Fachhochschule Wedel (University of Applied Sciences Wedel)

Diplomarbeit im Fachbereich Wirtschaftsinformatik

Konzeption und Implementation eines XPath-Moduls für die Haskell XML Toolbox

20.02.2003

Eingereicht von: Torben Kuseler
Kroonhorst 80
D-22549 Hamburg
EMail: torben@kuseler.de

Betreuer: Prof. Dr. Uwe Schmidt
Fachhochschule Wedel
Feldstrasse 143
D-22880 Wedel
EMail: si@fh-wedel.de

dedicated to all friends who helped me in times of need...

Inhaltsverzeichnis

Inhaltsverzeichnis 3

Abbildungsverzeichnis 5

Tabellenverzeichnis 6

1 Einleitung 7

2 Funktionale Programmierung 9
- 2.1 Programmierparadigma . . . 9
- 2.2 Funktionen . . . 9
  - 2.2.1 Funktionen höherer Ordnung . . . 9
  - 2.2.2 Funktionskomposition . . . 10
  - 2.2.3 Kombinatoren . . . 10
- 2.3 Typen . . . 10
  - 2.3.1 Basis-Typen, Listen und Tupel . . . 10
  - 2.3.2 Algebraische Typen . . . 11
  - 2.3.3 Typ-Synonyme . . . 11
- 2.4 Pattern Matching . . . 11
- 2.5 Guards . . . 12

3 Das XPath-Datenmodell 13
- 3.1 Repräsentation eines XML-Dokumentes . . . 13
  - 3.1.1 Knotentypen . . . 14
  - 3.1.2 Dokumentordnung . . . 16
- 3.2 Parsen von Ausdrücken . . . 16
- 3.3 Boolsche und arithmetische Ausdrücke . . . 17
- 3.4 Zahlen und Zeichenketten . . . 19
- 3.5 Variablen und Funktionen . . . 20
- 3.6 Lokalisierungspfade . . . 21
  - 3.6.1 Achsen . . . 21
  - 3.6.2 Knotentests . . . 25
  - 3.6.3 Prädikate . . . 26
  - 3.6.4 Filter-Ausdrücke . . . 26
  - 3.6.5 Relative und absolute Lokalisierungspfade . . . 26
  - 3.6.6 Abkürzungen . . . 27
- 3.7 Darstellung eines geparsten Ausdruckes . . . 28

4 Auswertung eines Ausdruckes 30
- 4.1 Kontext . . . 31
  - 4.1.1 Variablen . . . 32
  - 4.1.2 Funktionsbibliothek . . . 32
  - 4.1.3 Funktionsverarbeitung . . . 34
  - 4.1.4 Namensraumdeklaration . . . 36

4.2 XPathFilter . . . 36
4.3 Boolsche Ausdrücke . . . 38
4.4 Numerische Ausdrücke . . . 38
4.5 Unärer Ausdruck . . . 39
4.6 Relationale Ausdrücke . . . 39
4.6.1 Knotenmengen Vergleiche . . . 39
4.6.2 Numerische Vergleiche . . . 40
4.7 Lokalisierungspfade . . . 40
4.7.1 Navigierbare Bäume . . . 41
4.7.2 Relativer und absoluter Pfad . . . 42
4.7.3 Lokalisierungsschritte . . . 42
4.7.4 Knotentests . . . 43
4.7.5 Prädikate . . . 43
4.7.6 Entfernen doppelter Teilbäume . . . 44
4.8 Filter-Ausdrücke und Mengenvereinigung . . . 45
4.9 Darstellung eines Ergebnisses . . . 46
4.10 Modul-Hierarchie . . . 46

5 Schlussbetrachtung **47**
5.1 Erreichtes . . . 47
5.2 Konformität . . . 47
5.3 Ausblick . . . 48

Literaturverzeichnis **49**

Weitere Quellen **50**

Eidesstattliche Erklärung **51**

Abbildungsverzeichnis

3.1 XML-Dokument als Baum-Repräsentation 14
3.2 Dokumentordnung 16
3.3 ancestor-Achse 22
3.4 ancestor-or-self-Achse 22
3.5 child-Achse 22
3.6 parent-Achse 22
3.7 descendant-Achse 23
3.8 descendant-or-self-Achse 23
3.9 following-sibling-Achse 23
3.10 following-Achse 23
3.11 preceding-sibling-Achse 23
3.12 preceding-Achse 23
3.13 self-Achse 24
3.14 Zusammenwirken der Achsen 24

4.1 Doppelte Teilbäume 45
4.2 Modul-Hierarchie 46

Tabellenverzeichnis

3.1 Boolsche und arithmetische Operatoren 17
3.2 Zahlenwerte im IEEE-754-Standard 19
3.3 XPath-Achsen 22
3.4 XPath-Abkürzungen 27

4.1 Ergebnis-Grundtypen einer Auswertung 30
4.2 Auswertungsfunktionen für Teilausdrücke 37

1 Einleitung

Die Sprache XPath (XML Path Language) dient zur Adressierung einzelner Teile eines XML-Dokumentes. XPath bildet damit die Grundlage für XSLT und XPointer [W3C 99].

Im Umfeld einer ständig wachsenden Popularität von XML ist das Selektieren bestimmter Elemente eines XML-Dokumentes eine der zentralen Aufgaben. Das World Wide Web Consortium (W3C) spezifizierte hierfür im November 1999 die XML Path Language (XPath), Version 1.0.

Der Entwurf einer eigenen Empfehlung (W3C Recommendation) für das Adressieren von Dokumentenbestandteilen resultiert daraus, dass sowohl für XSLT als auch für XPointer ein Adressierungsmechanismus gebraucht wird, der in weiten Teilen identisch ist.
XPointer dienen einerseits zur Vernetzung unterschiedlicher Bereiche desselben XML-Dokumentes, andererseits können mit ihnen auch Verweise zwischen verschiedenen XML-Dokumenten erstellt werden. XSLT verwendet den XPath-Adressierungsmechanismus, um eine Teilmenge des XML-Dokumentes für eine Weiterverarbeitung (zur Transformation) auszuwählen.

Neben der Adressierung von Teilen eines Dokumentes ermöglicht XPath den Test eines Elementes auf ein bestimmtes Muster. Diese natürliche Teilmenge von XPath wird Pattern Matching genannt und hauptsächlich von XSLT verwendet. Außerdem stellt XPath Möglichkeiten der Manipulation von Zeichenketten und der Berechnung einfacher arithmetischer und boolscher Ausdrücke zur Verfügung.

Diese Arbeit bezieht sich auf die kommentierte deutsche Übersetzung der W3C-Spezifikation für XPath von Oliver Becker [Becker 02]. Bei nicht eindeutigen Abschnitten der Übersetzung wird die normative englische Version [W3C 99] als einzig gültige Spezifikation von XPath verwendet.
Der aktuelle W3C Working Draft für die Spezifikation der XML Path Language, Version 2.0 wird nicht Gegenstand dieser Arbeit sein und im Weiteren auch nicht näher betrachtet.

Die Haskell XML Toolbox [Toolbox 02] ist eine Sammlung von Werkzeugen für die Verarbeitung von XML-Dokumenten. Die Toolbox ist in Haskell geschrieben und wurde an der Fachhochschule Wedel von Prof. Dr. Uwe Schmidt ins Leben gerufen.
Sie umfasst in der aktuellen Version einen Parser sowie ein Modul zum Validieren von XML-Dokumenten. Ein Programm zum Ableiten von Java-Klassen aus DTDs wird zurzeit von einem Studenten der Fachhochschule Wedel entwickelt.

Mit der Haskell XML Toolbox ist es derzeit möglich, ein XML-Dokument einzulesen und auf seine Gültigkeit zu prüfen. Ebenso können einzelne XML-Elemente durch die Haskell XML Toolbox verändert oder gelöscht werden. Das bearbeitete XML-Dokument kann abschließend wieder ausgegeben und gesichert werden. Die Rückgabe einer ausgewählten Menge von Elementen eines Dokumentes ist hingegen nicht möglich.

Das Ziel dieser Arbeit besteht in der Analyse der grundlegenden Arbeitsweise von XPath sowie in der Umsetzung der Spezifikation in ein XPath-Modul für die Haskell XML Toolbox. Das Design und die Implementation des Moduls erfolgt dabei auf der Grundlage der Datenstrukturen und Verarbeitungstechniken der Haskell XML Toolbox.

In Kapitel zwei wird zunächst auf die funktionale Programmiersprache Haskell eingegangen. Es werden kurz die wichtigsten Paradigmen der funktionalen Programmierung dargelegt, und einige Spracheigenschaften von Haskell erläutert. Dieses Kapitel legt die Grundlagen für das Verständnis der weiteren Abschnitte. Alle verwendeten Datenstrukturen und Beispiele des XPath-Moduls werden im Folgenden in der Sprache Haskell dargestellt.

Das dritte Kapitel dient zur Analyse der XPath-Spezifikation. Parallel zur Betrachtung der verschiedenen Elemente von XPath wird ein Parser für die Grammatik entwickelt. Die einzelnen Regeln der Grammatik werden hierfür schrittweise in ein Datenmodell überführt, das in der Lage ist, einen beliebigen XPath-Ausdruck abzubilden.

Kapitel vier beschreibt die Umsetzung eines geparsten Ausdruckes in ein XPath-Ergebnis. Es wird hierbei herausgearbeitet, wie die verschiedenen Teilausdrücke von XPath bezüglich eines XML-Dokumentes durch das Modul ausgewertet werden. Die zu berücksichtigen Randbedingungen der XPath-Spezifikation sind ebenso Inhalt dieses Kapitels, wie die Entwicklung eines Datenmodells für die unterschiedlichen Ergebnistypen eines Ausdruckes.

Das letzte Kapitel beinhaltet die Schlussbetrachtungen dieser Arbeit. Es wird dabei auf die erreichten Ziele und die auf dem Weg zu überwindenden Probleme eingegangen. Ein weiterer Abschnitt beschäftigt sich mit der Konformität von XPath zu umgebenen Prozessoren. Abschließend werden mögliche Einsatzgebiete und Erweiterungen, die sich aus dieser Arbeit ergeben, diskutiert.

2 Funktionale Programmierung

Dieses Kapitel beschreibt die wichtigsten Eigenschaften einer funktionalen Programmiersprache am Beispiel von Haskell [Haskell 02]. Es soll dabei keine vollständige Sprachreferenz gebildet, sondern nur die Basis für das Verständnis der folgenden Kapitel gelegt werden. Für weitere Informationen wird das Buch "Haskell" von Simon Thompson [Thompson 99] empfohlen. Leser, die mit den Methoden der funktionalen Programmierung vertraut sind, können dieses Kapitel überspringen.

2.1 Programmierparadigma

Im Gegensatz zu imperativen Programmiersprachen, wie beispielsweise der Sprache C oder Pascal, besitzen funktionale Sprachen keine Zuweisungen, Schleifen oder Prozeduren und somit auch keine Zustandstransformationen. Der ermittelte Wert eines Ausdruckes wird nicht in einer Speicherzelle abgelegt (Variablenzuweisung), sondern dient direkt als Eingabe für eine folgende Funktion. Bei funktionalen Sprachen steht im Mittelpunkt der Betrachtung *WAS* berechnet werden soll und nicht *WIE* die Berechnung erfolgt.

2.2 Funktionen

Funktionen und ihre Definition sind der zentrale Aspekt der funktionalen Programmierung. Eine formale Funktionsdefinition besteht aus einer Typdefinition und einem Funktionskörper. Der Typ einer Funktion wird vollständig statisch (zur Übersetzungszeit) bestimmt. Die Auswertung des Funktionskörpers bildet den Rückgabewert der Funktion.

Beispiel: (Funktionsdefinition ohne Argumente)

```
aNumber :: Int              (Typdefinition)
aNumber = 6 * (9 - 2)       (Funktionskörper)
```

Eine Typdefinition kann entweder aus einem einzigen Typ (dem Rückgabetyp der Funktion) oder einer beliebigen Anzahl von Argumenttypen, gefolgt vom Rückgabetyp, bestehen.

Beispiel: (Funktionsdefinition mit einem Argument)

```
doubleInteger :: Int → Int
doubleInteger arg = 2 * arg
```

2.2.1 Funktionen höherer Ordnung

Haskell ermöglicht als Typen nicht nur einfache Basis-Typen (Int, Char, usw.), sondern auch Funktionen zu verwenden. Kommt eine Funktion als Argument bzw. als Rückgabetyp vor, so spricht man von einer "Funktion höherer Ordnung".

Beispiel: (Funktionsdefinition höherer Ordnung)

```
twice :: (Int → Int) → Int → Int
twice fct arg = fct ( fct arg )
```

Die Funktion *fct* wird zweimal auf das Argument *arg* angewendet:
```
twice doubleIntegers 4
⇝ doubleIntegers (doubleIntegers 4)
⇝ doubleIntegers 8
⇝ 16
```

2.2.2 Funktionskomposition

Das Zusammensetzen von Funktionen, die Funktionskomposition, erlaubt eine sehr kompakte Notation schwieriger Algorithmen. Sie existiert nicht in klassischen imperativen Programmiersprachen.
Eine Funktion f berechnet aus einer Menge von Eingaben einen Rückgabewert, der als Eingabe für eine zweite Funktion g verwendet wird. Dies entspricht dem mathematischen Ausdruck $g \circ f$ und wird in Haskell als $g.f$ notiert. Die Komposition kann beliebig mit weiteren Funktionen fortgesetzt werden.

2.2.3 Kombinatoren

Kombinatoren nutzen die Eigenschaft der Funktionskomposition um einfache, elementare Funktionen zu komplexeren zusammenzusetzen. Der Rückgabetyp eines Kombinators ist eine Funktion.

Beispiel: (Definition eines Kombinators)
```
newFct :: (Int → Int) → (Int → Int) → (Int → Int)
newFct f g = (g.f)
```

2.3 Typen

Haskell unterscheidet eine Reihe von Typen:

- Basis-Typen
- Listen und Tupel
- Algebraische Typen
- Abstrakte Typen
- Typ Klassen

Die folgenden Abschnitte geben einen kurzen Überblick über die für diese Arbeit wichtigsten Typen.

2.3.1 Basis-Typen, Listen und Tupel

Zu den einfachen Basis-Typen zählen Int, Float, Bool und Char.
Listen und Tupel setzen Typen zu umfangreicheren Strukturen zusammen. Die Elemente einer Liste sind dabei immer vom selben Typ. Tupel ermöglichen das Zusammenfassen unterschiedlicher Typen.

Beispiel: (Definition und Initialisierung einer Liste und eines Tupels)
```
myList :: [Int]
myList = [1,21,42]
```

```
myTupel :: (Int, Char, Bool)
myTupel = (42, 'c', True)
```

2.3.2 Algebraische Typen

Für die Definition komplexerer Datenstrukturen als Listen und Tupel stellt Haskell die algebraischen Typen zur Verfügung. Der senkrechte Strich | trennt die verschiedenen Optionen, die für den Typ möglich sind.

Beispiel: (Typdefinition eines Aufzählungstypen)
```
data Season = Spring | Summer | Autumn | Winter
```

Ein algebraischer Typ kann auch rekursiv definiert werden und eignet sich so beispielsweise für eine Baumstruktur.

Beispiel: (Typdefinition eines binären Baumes)
```
data BinTree Int = Leaf Int | Branch (BinTree Int) (BinTree Int)
```

Die Definition beginnt mit dem Schlüsselwort *data*, gefolgt von dem Namen des Typen und den Typkonstruktoren. Algebraische Typen, die neben ihren Typkonstruktoren noch weitere Komponenten besitzen, folglich keine Aufzählungstypen sind, werden auch Produkttypen genannt.
Der Baum (ein Produkttyp) kann entweder ein Blatt oder ein Ast sein, wobei der Ast wiederum zwei (Teil-)Bäume enthält. BinTree speichert als Information in den Blättern allerdings nur Werte vom Typ *Int*. Das Speichern von Zeichen ist in diesem Baum nicht möglich.

Polymorphismus

Um beliebige Typen speichern und einheitlich verarbeiten zu können, ist die Definition einer polymorphen Baumstruktur notwendig.

Beispiel: (Typdefinition eines binären polymorphen Baumes)
```
data BinTree1 a = Leaf a | Branch (BinTree1 a) (BinTree1 a)
```

BinTree1 kann als Information in den Blättern einen beliebigen Typ *a* enthalten. Listen (siehe Abschnitt 2.3.1) sind ebenfalls polymorph, so dass alle über sie definierten Funktionen für jede Art von Listen gültig sind.

2.3.3 Typ-Synonyme

Typ-Synonyme erhöhen die Lesbarkeit und somit die Wartbarkeit eines Haskell Programms. Ein Synonym ist kein neuer Typ, sondern gibt einem bestehenden einen neuen Namen.

Beispiel: (String ist ein Synonym für eine Liste von Zeichen)
```
type String = [Char]
```

2.4 Pattern Matching

Funktionen können viele unterschiedliche Werte als aktuelle Parameter besitzen. Das Pattern Matching ermöglicht eine sehr mächtige Art der Verarbeitung der verschiedenen Fälle. Eine

Funktion wird hierbei mehrfach deklariert, wobei jede Deklaration einen anderen Fall der Eingabewerte abdeckt. Die erste Zeile mit einem erfolgreichen Mustervergleich wird für die Berechnung des Funktionswertes verwendet.

Beispiel: (Berechnung der Länge einer Liste)

```
length :: [a] -> Int
length []     = 0
length (x:xs) = 1 + length xs
```

Die Funktion *length* erhält eine Liste als Argument. Der erste Fall tritt ein, wenn die übergebene Liste leer *[]* ist. Enthält die Liste mindestens ein Element, so bestimmt die zweite Definition die weitere Verarbeitung. x enthält dabei den Kopf der Liste, xs den gesamten Rest. Da der Wert des Kopfelementes in diesem Fall nicht relevant ist, kann statt des x auch der Unterstrich _ als Platzhalter angegeben werden.

Beispiel: (Pattern Matching mit Platzhalter)

```
length (_:xs) = 1 + length xs
```

Pattern Matching ist auch für algebraische Typen (siehe Abschnitt 2.3.2) möglich. Haskell führt hier einen Test des aktuell übergebenen Typkonstruktors gegen die verschiedenen definierten Fälle durch.

Beispiel: (Addition aller Werte eines Baumes)

```
addTreeValues :: Bintree Int -> Int
addTreeValues (Leaf x)       = x
addTreeValues (Branch t1 t2) = addTreeValues t1 + addTreeValues t2
```

As-Pattern

Um auf ein, aus mehreren Teilen bestehendes, Muster zugreifen zu können, existiert der Operator @.

Beispiel: (Restliste ab dem Wert 42 bestimmen)

```
getFrom42 :: [Int] -> [Int]
getFrom42 []          = []
getFrom42 list@(x:xs) = if x == 42 then list else getFrom42 xs
```

Wenn der Wert des Listenkopfes 42 beträgt, soll die Liste inklusive des Kopfes zurückgegeben werden. Ohne die Verwendung des @-Operators müsste die Ergebnisliste neu konstruiert werden *(x:xs)*, das Nutzen eines As-Pattern erspart diese erneute Berechnung.

2.5 Guards

Guards ermöglichen boolsche Tests auf den Argumenten einer Funktion. Die erste erfüllte Bedingung bestimmt dabei den Wert der Funktion. Das Schlüsselwort *otherwise* führt immer zu einem erfolgreichen Test.

Beispiel: (Bestimmen des Minimums zweier Zahlen)

```
min :: Int -> Int -> Int
min x y
   | x <= y    = x
   | otherwise = y
```

3 Das XPath-Datenmodell

Die Sprache XPath dient zur Adressierung von Teilen eines XML-Dokumentes. Sie nutzt eine kompakte Nicht-XML-Syntax, um die Verwendung von XPath-Ausdrücken innerhalb von URIs[1] und XML-Attributen zu ermöglichen. Die Nicht-XML-Syntax wird vor allem in den Implementationen von XSLT[2] und XPointer benötigt, für die XPath entworfen wurde. Aber auch weitere Spezifikationen können die zur Verfügung gestellte Adressierungstechnik nutzen.
Der Name XPath leitet sich von einer, auch in URLs[3] genutzten, Pfad-Notation (path) ab, mit der sich durch die hierarchische Struktur eines XML-Dokumentes navigieren lässt.

3.1 Repräsentation eines XML-Dokumentes

XPath operiert auf der logischen Struktur eines XML-Dokumentes, nicht auf seiner äußerlichen Syntax. Ein Dokument wird als Baum mit verschiedenen Knotentypen repräsentiert.

Beispiel: (XML-Dokument mit der entsprechenden XPath-Baumstruktur [Becker 02])

```
<?xml version="1.0"?>
<!DOCTYPE rezept SYSTEM "rezept.dtd">
<?xml-stylesheet href="style.xsl" type="text/xml"?>
<rezept>
   <zutat id="mehl">200g Mehl</zutat>
   <!-- weitere Zutaten -->
   <anleitung>
      Zuerst nehmen Sie das
      <zutat xmlns:xlink="http://www.w3.org/1999/xlink"
             xlink:type="simple" xlink:href="mehl">Mehl</zutat>
      und mischen es mit ...
   </anleitung>
</rezept>
```

Die für die Verarbeitung eines XML-Dokumentes benötigte Baumstruktur (siehe Abbildung 3.1) wird bereits durch die Haskell XML Toolbox bereitgestellt und ist nicht Gegenstand dieser Arbeit.

Hinweis: Die grundlegenden Datenstrukturen und Verarbeitungstechniken der Haskell XML Toolbox werden in der Master Thesis von Martin Schmidt [Schmidt 02] beschrieben. Die folgenden Abschnitte beziehen sich auf diese Strukturen, so dass sich ein Blick in die Thesis empfiehlt.

[1] Uniform Resource Identifier: generische Menge aller Benennungs- und Adressierungsarten von Ressourcen
[2] eXtensible Stylesheet Language Transformations
[3] Uniform Resource Locator: vereinheitlichte, explizite Zugriffsanweisung zu einer Ressource im Internet

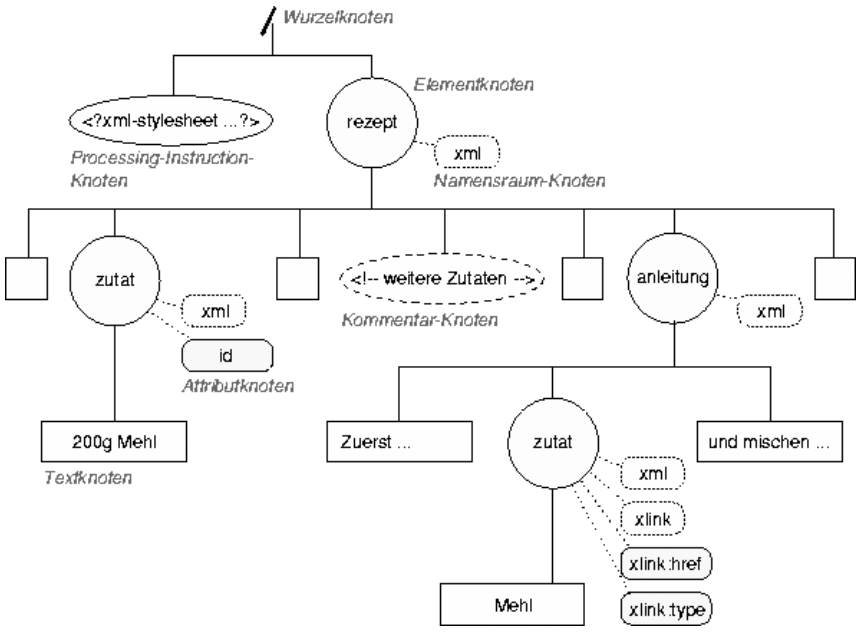

Abbildung 3.1: XML-Dokument als Baum-Repräsentation

3.1.1 Knotentypen

Die XPath-Spezifikation unterscheidet 7 Knotentypen:

- Wurzelknoten
- Elementknoten
- Attributknoten
- Namensraumknoten
- Processing-Instruction-Knoten
- Kommentarknoten
- Textknoten

Elemente der Document Type Definition (DTD) sowie die XML-Deklaration gehören nicht zum XPath-Datenmodell. Es kann durch einen XPath-Ausdruck auch nicht auf diese Informationen zugegriffen werden. Die entsprechenden *XNodes*: *XDTD* und *XPi* (mit dem Attribut *version*) werden vor der Verarbeitung, durch die Funktion *canonicalizeForXPath*, aus dem Baum entfernt.

Jeder Knotentyp besitzt einen Zeichenkettenwert. Bei einigen Knotentypen ist er Teil des Knotens, bei anderen wird er aus den Zeichenkettenwerten der Nachkommen berechnet.

Wurzelknoten

Der Wurzelknoten ist die Wurzel des Baumes und kommt genau einmal im Baum vor. Er besitzt neben Processing-Instruction- und Kommentarknoten, die im Prolog oder Epilog des XML-Dokumentes auftreten können, exakt einen weiteren Kindknoten, den Elementknoten für das Dokumentelement. Der Wurzelknoten wird durch keine eigene *XNode* abgebildet, sondern entspricht einem *XTag* mit dem Tagnamen /.
Der Zeichenkettenwert des Wurzelknotens ist die Verkettung der Zeichenkettenwerte aller nachfolgenden Textknoten.

Elementknoten

Elementknoten haben ihre Entsprechung in den XML-Elementen eines Dokumentes. Sie können weitere Element- sowie Namensraum-, Processing-Instruction-, Kommentar- oder Textknoten als Kinder haben. Attribute sind hingegen keine Kinder eines Elementes.
Ein Elementknoten wird in der Haskell XML Toolbox durch einen *XTag* dargestellt und kann einen eindeutigen Bezeichner (ID) als Attribut besitzen. Bei der Verarbeitung von IDs hat das Vorhandensein einer DTD somit Einfluss auf das Ergebnis eines XPath-Ausdruckes. Aus diesem Grund müssen XPath-Implementationen, vor dem Reduzieren des Baumes um die DTD, die benötigten Information extrahieren (siehe Abschnitt 4.1.3). Der Zeichenkettenwert eines Elementknotens ergibt sich aus der Verkettung der Zeichenkettenwerte aller Textknoten, die Nachkommen des Elementknotens in Dokumentordnung sind.

Attributknoten

Jeder Elementknoten besitzt eine Menge mit ihm verbundener Attributknoten. Der Elementknoten ist Elternknoten des Attributes, das Attribut selbst aber nicht Kindknoten. Ein Attribut besitzt keine weiteren Kinder oder Attribute und wird durch ein *XAttr* Knoten in der Haskell XML Toolbox dargestellt. Der Zeichenkettenwert ist der normalisierte Attributwert.

Namensraumknoten

Jedes Element hat assoziierte Namensraumknoten. Sie entstehen durch die Namensraumdeklaration mit den Attributen *xmlns* oder *xmlns:praefix*, in deren Sichtbarkeitsbereich sich das Element befindet. Der Zeichenkettenwert ist der Namensraum-URI. Namensraumknoten werden in der Haskell XML Toolbox nicht durch einen eigenständigen Knotentyp repräsentiert.

Processing-Instruction-Knoten

Für jede Processing-Instruction (im Folgenden als PI bezeichnet), die außerhalb der DTD definiert wurde, existiert ein PI-Knoten (*XPi*). Die XML-Deklaration ist keine PI, so dass für sie auch kein entsprechender Knoten vorhanden ist. Der Zeichenkettenwert eines PI-Knotens ist der Teil, der nach dem Ziel und sämtlichem Leerraum folgt, jedoch ohne das abschließende ?>.

Kommentarknoten

Kommentare, die nicht innerhalb der DTD auftreten, werden im XPath-Baum als Kommentarknoten (*XCmt*) abgebildet. Der Zeichenkettenwert ist der Inhalt des Kommentars ohne die öffnenden <!−− und schließenden Kommentarzeichen −−>.

Textknoten

Die Schriftzeichen eines XML-Dokumentes werden in Textknoten zusammengefasst. Ein Textknoten (*XText*) besitzt niemals einen weiteren Textknoten als direkten Vorgänger oder Nachfolger und enthält mindestens ein Zeichen. Der Zeichenkettenwert besteht aus den Schriftzeichen des Knotens. Textknoten verwalten keine Information darüber, wie die Zeichen im Dokument dargestellt wurden. Das Textknoten-Zeichen A kann als Zeichen A, aber auch als A oder A im XML-Dokument vorkommen.

Textknoten, die nur Leerraumzeichen enthalten, sind möglich. Die XPath-Spezifikation überlässt es der Anwendung, die sie verwendet, ob Leerraumzeichen durch einen Textknoten repräsentiert werden sollen oder nicht. Die Haskell XML Toolbox legt eigene Textknoten für Leerraumzeichen an, was dem Standardverhalten bei der Verarbeitung von Dokumenten durch XSLT entspricht.

3.1.2 Dokumentordnung

Innerhalb des entstehenden Baumes herrscht eine Ordnung, die Dokumentordnung. Sie korrespondiert zu der Anordnung der Elemente im XML-Dokument. Der Wurzelknoten ist der erste Knoten des Baumes. Elementknoten werden in der Reihenfolge ihrer Start-Tags im XML-Dokument angeordnet. Die Attribut- und Namensraumknoten eines Elementes stehen vor den Kindern des Elementes, wobei Namensraumknoten vor den Attributknoten erscheinen. Die relative Ordnung innerhalb der Namensraum- und Attributknoten ist implementationsabhängig und wird im konkreten Fall durch die Haskell XML Toolbox bestimmt.

Die XPath-Spezifikation verwendet den Begriff Menge abweichend von seiner mathematischen Bedeutung. Innerhalb einer Knotenmenge liegt in XPath eine Ordnung vor, so dass die Berechnung der Position eines Elementes in der Menge möglich ist.

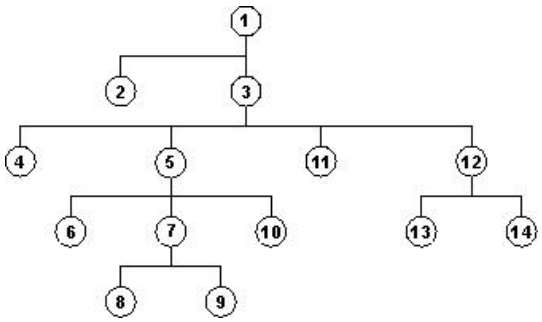

Abbildung 3.2: Dokumentordnung

3.2 Parsen von Ausdrücken

Der Ausdruck ist das primäre syntaktische Konstrukt von XPath. Er kann aus boolschen Operatoren, arithmetischen Ausdrücken, Variablenreferenzen, Zeichenketten (Strings), Zahlen, Lokalisierungspfaden (Locationpaths) oder aus Funktionen der Funktionsbibliothek (Core-Functions) bestehen.

Zum Parsen von Ausdrücken wird, wie bereits für das Parsen von XML-Dokumenten durch die Haskell XML Toolbox, Parsec [Parsec 00] verwendet. Parsec ist eine auf Monads basierende Parser-Kombinatoren-Bibliothek für Haskell und wurde von Daan Leijen entwickelt.
Die einfache Grammatik von XPath erlaubt es, auf ein Zerlegen des Ausdruckes in lexikalische Einheiten (Token) zu verzichten. Es ist kein separater Scanner notwendig, so dass ein Ausdruck direkt geparst werden kann.
Die Repräsentation eines geparsten XPath-Ausdruckes erfolgt unabhängig von dem Datenmodell der Haskell XML Toolbox. Durch die NICHT-XML-Syntax (siehe Abschnitt: 3) ist es notwendig, ein eigenes Modell für die Abbildung eines Ausdruckes zu entwickeln.

Der algebraische Typ *Expr* ist der zentrale Punkt des Datenmodells. Er bildet die möglichen, unterschiedlichen Arten eines Ausdruckes ab. Zu ihnen zählen die boolschen/arithmetischen Ausdrücke; aber auch Lokalisierungspfade und Variablen werden im Modell als Ausdrücke angesehen.

3.3 Boolsche und arithmetische Ausdrücke

Die boolschen und arithmetischen Ausdrücke werden zu einem Ausdruckstyp zusammengefasst, da ihre Verarbeitung konzeptionell identisch ist.

Priorität	Operator	Bedeutung	Name im Modell
1	or	logisches oder	Or
2	and	logisches und	And
3	=	gleich	Eq
3	!=	ungleich	NEq
4	<	kleiner	Less
4	<=	kleiner oder gleich	LessEq
4	>	grösser	Greater
4	>=	grösser oder gleich	GreaterEq
5	+	plus	Plus
5	-	minus	Minus
6	*	multiplikation	Mult
6	div	division	Div
6	mod	modulo	Mod
7	-	unäres minus	Unary

Tabelle 3.1: Boolsche und arithmetische Operatoren (1 = niedrigste Priorität)

Aus den Operatoren leitet sich der erste Ansatz für den Datentyp *Expr* ab.

Beispiel: (Erster Ansatz für den Datentyp Expr)

```
data Expr = OrExpr Expr Expr
          | AndExpr Expr Expr
          | PlusExpr Expr Expr
          | EqExpr Expr Expr
          | ...
```

Jedem Operator wird ein Typkonstruktor zugeordnet, der jeweils zwei Werte vom Typ *Expr* beinhaltet. Sie entsprechen der linken bzw. rechten Seite des Ausdruckes.

Beispiel: (Darstellung eines additiven Ausdruckes)

`1 + 2 + 3 = 6`
$\rightsquigarrow$ `(1 + 2 + 3) = 6` (implizite Klammerung der höheren Priorität)
$\rightsquigarrow$ `EqExpr (PlusExpr 1 (PlusExpr 2 3) 6)`[4]

Dieser Ansatz besitzt allerdings zwei gravierende Schwächen:

1. Für jeden zweistelligen Ausdruck wird ein eigenes Element erzeugt.
2. Die Links-Assoziativität der Operatoren gleicher Priorität wird nicht berücksichtigt.

[4]der Datentyp für Zahlen wird später betrachtet und kann an dieser Stelle vernachlässigt werden

Obwohl derselbe Operator + auf die drei Werte angewendet wird, müssen zwei Elemente des Typen *PlusExpr* erzeugt werden. Dies ergibt vor allem bei längeren Ausdrücken einen zusätzlichen Verwaltungsaufwand, der zu vermeiden ist.
Operatoren gleicher Priorität sind in XPath links-assoziativ zu klammern. Der obige Ansatz klammert allerdings rechts-assoziativ ((1 + (2 + 3)) = 6), was bei relationalen Ausdrücken zu Fehlern führt.

Beispiel: (Assoziativität)
`1 < 2 < 3`
⇝ `1 < (2 < 3)` → `Falsch` (rechts-assoziativ)[5]
⇝ `(1 < 2) < 3` → `Richtig` (links-assoziativ)

Durch die Verwendung einer Liste von Expressions, anstatt der beiden getrennten Teilausdrücke, lassen sich diese Nachteile beheben.

Beispiel: (Datentyp Expr)
```
data Expr = OrExpr [Expr]
          | AndExpr [Expr]
          | PlusExpr [Expr]
          | EqExpr [Expr]
          | ...
```
Die Elemente der Liste können sequentiell verarbeitet werden (links-assoziativ), und es existiert für einen beliebig langen Ausdruck gleicher Operatoren genau ein Typkonstruktor.

Beispiel: (Darstellung der Operanden als Liste)
`1 + 2 + 3 = 6`
⇝ `EqExpr (PlusExpr 1 2 3) 6)`

Die 15 Operatoren, die im Sprachumfang von XPath vorhanden sind, führen im jetzigen Modell zu 15 Produkttypen (siehe Abschnitt: 2.3.2), die jeweils eine Liste von Expressions enthalten. An dieser Stelle bietet sich die Verwendung eines Aufzählungstypen *Op* für die Operatoren an, so dass nur noch ein allgemeiner Produkttyp *GenExpr* im Modell vorhanden ist[6].

Beispiel: (Operatoren im Datenmodell von XPath)
```
data Op   = Or | And | Eq | NEq | GreaterEq | Greater | LessEq
          | Less |Plus | Minus | Div | Mod | Mult| Unary | Union

data Expr = GenExpr Op [Expr]
```

Unäres Minus

Der Operator *Unary* berechnet das unäre Minus eines Wertes. Obwohl das unäre Minus eine einstellige Funktion ist, alle anderen Operatoren entsprechen zweistelligen Funktionen, kann es einfach durch das Datenmodell dargestellt werden. Die Liste von Ausdrücken besteht in diesem Fall aus genau einem Wert. Die zweifache Anwendung des Operators auf einen Wert ergibt zwei Unary-Elemente.

Beispiel: (Doppelte Negation)
`--42`
⇝ `UnaryExpr (UnaryExpr 42)`

Ein Ausdruck kann nicht durch das Entfernen eines doppelten Minus-Zeichens, während des

[5] es erfolgt eine Konversion des boolschen Zwischenergebnisses in eine Zahl (False ↦ 0, True ↦ 1)
[6] der Union-Operator vereinigt Knotenmengen (siehe Abschnitt 3.6.5)

Parsens, vereinfacht werden. Hierbei kann ein Fehler auftreten, wenn der zu negierende Wert der kleinsten darstellbaren Zahl entspricht.

3.4 Zahlen und Zeichenketten

Im nächsten Schritt soll das Datenmodell um Typen für Zahlen und Zeichenketten ergänzt werden. Zeichenketten (Literale) können innerhalb der Haskell XML Toolbox aus Zeichen des gesamten Unicode Bereiches bestehen und werden durch den Haskell Datentyp String dargestellt.

```
type Literal = String
```

Einige Funktionen der Funktionsbibliothek, die auf Zeichenketten operieren, wie beispielsweise *string-length* oder *translate* arbeiten noch nicht mit beliebigen Unicode-Zeichen. Sobald die zurzeit noch fehlende Funktionalität durch das Modul *Unicode* der Haskell XML Toolbox bereitgestellt wird, ist aber der Einsatz beliebiger Zeichen innerhalb von XPath möglich.

Zahlen können jeden Gleitkommawert nach dem IEEE-754-Standard [IEEE 00] annehmen. Dieser beinhaltet einige besondere Werte, die in der folgenden Tabelle dargestellt sind.

Zahlenwert	Name im Modell
normale Fließkommazahl	Float f
negativ Unendlich	NegInf
positiv Unendlich	PosInf
negativ Null	Neg0
positiv Null	Pos0
Not-a-Number	NaN

Tabelle 3.2: Zahlenwerte im IEEE-754-Standard

Es ist allerdings nicht erlaubt, die speziellen Werte direkt in einem Ausdruck anzugeben. Sie können nur berechnet werden.

```
1 div 0 ↦ PosInf
```

Bei Berechnungen in XPath treten keine Fehler, Ausnahmen oder Überläufe auf. Ist die Berechnung eines Ausdruckes nicht möglich, weil es sich bei einem Operanden beispielsweise um eine Zeichenkette handelt, wird der Wert NaN als Ergebnis geliefert.

Beispiel: (Datentyp für Zahlen und Zeichenketten)

```
data XPNumber = Float Float
              | NaN
              | NegInf
              | Neg0
              | Pos0
              | PosInf

Expr = GenExpr Op [Expr]
     | LiteralExpr Literal
     | NumberExpr XPNumber
```

Zahlendarstellung

Eine Zahl kann nicht in Exponentialdarstellung (2.99792E+08) angegeben werden. Es existiert weder ein spezieller Typ für ganzzahlige Werte, noch für besondere Darstellungen, die eine Zahl als Oktal- oder Hexadezimalzahl interpretieren. Ein Wert kann ein negatives Vorzeichen besitzen. Die Angabe eines expliziten positiven Vorzeichens ist jedoch nicht erlaubt.
Der Parser normalisiert alle syntaktisch korrekten Eingaben für Zahlen in die Form a.b[7], um eine einheitliche Verarbeitung zu gewährleisten.

Beispiel: (Normalisieren von Zahlen durch den Parser)
```
21. ↦ 21.0
42  ↦ 42.0
.1  ↦  0.1
```

```
number :: Parser String
number
    = do
      tokenParser (symbol ".")
      d <- many1 digit
      return ("0." ++ d)
      <|>
      do
      d <- many1 digit
      d1 <- option "" ( do
                        tokenParser (symbol ".")
                        d2 <- option "0" (many1 digit)
                        return ("." ++ d2)
                      )
      return (d ++ d1)
      <?> "number"
```

3.5 Variablen und Funktionen

Über Variablen lassen sich Informationen aus der Umgebung in die Verarbeitung eines XPath-Ausdruckes integrieren. Variablen besitzen einen Namen, über den sie eindeutig identifiziert werden.

Beispiel: (Datentyp für einen Variablennamen)
```
type VarName = String
```

Während des Parsens wird jeder syntaktisch korrekte Variablenname als Variable erkannt. Es wird zu diesem Zeitpunkt noch nicht überprüft, ob eine Variable mit diesem Namen existiert.

Funktionen

Funktion besitzen wie Variablen einen eindeutigen Namen und zusätzlich eine durch Kommata getrennte Liste von Argumenten, für die ein beliebiger Ausdruck zugelassen ist.

[7] a und b entsprechen einer beliebigen Ziffernfolge ≥ 1

Beispiel: (Darstellung einer Funktion im Modell)

```
type FctName      = String
type FctArguments = [Expr]
```

Eine Kontrolle der Eingabe erfolgt auch hier nur auf syntaktischer Ebene. Alle weiteren Überprüfungen[8] finden erst während der Auswertung (siehe Abschnitt 4.1.2) statt. Hierdurch wird eine strikte Trennung der syntaktischen (Parsen) von der inhaltlichen Überprüfung (Auswertung) erreicht.

Beispiel: (Datentyp Expr inkl. der beiden Produkttypen VarExpr und FctExpr)

```
data Expr = GenExpr Op [Expr]
          | LiteralExpr Literal
          | NumberExpr XPNumber
          | VarExpr VarName
          | FctExpr FctName FctArguments
```

3.6 Lokalisierungspfade

Lokalisierungspfade sind die wichtigsten Teilausdrücke von XPath. Sie erlauben das Navigieren in einem XML-Baum über Achsen. Ein Lokalisierungspfad besteht aus einer Liste von Lokalisierungsschritten, die nacheinander auf einen Baum angewendet werden. Jeder Lokalisierungsschritt selektiert dabei eine Teilmenge des aktuellen Baumes. Diese resultierende Teilmenge ist Ausgangsmenge für den nächsten Lokalisierungsschritt.
Ein Schritt besteht aus drei Teilen:

1. aus einer Achse, die die Beziehung, zwischen den durch den Lokalisierungsschritt ausgewählten Knoten und dem Kontextknoten, innerhalb des Baumes spezifiziert.
2. aus einem Knotentest, der den Knotentyp und den erweiterten Namen der durch den Lokalisierungsschritt ausgewählten Knoten spezifiziert.
3. aus null oder mehr Prädikaten, die mittels beliebiger Ausdrücke die durch eine Achse und anschließendem Knotentest ausgewählte Teilmenge weiter verfeinern können.

3.6.1 Achsen

Achsen filtern die Knoten eines Baumes bezüglich einer Richtung. Eine vorwärts gerichtete Achse enthält immer nur den aktuellen Knoten (Kontextknoten) oder Knoten, die nach dem Kontextknoten im Dokument (siehe Abschnitt: 3.1.2) auftreten. Die Position eines Knotens in der resultierenden Menge, welche in Dokumentordnung vorliegt, wird als Näheposition bezeichnet. Eine rückwärts gerichtete Achse umfasst entsprechend den Kontextknoten und Knoten, die vor ihm auftreten. Die Näheposition wird anhand der umgekehrten Dokumentordnung bestimmt.

Die self-Achse enthält immer nur einen Knoten (den Kontextknoten), sie kann daher als vorwärts bzw. rückwärts gerichtet angesehen werden. Die attribute- und namespace-Achsen haben keine Richtung, da die relative Ordnung der Attribut- bzw. Namensraumknoten nicht durch XPath festgelegt wird, sondern implementationsabhängig ist.

Die folgenden Abbildungen [Becker 02] verdeutlichen die durch die jeweilige Achse selektierten Knoten. Der grau hinterlegte Knoten entspricht dem aktuellen Kontextknoten.

[8] Funktion mit dem Namen vorhanden, Anzahl und Art der Argumente

Achsenname	Name im Modell	Richtung
ancestor	Ancestor	rückwärts gerichtet
ancestor-or-self	AncestorOrSelf	rückwärts gerichtet
attribute	Attribute	keine Richtung
child	Child	vorwärts gerichtet
descendant	Descendant	vorwärts gerichtet
descendant-or-self	DescendantOrSelf	vorwärts gerichtet
following	Following	vorwärts gerichtet
following-sibling	FollowingSibling	vorwärts gerichtet
namespace	Namespace	keine Richtung
parent	Parent	vorwärts gerichtet
preceding	Preceding	rückwärts gerichtet
preceding-sibling	PrecedingSibling	rückwärts gerichtet
self	Self	vorwärts/rückwärts gerichtet

Tabelle 3.3: XPath-Achsen

Die Ziffern geben die Näheposition in der resultierenden Menge an. Es handelt sich bei den dargestellten Knoten um keine speziellen Knotentypen (siehe Abschnitt: 3.1.1): Es können Element-, Text-, Kommentar- oder Processing-Instruction-Knoten sein. Sie repräsentieren allerdings niemals Attribut- oder Namensraumknoten, da diese nur über eigens dafür definierte Achsen erreicht werden können.

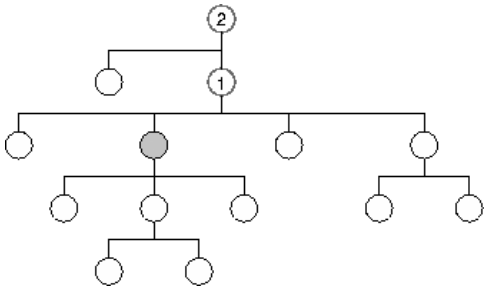

Abbildung 3.3: ancestor-Achse

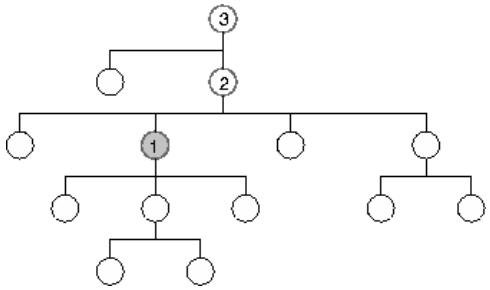

Abbildung 3.4: ancestor-or-self-Achse

Die ancestor-Achse enthält die Vorfahren des Kontextknotens. Die Vorfahren bestehen aus dem Elternknoten des Kontextknotens, dessen Elternknoten usw. Sie enthält somit immer den Wurzelknoten, es sei denn, der Kontextknoten selbst ist der Wurzelknoten. Die ancestor-or-self-Achse beinhaltet neben den Vorfahren noch den Kontextknoten selbst.

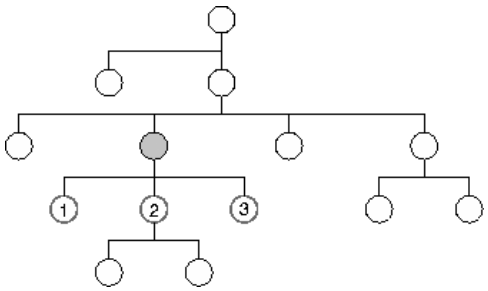

Abbildung 3.5: child-Achse

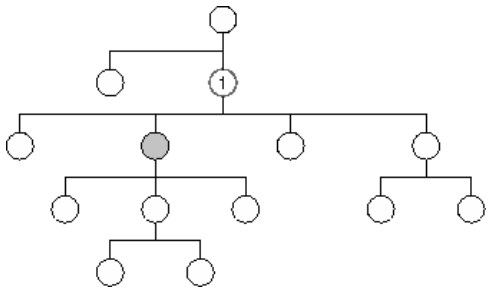

Abbildung 3.6: parent-Achse

Die Achse child enthält die Kinder, die parent-Achse den Elternknoten des Kontextknotens. Ist der Kontextknoten die Wurzel, selektiert die parent-Achse keine Elemente.

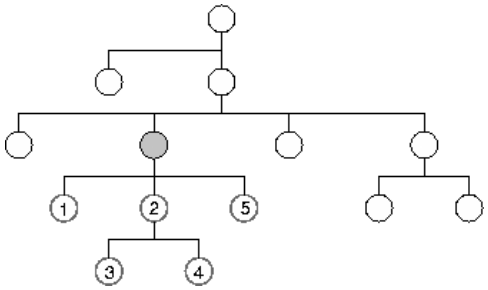

Abbildung 3.7: descendant-Achse

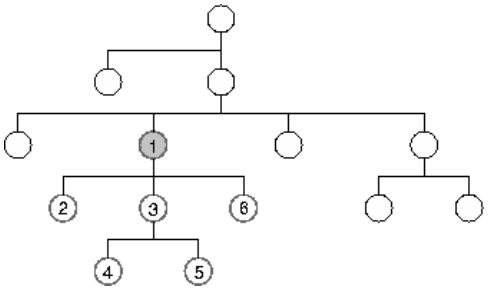

Abbildung 3.8: descendant-or-self-Achse

Die Achse descendant beinhaltet die Nachkommen des Kontextknotens. Ein Nachkomme ist ein Kind oder ein Kind eines Kindes usw. Die descendant-or-self-Achse enthält zusätzlich den Kontextknoten.

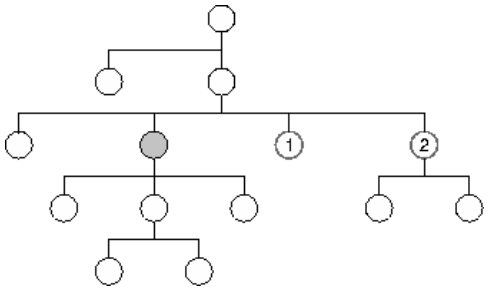

Abbildung 3.9: following-sibling-Achse

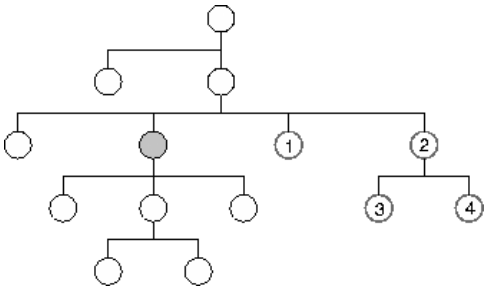

Abbildung 3.10: following-Achse

Die following-sibling-Achse umfasst alle nachfolgenden Geschwister des Kontextknotens. Ist der Kontextknoten ein Attribut- oder Namensraumknoten, ist diese Achse leer. Die following-Achse selektiert alle Knoten, die nach dem Kontextknoten in Dokumentordnung auftreten, und zwar ohne seine Nachkommen, Attribut- und Namensraumknoten.

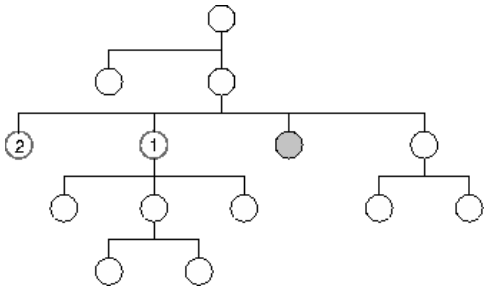

Abbildung 3.11: preceding-sibling-Achse

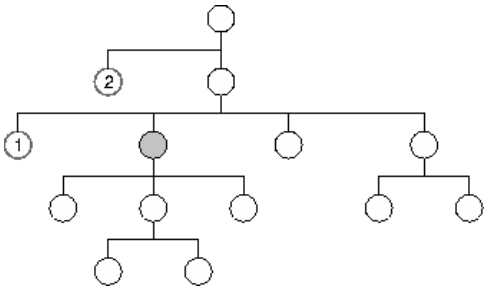

Abbildung 3.12: preceding-Achse

Die Achse preceding-sibling enthält alle vorhergehenden Geschwister des Kontextknotens. Die preceding-Achse umfasst alle Knoten, die vor dem Kontextknoten in Dokumentordnung auftreten, ohne seine Vorfahren, Attribut- und Namensraumknoten.

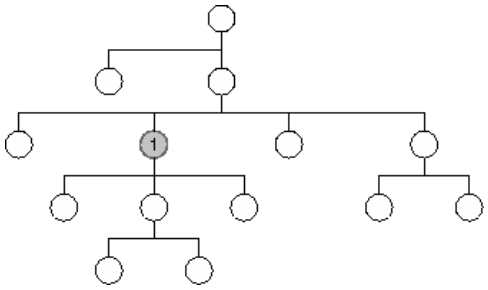

Abbildung 3.13: self-Achse

Die Achse self beinhaltet nur den Kontextknoten selbst.

Die attribute- bzw. namespace-Achse selektiert die entsprechenden Knotentypen des Kontextknotens. Diese Achsen sind leer, es sei denn, der Kontextknoten ist ein Elementknoten.

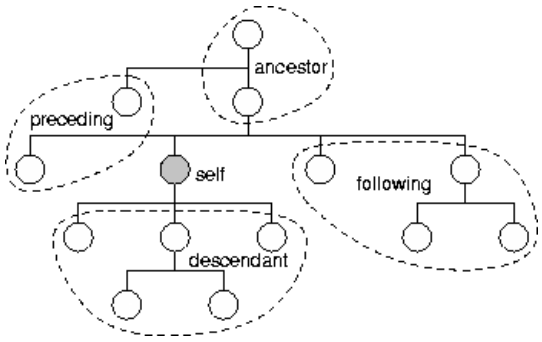

Abbildung 3.14: Zusammenwirken der Achsen

Die Achsen ancestor, descendant, following, preceding und self erreichen zusammen alle Knoten des Dokumentes[9] und überschneiden sich dabei nicht.

Das Datenmodell für einen Lokalisierungspfad

Ein Lokalisierungspfad lässt sich auf die folgende Weise in Haskell modellieren:

```
data LocationPath = LocPath [XStep]
```

XStep entspricht einem Lokalisierungsschritt. Jeder Konstruktor von *XStep* bildet eine der XPath-Achsen ab.[10]

Beispiel: (Datentyp für eine Lokalisierungsschritt)

```
data XStep = XAncestor NodeTest Predicates
           | XChild NodeTest Predicates
           | XParent NodeTest Predicates
           | XSelf NodeTest Predicates
           | ..
```

[9] Attribut- und Namensraumknoten werden hierbei nicht betrachtet

[10] die Datentypen für NodeTest und Predicates werden in den nächsten Abschnitten entwickelt

Auch an dieser Stelle wird das Datenmodell durch einen Aufzählungstypen für die Achsen vereinfacht (siehe Abschnitt: 3.3).

Beispiel: (Abbildung eines Lokalisierungspfades)

```
data LocationPath = LocPath [XStep]

data XStep    = Step AxisSpec NodeTest Predicates
data AxisSpec = Ancestor | AncestorOrSelf | Attribute | Child
              | Descendant | DescendantOrSelf | Following | FollowingSibling
              | Namespace | Parent | Preceding | PrecedingSibling | Self
```

3.6.2 Knotentests

Knotentests schränken die durch eine Achse selektierten Knoten um einen bestimmten Typ (siehe Abschnitt: 3.1.1) ein. Erfüllt ein Knoten den Test, wird er in die Resultatmenge aufgenommen. Jede Achse besitzt hierfür einen Hauptknotentyp. Der Hauptknotentyp für die attribute-Achse ist der Attributtyp, für die namespace-Achse der Namensraumtyp. Alle anderen Achsen haben den Elementtyp als Hauptknotentyp.
XPath unterscheidet zwei Arten von Knotentests, den Namenstest und den Typtest.

Namenstest

Ein Namenstest ist für einen Knoten erfolgreich, wenn der Knotentyp mit dem Hauptknotentyp der Achse übereinstimmt und sein Name dem Namenstest entspricht.

Beispiel: (Auswählen aller Kinder, die den Namen para besitzen)

```
child::para
```

Der Namenstest * ist für alle Knoten des Hauptknotentyps, unabhängig von ihrem Namen, erfüllt. `attribute::*` wählt alle Attribute eines Elementtyps, `child::*` entsprechend alle Kinder.

Typtest

Ein Typtest ist für einen Knoten erfolgreich, wenn sein Knotentyp dem geforderten Typ entspricht.
Der Test `comment()` selektiert alle Knoten des Typen Kommentar, `text()` alle Textknoten. Processing-Instruction-Knoten können über `processing-instruction()` ausgewählt werden. Bei PI-Knotentests ist noch die Angabe einer zusätzlichen Zeichenkette möglich:

```
child::processing-instruction('xml-stylesheet')
```

In diesem Beispiel werden alle Kinder ausgewählt, die vom Typ Processing-Instruction sind und zusätzlich den Namen *xml-stylesheet* besitzen.
Der Test `node()` ermöglicht es, jeden Knoten, unabhängig von seinem Typ, in die Resultatmenge aufzunehmen.

Beispiel: (Modell für einen Typtest)

```
data XPathNode = XPNode
               | XPCommentNode
               | XPPINode
               | XPTextNode
```

Der resultierende Datentyp für einen Typtest enthält vier Konstruktoren; sie entsprechen jeweils einem Test auf einen Knotentyp. Der Konstruktor *XPPINode* bildet einen

PI-Test, ohne die Angabe einer zusätzlichen Zeichenkette, ab. Um eine einheitliche Verarbeitung zu erreichen, wird ein PI-Knoten inklusive eines Namensvergleiches nicht mit in den Datentypen *XPathNode* aufgenommen. Bei der Auswertung von *XPathNode* soll nur auf den Typ, nicht aber auf den Namen getestet werden.
Die beiden Knotentests, die einen Namensvergleich fordern, werden als eigenständige Konstruktoren mit dem zu testenden Namen als Wert modelliert. Für den allgemeinen Knotentest ergibt sich der folgende Datentyp.

Beispiel: (Allgemeiner Knotentest)
```
type Name     = String
data NodeTest = NameTest Name
              | PI Name
              | TypeTest XPathNode
```

3.6.3 Prädikate

Die durch eine Achse und einen Knotentest bestimmte Menge von Elementen kann durch eine Liste von Prädikaten weiter gefiltert werden. Ein Prädikat kann ein beliebiger XPath-Ausdruck sein, der auf eine Knotenmenge angewendet wird. Die Ergebnismenge des ersten Prädikats ist Ausgangsmenge für das zweite Prädikat usw. Die Verarbeitung wird detailliert im Abschnitt 4.7.5 beschrieben.

3.6.4 Filter-Ausdrücke

Die im vorherigen Abschnitt beschriebenen Prädikate beziehen sich auf einen Lokalisierungsschritt und werden im Kontext der Achse ausgewertet. XPath ermöglicht auch, das Ergebnis eines gesamten Lokalisierungspfades durch ein Prädikat zu filtern. Wird ein Prädikat auf einen allgemeinen Ausdruck angewendet, der eine Knotenmenge liefert, so spricht man von einem Filter-Ausdruck. Ein Filter-Ausdruck kann eine Liste von beliebigen Ausdrücken sein.

Beispiel: (Filter-Ausdruck mit 2 Prädikaten)
```
(preceding-sibling::image)[@type='item'][42]
```

Filter-Ausdrücke können nur auf Knotenmengen angewendet werden, das Filtern einer Zeichenkette würde einen Fehler verursachen. Da das XPath-Modul eine strikte Trennung zwischen dem Parsen und der Verarbeitung macht, wird jeder syntaktisch korrekte Ausdruck durch den Parser erkannt. Das Identifizieren eines Fehlers erfolgt erst bei der Auswertung des Ausdruckes.

Beispiel: (Filter-Ausdruck im Datentyp Expr)
```
data Expr = FilterExpr [Expr]  -- NodeSet ++ [predicate]
          | ...
```

3.6.5 Relative und absolute Lokalisierungspfade

Es gibt zwei Arten von Lokalisierungspfaden, relative und absolute. Ein relativer Pfad ist, wie in den vorhergehenden Absätzen beschrieben, eine Liste von Lokalisierungsschritten, die durch den Schrägstrich / getrennt sind. Ein absoluter Pfad beginnt mit einem Schrägstrich, gefolgt von einem relativen Lokalisierungspfad. Durch den einleitenden Schrägstrich wird die Auswertung des Pfades nicht am aktuellen Kontextknoten, sondern an der Wurzel begonnen.

Beispiel: (relativer und absoluter Lokalisierungspfad)
```
child::chapter/descendant::para      (relativer Pfad)
/child::item/child::text()           (absoluter Pfad)
```

Im obigen Beispiel wählt der relative Pfad alle para-Elemente aus, die Nachkommen der chapter-Kindelemente des Kontextknotens sind. Der absolute Pfad selektiert alle Textknoten, die ein item-Vaterelement besitzen, das Kind der Dokumentenwurzel ist.

Das Modell für einen Lokalisierungspfad wird um die Art des Pfades erweitert.

Beispiel: (Vollständige Darstellung eines Lokalisierungspfades)

```
data LocationPath = LocPath Path [XStep]
data Path         = Rel | Abs
data XStep        = Step AxisSpec NodeTest [Expr]
```

Ein Lokalisierungspfad kann über den Schrägstrich / bzw. den doppelten Schrägstrich // mit einem Ausdruck verbunden werden. So ist die Anwendung eines Lokalisierungspfades auf einen Filter-Ausdruck möglich. Die Schrägstriche haben dieselbe Bedeutung wie bei Lokalisierungsschritten.

Beispiel: (Verbinden eines Filter-Ausdruckes mit einem Lokalisierungspfad)

```
(preceding-sibling::image)[@type='item'][42]/parent::text()
```

Allgemeiner Pfad

Ein allgemeiner Pfad kann aus einem optionalen Ausdruck und einem optionalen Lokalisierungspfad bestehen. Einer der beiden Teile muss vorhanden sein, beide Teile können vorhanden sein. Das Ergebnis eines allgemeinen Pfades ist immer eine Knotenmenge.

Beispiel: (Darstellung eines allgemeinen Pfades)

```
data Expr = PathExpr (Maybe Expr) (Maybe LocationPath)
          | ...
```

Vereinigung von Knotenmengen

Die durch einen allgemeinen Pfad ausgewählte Knotenmenge kann mit einer weiteren Knotenmenge durch den Operator | vereinigt werden. Er entspricht der mathematischen Mengenvereinigung. Der Vereinigungsoperator wird im Modell den boolschen und arithmetischen Operatoren (siehe Abschnitt 3.3) zugeordnet, da die Verarbeitung konzeptionell identisch ist.

3.6.6 Abkürzungen

Einige Elemente von Lokalisierungspfaden lassen sich durch Abkürzungen beschreiben.

Abkürzung	Langform	Beschreibung
	child::	die child-Achse kann weggelassen werden, sie entspricht der Standardachse von XPath
.	self::node()	der einfache Punkt ist ein Alias für die self-Achse
..	parent::node()	durch zwei Punkte läßt sich die parent-Achse darstellen
//	/descendant-or-self::node()/	Abkürzung der descendant-or-self-Achse
@	attribute::	Kurzform der attribute-Achse

Tabelle 3.4: XPath-Abkürzungen

Das folgende Beispiel stellt die Verwendung aller Abkürzungen dar. Der resultierende Lokalisierungspfad entspricht keinem sinnvollen XPath-Ausdruck.

Beispiel: (Abgekürzte Syntax in einem Lokalisierungspfad)

```
item[@para='elem']//./../text()
⇝ child::item[attribute::para='elem']/descandent-or-self::node()
    /self::node()/parent::node()/child::text()
```

Parsen von Abkürzungen

Abkürzungen können beim Parsen prinzipiell auf zwei Arten verarbeitet werden. In der ersten Variante durchläuft ein zusätzlicher Parser den Ausdruck und ersetzt alle Abkürzungen durch ihre Normalformen. Der eigentliche Parser arbeitet anschließend auf einer normalisierten Form des Ausdruckes. Bei der zweiten Version werden die Abkürzungen direkt durch den Ausdrucksparser interpretiert und verarbeitet.
Da XPath nur wenige einfache Abkürzungsmöglichkeiten bietet, arbeitet der Haskell XML Toolbox Parser mit der zweiten Variante. Er bildet die Kurzformen direkt auf die entsprechenden Lokalisierungsschritte ab. Ein eigener Datentyp für Abkürzungen wird folglich nicht gebraucht.

Beispiel: (Parser für die parent- und self-Achse)

```
abbrStep :: Parser XStep
abbrStep
   =  do
      tokenParser (symbol "..")
      return (Step Parent (TypeTest XPNode) [])
      <|>
      do
      tokenParser (symbol ".")
      return (Step Self (TypeTest XPNode) [])
      <?> "abbrStep"
```

3.7 Darstellung eines geparsten Ausdruckes

Jeder XPath-Ausdruck lässt sich durch das folgende Datenmodell modellieren.

Beispiel: (Vollständiges Datenmodell für einen Ausdruck)

```
data Expr = GenExpr Op [Expr]
          | PathExpr (Maybe Expr) (Maybe LocationPath)
          | FilterExpr [Expr]  -- NodeSet ++ [predicate]
          | VarExpr VarName
          | LiteralExpr Literal
          | NumberExpr XPNumber
          | FctExpr FctName FctArguments
```

Mit der Funktion *expr2XPathTree* aus dem Modul *XPathToString* ist es möglich, einen geparsten Ausdruck als Baum darzustellen. Die Funktion konvertiert dabei das XPath-Datenmodell für einen Ausdruck (*Expr*) in das Modell für einen allgemeinen Baum (*NTree*).

Beispiel: (Funktion expr2XPathTree)

```
type XPathTree = NTree String

expr2XPathTree                  :: Expr -> XPathTree
expr2XPathTree (GenExpr op ex)  = NTree (show op) (map expr2XPathTree ex)
expr2XPathTree (NumberExpr f)   = NTree (show f) []
expr2XPathTree (LiteralExpr s)  = NTree s []
expr2XPathTree (VarExpr name)   = NTree ("Var: " ++ name) []
expr2XPathTree (FctExpr n arg)  = NTree ("Fct: " ++ n) (map expr2XPathTree arg)
```

Alle weiteren Funktionen zur Formatierung des *NTree*-Baumes werden durch die Haskell XML Toolbox zur Verfügung gestellt.

Beispiel: (Darstellung eines XPath-Ausdruckes als Baum)

```
parent::text()[42] - round('str') - (/self::item)[$varName]

---Minus
   |
   +---RelLocationPath
   |   |
   |   +---Parent
   |       |
   |       +---TypeTest: text()
   |       |
   |       +---Predicates
   |           |
   |           +---42.0
   |
   +---Fct: round
   |   |
   |   +---str
   |
   +---FilterExpr
       |
       +---AbsLocationPath
       |   |
       |   +---Self
       |       |
       |       +---NameTest: "item"
       |
       +---Predicates
           |
           +---Var: varName
```

4 Auswertung eines Ausdruckes

Dieses Kapitel beschreibt, wie ein, durch die im vorherigen Kapitel dargelegten Techniken, geparster Ausdruck ausgewertet wird. Das Ergebnis der Auswertung ist ein XPath-Objekt. XPath definiert vier Grundtypen, die bei jeder Implementierung enthalten sein müssen. Auf XPath aufbauende Spezifikationen können weitere Objekttypen[1] definieren, um das Ergebnis eines XPath-Ausdruckes ihrem Arbeitskontext anzupassen.

Ergebnistyp	Beschreibung	Name im Modell
node-set	eine Knotenmenge ohne Duplikate	XPVNode NodeSet
boolean	ein Wahrheitswert (wahr oder falsch)	XPVBool Bool
number	eine Gleitkommazahl nach IEEE 754	XPVNumber XPNumber
string	eine Zeichenkette bestehend aus UCS-Zeichen	XPVString String

Tabelle 4.1: Ergebnis-Grundtypen einer Auswertung

Es existiert innerhalb von XPath kein Objekt, das einen Knoten repräsentiert. Ein einzelner Knoten kann aber als Knotenmenge (node-set) mit nur einem Element verstanden werden. Eine Knotenmenge wird durch eine Liste von navigierbaren Bäumen (siehe Abschnitt 4.7.1) dargestellt.

Beispiel: (Typdefinition für eine Knotenmenge)

```
type NavXmlTrees = [NavXmlTree]
type NodeSet     = NavXmlTrees
```

Die Objekte für Zahlen (number) und Zeichenketten (string) entsprechen den Typen, die im Kapitel 3.4 beschrieben wurden.

Die Haskell XML Toolbox Implementation von XPath erweitert die Grundtypen um einen Typ für den Fehlerfall.

Beispiel: (Ergebnistypen der Haskell XML Toolbox)

```
data XPathValue = XPVNode NodeSet
                | XPVBool Bool
                | XPVNumber XPNumber
                | XPVString String
                | XPVError String
```

Der Wert des Typkonstruktors *XPVError* wird verwendet, um detaillierte Fehlermeldungen an den Nutzer der Haskell XML Toolbox zu übermitteln. Einerseits werden dabei Fehler abgedeckt, die aus einer falschen Eingabe des Benutzers resultieren. Andererseits dient *XPVError* auch dazu, bei internen, nicht durch den Nutzer verursachten Fehlern, den Ablauf mit einer sinnvollen Fehlermeldung zu beenden.

[1] Die XSLT-1.0-Spezifikation enthält beispielsweise den neuen Typ Ergebnisteilbaum (result tree fragment)

Beispiel: (ein durch den Nutzer verursachter Fehler)

```
evalFct :: FctName -> Env -> Context -> [XPathValue] -> XPathValue
evalFct name env cont args
    = case (lookup name fctTable) of
        Nothing -> XPVError ("Call to undefined function: "++ name)
        Just (fct, checkArgCount) -> ...
```

Die Haskell XML Toolbox findet im obigen Beispiel eine durch den Anwender angegebene Funktion nicht und beendet die Verarbeitung des Ausdruckes mit der entsprechenden Fehlermeldung.

Beispiel: (ein interner Fehler der Haskell XML Toolbox)

```
xround' :: XFct
xround' _ _ [XPVNumber (Float f)] = ...
xround' _ _ [XPVNumber a] = XPVNumber a
xround' _ _ _ = XPVError "Call to xround' without a number"
```

Die Funktion *xround'* wurde mit einem falschen Argument aufgerufen. Dieser Fehler kann nicht aus einer falschen Eingabe des Anwenders resultieren, da Argumente automatisch in den richtigen Typ konvertiert werden. Eine sinnvolle Fortführung des Programms ist nach der fehlgeschlagenen Konvertierung allerdings nicht mehr möglich, so dass die Haskell XML Toolbox an dieser Stelle mit einer Fehlermeldung abbricht.

4.1 Kontext

Ein Ausdruck wird in XPath innerhalb eines Kontextes berechnet. Die Auswertungsumgebung besteht dabei aus den folgenden Elementen:

- einem Knoten (dem Kontextknoten)
- einem Paar von positiven ganzen Zahlen ungleich null (der Kontextposition und der Kontextgröße)
- einer Menge von Variablenbelegungen
- einer Funktionsbibliothek
- der Namensraumdeklaration, in deren Gültigkeitsbereich der Ausdruck liegt

Der Kontextknoten ist der aktuell zu verarbeitende Knoten. Die Kontextgröße entspricht der Anzahl aller zu verarbeitenden Elemente. Die Kontextposition stellt die Position des Kontextknotens in der Liste der gesamten Knoten dar und ist somit immer kleiner oder gleich der Kontextgröße. Die Werte dieser Parameter verändern sich bei der Verarbeitung eines Ausdruckes entsprechend des aktuellen Verarbeitungsabschnittes. Es gibt mehrere Arten von Ausdrücken, die den Kontextknoten ändern können. Die Kontextposition und die Kontextgröße werden allerdings nur durch Prädikate (siehe Abschnitt: 4.7.5) verändert.

Die Variablenbelegung, die Funktionsbibliothek und die Namensraumdeklaration ändern sich hingegen, während der Berechnung eines Teilausdruckes, nicht. Es handelt sich bei ihnen um Parameter, die dem XPath-Auswertungsmechanismus von außen[2] mitgeteilt werden. Im Modell der Haskell XML Toolbox werden sie daher nicht direkt dem Datentyp *Context* zugeordnet, sondern durch eigenständige Datentypen repräsentiert.

[2]beispielsweise Variablen, die in einem Stylesheet eines XSLT-Prozessors definiert sind

Beispiel: (Datentyp für den Kontext eines Ausdruckes)
```
type Context = (ConPos, ConLen, ConNode)
type ConPos  = Int
type ConLen  = Int
type ConNode = NavXmlTree
```

4.1.1 Variablen

Einem Variablennamen wird ein Variablenwert zugeordnet. Als Wert ist neben den vier Grundtypen jedes Objekt möglich, das Resultat eines Ausdruckes sein kann. Im Datenmodell entspricht eine Variable einem Tupel bestehend aus Name und Wert.

Beispiel: (Datentyp für Variablen und einige Variablendeklarationen)
```
type VarName = String
type Env     = [(VarName, XPathValue)]

varEnv       :: Env
varEnv       = [ ("numberStr", XPVString "3.4")
               , ("five", XPVNumber (Float 5))
               , ("inf", XPVNumber PosInf)
               , ("true", XPVBool True)
               , ("nodeSet", XPVNode aNodeSet)
               ]
```
Alle Variablen des Kontextes werden über den Datentyp *Env* zusammengefasst. Wird, während der Auswertung eines Ausdruckes, eine Variable referenziert, sucht die Funktion *getVariable* den passenden Namen in der Umgebung und liefert den entsprechenden Wert zurück. Durch den Parser werden alle syntaktisch korrekten Variablennamen erkannt. Es ist jedoch möglich, dass eine referenzierte Variable nicht in der Umgebung existiert. In diesem Fall wird das Programm mit einer Fehlermeldung beendet.

Beispiel: (Abbruch des Programmes mit einer Fehlermeldung)
```
getVariable :: VarName -> XPathValue
getVariable name
    = case lookup name varEnv of
        Nothing -> XPVError ("Variable: " ++ name ++ " not found")
        Just v  -> v
```
Das Definieren von Variablen ist Aufgabe des XPath nutzenden Prozessors. Innerhalb von XPath können Variablen nur verwendet, aber nicht erzeugt werden.

4.1.2 Funktionsbibliothek

XPath spezifiziert, neben den Grundtypen für das Ergebnis eines Ausdruckes, eine Reihe von Funktionen, die bei jeder Implementierung enthalten sein müssen. Diese Bibliothek der Grundfunktionen besteht aus der Abbildung eines Funktionsnamens auf eine Funktion. Die möglichen Argumente sowie das Ergebnis einer Grundfunktion, gehören den vier Grundtypen an. XPath nutzende Spezifikationen können zusätzliche Funktionen definieren, die auf weiteren Typen operieren.

Die Schnittstellenbeschreibung, die in den folgenden Abschnitten für die Funktionen verwendet wird, entspricht der XPath-Spezifikation. Das Fragezeichen ? kennzeichnet hierbei ein optionales Argument. Ein Argument mit dem Stern * darf keinmal, einmal oder mehrmals auftreten. Der Argumenttyp *object* steht für einen beliebigen Typ.

Hinweis: Die Funktionen werden der Vollständigkeit halber an dieser Stelle aufgeführt, wobei nicht alle zu beachtenden Sonderfälle angesprochen werden. Weiterführende Informationen sind in der XPath-Spezifikation [W3C 99] nachzulesen.

Funktionen auf Knotenmengen

- number **last**()
  Die Funktion **last** gibt die Kontextgröße des Auswertungskontextes zurück.
- number **position**()
  Die Funktion **position** liefert die aktuelle Kontextposition des Auswertungskontextes.
- number **count**(node-set)
  Die Funktion **count** berechnet die Anzahl der Elemente einer Knotenmenge.
- node-set **id**(object)
  Die Funktion **id** liefert eine Knotenmenge, die aus den Elementen besteht, deren ID mit dem Wert aus object übereinstimmt.
- string **local-name**(node-set?)
  Die Funktion **local-name** liefert den lokalen Namen des Knotens, der in der Knotenmenge an erster Position steht. Ist das Argument nicht vorhanden, wird der lokale Name des Kontextknotens zurückgegeben.
- string **namespace-uri**(node-set?)
  Die Funktion **namespace-uri** gibt den Namespace-URI des expandierten Namens des ersten Knotens in der Knotenmenge zurück. Wird das Argument weggelassen, so wird der Kontextknoten verwendet.
- string **name**(node-set?)
  Die Funktion **name** ermittelt den qualifizierten Namen des ersten Knotens der Knotenmenge. Ist das Argument nicht vorhanden, wird der Kontextknoten verwendet.

Zeichenkettenfunktionen

- string **string**(object)
  Die Funktion **string** berechnet den Zeichenkettenwert eines Objektes.
- string **concat**(string s1, string s2, string sn*)
  Die Funktion **concat** verknüpft ihre Argumente.
- boolean **starts-with**(string s1, string s2)
  Die Funktion **starts-with** liefert den Wert wahr, wenn die Zeichenkette s1 mit der Zeichenkette s2 beginnt.
- boolean **contains**(string s1, string s2)
  Die Funktion **contains** liefert den Wert wahr, wenn die Zeichenkette s1 die Zeichenkette s2 an einer beliebigen Position enthält.
- string **substring-before**(string s1, string s2)
  Die Funktion **substring-before** berechnet die Zeichenkette von s1 vor dem ersten Auftreten von s2 innerhalb von s1. Ist s2 nicht in s1 enthalten, wird die leere Zeichenkette geliefert.
- string **substring-after**(string s1, string s2)
  Die Funktion **substring-after** liefert die Zeichenkette von s1 nach dem ersten Auftreten von s2 innerhalb von s1. Ist s2 nicht in s1 enthalten, wird die leere Zeichenkette geliefert.

- string **substring**(string s1, number p, number n?)
  Die Funktion **substring** liefert die Teil-Zeichenkette von s1 ab der Position p[3], und zwar n Zeichen lang oder alle Zeichen ab der Position p, wenn n nicht angegeben wurde.
- number **string-length**(string s1?)
  Die Funktion **string-length** ermittelt die Länge der Zeichenkette s1. Ist kein Argument angegeben, wird der Zeichenkettenwert des Kontextknotens verwendet.
- string **translate**(string s1, string s2, string s3)
  Die Funktion **translate** ersetzt alle Zeichen in s1, die in s2 vorkommen, durch das Zeichen aus s3, das an der Stelle des Zeichens in s2 steht. Gibt es kein Zeichen in s3 an der Stelle des Zeichens in s2, wird das betrachtete Zeichen in s1 entfernt.

Boolsche Funktionen

- boolean **boolean**(object)
  Die Funktion **boolean** konvertiert ihr Argument in einen Wahrheitswert.
- boolean **not**(boolean b)
  Die Funktion **not** negiert b.
- boolean **true**()
  Die Funktion **true** liefert immer den Wert wahr.
- boolean **false**()
  Die Funktion **false** liefert immer den Wert falsch.
- boolean **lang**(string lng)
  Die Funktion **lang** gibt den Wert wahr zurück, wenn sich der Kontextknoten innerhalb der Sprache lng befindet.

Numerische Funktionen

- number **number**(object?)
  Die Funktion **number** konvertiert ihr Argument in eine Zahl.
- number **sum**(node-set)
  Die Funktion **sum** berechnet die Summe der Zahlenwerte aller Knoten in der Knotenmenge. Dabei wird jeder Knoten durch die **number**-Funktion in eine Zahl konvertiert.
- number **ceiling**(number)
  Die Funktion **ceiling** liefert die kleinste Zahl, die nicht kleiner als das Argument und eine ganze Zahl ist.
- number **floor**(number)
  Die Funktion **floor** liefert die größte Zahl, die nicht größter als das Argument und eine ganze Zahl ist.
- number **round**(number)
  Die Funktion **round** rundet ihr Argument zur nächsten ganzen Zahl.

4.1.3 Funktionsverarbeitung

Der erste Ansatz für das Auswerten einer Funktion *fctEval* besteht darin, über das Pattern Matching die richtige Funktion zu einem geparsten Namen zu wählen. Ist die Funktion vorhanden, wird der Funktionsrumpf ausgewertet und das Ergebnis zurückgegeben. Kann

[3] die Zählung der Zeichen beginnt bei eins

eine Funktion nicht in der Bibliothek gefunden werden, bricht das Programm mit einer Fehlermeldung ab.

Beispiel: (Erster Ansatz für die Funktionsverarbeitung)
```
fctEval :: String -> [Expr] -> XPathValue
fctEval "number" args = ...
fctEval "round"  args = ...
...
fctEval f _           = XPVError ("Call to unknown function: " ++ f)
```
Dieser Ansatz lässt die Erweiterung der Bibliothek um zusätzliche Funktionen allerdings nur schwerlich zu. Die Zuordnung eines Namens zu einer Funktion wird nicht zentral verwaltet. Der Einsatz einer Funktionstabelle behebt dieses Problem und erlaubt das einfache Hinzufügen von Funktionen.

Beispiel: (Datenmodell der Funktionstabelle)
```
type XFct     = (Context -> Env -> [XPathValue] -> XPathValue)
type FctName  = String
type FctTable = [(FctName, XFct)]
fctTable :: FctTable
fctTable = [ ("number", xnumber)
           , ("round", xround)
           , ...
           ]
```
Alle Funktionen besitzen in der Implementation der Haskell XML Toolbox die einheitliche Typdefinition *XFct*. Spezifikationen, die die Menge der Grundfunktionen erweitern wollen, müssen ebenfalls dieser Schnittstelle genügen.
Jede Funktion kann null oder mehr Argumente vom Typ *XPathValue* besitzen. Die Anzahl, der im Ausdruck übergebenen Argumente, muss mit der, durch die Funktion geforderten Anzahl, korrespondieren. Ist dies nicht der Fall, handelt es sich um einen Fehler. Das Datenmodell wird hierfür um die Typen *CheckArgCount* erweitert.

Beispiel: (Datenmodell für das Prüfen der Argumentanzahl)
```
type CheckArgCount = ([XPathValue] -> Bool)
zero, zeroOrOne, one :: CheckArgCount
zero ex       = length ex == 0
zeroOrOne ex  = length ex == 0 || length ex == 1
one ex        = length ex == 1

type FctTableElem = (XFct, CheckArgCount)
type FctTable     = [(FctName, FctTableElem)]

fctTable = [ ("number", (xnumber, zeroOrOne))
           , ("round", (xround, one))
           , ...
           ]
```
Zu jedem Funktionsnamen wird jetzt neben der Funktion auch die Anzahl der erwarteten Argumente gespeichert.
Der Wert einer Funktion wird bestimmt, indem erst alle Werte der Argumente berechnet und diese in den erwarteten Typ konvertiert werden. Anschließend erfolgt die Auswertung des Funktionsrumpfes. Die Konvertierung eines Argumentes in den Typ *string* erfolgt über den Aufruf der Funktion *string*. Die Funktionen *number* bzw. *boolean* erzeugen die entsprechenden Typen. Ein Argument kann nicht in den Typ *node-set* konvertiert werden. Erwartet eine Funktion eine Knotenmenge, muss das Argument von diesem Typ sein. Anderenfalls wird die Auswertung mit einer Fehlermeldung abgebrochen.

Beispiel: (Konvertierung der Argumente einer Funktion)

```
-- Funktion: string concat(string, string, string*)
xconcat :: XFct
xconcat c e args
 = XPVString (foldr (\(XPVString s) ->(s++)) [] (toXValue xstring c e args))
```

Die Funktion *concat* erwartet zwei oder mehr Argumente vom Typ *string*. Die Argumente werden erst über den Aufruf von `toXValue xstring` in den benötigten Typ konvertiert. Im Anschluss erfolgt die Berechnung des Ergebnisses und die Rückgabe als *XPVString*.

id-Funktion

Die *id*-Funktion benötigt für die Auswertung Informationen der DTD. Dies ist einer der wenigen Fälle, in denen das Ergebnis eines XPath-Ausdruckes von der Kenntnis der DTD des Dokumentes abhängt. Die Funktion *getIdAttr* extrahiert die benötigten Informationen vor der Reduzierung des XML-Baumes um die DTD. Sie liefert eine durch Leerzeichen getrennte Zeichenkette, die alle ID-Attribute der DTD enthält. Die Zeichenkette wird unter dem Variablennamen *idAttr* in der Liste der Variablen gespeichert. Die *id*-Funktion referenziert bei der Auswertung die benötigte Variable.

Beispiel: (Extrahieren der ID-Attribute aus der Variablenumgebung)

```
getIds :: Env -> [String]
getIds env
    = words $ (\ (XPVString str) -> str) . fromJust $ lookup "idAttr" env
```

4.1.4 Namensraumdeklaration

Das letzte benötigte Element für den Kontext eines Ausdruckes ist die Namensraumdeklaration. Namensräume werden von der Haskell XML Toolbox in der aktuellen Version jedoch nicht unterstützt, so dass auch innerhalb von XPath nicht mit ihnen gearbeitet werden kann. Alle Funktionen und Module, die auf Namensräume zugreifen, sind aber für die Verwendung vorbereitet und entsprechend gekennzeichnet. Sobald die Haskell XML Toolbox Namensräume unterstützt, ist auch der Einsatz im XPath-Modul möglich.

Beispiel: (Vorbereitung des XPath-Moduls für Namensräume)

```
-- |
-- evaluate a single XPath step
-- namespace-axis is not supported
evalStep :: Env -> XPathValue -> XStep -> XPathValue
evalStep _ _ (Step Namespace _ _)  = XPVError "namespace-axis not supported"
evalStep env ns (Step axisSpec nt pr)
    = evalStep' env pr nt (getAxisNodes axisSpec ns)
```

4.2 XPathFilter

Alle Funktionen, die einen Teilausdruck berechnen, sind als *XPathFilter* implementiert. Der Einsatz von Filtern bildet die Grundlage für die Verarbeitung von XML-Dokumenten (*XmlFilter*) in der Haskell XML Toolbox und wurde für das XPath-Modul übernommen. Da, im Gegensatz zur Verarbeitung von XML, als Ergebnis eines XPath-Ausdruckes nicht nur Knotenmengen möglich sind, war der Ansatz

```
type XPathFilter = NodeSet -> NodeSet
```

nicht ausreichend.

Ein *XPathFilter* muss jeden Ergebnistyp von XPath verarbeiten können:
`type XPathFilter = XPathValue -> XPathValue`

Der Typ *XPathValue* erlaubt die Konversion zwischen beliebigen Ergebnistypen, beispielsweise die Umwandlung einer Knotenmenge in eine Zeichenkette. Er enthält alle möglichen Resultattypen eines Teilausdruckes. Funktionen, die einen Teilausdruck berechnen, können somit über die Funktionskomposition verkettet werden. Dies ermöglicht vor allem im Bereich der Auswertung von Lokalisierungspfaden (siehe Abschnitt 4.7) eine sehr kompakte Implementation von Knotentests und Prädikaten.

evalExpr

Die Funktion *evalExpr* ist der zentrale *XPathFilter*.
`evalExpr :: Env -> Context -> Expr -> XPathFilter`

Sie enthält neben den notwendigen Umgebungsinformationen *Env* und *Context* den aktuell zu verarbeitenden Teilausdruck *Expr*. Durch das Pattern Matching werden die unterschiedlichen Teilausdrücke (GenExpr, FctExpr, usw.) den entsprechenden Verarbeitungsfunktionen zugeordnet.

Beispiel: (Ausschnitt aus der Funktion evalExpr)

```
evalExpr env cont (GenExpr Or ex)
    = boolEval env cont Or ex
evalExpr env cont (GenExpr And ex)
    = boolEval env cont And ex
...
evalExpr env cont (FctExpr name args)
    = fctEval env cont name args
```

Die folgende Tabelle stellt alle Teilausdrücke von XPath dar. Ihnen wird der entsprechende Datentyp des Modells für einen Ausdruck sowie die für die Auswertung verantwortliche Funktion zugeordnet.

Ausdrucksart	Typkonstruktor von Expr	Verarbeitende Funktion
boolscher Ausdruck	GenExpr mit den Operatoren Or, And	boolEval
relationaler Ausdruck	GenExpr mit den Operatoren NEq, Eq, Less, LessEq, Greater, GreaterEq	relEqEval
unärer Ausdruck	GenExpr mit dem Operator Unary	xPathUnary
numerischer Ausdruck	GenExpr mit den Operatoren Plus, Minus, Div, Mod, Mult	numEval
Mengenvereinigung	GenExpr mit dem Operator Union	unionEval
Funktion	FctExpr	fctEval
Lokalisierungspfad	PathExpr	locPathEval
Filter-Ausdruck	FilterExpr	filterEval
spezielle Ausdrücke	NumberExpr, LiteralExpr, VarExpr	evalSpezExpr

Tabelle 4.2: Auswertungsfunktionen für Teilausdrücke

Die speziellen Ausdrücke transformieren keinen Eingabewert in einen Ausgabewert. Sie liefern lediglich einen *XPathValue* zurück. Aus Gründen der einheitlichen Verarbeitung werden aber auch die speziellen Ausdrücke als *XPathFilter* implementiert.

Beispiel: (Ausschnitt aus der Funktion evalSpezExpr)

```
evalSpezExpr :: Expr -> XPathFilter
evalSpezExpr (LiteralExpr s) _ = XPVString s
...
```

Die nächsten Abschnitte beschreiben, wie die unterschiedlichen Teilausdrücke durch die Haskell XML Toolbox ausgewertet werden und welche Bedingungen der XPath-Spezifikation dabei zu berücksichtigen sind.

4.3 Boolsche Ausdrücke

Analog zur Strategie von funktionalen Programmiersprachen wird ein boolscher Ausdruck auch in XPath nicht strikt ausgewertet.

Beispiel: (Auswertung eines Oder-Ausdruckes)

```
boolEval :: Env -> Context -> Op -> [Expr] -> XPathFilter
...
boolEval env cont Or (x:xs) ns
    = case xboolean cont env [evalExpr env cont x ns] of
        e@(XPVError _) -> e
        XPVBool True   -> XPVBool True
        _              -> boolEval env cont Or xs ns
```

Wenn das Argument x über die Funktion *evalExpr* zu dem Wert wahr ausgewertet wird, oder zu einem Fehler führt, steht das Ergebnis des Oder-Ausdruckes fest und die Berechnung wird beendet. Es treten keine Seiteneffekte bei der Berechnung auf.

4.4 Numerische Ausdrücke

Die Berechnung eines numerischen Ausdruckes erfolgt nach den Regeln des IEEE 754 [IEEE 00] für die Operatoren Plus, Minus, Div und Mult. Der Mod-Operator berechnet nicht dasselbe wie die IEEE Rest-Operation, er verhält sich wie der Operator % in Java. Alle numerischen Berechnungen werden in XPath mit Fließkommawerten ausgeführt; es gibt keine Ganzzahl-Arithmetik.
Die im IEEE definierten speziellen Werte[4] sind durch das Sprachkonzept von Haskell nicht direkt abbildbar. Der Ausdruck *1 div 0* ergibt in XPath positiv Unendlich. Haskell würde diese Berechnung mit einem Programmfehler abbrechen. Aus diesem Grund war es notwendig, alle möglichen Kombinationen der speziellen Werte durch das Pattern Matching abzudecken.

Beispiel: (Ausschnitt aus der Funktion xPathMulti)

```
xPathMulti _ (XPVNumber (Float a)) (XPVNumber (Float b))
    = XPVNumber (Float (a * b))
xPathMulti _ (XPVNumber NegInf) (XPVNumber (Float a))
    | a < 0     = XPVNumber PosInf
    | otherwise = XPVNumber NegInf
xPathMulti _ (XPVNumber PosInf) (XPVNumber (Float a))
    | a < 0     = XPVNumber NegInf
    | otherwise = XPVNumber PosInf
...
xPathMulti a b c = xPathSpez a b c
```

[4] Pos0, Neg0, PosInf, NegInf, NaN

Um die Anzahl der zu beachtenden Kombinationen möglichst gering zu halten, wurden identische Fälle der verschiedenen Operatoren zusammengefasst. Dies ist allerdings nur für den Fehlerfall und das Ergebnis NaN möglich. Alle anderen Kombinationen lassen sich nicht sinnvoll zusammenfassen, da die Ergebnisse der Operatoren zu unterschiedlich sind.

Beispiel: (Zusammenfassen gleicher Kombinationen)

```
xPathSpez _ (XPVError e) _ = XPVError e
xPathSpez _ _ (XPVError e) = XPVError e
xPathSpez _ _ _            = XPVNumber NaN
```

4.5 Unärer Ausdruck

Das unäre Minus negiert sein Argument. Der spezielle Wert NaN kann nicht negiert werden und wird unverändert zurückgeliefert. Alle anderen Werte sind berechenbar.

Beispiel: (Ausschnitt aus der Funktion xPathUnary)

```
xPathUnary :: XPathFilter
xPathUnary (XPVNumber (Float f)) = XPVNumber (Float (-f))
xPathUnary (XPVNumber Pos0)      = XPVNumber Neg0
...
xPathUnary (XPVNumber NaN)       = XPVNumber NaN
```

4.6 Relationale Ausdrücke

Bei relationalen Ausdrücken werden drei Vergleiche unterschieden:

1. Vergleiche zwischen zwei Knotenmengen
2. Vergleiche zwischen einer Knotenmenge und einem beliebigen Typ
3. Vergleiche, an denen keine Knotenmenge beteiligt ist

4.6.1 Knotenmengen Vergleiche

Zwei Knotenmengen

Der Vergleich zweier Knotenmengen erfolgt über den Vergleich der Zeichenkettenwerte (siehe Abschnitt 3.1.1) der beteiligten Knoten. Er liefert den Wert wahr, wenn mindestens ein Knoten in der ersten Knotenmenge existiert, für den der Vergleich mit einem Knoten der zweiten Menge den Wert wahr ergibt. Dieses Verhalten gilt für alle Operatoren[5].

Eine Knotenmenge

Der Vergleich einer Knotenmenge mit einer Zeichenkette Z1 ist erfolgreich, wenn ein Knoten in der Menge existiert, für den der Vergleich seines Zeichenkettenwerts mit Z1 wahr ergibt. Wird eine Knotenmenge mit einer Zahl verglichen, wird der Zeichenkettenwert der Knoten über die Funktion *number* in einen numerischen Wert konvertiert und anschließend verglichen. Ein boolscher Wert wird identisch behandelt. Die Konvertierung erfolgt über die Funktion *boolean*.

[5] Eq, NEq, Less, LessEq, Greater, GreaterEq

Keine Knotenmenge

Ist keine Knotenmenge und als Operator entweder Eq oder NEq beteiligt, werden die beiden Teilausdrücke vor dem Vergleich in einen gemeinsamen Typ konvertiert.
Liefert dabei wenigstens einer der Teilausdrücke einen boolschen Wert, wird das Ergebnis des anderen Ausdruckes, über die Funktion *boolean*, in den boolschen Typ umgewandelt. Ist einer der Teilausdrücke vom Typ *number*, erfolgt der Vergleich über *number*. In allen anderen Fällen werden beide Teilausdrücke in Zeichenketten konvertiert.
Für die Anwendung der Operatoren Less, LessEq, Greater oder GreaterEq werden beide Teilausdrücke in Zahlen umgewandelt und numerisch verglichen.

4.6.2 Numerische Vergleiche

Um zwei Werte des Datentypen *XPNumber* numerisch vergleichen zu können, ist es notwendig, den Datentyp als Instanz der Klassen *Eq* und *Ord* zu definieren. Innerhalb der Instanz-Deklaration wird beschrieben, wie sich die speziellen Werte des IEEE zu den normalen Fließkommazahlen verhalten. Der Wert negativ Unendlich ist kleiner, positiv Unendlich größer als jede Fließkommazahl. Der Gleichheitstest für die Werte positiv und negativ Null ergibt den Wert wahr. NaN lässt sich mit keiner Zahl vergleichen, das Ergebnis des Tests ist immer falsch.

Beispiel: (Ausschnitt aus der Definition von Eq und Ord)

```
instance Eq XPNumber
    where
    ...
    NegO    == NegO    = True
    Float f == Float g = f == g
    PosInf  == PosInf  = True
    _       == _       = False -- NaN is always false

instance Ord XPNumber
    where
    a       <= b       = (a < b) || (a == b)
    a       >= b       = (a > b) || (a == b)
    a       >  b       =  b < a

    NaN     < _        = False
    ...
    Float f < Float g  = f < g
    _       < PosInf   = True
```

4.7 Lokalisierungspfade

Für die Auswertung von Lokalisierungspfaden ist es notwendig, innerhalb des XML-Baumes frei navigieren zu können. Die Berechnung der child-Achse benötigt eine Abwärtsbewegung im Baum. Die following-sibling- bzw. preceding-sibling-Achse ist eine Seitwärtsbewegung auf derselben Ebene. Eine Aufwärtsbewegung ist schließlich für die parent- oder ancestor-Achse erforderlich. Das Datenmodell der Haskell XML Toolbox erlaubt aber lediglich, den Baum abwärts zu traversieren. Eine Aufwärtsbewegung ist nicht möglich.

4.7.1 Navigierbare Bäume [English 02]

Das von Joe English im Rahmen von HXML[6] entwickelte Modul *navigable trees* stellt die benötigte Funktionalität zur Verfügung. Es basiert auf dem Datentyp *NavTree a*, welcher das gesamte XML-Dokument im Speicher hält.

Beispiel: (Datentyp für navigierbare Bäume)
```
data NavTree a = NT
                 (NTree a)    -- self
                 [NavTree a]  -- ancestors
                 [NTree a]    -- previous siblings (in reverse order)
                 [NTree a]    -- following siblings
```

Über die Funktionen
`upNT, downNT, leftNT, rightNT :: NavTree a -> Maybe (NavTree a)`
ist es möglich, sich durch den Baum zu bewegen. Zusätzlich definiert das Modul Funktionen, die die Knoten einer XPath-Achse berechnen. Es werden alle Achsen, mit Ausnahme der attribute- und namespace-Achsen, unterstützt.

Beispiel: (unterstützte XPath-Achsen im Modul navigable trees)
```
ancestorAxis, ancestorOrSelfAxis, childAxis, descendantAxis,
descendantOrSelfAxis, followingAxis, followingSiblingAxis,
parentAxis, precedingAxis, precedingSiblingAxis, selfAxis
  :: NavTree a -> [NavTree a]
```

Attribute-Achse

Das Verhalten der attribute-Achse ist implementationsabhängig und wird nicht exakt von XPath spezifiziert. Attribute sind in der Haskell XML Toolbox über die Kindknoten an einen *XTag* gebunden. Der Attributname ist der Wert des Typen *XAttr*. Der Attributwert wird als *XText* Knoten in den Kindern von *XAttr* gespeichert.

Beispiel: (Datentyp für Attribute in der Haskell XML Toolbox)
```
type AttrName = String
data XNode    = ...
              | XTag  TagName XmlTrees -- tag with list of attributes
                                       -- (inner node or leaf)
              | XAttr AttrName         -- attribute, attribute value
                                       -- is stored in children
```

Der allgemeine Typ `NavTree a -> [NavTree a]`, der im Modul *navigable trees* für die Achsen verwendet wird, kann nicht für die Definition der attribute-Achse genutzt werden. Um an die Attribut-Liste eines *XTag* zu kommen, ist es notwendig, den Typ *XNode*, anstatt des polymorphen Typen *a*, zu verwenden.

Beispiel: (Definiton der attribute-Achse)
```
attributeAxis :: NavTree XNode -> [NavTree XNode]
attributeAxis t@(NT (NTree (XTag _ al) _) a _ _)
    = foldr (\ attr -> ((NT attr (t:a) [] []):)) [] al
attributeAxis _ = []
```
Die namespace-Achse wird noch nicht durch die Haskell XML Toolbox unterstützt (siehe Abschnitt: 4.1.4).

[6] ein in Haskell geschriebener, nicht validierender XML-Parser

Konvertieren eines Baumes

Der Parser der Haskell XML Toolbox liefert das zu verarbeitende XML-Dokument als Baum vom Typ *NTree*. Vor der Auswertung eines XPath-Ausdruckes muss der XMLTree in einen navigierbaren Baum überführt werden. Dies erfolgt über die Funktion *ntree*, die ebenfalls vom Modul *navigable trees* bereitgestellt wird. Die Funktion *subtree* konvertiert den navigierbaren Baum wieder in einen XMLTree.

4.7.2 Relativer und absoluter Pfad

Die Auswertung eines relativen Pfades beginnt am aktuellen Kontextknoten. Für einen absoluten Lokalisierungspfad muss die Dokumentenwurzel berechnet werden, da die Auswertung eines absoluten Pfades immer von der Wurzel aus erfolgt.

Beispiel: (Auswertung eines Lokalisierungspfades)

```
locPathEval :: Env -> LocationPath -> XPathFilter
locPathEval env (LocPath Abs steps) = evalSteps env steps . getRoot
locPathEval env (LocPath Rel steps) = evalSteps env steps
```

Die Dokumentenwurzel wird über die Funktion getRoot berechnet. Sie traversiert innerhalb des Baumes aufwärts, bis die Wurzel erreicht ist.

Beispiel: (Berechnung der Dokumentenwurzel)

```
getRoot :: XPathFilter
getRoot (XPVNode (n:_))
    = XPVNode [getRoot' n]
      where
      getRoot' tree
        = case upNT tree of
            Nothing -> tree
            Just t -> getRoot' t
```

4.7.3 Lokalisierungsschritte

Ein Lokalisierungspfad besteht aus einer Liste von Lokalisierungsschritten (*XStep*), die nacheinander verarbeitet werden. Jeder Schritt berechnet über eine Achse eine Knotenmenge, die die Ausgangsmenge für den nächsten Schritt darstellt.

Beispiel: (Bestimmen der XPath-Achse)

```
evalStep :: XStep -> XPathFilter
evalStep (Step Child nt pred)  = evalStep' nt pred . getAxis childAxis
evalStep (Step Parent nt pred) = evalStep' nt pred . getAxis parentAxis
...
```

Dieser Ansatz führt zu einer Codeverdoppelung, da jede Achse über das Pattern Matching definiert sein muss. Der zweite Ansatz nutzt eine Tabelle, um eine Zuordnung zwischen einer Achse und der entsprechenden Achsenfunktion herzustellen.

Beispiel: (Bestimmen der XPath-Achse über eine Tabelle)

```
axisFctL :: [ (AxisSpec, (NavXmlTree -> NavXmlTrees)) ]
axisFctL =  [ (Ancestor, ancestorAxis)
            , (AncestorOrSelf, ancestorOrSelfAxis)
            , ...
            ]

evalStep :: Env -> XPathValue -> XStep -> XPathValue
evalStep env ns (Step axisSpec nt pr)
    = evalStep' env pr nt (getAxisNodes axisSpec ns)
```

4.7.4 Knotentests

Die durch eine Achse ausgewählte Knotenmenge wird durch einen Knotentest gefiltert. Die Achsenfunktionen stellen dabei sicher, dass sich nur Knoten des richtigen Hauptknotentyps in der zu filternden Menge befinden. Die attribute-Achse liefert nur Knoten vom Typ Attribut, alle anderen Achsen[7] haben als Hauptknotentyp den Elementtyp.
Der Namens- oder Typtest wird über Filterfunktionen der Haskell XML Toolbox realisiert. *isXTextNode* prüft beispielsweise, ob es sich bei dem Knoten um den Typ *XText* handelt. Alle anderen Knotentypen werden aus der Menge entfernt. Der Namenstest * sowie der Typtest *node()* werden über die *id*-Funktion abgebildet, da diese Tests für jeden Knoten der Menge erfüllt sind.

Beispiel: (Namens- und Typtest)

```
nodeTest :: NodeTest -> XPathFilter
nodeTest (NameTest "*") = id
nodeTest (NameTest s)   = filterNodes (isTagNode s)
nodeTest (PI s)         = filterNodes (isPiNode s)
nodeTest (TypeTest t)   = typeTest t

typeTest :: XPathNode -> XPathFilter
typeTest XPNode         = id
typeTest XPCommentNode  = filterNodes isXCmtNode
typeTest XPPINode       = filterNodes isXPiNode
typeTest XPTextNode     = filterNodes isXTextNode

filterNodes :: (XNode -> Bool) -> XPathFilter
filterNodes fct (XPVNode ns)
    = XPVNode ([n | n@(NT (NTree node _) _ _ _) <- ns , fct node])
```

Da das XPath-Modul auf navigierbaren Bäumen (NavTree) arbeitet, können Funktionen der Haskell XML Toolbox, die Mengen filtern, nicht verwendet werden. Diese Filter arbeiten auf dem Typ *NTree*. Die Funktion *filterNodes* selektiert daher aus dem Typ *NavTree* die *XNode*-Komponente, die durch einen Filter der Haskell XML Toolbox getestet werden kann.

4.7.5 Prädikate

Prädikate ermöglichen komplexere Filter als Knotentests. Ein Prädikat, das gegen eine Knotenmenge getestet wird, kann ein beliebiger Ausdruck sein.
Prädikate sind die einzigen Ausdrücke in XPath, die bei der Berechnung von Teilausdrücken

[7] die namespace-Achse wird nicht betrachtet

die Kontextposition und die Kontextgröße ändern. Alle anderen Ausdrücke wirken sich maximal auf den Kontextknoten aus. Vor der Auswertung eines Prädikats wird die Kontextgröße (`length ns`) bestimmt und die Kontextposition auf den Wert eins gesetzt.

Beispiel: (Bestimmen der Kontextgröße und Position)

```
evalPredL :: Env -> [Expr] -> XPathFilter
evalPredL env pr n@(XPVNode ns)
    = foldl (evalPred env 1 (length ns)) n pr
```

Die Funktion *evalPred* setzt diese beiden Werte und den aktuellen Kontextknoten zum Auswertungskontext zusammen. Der Test des Kontextknotens erfolgt durch *testPredicate*. Liefert *testPredicate* den Wert wahr, wird der Knoten in die Ergebnismenge aufgenommen. Anderenfalls wird die Kontextposition um eins erhöht und der Test mit dem nächsten Knoten der Menge als Kontextknoten fortgesetzt.

Beispiel: (Auswerten eines Prädikats)

```
evalPred :: Env -> Int -> Int -> XPathValue -> Expr -> XPathValue
evalPred env pos len (XPVNode (x:xs)) ex
    = case testPredicate env (pos, len, x) ex (XPVNode [x]) of
        XPVBool True   -> XPVNode (x : n)
        XPVBool False  -> nextNode
      where nextNode@(XPVNode n) = evalPred env (pos+1) len (XPVNode xs) ex

testPredicate :: Env -> Context -> Expr -> XPathFilter
testPredicate env context@(pos, _, _) ex ns
    = case evalExpr env context ex ns of
        XPVNumber (Float f) -> XPVBool (f == fromIntegral pos)
        XPVNumber _         -> XPVBool False
        _            -> xboolean context env [evalExpr env context ex ns]
```

Die Rückgabe der Funktion *testPredicate* wird durch die Berechnung des Teilausdruckes und anschließende Konvertierung des Ergebnisses in einen boolschen Wert bestimmt. Ist das Ergebnis des Teilausdruckes eine Zahl, entfällt die Konvertierung. Wenn diese Zahl der aktuellen Kontextposition entspricht, wird der Wert wahr zurückgegeben, sonst der Wert falsch. Der häufig benötigte Test, ob ein Knoten an einer bestimmten Stelle in der Knotenmenge vorkommt, wird hierdurch vereinfacht. Der Lokalisierungspfad *para[3]* ist äquivalent zu *para[position()=3]*.

4.7.6 Entfernen doppelter Teilbäume

Die Resultatmenge eines Lokalisierungsschrittes kann identische Teilbäume mehrfach enthalten. In der folgenden Abbildung entspricht der Knoten mit der Nummer 1 dem aktuellen Kontextknoten. Die child-Achse wählt die grau markierten Knoten aus. Wird auf die resultierenden Teilbäume die parent-Achse angewendet, wird der Knoten 1 insgesamt viermal in der Ergebnismenge vorhanden sein.

Vor der Auswertung des nächsten Lokalisierungsschrittes müssen gleiche Teilbäume entfernt werden. Zwei Teilbäume sind gleich, wenn die Anzahl ihrer Vorgänger (oder Nachfolger) auf jeder Ebene bis zur Dokumentenwurzel identisch ist.

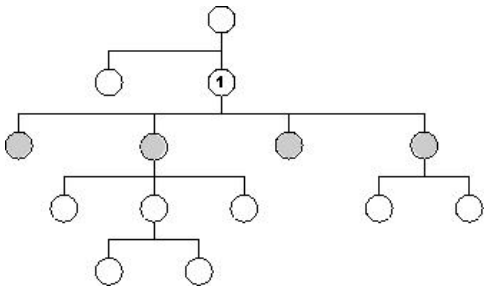

Abbildung 4.1: Doppelte Teilbäume

Beispiel: (Entfernen doppelter Teilbäume)

```
remDups :: XPathFilter
remDups (XPVNode (x:xs))
    | isNotIn x xs = XPVNode (x : y)
    | otherwise = remDups (XPVNode xs)

isNotIn :: Eq a => NavTree a -> [NavTree a] -> Bool
isNotIn n xs
    = getRelPosL (Just n) `notElem` map (\ x -> getRelPosL (Just x)) xs

getRelPosL :: Maybe (NavTree a) -> [Int]
getRelPosL (Just t@(NT _ _ prev _)) = length prev : getRelPosL (upNT t)
```

Die Funktion *getRelPosL* berechnet die Anzahl der Vorgänger eines Kontextknotens in einer Liste. Ist in der Resultatmenge kein Teilbaum mit einer identischen Liste vorhanden (*isNotIn*), wird der Teilbaum der Menge hinzugefügt.

4.8 Filter-Ausdrücke und Mengenvereinigung

Liefert ein Ausdruck eine Knotenmenge, wird sie, wie im Abschnitt 4.7.5 beschrieben, durch eine Liste von Prädikaten gefiltert. Ist das Resultat eines zu filternden Ausdruckes keine Knotenmenge, bricht die Verarbeitung mit einer Fehlermeldung ab.

Beispiel: (Verarbeitung eines Filter-Ausdruckes)

```
filterEval :: Env -> Context -> [Expr] -> XPathFilter
filterEval env cont (prim:predicates) ns
 = case evalExpr env cont prim ns of
   n@(XPVNode _) -> evalPredL env predicates n
   _             -> XPVError "Return of a filterexpression is not a nodeset"
```

Mengenvereinigung

Zwei Knotenmengen, die durch Filter-Ausdrücke oder Lokalisierungspfade ausgewählt wurden, lassen sich über den Union-Operator vereinigen. Da es sich bei der internen Darstellung der Mengen um Listen handelt, wird hierfür der Haskell Operator ++ verwendet. Eventuell vorhandene doppelte Teilbäume werden anschließend über die Funktion *remDubs* entfernt.

4.9 Darstellung eines Ergebnisses

Die Darstellung eines XPath-Ergebnisses als Text-Repräsentation erfolgt durch die Funktion *xPValue2String*. Die folgenden Beispiele beziehen sich auf das XML-Dokument aus Abschnitt 3.1. Die XPath-Ausdrücke decken alle fünf möglichen Ergebnistypen ab.

Beispiel: (Knotenmenge: rezept/zutat)

```
---XTag zutat
   |   id="mehl"
   |
   +---XText "200g Mehl"
```

Beispiel: (Zeichenkette: string(rezept/anleitung))

```
Zuerst nehmen Sie das
Mehl
und mischen es mit ...
```

Beispiel: (Boolscher Wert: $1 < 2$)

```
true
```

Beispiel: (Zahl: $2 + 4$ div 0)

```
Infinity
```

Beispiel: (Fehlerfall: 2 + myFunction())

```
Error: Call to undefined function myFunction
```

4.10 Modul-Hierarchie

Das Modul *XPath* exportiert alle notwendigen Funktionen für die Auswertung eines Ausdruckes. Es bildet damit den zentralen Einstiegspunkt. *XPathToString* stellt sowohl einen geparsten Ausdruck, als auch das Ergebnis des Ausdruckes dar. Im Modul *XPathFct* sind alle Funktionen der Grund-Bibliothek implementiert. *XPathArithmetic* enthält die arithmetischen Berechnungen nach IEEE 754. *XPathEval* fasst die Auswertung der verschiedenen Teilausdrücke zusammen. Das Modul *XPathDataTypes* enthält schließlich alle notwendigen Typ-Deklarationen für XPath.

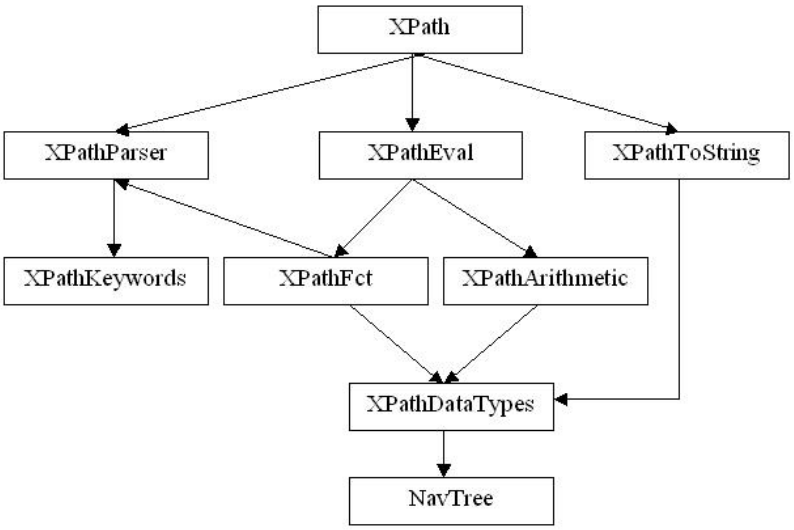

Abbildung 4.2: Modul-Hierarchie

5 Schlussbetrachtung

Das Ziel dieser Diplomarbeit bestand in der Konzeption und Implementation eines XPath-Moduls für die Haskell XML Toolbox.

5.1 Erreichtes

Der Entwurf des Parsers konnte vollständig abgeschlossen werden. Das entstandene Datenmodell ist in der Lage, einen beliebigen XPath-Ausdruck abzubilden. Die Verwendung der Parsec-Bibliothek erlaubte eine intuitive und zielgerichtete Umsetzung der XPath-Grammatik in einen Parser.
Das zu Beginn aufgetretene Problem der langsamen Verarbeitungsgeschwindigkeit konnte im Laufe der Entwicklung behoben werden. Es resultierte aus einem zu großen "look-ahead" des Parsers bei der Umsetzung einiger Grammatikregeln. Die aktuelle Version des Parsers reduziert den Einsatz eines "look-ahead" auf die Regel für Filter-Ausdrücke und die Schlüsselwörter der XPath-Grammatik.

Der Parser und das Datenmodell, das einen geparsten Ausdruck repräsentiert, sind weitestgehend unabhängig von der Haskell XML Toolbox implementiert. Es werden nur einige Hilfsfunktionen des XML-Parsers innerhalb von XPath verwendet. Dies ermöglicht, den Parser außerhalb der Haskell XML Toolbox Umgebung zu nutzen und ihn auf individuelle Anforderungen anzupassen.

Die Auswertung eines Ausdruckes konnte nicht vollständig erreicht werden, da die aktuelle Version der Haskell XML Toolbox XML-Namensräume nicht unterstützt. Alle anderen Teilausdrücke werden durch das XPath-Modul in vollem Umfang verarbeitet.

Die Entwicklung des Moduls erfolgte parallel zum Ausbau der Haskell XML Toolbox. Veränderungen an der Datenstruktur der Haskell XML Toolbox wurden von Herrn Prof. Dr. Schmidt zeitnah umgesetzt. Diese Notwendigkeit ergab sich vor allem im Bereich der Darstellung von Attributen. Sie repräsentieren innerhalb von XPath einen eigenen Knotentyp, der mit früheren Versionen der Haskell XML Toolbox nicht abbildbar war.

Das XPath-Modul wurde mit Hugs98 [Hugs 98] und dem Glasgow Haskell Compiler [GHC 02] in der Version 5.04 unter Linux- und Windows-Systemen entwickelt. Alle selbst geschriebenen Module lassen sich unter GHC mit dem Flag *-Wall* ohne Warnungen compilieren.

5.2 Konformität

XPath ist hauptsächlich für den Einsatz als Komponente innerhalb von umgebenen Prozessoren konzipiert worden. XPath überlässt es diesen Spezifikationen, Kriterien für die Konformität festzulegen und definiert selbst keinerlei Konformitätskriterien für eine unabhängige XPath-Implementation.

Das entwickelte Modul konnte aus diesem Grund nicht in einer standardisierten Testumgebung für XPath geprüft werden. Alle Tests mussten manuell durchgeführt und ausgewertet werden.
Die Kontrolle der Parser-Funktionalität erfolgte durch den XQuery Grammar Test [Grammar 02]. Er enthält über 1000 XPath-Ausdrücke, die alle vom Parser verarbeitet werden. Der Test beinhaltet allerdings keine Baum- oder XML-Darstellung des geparsten Ausdruckes, so dass das Ergebnis nicht automatisch verifiziert werden konnte. Der manuelle Vergleich, von circa 20 Prozent der Ausdrücke, liefert eine 100 prozentige Erfolgsquote.

Die Auswertung eines Ausdruckes wurde über die XSLT-Testsuite des National Institut of Standards and Technology [NIST 02] geprüft. Sie umfasst 200 Testfälle, von denen circa 50 Prozent erfolgreich durchlaufen werden. Ein Test der restlichen 100 Fälle kann nicht ohne XSLT-Prozessor durchgeführt werden.

5.3 Ausblick

Die Popularität von XML zur Darstellung von Informationen wird weiterhin zunehmen. Die einheitliche Repräsentation von XML-Dokumenten als Baum mit verschiedenen Knotentypen sowie die effiziente Filtertechnik zur Bearbeitung der Dokumente, versetzen die Haskell XML Toolbox in die Lage, den steigenden Anforderungen durch XML zu genügen.

Der Ausbau der Haskell XML Toolbox wird fortgesetzt. Namensräume können schon in der folgenden Version der Haskell XML Toolbox eingesetzt werden. Alle hierfür notwendigen Änderungen am Datenmodell wurden bereits durchgeführt.

Christine Apfel, Master-Studentin der Fachhochschule Wedel, hat in ihrer Thesis “Konzeption und Design eines XSLT Prozessors unter dem Aspekt der funktionalen Programmierung mit Haskell“ [Apfel 02] die Anforderungen an einen XSLT-Prozessor sehr exakt beschrieben. In absehbarer Zeit wird auf Basis der Arbeit von Frau Apfel und dem vorliegenden XPath-Modul die Implementation eines XSLT-Prozessors für die Haskell XML Toolbox erfolgen.
Durch die Integration des entwickelten XPath-Moduls in einen umgebenen Prozessor wird ein standardisierter Test möglich. Aus den Testergebnissen resultierende Optimierungen der Verarbeitungsgeschwindigkeit sowie die Korrektur eventuell vorhandener Fehler, werden weitere Aufgaben für die nahe Zukunft sein.

Literaturverzeichnis

[Apfel 02] Christine Apfel: *Master Thesis: Konzeption und Design eines XSLT Prozessors unter dem Aspekt der funktionalen Programmierung mit Haskell*, 2002

[Bach 00] Mike Bach: *XSL und XPath - verständlich und praxisnah*, Addison-Wesley, ISBN 3-8273-1661-8, 2000

[Becker 02] Oliver Becker: *Deutsche Übersetzung der XML Path Language (XPath) Version 1.0*, http://www.obqo.de/w3c-trans/xpath-de

[English 02] Joe English: *HXML*, http://www.flightlab.com/ joe/hxml/

[GHC 02] *Homepage des Glasgow Haskell Compilers*, http://www.haskell.org/ghc/

[Grammar 02] *The XPath 2.0 Grammar Test Page*, http://www.w3.org/2002/08/applets/xpathApplet.html

[Haskell 02] *Homepage von Haskell*, http://www.haskell.org/

[Hugs 98] *Homepage von Hugs98*, http://www.haskell.org/hugs/

[IEEE 00] Sun Microsystems: *The Java Language Specification*, http://java.sun.com/docs/books/jls/second_edition/html/

[NIST 02] NIST: *National Institut of Standards and Technology*, http://xw2k.sdct.itl.nist.gov/xml/page5.html

[Parsec 00] Daan Leijen: *Parsec: a free monadic parser combinator library for Haskell*, http://www.cs.uu.nl/∼daan/parsec.html

[Schmidt 02] Martin Schmidt: *Master Thesis: Design and Implementation of a validating XML parser in Haskell*, 2002

[Thompson 99] Simon Thompson: *Haskell: The Craft of Functional Programming, Second Edition*, Addison-Wesley, ISBN 0-201-34275-8, 1999

[Toolbox 02] *Homepage der Haskell XML Toolbox*, http://www.fh-wedel.de/∼si/HXmlToolbox/index.html

[W3C 99] World Wide Web Consortium: *XML Path Language (XPath) Version 1.0*, http://www.w3.org/TR/xpath

Weitere Quellen

In diesem Abschnitt werden die nicht in das Literaturverzeichnis passenden Quellen aufgeführt, die aber mindestens genauso viel zu dieser Arbeit beigetragen haben, wie die dort aufgeführten.

LaTeX

Helmut Kopka: *LaTeX: Eine Einführung*, Addison-Wesley, ISBN 3-89319-199-2, 1990

Wörterbücher

- *Die deutsche Rechtschreibung*, DUDENVERLAG, ISBN 3-411-04011-4, 1996
- *Oxford Advanced Learner's Dictionary*, Cornelsen Verlag, ISBN 3-464-11223-3, 1995
- *Dictionary LEO*, http://dict.leo.org

Antworten auf alle möglichen (mehr oder weniger) fachlichen Fragen gaben:

- Prof. Dr. Uwe Schmidt, Fachhochschule Wedel
- Dipl.-Ing.(FH), MSc. Henry Kleta, Fachhochschule Wedel
- Andreas Ehlers

Korrekturleser:

- Sabrina Türke
- Sven Dietze
- Wiebke Leander

seid bedankt...

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Hamburg, 20.02.2003

(Torben Kuseler)