XML: Syntax, Sin, and Synergy

Philip Wadler
Bell Labs, Lucent Technologies
www.cs.bell-labs.com/ wadler

September 7, 1999

Abstract

XML is intended as a basic syntax for trees, onto which various domain-specific languages might be mapped. The expressive power of XML language is similar to that of Lisp S-expressions, though its syntax is more baroque. An abcedary of applications is under development, starting with air lines, banks, cell phones, and continuing through the rest of the alphabet. A few of the intended applications embed programming languages within XML, including stylesheets (XSL), schemas (XML Schema), and general purpose programming languages (XFA, PIA). These embeddings are reminiscent of the embedding of Lisp with S-expressions, but suffer from problems that Lisp avoided. By adopting two complementary syntaxes, these problems can be overcome.

Syntax

XML is a syntax for specifying trees, descended from SGML and HTML. Here is an example of a tree rendered in XML.

```
<html>
  <head>
    <title>Philip Wadler's home page</title>
    <meta name="keywords" content="Java, GJ, Haskell, SML, monads"/>
  </head>
  <body>
    <h1>Philip Wadler's home page</h1>
    <img align="top" src="phil.gif" alt="A photo of Phil"/>
    <p>My publications include:<ul>
      <li><em>GJ: Making the future safe for the past</em>, G. Bracha,
      M. Odersky, D. Stoutamire and P. Wadler, <b>OOPSLA 98</b>.</li>
      <li><em>Featherweight Java: A core calculus for Java and GJ</em>,
      A. Igarishi, B. Pierce and P. Wadler, <b>OOPSLA 99</b>.</li>
```

```
    </ul></p>
  </body>
</html>
```

An XML tree consists of *element* nodes, which consist of a *tag* (such as `html`) and zero or more *attributes* and *children*. The children nodes are element nodes or *text*.

Here is a second example.

```
<papers>
  <paper>
    <author>G. Bracha</author>
    <author>M. Odersky</author>
    <author>P. Wadler</author>
    <title>GJ: Making the future safe for the past</title>
    <conference>OOPSLA 98</conference>
  </paper>
  <paper>
    <author>A. Igarishi</author>
    <author>B. Pierce</author>
    <author>P. Wadler</author>
    <title>Featherweight Java: A core calculus for Java and GJ</title>
    <conference>OOPSLA 99</conference>
  </paper>
</papers>
```

It might be desirable to store a bibliography in the second form, and to generate a part of the first form from the second; this is typical of one usage of XML.

XML is merely a flexible notation for representing trees. Other notations would work just as well, or better. For instance, here is the first of the examples above, re-rendered as a Lisp S-expression.

```
(html
  (head
    (title "Philip Wadler's home page")
    (meta (@name "keywords") (@content "Java, GJ, Haskell, SML, monads")))
  (body
    (h1 "Philip Wadler's home page")
    (img (@align "top") (@src "phil.gif") (@alt "A photo of Phil"))
    (p "My publications include:"
      (ul
        (li (em "GJ: Making the future safe for the past")
            "G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler,"
            (b "OOPSLA 98"))
        (li (em "Featherweight Java: A minimal core calculus for Java and GJ")
```

```
"A. Igarishi, B. Pierce, and P. Wadler,"
(b "OOPSLA 99")))))
```

We have indicated which nodes correspond to attributes by prefacing their name with the symbol `@`. XML places some restrictions that S-expressions do not: an attribute node may only contain a text node; any two adjacent text nodes may be merged; a text node consisting solely of white space may (sometimes) be ignored.

Sub-languages based on XML have been proposed for a variety of uses, for commerce (airlines, banks, cell phones, ...), academia (astronomy, biology, chemistry, ...), and government (DOD, IRS, ...).

Sin

One use of XML is to describe programs that process XML, reminiscent of the use of S-expressions to encode Lisp. Examples of such applications include stylesheets (XSLT), schemas (XML Schemas), and general purpose programming languages (XFA, PIA). However, because XML is more verbose than Lisp, these encodings turn out to be problematic for XML in a way that they are not for Lisp.

For instance, here is a program in XFA (XML For All, `www.xfa.com`) to compute a web page of factorials.

```
<xfa:function exp="Factn">
  <xfa:let name="n" exp="8">
  <xfa:function exp="Factorial(i)">
     <xfa:if exp="IntEq(i,0)"><xfa:val exp="1"/></xfa:if>
     <xfa:else><xfa:val exp="Mult(i,Factorial(Sub(i,1)))"/></xfa:else>
  </xfa:function>
  <h1 color="red">Factorials up to <xfa:val exp="n"></h1>
  <ul>
    <xfa:for name="i" exp="n">
      <li> factorial(<xfa:val exp="i"/>)=<xfa:val exp="Factorial(i)"/>
    </xfa:for>
  </ul>
</xfa:function>
```

The first problem is that while XML may be a suitable notation for web-page markup, it is less readable when applied to programming languages. The second problem is that while the tree structure of commands is properly revealed as XML, the tree structure of expressions is concealed within a string, undoing much of the advantage of a tree-based notation. The two problems are related: if expressions were represented as XML rather than as strings then the program would be even less readable.

Most programming languages embedded in XML suffer from the same two problems. For instance, here is an XSLT program to translate XML bibliography data into HTML (that is, to translate the data in the second display of this paper into a part of the second display).

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
                xmlns:html="http://www.w3.org/TR/xhtml1">
                xmlns:bib="http://www.cs.bell-labs.com/~wadler/bib/">
  <xsl:template match="bib:papers">
    <html:ul>
      <xsl:apply-templates/>
    </html:ul>
  </xsl:template>
  <xsl:template match="bib:paper">
     <html:li>
       <xsl:apply-templates select="bib:title"/>
       <xsl:text>, </xsl:text>
       <xsl:apply-templates select="bib:author"/>
       <xsl:text>, </xsl:text>
       <xsl:apply-templates select="bib:conference"/>
       <xsl:text>.</xsl:text>
    </html:li>
  </xsl:template>
  <xsl:template match="bib:title">
     <html:em><xsl:apply-templates/></html:em>
  </xsl:template>
  <xsl:template match="bib:author[position()=1]">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="bib:author[position()=last()]">
    <xsl:text> and <xsl:text><xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="bib:author">  <!-- otherwise -->
    <xsl:text>, <xsl:text><xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="bib:title">
     <html:b><xsl:apply-templates/></html:b>
  </xsl:template>
</xsl:stylesheet>
```

Again, commands are represented as XML trees, while expression (in the `match` and `select` attributes) are represented as strings. (Indeed, there is a separate specification, XPath, for the expression language.)

Since it violates the uniformity of trees, coding expressions as strings can and does lead to problems. For instance, XML supports modularity with a

convention for namespaces, which associates a prefix with a URL. In the XSL program above, program elements are in the `xsl` namespace, output elements are in the `html` namespace, and bibliography elements are in the `bib` namespace. Prefixes can be renamed. For instance, the above program is entirely equivalent to one which begins

```
<xsl:stylesheet xmlns:x="http://www.w3.org/XSL/Transform/1.0"
                xmlns:h="http://www.w3.org/TR/xhtml1">
                xmlns:b="http://www.cs.bell-labs.com/~wadler/bib/">
```

and replaces each occurrence in a tag of `xsl` by `x`, of `html` by `h`, and of `bib` by `b`. It was originally intended that such renaming could be performed automatically by an XML processor. However, because tags strings encode expressions, this is problematic: a processor cannot determine which occurrences of, say, `xsl` in a string correspond to prefixes that should be renamed, and which correspond to literal strings that should not be renamed.

Synergy

The two problems described above arise as an artifact of the XML syntax, which hinders readability and encourages one to encode expressions as strings. An obvious solution is to move to a different syntax.

Indeed, XFA does support an alternative syntax, for exactly this reason. Here is the factorial problem from above, re-expressed in the alternative syntax.

```
<xfa:function Factn>
  <xfa:let n=8>
  <xfa:function Factorial(i)>
     <xfa:if IntEq(i,0)><xfa:val 1/></xfa:if>
     <xfa:else><xfa:val Mult(i,Factorial(Sub(i,1)))/></xfa:else>
  </xfa:function>
  <h1 color="red">Factorials up to <xfa:val n></h1>
  <ul>
    <xfa:for i=n>
      <li> factorial(<xfa:val i/>)=<xfa:val Factorial(i)/>
    </xfa:for>
  </ul>
</xfa:function>
```

In order to keep the relationship to XML clear, XFA supports only a minimal change to the syntax. One may write

$$n\texttt{=}e$$

in place of the slightly more longwinded

$$\texttt{name="}n\texttt{" exp="}e\texttt{"}$$

Arguably, the result is as unreadable as XML while not actually being XML: the worst of both worlds.

On the other hand, the goal of having a close correspondence between XML and the alternative syntax is a good one. This note proposes it can be achieved by the following means: use two *isomorphic* languages, one of which is XML and the other of which is suited to programming, and allow each to *escape* to the other. This permits *synergy*: the sum of the two syntaxes may be greater than its parts.

To illustrate this thesis, this note presents a simple programming language based on this notational trick. All programming languages for XML satisfy the invariant that the name begins with the letter X, so this language will be given the minimal name satisfying the invariant.

Here is the factorial example rendered in X.

```
factorial(k) = if(k == 0, 1, k*factorial(k-1)),
n = 8,
{ <h1 color="red">Factorials up to {n}</h1>
  <ul>
   { for((i = 0), (i <= n), (i = i+1),
      { <li> factorial({i}) = {factorial(i)} </li> } ) }
  </ul> }
```

The program is written in two isomorphic notations, XML proper and an alternative notation, which we'll dub *X-expressions*. Curly braces are used to switch from X-expressions to XML, and to switch back again.

Curly braces can be used wherever convenient, or omitted entirely. Thus there are many possible renderings of the above program.

Here is the rendering entirely in XML notation.

```
<op name="=">
  <factorial><var name="k"/></factorial>
  <if>
    <op name="=="><var name="k"/><int val="0"/></op>
    <int val="1"/>
    <op name="*">
      <var name="k"/>
      <factorial><op name="-"><var name="k"/><int val="1"/></op></factorial>
    </op>
  </if>
</op>
<op name="="><var name="n"/><int val="8"></op>
<h1 color="red">Factorials up to <var name="n"></h1>
<ul>
  <for>
    <op name="="><var name="i"/><int val="0"/></op>
```

```
      <op name="&lt;="><var name="i"/><var name="n"/></op>
      <op name="=">
        <var name="i">
        <op name="+"><var name="i"/><int val="1"/></op>
      </op>
      <li>
        factorial(<var name="i"/>) =
        <factorial><var name="i"/></factorial>
      </li>
    </for>
  </ul>
```

This rendering is, as expected, is not easily read. The main difficulty is length, but there are other small matters. XML does not allow '`<`' as a character (even between quotes), lest it be confused with the beginning of a tag, and so one must write '`<=`' as '`<=`', using XML's escape convention.

And here is the rendering entirely as an X-expression.

```
factorial(k) = if(k == 0, 1, k*factorial(k-1))
n = 8
h1(@color({red}),{Factorials up to},n)
ul(
  for(i = 0, i <= n, i = i+1,
    li({factorial({i}) = {factorial(i)}})
  )
)
```

Here curly braces are still used to escape to plain text, and to escape back to X-expressions. Arguably, XML is just as readable when rendered as X, though not conversely.

The isomorphism between X-expressions and XML is summarised in the following table.

{t(${e_1}$,...,${e_n}$)}	≡	<t> {${e_1}$} ... {${e_n}$} </t>
{t()}	≡	<t/>
{t(@${a_1}$=${s_1}$,...,@${a_m}$=${s_m}$,${e_1}$,...,${e_n}$)}	≡	<t ${a_1}$="${s_1}$" ...${a_m}$="${s_m}$"> {${e_1}$} ...{${e_n}$}
{l = r}	≡	<op name="="> {l} {r} </op>
{${e_1}$ + ⋯ + ${e_n}$}	≡	<op name="+"> {${e_1}$} ... {${e_n}$} </op>
{n}	≡	<var name="n"/>, where n is a name
{i}	≡	<int val="i"/>, where i is an integer
{"s"}	≡	<str val="s"/>, where s is a string
{{x}}	≡	x

The general rule is to convert an X-term that looks like a function call into an XML tree with the function name as the tag. Special rules handle infix operators, variables, and integers. Thus, the X-expression

```
if(k == 0, 1, k*factorial(k-1))
```

converts to the XML tree

```
<if>
  <op name="==><var name="k"/><int val="0"/></op>
  <int val="1"/>
  <op name="*">
    <var name="k"/>
    <factorial><op name="-"><var name="k"/><int val="1"/></op></factorial>
  </op>
</if>
```

Curly braces toggle between X-expressions and XML trees, so two adjacent curly braces cancel out.

The notation is a *positional* one. As in Lisp, the order of the components is significant. A contrasting design might label the components. For instance, instead of writing

```
if(k == 0, 1, k*factorial(k-1))
```

one might have chosen to use extra tags to label the components

```
if(test(k == 0), then(1), else(k*factorial(k-1)))
```

which is more like "call-by-keyword". This was avoided mainly for brevity, and to avoid the interlacing of tags that indicate computation (`if`) with tags that act as keywords (`test`, `then`, `else`).

The positional notation contrasts with the requirement in XML that all text nodes are run together. For convenience, a second form of text is provided that generates an element, analogous to the special forms for variables and integers. Thus, for instance the X-expression `"foo"` is equivalent to the XML tree `<string val="foo">`.

Iterated infix operators give rise to a single XML tag. Thus, for instance the X-expression

```
1+2+3
```

is converted to the XML tree

```
<op name="+"><int val="1"><int val="2"><int val="3"></op>
```

No precedence of operators is assumed, so parentheses must be used to disambiguate. Further, parentheses are only allowed where required for disambiguation, so that the mapping from X-expressions to XML loses no information. Thus,

```
1-2-3
```

translates to

```
<op name="-"><int val="1"><int val="2"><int val="3"></op>
```

while

```
1-(2-3)
```

translates to

```
<op name="-">
  <int val="1">
  <op name="-"><int val="2"><int val="3"></op>
</op>
```

Furthermore,

```
(1*2*3)+4+5
```

translates to

```
<op name="+">
  <op name="*"><int val="1"><int val="2"><int val="3"></op>
  <int val="4">
  <int val="5">
</op>
```

However, the X-expression `1+2*3` is illegal because it needs parentheses to disambiguate, and the X-expression `(1+2)` is illegal because it has parentheses which are not needed to disambiguate.