HS-Zittau/Görlitz (FH) - University of Applied Sciences
Dept. Computer Science

XML with Scheme

Filip Martinovský, Philipp Wagner
01.01.2008

Contents

1. Introduction

The Extensible Markup Language *XML* as a universal format for data gains more and more popularity today. with the help of xml it is possible to save data in a structured form, transform ist with XSLT and use XPath to extract informations. XML is an opened Standard that can be use for many purposes. Just to mention a few popular applications there is SVG, RSS-Feeds, Collada or DocBook.

The multiparadicmatic programming language Scheme is widely used at universities [3] worldwide and it may be of interest for professors to bring both together. This document should analyse the current situation and serve as an introduction for XML with Scheme.

2. Preliminary considerations

2.1. Transformation from XML to Scheme

To work with XML Data in Scheme one has to find a notation that Scheme can understand. Everything in Scheme is a list so it seems reasonable to transform XML into lists. It's of high importance that the transformation is done without loss of information, because the transformation for Scheme XML to XML won't be possible then.

We transform an XML document into a list by hand. Please note: functions to do that automatically could be written, too. But we don't do that here.

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

Figure 2.1.: XML document

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

Figure 2.2.: Abstract represantation of XML

2.2. Locating elements

The great advantadge of such an approach is obvious. It's a syntax that Scheme can natively understand and it is possible to use the well-known functions Scheme has built-in. Kurt Normark calls this Program Subsumption[2]. Both, the XML Document and the Programming language use the same language.

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

Also more flexible functions like pattern matching can be applied to this nested list and make it possible to extract informations. The function 2.3 searches the abstract XML Tree and returns the title and description.

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

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

> (rss->links rss-feed)
(("Title:" "News") ("Description: " "Important News"))
```

Figure 2.3.: pattern matching to locate elements

Function 2.3 seems to be a little similiar to XPath, but it is in no way as intuitive and practical to program.

2.3. Transform Scheme XML to XML

The transformation from the abstract XML tree of Listing 2.1 to XML is simpler than the other way around. The Scheme XML is in form of S-Expressions[1] and we can imaginge

[1] S-Expressions are the form Scheme represents data and code

the Tags as functions with input parameters. This abstraction makes the transformation very easy you'll see.

```
(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>"
```

It's already possible at this point to modify the result of the transformation. The idea is the same that XSLT has, which is a turing complete programming language, too. Note that the transformation from Scheme XML to XML works well in this small educative example, but a more generic solution has to be found for larger documents.

2.4. Conclusion for the preliminary considerations

The preliminary considerations are the first important steps to xml parsing with scheme. We have seen that at this point we can already query an abstract xml document in form of lists, transform and modify it,but these thoughts miss the conformity to the W3C XML Standard[6]. XML seems to be very limited at first and the syntax may be clear within minutes, but XML has attributes, namespaces, processing instructions, doctype declarations - everything we left out in our examples.

Tools that make xml authoring with scheme useable and are conform to the XML Standard are needed for xml processing with scheme. It's also of importance to have interfaces to modern technologies like XPath and XSLT so developed applications can be included effectively.

Oleg Kiselyov had a similiar problem and being in need of XML Processing he wrote an XML Framework for Scheme.[10]

3. SXML

Mr. Kiselyov proposes SXML[4] as the grammar for XML Documents in Scheme. The idea behind SXML seems to resemble the thoughts in the previous chapter, because SXML also abstracts XML Documents with S-Expressions.
A SXML document is the abstract syntax tree of a XML Document. It's like the Document Object Model (DOM) another representation of the XML Infoset.[1]

3.1. XML and SXML

To get started with SXML we compare XML to SXML. Let's transform a fictional RSS-Feed into SXML with the help of a Parser from the SSAX package. The Parser is an instance of a SAX Parser which will be discussed in chapter "SSAX" .

```
<?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] In short: an infoset describes the set of informations in a well formed xml document. Please read the W3C.

```
(*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!<<")))))
```

Some differences between the two represantations are clear. The SXML Document misses the closing tags because there is simply no need for them. (XML was already designed with LISP in mind...) The hierarchie of the tree is inherent in the nested lists. It can also be noted that entities like greater-than and less-than don't need to be ampersamped anymore.

3.2. Properties of an SXML Document

- ***TOP*** is the root node of each sxml document. It doesn't belong to any specific xml element and can have attributes like the namespaces. (See the Grammar in appendix) The *TOP* Element has only one Child Node namely the root node of the XML Document.
- **Elements** are in a List of type

  `Element := (Tag (@ (Attribute) Content) Data)`

  whereas Data is either content or another Element. The list of attributes is optional and can be left out.
- **Attributes** are of course a List in SXML and identified by a preceeding @. The XML Recommendation states that no Node in a valid XML Document may use the @, so the @ node is valid to XML Recommendation too.
- **Processing Instructions** need to be parsed too and are *PI* Nodes in SXML.
- **Comments** are not processed when transforming XML into SXML. In SXML documents a comment is a *COMMENT* Node.

3.3. Namespaces in SXML

SXML is valid to the XML Recommendations so it has to support Namespaces, too. Listing 3.1 is an example for the usage of namespaces in SXML. The Namespaces are

```
(*TOP*
        (@ (*NAMESPACES*
                (namespace1 "http://www.somewhere.com/test")
                (namespace2 "http://www.anywhere-else.com/test")))
        (*PI* xml "version=\"1.0\"")
        (namespace1:html
                (namespace1:head "Head")
                (namespace2:body "Body")
                (namespace2:end "Ende")
        )
)
```

Table 3.1.: Namespaces in SXML

included in the attribute list of the *TOP* node. This is conform to the definition of the xml infoset which permits it. [6]

3.4. XML to SXML and SXML to XML

XML to SXML:

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

SXML to XML:

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

3.5. Advantages of SXML

SXML Documents can be written with every texteditor. They can be saved or loaded into any database which makes them as flexible as XML Documents. Tools for XML Authoring with Scheme are included in the SSAX Package and make it possible to transform XML to SXML. The installation of SSAX is explained in the appendix. SXML can represent XML without a loss of Information and one can say it is even more powerful. SXML being S-Expressions removes the border between data and code, so Schemecode you have already written can be included in an SXML Document and doesn't need to be translated to XSLT.

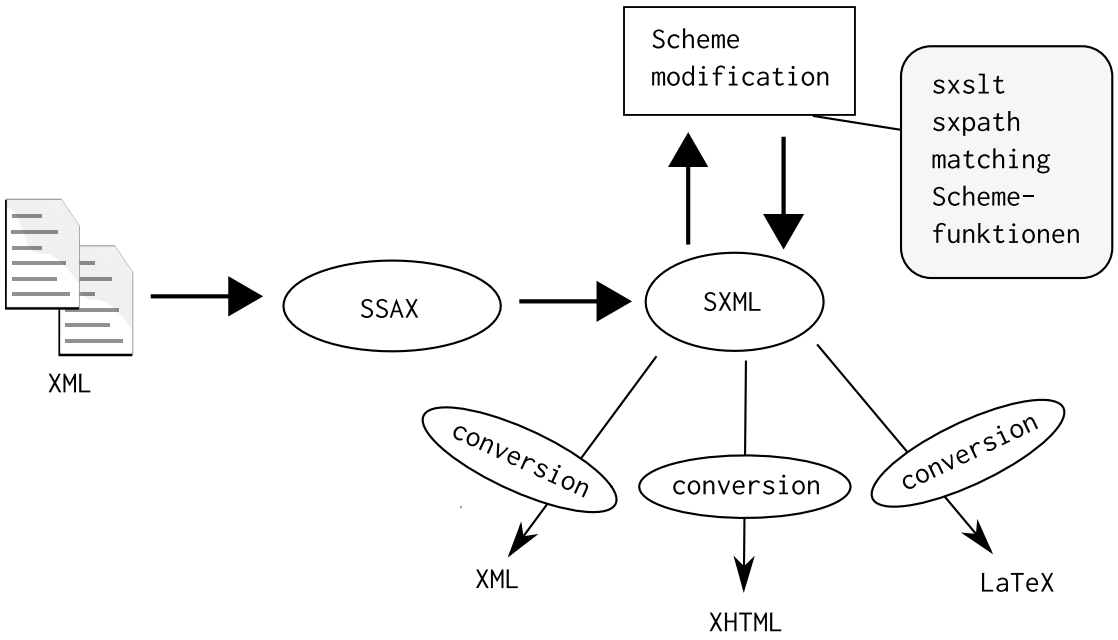

Figure 3.1.: Processing Flow of XML in Scheme

4. Parsing a XML Document in Scheme

To parse an XML Document in Scheme it has to be imported and transformed to a form scheme understands. This can be SXML. The parsing job can be done by two kinds of parsers:

DOM-Parser creates a Document Object Model withing the memory and traverses the model then in memory. DOM Parsers are used in applications where an XML Document is searched through very often, because modification of a tree is very fast. The disadvantadge of course is that the Document is in memory which can cause problems with very large files.

SAX-Parser doesn't create a tree of given document and traverses the tree only once, e.g. to read specific nodes. It doesn't hold a model in memory and can transform large documents *on-the-fly*. The disadvantage is that the document has to be read with each traversal, which can be very time consuming with large documents.

The decision to use a SAX or DOM Parser is ruled by the job that has to be done. We have chosen to use SSAX as the XML Parser for Scheme, because it is the most mature solution.

4.1. SSAX

SSAX written by Oleg Kiselyov is a fully functional SAX Parser in Scheme. The idea is to abstract the XML Document as a k-tree. Figure 4.1 should explain this. The SXML Tree is traversed depth-first. A handler is called when entering, processing (only if all child nodes have been processed) and leaving a node.[9] With an interface the Parser tries to hide the traversal and steps of parsing. The programmer only has to write the handler functions like it is done for all SAX Parsers.

4.1.1. Event Handler

A SSAX Parser in Scheme is defined with the Macro ssax:make-parser. The user can register his event-handler, that are invoked as call backs. The following callbacks *must* be defined

NEW-LEVEL-SEED *elem-gi attributes namespaces expected-content seed*

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

Figure 4.1.: Beispiel eines XML Dokumentes

is called each time a node is read in. The current element elem-gi has no information about its position in the xml document. the attributes are (in the parameter attributes) are a list of pairs.

FINISH-ELEMENT *elem-gi attributes namespaces parent-seed seed*
is called each time the current node elem-gi is left. The attributes are the same like at NEW-LEVEL-SEED. The parent-seed has the same value like the seed at NEW-LEVEL-SEED.

CHAR-DATA-HANDLER *string1 string2 seed*
is applied to the content CDATA. The content of the Element is in string1 and in string2 if it is too large.

The return value of these functions are written into the seed and used as parameters when calling the next function. An Exception is the FINISH-ELEMENT handler, which also has the seed of the parent node and can concatenate the current seed with the parent seed. So the parsing of a SXML Document should be imagined more like traversing a tree than scanning a data stream. Figure 4.3 explains this. A node has at time of processing no information about its position in the XML Document.

The parser we get from these handlers checks the well-formedness of the XML file but can also be extended by using Schemes built-in functions and that's why SSAX can also be called a "semi-validating" SAX Parser[9].

4.1.2. Examples of SSAX Parsing

Imagine we want to parse the XML Document of Listing 4.1 with the SSAX Parser of Listing 4.1.2. When reading the opening Tags the handler of NEW-LEVEL-SEED is called and the seed of the parent node is passed. In case of the root node the seed is initialized when calling the parser. In our example we would have the root node a and the seed that is passed to it is '(*TOP*). If you again think of parsing as tree traversal the return value of NEW-LEVEL-SEED of Node a is passed to the NEW-LEVEL-SEED

```
(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)));current 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)))); text will be appended to seed
                 ))
```

Figure 4.2.: SSAX Parser, creates a simple SXML document

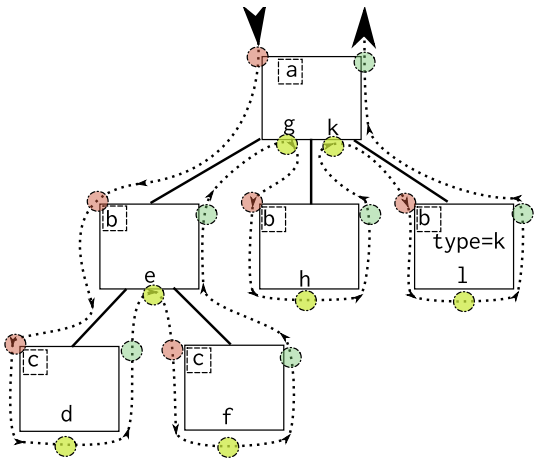

Figure 4.3.: XML Document of 4.1 as a tree. The path is the flow how parsing is done in SSAX.

handler of Node b.

This is also shown in Figure 4.3. If we follow this path the red dots are NEW-LEVEL-SEEDs, yellow CHAR-DATA-HANDLER and green are FINISH-ELEMENT handler. In our SAX Parser we have three cases:

- `NEW-LEVEL-SEED` is called and a new seed with the name of the node and a list of attributes is created.
- `CHAR-DATA-HANDLER` is called. string1 is appended to the seed.
- `FINISH-ELEMENT` is called, the current seed is appended to the seed of the parent node.

During the traversion of the document you don't need a stack holding the opened tags and the seed is only local passed instead of using a globald variable. The handlers in Listing are far from perfect, but the demonstrate how easy it is to implement your own handler. A more complex SAX Parser can be found in the Appendix, it doesn't produce SXML, but a list we later use in Scheme.

5. XPath in Scheme: SXPath

XPath is the query language of XML and responsible for navigation in XML Trees. The SXPath package that comes with SSAX was developed by Dr. Kirill Lisovsky and makes it possible to query SXML documents.[5]

The queries can either be written in the known textual XPath Syntax or with SXPath native Queries. Another possibility is to use the low-level-interface of SXPath and create Queries. The low-level-interface approach can lead to a better performance, because sxpath doesn't need to validate and transform the XPath query.

5.1. usage and examples

An example that is also used in "XML in a Nutshell".[7].

```
<?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>
```

Figure 5.1.: nutshell.xml

Queries can be formulated exactely like the XML XPath queries. The following function searches the hobby of the second person in the xml document:

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

 ;formulated as SXPath query
> ((sxpath '(people (person 2) hobby *text*)) sxml)
("Playing the bongoes")

 ; or with the help of the low-level-interface
> ((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")
```

You can also bind these functions to a variable and use them at any point in your program. The power of SXPath is that you can combine your queries with Scheme functions. A small example should demonstrate this:

```
(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))
)))
```

You can start it with:

```
(Bsp01 (ssax:xml->sxml (open-input-file "nutshell.xml") '()))
```

It now searches `nutshell.xml` for Persons and adds a node called “age“ to the result. After the localisation steps people and person the lambda function get's the nodeset. It now maps function on the nodeset that calculated the age and inserts a node afterwards.

This kind of querying would belong to Query-Languages and this makes SXML and SXPath so effective. It is possible to switch between scheme and xml, but stay in the same programming language.

6. SXSL Transformation

There are many possibilities to transform XML. W3C defines a XML Language called XSLT[11] for this purpose. But if we already have SXML it would be nice to have a language for this transformation, too. This is done with SXSLT by Mr. Kiselyov.[8]

6.1. Before using SXSLT

In the prelimanary thought chapter we hat the idea to see XML nodes as functions and modify the content by those functions. This may work for relatively small documents, but it is too complicated for larger files. So the solution is to make everything more generic. The functions for the transformations could be stored in a kind of table, in Scheme this is usually done through lists.

Our Approach:

```
(define handlers
'((<tagname1> . ,(lambda (<tagname> <tagcontent>) <procedure body>))
  (<tagname2> . ,(lambda (<tagname> <tagcontent>) <procedure body>))
  ))
```

Now we need to traverse the SXML Tree and invoke the correct function (called handler). To make the invocation of the handler easier we use the function (assq obj alist) which takes an associative list and returns the pair where obj is at first index or #f if obj is not in the list. *assq* uses the Scheme function *eq?* and because the tagname is a Symbol it is the right function for our purpose. We are using a quasiquote because the function should be stored as a procedure (*#<procedure>*) in the list. Let's go further. Imagine no tagname matches then you need a default handler which you could save as tagname *default*. A procedure using these rules could be:

```
(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))
)))
```

As you can see this procedure is still readable. Of course it is very simplified, but hey the first step is done. Another advantadge is that we define this function only once and can use it with different rules. The rules from first chapter would now look like:

```
(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")))))
```

This simplifies the transformation if you look at the rule part, because you just define the rules that should not match the default rule. This is the same approach that SXSLT uses. SXSLT goes of course further than that by making it possible to influence the order if the transformation process, overwrite or define local rules etc.

When doing a SXSL Transformation the *pre-post-order* function traverses the SXML Tree. At each new node it searches in a list for the given tagname and calls (just like in our example) the *default* handler if none matched.

If the *default* handler is not defined, then the transformation process quits with an error. If a rule was found *pre-post-order* controls if the rule is of form *(<tagname> *preorder* . <handler>)*.

Now if the rule is valid the handler is immediately called on each child node of the current node (again same like above procedures) before calling the handler for the current node. The handler passes a tree to the function so it can do the recursive call correctly. We have had a little problem while working with *pre-post-order*, we didn't find out how to simply delete a node off the tree. A possible solution is to use an empty tree as return value, but this wouldn't result in a valid SXML Document. The invalide resulting document could use the function 6.1 to be valid again.

Even though there may be problems that can be solved very easy with SXSLT insted of XSLT. But there are also examples where it would be much more easy with XSLT. A good example for this is the numbering of chapters in a SXML document. There many solutions and the imperative solution is implemented very quickly and understandably. The solution in Scheme is functional and at first sight not very easy and understandably

```
(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))))))
```

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

as you may see in the zip at Filips Website.[1]

6.2. Grammar for the SXSLT Rules

The rules for a transformation with SXSLT are saved in a list with the grammar:

$$
\begin{array}{lcl}
\langle \text{bindings} \rangle & := & \text{binding+} \\
\langle \text{binding} \rangle & := & \langle \text{trigger-symbol} \rangle \ \text{*preorder*} \ . \ \langle \text{handler} \rangle \ | \\
 & & \langle \text{trigger-symbol} \rangle \ \langle \text{bindings} \rangle ? \ . \ \langle \text{handler} \rangle
\end{array}
$$

The following trigger-symbols are used for special purposes:

default is called if no tagname matched.

text is called on the Text (makes possible to comment out special characters)

The <bindings>? in line three is for overwriting local Rules. *preorder* was mentioned in preceeding chapter.

7. Summary

SXML offers the Scheme programmers interesting tools to parse XML in Scheme. With SSAX, SXPath and SXSLT schemers have the choice between many ways to solve a problem. Due to its public license the user can extend the SSAX functionality for his own purposes and rely on the built-in scheme functions. SSAX is available for many interpreters like Gauche or PLT Scheme. We used PLT Scheme version 370 for our examples.

We noticed that it is hard for SXML beginners to find an introduction, though there is a rich amount of documents on Mr. Kiselyovs Website. We hope this document can help interested readers to get a basic knowledge on how to use the tools provided by SSAX.

There are lot's of example parsers in a zip you can find on Mr. Martinovsky's Website where this document is included, the tex file for this document and a presentation that was given. Examples include a RSS-Reader using SSAX, a SXML <-> XML Tool with SSAX and MrEd and a lot of other example.The URL is `http://www.inf.hs-zigr.de/~safimart`.

Some things were left out here like using XSL Files with SXSLT. The interested reader should follow the links on Oleg's Website which has a wide variety of SXML documents.

A. Grammar of 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-id" "system-id")
[9]	<name>	::=	<LocalName> | <ExpName>
[10]	<LocalName>	::=	NCName
[11]	<ExpName>	::=	make-symbol(<namespace-id>:<LocalName>)
[12]	<namespace-id>	::=	make-symbol("URI") | user-ns-shortcut
[13]	<namespaces>	::=	*NAMESPACES* <namespace-assoc>*
[14]	<namespace-assoc>	:=	(<namespace-id> "URI" original-prefix?)
[15]	<annotations>	::=	@ <namespaces>? <annotation>*
[16]	<annotation>	::=	To be defined in the future

B. Install SSAX

SSAX is public domain and freely available from:

- `http://sourceforge.net/project/showfiles.php?group_id=30687`

A recent port to PLT Scheme is available at the PLaneT Scheme Repository. In the following sections we explain how to install the official sources and the PLaneT Scheme Sources, just because not all versions of Scheme can use PLaneT Scheme resources and an offline installation of the official sources is maybe important for someone. We refer to the most recent PLT Scheme Version 371 as time of writing this, the Install Guide should be analog for older versions.

B.1. Install official sources

This install guide is a generic way that works well for PLT Scheme Versions > 200. If you try to install the sources with the DrScheme GUI you'll fail with a version conflict.

1. **Download setup file**
   Get `ssax-sxml-060530` from above Sourceforge Link.
2. **Switch to a terminal**
   Windows:
   `Start`→ `Run`→ '`cmd`' oder
   `Start`→ `Programs`→ `Accessories`→ `Command Line`
   Linux:
   Switch to a terminal of your choice.
3. **cd to the path of Scheme libraries**
   In our Linux Distribution this is done by `cd /opt/plt/lib/plt/collects/`, in Windows this would be `cd C:/Programme/PLT/collects`.
4. **Install the PLT File**
   where as `[url]` is the path to your `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. **Compile the Library**
   Windows: "`setup plt.exe`" `-r`
   Linux: `setup-plt -r`

Now the Library is available for PLT Scheme and can be loaded with:

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

.

B.2. Install with PLaneT Scheme Repository

The easier way is to install SSAX with the PLT Repository. This is done by:

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

Now SSAX and SXML get downloaded and compiled for your PLT Scheme version. Note that again these are the most recent version while writing this document.

Bibliography

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

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

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

[11] w3c. Xsl transformations (xslt).