Fachhochschule Zittau-Görlitz
Fachbereich Informatik

Belegarbeit:

XML mit Scheme

Datenrepräsentation mit XML
Prof. Dr. rer. nat. Christian Wagenknecht

von
Filip Martinovsky und Philipp Wagner
1.1.2008

Inhaltsverzeichnis

1. Einleitung **3**

2. Vorüberlegung **4**
2.1. Transformation in geeignete Form 4
2.2. Lokalisieren von Elementen 5
2.3. Transformation zu XML 6
2.4. Fazit zu den Vorüberlegungen 6

3. SXML **8**
3.1. XML und SXML 8
3.2. Eigenschaften eines SXML Dokuments 9
3.3. Namensräume in SXML 10
3.4. XML zu SXML und SXML zu XML 10
3.5. Vorteile von SXML gegenüber XML 10

4. Parsing eines XML Dokumentes **12**
4.1. SSAX Parser 12
4.1.1. Event Handler 13
4.1.2. Beispiel SSAX Parsing 13

5. XPath in Scheme: SXPath **16**
5.0.3. Beispiel 16

6. SXSL Transformationen **19**
6.1. Aufbau der Regeln für SXSLT 21

7. Fazit **23**

A. Grammatik von SXML **24**

B. Installation von SSAX **25**
B.1. Installation der offiziellen Quellen 25
B.2. Installation mit PLaneT Scheme Repository 26

C. Komplexbeispiel **27**

1. Einleitung

Die Extensible Markup Language *XML* als universales Austauschformat für Daten gewinnt in den letzten Jahren immer größere Bedeutung. Mit Hilfe von XML ist es möglich Daten in strukturierter Form abzuspeichern, sie mit Hilfe von XSLT zu transformieren und mit XPath zu durchsuchen. XML ist ein offener Standard der für viele Zwecke verwendbar ist, genannt seien hier SVG, RSS-Feeds, Collada oder auch DocBook. Die multiparadigmatische Programmiersprache Scheme ist in der Lehre an Hochschulen und Universitäten weit verbreitet[3] und es wäre für Dozenten von Intresse Schnittstellen zwischen diesen beiden Themengebieten zu finden. Dieser Beleg soll die aktuellen Entwicklungen in diesem Bereich analysieren und eine Anleitung geben zur XML Verarbeitung in Scheme.

2. Vorüberlegung

Um XML Dateien in Scheme verarbeiten zu können muss eine XML Notation gefunden werden die Scheme versteht. Da Scheme nativ mit Listen umgehen kann liegt es nahe zu versuchen den Inhalt eines XML Dokuments in eine Liste zu transformieren. Es ist wichtig, daß diese Transformation ohne den Verlust von Informationen möglich ist da sonst eine Rückgewinnung des Originaldokuments unmöglich wird.

2.1. Transformation in geeignete Form

Wir transformieren ein einfaches XML Dokument manuell in eine Liste.

```
<rss>
    <item>
        <title>News</title>
        <link>http://www.news.de</link>
        <description>Important News</description>
    </item>
</rss>
```

Abbildung 2.1.: Dokument in XML

```
(define rss-feed '(rss
 (item
  (title "News")
  (link "http://www.news.de")
  (description "Important News")
  )
 )
)
```

Abbildung 2.2.: Abstrakte Darstellung von XML

2.2. Lokalisieren von Elementen

Der große Vorteil einer solchen Transformation ist ersichtlich. Das XML Dokument liegt in einer für Scheme verständlichen Form vor, auf die bereits mit den von Scheme bereitgestellten Funktionen zugegriffen werden kann.

Kurt Normark nennt dies *Program Subsumption*[2], die Teile des XML Dokuments und der des Programmes liegen in einer Sprache vor.

```
> (car rss-feed)
rss
> (car (cddr rss-feed))
(title "News")
```

Auch sehr flexible Funktionen wie das pattern matching lassen sich bereits auf diese verschachtelte Liste anwenden lassen und ermöglichen es Informationen zielgerichtet zu extrahieren. Die Funktion 2.3 durchsucht den transformierten abstrakten XML Baum und gibt die title und description zurück.

```
(require (lib "match.ss"))

(define (rss->links xml)
  (match xml
   [('title text)
    (list (list "Titel: " text))]
   [('description text)
    (list (list "Beschreibung:" text))]
   [(somenode ...)
    (apply append
           (map rss->links xml))]
   [else '()]))

> (rss->links rss-feed)
(("Titel:" "News") ("Beschreibung: " "Important News"))
```

Abbildung 2.3.: Pattern Matching zum lokalisieren von Elementen

Dieser Ansatz Elemente in einem Dokument zu finden ähnelt schon dem von XPath, obwohl er bei weitem weder so intuitiv, noch so praktisch zu programmieren ist.

2.3. Transformation zu XML

Die Rückgewinnung des XML Dokuments aus Listing 2.1 gestaltet sich erheblich leichter. Da das abstrakte XML Dokument in Form von S-Expressions[1] vorliegt können wir uns die einzelnen Tags als Funktionen mit Eingabeparametern vorstellen. Diese Abstraktion erleichtert das Zurückschreiben ungemein. Bezug nehmend auf Listing 2.1 definieren wir:

```
(define rss
  (lambda lst
  (string-append "<rss>" (apply string-append lst) "</rss>")))

(define item
  (lambda lst
  (string-append "<item>" (apply string-append lst) "</item>")))

(define (title t)
  (string-append "<title>Title: " t "</title>"))

(define (link l)
  (string-append "<link>Link: "  l "</link>"))

(define (description d)
  (string-append "<description>Description: "d "</description>"))

> (eval rss-feed)
"<rss><item><title>Title: News</title><link>Link: http://www.news.de</link>
<description>Description: Important News</description></item></rss>"
```

Jeder Tag stellt also eine Funktion dar, die den Inhalt des Tags als Eingabe nimmt. Es ist schon zu diesem Zeitpunkt möglich Einfluss auf den Inhalt zu nehmen und diesen nach Vorstellung zu modifizieren. Diese Grundidee ist die gleiche die XSLT verfolgt, welche auch eine Turing-vollständige funktionale Programmiersprache ist. Jedoch muss gesagt werden, daß diese Rückgewinnung des XML Dokuments zwar in diesem kleinen Dokument gut funktioniert, für größere Dokumente muss aber ein etwas generischerer Weg gefunden werden.

2.4. Fazit zu den Vorüberlegungen

Die aufgestellten Vorüberlegungen sind die ersten wichtigen Schritte in Richtung XML-Verarbeitung mit Scheme. Es schon in diesem Stadium möglich ein abstraktes XML Dokument nach Informationen zu durchsuchen und es zu transformieren. Jedoch mangelt es diesen Vorüberlegungen an der Konformität zum XML Standard[6]. XML erscheint

[1] S-Expressions sind die Form in der in Scheme Daten und Code dargestellt werden

auf den ersten Blick zwar recht einfach und die Syntax mag in wenigen Minuten erlernbar sein, aber XML behandelt Attribute, Namespaces, Processing Instructions, Doctype Declarations - alles was in diesen Beispielen nicht berücksichtigt wird.
Es wäre wünschenswert Werkzeuge in der Hand zu haben die dem Anwender das XML Authoring mit Scheme erleichtern und gleichzeitig den XML Standard erfüllen. Es ist ausserdem von Wichtigkeit eine Schnittstelle zu modernen Technologien wie XPath und XSLT zu besitzen, damit bereits vorhandene Programme effektiv eingebunden werden können. Vor einem ähnlichen Problem stand auch Oleg Kiselyov der aus der Notwendigkeit heraus ein XML Framework für Scheme schrieb [11].

3. SXML

SXML[4] ist die von Oleg Kiselyov vorgeschlagene Form um XML Dokumente in Scheme zu repräsentieren. Die Idee hinter SXML ist nicht weit entfernt von den Vorüberlegungen aus Kapitel 1, denn SXML abstrahiert XML Dokumente ebenfalls mit S-Expressions. Ein SXML Dokument ist der abstrakte Syntaxbaum eines XML Dokuments und zugleich die konkrete Repräsentation seines XML Infosets.[4] SXML ist also neben dem Document Object Model (DOM) eine weitere Darstellungsform des XML Infosets.[1]

3.1. XML und SXML

Ein erster Schritt ist es sich die Darstellung eines XML Dokuments in SXML anzusehen. Gegeben sei ein fiktives RSS-Feed, welches mittels des im SSAX Paket enthaltenen Parsers in ein SXML Dokument transformiert wird. Der Parser ist die Instanz eines SAX Parsers auf dessen Arbeitsweise im Kapitel "SSAX" eingegangen wird.

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0">
  <channel>
    <title>Recent Package Updates</title>
    <link>http://archlinux.org/packages/</link>
    <description>Recent Package Updates</description>
    <language>en-us</language>
    <item>
      <title> Important Updates</title>
      <link>http://www.archlinux.org</link>
      <description>&gt;&gt;Get the updates!&lt;&lt;</description>
    </item>
  </channel>
</rss>
```

[1] Ein Infoset beschreibt kurz gesagt den Satz an verfügbaren Informationen in einem wohlgeformten XML Dokument.

```
(*TOP*
 (*PI* xml "version=\"1.0\" encoding=\"utf-8\"")
 (rss
  (@ (version "2.0"))
  (channel
    (title "Recent Package Updates")
    (link "http://archlinux.org/packages/")
    (description "Recent Package Updates")
    (language "en-us")
    (item
     (title " Important Updates")
     (link "http://www.archlinux.org")
     (description ">>Get the updates!<<")))))
```

Einige Unterschiede sind offensichtlich. So fehlen im SXML Dokument die schliessenden Tags, weil für sie keine Notwendigkeit mehr besteht. Die Hierarchie des Baumes ist schon in der verschachtelten Liste enthalten. Auch muss im SXML Dokument keine Rücksicht auf Entitäten wie Größer-als und Kleiner-als genommen werden.

3.2. Eigenschaften eines SXML Dokuments

- ***TOP*** ist das Wurzelelement eines jeden SXML Dokuments. Es gehört zu keinem spezifischen XML Element und kann, wie aus der SXML Grammatik ersichtlich ist, weitere Attribute wie Namensräume besitzen. Das *TOP* Element hat nur ein Kind, den Wurzelknoten des XML Dokuments.
- **Elemente** sind in einer Liste der Art $Element := (Tag(@(Attribute))Daten)$ gespeichert, wobei *Daten* entweder Inhalt darstellt oder wiederum ein *Element* ist. Die Liste der Attribute ist optional und kann auch weggelassen werden.
- **Attribute** werden in SXML als Liste dargestellt und durch ein vorangehendes @ gekennzeichnet. Nach der XML Recommendation darf kein Knoten eines wohlgeformten XML Dokuments das Zeichen @ enthalten, was Überschneidungen mit den Namen anderer Knoten verhindert.
- **Processing Instructions** müssen vom Parser auch behandelt werden und sind als *PI* Knoten in einem SXML Dokument verfügbar.
- **Kommentare** bei der Transformation von XML zu SXML werden Kommentare nicht übersetzt. In manuell erstelltem SXML werden Kommentare durch das Element *COMMENT* eingefügt.

```
(*TOP*
        (@ (*NAMESPACES*
                (namensraum1 "http://www.somewhere.com/test")
                (namensraum2 "http://www.anywhere-else.com/test")))
        (*PI* xml "version=\"1.0\"")
        (namensraum1:html
                (namensraum1:head "Head")
                (namensraum2:body "Body")
                (namensraum2:end "Ende")
        )
)
```

Tabelle 3.1.: Verwendung von Namensräumen in SXML

3.3. Namensräume in SXML

SXML stellt an sich die Anforderung die XML Recommendations zu erfüllen und muss demzufolge auch Namensräume unterstützen. Listing 3.3 zeigt die Verwendung von Namespaces in SXML. Namespaces können in der Attributsliste des *TOP* Elements enthalten sein. Dies stimmt auch mit der Definition des XML Infosets überein die diese Handhabung nicht ausdrücklich verbietet.[6]

3.4. XML zu SXML und SXML zu XML

XML in SXML umwandeln:

```
(ssax:xml->sxml (open-input-file "file.xml") '())
```

SXML in XML umwandeln:

```
(srl:sxml->xml sxml_doc)
```

3.5. Vorteile von SXML gegenüber XML

SXML Dokumente können mit einem einfach Texteditor geschrieben werden, sie können gespeichert werden oder aus Datenbanken geladen werden, was sie so flexibel wie XML Dokumente macht. Werkzeuge zum XML Authoring in Scheme sind im SSAX Paket enthalten und bieten die Möglichkeit ein vorhandenes XML Dokument in ein SXML Dokument zu transformieren. Im Anhang "Installation von SSAX" wird die Installation von SSAX unter PLT Scheme erklärt.
SXML kann XML ohne Informationsverlust darstellen, man kann sogar sagen dass es mächtiger ist. Aufgrund der Darstellung als S-Expressions verschwimmen die Grenzen

zwischen Daten und Code, bereits existierender Schemecode kann also ohne Probleme in ein SXML Dokument eingebunden werden und muss nicht umständlich nach XSLT übersetzt werden. Abbildung 3.5 zeigt die Verarbeitung eines XML Dokuments mit Scheme.

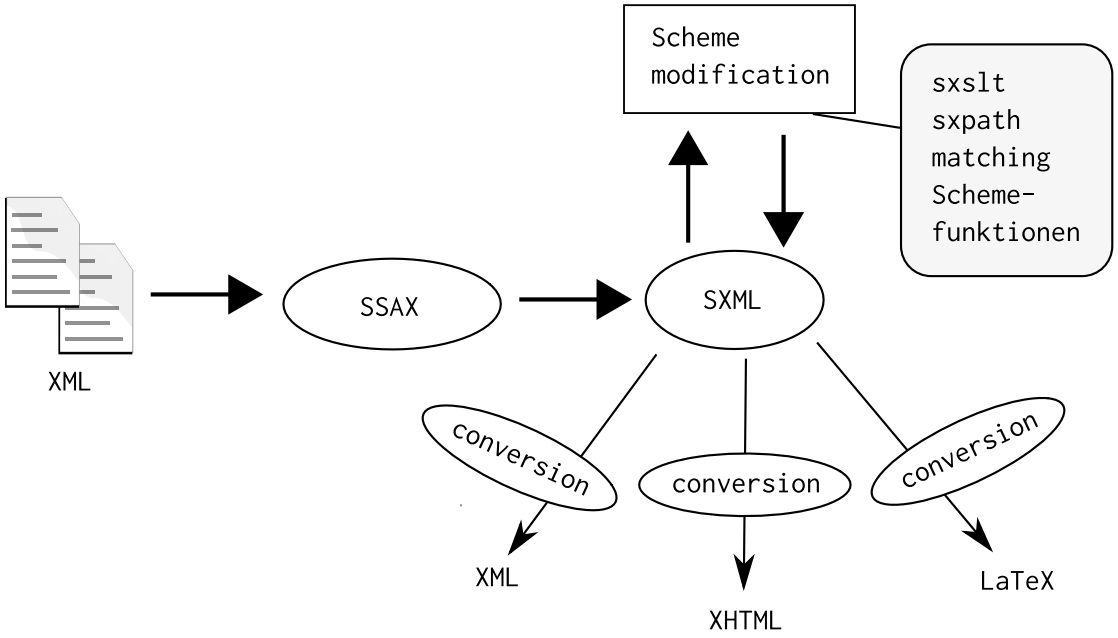

Abbildung 3.1.: Verarbeitung von XML in Scheme

4. Parsing eines XML Dokumentes

Um ein XML Dokument in Scheme zu verarbeiten, muß dieses eingelesen und in einer geeigneten Form abgespeichert werden. Das kann zum Beispiel SXML sein. Das Einlesen wird durch einen Parser durchgeführt. Es gibt zwei Arten von Parsern:

DOM-Parser erstellt ein Document Object Modell [referenz dom] im Speicher, dieses Modell wird danach im Speicher traversiert. DOM-Parser werden in Anwendungen genutzt wo ein XML-Dokument sehr oft traversiert wird, weil das Modifizieren eines Baumes sehr schnell möglich ist, was jedoch zu Lasten des Arbeitsspeichers geht.

SAX-Parser erstellt keinen[1] DOM. SAX traversiert den Baum nur einmal (z.B. um nur Bestimmte Elemente einzulesen) und hält kein Abbild des Baumes im Arbeitsspeicher. Aus diesem Grund benötigt er auch bei größeren Dokumenten wenig Arbeitsspeicher und kann dadurch auch komplexere Dokumente *on-the-fly* transformieren. Der Nachteil ist jedoch, dass das komplette Dokument bei jedem Zugriff erneut geparsed werden muss, was gerade bei größeren Dokumenten sehr zeitintensiv sein kann.

Die Wahl zwischen einem DOM-Parser und einem SAX-Parser muss je nach Aufgabe gefällt werden. Wir haben uns für SSAX als XML Prarser in Scheme etschieden weil er im Gegensatz zu anderen verfügbaren Lösungen der ausgereifteste ist.

4.1. SSAX Parser

SSAX von Oleg Kiselyov ist ein rein funktionaler SAX Parser in Scheme. Die Entwurfsidee war das XML Dokument als einen n-nären Baum zu sehen. Abbildung 4.1 und 4.3 sollen dies verdeutlichen. Der XML Baum wird mit der Tiefensuche traversiert, sind alle Kindknoten abgearbeitet wird zum Elternknoten zurückgegangen und dieser weiterverarbeitet.[10] Ein Handler wird bei jedem Einlesen eines Knotens und beim Verlassen eines Knotens (wenn alle Kinder des Nodes geparst wurden) aufgerufen. Durch ein benutzerfreundliches Interface versucht der Parser vor dem Programierer das eigentliche Traversieren und die einzelnen Schritte beim Parsing zu verstecken. Der Programierer muss nur seine Event Handler Funktionen schreiben.

[1] Ein SAX-Parser kann ein DOM erstellen und somit einen DOM-Parser erstezen, jedoch kann ein DOM-Parser nicht das Verhalten eines SAX-Parsers ersetzen.

4.1.1. Event Handler

Ein SSAX Parser definiert man in Scheme über das Makro *ssax:make-parser*. Der Benutzer kann sich seine eigenen Event-Handler registrieren, die gemaeß der SAX Philosophie als call-back Methoden aufgerufen werden. Daraus wird eine Instanz eines konkreten SSAX Parsers erstellt. Folgende Call-Back Methoden *müssen* vom Programierer definiert werden:

NEW-LEVEL-SEED *elem-gi attributes namespaces expected-content seed*
: wird aufgerufen falls ein neuer öffnender Tag gelesen wurde. Das aktuelle Element *elem-gi* hat keine Information über seine Position im XML Dokument. Die Attribute (in den Parameter *attributes*) sind als Liste von Paaren gegeben.

FINISH-ELEMENT *elem-gi attributes namespaces parent-seed seed*
: wird beim schliessenden Tag von *elem-gi* aufgerufen. Die Attribute sind die gleichen wie bei *NEW-LEVEL-SEED*. Die *parent-seed* hat den gleichen Wert wie die *seed* die beim Aufruf von *NEW-LEVEL-SEED* übergeben wurde.

CHAR-DATA-HANDLER *string1 string2 seed*
: Wird auf den Inhalt `CDATA` angewendet. Der Inhalt des Elements steht in *string1* und wenn der Inhalt zu groß ist wird *string2* mit verwendet.

Der Rückgabewert dieser Funktionen wird in die *seed* geschrieben und als Parameter beim aufruf der nächsten Funktion genutzt. Eine Ausnahme stellt der Handler *FINISH-ELEMENT* dar, dieser hat auch die *parent-seed* zur Verfügung und kann somit die *seed* vom Elternknoten mit der *seed* des aktulellen Knotens verknüpfen. Das Parsen eines Dokumentes ist so eher die Traversion eines Baumes [10] als das Scannen eines Datenstroms. Abbildung 4.3 verdeutlicht diese Idee. Ein Knoten hat beim Parsen des XML Dokumentes keine Informationen[2] über seine aktulelle Tiefe und die Elemente die bereits durchlaufen sind. Das Framework gibt dem Programierer aber die freie Wahl wie er mit der *seed* umgeht. Der so entstandene Parser prüft nur die Wohlgeformtheit eines Dokuments, jedoch kann man das Validieren auch "fest"durch Scheme-funktionen (z.b. *number?*) in den parser einbauen und auf gegebenen Kontext reagieren. Der SSAX Parser wird aus diesem Grund als “semi-validating SSAX Parser “ [10] bezeichnet.

4.1.2. Beispiel SSAX Parsing

Angenommen das XML Dokument in Abbildung 4.1 soll mit dem Parser aus Abbildung 4.1.2 verarbeitet werden. Beim einlesen eines öffnenden Tags wird der Handler `NEW-LEVEL-SEED` aufgerufen und der *seed* wird die *seed* des Vaterknotens übergeben. Im Falle des Wurzelknotens wird die *seed* beim Aufruf des Parsers festgelegt. In unseren Beispiel wären wir im Wurzelknoten *a* und die *seed* die beim Aufruf übergeben wird ist `'(*TOP*)`. Stellt man sich nun das Parsing als Baum-Traversion vor, wird der Rückgabewert des Handlers `NEW-LEVEL-SEED` von Element *a* an den Handler `NEW-LEVEL-SEED` von *b* übergeben. Der “Fluss der *seed*“ ist auch in Abbildung 4.3 gekennzeichnet. Ge-

[2] Jedoch kann diese über die *seed* aufgebaut und mitgegeben werden

```
<a>
  <b>
    <c>d</c>
    e
    <c>f</c>
  </b>
  g
  <b>h</b>
  j
  <b type="k">l</b>
</a>
```

Abbildung 4.1.: Beispiel eines XML Dokumentes

```
(define (simple-xml2sxml xml-port)
  (pars xml-port '(*TOP*)))
   (define pars (ssax:make-parser
                 NEW-LEVEL-SEED
                 (lambda (elem-gi attributes namespace expected-content seed)
                   (list elem-gi  (append '(@) attributes)));Aktuele seed -> (tag (@ attribute))
                 FINISH-ELEMENT
                 (lambda (elem-gi attributes namespaces parent-seed seed)
                   (append parent-seed (list seed)))
                 CHAR-DATA-HANDLER
                 (lambda (string1 string2 seed)
                   (append seed (list string1))); An die seed wird Text ängehangen"
                 ))
```

Abbildung 4.2.: SSAX Parser, erstellt ein einfaches SXML Dokument

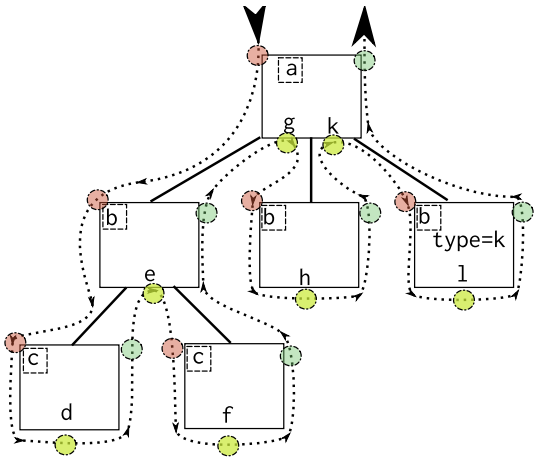

Abbildung 4.3.: Das XML Dokument aus Abbildung 4.1 als Baum, der Pfad veranschaulicht die Reihenfolge beim Parsing

hen wir nun entlang des Pfades in Abbildung 4.3 sind die roten Kreise Aufrufe von `NEW-LEVEL-SEED`, gelbe von `CHAR-DATA-HANDLER` und grüne sind `FINISH-ELEMENT` Handler. Es kann jetzt nur einer der drei[3] Fälle auftreten:

- `NEW-LEVEL-SEED` wird aufgerufen, eine neue *seed* mit den Namen des Tags und einer Liste der Attribute[4] wird erstellt.
- `CHAR-DATA-HANDLER` wird aufgerufen, *string1* wird an die *seed* angehangen.
- `FINISH-ELEMENT` wird aufgerufen, die aktuelle *seed* wird an die *seed* des Elternknotens angehangen und zurückgegeben (im Elternknoten wird mit der *seed* in den Rückgabewert weitergearbtet.)

Während der Traversion des Dokuments wird kein Stack benötigt, der die öffnenden Tags hält und die seed wird nur lokal weitergegeben, anstatt eine globale Variable zu benutzen. Die Hander des Parsers in Abbildung 4.1.2 sind keineswegs perfekt, aber sie zeigen wie einfach es ist einen eigenen Handler zu implementieren. Komplexere SSAX Parser sind unter [1] zu finden, unter anderem ein RSS-Reader der die Handhabung von SSAX in einem abstrakteren Kontext illustriert.

[3] In unserem Beispiel nur einer der drei, jedoch können auch optionale Handler für Processing Instructions, Doctypes usw. erstellt werden.

[4] Namespaces werden hier nicht berücksichtigt um das Beispiel einfach zu halten.

5. XPath in Scheme: SXPath

XPath ist die Anfragesprache von XML und verantwortlich für die Navigation in Bäumen. Das in SSAX enthaltene SXPath wurde von Dr. Kirill Lisovsky[5] entwickelt und erlaubt es Anfragen an ein SXML Dokument zu stellen. Das Ziel war es eine Sprache zur entwickeln die die XPath 1.0 Recommendation erfüllt. Die Anfragen an ein SXML Dokument sind entweder in der bekannten XPath Art mit Pfadangaben zu stellen oder in SXPath nativen Anfragen. Eine weitere Möglichteit besteht darin das *low-level-interface* zu nutzen und dadurch die Anfragen selbst zu gestalten. Bei komplizierteren Anfragen kann dies zu einer erheblichen Performanzsteigerung führen, weil die Prozedur *sxpath* nicht zuerst den Typ der Ausdruckform (XPath oder SXPath) überprüfen muss und danach diese ausführt.

5.0.3. Beispiel

Beispiel nach XML in a Nutshell [7].

Aufrufe in der Pfadform von XPath lassen sich wie gewohnt formulieren. Die folgende Funktion sucht nach dem Hobby der zweiten Person im XML Dokument.

```
> ((sxpath "people/person[2]/hobby/text()") sxml)
("Playing the bongoes")

 ;In SXPath nativer Form
> ((sxpath '(people (person 2) hobby *text*)) sxml)
("Playing the bongoes")

 ;Mit Hilfe des low-level-interface's
> ((node-reduce
    (select-kids (ntype?? 'people))
    (select-kids (ntype?? 'person))
    (node-pos 2)
    (select-kids (ntype?? 'hobby))
    (select-kids (ntype?? '*text*))
    )sxml)
("Playing the bongoes")
```

Diese Funktionen könnten an eine Variable gebunden werden und bei Bedarf an jeder Stelle im Programm benutzt werden. Die Mächtigkeit der SXPath Ausdrücke ist darin

```
<?xml-stylesheet type="application/xml" href="people.xsl"?>
          <people>
                    <person born="1912" died="1954" id="p342">
                              <name>
                                        <first_name>Alan</first_name>
                                        <last_name>Turing</last_name>
                              </name>
                              <profession>computer scientist</profession>
                              <profession>mathematician</profession>
                              <profession>cryptographer</profession>
                              <homepage href="http://www.turing.org.uk/"/>
                    </person>
                    <person born="1918" died="1988" id="p4567">
                              <name>
                                        <first_name>Richard</first_name>
                                        <last_name>Feynman</last_name>
                              </name>
                              <profession>physicist</profession>
                              <hobby>Playing the bongoes</hobby>
                    </person>
          </people>
```

Abbildung 5.1.: people.xml

begründet, dass sie die klassischen Pfadausdrücke mit Schemefunktionen kombinieren kann. Ein nicht ganz triviales Beispiel soll dies verdeutlichen:

```
(define Bsp01 (sxpath
        `(people person ,(lambda (node-set var-bind);Rechnet das Alter aus
                  (map
                   (lambda (x)
                    ((sxml:modify `("."
                              insert-into
                        ( age ,(number->string
                          (- (string->number (car ((sxpath '( @ died *text*))x)))
                          (string->number (car ((sxpath '( @ born *text*))x))))))
                          ))
                        x))
                           node-set))
)))
```

Dieser SXPath Aufruf

(*Bsp01 sxml-doc*)

durchsucht Listing 5.1 nach Personen und fügt einen Knoten “age“ mit ihrer Lebensdauer ein. Nach den Lokalisierungsschritten people und person bekommt die lambda Funktion das aktuelle Nodeset. Sie mappt nun eine Funktion auf dieses Nodeset, der das Alter berechnet und anschliessend den Knoten einfügt.

Diese Art von Änderungen sind eigentlich den Query-Sprachen vorbehalten und genau darin besteht die Effektivität der XML Verarbeitung mit Scheme: Es ist möglich fliessend zwischen den Technologien zu wählen und dabei innerhalb einer Programmiersprache zu bleiben.

6. SXSL Transformationen

Es gibt viele Möglichkeiten ein XML Dokument zu transformieren. W3C definiert dafür auch eine XML Sprache XSLT[12]. Falls wir aber das XML Dokument in SXML schon vorliegen haben, wäre es sinnvoll eine Sprache für das transformieren dieser Dokumente zu haben. Die Sprache SXSLT von Oleg Kiselyov[9] erfüllt genau diesen Zweck.

In den Vorüberlegungen hatten wir schon einmal die Idee SXML als einfache Aufrufe von Funktionen zu verstehen. Dies mag für kleine und überschaubare Dokumente durchaus praktikabel sein, für eine Verarbeitung größerer Dokumente ist sie jedoch zu aufwändig. Ein Ausweg wäre diese Idee weiterführen und etwas allgemeiner zu gestalten. Die Prozeduren für die Verarbeitung der einzelnen Knoten könnten in einer Art Tabelle gespeichert werden. Solche Aufgaben werden in Scheme meistens mit Listen bewältigt, somit kommen wir zur folgenden Ergebnis:

```
(define handlers
'((<tagname1> . ,(lambda (<tagname> <taginhalt>) <Prozedur-körper>))
  (<tagname2> . ,(lambda (<tagname> <taginhalt>) <Prozedur-körper>))
  ))
```

Jetzt müssen wir nur den SXML Baum durchtraversieren und bei jeden Knoten die dazugehörige Prozedur (auch Handler genannt) aufrufen. Um das auffinden des richtigen Handlers zu erleichtern gibt es in Scheme eine Prozedur (*assq obj alist*), die eine "Assoziationsliste" (also eine Liste von Paaren) durchsucht und das Paar zurückgibt wo *obj* an erster Stelle steht oder *#f* falls das Objekt nicht gefunden wurde. Um den Vergleich durchzuführen benutzt *assq* die Prozedur *eq?* und da wir die Tagnamen in einem SXML Dokument als Symbole gespeichert haben, ist diese genau die richtige Funktion für unseren Zweck. Es bleibt also nur noch das Quasiqoute zu erklären, da wir einen Handler aufrufen möchten, wäre es schön falls dieser in der Liste als Prozedur (#<procedure>) gespeichert wäre. Dieses ermöglicht uns das Quasiquote, weil wir genau sagen können was evaluiert wird und was nicht. Nun könnten noch Erweiterungen vereinbart werden, wie zum Beispiel ein Standard-Handler, der immer dann aufgerufen wird wenn ein Tag keinen Handler definiert. In unserem Beispiel wird der Default-Handler in der Liste unter den Tagnamen **default** gespeichert. Eine Prozedur die diese Regeln nutzt könnte wie folgt aussehen:

```
(define sxml-trans (lambda (handl doc)
 (cond
  ((null? doc) "")
  ((list? (car doc))
   (string-append (pars handl (car doc)) (pars handl (cdr doc))))
```

```
  ((string? (car doc))
   (string-append (car doc) (pars handl (cdr doc))))
  ((symbol? (car doc))
   (let ((hand (assq (car doc) handl))  ;Sucht den Handler für den Tag
         (def  (assq '*default* handl)));Sucht den Default-Handler
     (if hand ;Ausgeführt falls Tag-Handler gefunden wurde
      ((cdr hand) (car hand) (pars handl (cdr doc)))
      (if def ;Taghandler nicht vorhanden, Default-Handler
       ((cdr def) (car doc) (pars handl (cdr doc))) ;Ausführen des Def. Handlers
       (error 'kein-default-handler) ;Kein Defaulthandler - Fehler!
  ))))
  (else (error 'fehler))
)))
```

Wie man sehen kann ist diese immer noch überschaubar. Natürlich ist diese Prozedur auch sehr vereinfacht aber der erste Schritt ist getan. Ein weiterer Vorteil ist, dass diese Prozedur nur einmal definiert werden muss und mit verschiedenen Regeln unterschiedliche SXML Dokumente transformiert werden können. Die Regeln für das Beispiel 2.1 aus der Vorüberlegung sähen nun wie folgt aus:

```
(define rss-xml
 `((*default* . ,(lambda (t inh) ;Default-Handler
    (string-append "<" (symbol->string t) ">"  inh "</" (symbol->string t) ">\n")))
   (rss . ,(lambda (t inh) ;der öffnende und schließende tag auf einer neuen Zeile
    (string-append "<rss>\n"  inh "</rss>\n")))
   (item . ,(lambda (t inh) ;der öffnende und schließende tag auf einer neuen Zeile
    (string-append "<item>\n" inh "</item>\n")))))
```

Falls nur der Regelteil betrachtet wird ist dies eine enorme Vereinfachung da nur die Elemente beachtet werden müssen die nicht der **default** Regel entsprechen. Genau diese Idee steht auch hinter SXSLT. Jedoch geht SXSLT noch weiter, man kann zum Beispiel die Reihenfolge der Abarbeitung beeinflussen, lokal Regeln überschreiben oder definieren und weitere Besonderheiten.

Bei einer SXSL Transformation traversiert die Funktion *pre-post-order* den gegebenen SXML Baum. Bei jedem neuen Element sucht sie in der Liste den gegebenen Tagnamen und ruft falls nicht vorhanden, ähnlich wie bei unserer Prozedur, den **default** Handler auf. Ist auch dieser nicht vorhanden, muss der Transformationsprozess mit einer Fehlermeldung beenden. Wird eine entsprechende Regel gefunden kontrolliert *pre-post-order* ob diese Regel in der Form *(<tagname> *preorder* . <handler>)* ist. Ist dies der Fall wird der Handler *sofort* auf den Knoten aufgerufen, ansonsten wird die Transformationsfunktion auf jeden Kindknoten des aktuellen Knotens aufgerufen (ebenso wie in userer vereinfachten Prozedur), bevor dessen Handler aufgerufen wird. Der Handler sollte wieder einen Baum an seine aufrufende Funktion weitergeben, so dass diese bei einem

```
(define (remove-null ls)
  (cond
    ((null? ls)       '())
    ((null? (car ls)) (remove-null (cdr ls)))
    ((list? (car ls)) (append (list (remove-null (car ls))) (remove-null (cdr ls))))
    (else             (append (list (car ls)) (remove-null (cdr ls))))))
```

Abbildung 6.1.: Eine Einfache Scheme-Funktion zum löschen leerer Listen

rekursiven Aufruf korrekt arbeiten kann[1]. Bei der aktuellen Implementation von *pre-post-order* gibt es ein kleines Problem: es gibt keine Möglichkeit Knoten einfach zu löschen. Ein möglicher Ausweg ist es, als Rückgabewert einen leeren Baum zu benutzen. Diese Vorgehensweise hat leider einen Nachteil: will man ein SXML Dokument in ein anderes SXML Dokument transformieren und bestimmte Knoten löschen (indem mann einfach eine leere Liste zurück gibt), erhält man kein gültiges SXML Dokument. Dieses Problem kann jedoch mit einer einfachen Funktion wie in Abbildung 6 behoben werden.

Obwohl es für viele in XSLT recht schwierige Aufgaben in SXSLT einfache und elegante Lösungen gibt, existieren auch Beispiele welche mit SXSLT nur sehr mühsam lösbar sind. Ein sehr Beispiel ist das Durchnummerieren von einzelnen Kapiteln in einem SXML Dokument. Es gibt mehrere Lösungsansätze und der imperative Ansatz ist sehr schnell implementiert und ohne Probleme verständlich. Die Lösung mit dem in Scheme bevorzugten funktionalen Stil, ist auf dem ersten Blick nicht so einfach und überschaubar. Das Beispiel ist unter [1] einzusehen.

6.1. Aufbau der Regeln für SXSLT

Die Regeln einer Transformation werden wie schon erwähnt in Listen gespeichert mit folgender Struktur:

⟨bindings⟩ := binding+
⟨binding⟩ := ⟨trigger-symbol⟩ *preorder* . ⟨handler⟩ |
⟨trigger-symbol⟩ ⟨bindings⟩? . ⟨handler⟩

Folgende *trigger-symbole* (Tagnamen) sind für spezielle Zwecke bestimmt:

default wird aufgerufen falls kein passender Tag gefunden wurde

text wird auf den Text aufgerufen (ermöglicht z.b. Auskommentieren spezieller Zeichen u.ä.)

Die Regel ⟨bindings⟩? in der driten Zeile ist für das lokale überschreiben von Regeln zuständig. Wozu die option **preorder** dient wurde in 6 beschieben.

[1] Die oben definierte Prozedur könnte nur mit String rückgabewerten arbeiten aber zur einer einfachen Vorstellung über die Arbeitsweise genügt sie.

Beispiel für eine SXSLT Regel: '((tagname . ,(**lambda** (*tagname el1*)(*list el1*)))). Diese Regel würde für den Tag *tagname* aufgerufen und als Parameter würde zuerst der Tagname[2] (als Scheme Symbol) und als zweiter der "Inhalt" des Elements. Hätte das Element keinen Inhalt wird die Transformation mit einer Fehlermeldung beendet.

[2]Dieses ist sinnvoll wenn man mehrere Tags in einer Regel bearbeiten möchte, zum Beispiel die **default** Regel.

7. Fazit

Abschliessend ist zu sagen, dass SXML interessante Möglichkeiten zu XML Verarbeitung mit Scheme bietet. Durch SSAX, SXPath und SXSLT hat der Schemer die Wahl zwischen vielen Ansätzen ein Problem zu lösen und braucht nicht auf moderne Technologien verzichten. Ausserdem ist es zu keiner Zeit nötig die Programmiersprache zu wechseln, so erreicht SXML ein Höchstmaß an Homogenität, da sowohl XML als auch die verarbeitende Sprache in der gleichen Darstellung vorliegen. Dem Anwender ist es darüber hinaus durch die Public License erlaubt das SSAX Paket durch eigene Funktionen zu erweitern und muss sich nicht auf die Werkzeuge verlassen die das Paket mitliefert.

Leider ist es schwer im Internet geeignete Dokumentationen zu finden und so ist es am Benutzer viel Zeit in die Erforschung des Systems zu investieren und ein Grundverständnis des SSAX Paketes zu entwickeln. Wir hoffen, daß unser Beleg dabei eine kleine Hilfe auf dem Weg zur Verarbeitung von XML mit Scheme darstellt und das Komplexbeispiel im Anhang eine kleine Anregung dafür ist.

Sämtliche Beispiele aus dem Beleg und weitere können sie unter `http://www.inf.hs-zigr.de/~safimart` herunterladen.

A. Grammatik von SXML

[1]	<TOP>	::=	(*TOP* <annotations>? <PI>* <comment>* <Element>)
[2]	<Element>	::=	(<name> <annot-attributes>? <child-of-element>*)
[3]	<annot-attributes>	::=	@ <attribute>* <annotations>?
[4]	<attribute>	::=	(<name> "value"? <annotations>?)
[5]	<child-of-element>	::=	<Element> | "character data <PI> | <comment> | <entity>
[6]	<PI>	::=	(*PI* pi-target <annotations>? "processing instruction content string")
[7]	<comment>	::=	(*COMMENT* "comment string")
[8]	<entity>	::=	(*ENTITY* "public-idsystem-id")
[9]	<name>	::=	<LocalName> | <ExpName>
[10]	<LocalName>	::=	NCName
[11]	<ExpName>	::=	make-symbol(<namespace-id>:<LocalName>)
[12]	<namespace-id>	::=	make-symbol(ÜRI") | user-ns-shortcut
[13]	<namespaces>	::=	*NAMESPACES* <namespace-assoc>*
[14]	<namespace-assoc>	:=	(<namespace-id> ÜRIöriginal-prefix?)
[15]	<annotations>	::=	@ <namespaces>? <annotation>*
[16]	<annotation>	::=	To be defined in the future

B. Installation von SSAX

SSAX steht unter Public Domain und ist frei unter der Adresse: `http://sourceforge.net/project/showfiles.php?group_id=30687` verfügbar. Ein aktueller Port für PLT Scheme ist auch beim PLaneT Scheme Repository zu finden. Im folgenden werden sowohl die Installation der offiziellen Quellen und der PLaneT Scheme Quellen erläutert, denn nicht alle Scheme Versionen beherrschen den Umgang mit PLaneT Scheme Ressourcen und eine Offline Installation der offiziellen Quellen ist in manchen Fällen wünschenswert. Wir beziehen uns auf die aktuellste PLT Scheme Version 371, die Installationsschritte sind für ältere PLT Scheme Versionen jedoch analog zu tätigen.

B.1. Installation der offiziellen Quellen

Der hier beschriebene Weg ist ein allgemeinerer Weg, der auch für PLT Scheme Versionen > 200 funktioniert. Ein Versuch die ofiziellen Pakete mit der aktuellen Version von DrScheme zu installieren, scheitert an einem Versionskonflikt der die Installation abbricht.

1. **Installationsdatei herunterladen**
   Unter oben genannter Adresse wird die Installationsdatei `ssax-sxml-060530.plt` heruntergeladen.
2. **Wechsel in die Konsole**
   Windows:
   `Start`→ `Ausführen`→ '`cmd`' oder
   `Start`→ `Programme`→ `Zubehör`→ `Eingabeaufforderung`
   Linux:
   Wechsel in die Konsole ist je nach Distribution verschieden.
3. **Wechsel in den Pfad der Scheme Libraries**
   In der Testumgebung unter Linux ist dies `cd /opt/plt/lib/plt/collects/`,
   in einer Windows-Umgebung ist dieser Pfad unter `cd C:/Programme/PLT/collects`.
4. **Installation des PLT Files**
   wobei `[url]` der Pfad zu `ssax-sxml-060530.plt` ist.
   Windows: "`setup plt.exe` " `--force [url]/ssax-sxml-060530.plt`
   Linux: `setup-plt --force [url]/ssax-sxml-060530.plt`

5. **Kompilieren der Bibliothek**
   Windows: "`setup plt.exe`" `-r`
   Linux: `setup-plt -r`

Nun ist die Bibliothek verfügbar und kann mit:

(*require* (*lib* "ssax-sxml.ss" "ssax-sxml"))

geladen werden.

B.2. Installation mit PLaneT Scheme Repository

Ein einfacherer Weg das SSAX Paket zu installieren ist über das PLaneT Scheme Repository. Die Aufrufe:

(*require* (*planet* "sxml.ss" ("lizorkin" "sxml.plt" 1 4)))
(*require* (*planet* "ssax.ss" ("lizorkin" "ssax.plt" 1 3)))

laden die SSAX und die SXML Bibliothek herunter und kompilieren sie.

C. Komplexbeispiel

Das folgende Beispiel soll exemplarisch die Verwendung von SSAX zeigen.

Szenario:
Es liegt ein XML Dokument "geld.xml"vor, das Geldeingänge und Geldausgänge in den 4 Quartalen eines Jahres verzeichnet. Aufgabe soll es sein aus diesem Dokument eine HTML Seite zu generieren, die den Verlust bzw. Gewinn in einem Quartal anzeigt und abschliessend alle 4 Quartale bilanziert. Die Tabelle soll folgendermassen aussehen:

	1. Quartal	2. Quartal	3. Quartal	4. Quartal	
Einnahmen (Jahr)					
Ausgaben (Jahr)					
Ergebnis:					Gesamtergebnis

Die Datei *geld.xml*:

```
<Zahlungen>
        <Zahlung>
                <Zahlungstyp>Ein</Zahlungstyp>
                <Jahr quartal="1">2005</Jahr>
                <Person>Karl Mustermann</Person>
                <Betrag>100</Betrag>
        </Zahlung>
        <Zahlung>
                <Zahlungstyp>Aus</Zahlungstyp>
                <Jahr quartal="1">2005</Jahr>
                <Person>Filip Martinovsky</Person>
                <Betrag>2000</Betrag>
        </Zahlung>
        <Zahlung>
                <Zahlungstyp>Ein</Zahlungstyp>
                <Jahr quartal="1">2005</Jahr>
                <Person>Philipp Wagner</Person>
                <Betrag>2001</Betrag>
        </Zahlung>
        <Zahlung>
                <Zahlungstyp>Ein</Zahlungstyp>
```

```
                <Jahr quartal="2">2005</Jahr>
                <Person>Philipp Wagner</Person>
                <Betrag>2001</Betrag>
        </Zahlung>
        <Zahlung>
                <Zahlungstyp>Ein</Zahlungstyp>
                <Jahr quartal="3">2005</Jahr>
                <Person>Philipp Wagner</Person>
                <Betrag>2001</Betrag>
        </Zahlung>
</Zahlungen>
```

Vorgehensweise:

Zuerst muss das XML Dokument in ein SXML Dokument umgewandelt werden:

(**define** *geld* (*ssax:xml->sxml* (*open-input-file* "geld.xml") '()))

Dann folgt das Ermitteln des XPath Ausdrucks für die Einnahmen im 1. Quartal 2005. Der folgende XPath Ausdruck könnte dafür verwendet werden:

```
"//Zahlungen/Zahlung/Jahr[@quartal=1 and .='2005']
/../Zahlungstyp[.='Ein']/../Betrag/text()"
```

Dieser lässt sich 1:1 für SXPath übernehmen. Angewendet auf *geld.xml* würde er wie folgt lauten:

```
> ((sxpath "//Zahlungen/Zahlung/Jahr[@quartal=1 and
.='2005']/../Zahlungstyp[.='Ein']/../Betrag/text()") geld)
("10" "2001")
```

Die Rückgabe des sxpath Aufrufs ist eine Liste mit den Quartalseinnahmen die aufaddiert werden sollen. Der nächste Schritt ist also das Entwickeln einer Funktion die Strings in Nummern überführt und miteinander addiert.

Die Funktion könnte so definiert sein:

```
(define getEinzahlungen
  (lambda (year quartal)
    (apply
     +
     (map
      string->number
      ((sxpath
        (format
         "//Zahlungen/Zahlung/Jahr[@quartal=~a and
```

.='~a']/../Zahlungstyp[.='Ein']/../Betrag/text()" *quartal year*)) *geld*)
))))

Diese Funktion nimmt 2 Parameter: Das Jahr und das Quartal, für welches die Einnahmen zusammengerechnet werden sollen. Die Funktion für die Ausgaben ist analog definiert, nur muss der Zahlungstyp im XPath-Ausdruck auf 'Aus' gesetzt werden. Es ist zu sehen, daß der sxpath Ausdruck ein einfacher Scheme Ausdruck ist.
Anschliessend wird das Gerüst der HTML Seite in der Funktion "Seite" definiert:

```
(define Seite
 (lambda (jahr)
 `(html
   (head
    (title "Jahresübersicht")
    )
   (body
    ,(Jahresplan jahr)))))
```

Und abschliessend der eigentliche Code zum generieren der Tabelle:

```
(define Jahresplan
  (lambda (jahr)
    `(table (@ (border "1"))
      (tr (th "") (th "1. Quartal") (th "2. Quartal") (th "3. Quartal") (th "4. Quartal" ))
      (tr (td ,(string-append "Einzahlungen " (number->string jahr)))
          (td ,(getEinzahlungen jahr 1))
          (td ,(getEinzahlungen jahr 2))
          (td ,(getEinzahlungen jahr 3))
          (td ,(getEinzahlungen jahr 4)))
      (tr (td ,(string-append "Auszahlungen " (number->string jahr)))
          (td ,(getAusgaben jahr 1))
          (td ,(getAusgaben jahr 2))
          (td ,(getAusgaben jahr 3))
          (td ,(getAusgaben jahr 4)))
      (tr (td "Ergebnis:")
          (td ,(- (getEinzahlungen jahr 1) (getAusgaben jahr 1)))
          (td ,(- (getEinzahlungen jahr 2) (getAusgaben jahr 2)))
          (td ,(- (getEinzahlungen jahr 3) (getAusgaben jahr 3)))
          (td ,(- (getEinzahlungen jahr 4) (getAusgaben jahr 4)))
          (td (b ,(apply +
                    (map
                     (lambda (x)
                       (- (getEinzahlungen jahr x) (getAusgaben jahr x)))
                     '(1 2 3 4)))))))))
```

Die HTML Seite kann nun durch:

(*srl:sxml->xml* (*Seite* 2005))

erzeugt werden. Nun ist es natürlich möglich das Ergebnis wieder in eine Datei zu schreiben, genau so gut ist es auch möglich den Output an einen Webserver zu senden. Hier soll sie einfach nur angezeigt werden.

Ouput für *(display (srl:sxml->xml (Seite 2005)))*:

```
<html>
  <head>
    <title>Jahresübersicht</title>
  </head>
  <body>
    <table border="1">
      <tr>
        <th></th>
        <th>1. Quartal</th>
        <th>2. Quartal</th>
        <th>3. Quartal</th>
        <th>4. Quartal</th>
      </tr>
      <tr>
        <td>Einzahlungen 2005</td>
        <td>2011</td>
        <td>2001</td>
        <td>2001</td>
        <td>0</td>
      </tr>
      <tr>
        <td>Auszahlungen 2005</td>
        <td>2000</td>
        <td>0</td>
        <td>0</td>
        <td>0</td>
      </tr>
      <tr>
        <td>Ergebnis:</td>
        <td>11</td>
        <td>2001</td>
        <td>2001</td>
        <td>0</td>
        <td>
          <b>4013</b>
        </td>
```

```
      </tr>
    </table>
  </body>
</html>
```

Literaturverzeichnis

[1] Beispiele zu sxml-ssax, 2008. http://www.inf.hs-zigr.de/~safimart/.

[2] The laml homepage, jan 2008. http://www.cs.aau.dk/~normark/laml/.

[3] Schools using scheme, jan 2008. http://www.schemers.com/schools.html.

[4] Sxml specification (revision 3.0), jan 2008. http://okmij.org/ftp/Scheme/SXML.html.

[5] Sxpath - sxml query language, jan 2008. http://www196.pair.com/lisovsky/query/sxpath/.

[6] Xml recommendations, jan 2008. http://www.w3.org/TR/REC-xml/.

[7] W. Scott Means Elliotte Rusty Harold. *XML in a Nutshell 3. Auflage.* O'Reilly, 2005.

[8] Philipp Wagner Filip Martinovsky. Alle belegbeispiele, vortrag und weiterführende beispiele, 2008. http://www.inf.hs-zigr.de/~safimart.

[9] Oleg Kiseljov. Xml and scheme, jan 2008. http://okmij.org/ftp/Scheme/xml.html.

[10] Oleg Kiselyov. A better xml parser through functional programming, jan 2008. http://okmij.org/ftp/papers/XML-parsing.ps.gz.

[11] Kirill Lisovsky Oleg Kiselyov. Xml, xpath, xslt implementations as sxml, sxpath, and sxslt, 2002. http://okmij.org/ftp/papers/SXs.pdf.

[12] w3c. Xsl transformations (xslt).