soll für eine Menge von folgenden XQuery-Ausdrücken gelten.
- Für jedes Item in einer Sequenz wird ein Block Java-Code ausgeführt. Eine Syntax für diese Iteration wurde bereits in Abschnitt 4.5.1 auf Seite 63 vorgeschlagen.

Dabei sind die folgenden informellen Anforderungen an eine Syntax zu beachten:

- Die Syntax sollte leicht zu analysieren sein. Vor allem sollten keine Mehrdeutigkeiten auftreten, die sich nicht in späteren Phasen der Sprachanalyse auflösen ließen.
- Aus Benutzersicht sollte auf Natürlichkeit sowie auf die schwer messbare Ästhetik Wert gelegt werden: Die Syntax sollte für Kenner von Java und XQuery leicht zu schreiben und zu lesen sein.

5.1.1 XQuery-Ausdrücke

Zwischen XQuery-Ausdrücken und Java-Ausdrücken besteht eine enge Analogie: Beide können jeweils beliebig kombiniert werden und stehen für einen bestimmten Wert, der durch ihre Auswertung ermittelt werden kann. Deswegen sollten XQuery-Ausdrücke genau dort in einem Java-Programm erlaubt sein, wo es auch gewöhnliche Java-Ausdrücke sind.

Um XQuery-Ausdrücke von Java-Ausdrücken unterscheiden zu können, bietet sich eine bestimmte Art der Klammerung von XQuery-Ausdrücken an. In Anlehnung an das `#sql` von SQLJ wären `#xquery` oder `#xq` Alternativen, um den Beginn eines XQuery-Ausdrucks anzudeuten. Es gibt allerdings einen grundlegenden Unterschied zwischen SQLJ und der hier vorgeschlagenen XQuery-Einbettung: In SQLJ haben Einbettungen Anweisungscharakter, hier Ausdruckscharakter.

Obwohl auch SQL prinzipiell wie XQuery aus geschachtelten Ausdrücken aufgebaut ist, werden dort einzelne Anfragen völlig isoliert betrachtet. Vereinfacht dargestellt werden Anfragen in einem Schritt formuliert, dann ausgewertet, danach wird das flach strukturierte Ergebnis Tupel für Tupel abgearbeitet. Entsprechend diesem Verwendungsmuster von Anfragen kombiniert SQLJ immer eine Anfrage mit Informationen zu ihrer Bindung an Iteratoren und Kontexte zu einer Art Deklaration, die aus Java-Sicht eine Anweisung darstellt. Danach kann dann mit weiteren Anweisungen der Iterator benutzt werden. Damit taucht das `#sql` bei üblicher Formatierung von Quelltexten immer nur am Anfang einer Zeile auf, nie geschachtelt in anderen Strukturen[1].

Im Gegensatz dazu stehen bei unserer Kopplung XQuery-Anfragen direkt für den Wert, zu dem sie ausgewertet werden, und können deswegen auch als kleiner Teil innerhalb einer Anweisung auftreten, geschachtelt in andere Java-Ausdrücke.

[1] abgesehen von der Schachtelung von Anweisungen in Blöcke

Deswegen ist eine Markierungsweise angebrachter, die diesem Ausdruckscharakter entspricht. Da es mit diesem Einbettungsansatz außerdem möglich und sinnvoll ist, ein Vielzahl sehr kurzer Ausdrücke nacheinander zu verwenden, sollte die Markierung sehr kurz sein.

Damit deuten alle Argumente auf Klammern im Wortsinn zur Markierung von XQuery-Ausdrücken. Runde und geschweifte Klammern werden schon für Java-Ausdrücke und -Blöcke gebraucht. Spitze Klammern würden auf den ersten Blick zu XML passen, allerdings ergäben sich so Schachtelungen der Form

```
< ... <tag/> ... >
```

die der gewohnten XML-Syntax zuwiderlaufen, denn dort können Paare spitzer Klammern nie geschachtelt auftreten. Eckige Klammern hingegen werden in Java bisher nur für den Zugriff auf Arrays oder ihre Typdefinition gebraucht. Beide Möglichkeiten können nie an der Position eines Ausdrucks auftauchen. Deswegen verwenden wir hier eckige Klammern zur Markierung von XQuery-Ausdrücken in Java.

5.1.2 Sequenztypen

Analog zu den Ausdrücken können Sequenztypen überall dort eingesetzt werden, wo sonst Java-Typen erlaubt sind. Zur Markierung können auch hier die eckigen Klammern dienen.

Als Alternative soll hier noch die Möglichkeit betrachtet werden, Sequenztypen überhaupt nicht speziell zu markieren. Dieser Ansatz macht die Analyse des Quelltextes schwieriger, sorgt aber für mehr Symmetrie zwischen Java- und XQuery-Typen. Bei der Auflösung von Mehrdeutigkeiten geben lange Typnamen mit Bestandteilen wie `context` oder `of type` klare Hinweise auf XQuery-Typen; bei Typen, die aus einem Wort bestehen, gibt es Probleme: Für Typen wie `comment` oder `text` werden Mehrdeutigkeiten durch die Konvention, Java-Klassennamen mit einem Großbuchstaben zu beginnen, vermieden. Bei atomaren Typen wie `int` oder anderen selbst definierten sieht das anders aus. Hier könnten die Namensräume aus XQuery Abhilfe schaffen und die Mehrdeutigkeiten auflösen. Erleichtert wird das durch die Tatsache, dass Doppelpunkte, die Namensraum-Präfixe von lokalen Namen trennen, nicht in Java-Bezeichnern auftreten können.

5.1.3 Java-Variablen in XQuery

Zur Verwendung von Java-Variablen in XQuery-Ausdrücken ist einzig und allein die Frage zu klären, wie sie von XQuery-Variablen unterschieden werden.

Dies könnte wie in SQLJ durch ein bestimmtes neues Präfix der Variablennamen (dort ein Doppelpunkt) geschehen. Ein solches Präfix definiert im Prinzip einen neuen Namensraum. Da XQuery im Gegensatz zu SQL schon ein eigenes Namensraum- und Präfixkonzept besitzt, sollte dieses verwendet werden, anstatt Verwirrung durch noch eine Art der Markierung von bestimmten Variablen zu stiften. Konkret würde dies die Definition eines speziellen Namensraums, also einer URI, für die aus dem umgebenden Java-Code sichtbaren Variablen bedeuten. Aus praktischen Gründen könnte diese Definition von einem neuen vordefinierten Präfix wie etwa `java` begleitet werden.

Andererseits kann es in vielen Fällen gar nicht nötig sein, explizit anzugeben, dass es sich bei einer Variable um eine Java-Variable handelt, nämlich dann, wenn eine XQuery-Variable des angegebenen Namens gar nicht existiert. Die einfachste Lösung ist also, alle sichtbaren Java-Variablen (eines Typs, der als XQuery-Ausdruck verwendet werden kann) mit dem gleichen Namen in den Ausdruckskontext einer XQuery-Anfrage zu übernehmen. Diese Variablen können dann wie jede andere XQuery-Variable innerhalb der Anfrage benutzt oder auch durch gleichnamige überdeckt werden.

5.1.4 Prologe

Isolierte Prologe, die für eine Reihe von XQuery-Ausdrücken Gültigkeit haben, könnte man wie andere XQuery-Anfragen, die Prologe enthalten können, durch Einschließung in eckige Klammern vom umgebenden Java-Code trennen. Allerdings wäre dann der Unterschied im Gültigkeitsbereich zwischen isolierten Prologen und Prologen vor XQuery-Ausdrücken nicht deutlich genug sichtbar. Außerdem würde die Gültigkeit der Deklarationen und Definitionen im Prolog gewissermaßen die durch die eckigen Klammern gezogene Grenze überschreiten.

Deswegen scheint es sinnvoller, die Inhalte des Prologs auf die gleiche syntaktische Ebene zu heben wie Variablendeklarationen und Methodendefinitionen in Java. Die Tatsache, dass jeder Bestandteil des Prologs mit einem der Schlüsselwörter `declare`, `default`, `define` oder `import` eingeleitet wird, erleichtert die syntaktische Trennung von Java-Anweisungen und eingestreuten Prolog-Elementen.

5.2 Typisierung der Anfrageergebnisse

Jedes Ergebnis einer XQuery-Anfrage ist eine Sequenz eines gewissen Typs. Beim Arbeiten mit Anfrageergebnissen in Java könnte man jedes Ergebnis als Objekt einer generischen Klasse darstellen, ähnlich wie der Typ ResultSet in JDBC. Allerdings verschenkt man so wertvolle Information. Eine feinere Typisierung der Ergebnisse bietet nämlich verschiedene Vorteile:

- Wenn Anfrageergebnisse zur Weiterverarbeitung an andere Java-Methoden übergeben werden, ermöglichen speziellere Typen eine genauere Definition der Schnittstelle.
- Im Rahmen der statischen Analyse der XQuery-Ausdrücke ist eine bessere Prüfung möglich: Die Typen der Anfrageergebnisse erlauben eine anfrageübergreifende Prüfung bei aufeinander aufbauenden Anfragen.

Wie bei jedem anderen Typsystem liegt also der Nutzen in der Zusicherung zur Compilezeit, dass bestimmte Restriktionen bei der Verwendung von Daten eingehalten werden.

5.2.1 Sequenztypen

Als Typen für Anfrageergebnisse bieten sich die Sequenztypen aus XQuery an, denn diese sind einem Benutzer von XQuery schon bekannt. Variablen, die Anfrageergebnisse aufnehmen, sollten also mit beliebigen Sequenztypen deklariert werden können.

5.2.2 Typprüfung

Die Verträglichkeit zweier Typen, etwa bei einer direkten Zuweisung oder einem Methodenaufruf, ist anhand einer Subtyprelation zu bestimmen, die in der formalen Semantik zu XQuery definiert ist (W3C 2002e). Ein Typ T_1 ist erwartungsgemäß ein Subtyp eines Typs T_2, wenn jeder Wert des Typs T_1 auch ein Wert des Typs T_2 ist.

Wenn zur Übersetzungszeit die beiden zu vergleichenden Typen bekannt sind, kann die Prüfung im Präcompiler mit Algorithmen erfolgen, die sowieso in einer XQuery-Engine implementiert sind. Die gerade genannte Voraussetzung bedarf allerdings genauerer Betrachtung, denn nicht ohne weiteres sind die Typen immer bekannt. Dies ist dann der Fall, wenn bei einer Typprüfung die Grenzen einer

Übersetzungseinheit überschritten werden: Da der Java-Compiler nichts von Sequenztypen weiß, findet nach der Typprüfung im Präcompiler eine irgendwie geartete Abbildung der Anfrageergebnisse auf Java-Klassen statt. In weiteren Läufen des Präcompilers für andere Übersetzungseinheiten stehen also Schnittstellen-Informationen mit Sequenztypen nicht mehr zur Verfügung. Für den Ausweg aus dieser Situation existieren mindestens zwei Varianten:

1. Der Präcompiler sichert Methodensignaturen mit Informationen über Sequenztypen an geeigneter Stelle.
2. Die Sequenztypen werden vom Präcompiler so in Java-Typen überführt, dass der Java-Compiler im Rahmen seiner eigenen Typprüfung unwissend auch die Subtypeigenschaft zweier Sequenztypen überprüft.

Nach einigen Bemerkungen zur ersten untersuchen wir die reizvolle zweite Variante auf ihre Umsetzbarkeit hin.

Zusätzliche Signaturinformationen für den Präcompiler

Der Präcompiler hat die freie Wahl, wo er zusätzliche Informationen über Java-Klassen, nämlich deren Schnittstellen in Bezug auf XQuery-Typen, ablegt. Denkbar sind eine separate Datei pro Klasse, eine Einbettung direkt in die Klassendatei oder eine zentrale Speicherung etwa auf Package-Ebene. Einzige Bedingung ist, dass die Informationen eindeutig der betreffenden Klasse zugeordnet werden können. Außerdem sollte diese Zuordnung von jeder Realisierung des Präcompilers ermittelt werden können; eine Standardisierung ist also angebracht.

Typprüfung durch Java-Compiler

Besonders reizvoll ist die Vorstellung, die XQuery-Typprüfung komplett auf die Java-Typprüfung abbilden zu können.

Um die Subtyprelation durch den unveränderten Java-Compiler nach dem Vorübersetzungsschritt überprüfen zu lassen, ist eine Abbildung aller möglichen Sequenztypen auf Java-Typen, also Klassen, nötig.

In Java wird die Verträglichkeit zweier Typen anhand von Vererbungsbeziehungen ermittelt (das Implementieren eines Interfaces ist dabei eine degenerierte Vererbungsbeziehung); also ist das Ziel, die Subtyprelation unter den Sequenztypen möglichst genau durch Vererbungsbeziehungen in Java zu modellieren.

Ein Sequenztyp besteht aus mehreren Aspekten, die unabhängig voneinander geprüft werden können und gemeinsam die Subtypeigenschaft ergeben. Beispielsweise besteht nach Abschnitt 3.3.4 auf Seite 23 ein Sequenztyp aus einem Item-Typ und einer groben Angabe über die Länge der zum Typ gehörigen Sequenzen. Damit ist ein Sequenztyp *A* ein Untertyp eines Sequenztyps *B*, wenn der Item-Typ von *A* Untertyp des Item-Typs von *B* ist und für die Längen-„Typen" von *A* und *B* entsprechendes gilt. Deswegen sollten diese und eventuell weitere unabhängige Aspekte, in die sie zerlegt werden können, auch in Java unabhängig modelliert werden. Ein Sequenztyp kann also in Java durch eine Menge von Klassen dargestellt werden, die in unterschiedliche Vererbungshierarchien eingebunden sind. Vier dieser Hierarchien lassen sich identifizieren; sie beschreiben folgende Aspekte eines Sequenztyps:

1. Länge der Sequenzen (fünf Werte von „leer" bis „beliebig lang")
2. Grober Item-Typ (Atomarer Wert oder einer von sieben Knotentypen)
3. Notwendiger Schematyp (bei Attributen, Elementen und atomaren Werten)
4. Notwendiger Name (bei Attributen und Elementen)

In Abbildung 5.1 auf der nächsten Seite sind die feststehenden Teile der vier Hierarchien dargestellt sowie die zu benutzerdefinierten Typen zu generierenden Teile angedeutet.

Die Typprüfung auf den Sequenztypen kann dann wie folgt realisiert werden: Alle Zuweisungen von Sequenztypen werden ersetzt durch eine Zuweisung einer Variable generischen Typs (quasi ohne Typprüfung) plus vier Zuweisungen zur Prüfung der verschiedenen Typaspekte, die aber sonst keine Funktion zur Laufzeit erfüllen. Im Java-Quelltext in Listing 5.1 auf Seite 75 ist eine solche Ersetzung einmal prototypisch für verschiedene Szenarien wie Deklaration, Zuweisung oder Funktionsaufruf mit Sequenztypen durchgespielt.

Bewertung

Das eben vorgestellte Vorgehen hat mehrere Nachteile: Für alle benutzerdefinierten Typen müssen Klassen in den vier Vererbungshierarchien generiert werden. Zur Laufzeit werden pro Sequenztypzuweisung vier unnötige Zuweisungen ausgeführt; dies verschlechtert das Leistungsverhalten. Das Vorhaben, die Typprüfung auf den Sequenztypen elegant auf bestehende Funktionalität abzubilden,

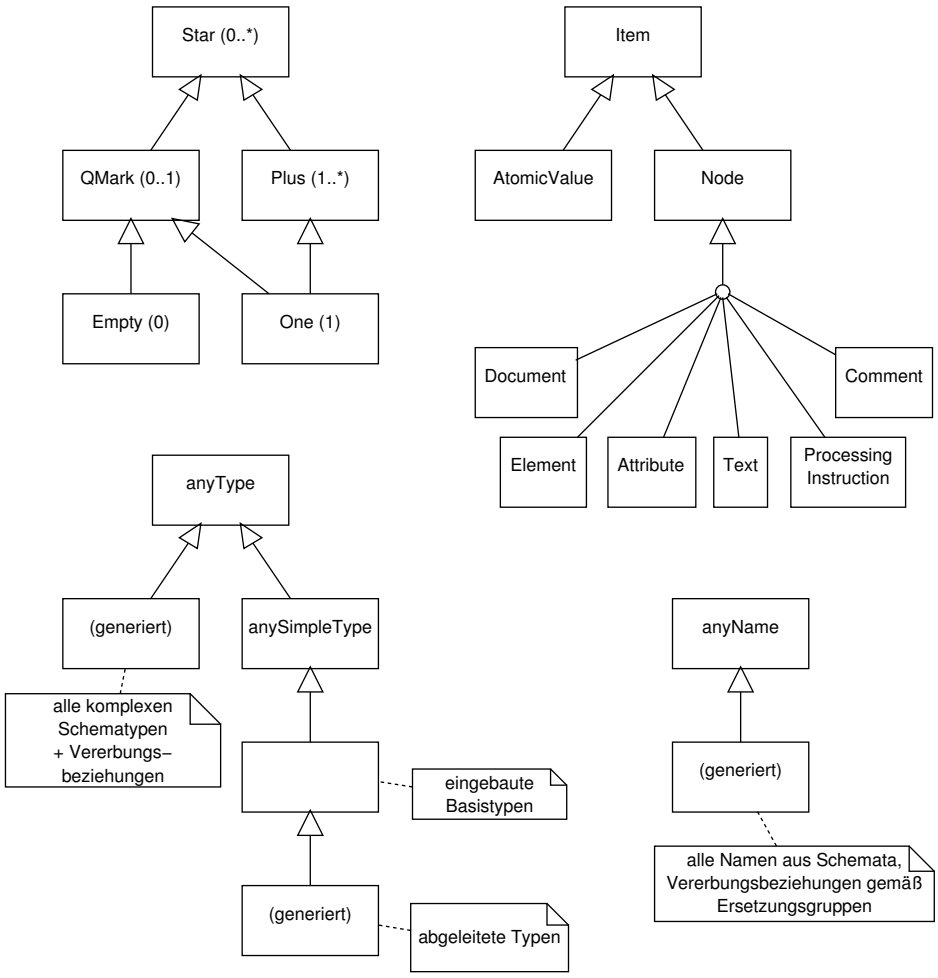


Abbildung 5.1: Vier Aspekte eines Sequenztyps, jeweils durch eine Vererbungshierarchie modelliert.

Listing 5.1: Typprüfung von XQuery-Typen durch Abbildung auf fünf Java-Typen

```
static class MySequenceType extends SequenceType {
    static Plus occ;         static Node itemtype;
    static AnyType reqtype;  static Person reqname;
}
static class MySuperType extends SequenceType {
    static Star occ;         static Item itemtype;
    static AnyType reqtype;  static Name reqname;
}

// function with "one" argument: generic sequence type
// + 4 dummy arguments for type checking
private static void fun(SequenceType t,
    Plus s, Node i, AnyType a, Name n) {}

public static void main(String[] args) {
    // declaration: MySequenceType b, MySuperType c;
    MySequenceType b_typed = new MySequenceType();
    SequenceType b = b_typed;

    MySuperType c_typed = new MySuperType();
    SequenceType c = c_typed;

    // function call: fun(b);
    fun(b, b_typed.occ, b_typed.itemtype,
        b_typed.reqtype, b_typed.reqname);

    // assignments: c = b; b = c;
    c = b;
    c_typed.occ = b_typed.occ;
    c_typed.itemtype = b_typed.itemtype;
    c_typed.reqtype = b_typed.reqtype;
    c_typed.reqname = b_typed.reqname;

    b = c;
    b_typed.occ = c_typed.occ;             // type error
    b_typed.itemtype = c_typed.itemtype;   // type error
    b_typed.reqtype = c_typed.reqtype;
    b_typed.reqname = c_typed.reqname;     // type error
```

bleibt so ein frommer Wunsch. Der Java-Compiler wird für etwas missbraucht, für das er nicht konstruiert wurde.

Sauberer und einfacher ist also die Realisierung der Typprüfung im Präcompiler, indem dieser separat zusätzliche Typinformationen verwaltet.

5.3 Fehlerbehandlung

Während der Behandlung einer XQuery-Anfrage können Fehler auftreten. Sofern diese erkannt werden, sollte es möglich sein, im umgebenden Java-Code darauf zu reagieren.

XQuery unterscheidet seine Fehlerarten nach dem Auftreten in Analysephase oder Auswertungsphase. Während der Analysephase können bei Typfehler im Rahmen der statischen Typprüfung auftreten, außerdem weitere statische Fehler wie beispielsweise Syntaxfehler. Zu den Fehlern in der Auswertungsphase gehören dynamische Typfehler und andere, etwa numerische Überläufe.

Bei einem Einbettungsansatz mit Präcompiler kann die Analysephase für eine XQuery-Anfrage zur Übersetzungszeit stattfinden. Fehler aus dieser Phase verhindern entsprechend die erfolgreiche Übersetzung.

Fehler zur Laufzeit, entweder während der Auswertungsphase oder einer vom Präcompiler in die Laufzeit verschobenen Analysephase, werden sinnvollerweise auf Java-Exceptions abgebildet und so an den umgebenden Java-Code weitergegeben.

5.4 Semantik

Nach der Betrachtung der Syntax und der Typprüfung bleibt die Frage, wie die beschriebene Semantik der aufeinander aufbauenden Anfragen an sich realisiert werden kann. Bei der Beantwortung dieser Frage soll vor allem geklärt werden, inwieweit eine Realisierung der Kopplung auf bestehenden Implementierungen von XQuery aufsetzen kann.

5.4.1 Unterstützung durch die XQuery-Engine

Im Folgenden werden drei verschiedene Szenarien zur Realisierung der Kopplung mit aufeinander aufbauenden Anfragen vorgestellt. Besonderes Augenmerk liegt dabei auf den Anforderungen an eine Anfrageverarbeitungsschicht (AV), auf die

die Realisierungen sinnvollerweise aufsetzen. Ob diese AV dabei nur zur Laufzeit zur Analyse und Ausführung von Anfragen benutzt wird oder ob bestimmte Analysen und Transformationen in die Compilezeit vorgezogen werden, ist weniger relevant.

Die drei ausgewählten Szenarien sind eher als markante Punkte aus dem Raum denkbarer Lösungen zu verstehen und nicht als einzig mögliche Varianten.

Szenario 1 *Aufeinander aufbauende XQuery-Ausdrücke werden quasi auf syntaktischer Ebene zusammengesetzt und im Ganzen analysiert und ausgeführt.*

Das Zusammensetzen kann dabei einfach durch Ersetzen von Variablen, die andere Anfrageergebnisse referenzieren, durch die entsprechenden Anfragen erfolgen. Dies erlaubt die beliebige Komponierbarkeit von XQuery-Ausdrücken, die frei sind von Seiteneffekten. Lediglich Namenskonflikte, z. B. bei im Prolog gebundenen Namensraum-Präfixen, müssen durch Umbenennung aufgelöst werden.

Vorteil dieses Vorgehens ist, dass an die AV keine besonderen Anforderungen bezüglich der Unterstützung aufeinander aufbauender Anfragen gestellt werden. Diese kennt und sieht nur einfache Anfragen.

Ein großer Nachteil ist der Verlust der Struktur der Anfragen, vor allem die Kenntnis gemeinsam benutzter Teile. Gleiche Anfrageteile werden in diesem Szenario immer wieder aufs Neue analysiert und ausgeführt. Die AV kann nicht (oder nur mit großem Aufwand) erkennen, dass Zusammenhänge zwischen verschiedenen Anfragen bestehen. Dabei könnte die explizite Zerlegungs- und Wiederverwendungsstruktur unter den Teilanfragen zur Optimierung ausgenutzt werden.

Bei konsequenter Anwendung des Ersetzungsprinzips für XQuery-Ausdrücke würden selbst für jede einzelne Position in einer Iteration über eine Ergebnissequenz (Abschnitt 4.5.1 auf Seite 63) einzelne Anfragen generiert, die unabhängig voneinander ausgewertet würden. Zumindest für Iterationen sollte also analog zum Cursor-Konzept bei der Verarbeitung von SQL-Ergebnissen ein alternativer Ansatz gewählt werden.

Szenario 2 *Die AV hat eine rudimentäre Unterstützung für Anfragen, die auf den Ergebnissen anderer Anfragen aufbauen. Vor allem existiert ein Mechanismus, um Anfrageergebnisse anderen Anfragen als Eingabe zuzuführen. Im primitivsten Fall könnten dazu die Ergebnisse einer vorgeschalteten Anfrage materialisiert werden.*

Eine Materialisierung von Zwischenergebnissen ist wegen Zeit- und Speicherplatzrestriktionen nur für Zwischenergebnisse bis zu einer bestimmten Größe

praktikabel. Vor allem eine anwendungsbezogene Sichtenbildung (Abschnitt 4.3 auf Seite 57), bei der der Umfang der Daten möglicherweise noch nicht entscheidend reduziert wird (sondern erst in den folgenden Schritten) scheidet damit aus.

Szenario 3 *Die AV bezieht das Konzept der aufeinander aufbauenden Anfragen mit ein und kann solche selbständig verwalten. Sie hat dann die Möglichkeit, über die verzögerte Auswertung oder vorzeitige Materialisierung auch von Teilen von Anfragen zu entscheiden. Im besten Fall werden nur solche Daten berechnet, auf die durch „Blattanfragen" im Anfragegraph tatsächlich zugegriffen wird.*

Diese Lösung ist die flexibelste im Hinblick auf den Einsatz von aufeinander aufbauenden Anfragen zur Sichtenbildung oder auch nur zur besseren Strukturierung von Anfragen. Sie hat das Potential, teure Materialisierungen zu vermeiden, und kann die explizit gegebene Struktur des Anfragegraphen zur Optimierung ausnutzen.

Ein Nachteil besteht darin, dass hier die zusätzlichen für die Kopplung von XQuery und Programmiersprache benötigten Konzepte schon in der Anfrageverarbeitung berücksichtigt sein müssen. Hiermit verschiebt sich die Komplexität in die Realisierung der Anfrageverarbeitung, für die konkrete Kopplung an Java ist dann nicht mehr viel zu leisten. Allerdings ist das Konzept der aufeinander aufbauenden Anfragen so allgemein, dass es zusammen mit seiner Implementierung leicht auch in Anbindungen von XQuery an andere Sprachen wiederverwendet werden kann.

Sicher wäre eine Implementierung des Kopplungsansatzes im Sinne des letzten Szenarios auf lange Sicht die beste Alternative. Nur in Kombination mit einer angepassten Anfrageverarbeitung kann das volle Optimierungspotential der aufeinander aufbauenden Anfragen ausgenutzt werden. Um allerdings erste Erfahrungen mit dem Einsatz der Kopplung zu sammeln, genügt eine weniger aufwändige (weil auf bestehenden XQuery-Implementierungen aufbauende) Implementierung mit Techniken der ersten beiden Szenarien, auch wenn man hier eine Beschränkung der sinnvoll bearbeitbaren Datenvolumina durch Zeit (Szenario 1) oder Speicherplatz (Szenario 2) in Kauf nehmen muss.

Als Bestandteil dieser Arbeit wurde zunächst eine solche prototypische Implementierung angestrebt, allerdings ergaben sich bei der Umsetzung der Materialisierung nach Szenario 2 mit den bestehenden XQuery-Implementierungen grundlegende Probleme, deren Ursachen im folgenden Abschnitt erläutert werden.

5.4.2 Probleme bei der Materialisierung von Zwischenergebnissen

Alle frei verfügbaren Prototyp-Implementierungen von XQuery, die während der Erstellung dieser Arbeit betrachtet wurden, (IPSI 2002; Quip 2002; Galax 2003) haben einen gemeinsamen Mangel: die Darstellung der Ergebnisse einer XQuery-Anfrage und die Möglichkeiten des Zugriffs auf diese. Meist sieht die Benutzung der Implementierungen so aus: Ein XQuery-Ausdruck wird als String übergeben, das Ergebnis wird als XML-Dokument per DOM-Schnittstelle oder im Extremfall in serialisierter Form zurückgegeben. Es bietet sich also an, dieses Ergebnis folgenden Anfragen als Eingabe zuzuführen, um so die Auswertung aufeinander aufbauender Anfragen zu erreichen.

Dass im Allgemeinen das Ergebnis eines XQuery-Ausdrucks kein wohlgeformtes XML-Dokument ist, stört dabei weniger. Die Implementierungen zwingen durch Einführung eines zusätzlichen (herstellerabhängigen) Wurzelelements jedes Ergebnis aus dem Datenmodell in Dokumentform. Das Ergebnis der Auswertung eines XQuery-Ausdrucks ist so bei den bestehenden Implementierungen ein neues Dokument anstelle der eigentlichen Datenmodell-Instanz. Allerdings geht bei dieser Transformation eine wichtige Eigenschaft des Datenmodells verloren. Dieser Verlust sorgt dafür, dass eine korrekte Implementierung von aufeinander aufbauenden Anfragen mit diesen XQuery-Prototypen nicht möglich ist. Welche Eigenschaft verlorengeht, zeigt das folgende Beispiel.

Wir betrachten folgende einfache Struktur zweier Anfragen `a` und `b`, auf deren Ergebnissen eine dritte Anfrage `c` aufbaut:

```
item* a = [document("abc")];
item* b = [document("abc")];
item* c = [$a intersect $b];
```

Offensichtlich sollte folgende einzelne Anfrage bezüglich des Ergebnisses in `c` dazu semantisch äquivalent sein:

```
item* c = [
  let $a := document("abc")
  let $b := document("abc")
  return ($a intersect $b)
];
```

Nehmen wir allerdings an, dass die Anfragen wie oben geschildert realisiert wurden, enthält `c` im zweiten Fall wie erwartet den Dokument-Knoten des Dokuments `abc`, im ersten Fall dagegen die leere Sequenz. Das liegt daran, dass `a` und

`b` jeweils durch *neue* Dokumente mit eigener Identität repräsentiert wurden. Die Schnittmengenbildung basiert aber auf der Knotenidentität und hält `a` und `b` so irrtümlich für disjunkt.

Das grundlegende Problem liegt also im Verlust der Knotenidentität bei der gegenwärtigen Darstellung der Ergebnisse in XQuery-Implementierungen. Es ist mit den verfügbaren Schnittstellen nicht möglich, die Ergebnisse einer Anfrage verlustfrei einer weiteren zuzuführen, obwohl die Abgeschlossenheit bezüglich des Datenmodells gerade eine der Grundideen von XQuery ist.

Generell ist die Knotenidentität eine Herausforderung bei der Arbeit mit dem XQuery-Datenmodell. Zu jedem Knoten muss sozusagen immer festgestellt werden können, woher (z. B. aus welchem Dokument, aus welcher Datenquelle) er stammt. Im Gegensatz dazu hat es das relationale Datenmodell einfacher: Dort gibt es kein Konzept der Identität von Tupeln, so dass bei Materialisierungen wirklich einfache Kopien angelegt werden können, ohne Informationen zu verlieren.

Allerdings hat die Knotenidentität im XQuery-Datenmodell auch einen Vorteil, der Updates betrifft, die zwar noch nicht in der Sprache XQuery berücksichtigt sind, aber es sicher in einer weiteren Version sein werden (Tatarinov u. a. 2001): Bei Änderungen an einem Knoten ist immer klar, welcher Knoten genau betroffen ist, also auch wohin die Änderungen propagiert werden müssen, gegebenenfalls auch durch mehrere Sichten (oder in unserem Fall aufeinander aufbauende Anfragen). Dabei ist dieser Vorteil aber nicht überzubewerten; die Semantik von XQuery enthält Situationen, in denen Kopien von Knoten angelegt werden, also der Bezug zur Herkunft der Daten verlorengeht. Das ist der Fall beim Inhalt von konstruierten Elementen (Abschnitt 3.4.7 auf Seite 39) oder bei der Validierung (Abschnitt 3.4.13 auf Seite 50).

5.4.3 Optimierungen

Der Einbettungsansatz mit Präcompiler erlaubt eine Vorbereitung von auszuführenden Anfragen zur Compilezeit. Betrachtet man eine Anfrage isoliert, so kann diese die Analysephase bei der Verarbeitung von XQuery-Ausdrücken durchlaufen (Abschnitt 3.3.4 auf Seite 23): Die Syntax kann geprüft werden und statische Typprüfungen können erfolgen, auch mit Einbeziehung referenzierter Schemata, wenn man bereit ist, sich an deren Zustand zur Compilezeit zu binden.

Über die einzelne Anfrage hinaus sollte auch der Anfragegraph der aufeinander aufbauenden Anfragen in die Prüfung einbezogen und möglicherweise selbst

optimiert werden. Dazu ist zunächst ein solcher Graph zu ermitteln, der angibt, in welcher Weise sich die einzelnen Anfragen zur Laufzeit aufeinander beziehen können. Eine vollständige Erstellung eines solchen Graphen wird nicht immer möglich sein, sei es, weil sich die beteiligten Anfragen über mehrere Übersetzungseinheiten erstrecken oder weil der Bezug der Anfragen durch beliebig komplexe Operationen auf den ihnen zugeordneten Java-Variablen verschleiert ist.

Liegt aber ein solcher Anfragegraph in Teilen vor, kann er im Großen voroptimiert werden, ohne die Details der Anfragen zu betrachten. Anfragen, die von genau einer anderen Anfrage benutzt werden, können durch Einsetzung mit dieser zu einer Anfrage vereinfacht werden. Solche Anfragen entstehen z. B. dann, wenn der Programmierer das Konzept der aufeinander aufbauenden Anfragen an bestimmten Stellen nur zur Reduzierung der Komplexität einzelner Anfragen einsetzt.

6 Zusammenfassung

Eine Kopplung zweier Sprachen sollte versuchen, die Vorteile beider Sprachen zu verbinden. Seine Flexibilität und Mächtigkeit schöpft XQuery vor allem aus der beliebigen Komponierbarkeit von Ausdrücken. Durch die Reflexion dieser Komponierbarkeit in der Einbettung von XQuery in Java wird diese Mächtigkeit erhalten: Es entsteht ein System von eingebetteten Anfragen, die aufeinander aufbauen.

Aus diesem Konzept der aufeinander aufbauenden Anfragen ergeben sich parametrisierte Anfragen auf natürliche Art und Weise. Anfragen, die von vielen anderen Anfragen benutzt werden, etablieren eine Art anwendungsbezogene Sicht. Weiterhin erlauben aufeinander aufbauende Anfragen eine Zerlegung von komplexen Anfragen in übersichtliche Teile.

Java kennt keine mengenorientierten Operationen, wogegen XQuery hauptsächlich mengen- oder sequenzorientiert arbeitet. Deswegen muss in der Kopplung zusätzlich ein Iterationsmechanismus für Sequenzen vorhanden sein. Allerdings wird der bekannte „Impedance Mismatch“ dadurch abgeschwächt, dass mit Anfrageergebnissen weiterhin mengenorientiert gearbeitet werden kann, nämlich mit Hilfe weiterer XQuery-Ausdrücke. Dies kann soweit fortgeführt werden, bis schließlich atomare Wert direkt als Java-Werte weiterverwendet werden.

Das Konzept der aufeinander aufbauenden Anfragen ließe sich prinzipiell auch durch eine Aufrufschnittstelle realisieren, aber eine Einbettung verspricht einen natürlicheren Umgang mit XQuery in Java sowie bessere Typprüfungen und weitere Analysen und Optimierungen zur Compilezeit.

Momentan verfügbare XQuery-Implementierungen erlauben keine Umsetzung des vorgestellten Konzepts durch Materialisierung von Zwischenergebnissen, da diese nicht verlustfrei weitergegeben werden können. Ursache dafür ist die Identitätseigenschaft der Knoten im XQuery-Datenmodell. Eine Implementierung, die allein auf syntaktischer Substitution beruht, würde nicht tragbare Leistungseinbußen bedeuten. Gefordert ist eine Realisierung, die tiefer in eine Implementierung von XQuery eingreift.

Abschließend ermöglicht die Einbettung von XQuery in Java eine Art und Weise der Anwendungsprogrammierung mit dem Zugriff auf XML-Daten, die keine

Abhängigkeiten durch herstellerspezifische Initialisierung- und Zugriffsmethoden enthält, sondern sich allein auf den kommenden Standard XQuery stützt. Jede Bindung an die Systemumgebung kann im Präcompiler oder später vorgenommen werden. Es bleibt deshalb allein zu hoffen, dass nicht wie bei SQL Dutzende von nur teilweise kompatiblen XQuery-Dialekten entstehen werden.

6.1 Ausblick

Rund um die Einbettung von XQuery in Java und das Konzept der aufeinander aufbauenden Anfragen gibt es noch eine Reihe von Betätigungsfeldern für die Zukunft:

- Implementierung der Einbettung durch einen Präprozessor, der die eingebetteten XQuery-Ausdrücke auf ein geeignetes Laufzeitsystem abbildet und die statische Typprüfung durchführt.
- Als Voraussetzung dazu ist ein System zu konstruieren, das in der Lage ist, aufeinander aufbauende Anfragen auszuwerten. Dabei sind mehrere Schritte in steigendem Schwierigkeitsgrad möglich:
  - Finden einer geeigneten Repräsentation zur Materialisierung von Anfrageergebnissen, also von Datenmodell-Instanzen, so dass alle wichtigen Eigenschaften wie die Knotenidentität erhalten bleiben. Damit ist eine prototypische Realisierung der aufeinander aufbauenden Anfragen durch Materialisierung nach jedem Schritt möglich.
  - Vermeidung aufwändiger Materialisierungen durch Substitution von Variablen durch referenzierte Anfragen. Sinnvollerweise muss im Zusammenhang damit die Entwicklung von Modellen stehen, die entscheiden, wann oder an welchen Stellen im Anfragegraph sich eine Materialisierung lohnt, wann eine Substitution auf Anfrageebene.

    Außerdem kann untersucht werden, ob es sinnvolle Zwischenformen gibt, etwa eine unvollständige Materialisierung nur bis zu einer bestimmten Tiefe oder Breite der Ergebnisbäume oder auch die Heraustrennung zu materialisierender Ausdrücke aus Anfragen (Ausnutzung des Kompositionsprinzips in anderer Richtung).
  - Die vorhergehenden Schritte führen zur Einbeziehung der Anfrageoptimierung in die Auswertung des Anfragegraphen im Gegensatz zur

Auswertung einzelner Anfragen. Es werden dann also nicht mehr nur einzelne Anfragen lokal optimiert. Alles deutet darauf hin, dass nur eine Anfrageverarbeitung für XQuery, die unter Kenntnis des Konzepts der aufeinander aufbauenden Anfragen entwickelt wurde, solche effizient verarbeiten kann.

- Entwicklung eines Konzepts, wie in (eingebettetem) XQuery zur Identifizierung von Dokumenten oder Kollektionen benutzte URIs außerhalb der Anwendung in die richtigen Datenquellen aufgelöst werden. Innerhalb eines XQuery-Java-Programms sollten nie hersteller-, orts- oder zeitabhängige URIs benutzt werden, sondern immer nur logische Namen. Von früher Bindung durch den Präcompiler bis zur später Bindung zur Laufzeit ist dabei alles möglich.

- Mit dem vorigen Punkt hängt die Frage zusammen, wie Anfragen anhand der betroffenen Dokumente auf verschiedene Datenquellen und ihre angeschlossenen XQuery-Prozessoren verteilt werden. Daran schließt sich die Aufgabe an, über mehrere Datenquellen verteilte Anfragen auszuwerten. Es ist wünschenswert, dass der Anwendungsprogrammierer nur eine einzige XQuery-Verarbeitungsinstanz wahrnimmt, die ihm alle seine Anfragen beantworten kann.

- Beobachtung der Entwicklung von Änderungsoperationen in XQuery. Für eine zukünftige Version der Sprache ist dann zu untersuchen, wie Änderungen durch eine Folge von Abfragen im Anfragegraph hindurch zurück zur Datenquelle propagiert werden können. Reicht die Knotenidentität dafür aus und sollen Änderungen auf umstrukturierten Daten verboten werden? Außerdem wird dann eine Erweiterung der Einbettung nötig, um etwa Anfragen an Transaktionskontexte binden zu können.

- Untersuchung der Integration weiterer Sichten auf die Daten in die Einbettung (Abschnitt 4.4.2 auf Seite 62), etwa in Form direkter Verfügbarkeit von DOM-Methoden an Anfrageergebnissen oder in Form der Überführung der Anfrageergebnisse in einen Data-Binding-Objektgraph per einfacher Zuweisung. Dabei steht die interessante Frage im Raum, inwieweit eine Rückkehr zur XQuery-Datenmodell-Sicht möglich ist, so dass nach Operationen auf den anderen Sichten weitere XQuery-Anfragen möglich sind.

Literaturverzeichnis

Castor 2003 CASTOR PROJECT (Hrsg.): *Using Castor XML*. 2003. – URL `http://castor.exolab.org/xml-framework.html`. – Zugriffsdatum: 2003-04-09

Chamberlin 2002 CHAMBERLIN, D.: XQuery: An XML query language. In: *IBM Systems Journal* 41 (2002), Nr. 4, S. 597–615. – URL `http://www.research.ibm.com/journal/sj/414/chamberlin.pdf`. – Basiert auf einem älteren Entwurf der XQuery-Spezifikation (z. B. gibt es noch einen `sortby`-Operator, dafür kein `order by` in FLWR); nichtsdestotrotz eine schöne Einführung.

Chase 2002 CHASE, Nicholas: *Process XML using XML Query*. Tutorial von IBM developerWorks. September 2002. – URL `http://www.ibm.com/developerworks/xml/edu/x-dw-xxquery-i.html`

Florescu u. a. 2001 FLORESCU, Daniela ; GRÜNHAGEN, Andreas ; KOSSMANN, Donald: XL: An XML Programming Language / INRIA/TU München. URL `http://xl.in.tum.de/publ/techrep01_XL.pdf`, Dezember 2001. – Forschungsbericht

Florescu u. a. 2002 FLORESCU, Daniela ; GRÜNHAGEN, Andreas ; KOSSMANN, Donald: XL: An XML Programming Language for Web Service Specification and Composition. In: *Proceedings of the eleventh international conference on World Wide Web*, ACM Press, 2002, S. 65–76. – URL `http://xl.in.tum.de/publ/www2002.pdf`. – ISBN 1-58113-449-5

Galax 2003 BELL LABS: *Galax: An Implementation of XQuery*. Januar 2003. – URL `http://db.bell-labs.com/galax/`. – Implementiert in O'Caml; aktuelle Version: 0.3.0

Gosling u. a. 2000 GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *The Java Language Specification, Second Edition*. Addison-Wesley, 2000. – URL `http://java.sun.com/docs/books/jls/`. – ISBN 0-201-31008-2

Härder u. a. 2000 HÄRDER, Theo ; NINK, Udo ; RITTER, Norbert: Generierte DB-Aufrufschnittstellen – Anwendungsspezifische Zugriffsoptimierung durch Bindungsflexibilität. In: *Informatik Forschung und Entwicklung* 15 (2000), S. 67–82. – ISSN 0178-3564

IPSI 2002 FRAUNHOFER IPSI: *IPSI-XQ – Der XQuery Demonstrator*. Dezember 2002. – URL `http://www.ipsi.fraunhofer.de/oasys/projects/ipsi-xq/index_d.html`. – Implementiert in Java; aktuelle Version: 1.2.2

ISO 2002a MELTON, Jim (Hrsg.): Information technology – Database languages – SQL – Part 10: Object Language Bindings (SQL/OLB) / ISO-ANSI. 9. August 2002 (ISO/IEC 9075-13:2002(E)). – Working Draft

ISO 2002b MELTON, Jim (Hrsg.): Information technology – Database languages – SQL – Part 13: SQL Routines and Types Using the Java Programming Language (SQL/JRT) / ISO-ANSI. 11. März 2002 (ISO/IEC 9075-10:2000(E)). – Working Draft

JAXB 2003 SUN MICROSYSTEMS (Hrsg.): *Java Architecture for XML Binding (JAXB)*. 2003. – URL `http://java.sun.com/xml/jaxb/`. – Zugriffsdatum: 2003-04-09

Quip 2002 SOFTWARE AG: *XQuery prototype Quip*. Juni 2002. – URL `http://developer.softwareag.com/tamino/quip/`. – Implementiert in Haskell; aktuelle Version: 2.2.1

Saake und Sattler 2000 SAAKE, Gunter ; SATTLER, Kai-Uwe: *Datenbanken & Java: JDBC, SQLJ und ODMG*. 1. Aufl. Heidelberg : dpunkt, 2000. – ISBN 3-932588-54-1

SAX 2002 SAX PROJECT (Hrsg.): *SAX – Simple API for XML*. 2002. – URL `http://www.saxproject.org/`. – Zugriffsdatum: 2002-12-13

Sosnoski 2002 SOSNOSKI, Dennis M.: *XML and Java technologies: Data binding with Castor*. April 2002. – URL `http://www.ibm.com/developerworks/xml/library/x-bindcastor/`

Steinberg und Thinking 2002 STEINBERG, Daniel ; THINKING, Dim S.: *Data binding with JAXB*. Tutorial von IBM developerWorks. Januar 2002. – URL `http://www.ibm.com/developerworks/xml/edu/x-dw-xjaxb-i.html`

Tatarinov u. a. 2001 TATARINOV, Igor ; IVES, Zachary G. ; HELEVY, Alon Y. ; WELD, Daniel S.: Updating XML. In: *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, ACM Press, 2001, S. 413–424. – URL `http://data.cs.washington.edu/papers/updatingXML.pdf`. – ISBN 1-58113-332-4

W3C 1999a BRAY, Tim (Hrsg.) ; HOLLANDER, Dave (Hrsg.) ; LAYMAN, Andrew (Hrsg.): Namespaces in XML. World Wide Web Consortium, 14. Januar 1999. – Recommendation. – URL `http://www.w3.org/TR/1999/REC-xml-names-19990114`

W3C 1999b CLARK, James (Hrsg.) ; DEROSE, Steve (Hrsg.): XML Path Language (XPath) Version 1.0. World Wide Web Consortium, 16. November 1999. – Recommendation. – URL `http://www.w3.org/TR/1999/REC-xpath-19991116`

W3C 2000a LE HORS, Arnaud (Hrsg.) ; LE HÉGARE, Philippe (Hrsg.) ; WOOD, Lauren (Hrsg.) ; NICOL, Gavin (Hrsg.) ; ROBIE, Jonathan (Hrsg.) ; CHAMPION, Mike (Hrsg.) ; BYRNE, Steve (Hrsg.): Document Object Model (DOM) Level 2 Core Specification. World Wide Web Consortium, 13. November 2000. – Recommendation. – URL `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113`

W3C 2000b BRAY, Tim (Hrsg.) ; PAOLI, Jean (Hrsg.) ; SPERBERG-MCQUEEN, C. M. (Hrsg.) ; MALER, Eve (Hrsg.): Extensible Markup Language (XML) 1.0 (Second Edition). World Wide Web Consortium, 6. Oktober 2000. – Recommendation. – URL `http://www.w3.org/TR/2000/REC-xml-20001006`

W3C 2001a COWAN, John (Hrsg.) ; TOBIN, Richard (Hrsg.): XML Information Set. World Wide Web Consortium, 24. Oktober 2001. –

Recommendation. – URL http://www.w3.org/TR/2001/REC-xml-infoset-20011024

W3C 2001b CHAMBERLIN, Don (Hrsg.) ; FANKHAUSER, Peter (Hrsg.) ; MARCHIORI, Massimo (Hrsg.) ; ROBIE, Jonathan (Hrsg.): XML Query Requirements. World Wide Web Consortium, 15. Februar 2001. – Working Draft. – URL http://www.w3.org/TR/2001/WD-xmlquery-req-20010215

W3C 2001c FALLSIDE, David C. (Hrsg.): XML Schema Part 0 : Primer. World Wide Web Consortium, 2. Mai 2001. – Recommendation. – URL http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/

W3C 2001d THOMPSON, Henry S. (Hrsg.) ; BEECH, David (Hrsg.) ; MALONEY, Murray (Hrsg.) ; MENDELSOHN, Noah (Hrsg.): XML Schema Part 1 : Structures. World Wide Web Consortium, 2. Mai 2001. – Recommendation. – URL http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/

W3C 2001e BIRON, Paul V. (Hrsg.) ; MALHOTRA, Ashok (Hrsg.): XML Schema Part 2 : Datatypes. World Wide Web Consortium, 2. Mai 2001. – Recommendation. – URL http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

W3C 2001f MALHOTRA, Ashok (Hrsg.) ; ROBIE, Jonathan (Hrsg.) ; RYS, Michael (Hrsg.): XML Syntax for XQuery 1.0 (XQueryX). World Wide Web Consortium, 17. Juni 2001. – Working Draft. – URL http://www.w3.org/TR/2001/WD-xqueryx-20010607

W3C 2001g MUENCH, Steve (Hrsg.) ; SCARDINA, Mark (Hrsg.) ; FERNANDEZ, Mary (Hrsg.): XPath Requirements Version 2.0. World Wide Web Consortium, 14. Februar 2001. – Working Draft. – URL http://www.w3.org/TR/2001/WD-xpath20req-20010214

W3C 2002a BERGLUND, Anders (Hrsg.) ; BOAG, Scott (Hrsg.) ; CHAMBERLIN, Don (Hrsg.) ; FERNANDEZ, Mary F. (Hrsg.) ; KAY, Michael (Hrsg.) ; ROBIE, Jonathan (Hrsg.) ; SIMÉON, Jérôme (Hrsg.): XML Path Language (XPath) 2.0. World Wide Web Consortium, 15. November 2002. – Working Draft. – URL http://www.w3.org/TR/2002/WD-xpath20-20021115

W3C 2002b CHAMBERLIN, Don (Hrsg.) ; FANKHAUSER, Peter (Hrsg.) ; FLORESCU, Daniela (Hrsg.) ; MARCHIORI, Massimo (Hrsg.) ; ROBIE, Jonathan (Hrsg.): XML Query Use Cases. World Wide Web Consortium, 15. November 2002. – Working Draft. – URL `http://www.w3.org/TR/2002/WD-xmlquery-use-cases-20021115/`

W3C 2002c BOAG, Scott (Hrsg.) ; CHAMBERLIN, Don (Hrsg.) ; FERNANDEZ, Mary F. (Hrsg.) ; FLORESCU, Daniela (Hrsg.) ; ROBIE, Jonathan (Hrsg.) ; SIMÉON, Jérôme (Hrsg.): XQuery 1.0: An XML Query Language. World Wide Web Consortium, 15. November 2002. – Working Draft. – URL `http://www.w3.org/TR/2002/WD-xquery-20021115/`

W3C 2002d FERNÁNDEZ, Mary (Hrsg.) ; MALHOTRA, Ashok (Hrsg.) ; MARSH, Jonathan (Hrsg.) ; NAGY, Marton (Hrsg.) ; WALSH, Norman (Hrsg.): XQuery 1.0 and XPath 2.0 Data Model. World Wide Web Consortium, 15. November 2002. – Working Draft. – URL `http://www.w3.org/TR/2002/WD-query-datamodel-20021115/`

W3C 2002e DRAPER, Denise (Hrsg.) ; FANKHAUSER, Peter (Hrsg.) ; FERNÁNDEZ, Mary (Hrsg.) ; MALHOTRA, Ashok (Hrsg.) ; ROSE, Kristoffer (Hrsg.) ; RYS, Michael (Hrsg.) ; SIMÉON, Jérôme (Hrsg.) ; WADLER, Philip (Hrsg.): XQuery 1.0 and XPath 2.0 Formal Semantics. World Wide Web Consortium, 15. November 2002. – Working Draft. – URL `http://www.w3.org/TR/2002/WD-query-semantics-20021115/`

W3C 2002f MALHOTRA, Ashok (Hrsg.) ; MELTON, Jim (Hrsg.) ; ROBIE, Jonathan (Hrsg.) ; WALSH, Norman (Hrsg.): XQuery 1.0 and XPath 2.0 Functions and Operators. World Wide Web Consortium, 15. November 2002. – Working Draft. – URL `http://www.w3.org/TR/2002/WD-xquery-operators-20021115/`

W3C 2003 LE HORS, Arnaud (Hrsg.) ; LE HÉGARE, Philippe (Hrsg.) ; WOOD, Lauren (Hrsg.) ; NICOL, Gavin (Hrsg.) ; ROBIE, Jonathan (Hrsg.) ; CHAMPION, Mike (Hrsg.) ; BYRNE, Steve (Hrsg.): Document Object Model (DOM) Level 3 Core Specification. World Wide Web Consortium, 26. Februar 2003. – Working Draft. – URL `http://www.w3.org/TR/2003/WD-DOM-Level-3-Core-20030226`

Syntaxdiagramme