

Little Languages for XML

Axel T. Schreiner

Department of Computer Science

Rochester Institute of Technology

<http://www.cs.rit.edu/~ats/talks/xml/> [[pdf](#)] [[code](#)] [[code.zip](#)]

XML is hailed to replace ASCII as a human-readable way to represent structured data; however, the design placed only a low priority on the need to generate XML documents by hand. Still, XSLT and other programming languages use XML syntax and tend to be cumbersome for humans to read and write.

This talk describes prototypes for an imperative and a declarative language to manipulate XML documents and tries to show that less can be more. Neither of these little languages uses XML syntax and that seems to be quite an improvement...

Contents

Terminology — XML, SAX, DOM, XSLT
A little declarative language — SL
A little imperative language — XOML
Summary

Links

[W3C reports](#)
[German lecture notes](#) 2001
 [SL code](#)
 [XOML code](#)
[oops homepage](#) 2000

XML — Extensible Markup Language

XML [W3C 1998] is a simplified version of SGML [ISO 1986] and inherits its confused syntax.

"The design goals for XML are:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance."

[From the [Recommendation](#).]

Example

The following XML-based document could be the result of a database query:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE people [
  <!ELEMENT person (person*)>
  <!ELEMENT person (first,last)>
  <!ATTLIST person
    pid ID #REQUIRED
    boss IDREF #IMPLIED
    sex (male|female) #REQUIRED
  >
  <!ELEMENT first (#PCDATA)>
  <!ELEMENT last (#PCDATA)>
]>
<people>
  <person pid="i4711" boss="i4712" sex="female" >
    <first>Jane</first>
    <last>Doe</last>
  </person>
  <person pid="i4712" sex="male">
    <first>John</first>
    <last>Doe</last>
  </person>
</people>
```

SAX — Simple API for XML

SAX [Megginson 1998] essentially is an observer interface for an XML parser, i.e., as the parser recognizes XML elements etc. it calls certain methods in certain handlers.

- yacc calls actions at the phrase level.
- SAX calls the handler at the lexical level.

XML parsers supporting SAX are readily available; it is thus trivial to write a Java program to sequentially process an XML-based document.

Examples — typical database queries

- count people, or females and males.
- find boss of Jane Doe.
- match couples by last name.
- who works for John Doe.

DOM — Document Object Model

DOM [W3C 1998] is an IDL-based system of interfaces for objects to represent and manipulate XML-based documents.

Given a Java implementation of DOM and a SAX parser it is trivial to write a Java program to represent an XML-based document as a tree of generic objects.

- XML parsers supporting DOM are readily available.
- DOM permits tree traversal and queries of an XML-based document.

Example — who works for whom

```
package doc;
import java.io.File;
import java.net.URL;
import javax.xml.parsers.DocumentBuilderFactory;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class Boss {
    /** finds all people working for each boss.
     */
    public static void main (String args []) throws Exception {
        URL cwd = new File(System.getProperty("user.dir")).toURL();
        Document doc = DocumentBuilderFactory.newInstance()
            .newDocumentBuilder()
            .parse(System.in, cwd.toString());
        NodeList person = doc.getDocumentElement()
            .getElementsByTagName("person"); // recursive...

        // find people without 'boss'...
        for (int c = 0; c < person.getLength(); ++ c) {
            Element boss = (Element)person.item(c);
            if (boss.getAttributeNode("boss") == null) {
                System.out.print(first(boss)+":");
                String bossPid = boss.getAttribute("pid");

                // find people matching 'boss' to bossPid...
                for (int s = 0; s < person.getLength(); ++ s) {
                    Element sub = (Element)person.item(s);
                    if (bossPid.equals(sub.getAttribute("boss")))
                        System.out.print(" "+first(sub));
                }
                System.out.println();
            }
        }
    }

    /** encapsulates access to text of 'first'.
     */
    public static String first (Element person) {
        person = (Element)person.cloneNode(true); // avoid side effect
        person.normalize();
        return person.getElementsByTagName("first")
            .item(0).getFirstChild().getNodeValue();
    }
}
```

XSLT — Extensible Stylesheet Language / Transformations

XSL [W3C 2001] uses XSLT and Formatting Objects to describe high-quality rendering of XML-based documents.

XSLT [W3C 1999] is an XML-based functional language to describe tree transformations.

- XSLT looks like macro processor code: output material interspersed with instructions.
- XSLT cannot be described with a DTD.

Example — table of people with boss' first name

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <!-- convert tree to HTML table -->
  <xsl:template match="/">
    <html><body><table>
      <xsl:for-each select="/people/person">
        <tr>
          <td><xsl:value-of select="substring(@pid, 2)"/></td>
          <td>
            <xsl:if test="@sex='female'">Mrs.</xsl:if>
            <xsl:if test="@sex='male'">Mr.</xsl:if>
          </td>
          <td><xsl:value-of select="first"/></td>
          <td><xsl:value-of select="last"/></td>
          <td><xsl:value-of select="substring(@boss, 2)"/></td>
          <td>
            <xsl:call-template name="first">
              <xsl:with-param name="boss" select="@boss"/>
            </xsl:call-template>
          </td>
        </tr>
      </xsl:for-each>
    </table></body></html>
  </xsl:template>

  <!-- return first name of subordinate -->
  <xsl:template name="first">
    <xsl:param name="boss"/>
    <xsl:value-of select="/people/person[@pid=$boss]/first"/>
  </xsl:template>
</xsl:stylesheet>
```

A little declarative language — SL

🔴 Don't create a language with a new syntax and new features [Hoare 1974].

XSLT combines pattern matching and macro processing wrapped into XML syntax.

XSLT is essentially a successor to DSSSL for SGML which had a LISP-like syntax.

Example — table of people with boss' first name

SL is XSLT without the X. It uses a prefix notation with braces for lists with more than one element. Dependencies can be shown by indentation but whitespace is irrelevant.

```
// convert tree to HTML table
main
  <html>
    <body>
      <table>
        for "/people/person"
          <tr> {
            <td>
              value-of "substring(@pid, 2)"
            <td>
              if "@sex='female'" "Mrs."
              elif "@sex='male'" "Mr."
              fi
            <td>
              value-of "first"
            <td>
              value-of "last"
            <td>
              value-of "substring(@boss, 2)"
            <td>
              first(boss = "@boss")
          }
      </table>
    </body>
  </html>

// return first name of subordinate
proc first ( boss )
  value-of "/people/person[@pid=$boss]/first"
```

Implementation

SL is simply a frontend for the algorithms underlying XSLT.

- 🟢 *oops* creates and serializes a Java parser from an EBNF grammar for SL.
- 🟢 The parser creates a DOM tree for an SL program using a DOM implementation like Xerces.
- 🟢 The DOM tree is run as a transformer using JAXP and an XSLT implementation like Saxon.

Control structure example — Euclid's algorithm

```

pairs = {                                     // a global 'variable' containing nodes
  <pair> { <x> 36 <y> 54 }
  <pair> { <x> 54 <y> 36 }
}

main {
  message { "pairs: " copy-of "$pairs" }

  for "$pairs/pair" {
    result = gcd(x = "x", y = "y")
    <gcd value="{ $result }"/>
  }
}

proc gcd ( x, y )
  if "$x = $y"      value-of "$x"
  elif "$x < $y"   gcd(x = "$x", y = "$y - $x")
  else              gcd(x = "$x - $y", y = "$y")
  fi

```

Pattern matching example — evaluating arithmetic expressions

```

proc "add" () {                               // pattern for <add>: binary operator node
  left = apply-to "[1]"
  right = apply-to "[2]"
  typed(arg = "$left + $right")
}

proc "minus" () {                             // pattern for <minus>: unary operator node
  right = apply-to ""
  typed(arg = "- $right")
}

proc "literal" ()                             // pattern for <literal value="n">
  value-of "@value"
}

proc typed ( arg ) {                          // typed evaluation, looks for <?eval.sl int?>
  pi = value-of "processing-instruction('eval.sl')"
  if "$pi = 'int'"
    if "$arg >= 0"  value-of "floor($arg)"
    else           value-of "ceiling($arg)"
    fi
  else
    value-of "$arg"
  fi
}

```

A little imperative language — XOML

awk combines a processing model (records/lines with fields), patterns, hashtables, and some conventional programming language concepts into a powerful report generator for flat data.

awk has a very flat learning curve.

XOML attempts a similar thing for XML-based data.

Example — table of people with boss' first name

```
print name(list = input("people.xml")) ":\n"; // get document and print root name
for (boss in list)                               // show boss and subordinates
  if (! (attributes(boss) contains boss)) {     // a boss has no reference to boss
    print text(boss:first) ":";
    for (person in list)
      if (person.boss == boss.pid)             // a subordinate references the boss
        print " " text(person:first);
    print "\n";
  }
}
```

- Syntax and control structures are similar to C and *awk*.
- Usual arithmetic expressions , string concatenation, and embedded assignment.
- Dynamically typed variables can contain numbers, strings, hashtables, and nodes.
- Period references node attributes (like structure components).
- Colon references contained nodes (like Mac path components).

Implementation

A XOML program is an object tree which operates on a stack of Java objects, among them numbers, strings, hashtables, and nodes.

- *oops* creates a Java parser from an EBNF grammar for XOML.
- The parser creates an executable object tree for a XOML program.
- *input* uses a SAX parser to map an XML-based document to nodes consisting of an `ArrayList` referencing nested nodes and `String` texts and a `HashMap` for attributes.

Example — hashtables

```
for (person in input("people.xml")) // count sexes in hashtable
  count[person.sex] = count[person.sex] + 1;
for (sex in count) // loop over hashtable
  print count[sex] " " sex "\n";
```

Example — preorder traversal of an unknown tree

```
function traverse ( indent, tree ) {
  print indent name(tree); // show name
  for (attr in attributes(tree)) // show attributes if any
    print " " attr "=" tree.(attr) "";
  t = trim(tree); // show text if any
  if (t != "") print ": " t;
  if (name(tree) == "person") // signal bosses
    if (tree.boss == "") print ": boss";
  print "\n";

  for (node in tree) // show content
    traverse(indent " ", node);
}
traverse("", input("people.xml"));
```

Example — modifications

```
list = input("people.xml");
do { // assume each element is a person
  for (a in attributes(list:person)) // remember each attribute name...
    attr[a] = 1; // ...in an associative array
  for (a in attr) // delete each attribute
    delete list:person.(a);
  print list:person "\n"; delete list:person; // print and delete each person
} while (elements(list) > 0);
```

Example — creating XHTML

```
xhtml = new("html"); // new root node
xhtml.id = "value"; // new attribute
if (xhtml contains id) print "subnode?\n"; // no subnode by tag name
if (xhtml contains 0) print "index 0?\n"; // no subnode by position
if (attributes(xhtml) contains id)
  xhtml.id = "mmr"; // replace attribute

xhtml:0 = new("body"); // new subnodes into positions
p = xhtml:body:0 = new("p");
p:0 = "Happy "; // add text
p:1 = new("b"); // add subnode
p:b:0 = "Birthday"; // add text
p:2 = "!"; // add text
p:2 = ", " xhtml.id p:2; // replace with attribute text

print xhtml "\n";
// <html id="mmr"><body><p>Happy <b>Birthday</b>, mmr!</p></body></html>
```

❌ No syntax for insertion (yet).

Summary — SL

- SL is just syntactic sugar; it does not improve on XSLT's inefficiencies or learning curve.
- SL tends to unearth the algorithms behind XML's white noise imposed upon XSLT.
- SL could be improved if XPath expressions were integrated into the syntax and not just included as strings.

Summary — XOML

- XOML is a conventional programming paradigm, hopefully with a flat learning curve.
- Extending XOML is the subject of a master's thesis at Osnabrück.
- Adding regular expressions was trivial.
- Adding XPath expressions (tree pattern matching) is more difficult; it requires DOM to represent input.