Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

XPath-Unterstützung in einer Linux Shell

Semesterarbeit
SS 06

Kaspar Giger
April 2006

Betreuer: Dr. Erik Wilde
Professor: Prof. Bernhard Plattner

XPath-Unterstützung in einer Linux Shell

Mit XPath gibt es eine Sprache, mit der man auf Bäumen (genauer gesagt Infosets, der Abstraktion von XML-Dokumenten) Knoten selektieren kann. XPath funktioniert ähnlich Pfaden in Filesystemen, bietet aber einen deutlich grösseren Funktionsumfang. Überträgt man einige oder alle der Konzepte von XPath aus der Welt von XML-Dokumenten in die Welt von Filesystemen, so könnte die Leistungsfähigkeit gängiger Pfadausdrücke in Unix Shells massiv gesteigert werden. Für viele Problemstellungen wären Tools wie das find Kommando dann überflüssig, sie könnten direkt in "XPath"gelöst werden. In dieser Arbeit soll untersucht werden, inwieweit XPath für dieses Ziel verwendet werden kann, und anhand einer Implementierung in einer Linux-Shell der Wahl soll das Konzept getestet werden.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 XML und das Unix Dateisystem

Eines der zentralen Konzepte von XML ist, dass jede XML-Datei ein Baum ist. Auch das Unix Dateisystem ist ein Baum, mit dem Root-Verzeichnis `/` als Wurzelknoten und Dateien und Ordnern als "XML-Elemente". Daher ist es naheliegend, Konzepte, die sich in der XML-Welt erfolgreich etablieren konnten auch auf das Unix Dateisystem anzuwenden.
Der erste Schritt dazu ist, das Dateisystem auf eine XML-Struktur abzubilden. Dazu werden die Ansätze aus [9] verwendet. Anschliessend kann man sich praktisch beliebige Möglichkeiten für dessen Weiterverwendung denken.
Eine Anwendung ist die für XML 1.0 definierte Sprache XPath. Mit XPath lassen sich in einer XML-Baum-Struktur beliebige Knoten selektieren, ähnlich der Dateisuche in Verzeichnissen. Mit XPath lassen sich fast beliebig komplexe Anweisungen schreiben. So ist es z.B. möglich verschachtelte Bedingungen aufzustellen oder in Ordnern rekursiv zu suchen.

1.2 FSX - XML Filesystem

Wie in [9] erklärt, lassen sich praktisch beliebige Dateisysteme, in eine XML Struktur umwandeln. Einzige Bedingung ist, dass die Dateien und Ordner baumartig miteinander verknüpft sind – was bei den bekannten Lösungen, wie sie unter Unix und Windows verwendet werden der Fall ist.
In FSX wird ein zentraler Knotentyp definiert: `fsx:node`. Abgeleitet von diesem node-Knotentyp existieren die drei Knotentypen `fsx:file`, `fsx:dir` und `fsx:symlink`. Mit diesen drei Arten von Knoten lassen sich nun alle Einträge in einem Dateisystem repräsentieren. Theoretisch würde `file` reichen, da beispielsweise unter Unix sowieso der Grundsatz herrscht: *"Everything is a file"* (*Alles ist eine Datei*). Um jedoch klar zu trennen (wie es z.B. bei Windows gemacht wird), wurden die zwei Typen `symlink` und `dir` eingeführt. Jeder Knoten besitzt nun mindestens die folgenden Attribute:

- `name`: Name der Datei, des Symlinks oder Ordners (=Knoten)
- `mtime`: Zeitpunkt, wann der Knoten zuletzt verändert wurde
- `canread`: Ob der Benutzer Leserecht dieses Knotens besitzt
- `hidden`: Ob der Knoten versteckt ist

Zusätzlich zu diesen vier Grundattributen besitzen Dateien die Attribute

- `size`: Grösse der Datei
- `regular`: Ob die Datei eine reguläre Datei ist (Ordner oder Device-dateien sind z.B. nicht regulär)
- `mime`: Was für einen MIME Typ die Datei beinhaltet
- `dev`*: Identifikator für das Gerät, auf dem die Datei liegt
- `ino`*: Die Seriennummer der Datei, welche sie von den anderen Dateien unterscheidet, die auf diesem Gerät liegen
- `nlink`*: Die Anzahl Hard-Links zu dieser Datei (Symlinks werden hier nicht mitgezählt)
- `uid`*: Die User ID des Benutzer, dem diese Datei gehört
- `gid`*: Die Group ID der Gruppe, welcher diese Datei gehört
- `blksize`*: Die optimale Blockgrösse in Bytes um von dieser Datei zu lesen oder in diese Datei zu schreiben
- `blocks`*: Der Platz, den die Datei braucht, gemessen in 512-Byte Blöcken
- `atime`*: Der Zeitpunkt des letzten Zugriffs
- `ctime`*: Der Zeitpunkt der letzten Datei-Attribut-Änderungen
- `mode`*: Zugriffsrechte der Datei
- `sticky`*: Das Sticky Bit der Datei (gesetzt oder nicht, '1' oder '0')
- `rdev`*: Typ des Geräts (wenn die Datei ein Gerät darstellt)
- `blockdev`*: Ob die Datei ein Block-Device ist
- `chardev`*: Ob die Datei ein Character-Device ist
- `fifo`*: Ob die Datei eine Named Pipe ist
- `socket`*: Ob die Datei ein Socket ist

Ähnliche Attribute besitzen auch symbolische Links (`symlink` Knoten). Bloss können sie keine Geräte symbolisieren, womit sie die Attribute rdev, blockdev, chardev, fifo und socket nicht besitzen. Und da die Grösse bei symbolischen Links auch nur eine untergeordnete Rolle spielen, wird auf das size Attribut ebenfalls verzichtet. Weiter kann ein symbolischer Link auch keinen Inhalt haben (ausser dem Target natürlich), womit das MIME-Typ Attribut hinfällig wird. Zusätzlich bekommen die Symlink-Knoten aber noch einen Eintrag für das Ziel des Verweises:

- `target`: Ziel des symbolischen Links

`dir` Knoten besitzen die gleichen Attribute wie Symlinks nur, haben sie zwar kein Ziel, also kein `target` Attribut, dafür aber im Gegensatz zu den Verweisen das Sticky Bit Attribut (`sticky`).

*diese Attribute stehen nur unter Unix zur Verfügung

1.2.1 Integration von Meta-Daten

Mit dem bisherigen Ansatz kann ein gesamtes Dateisystem auf eine XML-Struktur abgebildet werden. Erweitert wird dieser Baum nun um Metainformationen der verschiedenen Knoten. Zu jedem Knoten könnte man sich zusätzliche interessante Informationen vorstellen, die mit den bisher vorgestellten Attributen nicht abgedeckt werden, aber wichtig sein könnten. Solche Zusatz-Informationen könnten z.B. Einträge eines Versionsverwaltungssystems (SVN, CVS, ...) oder erweiterte Zugriffsrechte eines Verzeichnisdienstes (LDAP, ...) sein. Diese Zusatzinformationen sind weder Inhalt der Dateien/Ordner noch direkt Basis Attribute, womit sich ihre Sonderbehandlung rechtfertigen lässt. Es geht nicht primär um Inhalte einer Datei oder eines Ordners, sondern bloss um dessen (erweiterte) Eigenschaften. Diese Metadaten werden angehängt an einen `metadata` Kind-Knoten jedes Dateisystem-Knotens angeführt.
Dieser Metadata-Unterbaum kann selbst ein vollständiger XML-Baum sein, wie auch nur Attribute besitzen.

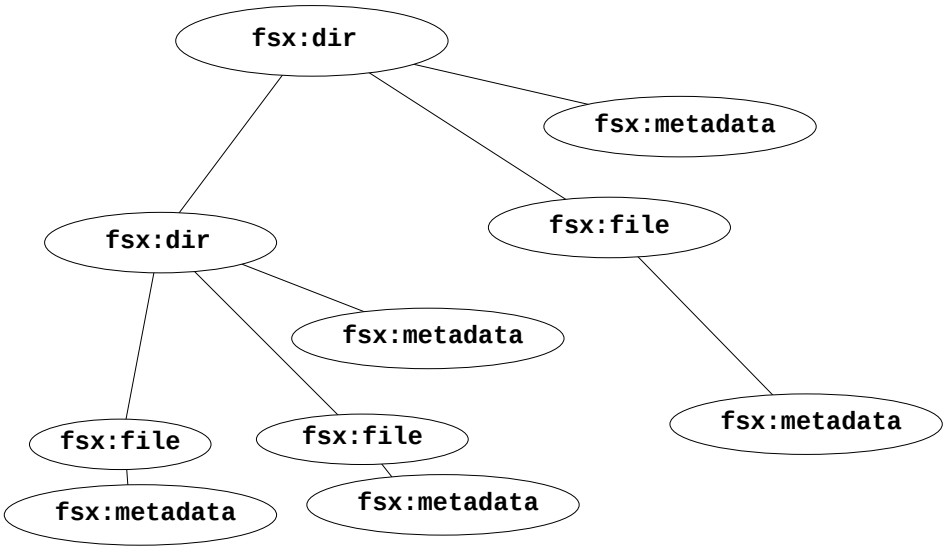

Abbildung 1.1: Beispiel einer Verzeichnisstruktur inklusive Metadaten, mit einem Root-Ordner, der eine Datei und einen Ordner beinhaltet, welcher selber zwei Datein beinhaltet. Jeder Knoten stellt ein XML-Element dar.

1.2.2 Integration von Datei-Inhalten

Abgerundet wird das gesamte Konzept durch die Integration von Datei-Inhalten in den FSX-Baum. Erreicht wird das durch Hinzufügen eines `content` Kind-Knoten zu jedem `file` Knoten. Wie der Metainformationen-Unterbaum, kann der Content-Unterbaum einen kompletten XML-Baum beinhalten, inklusive Definitionen von neuen Namespaces.

1.2.3 Adaptoren

Diese Integration von Datei-Inhalten und Meta-Informationen in den *FSX*-Baum wird nicht direkt von *FSX* vorgegeben, sondern kann dynamisch von sogenannten Adaptoren durchgeführt werden. Anhand des Dateityps (z.B. MIME-Typ) wird bestimmt, welcher Adaptor

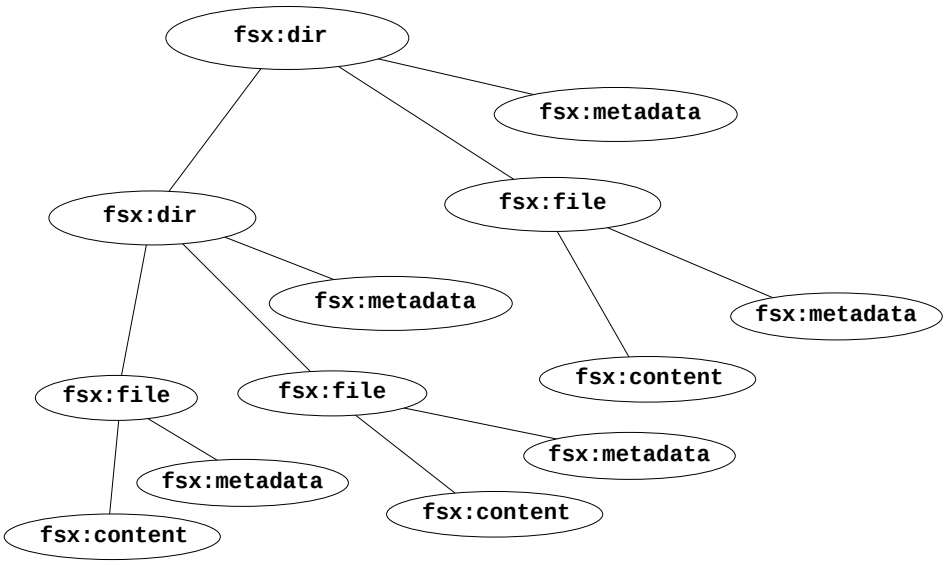

Abbildung 1.2: Die gleiche Verzeichnisstruktur wie in Abbildung 1.1, erweitert um die `content` Knoten. Jeder Knoten stellt ein XML-Element dar.

verwendet werden soll, um das bestmögliche Resultat zu erzielen. Auf die Adaptoren und ihr Interface wird im Kapitel 7 weiter eingegangen.

1.3 XPath und Shells

Die Aufgabe von XPath in XML ist die gleiche wie Pfadausdrücke in Shells auf Betriebssystem-Seite: Die Selektion von einzelnen Knoten, bzw. Dateien und Ordnern (was ja entpsrechend dem FSX-Konzept das gleiche bedeutet). Dadurch wurde die Syntax von XPath und deren Semantik stark an Shell-Pfadausdrücke angelehnt.

1.3.1 Unterschied zwischen XPath und Shell-Pfadausdrücken

Laut Definition der Shell-Kommando-Sprache [5] lassen sich Knoten (Dateien/Ordner) mit Wildcard-Ausdrücken selektieren. Folgende Wildcards sind erlaubt:

- `*`: Passt auf eine beliebig lange Folge von beliebigen Zeichen
- `?`: Passt auf ein beliebiges Zeichen
- `[`*Zeichenmengendefinition*`]`: Passt auf ein Zeichen, das in der Zeichenmenge enthalten ist
- `[!`*Zeichenmengendefinition*`]`: Passt auf ein Zeichen, das nicht durch die Zeichenmenge abgedeckt wird

Diese Wildcards lassen sich beliebig kombinieren. Im Gegensatz dazu erlaubt XPath nur den Asterisk `*` Operator, allerdings nur exklusiv, also nicht in Kombination mit weiteren Zeichen oder gar Wildcards. Wenn nun die Knoten mit ihren Dateinamen benannt würden, wäre es

deshalb trotzdem nicht möglich, in einem Location Path den Namenstest `*.txt` zu schreiben (um z.B. Text-Dateien zu selektieren), was in Shells ein Selbstverständlichkeit darstellt.
XPath bietet hingegen zahlreiche Konstrukte, die in Shell-Ausdrücken entweder nicht erlaubt sind oder gar nicht vorkommen. Ein mächtiges Konzept sind z.B. die Prädikate, welche Knoten anhand wahr/falsch Bedingungen selektieren.
Nur ganz einfach werden in Shells Achsen verarbeitet. Im Gegensatz zu XPath, welches beliebige Suchrichtungen zulässt. In einer Shell gibt es genau drei Achsen:

- `.` → `self::node()`
- `..` → `parent::node()`
- *Verzeichnisname* → `child::` *Verzeichnisname*

Eine der zentralen Aufgabe des XPath-Shell Projekts ist es, diese drei Grundachsen mit einigen von XPath zu erweitern.

1.3.2 Kombination XPath und Shellausdrücke

Um nun die beiden Konzepte von XPath und den Shell-Pfadausdrücken kombinieren zu können, wird eine neue Sprache eingeführt. Diese Sprache ist sehr stark an XPath angelehnt, erweitert um Wildcard-Ausdrücke.
Um zu demonstrieren, dass XPath direkt keine hohe Benutzerfreundlichkeit erreichen würde sei folgender Vergleich angestellt. Um die Datei `testfile` im Unterordner `testdir` zu selektieren, wird in einer Shell `testdir/testfile` eingegeben. Um das gleiche in einem *FSX*-Baum mit korrektem XPath zu erreichen, müsste
`fsx:dir[@name="testdir"]/fsx:file[@name="testfile"]` geschrieben werden. Wenn das Ganze dann noch um Wildcards erweitert werden soll, ist das Ergebnis nicht mehr alltagstauglich.
Aus diesem Grund wird eine neue Sprache nötig. Wie in Kapitel 3 beschrieben, wird die neue Sprache durch eine Abänderung der XPath-EBNF-Grammatik ebenfalls mit einer kontextfreien Grammatik definiert.

1.3.3 XPath Shell

Das Ergebnis mit dem Zwischenschritt über eine neue Sprache wird im folgenden als XPath Shell, kurs XPsh, bezeichnet. Es ist jedoch keine reine XPath Shell, da wie in Kapitel 1.3.2 erläutert, keine reinen XPath Ausdrücke erlaubt sind.

Kapitel 2

Syntax Auswertung

Die zentrale Funktion der XPath Shell ist, XPath- oder XPath-ähnliche Ausdrücke auszuwerten und daruf ein Nodeset mit den Dateien zurückzugeben. Dazu gibt es mehrere Möglichkeiten, wie in den Kapiteln 2.2.1-2.2.3 erklärt wird.

2.1 Interpretation

Wie in Kapitel 1.3.2 beschrieben wird, sollten als Pfadausdrücke keine kompletten XPaths erlaubt werden, sondern nur XPaths in einer angepassten, abgekürzten Form. Um die Ausdrücke auszuwerten gibt es zwei Möglichkeiten, entweder wird die Eingabe in einen korrekten XPath umgewandelt und dann einer XPath-Prozessor übergeben 2.1.2 oder die Interpretation erfolgt komplett ohne Grammatik, durch stufenweise, rekursive Abarbeitung 2.1.1.

2.1.1 Stufenweise, rekursive Interpretation

Das Grundkonzept einer stufenweisen, rekursiven Interpretation eines Ausdrucks ist, die stufenweise Auftrennung der Location-Steps (Verzeichnis-Tiefen-Stufen) beim Auftreten eines einfachen Slash (`/`). Damit könnten der Location-Step und die Prädikate relativ einfach herausgefiltert und durch einen kleine Parser interpretiert werden. Die Vorteile, die diese Umsetzung der Eingabe-Verarbeitung hat, werden in Kapitel 2.2.2 erklärt.
Damit Konstrukte, wie der Doppelslash (`//`) oder die `ancestor-or-self`-Achse verarbeitet werden können, muss die Abarbeitung der Eingabe rekursiv erfolgen.
Um die Eingabe-Grammatik nicht unnötig einfach halten zu müssen, müssten auch verschachtelte Konstrukte verarbeitet werden können, insbesondere in Prädikaten. Da dies darauf hinauslaufen würde, dass praktisch ein vollständiger XPath-Prozessor entwickelt würde, wird hier das Konzept der stufenweisen, rekursiven Interpretation der Ausdrücke nicht weiterverfolgt.

2.1.2 Mapping

Der erfolgsversprechendere Ansatz ist, eine vollständige Grammatik zu definieren und mittels eines Parser-Generators die Eingaben zu parsen. Auf die Realisierung des Parsers wird im Kapitel 3 weiter eingegangen.
Anschliessend gibt es zwei Möglichkeiten: Entweder wird der geparste Ausdruck direkt intepretiert und ausgewertet oder er wird zuerst in einen korrekten XPath umgewandelt, der von einem XPath-Prozessor verarbeitet werden kann.

2.2 Auswertung

Die eigentliche Auswertung des zuvor geparsten oder, wie im Falle der stufenweisen, rekursiven Interpretation, vorverarbeiteten Ausdrucks lässt verschiedene Möglichkeiten offen, wie in den nächsten Kapiteln beschrieben wird.

2.2.1 Statisch

Eine erste Idee ist, das gesamte Dateisystem in eine XML-Datei oder zumindest in eine DOM-Repräsentation umzuwandeln. Auf diesen DOM-Baum wird anschliessend der korrekte XPath angewandt.
Diese Lösung ist absolut korrekt, macht genau, was sie soll hat aber einige Nachteile. Zum einen, wie der Titel schon sagt, ist diese Verfahren absolut statisch. Der Auswerter ist darauf angewiesen, dass eine DOM-Repräsentation des Dateisystems vorliegt. Doch es wäre unsinnig, vor jedem erneuten Aufruf, das gesamte Dateisystem abzuarbeiten und jeder einzelne Ordner und jede einzelne Datei in den DOM-Baum einzufügen. Darum müsste diese Lösung mit einer Datenbank-ähnlichen DOM-Struktur arbeiten, die sich in bestimmten Abständen aktualisiert. Ähnich also, wie es z.B. das GNU `locate` Programm macht, bei dem eine zentrale Datenbank alle Dateien speichert und die mit `updatedb` auf den neusten Stand gebracht werden kann.
Ein weiterer Nachteil dieses Ansatzes ist der grosse Speicherverbrauch. Es werden jeweils grosse Mengen Knoten gespeichert, die schliesslich gar nie gebraucht werden. Zudem würde bereits schon die Integration von Inhalten und Meta-Daten zu einer starken Vergrösserung des Speicherverbrauchs führen.
Somit ist diese Lösung zwar korrekt, aber aus Implementierungs-Sicht nicht weiter sinnvoll.

2.2.2 Step-By-Step

Eine zweite Möglichkeit wäre, wie schon weiter oben in Kapitel 2.1.1 angesprochen wurde, die verschiedenen Location-Steps rekursiv zu durchlaufen.
Ein grosser Vorteil dieses Ansatzes ist sicher die hohe Effizienz. Es werden nur genau die Knoten erstellt/durchlaufen, die auch zwingend notwendig sind. Zudem könnte genau bestimmt werden, welche durchlaufenen Knoten noch gebraucht werden und welche bedenkenlos aus dem Speicher gelöscht werden können, was einen sparsamen Umgang mit den Ressourcen zur Folge hätte. Zwar könnte es durchaus vorkommen, dass Knoten gelöscht und dann später wieder erzeugt werden, doch ist dies dem sparsamen Speicherverbrauch nicht abträglich, es bleibt stets nur ein Minimum an Knoten im Speicher. Aus Implementierungs-Sicht ist dies daher ein recht erfolgsversprechender Ansatz, wenn die Interpretation der Grammatik gut gelöst werden kann.

2.2.3 Halb-dynamisch

Der dritte Ansatz basiert auch auf einem Abbilden des Eingabe-Ausdrucks mittels eines Parsers auf einen korrekten XPath. Für dessen Auswertung wird ein beliebiger XPath-Prozessor benutzt, dem ein XML-Pull-Parser unterlegt wird. Der Pull-Parser erzeugt die Knoten jeweils dynamisch und übergibt sie anschliessend dem XPath-Prozessor zur weiteren Betreuung.
Um zu verhindern, dass ein Knoten mehrmals erzeugt wird, wird ständig Buch geführt, welche Knoten schon im Speicher existieren und die je nach Gebrauch auch gleich wieder zurückgeben werden können.

Der Nachteil dieses Ansatzes ist sicher, dass die Lösung zwar korrekt ist, aber je nachdem auch ineffizient. Wenn der XPath-Prozessor die Eingaben ineffizient verarbeitet, gibt es praktisch keine Möglichkeit dies zu verändern. Ausser man würde auch diesen Prozessor selber implementieren. Da aber davon ausgegangen werden kann, dass die meisten solchen XPath-Bibliotheken einigermassen effizient arbeiten, fällt diesem Argument keine grosse Bedeutung zu.
Ein gewichtiger Nachteil ist, dass der DOM Standard [6] zwar vorgibt, welche Funktionen gebraucht werden, um z.B. ein Kind- oder ein Eltern-Knoten des aktuellen Kontext-Knoten zu erhalten. Aber er behandelt das Thema Speicherverwaltung nicht, sondern schiebt es der DOM-Implementation zu. Somit kann während der Abarbeitung eines XPath-Ausdrucks kein Knoten gedankenlos gelöscht werden. Eine mögliche Lösung dieses Problems wäre, dass der XPath-Prozessor vorausblickt und erkennt, welche Knoten noch gebraucht werden und welche nicht und so eine weitere Funktion einführt, die das Löschen eines Nodesets bewirken kann (z.B. als eine Art DOM Memory-Management Erweiterung).

2.3 Implementierung

In der vorliegenden wurde Arbeit der halb-dynamische Ansatz gewählt, wie in Kapitel 2.2.3 beschrieben. Als Parser-Generator wurden die Unix Werkzeuge *Yacc* und *Lex* verwendet, als XPath-Prozessor das Projekt Sablotron [7].

Kapitel 3

XPsh Syntax

3.1 Syntax-Beschreibung

Um eine möglichst hohe Anwederfreundlichkeit zu erlangen, sollte die neue Sprache zwar beliebig komplexe Konstrukte erlauben aber sehr einfach sein. Darum werden die Wildcard-Ausdrücke, bekannt aus Shells, übernommen und die Syntax an XPath angelehnt.

3.1.1 Achsen

In XPath existieren dreizehn verschiedene Achsen, die nicht alle für eine Shell geeignet sind oder Sinn machen. Übernommen werden die `child`-, `self` und `parent`-Achsen, die in Shells bereits schon existieren (siehe Kapitel 1.3.1). Neu hinzu kommen nun noch die `descendant-or-self`-, die `attribute`- und die `ancestor`-Achsen.
Um die Syntax einfach zu halten, sind ausgeschriebene Lokalisierungspfade bewusst nicht erlaubt. Die Achsen existieren somit nur noch in einer abgekürzten Form:

`child::`*abc*	→ *abc*
`parent::node()`	→ `..`
`self::node()`	→ `.`
`descendant-or-self`	→ `//`
`ancestor`	→ `...`[1]
`attribute`	→ `@`

3.1.2 Dateien und Ordner

Dateien und Ordner werden, wie von XPath bekannt, mittels Namenstest selektiert. Das Problem, das damit entsteht ist, dass unter den meisten Betriebssystemen Dateien Namen tragen können, die keine XML-Namen sind. Z.B. wären `.abc`, `123.txt` oder `a b.c` keine gültigen XML-Namen aber durchaus zulässige Dateinamen. Dies ist aber bloss eine Erweiterung von XPath, keine Einschränkung. Wie in der Shell Kommando Sprache definiert, müssen dabei die *Spezial Zeichen* (wie z.B. `!`, `*`, *Leerzeichen...*) *escaped* werden, also mit einem vorangehenden Backslash (\).

[1] Existiert in XPath noch nicht, wurde hier zusätzlich eingeführt

3.1.3 Wildcards

Analog zu Shell-Pfadausdrücken sind die vier in Kapitel 1.3.1 vorgestellten Wildcards erlaubt. Damit entstehen zwei Probleme.
Da wäre der Asterisk (`*`), das in XPath sowohl als Knotentest, wie auch als Multikplikationsoperator verwendet wird. Da eine der wichtigsten Shell-Wildcards nicht verändert werden sollte, wird als Lösung vorgeschlagen, den Multiplikationsoperator neu auszuschreiben. Also statt '`1*1`' neu '`1 mult 1`' zu schreiben. Wichtig dabei ist, vor und nach dem mult ein nicht-escapedes Leerzeichen zu schreiben, sonst würde der Ausdruck wieder als Namenstest erkannt und falsch verarbeitet.
Die andere Zweideutigkeit betrifft die eckige Klammer, die sowohl in XPath für Prädikate, wie auch bei Shell-Ausdrücken als Wildcard verwendet wird. Als Lösung wird vorgeschlagen die wenig verwendete Wildcard neu mit einer doppelten eckigen Klammer zu schreiben und eine eckige Klammer für Prädikate zu verwenden. Damit bleibt allerdings eine gewisse Inkompatibilität zu Shell-Pfadausdrücken, die aber marginal bleiben dürfte, weil diese Wildcard selten eingesetzt wird. Auf der anderen Seite wäre es unnötig, die oft verwendeten Prädikate zu verlängern[2]. Auch hier ist wichtig, dass sowohl für Prädikate, wie auch für diese Shell-Wildcards keine escapeden eckigen Klammern verwendet werden, da das sonst zu einer Erkennung als Namenstest führen würde.

3.1.4 Zahlen

Wie schon vorher angesprochen, sind in Dateisystem-Pfadausdrücken Ziffern an beliebigen Stellen erlaubt. Somit kann man nicht direkt unterscheiden, ob eine Ziffernfolge eine Zahl ist oder ein Namenstest. Um in XPath trotzdem mit Zahlen arbeiten zu können, wird hier als Lösung vorgeschlagen, in der XPath Shell Zahlen in nicht escapede runde Klammmern zu setzen. Z.B. statt `@size < 100` neu zu schreiben `@size < (100)`. Wobei Zahlen auch ein Komma (also ein Punkt `.`) enthalten dürfen.
Eine andere Lösung wäre, anhand des Kontextes, in dem die Ziffernfolge auftaucht, zu entscheiden, ob es sich um eine Zahl oder einen Namenstest handelt. Dies könnte jedoch auch zu Zweideutigkeiten führen und braucht einen erheblichen zusätzlichen grammatikalischen Aufwand und wird darum hier nicht weiter verfolgt.

3.1.5 arithmetische Operationen

Wie in XPath sollen auch in der XPath Shell arithmetische Rechenoperationen erlaubt sein. Ähnlich wie der Multikplikationoperator, sind die Operatoren `+` und `-` erlaubte Ausdrücke in Datei- und Verzeichnisnamen. Damit diese Operatoren trotzdem verwendet weren können, werden sie, wie der Multiplikations-Stern, mit Buchstaben ausgeschrieben: ' `plus` ' und ' `minus` '. Diese Lösung wird unter anderem auch dadurch gerechtfertigt, dass der Modulo-Operator in XPath nur ausgeschrieben existiert. Doch wichtig ist bei allen vier arithmetischen Operatoren, dass vor und nach dem Wort ein nicht escapedes Leerzeichen steht, um die Ausdrücke nicht mit möglichen Pfadausdrücken zu verwechseln.

[2] Dieser Entscheid stellte sich im Verlauf der Programmierungs-Phase dieses Projekts als durchaus berechtigt heraus, da oftmals unnötige Fehler gemacht wurden, als die Prädikate in doppelten eckigen Klammern geschrieben werden mussten

```
'mod'  → ' mod '
'*'    → ' mult '
'div'  → ' div '
'+'    → ' plus '
'-'    → ' minus '
```

3.1.6 Namensräume

Wie in XPath sollten auch in der XPath Shell Namensräume verwendet werden können. In XPath werden die Namensräume über die Präfixe eingebunden, ohne sie jedoch direkt an eine Namensraum-URI zu koppeln. Die Präfixe werden deshalb genau so übernommen. Wobei, wie in XML definiert, Namensraum-Präfixe XML-Namen sein müssen[3]. Die Namensräume und ihre Präfixe lassen sich durch Shell-Variablen definieren, indem eine Umgebungsvariable mit dem Namen `xmlns_`*präfix* gesetzt wird, deren Wert dann dem Namensraum URI entspricht. Zusätzlich soll die Clark-Notation [8] erlaubt sein, bei welcher statt ein Präfix (welches an eine Namensraum-URI gebunden ist) und ein Doppelpunkt bloss die Namensraum-URI verwendet wird, indem sie in geschweiften Klammern vor dem Namenstest hingeschrieben wird. Z.B. kann statt `pref:name` (mit der Namensraum-Definition `xmlns:pref="http://www.dot.com"`) auch {`http://www.dot.com`}`name` geschrieben werden.

3.1.7 Content-, Meta- und Containing-Achsen

In *FSX* können an die Verzeichnis-, Symlink- und Datei-Knoten zusätzlich noch die Content- und Metadata-Unterbäume angehängt werden. Um vereinfacht auf diesen Bäumen operieren zu können, werden drei zusätzliche Achsen eingeführt.
Die Content-Achse kann dazu verwendet werden von einer Datei aus direkt auf den Wurzelknoten ihres Inhaltes zu springen, also auf den `fsx:content`-Knoten. Dazu wird ein neuer Operator eingeführt: `??`.
Ähnlich wird die Meta-Achse definiert, die auf den Metainformations-Unterbaum zeigt: `@@`.
Um von beliebigen Knoten innerhalb einer Datei (Knoten im Content- oder Meta-Unterbaum) zu dem Datei-Knoten springen zu können wird als drittes die Containing-Achse eingeführt, welche mit `%%` abegekürzt wird.
Diese drei Achsen sind keine speziell neuen Achsen, sondern zeigen bloss auf eine bestimmte Stelle im *FSX* Baum. Somit lassen sie sich einfach durch einen XPath definieren:

```
??  →  self::fsx:file/fsx:content
@@  →  self::fsx:*/fsx:metadata
%%  →  ancestor::fsx:*[local-name()='metadata' or
       local-name()='content'][last()]/parent::fsx:*
```

Ein kleiner Unterschied zwischen der Definition der Content- und der Meta-Achse besteht darin, dass über die Content-Achse bloss Kinder von Dateien selektiert werden können, während bei der Meta-Achse sämtliche *FSX* Knoten angesprochen werden. Das darum, weil in *FSX* bloss Dateien Inhalt haben, Metainformationen aber auch Ordner und Symlinks.

[3] Das macht es aus Implementierungssicht schwierig zu unterscheiden, ob etwas ein Präfix oder ein "normaler" Namenstest ist, der "zufällig" auch noch ein XML-Name ist

3.2 EBNF Grammatik

Die neue Grammatik ist, wie in der XPath-Definition eine kontextfreie Grammatik und lässt sich somit, am einfachsten in der Erweiterten Backus-Naur-Form (EBNF) darstellen

S	:=	Expr
[1] LocationPath	:=	RelativeLocationPath
	|	AbsoluteLocationPath
[2] AbsoluteLocationPath	:=	'/' RelativeLocationPath
	|	AbbreviatedAbsoluteLocationPath
[3] RelativeLocationPath	:=	Step
	|	RelativeLocationPath '/' Step
	|	AbbreviatedRelativeLocationPath
[4] Step	:=	'@' NodeTest
	|	'@' NodeTest PredicateList
	|	NodeTest
	|	NodeTest PredicateList
	|	AbbreviatedStep
	|	'??'
	|	'%%'
	|	'@@'
[7] NodeTest	:=	NameTest
PredicateList	:=	Predicate PredicateList
	|	Predicate
[8] Predicate	:=	'[' PredicateExpr ']'
[9] PredicateExpr	:=	Expr
[10] AbbreviatedAbsoluteLocationPath	:=	'//' RelativeLocationPath
[11] AbbreviatedRelativeLocationPath	:=	RelativeLocationPath '//' Step
[12] AbbreviatedStep	:=	'.'
	|	'..'
	|	'...'
[14] Expr	:=	OrExpr
[15] PrimaryExpr	:=	'(' Expr ')'
	|	LITERAL

	|	NUMBER
	|	FunctionCall
[16] FunctionCall	:=	FUNCTIONNAME ')'
	|	FUNCTIONNAME ArgumentList ')'
ArgumentList	:=	Argument
	|	Argument ',' ArgumentList
[17] Argument	:=	Expr
[18] UnionExpr	:=	PathExpr
	|	UnionExpr '|' PathExpr
[19] PathExpr	:=	LocationPath
	|	FilterExpr
	|	FilterExpr '/' RelativeLocationPath
	|	FilterExpr '//' RelativeLocationPath
[20] FilterExpr	:=	PrimaryExpr
	|	FilterExpr Predicate
[21] OrExpr	:=	AndExpr
	|	OrExpr ' or ' AndExpr
[22] AndExpr	:=	EqualityExpr
	|	AndExpr ' and ' EqualityExpr
[23] EqualityExpr	:=	RelationalExpr
	|	EqualityExpr '=' RelationalExpr
	|	EqualityExpr '!=' RelationalExpr
[24] RelationalExpr	:=	AdditiveExpr
	|	RelationalExpr '<' AdditiveExpr
	|	RelationalExpr '>' AdditiveExpr
	|	RelationalExpr '<=' AdditiveExpr
	|	RelationalExpr '>=' AdditiveExpr
[25] AdditiveExpr	:=	MultiplicativeExpr
	|	AdditiveExpr ' plus ' MultiplicativeExpr
	|	AdditiveExpr ' minus ' MultiplicativeExpr
[26] MultiplicativeExpr	:=	UnaryExpr
	|	MultiplicativeExpr ' mult ' UnaryExpr
	|	MultiplicativeExpr ' div ' UnaryExpr
	|	MultiplicativeExpr ' mod ' UnaryExpr

[27] UnaryExpr	:=	UnionExpr
	|	' minus 'UnaryExpr
[37] NameTest	:=	XMLNAME ':' FILENAME
	|	XMLNAME ':' XMLNAME
	|	XMLNAME
	|	FILENAME
	|	CLARKPREFIX FILENAME
	|	CLARKPREFIX XMLNAME

Tabelle 3.1: XPath Shell EBNF Grammatik, wobei die Wörter In Grossbuchstaben Terminalsymbole darstellen. Das Startsymbol ist S. Die Regeln mit einer Nummer davor, wurden von XPath übernommen (evtl. noch abgeändert).

3.2.1 Erklärungen zu den Grammatik-Regeln

Die Regeln 1 bis 3 werden direkt von XPath übernommen.
Der erste Unterschied betrifft Regel 4:

[4] Step	:=	'@' NodeTest
	|	'@' NodeTest PredicateList
	|	NodeTest
	|	NodeTest PredicateList
	|	AbbreviatedStep
	|	'??'
	|	'%%'
	|	'@@'

Neu hinzu kommen die abgekürzten Achsen '??', '%%' und '@@', die genau gleich verwendet werden, wie ein üblicher Location-Step. Um Zweideutigkeiten zu verhindern, wird die abgekürzte Attribut-Achse auch gleich hier definiert, da sie in XPath noch über einen zusätzlichen abgekürzten Step (*AbbreviatedAxisSpecifier*) definiert ist, welcher bei der Implementation Schwierigkeiten macht, da er nur optional durch das '@'-Zeichen ersetzt wird. Um die Implementation zu vereinfachen, werden die Regeln zwei Mal definiert, einmal mit und einmal ohne Prädikate. In der XPath-Spezifikation ist das durch eine optionale Prädikatliste angegeben. Diese Implementierung ist deshalb zulässig, da ein '@'-Zeichen escpaed werden muss, wenn es normal in einem Namen verwendet wird (siehe Tabelle A.2). Steht ein '@' ohne Escaping-Backslash, wird es daher direkt als das Spezialzeichen erkannt und weiterverarbeitet.
Die XPath-Regeln 5 und 6 fallen weg, um die Syntax klein zu halten.
Regel 7

[7] NodeTest	:=	NameTest

wird verkürzt, und nur noch Namenstests zugelassen. Auf die Konstrukte `NodeType()` und

`processing-instruction("...")`, wie sie in XPath vorkommen, wird verzichtet, da die in dieser Anwendung keine Rolle spielen.
Die Regeln 8 und 9 sind wieder gleich, wie XPath. Hinzu kommt noch die Regel um Prädikate hintereinander zuzulassen:

PredicateList := Predicate PredicateList

Die Regeln 10 und 11 werden wieder ohne Änderungen aus der XPath Definition übernommen.
Regel 12

[12] AbbreviatedStep := '.'
| '..'
| '...'

wird dahingehend erweitert, dass nun neu die abgekürzte Achse `...` eingeführt wird.
Wie schon weiter oben erklärt, konnte die XPath Regel 13 weggelassen werden.
Regeln 14 und 15 sind gleich, wie sie in der XPath Spezifikation stehen, bloss wird bei der Regel 15, die Ersetzung durch eine Variablen-Referenz entfernt, da die in der XPath Shell nicht zum Einsatz kommt.
Die Regeln 16 und 17 werden so verändert und erweitert, dass sie nun Yacc-tauglich sind, aber ihr Effekt nicht geändert wird.
Die Regeln 18 bis 26 werden wieder von XPath übernommen.
Regel 37

[37] NameTest := XMLNAME ':' FILENAME
| XMLNAME ':' XMLNAME
| XMLNAME
| FILENAME
| CLARKPREFIX FILENAME
| CLARKPREFIX XMLNAME

Diese Regel wird deshalb viel komplizierter als in der XPath-Spezifikation, da in XPsh erweiterte Namenstests erlaub werden. Wie in Kapitel 3.3.1 erklärt muss das Token XMLNAME eine höhere Erkennungspriorität besitzen als ein normales FILENAME Token, da ein Dateiname nicht zwingend ein XML-Name sein muss. Umgekehrt ist aber ein XML-Name immer auch ein Dateiname. Deshalb kann es vorkommen, dass ein Dateiname als XMLNAME oder als FILENAME erkannt wird. Beide Token können noch ein Präfix (auch ein XML-Name) haben (erste zwei Regeln). Die letzten beiden Regeln sind für die Erkennung der Clark Namensraum-Notation (siehe Kapitel 3.1.6). Wieder zwei Mal aufgeschrieben für das FILENAME und das XMLNAME Token.
Die restlichen XPath Regeln (27 bis 39) werden weggelassen, da sie nun in der lexikalischen Analyse verarbeitet werden, in ihrem Grundsatz aber gleich bleiben.

3.3 Lexikon

Die noch nicht vorgegebenen Terminalsymbole, die in der Grammatik in Kapitel 3.2 definiert wurden, müssen aus der Eingabe-Zeichenfolge erkannt werden. Dazu werden, ähnlich wie bei einer Grammatik, verschiedene Regeln aufgestellt. Bei dieser lexikalischen Analyse werden die einzelnen Elemente mittels Regular Expressions gesucht.

3.3.1 Variablen-Definitionen

Folgende Hilfsvariablen werden verwendet:

white := [\t]+

Definiert White-Space als eine beliebig lange Folge von Leerzeichen oder Tabulatoren. Doch mindestens ein Zeichen.

digit := [0-9]
integer := digit+
real := integer (“.” integer)?

Definiert die Zahlen. Zuerst eine einzelne Ziffer (digit), dann eine Folge von Ziffern als Ganzzahl (integer) und schliesslich noch eine beliebige Fliesskommazahl als Folge zweier Ganzzahlen, getrennt durch ein Komma.

filenameletter := [\53 \55 \56 \60-\71 \101-\132 \136-\137 \141-\172 \241-\377]
filenameescapes := ’\ ’ | ’\”’ | ’\#’ | ’\$’ | ’\%’ | ’\&’ | ”\”” | ’\(’ | ’\)’ | ’*’ | ’\,’ | ’\;’ | ’\<’ | ’\=’ | ’\>’ | ’\?’ | ’\@’ | ’\[’ | ’\]’ | ’\\’ | ’\‘’ | ’\|’ | ’\˜’ | ’\:’ | ’\{’ | ’\}’ | ’\!’
filenamechar := {filenameletter} | {filenameescapes}

Mit diesen drei Variablen werden die Zeichen definiert, die in einem Pfadausdruck verwendet werden können. Weiterverwendet wird nun die Variable filenamechar, welche aus den normalen Zeichen und den Sonderzeichen mit vorangestelltem Backslash besteht.

bracketexpr := ’[[’ [ˆ(\[\])]* ’]]’

Die Variable bracketexpr deckt die Wildcard-Ausdrücke ab, die mit den eckigen Klammern verwendet werden. Wie weiter oben erklärt, weicht hier die Syntax leicht von der ab, die in der Shell Kommando Sprache definiert ist, indem hier zwei statt nur einer eckigen Klammer stehen müssen.

filenameexpr := {filenamechar} | ’?’ | ’*’ | {bracketexpr}

Die eigentliche Definition von Grundelementen der Dateinamensausdrücke, bestehend aus den normalen Zeichen (filenamechar) und den Wildcards ?, * und bracketexpr.

prefix := ({letter} | ’_’) ({letter} | {digit} | ’.’ | ’-’ | ’_’)*

Die Standard XML-Definition von Präfixen. Zu beachten gilt, dass ein Präfix auch ein Da-

teinamen sein kann und wegen der höhren Priorität vom Scanner zuerst erkannt wird. Die Lösung ist in Kapitel 3.2 angegeben, indem ein Namenstest sowohl ein Dateiname als auch Präfix sein kann.

3.3.2 Erkennungsregeln

Nun die eigentlichen Regeln, welche dem Parser Generator mitteilen, welches Terminalsymbol erkannt wurde oder das Symbol zurückgeben, falls es keinem Terminalsymbol entspricht.

"[^"]*"	:=	LITERAL
'[^']*'	:=	LITERAL
({real})	:=	NUMBER
last(| position(| count(| id(| local-name(| namespace-uri(| name(| string(| concat(| starts-with(| contains(| substring-before | substring-after(| substring(| string-length(| normalize-space(| translate(| match(| boolean(| not(| true(| false(| lang(| number(| sum(| floor(| ceiling(| round(	:=	FUNCTIONNAME
{prefix}	:=	XMLNAME
('...' '.'+) | ({filenameexpr} | '.')* {filenameexpr}+ ({filenameexpr} | '.')*	:=	FILENAME
{[^}]*}	:=	CLARKPREFIX
{white}	:=	
\n	:=	
.	:=	. [4]

Die Regel zum Generieren eines FILENAME-Terminalsymbol ist etwas speziell geschrieben, dies aufgrund des Punkt-Zeichens. Da es durchaus erlaubt ist, mehrere Punkte als Dateinamen anzugeben, muss das hier speziell überprüft werden. Diese mehreren Punkte sind nur gültig, falls es vier oder mehrere sind, um keine Konflikte mit den vergebenen Namen '.', '..' und '...' (abkeürzte Achsen) zu provozieren. Dieser Tatsache wird im ersten Teil der Regel

[4] Jedes einzelne Zeichen, das nicht durh obige Regeln erkannt wurde wird direkt dem Parser Generator zur Weiterverarbeitung übergeben

Rechnung getragen, während im zweiten Teil Dateinamen erkannt werden, die Punkte und andere Zeichen kombinieren. Wichtig dabei ist, dass dabei immer mindestens ein anderes Zeichen als der Punkt vorkommen muss.

Kapitel 4

XPath Bibliothek

Wie schon weiter oben in Kapitel 2.3 beschrieben, wird für XPsh der halb-dynamische Ansatz der Syntax-Interpretation gewählt. Darum geht es in einem ersten Schritt um die Suche eines geeigneten XPath Prozessors. Einige Lösungen werden in den folgenden Kapiteln beschrieben.

4.1 XPath Prozessor

Eine der wichtigsten Eigenschaften des gesuchten XPath Prozessors ist, dass er frei ist, also Open-Source, um das Projekt im Sinne von GNU auch unter der GPL veröffentlichen zu können und anderen zugänglich zu machen. Im folgenden werden einige solcher Open-Source Bibliotheken vorgestellt.

4.1.1 Eigenentwicklung

Einen XPath Prozessor selbst zu entwickeln ist sicher die effizienteste Lösung, da man diesen genau auf die Bedürfnisse von XPsh abstimmen und die nicht gebrauchten Teile weglassen kann. Die Grundlage dazu, der Parser besteht sowieso schon. Zudem wäre es möglich Speicherverwaltungsaufgaben direkt einzubauen und so den Speicherverbrauch während der Ausführung möglichst klein zu halten.
Auf der anderen Seite verlangt eine Eigenentwicklung sehr viel mehr Aufwand und wird deshalb nicht weiterverfolgt.

4.1.2 Libxml2

Libxml2 aus dem Gnome-Projekt [4] ist sicher die am weitesten verbreitete Bibliothek für den Umgang mit XML-Daten auf einer Open-Source Basis. In dieser Bibliothek ist neben XSLT-, Schema-, Parsing- und weiterer Funktionalität auch ein XPath Interface integriert.
Um mit Libxml2 XPath-Ausdrücke zu evaluieren, wird dazu zuerst eine XML-Datei geparst und so ein `xmlDoc` erstellt. Anschliessend wird ein XPath Evaluations-Kontext erstellt. Auf diesen XPath Evaluations-Kontext kann mit `xmlXPathEvalExpression` der XPath angewendet werden.
Aus dieser kurzen Einführung geht bereits deutlich hervor, dass die Absicht des Libxml2 XPath-Moduls ganz klar die Verarbeitung bereits mit Libxml2 erstellter interner DOM-Strukturen ist. Es findet also keine Trennung zwischen DOM und XPath statt, was für einen effizienten Einsatz in XPsh aber notwendig wäre.

Eine Möglichkeit dieses Problem zu umgehen, wäre den *FSX* Baum statisch zu erstellen und dann von Libxml2 parsen zu lassen. Diese Lösung ist aber unbefriedigend, weil sie viel Speicher brauchen würde und nicht dynamisch wäre. Die Dynamik könnte, wie schon weiter oben angetönt, erzeugt werden, indem vor jeder XPath Evaluation das Dateisystem wieder geparst würde. Das wäre aber extrem langsam und wird darum auch gleich wieder verworfen.
Die andere Möglichkeit den Zugriff auf interne DOM-Strukturen zu durchbrechen, wäre, das XPath-Modul von Libxml2 dahingehend zu verändern, dass es statt auf die Repräsentation im Speicher zuzugreifen, die Knoten dynamisch generiert/verarbeitet. Die negativen Aspekte dieses Ansatzes sind sicher, dass am Schluss praktisch ein separater XPsh-Parser geschrieben würde und es sich dann auch nicht mehr um Libxml2 handelt. Doch das wäre gleichzeitig auch der Vorteil. Die XPsh Syntax könnte so direkt in das Core-Modul integriert werden, womit eine hohe Effizienz erreicht werden könnte.
Aus oben genannten Gründen, die gegen den Einsatz von Libxml2 beim XPath-Processing sprechen, wird an dieser Stelle auf Libxml2 verzichtet.

4.1.3 java.xml.xpath

Eine ganz andere Lösung wäre, das ganz Projekt auf Java basiert zu entwickeln, da für Java schon eine Grosszahl von XPath-Interfaces besteht, die durchaus mit einem Pull-Parser arbeiten könnten. Weil es sich bei XPsh aber um eine Betriebssystem-nahe Anwendung handelt und aus Java nicht auf alle Eigenschaften des Dateisystems zugegriffen werden kann, wird hier bei diesem Projekt auf Java verzichtet.

4.1.4 Sablotron

Zum Einsatz kommt nun das Open-Source Projekt Sablotron [7]. Hierbei handelt es sich um eine Bibliothek, welche XML-Parsing, XSLT 1.0, DOM Level2 und XPath beherrscht. Im Gegensatz zu Libxml2 sind die einzelnen Module klar getrennt, was den Einsatz von eigenen Interfaces erlaubt.
Wie weiter unten erklärt wird, wird mit SXP[1] über festgelegte Callback-Funktionen kommuniziert, die selbst implementiert werden können.
Der Nachteil eines Einsatzes von SXP ist, dass dadurch ein Eingriff in die Core-Funktionalität praktisch verhindert wird und das gesamte Projekt von der Effizienz der Sablotron Bibliothek abhängt. Zudem überlässt Sablotron, entsprechend der DOM Level 2 Core Spezifikation [6], das Speichermanagement komplett der darunterliegenden Objekte. Es gibt somit praktisch keine Möglichkeit, bereits erstellte Knoten wieder zu löschen und dabei sicher zu sein, dass diese nicht mehr gebraucht werden.
Doch die Vorteile der definierbaren Callback-Funktionen überwiegen, womit nun im weiteren die Sablotron Bibliothek verwendet wird.

4.2 match()-Funktion

Im XPath 1.0 Standard sind nur einige wenige Funktionen festgelegt. Um Dateinamen (*FSX* `name`-Attribute) auf bestimmte Muster zu prüfen, ist aber zusätzliche Funktionalität erforderlich. Dazu wird die Zusatz-XPath Funktion `match` eingeführt. `match` dient im wesentlichen als Wrapper-Funktion für den Unix-Systemaufruf *fnmatch(. . .)*.

[1] Sablotron XPath Processor

Es wäre auch möglich die XSLT Erweiterung *EXSLT*[1] und daraus die `regexp:match(...)` zu verwenden. Nur bleibt zu beachten, dass Datei- und Pfadausdrücke keine echten *Regular Expressions* sind und somit noch Aufwand betrieben werden muss, die Pfadausdrücke in korrekte reguläre Ausdrücke umzuwandeln.
Zum Einbezug der `match()`-Funktion in Sablotron, steht auf der Projekt-Homepage [10] ein Patch zum Herunterladen bereit.

Kapitel 5

Mapper

Um aus der Eingabe einen korrekten XPath-Ausdruck für dessen Weiterverarbeitung zu generieren, wird ein Mapper-Programm eingesetzt, das diese Aufgabe übernimmt. Dazu wird mittels Parser Generator die Syntax überprüft. So kann nebenbei auch noch festgestellt werden, wenn der Eingabestring kein korrekter XPsh Ausdruck ist. Dieser Mapper bildet gleichzeitig den Rahmen der XPath Shell. Im folgenden werden die einzelnen Teil kurz vorgestellt.

5.1 Initialisierung

Als erstes wird die XPath-Bibliothek SXP und die Core-XPsh Bibliothek libxpsh initialisiert.

5.2 Parsing

Als zweites wird jede Eingabe mit den Unix Tools Yacc und Lex geparst. Dazu wird die Grammatik, wie sie in Kapitel 3.2 beschrieben ist in Yacc-Regeln umgestzt und die lexikalischen Grundelemente, wie in Kapitel 3.3 vorgestellt, in Lex-Regeln überführt.
Im Yacc-Programm werden erkannte Tokens direkt durch die jeweilige XPath-Repräsentation ersetzt. Folgende Abbildungen werden verwendet:

CONTENTAXIS	→	self::fsx:*/fsx:content
METAAXIS	→	self::fsx*/fsx:metadata
CONTAININGAXIS	→	ancestor::fsx:content/parent::fsx:*
...	→	ancestor-or-self::node()
content-mode = false:		
XMLNAME:FILENAME	→	XMLNAME:*[match(@name,"FILENAME")]
XMLNAME:XMLNAME	→	XMLNAME:*[match(@name,"XMLNAME")]
FILENAME	→	fsx:*[match(@name,"FILENAME")]

XMLNAME	→	fsx:*[match(@name,"XMLNAME")]
content-mode = true:		
XMLNAME:XMLNAME	→	XMLNAME:XMLNAME
XMLNAME	→	XMLNAME

Mit der Laufzeitvariablen *content-mode* wird bestimmt, ob sich die Location Steps auf das Dateisystem oder Dateiinhalte (d.h. Content- oder Metadata-Unterbaum) bezieht. Bei Location Steps auf Dateisystem-Ebene (*content-mode = false*) werden Nametests durch ein Aufruf der `match()`-Funktion ersetzt. Hierbei ist zu beachten, dass XMLNAME beim Erkennen eine höhrere Priorität als FILENAME hat und ein Konstrukt wie XMLNAME:XMLNAME ermöglicht. Dies weil ein Dateiname auch ein XML-Name sein kann (und somit als XMLNAME geparst wird).
Auf Dateiinhalts-Ebene (*content-mode = true*) wird der Namenstest eins zu eins übernommen. So ist es möglich "ausserhalb" von Dateien mit Wildcards und Spezialzeichen im *FSX*-Baum zu navigieren, innerhalb den Dateien wie auf einem normalen DOM-Baum. Dies wird z.B. gebraucht, wenn eine XML-Datei in den *FSX*-Baum integriert wird und davon Knoten selektiert werden sollen.
Die Variable *content-mode* wird standardmässig auf *false* gesetzt. Sobald eines der Token CONTENTAXIS oder METAAXIS erkannt wird, wird gewechselt und *content-mode* wird *true*. Umgekehrt wird der Content-Mode wieder verlassen, beim Parsen eines CONTAININGAXIS Tokens.
Wichtig beim Parser ist, dass wenn er einen Syntax-Fehler feststellt, die Ausführung des gesamten Programms nicht sofort stoppt, wie das per Default in Yacc passieren würde. Das deshalb, weil ein Eingabestring auch als Parameter für ein Programm stehen kann und so nicht zwingend ein Pfadausdruck sein muss. In so einem Fall liefert der Parser den Null-String `'\0'` zurück und die ursprüngliche Eingabe wird unverändert ausgegeben.

5.3 Ausgabe

Nachdem der XPath-Query-String erstellt wurde, wird er der Sablotron Bibliothek übergeben, die mit Hilfe von Libxpsh (siehe Kapitel 6) das Ergebnis erstellt. Dieses Ergebnis kann eine Zahl, eine Zeichenkette (String), eine Boolean-Variable oder eine Knotenmenge sein. Bei den ersten drei Fällen wird das Ergebnis direkt ausgegeben, bei einer Knotenmenge wird darüber iteriert und jeder Knoten ausgegeben. Ein `fsx:file`-, `fsx:dir`- oder `fsx:symlink`-Knoten wird mit seinem Pfad relativ zum aktuellen ausgegeben, um die Datei oder den Ordner eindeutig zu identifizieren. Text-, Namensraum- und Attribut-Knoten werden als deren Text, dessen Namensraum-URI und dessen Attribut-Wert ausgegeben (gemäss DOM Level 2 Core Spezifikation, mit der Ergänzung, dass bei Namensraum-Knoten die Namensraum-URI ausgegeben wird).

Kapitel 6

Libxpsh

Der zentrale Punkt von XPsh bildet die XPsh-Bibliothek Libxsph. Sie definiert die interne Struktur der verwendeten Knoten, kommuniziert mit SXP und integriert die Adaptors.

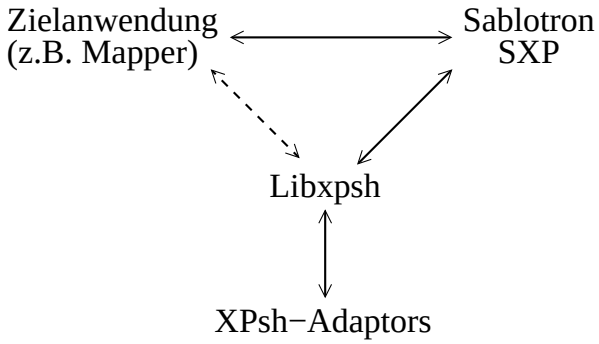

Abbildung 6.1: Kommunikationsschema von Libxsph. Die Verbindung zwischen der Zielanwendung und Libxpsh ist nicht ausgezogen, weil diese Kommunikation nur gerade während der Initialisierung stattfindet.

6.1 Kommunikation mit SXP

Die Sablotron-Bibliothek SXP verwendet zur XPath Evaluation die folgenden Funktionen:

- `getNodeType`
- `getNodeName`
- `getNodeNameURI`
- `getNodeNameLocal`
- `getNodeValue`
- `getNextSibling`
- `getPreviousSibling`

- `getNextAttrNS`
- `getPreviousAttrNS`
- `getChildCount`
- `getAttributeCount`
- `getNamespaceCount`
- `getChildNo`
- `getAttributeNo`
- `getNamespaceNo`
- `getParent`
- `getOwnerDocument`
- `compareNodes`

SXP erlaubt nun, für jede dieser achtzehn Funktionen eine eigene Version zu schreiben, im Sinne einer Callback-Funktion. SXP definiert selber keine Knotendatentypen, sondern operiert nur mit Zeigern. Wie in der DOM Level 2 Core Spezifikation definiert, kümmert sich SXP auch nicht um die Speicherverwaltung. Im folgenden Kapitel werden die Grund-Datentypen vorgestellt, die Libxpsh verwendet.

6.2 Datentypen

Der Datentyp, der für die Knoten verwendet wird, muss sicherstellen, dass zu jedem Zeitpunkt der XPath-Evaluation eindeutig feststellbar ist, um welchen Knoten es sich handelt und was sein Kontext ist. Also, welche Attribute er besitzt, in welchen Namensräumen er sich befindet, welches sein Eltern-Knoten ist, u.s.w. Zudem sollte einfach feststellbar sein, ob ein Knoten im Speicher bereits schon existiert und darum nicht mehr neu erstellt werden muss. Dazu bieten sich zwei Ansätze an:

6.2.1 Hash-Tabelle

```
typedef struct node_i {
                  char *path;
                  char *name;
                  char *valule;
                  node_type type;
} node_xpsh;
```

Jeder Knoten ist eindeutig über seinen Pfad und sein Name identifizierbar. Bevor ein neuer Knoten erstellt wird, wird in eine Hash-Tabelle überprüft, ob der Knoten schon existiert und allenfalls seine Adresse zurückgegeben. Der Suchschlüssel würde aus der Kombination von `path` und `name` geschehen. Zwei grosse Nachteile hat diese Lösung. Einerseits wäre es

recht aufwändig, in Meta- und Content-Unterbäumen herumzunavigieren, da dazu jeweils der Pfad zerlegt und herausgefiltert werden müsste, ob es sich bei einem Location-Step um einen Knoten des Dateisystems oder des "erweiterten" *FSX*-Baums handelt. Andererseits ist für einen effizienten Einsatz einer Hash-Tabelle notwendig, im Voraus dessen Grösse zu kennen[1]. Es sollte dabei darauf geachtet werden, dass der Füllgrad der Hash-Tabelle nicht grösser als 75% wird. Das würde bedeuten, dass zu Beginn eine XPath-Analyse gemacht werden muss oder ein Wert heurisitsch zur Kompilierzeit geschätzt wird. Doch beide Fälle sind nicht sonderlich befriedigend, da eine solche Tabellen-Grösse-Schätzung recht schwierig zu implementieren wäre. Aus diesen Gründen wird dieser Ansatz wieder verworfen.

6.2.2 Verkettete Liste

```
typedef struct node_i {
                node_xpsh *parent;

                node_xpsh *nextSibling;
                node_xpsh *prevSibling;

                node_xpsh *firstChild;
                node_xpsh *firstAttribute;

                node_xpsh *NS;
                node_xpsh *NSDecl;

                node_type type;

                char *name;
                char *value;
} node_xpsh;
```

Der Datentyp `node_xpsh` definiert die interne Repräsentation aller *FSX*-Knoten. Wie deutlich herauskommt, ist der *FSX*-Baum intern als doppelt verkettete Liste in Baumform im Speicher abgelegt.
`parent` ist ein Zeiger auf das Eltern-Element des Knotens. Ist dieser Wert *NULL*, muss es sich um den Wurzelknoten des Dateisystems handeln. `nextSibling` bzw. `prevSibling` ist ein Zeiger auf den nächsten bzw. vorherigen Nachbarknoten. Es spielt keine Rolle, was für ein Typ der aktuelle Knoten ist. Z.B. ein Metadaten-Knoten könnte als Nachbar durchaus ein Datei-, Symlink- oder Directory-Knoten haben. Bei Attributknoten, zeigt dieser Wert auf das nächste, bzw. vorherige Attribut. Gleich verhalten sich Namensraum-Knoten. Am Ende bzw. Anfang der Liste wird dieser Wert auf *NULL* gesetzt. Die zwei Einträge `firstChild` und `firstAttribute` zeigen auf die möglichen Kinder- und Attributs-Knoten, welche als Eltern-Knoten den aktuellen Knoten haben. Diese sind dann als `node_xpsh`-Knoten selbst wieder mit ihren Nachbarn über die `nextSibling` und prevSibling Einträge verbunden. Die Variable `NS` zeigt auf einen Namensraum-Knoten, in dem sich der aktuelle Knoten befindet (z.B. wenn er in einer XML-Datei als prefix:name geschrieben wird). `NSDecl` zeigt auf eine

[1] Eine Hash-Tabelle kann auch dynamisch erweitert werden, was aber wieder mit zusätzlichem (Rechen-) Aufwand verbunden ist

Liste von Namensraum-Knoten, die in dem aktuellen Element definiert werden (kann bei der Einbindung von XML-Content auftreten).
`type` speichert den Knoten-Typ. mögliche Werte sind:

- `fsx_FILE`
- `fsx_DIR`
- `fsx_SYMLINK`
- `fsx_ROOT_METADATA`
- `fsx_METADATA`
- `fsx_ROOT_CONTENT`
- `fsx_CONTENT`
- `fsx_ATTRIBUTE`
- `fsx_NS`
- `fsx_TEXT`

Die ersten drei Werte sind abgeleitet von den drei Grundknoten aus *FSX*. Sie werden hier unterschieden, damit deren Umgang effizient geschehen kann. Im Prinzip wären sie ja "normale" Element-Knoten. Die beiden Knotentypen `fsx_ROOT_METADATA` und `fsx_ROOT_CONTENT` werden verwendet, um die Meta-Informations- und Inhalts-Unterbäume erstellen zu können. Sobald ein Aufruf kommt, um ein Kind, Attribut oder Namensraum eines solchen Knotens zurückzugeben, bedeutet das nichts anderes, als dass der zugehörigen Unterbaum generiert werden muss und daraus der gewünschte Knoten zurückgegeben wird, sofern er existiert. Ein Knoten im Meta-Informations bzw. Content-Unterbaum hat automatisch den Typ `fsx_METADATA` bzw. `fsx_CONTENT` (ausser Attribute, Text- und Namensraum-Knoten). Soll eine Aktion auf einem solchen Knoten ausgeführt werden (z.B. ein bestimmtes Attribut oder Kind finden), bedeutet das, dass dieser Baum schon vollständig expandiert wurde und wenn der Knoten nicht existiert auch keiner zurückgegeben werden kann (siehe dazu auch Kapitel 7). Jeder Attribut-Knoten im gesamten *FSX*-Baum ist vom Typ `fsx_ATTRIBUTE`, jeder Namensraum-Knoten `fsx_NS`. Zuletzt kann ein Content- oder Meta-Unterbaum auch Text-Knoten beinhalten (beispielsweise, wenn eine XML-Datei eingebunden wird), diese werden dann mit `fsx_TEXT` spezifiziert. Alle anderen Knoten werden in XPsh aus Effizienz-Gründen nicht unterstützt (mit der Ausnahme, dass CDATA-Knoten in Text-Knoten umgewandelt werden können).
`name` und `value` repräsentieren den Namen und Wert des Knotens, gemäss DOM Level 2 Core Spezifikation, mit kleinen Änderungen. Da es sich bei den Datei-, Directory- und Symlink-Knoten um Element-Knoten handelt, sollte also ihre Namen `file`, `dir` und `symlink` sein. Aus praktischen Gründen wurde hier von der Spezifikation abgewichen und stattdessen den Namen der Datei oder des Ordners eingesetzt[2]. Mit dem `type`-Eintrag kann aber trotzdem festgestellt

[2]In einer ersten Version wurde im `name`-Eintrag der gesamte Pfad gespeichert, um schnell Aktionen auf dem Knoten durchführen zu können, die das Dateisystem betreffen (z.B. Attribute oder Kinder generieren). Aufgrund des unnötig hohen Speicherverbrauchs, wurde diese Idee weiterentwickelt und nur noch der Datei-

werden, um welchen Element-Knoten es sich handelt. Als weitere Ergänzung beinhaltet `name` bei Namensraum-Knoten den Präfix und `value` die Namensraum-URI. Ein weiterer Grund, dass an Stelle des richtigen Namens der Datei- bzw. Verzeichnis-Name steht, ist, dass zur rekursiven Pfadgenerierung direkt diese Namen verwendet werden können und nicht zuerst noch die *name*-Attribute der *FSX*-Knoten generiert werden müssen (was unnötiger Rechenaufwand und Speicherverbrauch bedeuten würde).
Diese `xpsh_node`-Struktur wird, gleich wie die `name`- und `value`-Einträge, mit `malloc()` erzeugt und somit auf dem Heap abgelegt. Das ist umso wichtiger, da diese Knoten global sichtbar und zugänglich sein müssen, also von allen Callback-Funktionen. Zu beachten gilt auch noch, dass die Knoten mit mindestens einer weiteren solchen Struktur verlinkt sein müssen, um zum Schluss den Speicher mit `free()` wieder freigeben zu können.

6.3 Initialisierung

Libxpsh bietet zur Initialisierung zwei Funktionen an:

6.3.1 `declareNamespaces()`

`declareNamespaces()` lädt in einem ersten Schritt alle Adaptoren (mehr dazu in Kapitel 7) und definiert deren Namensräume in SXP. Das erlaubt jedem Adaptor einen separaten Namensraum zu verwenden, der nachher in XPsh nicht mehr explizit angegeben werden muss. In einem zweiten Schritt werden alle Shell-Umgebungsvariablen, die mit `xmlns_` beginnen, als Quelle weiterer Namensräume verwendet. Als Beispiel soll die URI `http://www.dot.com` an den Präfix `dot` gebunden werden. Dazu wird vor der nächsten Ausführung des nächsten XPsh-Query die Umgebungsvariable `xmlns_dot` auf den Wert `http://www.dot.com` gesetzt[3]. Und somit lässt sich nun der Präfix `dot` in XPsh nutzen.

6.3.2 `genParents()`

`genParents()` erzeugt alle Knoten der *ancestor-or-self*-Achse bezüglich eines Dateisystem-Pfadausdrucks. Diese Funktion wird somit zur Generierung des Kontext-Knotens verwendet. Gleichzeitig wird die globale Libxpsh-Variable `root` auf den Wurzelknoten des *FSX*-Baumes gesetzt. Dieser Zeiger wird verwendet, wenn ein Location-Pfad ausgehend vom Root-Verzeichnis beginnt.
Der Grund, dass es bei der Initialisierung von XPsh nicht reicht, bloss den Kontext-Knoten mit einem simplen `malloc()` zu erzeugen liegt darin, dass die Vorgänger-Knoten dazu gebraucht werden, um den eindeutigen Pfad einer Datei angeben zu können, wie in Kapitel 6.2.2 beschrieben. Dieses Verhalten hat den grossen Nachteil, dass die Initialisierung bereits verhältnismässig lange dauert, verglichen z.B. mit dem Unix `find`-Utility. Auf der anderen Seite ist der grosse Vorteil der Generierung sämtlicher Vorgänger-Knoten, dass zu jedem Zeitpunkt und für jeden Knoten, der während der XPath-Evaluation gebraucht wird, der Eltern-Knoten bereits existiert und nicht mehr erstellt werden muss.

---

oder Verzeichnis-Name gespeichert. Somit wird der eindeutige absolute Name im Dateisystem erzeugt, indem, ausgehend vom Wurzelelement (wird erreicht durch wiederholtes verfolgen der `parent`-Zeiger, bis `parent` zu *NULL* wird) alle Namen der Knoten bis zum aktuellen verkettet werden.

[3] In Linux wird das mit folgendem Kommando erreicht: `export xmlns_dot="http://www.dot.com"`

6.4 Callback-Funktionen

An dieser Stelle werden nur diejenigen Funktionen vorgestellt, deren Funktionsweise nicht direkt durch den Namen und die interne Knoten-Struktur gegeben ist.

6.4.1 `getNodeType`

Liefert den Knoten-Typ zurück (entsprechend DOM Level 2 Core Spezifikation). Sämtliche *FSX*-Dateisystem Knoten liefern als Ergebnis `ELEMENT_NODE`, genau geich wie Content- und Meta-Informations-Knoten. Die drei weiteren Knoten-Typen (`ATTRIBUTE_NODE`, `NAMESPACE_NODE` und `TEXT_NODE`) sind über den `xpsh_node`-Eintrag `type` bereits vorgegeben.

6.4.2 `getNodeNameURI`

Um den Speicherverbrauch nicht unnötig zu vergrössern, besitzen die Knoten vom Typ `fsx_ROOT_CONTENT`, `fsx_ROOT_METADATA`, `fsx_FILE`, `fsx_SYMLINK` und `fsx_DIR` normalerweise keine expliziten Namensraum-Knoten, da sich alle diese Knoten (nur genau) im (Default-) *FSX*-Namensraum befinden. Besitzt der Kontext-Knoten einen Verweis zu einem Namensraum-Knoten, wird dessen URI zurückgegeben.

6.4.3 `getChildNo`

Ursprünglich war die Idee, mit `getChildNo` einen ersten Knoten und mit `getNextSibling` weitere zu erstellen. Da aber, im Gegensatz zu Dateisystemen, bei XML Geschwister gleich sein können (auf das Dateisystem bezogen hiesse das, ein Ordner kann zwei gleiche Dateien beinhalten), werden in der weiteren XPath-Abarbeitung mit sehr grosser Wahrscheinlichkeit sämtliche Kinder-Knoten durchlaufen. Darum werden nicht einzelne Kinder sondern sämtliche Kinder-Knoten erstellt, wenn die Funktion `getChildNo` aufgerufen wird. Ausser wenn die Kinder bereits existieren, werden bloss die jeweiligen Zeiger zurückgegeben. Dieses Verhalten bedeutet einen enormen Performance-Gewinn. Zwar werden unter Umständen Knoten alloziert, die nie gebraucht werden, dafür muss nicht bei jedem `getNextSibling`-Aufruf das Eltern-Verzeichnis geöffnet und dessen Inhalt durchlaufen werden um beim gewünschten Geschwister-Knoten zu landen.
Wie schon angetönt, werden Kinder, die bereits existieren direkt zurückgegeben und nicht nochmals alloziert. Das gilt indirekt auch für Text-, Attribut- und Namensraum-Knoten, die nie Kinder haben können.
Aus *FSX* geht weiter hervor, dass jeder *FSX*-Knoten einen Meta-Informations-Knoten als Kind besitzt. Implementiert wurde diese Eigenschaft, indem der erste Kind-Knoten aller *FSX*-Knoten der Meta-Informations-Knoten ist. Bei Datei-Knoten, welche zudem noch den Wurzelknoten dessen Contents als Kind besitzen, folgt dieser als zweites Kind.
Wird ein Kind eines Wurzelknotens des Meta-Informations- bzw. Content-Baumes angefordert, wird als erstes der gesamte Meta- bzw. Content-Unterbaum erstellt und dann der gewünschte Knoten gesucht [4].

[4] Die Content- und Meta-Unterbäume werden jeweils komplett erzeugt, wenn ein Knoten davon angefordert wird. Dies einerseits, um die Performance zu optimieren und andererseits die Adaptoren einfach zu halten. Wenn die Knoten komplett dynamisch erzeugt würden (also erst dann, wenn sie angefordert werden), müsste jedes Mal die Datei geöffnet werden und die Knoten gesucht werden, was unter Umständen recht aufwändig werden könnte.

6.4.4 `getNextSibling`

Wie schon in Kapitel 6.4.3 erklärt, muss an dieser Stelle der Geschwister-Knoten des aktuellen Kontext-Knotens zwingend bereits existieren, mit der Ausnahme, dass wenn es sich um den Wurzelknoten des Meta-Informations-Unterbaumes eines Directories handelt. In diesem Fall wird der Knoten erst erzeugt, gleichzeitig mit den anderen Kindern (Dateien) dieses Directories.

6.4.5 `getNextAttrNS`

Wie in den vorherigen Kapiteln erklärt sind Attribute, wenn sie sich im Meta- oder Content-Unterbaum befinden bereits komplett erstellt worden und können deshalb direkt zurückgegeben werden.
Attribute von Dateien, symbolischen Verweisen und Verzeichnissen werden hingegen dynamisch erzeugt, sobald sie gebraucht werden. Das darum, da laut XML-Spezifikation ein Knoten keine Attribute mit dem selben Namen besitzen dürfen. Die Werte der Attribute werden vorwiegend aus den Einträgen der `stat`-Struktur extrahiert, welche über den Systemaufruf `stat()` ausgelesen wird. Eine Ausnahme bildet das `mime`-Attribut, dieses wird anhand der vordefinierten Einträge in der Einstellungs-Datei aus der Dateiendung erstellt.

6.4.6 `getParent`

Wie schon weiter oben beschrieben, existiert zu jedem Zeitpunkt der Ausführung und für jeden Knoten der Eltern-Knoten bereits. Dadurch beinhaltet diese Funktion keine spezielle Funktionalität mehr.

6.4.7 `compareNodes`

Der Vergleich zweier Knoten ist einfach, wenn beide Knoten an der selben Adresse im Speicher abgelegt sind. Ist das nicht der Fall werden die `name`-Einträge verglichen.

6.4.8 `getOwnerDocument`

Liefert den Wurzelknoten (bzw. bloss dessen Adresse) des *FSX*-Baumes zurück, der mit `genParents()` erzeugt wurde (siehe Kapitel 6.3.2).

6.5 Adaptoren

Als zentraler Punkt von *FSX* werden Datei-Inhalte und Meta-Informationen mit der Hilfe von Adaptoren in den Dateisystem-Baum eingebunden. Um diese Adaptoren möglichst dynamisch behandeln zu können (wie es *FSX* auch vorschlägt), werden Adaptoren nicht in der Libxpsh implementiert, sondern in separaten Bibliotheken, die dann in Libxpsh dynamisch dazugeladen werden, sobald sie gebraucht werden. Dazu wird seitens Libxpsh ein einheitliches Interface definiert, welches sechs Funktionen vorgibt, die mindestens implementiert sein müssen, wie im nächsten Abschnitt beschrieben.

6.5.1 Interface

```
char **implemented_mime_types();
```

Gibt ein Zeichenketten-Array zurück, das sämtliche MIME-Typen beinhaltet, für die der Adaptor verwendet werden kann. Tritt ein MIME-Typ mehrmals auf, wird der Adaptor verwendet, der in der Konfigurationsdatei zuerst aufgeführt ist.

```
int n_implemented_mime_types();
```

Gibt an, wieviele Einträge das Array beinhaltet, das durch die vorherige Funktion zurückgegeben wird.

```
char *get_ns_prefix();
char *get_ns_uri();
```

Erlaubt den Adaptoren eigene Namensräume zu verwenden, die über diese zwei Funktionen definiert werden können. Sonst sollten beide Funktionen *NULL* zurückgeben. Die Verwendung separater Namensräume in den Adaptoren wird empfohlen, um Konflikte zu vermeiden (z.B. gleiche Attribut-Namen, wie *FSX* für seine Attribute verwendet).

```
void expand_metadata( root_metadata_node, filename, mime_type );
void expand_content( root_content_node, filename, mime_type );
```

Diese beiden Funktionen erstellen den Meta-Informations- bzw. den Content-Unterbaum. Sie erhalten als Parameter den Knoten, an dem der jeweilige Unterbaum angehängt werden soll (`root_metadata_node`), der Dateiname (`filename`) und der MIME-Typ der Datei (`mime_type`). Wichtig ist, dass diese beiden Funktionen die Knoten auf dem Heap allozieren (d.h. also mit `malloc()` oder ähnlichen Funktionen), damit sie auch in Libxpsh erhalten bleiben. Wie genau, bleibt dem Adaptor selbst überlassen.

6.5.2 Einbindung

Die Adaptoren werden in der Konfigurationsdatei angegeben, indem eine Zeile mit dem Wort `Adaptor` gefolgt von einem Leerzeichen beginnt. Anschliessend folgt der (absolute) Pfad zu der dynamischen Bibliothek.
Beim Initialisieren von Libxpsh werden vor dem eigentlichen Evaluieren des XPaths mit SXP dei Namensräume definiert. Dazu werden alle dynamischen Bibliotheken der Adaptoren geladen und über obige Funktionen die Namensräume herausgefiltert und in Sablotron definiert. Dadurch sind die Adaptoren bereits geladen, wenn sie später einen Baum erstellen müssen.
Daraus entsteht ein zusätzlicher, scheinbar unnötiger, Initialisierungs-Overhead, bevor die eigentliche XPath-Evaluation beginnen kann. Weil aber SXP die Definition der Namensräume zu diesem Zeitpunkt vorschreibt, lässt sich diese Problematik nicht einfach umgehen, um dabei noch dynamisch zu bleiben.

6.6 MIME-Typen

Um feststellen zu können, welcher Adaptor auf eine bestimmte Datei passt, muss möglichst genau festgestellt werden können, um welchen Typ Datei es sich handelt. Dazu wird der MIME-Typ verwendet. Um diesen feststellen zu können, sind verschiedene Varianten denkbar:

6.6.1 Unix `file` Programm

Der Vorteil dieser Lösung ist sicher, dass `file` sehr viele Datentypen kennt (kennen kann) und dabei den Typ nicht bloss anhand der Dateiendung ermittelt, sondern noch die binäre Struktur der Datei miteinbezieht. Über das Flag `-i` könnte man das Programm auch dazu bewegen den MIME-Typen zurückzugeben und nicht die Standard-Beschreibung die es normalerweise verwendet. Der überwiegende Nachteil, den die `file`-Lösung mit sich bringt ist, dass das Programm zur Typ-Evaluation eine Konfigurations-Datei verwendet, die je nach Betriebssystem/Distribution verschieden sein kann. Zudem braucht alleine die Initialisierung des Kommandos noch eine gewisse Zeit, was sich ebenfalls negativ auswirken könnte.

6.6.2 Externe Bibliothek

Verschiedene Programme unter Unix haben bereits eine MIME-Typ Evaluation implementiert, beispielsweise das Apache Modul `mod_mime`. Somit liesse sich der Dateityp auch ausserhalb von Libxpsh bestimmen. Der Nachteil ist, dass dabei sichergestellt werden müsste, dass diese externe Bibliothek erstens installiert ist und zweitens auf allen Plattformen die selben Resultate liefert. Besonders wegen dem ersten Negativpunkt wird auf diese Lösung verzichtet.

6.6.3 Dateiendung und Konfigurationsdatei

Die aktuell implementierte Lösung bestimmt den MIME-Typen anhand der Dateiendung. Dieses Verhalten versagt zwar bei Dateien ohne Endung oder bei Dateien, die eine falsche Endung tragen. Dafür ist diese Lösung flexibel und speziell für Libxpsh anpassbar. Die Abbildung der Dateiendungen auf den MIME-Typen wird in der Konfigurationsdatei abgespeichert. Die Syntax dau lautet: `MIME` *MIME-Typ Dateiendung (ohne .).*

6.6.4 Ordner und symbolische Verweise

Da Ordner normalerweise keine Dateiendung tragen, über die auf den MIME-Typen geschlossen werden kann, werden Ordner hardcoded als MIME-Typ *application/directory* deklariert. Dieser MIME-Typ wird verwendet, damit Adaptoren auch Meta-Informationen von Ordnern generieren können. Wenn Ordner als Dateien betrachtet werden, mit den beinhaltenden Dateien und Verzeichnissen als dessen Inhalt, lässt sich die Verwendung eines MIME-Typs für Ordner auch rechtfertigen.

6.7 Beenden

Libxpsh sollte mit der Funktion `xpsh_clear()` beendet werden. Sie bewirkt, dass die interne Repräsentation des `FSX`-Baums (doppelt verkettete Liste) mit `free()` gelöscht und der Speicher wieder freigegeben wird. Wenn mehere Abfragen hintereinander abgearbeitet werden sollten, wird empfohlen `xpsh_clear()` erst nach der letzten Abfrage auszuführen. Das

bewirkt, dass nicht alle Knoten erneut erzeugt werden müssen und somit die Geschwindigkeit gesteigert werden kann.

Kapitel 7

Adaptoren

In der jetztigen Version von XPsh sind drei Adaptoren implementiert. Sie werden in den nächsten Abschnitten kurz vorgestellt.

7.1 mp3

Der mp3-Adaptor behandelt bloss die Meta-Informationen einer mp3-Musikdatei (MIME Typ audio/mpeg). Das sind die (optionalen) ID3-Tags und Informationen über die Kodierung.
Die ID3-Tags werden über die, unter Unix weit verbreitete (und daher mit grosser Wahrscheinlichkeit auch auf dem System installiert), `id3lib`[2] Bibliothek ausgelesen. Das binäre Auslesen der Kodierungsinformationen aus der Datei ist direkt implementiert und benötigt daher keine weiteren Hilfs-Bibliotheken.
Da es sich bei den extrahierten Daten nicht um Datei Inhalte, sondern nur um Zusatz-Informationen handelt, wird in diesem Adaptor nur der Meta-Unterbaum expandiert. Es gibt dabei zwei Möglichkeiten um die Daten in eine XML-Struktur einzubetten. Zum einen könnten die Bezeichner Knoten sein und deren (Text-) Inhalt die eigentlichen Informationen beinhalten. Zum anderen könnten, und so ist es auch gelöst, die Daten über Attribute direkt an den Wurzelknoten des Meta-Baums angehängt werden. Diese Lösung wurde aus Performance-Gründen gewählt, da so pro Information nur ein Knoten (Attribut-Knoten) benötigt wird, während bei der ersten Variante zwei Knoten (Element- und Text-Knoten) benötigt werden.
Folgende Attribute werden erzeugt, wenn sie die Informationen dazu vorhanden sind:

- Title* *(Titel)*
- Artist* *(Künstler)*
- Album* *(Name des Albums)*
- Track* *(Track-Nummer)*
- Year* *(Jahr)*
- Length* *(Länge)*
- Version *(MPEG Version: "MPEG 1", "MPEG 2", "MPEG 2.5")*

*diese Attribute sind nur verfügbar, wenn id3lib installiert ist

- Layer *(Layer: "Layer I", "Layer II", "Layer III")*
- Mode *("Stereo", "Joint Stereo", "Mono", "Dual Channel")*
- Bitrate *(Bitrate, Zahl oder "VBR")*
- Frequency *(Abtastfrequenz)*

Die mp3-Attribute befinden sich im `mp3`-Namensraum. Um als Beispiel der Titel eines mp3-Liedes auszulesen müsste folgendes Kommando verwendet werden

```
Dateiname/@@/@mp3:Title
```

7.2 JPEG

Der JPEG-Adaptor fügt EXIF-Meta-Informationen über Bilddateien im JPEG Format (MIME Typ image/jpeg) in den *FSX*-Baum ein. Auch hier handelt es sich bei den Daten, die eingefügt werden, nur um Zusatz-Informationen über das Bild und nicht um dessen Inhalt. Somit wird im JPEG-Adaptor ebenfalls nur der Meta-Unterbaum expandiert. Aus gleichen Überlegungen wie beim mp3-Adaptor, werden die Informationen als Attribute des Wurzelknotens des Meta-Unterbaums eingebunden.
Die EXIF Tags, die im Header einer JPEG (oder TIFF) Bilddatei gespeichert sind, werden mit der `libexif`[3] Bibliothek ausgelesen und als Attribute an den `meta`-Knoten angehängt. Da es sich dabei um eine (viel zu) grosse Menge an verschiedenen Einträgen handelt, sind hier die Attribute auf ein paar wenige Tags beschränkt:

- Description
- Model
- Orientation
- Width
- Height
- XResolution
- YResolution
- ResolutionUnit
- ColorSpace
- DigitalZoomRatio
- Flash
- ExposureTime
- FNumber

- ISOSpeedRatings
- FocalLength

Sämtliche Attribute befinden sich im separaten `jpeg`-Namensraum. Um also z.B. das JPEG-Model-Tag auszulesen müsste folgendes Kommando verwendet werden

```
Dateiname/@@/@jpeg:Model
```

7.3 XML

Die vielfältigsten Möglichkeiten bietet sicher die Integration von XML-Dateien in den *FSX*-Baum. Diese Einbindung wird mit dem XML-Adaptor durchgeführt, welcher auf der frei verfügbaren, offenen `libxml2` [4] Libxml2 aufbaut.
Die Grundidee hinter der Einbindung von XML-Dateien ist, dass eine XML-Datei selbst auch einen Baum beinhalten kann (sofern es sich um eine korrekte, wohlgeformte XML-Datei handelt). Dessen Wurzel wird nun mit dem Wurzelknoten des Content-Unterbaumes gleichgesetzt und darunter dann der eigentliche Inhalt eingefügt.
Dabei tritt das Problem der Namensräume auf. Alle Knoten im *FSX*-Baum sind per Default im FSX-Namensraum. Daraus würde folgen, dass sämtliche Knoten und Attribute einer XML-Datei, die nicht explizit einem Namensraum zugeordnet sind, sich ebenfalls im FSX-Namensraum befinden. Aus diesem Grund wird bei der Implementierung darauf verzichtet einen Default-Namensraum zu definieren. Dies spielt auch keine grosse Rolle, da Knoten des FSX-Knoten (Dateien, Verzeichnisse und Symlinks) sowieso nicht direkt "angesprochen" werden.
Der zweite kritische Punkt betrifft die Performance und den Speicherverbrauch. Es kann durchaus sein, dass eine XML-Datei eingebunden werden soll, die sehr gross ist und viele Knoten beinhaltet. Das würde dazu führen, dass der Adaptor lange braucht, um den Content-Unterbaum zu expandieren und zweitens wäre dann der Speicherverbrauch sehr gross. Eine Lösung wäre, analog wie auf Dateisystem-Ebene, die Knoten dynamisch zu generieren. Das bringt jedoch wieder zwei Nachteile mit sich: Das Standard-Interface für Adaptoren könnte nicht verwendet werden, da dieses vorschreibt, dass gleich der ganze Baum angehängt werden muss. Und zum zweiten müsste die XML-Datei jedes Mal neu geparst werden, was unnötig Rechenleistung erfordert. Aus diesem Grund, wird im Adaptor zur Kompilier-Zeit die maximale Suchtiefe definiert, nach der der Adaptor abbricht. Dadurch könnte zwar trotzdem eine grosse Menge Knoten durchlaufen werden, wenn z.B. der XML-Baum sehr breit und nicht tief ist, aber im allgemeinen Fall, wird nur eine vernünftige Anzahl Knoten eingefügt. Der grosse Nachteil ist natürlich, dass Knoten, die sich weiter unten im Baum befinden, nicht berücksichtigt und gefunden werden können.
Processing Instruktionen werden in XPsh nicht berücksichtigt.
Zusätzlich zur Einbindung des XML-Baums, werden noch folgende Meta-Informationen im Meta-Unterbaum abgelegt (jeweils nur, falls in der XML-Datei angegeben):

- Version *(XML-Version)*
- Encoding *(Encoding, nur falls explizit angegeben)*
- DTD *(Dateiname)*

- Valid *(falls eine DTD angegeben ist, wird die XML-Datei gegen diese validiert. Das Ergebnis ist hier abrufbar)*

Diese Attribute des `meta`-Knotens befinden sich, nicht wie diejenigen der beiden vorher vorgestellten Adaptoren, in keinem Namensraum.

Kapitel 8

Test-Ergebnisse

Um abschätzen zu können, ob XPsh für den alltäglichen Gebrauch geeignet ist, werden in diesem Kapitel die Ergebnisse verschiedener Tests vorgestellt.

8.1 Messung der Ergebnisse

Bei den vorgestellten Messungen stehen die Ausführungszeiten im Vordergrund. Dies unter anderem, weil es schwierig ist unter Linux den Speicherverbrauch nachträglich festzustellen. Die Ausführungszeiten werden mit dem, in der Bash integrierten, Tool `time` gemessen. Der Speicherverbrauch wird mit `top` versucht zu ermitteln (Grösse des verwendeten Speicher = Code der Anwendung + Daten + Stack).
Um möglichst aussagekräftige Resultate zu erhalten, werden alle Tests fünf bis zehn Mal durchgeführt und dann gemittelt. Der Rest des Betriebssystems spielt keine Rolle, weil nur das Verhältnis von XPsh zu einer anderen Lösung entscheidend ist und nicht die absoluten Werte für sich.
Die zeitliche Auflösung ist mit Tausendstel zwar relativ gering aber meistens genug fein, da es sich hauptsächlich um längere Ausführungen handelt.

8.2 Einfache Tests

In einem ersten Schritt werden einfache Tests durchgeführt, welche noch nicht auf Meta-Daten oder Datei-Inhalte zugreifen.

8.2.1 Test-Umgebung

Im Root-Verzeichnis des Dateisystems befindet sich ein Ordner `xpsh-performance-test`, welcher eine Test-Umgebung beinhaltet, wie sie in Abbildung 8.1 abgebildet wird.
Jeder Ordner `dir1-10` beinhaltet selbst wieder die genau gleiche Struktur. Das ganze wird drei Mal wiederholt. Somit ist der Pfad zu den Dateien auf der tiefsten Stufe `dir?/dir?/dir?/file?.txt` (relativ aus dem Ordner `xpsh-performance-test` aus). Die Dateien `file?.txt` beinhalten 23 Zeilen Blindtext (alle den gleichen) und sind 1196 Bytes gross. Mit der Verwendung einfacher Text-Dateien wird verhindert, dass Adaptoren das Ergebnis verfälschen können.

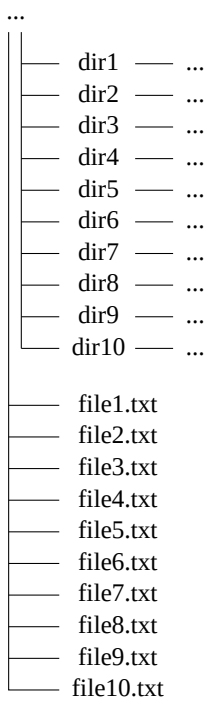

Abbildung 8.1: die Datei- und Ordner-Struktur der Testumgebung.

Zusammenfassend befinden sich somit in der Test-Umgebung 2110 Dateien `file*.txt` verteilt in 211 (Unter-) Ordnern.

8.2.2 Selektieren aller Inhalte eines Ordners

Eine einfache Aufgabe ist das Auflösen des `*`-Operators, welcher sämtliche Dateien und Ordner im aktuellen Verzeichnis selektiert. In diesem Beispiel soll verglichen werden, wieviel langsamer XPsh gegenüber der in der Bash integrierten Auflösung des `*`-Operators und dem Programm `ls` ist.
Kommando für die Bash-Variante: `echo *`, für die `ls`-Variante: `ls .`, für XPsh: `m "*"`.

	Bash	`ls`	XPsh
Ausführungszeit, Total [s]	0.000	0.008	0.033

Wie erwartet schneidet die Bash in diesem Test eindeutig am besten ab. Eine Erklärung dafür ist, dass bei den anderen beiden Varianten jeweils noch ein Programm gestartet werden muss, alleine schon das braucht mehr Zeit, als die Bash-internen Globbing-Funktionen auszuführen. Gegenüber `ls` braucht XPsh etwa 300% länger.

8.2.3 Selektieren aller Ordner und deren Inhalte

Etwas komplexer wird es, wenn die obige Aufgabe um eine weitere Stufe erweitert wird. In diesem Test geht es nun darum, sämtliche Inhalte aller Ordner zu selektieren.
In der Bash wird wiederum mit echo gearbeitet: `echo */*`, bei `ls` genau gleich: `ls *`, bei XPsh: `m "*/*"`.

	Bash	`ls`	XPsh
Ausführungszeit, Total [s]	0.001	0.018	0.188

Bereits bei dieser relativ einfachen Aufgabe zeigt sich die schlechte Performance von XPsh, da es von `ls`, wie auch von der Bash um Längen geschlagen wird.

8.2.4 Selektieren aller Nachfahren

Das nächste Problem scheint wie auf XPsh zugeschnitten. Bei dieser Aufgabe ist das Ziel sämtliche Nachfahren zu selektieren. Hier versagt zum ersten Mal die Bash. Es werden darum folgende drei Varianten verglichen:
`ls`-Programm: `ls -R`
`find`-Programm: `find . -name "*"`
XPsh: `m ".//*"`.

	`find`	`ls`	XPsh
Ausführungszeit, Total [s]	0.018	0.047	12.22
Speicherverbrauch, maximal [KB]	n/a	n/a	10380

Auch hier offenbart XPsh seine Schwächen. Neben der grossen Ausführungszeit wird auch eine beträchtliche Speichermenge (ca. 10MB) gebraucht. Ein Vorteil hingegen ist die kompakte Schreibweise, die intuitiver ist, als z.B. die von `find`.

8.2.5 Bedingte Selektion aller Nachfahren

Die Vergleichs-Aufgabe soll nun um eine bedingte Selektion der Nachfahren erweitert werden. Eine Bedingung ist die Beschränkung auf Dateien mit dem Muster `*[12].txt`, was auf `file1.txt` und `file2.txt` zutrifft. Das andere Kriterium beschränkt die Liste der Ergenisse auf Dateien, die eine bestimmte Group ID besitzen. Da alle Dateien vom selben Benutzer erstellt wurden, fallen keine Dateien weg, überprüft werden muss die Bedingung aber trotzdem. Hier versagt nun auch `ls`, darum wird nur noch das `find`-Programm mit XPsh verglichen:
`find` Kommando: `find . -name "*[12].txt" -gid 100`
XPsh: `m ".//*[[12]].txt[@gid='100']"`

	`find`	XPsh
Ausführungszeit, Total [s]	0.019	1.84
Speicherverbrauch, maximal [KB]	n/a	10280

Bei diesem Ergebnis fallen zwei Sachen auf. Während XPsh bedeutend schneller als in der vorherigen Aufgabe ist, benötigt `find` etwa gleich viel Zeit. Dazu gibt es zwei Erklärungsmöglichkeiten. Entweder ist die Sablotron-Bibliothek schlecht optimiert auf die vorherige Eingabe oder die Ausgabe des Resultats benötigt im Mapper verhältnismässig viel Zeit. Doch eine kleine Test-Modifikation im Mapper deutet klar darauf hin, dass das bessere Resultat wegen der XPath-Verarbeitung in der Sablotron-Bibliothek zustande kommt und nicht wegen der weniger aufwändigen Ausgabe.
Trotz der Verbesserung ist XPsh um einen Faktor hundert langsamer als das Unix `find` Programm.

8.3 Erweiterter Test

Im folgenden wird eine Aufgaben beschrieben und verglichen, die weiter geht als normale Navigation im Dateisystem, besonders durch den Einbezug von Dateiinhalten.

8.3.1 Selektieren bestimmter MP3-Dateien

Als Beispiel für den Einsatz des mp3-Adaptors (siehe dazu Kapitel 7.1) wird eine neue Umgebung verwendet: Im aktuellen Ordner befinden sich sieben MP3-Dateien ohne ID3-Tags, sowie fünf Ordner mit je etwa zwanzig MP3-Dateien, die z.T. ID3v2 Tags, ID3v1 oder auch beides besitzen. Die Aufgabe besteht nun darin alle MP3-Lieder einer bestimmten Band zu finden.
Kommando mit `find` und `id3v2`[1]: `find . -name "*.mp3" | while read fname; do if test "$(id3v2 -l "$fname" | grep "Lead performer.*`*`Bandname`*`")" != ""; then echo $fname; fi ; done`
XPsh: `m ".//*.mp3[@@/@mp3:Artist='`*`Bandname`*`']"`

	`find+id3v2`	XPsh
Ausführungszeit, Total [s]	1.07	0.155

Überraschenderweise schlägt hier XPsh die `find+id3v2` Lösung recht deutlich und das gerade in zweierlei Hinsicht. Zum einen ist die Syntax deutlich intuitiver, verständlicher und schneller aufschreibbar. Zum andern ist XPsh etwa sieben Mal schneller. Die Erklärung dafür ist, dass bei der anderen Variante mehrere Programme beteiligt sind, die jeweils eine gewisse Zeit nur schon zu ihrer Initialisierung benötigen.
Hier zeigen sich deshalb die Vorteile von XPsh: Die benutzerfreundliche Syntax, die dem Anwender vielfältige Möglichkeiten bietet und die Effizienz beim Einbezug von Dateiinhalten.
An dieser Stelle werden leider keine weiteren ähnlichen Vergleiche mehr präsentiert, weil es keine zu XPsh vergleichbaren Programme gibt, die noch komplexere Selektionen erlauben würden ohne grössere Shell-Skripte zu benötigen.

[1] http://id3v2.sourceforge.net

Kapitel 9

Fazit

Dass ein Programm wie XPsh durchaus Sinn macht, verdeutlicht unter anderem das Beispiel aus Kapitel 8.3.1. Demgegenüber stehen aber einige negative Punkte. In den folgenden Abschnitten wird auf die einzelnen Punkte weiter eingegangen.

9.1 Benutzerfreundlichkeit

Die XPsh-Syntax ist einfach zu erlernen und auch sehr kompakt, was zu einer hohen Benutzerfreundlichkeit führt. Die Zahlen und Operatoren, die in XPsh etwas speziell geschrieben werden müssen, könnten jedoch leicht zu unnötigen Fehlern führen und sind gewöhnungsbedürftig. Deshalb wäre es durchaus lohnenswert zu versuchen, diese Konstrukte zu umgehen und die Zahlen und arithmetischen Operator kontext-abhängig zu erkennen.
Im Moment braucht es für XPsh noch das Mapper-Programm `m`. Das hat den Vorteil, dass XPsh nur in Fällen eingesetzt werden kann, die speziell darauf zugeschnitten sind und sonst die effizienteren Algorithmen anderer Programme (z.B. `find` oder Bash-builtins) verwendet werden. Das Ziel wäre, dieses Mapper-Programm direkt in die Shell zu integrieren und dann für jede Eingabe separat zu entscheiden, ob sie mit XPsh oder mit den effizienteren in der Shell eingebauten Globbing-Funktionen evaluiert wird.

9.2 Leistungsfähigkeit

Wie im Kapitel 8 beschrieben, hat XPsh bezüglich Geschwindigkeit noch einige Defizite. Da jedoch die meisten Callback-Funktionen nicht viel effizienter gelöst werden können, wäre der nächste Schritt die Optimierung des zugrunde liegenden XPath-Prozessors. Entweder wird der XPath direkt in Libxpsh evaluiert (zusammen mit dem bereits im Mapper-Programm vorhandenen Parser) oder SXP wird auf diesen speziellen Gebrauch abgestimmt.
Ein weiterer negativer Punkt betrifft der hohe initiale Aufwand. Nur schon bis die Eingabe geparst und die Sablotron-Bibliothek initialisiert ist, vergeht unnötig viel Zeit. Für eine Lösung, wie sie jetzt vorliegt, ist dieses Problem jedoch nicht (einfach) lösbar. Wenn jedoch XPsh in eine Shell integriert wird, müsste die Sablotron-Bibliothek, wie auch Libxpsh nur einmal geladen werden, womit der initiale Aufwand verkleinert werden könnte.
Alles in allem muss aber ganz klar festgestellt werden, dass XPsh, wie es im Moment existiert, noch nicht ohne weiteres allgemein einsetzbar ist, da es noch um einiges zu langsam ist. Ganz

klar wurde aber aufgezeigt, dass die Verwendung von XPath-Konzepten in Dateisystemen durchaus Sinn macht und sehr vielfältige Möglichkeiten bietet.
Natürlich ist die XPath Shell nicht für Leute gedacht, die sämtliche Unix-Tools wie Gurus beherrschen, sondern für Leute, die XML-Konzepte im Dateisystem anwenden wollen oder einfach eine einheitliche Sicht auf das Dateisystem und die Inhalte von Dateien haben möchten.

Anhang A

Zeichensatz

A.1 Zeichen ohne Escaping

erlaubt in Pfadausdrücken sind die folgenden Zeichen (welche ohne Escaping verwendet werden):

Unicode Nummer	Zeichen	Unicode Nummer	Zeichen	Unicode Nummer	Zeichen
43	+	45	-	46	.
48	0	49	1	50	2
51	3	52	4	53	5
54	6	55	7	56	8
57	9	65	A	66	B
67	C	68	D	69	E
70	F	71	G	72	H
73	I	74	J	75	K
76	L	77	M	78	N
79	O	80	P	81	Q
82	R	83	S	84	T
85	U	86	V	87	W
88	X	89	Y	90	Z
94	^	95	_	97	a
98	b	99	c	100	d
101	e	102	f	103	g
104	h	105	i	106	j
107	k	108	l	109	m
110	n	111	o	112	p
113	q	114	r	115	s
116	t	117	u	118	v
119	w	120	x	121	y
122	z				

Tabelle A.1: Zeichen, die ohne Escaping-Backslash verwendet werden

zusätzlich kommen noch die Unicode-Zeichen 161-255, bzw. die zweite Hälfte von Latin-1, ISO-859-1 dazu, die hier nicht angegeben werden.

A.2 Zeichen mit Escaping Backslash

Unicode Nummer	Zeichen	Unicode Nummer	Zeichen	Unicode Nummer	Zeichen
32	*Space*	33	!	34	"
35	#	36	$	37	%
38	&	39	’	40	(
41	)	42	*	44	,
58	:	59	;	60	<
61	=	62	>	63	?
64	@	91	[	92	\
93	]	96	‘	123	{
124	|	125	}	126	~

Tabelle A.2: Zeichen, die ein Escaping Backslash \ benötigen

Literaturverzeichnis

[1] *EXSLT – Extensions to XSLT.* http://www.exslt.org.

[2] *id3lib – The ID3v1/ID3v2 Tagging Library.* http://id3lib.sourceforge.net.

[3] *libexif – Exif Tag Parsing Library.* http://libexif.sourceforge.net.

[4] *The XML C parser and toolkit of Gnome.* http://www.xmlsoft.org.

[5] *POSIX Standard P1003.2.* Technischer Bericht, IEEE Computer Society, 1992.

[6] *Document Object Model (DOM) Level 2 Core Specification.* Technischer Bericht, World Wide Web Consotrium, 2000.

[7] Alliance, Ginger: *Sablotron – XSLT, DOM and XPath processor.* http://www.gingerall.org.

[8] Clark, James: *XML Namespaces.* http://www.jclark.com/xml/xmlns.htm.

[9] Wilde, Erik: *Merging Trees: File System and Content Integration.* In: *WWW2006*, 2006.

[10] Wilde, Erik und Kaspar Giger: *XPsh – XPath-Shell.* http://xpsh.sourceforge.net.