XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT

Oleg Kiselyov
Fleet Numerical Meteorology and
Oceanography Center
Monterey, CA

Kirill Lisovsky
MISA University
Moscow

International Lisp Conference: ILC 2002.
October 27-31, 2002. San Francisco, CA.

`http://ssax.sourceforge.net/`

This talk {– XML, XPath, XSLT implementations as SXML, SXPath, and SXSLT –} is by Kirill Lisovsky of MISA University, Moscow, and by myself. We will talk about practical examples and our and other people experience with SXML tools. The emphasis of this talk is not on How but on What and Why. The paper and the project web site give all the technical details.

XML Programming

{Extra slide}

The scope of this talk is XML programming, Perhaps I need to define what I mean.

For one thing, XML programming means authoring of markup documents. Often, we write in one markup language – XML – and transform into another: HTML, TeX or PDF. These transformations are frequently written in a language XSLT, which the W3 Consortium has designed for that purpose.

Besides transformations, we are often interested in selections: selections of some interesting parts from a large XML document according to certain criteria. If an XML document is considered a database, these selection operations are queries. There is a special W3C language for that: XPath.

Incidentally, XSLT has XML syntax, with angular brackets and such. In other words, XSLT is a programming language with XML syntax. XPath does NOT have XML syntax.

We can also use a general-purpose programming language for creating XML documents and taking them apart. Sometimes, we have to, because XSLT is emphatically not a general-purpose programming language. Such languages {Perl, Java, C++ – on the slide} can work on the text of an XML document – in other words, concrete syntax. Or we can deal with a so-called document object model, which is an abstract data type for an XML document.

Languages for handling XML

Adam Bosworth:

"I noticed two new categories of books: books about programming with Java and XML and books about XSLT. In both cases, whole forests are dying to compensate for the XML community's one great failure – the lack of a decent programming model for manipulating XML."

...

"We need a language that can natively support XML as a data type and yet can gracefully integrate with the world of objects and can take advantage of the self-describing nature of XML by supporting querying of its own variables. This language as used by humans will look like a programming language, not an XML grammar. This is the language we will use to convert from one XML format to another."

Adam Bosworth is a vice president for engineering of the leading Web Services vendor BEA Systems Inc., the maker of a WebLogic web application server. In February 2002 he wrote an article in the XML Magazine, in which he outlined the problem with XML processing: "whole forests are dying to compensate for the XML community's one great failure – the lack of a decent programming model for manipulating XML. We need a language that can natively support XML as a data type and yet can gracefully integrate with the world of objects and can take advantage of the self-describing nature of XML..." Adam Bosworth then went to say that XSLT is usable only because of a compromise that allowed non-XML expressions in the grammar of XML Path Language. XPaths simply were unreadable otherwise. It turns out, he wrote, that there are things that XML isn't well suited for, and expression languages that humans can read and write easily is one of them.

Languages for handling XML...

"I contend that until we have a language that natively understands the data structures inherent in XML and enables optimized algorithms for processing it, we will not have real XML programming."

Adam Bosworth: Programming Paradox. XML Magazine, February 2002.

His article finishes with the phrase: "I contend that until we have a language that natively understands the data structures inherent in XML and enables optimized algorithms for processing it, we will not have real XML programming."

I contend that we do have such a language that natively supports XML data structures and their efficient manipulations and transformations – and also reflection. The language was specifically designed for processing of semi-structured data. The title of the founding 1960 paper 'Recursive Functions of Symbolic Expressions' makes this point clear. The 40+-year history of the language proved that it and its syntax are quite suitable for describing both data and code.

Symbolic expressions

S-expression data

S-expression is:

- a text string, a number, or a symbol
- a sequence of S-expressions in parentheses

```
(*TOP* (*NAMESPACES*
        (H "http://www.w3.org/TR/REC-html40"))
 (RESERVATION
  (NAME (@ (H:CLASS "largeSansSerif"))
      "Layman, A")
  (SEAT (@ (H:CLASS "largeMonotype")
           (CLASS "Y")) "33B")
  (H:A (@ (HREF "/cgi-bin/ResStatus"))
       "Check Status")
  (DEPARTURE "1997-05-24T07:55:00+1")))
```

{Another extra introductory slide}

We will be talking a lot about S-expressions. What are S-expressions? An S-expression is defined as a text string, a number, or a symbol. Or as a sequence of S-expressions in parentheses. Clearly, this is a recursive data type.

On this slide, this {"largeMonotype"} is an s-expression: it is a text string. This {`H:CLASS`} is another one: a symbol. Here {`(H:CLASS "largeMonotype")`} is the sequence of primitive S-expressions, in parentheses. This {`(NAME (@ (H:CLASS "largeSansSerif")) "Layman, A")`} is another sequence of smaller S-expressions enclosed in parentheses. One of its components – a child expression – is a parenthesized expression itself. This whole thing {`(*TOP* ...)`} is the S-expression by itself.

S-expression code

```
((node-join
   (node-closure (node-typeof? 'RESERVATION))
   (node-reduce
     (select-kids (node-typeof? 'SEAT))
     (filter
       (node-join
         (select-kids (node-typeof? '@))
         (select-kids
           (node-equal? '(CLASS "Y")))))))
  sxml-tree)
```

Scheme programming language:

http://www.schemers.org

{Another extra introductory slide}

The S-expression on the previous slide represented data. The S-expression on this slide stands for code. This is a program, in a language called Scheme. Scheme and other languages of the Lisp family have S-expression syntax.

Scheme – which will be mentioned often – is a general purpose mature programming language. Incidentally, it is quite often used in elite education, including high-school education. This web page, schemers.org, is a treasure trove of resources, including various free and commercial implementations. Scheme implementations run on anything you can imagine, including Palm Pilot and Sharp Zaurus.

Generative markup

```
print +(header(-cookie => $cookie),
  start_html("Missing cookies"),
  h1("Missing cookies"),
  p("This site requires a cookie to be set."),
  startform, submit("OK"), endform,
  end_html);
```

Randal Schwartz, "Basic Cookie Management", Web Techniques Column 61 (May 2001)

Thus S-expressions can represent data and code. One aspect of this code-data dualism of S-expressions is S-expression-based data being generated by S-expression-based code. This seems to be a fundamental principle, because it shows up in unexpected places, for example, in Perl CGI programming.

This code generates HTML. HTML is also an S-expression. Indeed, a markup document is either a piece of text, or something with tags around, or the sequence of the above. That something with the tags around can itself be either a piece of text, something with the tags around, or the sequence of the above.

This code that generates HTML markup looks like an S-expression itself, only with parentheses in different places. This code only *looks like* an S-expression, but is not truly an S-expression. It's somewhat verbose for one thing. Mainly, the code can't be easily queried and manipulated as data.

Use the real S-expressions throughout?

DATA		CODE
xML		Perl, Java, ...
	PHP, ASP JSP ... XSLT/XPath	

But in reality we have this divide. On one end, we have a great number of legacy markup documents. These are *data*, in a verbose S-expression format. Some processing tools, XSLT, share the same verbose format. However a critical component of these tools, XPath, is *not* in the XML format itself – and we saw why. Also, XSLT is emphatically not a general purpose programming language.

On the other end, we have programming languages for processing XML. As we saw, the code that makes an S-expression data often looks like an S-expression. Alas, this S-expression code is often too verbose. Furthermore, this code cannot easily be handled as data in the same language.

Finally, there is a mixed mode. Markup data are written in a markup language; query and transformation code is written in another, programming language. PHP, ASP, JSP and other Server pages are the examples of this mixed-mode processing. {if there is a chance, show an excerpt from any JSP or PHP page}. This approach is rather popular: however poorly, it separates authoring of content from writing the code. The result of such mixed authoring is an unwieldy jumble of two languages. The mixed approach does not fill this divide – it merely covers it. The linguistic gap still remains underneath – which makes for an excellent trap.

We would like to use genuine S-expression throughout – both for data and for processing code. I don't need

to explain the advantages in this audience. Kirill and I however regularly encounter a different kind of audience. We do have to explain the S-expression way to sceptical colleagues.

Selling S-expressions

Challenges:

- Interoperability with legacy systems and legacy *programmers*
- Play by the W3C rules
- Reuse the existing expertise and skills
- Do something better

Mainly, we have to explain our S-expression approach to sceptical managers. First the managers want to know if we can play with the rest of the team. Can we parse and create valid markup documents? Can we use already developed XSLT stylesheets and other markup documents? How steep is the learning curve for this new technology? Finally, what can we do better? Can we do something that was too unwieldy, too difficult to write or too difficult to maintain with the traditional tools?

W3C-compliant XML processing in Scheme

- Motivation
- SXML
  - XML AST
  - Inter-conversions
- XML and SXML queries: (S)XPath
  - SXPath as an XPath processor
  - Powerful SXML queries
- XML and SXML transformations
  - Interoperability with XSLT
  - Advanced SXML features
  - Web services as SXML transformations

To earn the acceptance within the XML community, we should show that we can play by their rules. We can take and produce their data. We can run the same queries and transformations – and many more. Therefore, the emphasis of this talk is on two points: W3C-compliance and powerful extensions.

This is the outline for the rest of the talk. First we will introduce SXML, which is an abstract syntax of an XML document. SXML is also a concrete representation of the XML Infoset in the form of S-expressions. All our query and transformation tools work on SXML data structures. Speaking of W3C-compliance, SXML fully supports XML Namespaces, processing instructions, parsed and unparsed entities. Our XML parser and pretty-printing tools convert between SXML and the angular-bracket-format of XML documents.

We will discuss SXML queries: XPath and SXPath. SXPath can do everything XPath can – because an SXPath interpreter is a compliant XPath processor. SXPath tools can also do more.

Finally, we will talk about SXML transformations – about interoperability with XSLT and powerful extensions.

XML and SXML

```
<RESERVATION
 xmlns:HTML='http://www.w3.org/TR/REC-html40'>
<NAME HTML:CLASS="largeSansSerif">
    Layman, A</NAME>
<SEAT CLASS='Y'
  HTML:CLASS="largeMonotype">33B</SEAT>
<HTML:A HREF='/cgi-bin/ResStatus'>
    Check Status</HTML:A>
<DEPARTURE>1997-05-24T07:55:00+1
</DEPARTURE></RESERVATION>
```

```
(*TOP* (*NAMESPACES*
        (H "http://www.w3.org/TR/REC-html40"))
 (RESERVATION
  (NAME (@ (H:CLASS "largeSansSerif"))
      "Layman, A")
  (SEAT (@ (H:CLASS "largeMonotype")
           (CLASS "Y")) "33B")
  (H:A (@ (HREF "/cgi-bin/ResStatus"))
       "Check Status")
  (DEPARTURE "1997-05-24T07:55:00+1")))
```

As we said, at the core of our tools is SXML, which is an abstract syntax tree of an XML document. SXML is also a concrete representation of the XML Information set in the form of S-expressions.

This example eliminates the need for any further explanations. We see an XML document and the corresponding SXML representation. The XML document on the slide is actually taken from the XML Namespaces Recommendation. You can see how SXML represents attributes {(@ ...)} and namespaces. In SXML, all names are fully resolved. {Because namespace (Universal Resource Identifiers) URIs are often too long, it's possible to tell the parser to use shorter namespace shortcuts, in this example, `H`. Note that these shortcuts have nothing to do with the XML prefixes in the original document {`HTML:CLASS`, etc.}. The XML prefixes are controlled by the author of the document; the shortcuts {`H:CLASS`} are controlled by the developer of an XML application. Furthermore, shortcuts and namespace URIs are in a 1-to-1 correspondence. This is not the case for XML prefixes.}

XML Queries

```
<RESERVATION
 xmlns:HTML='http://www.w3.org/TR/REC-html40'>
<NAME HTML:CLASS="largeSansSerif">
    Layman, A</NAME>
<SEAT CLASS='Y'
  HTML:CLASS="largeMonotype">33B</SEAT>
<HTML:A HREF='/cgi-bin/ResStatus'>
    Check Status</HTML:A>
<DEPARTURE>1997-05-24T07:55:00+1
</DEPARTURE></RESERVATION>
```

Query

```
//RESERVATION/SEAT[@CLASS='Y']
```

Let us turn to XML queries. Here's the sample XML document from the previous slide: a reservation manifest. Suppose we want to query this manifest to find out all seat assignments of the class Y. The following XPath expression, when executed, will give us the answer. XPath is a simple W3C XML query language, which is used in XSLT, XPointer, and XLink. This XPath expression {`//RESERVATION/SEAT`} locates all the SEAT elements within the reservation manifest, and this expression in brackets filters only those that have the attribute CLASS with the value of `Y`.

(S)XPath abbreviated

XPath abbreviated

```
//RESERVATION/SEAT[@CLASS='Y']
```

SXPath abbreviated

```
(// RESERVATION (SEAT (@ (equal? (CLASS "Y"))))
```

XPath is a basic query language of XML, which is used to access an abstract XML data structure. Queries over SXML documents are likewise expressed in SXPath. Just as SXML is closely related to XML, so is SXPath to XPath. {XPath addresses an abstract XPath data structure; SXPath queries SXML, which is a concrete representation of the XPath data structure.} As you see, the format of SXPath is rather similar to that of XPath, modulo path delimiters and parentheses.

Full form (S)XPath

XPath

```
/descendant-or-self::node()
   /child::RESERVATION
      /child::SEAT[attribute::CLASS='Y']
```

SXPath

```
((node-join
   (node-closure (node-typeof? 'RESERVATION))
   (node-reduce
     (select-kids (node-typeof? 'SEAT))
     (filter
       (node-join
         (select-kids (node-typeof? '@))
         (select-kids
            (node-equal? '(CLASS "Y")))))))
  sxml-tree)
```

XPath also has a full form, like this {at the top of the slide}. Not surprisingly, SXPath has a full form, too, like this {the second part of the slide}. The abbreviated form of XPath is defined as a shorthand notation for the full form. Likewise, the abbreviated SXPath expression we saw before is mechanically translated into this full form.

Both expressions – this {full-form XPath} and this {full-form SXPath} have to be *evaluated* to give any result. In the case of the full SXPath, the evaluator is Scheme itself. Indeed, the full SXPath notation is a composition of ordinary Scheme functions: predicates, filters, selectors and combinators. The full SXPath is a library of such functions. This expression is a Scheme application, which yields the answer to the query. As a matter of fact, we can see two applications here: this one {`(node-join ...)`} gives us a procedure, which, applied to the SXML tree, will run the query and return the list of matching nodes. In database-speak, this whole expression combines the step of preparation of a query {`(node-join ...)`} with that of execution of a query {`((node-join ...) sxml-tree)`}.

(S)XPath abbreviated

XPath abbreviated

```
((txpath
   "//RESERVATION/SEAT[@CLASS='Y']")
 sxml-tree)
```

SXPath abbreviated

```
((sxpath
   '(// RESERVATION
         (SEAT (@ (equal? (CLASS "Y")))))
 sxml-tree)
```

We saw that abbreviated XPath and SXPath are rather similar. Can we mechanically translate one to the other? Yes, we can. Here is an example of executing the SEAT query with our tools. We can either use the XPath expression as it is, written as a text string, or we can use the corresponding S-expression. Here `txpath` is a XPath interpreter (compiler, actually), and `sxpath` is an interpreter for the abbreviated SXPath. Both interpreters run on the top of the full SXPath, which can be considered bytecode, or SXPath virtual machine, so to speak. If we use this form {txpath}, our tool is a compliant XPath processor. It can handle all location paths defined in the XPath Recommendation, complete with preceding-sibling, ancestor, following-sibling axes and other fun stuff. So we indeed play by the rules and indeed can re-use the existing XPath writing skills.

{Again, this expression {(txpath ...)} prepares the query, and this outermost application executes the query.}

But why do we need the abbreviated SXPath {(`sxpath '(...)`)}? Well, we can do more. Instead of quote, here, there can be a quasiquote. The sxpath interpreter permits arbitrary Scheme procedures used as selection and filtering predicates. So the full power of Scheme is available during the selection, if we need it. For example,

Complex SXPath

Selecting passengers with confirmed reservations

```
((sxpath
  `(// (RESERVATION (
    ,(lambda (res-node)
       (run-check-status
            ; URL of the confirmation script
        (car ((sxpath '(H:A @ HREF *text*))
              res-node))
        ((select-kids
            (node-typeof?
              '(NAME SEAT DEPARTURE)))
         res-node)))
   )) NAME *text*)) document)
```

SXPath expressions can be more complex: for example, given a booking record we can select the names of the passengers with confirmed reservations as follows {on this slide}. This part {`// RESERVATION`} retrieves all reservations. This custom predicate { `(lambda ...)`} filters out unconfirmed reservations. This part {`NAME *text*`} selects the names of the passengers from the reservations that remain after the filtering. This lambda-expression is a custom predicate: given a reservation node, the predicate will return `#t` or `#f` depending on the reservation status. Recall that a reservation element had a link to a status-check CGI script. Here, {`(sxpath '(H:A @ HREF *text))`} we fetch the script URL from the HREF attribute of an anchor element. This expression {`(select kids ...)`} obtains the passenger's name, seat, and the departure time. The function run-check-status runs a HTTP transaction with the checker URL and these parameters. As you see, our custom predicate uses ordinary Scheme functions {`car`} and invokes the sxpath interpreter recursively. We also take advantage of the full SXPath library functions here {`(select-kids ...)`}.

This example illustrates the degree of integration of the SXML query language into Scheme. There is no linguistic barrier any more.

Integration of (S)XPath into Scheme

- Full compliance with the XPath Recommendation
- (S)XPath can be used in any Scheme function
- Any Scheme function can be used in SXPath
- No linguistic barrier

The ability to use Scheme functions as SXPath predicates, and to use SXPath selectors in Scheme functions makes SXPath a truly extensible language. A user can compose SXML queries following the XPath Recommendation – and at the same time rely on the full power of Scheme for custom selectors.

W3C-compliant XML processing in Scheme

- Motivation
- SXML
  - XML AST
  - Inter-conversions
- XML and SXML queries: (S)XPath
  - SXPath as an XPath processor
  - Powerful SXML queries
- XML and SXML transformations
  - Interoperability with XSLT
  - Advanced SXML features
  - Web services as SXML transformations

Let us now turn to XML and SXML transformations. We will use the most familiar examples of Web site authoring. Again, we will talk about reusing existing stylesheets and existing skills of writing them. We will show that the migration from XSLT to Scheme-based tools is beneficial, and relatively smooth.

Patterns of SXML transformations

XML - SXML -*- SXML - xML, DB, File

In particular:

XML $\rightarrow$ SXML $\rightarrow$ SXML $\rightarrow$ xML, LaTeX, DB

SXML $\rightarrow$ SXML $\rightarrow$ xML, DTD, LaTeX

The general pattern of SXML transformations can be written like this {first phrase}. We can apply it in both directions.

We can start with XML, parse it in SXML and do various transformations and queries. We store the result as a new XML or HTML or LaTeX document, or put the data into a database. This approach resembles the ordinary XSLT processing. And our tools can indeed do the standard XSLT – and more.

On the other hand, we can start with SXML. That SXML can be generated from a database query, retrieved from a file, or entered by hand in Emacs. We can transform it several times – sometimes in a rather intricate ways – and generate a markup or a TeX document. Incidentally this approach lets us author web pages, XML documents, or papers – in SXML. The SXML specification, for example, was authored in SXML, and later converted into a LaTeX document and a web page, given different stylesheets. The present paper in the proceedings and the slides you are seeing were also authored in SXML.

Let us see, how we can do all this and still play by the rules.

SXML for web site construction

Yahoo! Pick of the Day for May 30

http://www.censusscope.org/

Web site construction offers a good illustration of both points, compatibility and advance features. Here's one example: a nice social science site, with maps and charts based on US census data. This site was *The Yahoo! Pick of the Day* for May 30, 2002.

SXML for web site construction...

"Today's Yahoo! Pick (http://picks. yahoo.com) was constructed using PLT Scheme and Oleg Kiselyov's `SXML->HTML`. (Charts and maps were made with other tools, but Scheme was used to turn these raw materials into a web site.) The ease of development has been extraordinary."

Bill Abresch: for those keeping score...
A message on the PLT-Scheme mailing list. May 30, 2002.

On the same day, one of the developers of the CensusScope site posted the following message on the PLT Scheme mailing list. The message had a catchy title: for those keeping score...

I especially appreciate the last phrase of the message: "The ease of development has been extraordinary." I have never met Bill Abresch, nor communicated with him. He figured out how to use the SXML tools all by himself – to his advantage, it seems.

SXML for web site construction...

http://ir.misis.ru/english/about/general.htm

Moscow State Institute of Steel and Alloys

Technological University

Education for foreigners

General

Rector's preface

Mission and priorities

History

Awards

Facts

Location and map

Campus life

Administration

Important links

Search

Contact us

MISA is the leading institution of Higher Education, training engineers and researchers in:

- metallurgy and material science;
- metal production and treatment;
- composite, powder, super- and semi-conducting materials;
- developing advanced materials and technologies;
- raw material effectiveness and ecology;
- certification and quality management of materials;
- economics and management;

Here is another example. It is a rather official site, of a large educational institution: approximately 15 thousand students in Moscow alone. This institute is one of the top ten technical universities in the whole Russia. As you see, the web site follows the design that fortunately becomes the norm for serious sites: the site is accessible, clear and easy to navigate. Alas, colors in navigation bars didn't come out on this slide.

SXML for web site construction...

http://ir.misis.ru/english/about/general.htm

Page source

```
...
<P>
<CENTER>
<B>4 Leninsky prospekt, 119991 Moscow, Russia <B
tel. 7 095 237 22 22 <BR>
fax 7 095 237 80 07 </B>
</CENTER>

</div></TD></TR></TABLE>
</BODY></HTML>
 <!-- Generated by Beaver
 $Id: beaver.scm,v 2.52 2001/04/12 07:32:35
```

If we take a look at the source HTML, we will see a telltale sign at the end of the page. This audience doesn't need any explanation what the file suffix .scm stands for. You can also see that we've been using our tools for quite a while {emphasize the date}.

Building the MISA site

```
<PAGE NAME="Rector's preface"
      LPATH="about preface">

<img src="image/rector1.jpg" width="131"
     height="200" align="right"
     hspace="15" vspace="10" alt="" />

 <p>
 Based on fundamental sciences, Russian
 Higher Education of training
 Engineers in metallurgy and material
 science is given recognition
 throughout the world.
 </p>
...
</PAGE>
```

Let me stay on this example for a bit longer. It makes a good illustration of our playing in a team. Previously, the MISA site was an ad hoc conglomerate of various HTML fragments, written and maintained by a number of people in different departments. Often these people aren't programmers. But they do know HTML.

Eventually it came time when a common look-and-feel had to be introduced. An official site should look consistent and reputable. Furthermore, it should be easy to present the content in different formats, for example, a brochure or a CDROM. The old content – and its authors – should also be re-used as much as possible. As the first step, all the old content was turned to XHTML. XHTML is rather close to HTML, so the change wasn't much of a burden. This slide shows the source for a typical simple page. Local web designers are free to use _any_ XHTML tags (which are all in lowercase, as in here {`img`}). In addition, the designers use *a small number* of integration tags, in uppercase – like this {`PAGE`}.

Weaving the MISA site

```
<stx:load href="misa.scm"/>
<stx:import href="misa.stx"
     prefix="misa" type="stx"/>
<xsl:import href="misa.xsl"/>

<xsl:template match="PAGE">
...
<table cellpadding='10' width='100%'><tr valign=
<!-- Sub-menu -->
<td valign='baseline' bgcolor='#006699' width='1
<misa:v-menu/>
<misa:search-and-mail hidden="no"/>
</table>

<misa:info-files/>
<div align='justify'>
<xsl:apply-templates/>
</div> ...
```

The weaving of such fragments into the site was done in XSLT. Well, more than just XSLT. On this slide is the master stylesheet. Some of the imported files, like this one, `misa.xsl`, are truly XSLT. They were already developed by professional designers, and we had to use them. Some other stylesheets were more than XSLT. For example, this template {`<xsl:template`} : it looks like ordinary XSLT, with `<xsl:apply-templates/>`, etc. You can also see custom tags such as `<misa:v-menu/>` or `<misa:info-files/>`. They stand for Scheme functions, which are imported in here {`<stx:import...`}. This statement, {`<stx:load...`}, loads a number of Scheme utility functions. All this pure XSLT {`misa.xsl`} and the Scheme-extended one are translated into a Scheme code with SXML traversal combinators. XPath expressions of the XSLT stylesheets are translated into SXPath expressions, in the manner we saw earlier.

Note, this master template is merely XHTML with custom tags. The experience showed that it is far more natural for a web designer to use HTML-like markup tags (with the interface he himself defined) rather than heavyweight xsl:call-template etc. constructs of XSLT. When authoring documents, declarative approach is preferable. To a web designer, the content, the presentation, and special actions are just the markup. To a developer, the content, the presentation, and the actions are uniformly Scheme expressions. We can mechanically translate between these two views.

{Actually some of the "content providers" were not (and still not!) familiar with HTML, but know ... LaTeX! So, for them a basic set of LaTeX-like markup was prepared (ITEMIZE, and so on), which made such users quite happily.}

{The same XML information is used for Web site generation (since early 2001) and for customized HTML presentation distributed as an official MISA advertisement CD (since this summer).}

{The MISA HTML pages are static. STX can also be used for dynamic web pages – Scheme Server Pages – with a PLT Scheme web server, for example. Like JSP, Scheme Server pages provide custom tag libraries. Unlike, JSP, Scheme server pages are purely declarative. They do not mix markup and programming languages. There is no impedance mismatch in Scheme Server Page programming.}

More details on STX may be found in a "STX position paper" `http://pair.com/lisovsky/STX/`

Advanced SXML transformations

- `(tag "text") ==> <tag>text</tag>`

- ```
  (section "text") ==>
  (div (@ (align "center")) (p "text")) ==>
  text
  ```

- ```
  (foo "text") ==>  <bar>text1</bar>
                    ...
                    <quux>text2</quux>
  ```

We showed how we can do ordinary XSLT transformations and author sites from source XML documents or templates. We can indeed re-use legacy markup documents and stylesheets. Can we do something more advanced? Yes, we can.

The simplest case of XML authoring in SXML is converting an S-expression like this {(tag "text")} into the corresponding markup element. Lots of people do that. Simple? Yes. But there is a complication: some characters in "text" must be escaped during the translation. For example, greater- and less-than-signs and the ampersand. Surprisingly, this simple escaping is excruciatingly difficult in pure XSLT, for example. The set of characters to escape and the escaping rules may depend on the context – on the tag and its ancestors. For example, CDATA sections in XML and verbatim blocks in TeX have special escaping rules. In SXML, such context-sensitive translation is trivial – we merely need to instantiate a local text handler.

So-called higher-order tags increase the sophistication level of the transformation. We re-write one S-expression {(section ...)} into a more complex S-expression etc. until we reach primitive SXML elements – which are then transformed into XML. This successive re-writing is reminiscent of macro-expansion in Lisp and Scheme.

In the most advanced case, there doesn't appear to be any correspondence between a source SXML code and

the resulting XML document. {The paper in the proceedings gives one real-life example. The source SXML code holds a number of entities and their relationships in a normalized relational form. The output XML document represents the same entities in a unnormalized, hierarchical relationships. The corresponding SXML transformation is tantamount to denormalization: the conversion from a relational to a hierarchical database, so to speak. Because the source SXML document was normalized, it was notably more compact and easier to write.}

An example of a complex markup

```
(html:begin
   (Header
    (title "SXML") (Revision "2.5")
    (long-title "SXML")
    (Links (prev "xml.html") (next "web.html")))

  (body
   (navbar)
   (page-title)

   (TOC)

   (Section 2 "Introduction")
   (p ...)
   (Section 2 "Notation")
   (p ...)
...
```

The present slide gives another example of the non-obvious relationship between XML and SXML. This is an excerpt from the SXML specification, which is itself authored in SXML. For example, Header is just the collection of meta-data. Some of its child expressions, such as title, are expanded into the corresponding elements of the HTML HEAD. Some other elements, such as `Revision`, are not expanded at all. They are used in other parts of the document. I'd like to point out two elements: navbar and TOC. Obviously, they are not translated into navbar and TOC in angular brackets. Rather, (navbar) expands into a navigational bar, using the information from here {`(Links ...)`}. When generating the body of the page, the Section element here is converted into a header on the web page. The handler for the TOC element {here} re-traverses the whole document with a different stylesheet. That stylesheet expands a Section elements into a line of the table of contents, and everything else to nothing. In contrast to TeX, we don't need to re-run tex to get the references and citations right. XSLT can also traverse the source document with different stylesheets – with the help of modes. I submit that SXSLT accomplishes the same goal with far more simplicity and grace.

Retrieving Weather Advisories

`http://www.metnet.navy.mil/cgi-bin/oleg/get-advisories`

Aircraft Weather Advisory (SIGMET/AIRMET) Query

Retrieving advisories to aircraft of significant weather phenomena

parameter `lat-n` **must be of a type** `number`

Step 1. Specify advisories to retrieve

	Type	Class	Id
☒	**SIGMET**	CONVECTIVE ▽	
☐	**AIRMET**	ALL ▽	
☐	**OUTLOOK**	ALL ▽	

You have to specify at least one search parameter. All character strings are case-insensitive.

Step 2. Optionally select the area of interest

Latitude between 30 and 35x deg
Longitude between and deg

Lat/Lon are specified in degrees, e.g. `-30.75` ; negative numbers refer to Southern latitudes and Western longitudes correspondingly. *Longitudes may wrap around* .

Step 3. Optionally select the modification time

Reported since: minutes ago

Step 4. Check the Metcast server to query

Metcast server to query: http://zowie.metnet.navy.mil/cgi-bin/oleg/server

Let us consider the final example in more detail. It shows a rather unusual use of SXML and SXML transformations – not only for generating web forms but also for processing of the submission result.

This slide shows one of the front ends to the Metcast server. This web page lets a user retrieve aircraft weather advisories. For example, SIGMET advisories, which are advisories for all aircrafts of significant weather phenomena, such as thunderstorms, tornados, significant wind shear, clear air turbulence, etc. It is useful to check for SIGMETs before going on a trip. If you see a SIGMET along the way, you may want to take an extra book. {A SIGMET advisory indicates that the plane will be re-routed or diverted. The page connects to our developmental Metcast server – which may be used for illustration or development but may not be used for operation support. We have other servers for the latter purpose. Using them requires special arrangements.}

On this form you choose the type of an advisory to retrieve, optionally set the area of interest as a bounding lat/lon box, specify other parameters. The snapshot of the browser window on the slide shows that the user mistakenly typed a wrong character in the latitude field. When he submitted the form, he got back the page with the original form, all his input, plus the error message here, at the top.

Generating Web forms

```
(define Form
`(html:begin "Aircraft Weather Advisories"
  (body (@ (BGCOLOR "#88eebb") ...)
   (h1 "Aircraft Weather Advisory (SIGMET/AIRMET
   (p "Retrieving advisories to aircraft of sign
   (div (note-short))

    (search-form ""
     (step mandatory "Step 1. Specify advisories
     ...
     (step optional "Step 2. Optionally select t
     (table (@ (cellpadding "0") (cellspacing "0
      (tr
       (th "Latitude ")
       (td " between ")
       (td (ffield-input-text lat-s number 6))
       (td " and ")
       (td (ffield-input-text lat-n number 6))
       (td "deg"))
 ...
```

Here's an excerpt from the source code for the get-advisories page, edited for brevity. The full source code is freely available on the web.

The query web form is coded in SXML. We see obvious tags such as table, paragraph, etc. We also see not un-ordinary tags, for example, `step`. They are higher-order tags mentioned earlier. I'd like to point out this element: {(ffield-input-text lat-s number 6)}. It stands for the input field of a web form, for northern and southern latitudes. We see the name of the field and its size, {6}.

The script transforms this SXML into HTML, filling in form fields with default values. We get the web form we saw on the previous slide. When you submit the form, the script uses *this very same* SXML code – but a different stylesheet – to convert the QUERY_STRING into bindings for form variables, validating the user input in the process. Notice that {(ffield-input-text lat-s number 6)} had a type: number. The conversion stylesheet indeed tries to convert what you've entered into a number. The stylesheet throws an exception if it fails. CGI processing is therefore a tree transformation, too. The set of form variables is finally re-written into an SXML-RPC request (called an MBL request).

Aircraft Weather Advisory (SIGMET/AIRMET) Query

Retrieving advisories to aircraft of significant weather phenomena

The MBL expression for your request

```
(webq (bounding-box 35 -180.0 30 180.0)
      (products
        (AC-advisory
          ((SIGMET CONVECTIVE))
          (mime-type "text/plain"))))
```

Step 1. Specify advisories to retrieve

	Type	Class	Id
☒	**SIGMET**	CONVECTIVE ▽	
☐	**AIRMET**	ALL ▽	
☐	**OUTLOOK**	ALL ▽	

You have to specify at least one search parameter. All character strings are case-insensitive.

Step 2. Optionally select the area of interest

Latitude between 30 and 35 deg
Longitude between and deg

Lat/Lon are specified in degrees, e.g. `-30.75` ; negative numbers refer to Southern latitudes and Western longitudes correspondingly.
Longitudes may wrap around .

Step 3. Optionally select the modification time

The get-advisories page is an example is a three-tier Web service. A client connects to a get-advisories CGI script, the second tier. The client fills out a form, perhaps corrects noted errors, and finally submits it. The get-advisories script executes a "remote procedure call" to a Metcast server, receives an XML document – and sends it to a client or do additional processing. It should be emphasized that SXML is being used on all stages: to specify the form, to represent user's input, and to express the "remote procedure call". Moreover, the same small library of SXML transformers can be used over and over again – for very different purposes.

The get-advisories Web form has a special button to show the generated MBL request for a particular user input. If we press that button, we will see the result shown on this slide. At the server side, the MBL request is further re-written into a SQL query. The result of the query is re-written into SXML, which is converted into XML and sent to the client. We could have sent the SIGMET advisory in its SXML form; however some clients insist on a syntax-heavy XML.

Retrieved advisory

```
<!DOCTYPE Advisories SYSTEM 'OMF.dtd'>
<Advisories TStamp='1035315601'>

<SIGMET class='CONVECTIVE' id='43C' TStamp='1035
<VALID TRange='1035309300, 1035316500'>VALID UNT
<AFFECTING>LA TX AND CSTL WTRS
</AFFECTING>
<EXTENT Shape='AREA' LatLons='30.395 -93.822
  28.307 -93.110 27.220 -95.582 30.431 -95.894 3
</EXTENT>
<HAZARD Type='TS' Dir='260 13' Tops-max='42000'>
AREA TS MOV FROM 26025KT. TOPS TO FL420.</HAZARD
<REMARKS>SEE INTL SIGMET SERIES BRAVO FOR ADJ TS
</SIGMET>
</Advisories>
```

If we execute the MBL by pressing a button on the web form, we will get the following XML in the reply. There was one convective SIGMET within the area of interest, 30 through 35 degrees of northern latitude. The SIGMET warned about area of thunderstorms, up to the flight level 420. The thunderstorm moves from the west at 13 m/sec (25 knots). This set of numbers {`LatLons=...`} describes the enclosing polygon for the affected area. We have several graphical clients that use that information to draw the area on the map.

Conclusions

SXML and SXML tools:

- play in a team
- offer backwards compatibility: investment protection
- support legacy data migration
- rely on a mature programming language
- erase the linguistic barrier: content integration, maintainability, better productivity

`http://ssax.sourceforge.net/`

In conclusion, we tell our managers that our tools accept and generate valid XML and HTML, regardless of what the tools use internally. We play in a team, alongside other XML authors and developers. Our SXSLT and SXPath tools are backwards compatible with XSLT and XPath and thus protect the investment in those tools and the skills. Our tools are based on a mature programming language, with excellent educational resources. We also provide a relatively smooth migration path from legacy data and programs to the ones based on SXML. Moving to the S-expression-based formats and tools *is* worthwhile. We won't have a linguistic barrier any more. As the result, markup processing code becomes easier to maintain and easier to write.

Due to a unique combination of the expressive power of the Scheme language and compatibility with W3C Recommendations, such an approach provides the most sophisticated XML processing techniques along with a good protection of investments in XML/XSLT solutions.

The software mentioned in this talk is in public domain. It is a SourceForge project. You are very welcome to use it. If you have any question, please post it on the project mailing list, or send it to me.

Final Conclusion

Embrace and Extend XML

- S-expressions alongside and instead of XML

- Promote KQML instead of SOAP

- XSLT $\rightarrow$
  Lisp-extended XSLT $\rightarrow$
  Lisp instead of XSLT

- The language to manipulate trees of S-expressions already exists

Finally, let us embrace and extend XML. We all know a constructive proof that this is a sound policy. Let us follow XML rules – and emphasize how verbose XML is compared to other representations of S-expressions. Let us remark on the immaturity and bloatedness of SOAP, and on clarity and firm foundations of KQML. Let us use XSLT, and remind how limited and imperfect language it is. Let us build tools that take the existing stylesheets and XML queries – and offer powerful and convenient S-expression-based extensions. Let us engage in W3C XPath and XQuery discussions. Let us also point out that the language to manipulate trees of S-expressions already exists. There is no need to re-invent the wheel, especially a square one.