XML challenges to programming language design

Per Bothner
<per@bothner.com>

Abstract

People are using XML-based languages for a number of applications. The paper discusses what we can learn from this, and various programming language ideas and features inspired by XML applications and tools. Some but not all of the ideas have prototype implementations within Kawa.

1 Introduction

A number of popular "programming" languages use XML's syntax. Some languages are used for generating XML or HTML. Since these need to be able to construct XML/HTML fragments, it is reasonable to use XML syntax for constructors and perhaps the whole language. This category includes Sun's JSP and W3C's XSLT. Other languages are used to lay out documents and GUI interfaces. These applications are characterized by nested graphical regions that may have different properties, such as size, font, and color. Using XML attributes to specify optional properties works quite well, and matches what people are used to from HTML. Examples of such languages include Mozilla's XUL [XUL], Microsoft's XAML [XAML], and W3C's XSL-FO. These and other languages succeed in spite of XML's clumsy syntax, even though Lisp-like languages have long provided a more flexible and compact notation for expressing nested data structures. While the XML world could learn from the Lisp world, there are also some interesting things for Lisp people to learn from XML. Of course Lisp people have long provided libraries and ideas for XML processing.

In this article I will make my own suggestions. Some of these ideas and features have been implemented with the Kawa [Kawa] framework, which is both a Scheme implementation written in Java, and a compiler framework for compiling high-level languages to Java bytecodes. See particularly my KRL dialect, which is based on Bruce Lewis's BRL, and my new Q2 proto-language.

2 Data constructor syntax

We need a good syntax for constructing nested objects. In many simple programs most of the program consists of constructors, so it is good to have an efficient syntax, and avoid noise keywords such as `new` or `make`. We need a convenient syntax for attributes or keyword parameters.

We also need syntax for function calls, as well as syntax for control structures such as loops and conditionals. JSP uses XML syntax for all of these, and there are popular "tag libraries" which are essentially function libraries whose functions are called using XML syntax. The XQuery [XQuery] language, while powerful and elegant in many ways, has a more traditional mixed-style syntax, with XML constructors in XML syntax, but function calls and expressions closer to C/Java syntax. There are some advantages to using different syntactic styles for different operations, but it does make it harder when you need to change or refactor your code.

Lisp languages of course has a unified syntax for function application and data structures, but we still need a way to distinguish the two. The traditional mechanism is to use quotation or quasi-quotation for data, and some XML-in-Lisp embeddings use quotation conventions. However, quotation is a read-time operation, and returns values that are fixed early. Too early if you want to associate richer type information

with the data structures, perhaps because you want to do schema validation, or use different object types for diffent element tags.

My suggestion is to distinguish function calling from object construction not by syntax, but to distinguish them using name lookup. This requires a declaration of the element type, though I show below how use of namespaces makes this convenient.

```
(define (process-para p) ...)
;; Define para as a constructor for a
;; type with mixed child content.
(define-element para mixed)

(process-para
  (para (string-upcase "data")))
```

This yields, in XML syntax:

```
<para>DATA</para>
```

3 Typed node values

Some XML/HTML-producing languages, such as JSP and PHP, generate textual XML output. I.e. web pages are produced by explicitly or implicitly writing strings to a special file. This is OK when the goal is just generating web pages, but awkward if we want to be able process existing XML data, Furthermore, the result of a constructor should have a real type, distinct from lists or vectors. In other words, an element constructor returns an object.

The "standard" representation of XML elements and other node types is W3C's DOM interface, but that may not always be the best representation. We may want to customize the class used to represent a specific element type (as in JAXB). In that case XML is just a special input/output representation, and the structure of the XML file might not directly match the internal representation.

Both DOM and the more abstract XQuery/XSLT data model allow you to directly access a parent element from a child node. This requires a lot of either re-parenting or (conceptual) copying, which is awkward. Lisp's model where you can get at the children from a parent but not vice versa seems preferable. Instead, we can reference a node indirectly using the *path* we used to access it, rather like a Unix filesystem, which distinuishes file names from inodes. A path is a pointer, but it also includes a trail of parent pointers, so we can easily get to parent and sibling nodes.

4 Namespaces

XML, like Common Lisp's package system, uses two-level names: A *qualified name* consists of a local (simple) name, and a globally-unique URI (URL). The URI corresponds to a Common Lisp package, but a URI is a string, rather than an object. Qualified names (QNames) match if both the local name and the URI are the same, which in practice is not very different from interned Common Lisp symbols, which are the same if the local name and the package are the same.

In a Lisp program we write symbols using package prefixes, while in an XML we write qualified names using a namespace prefix, a colon, and a local name. A difference is that in Lisp the set of package names (and nicknames) is maintained in a global table, while in XML there is a local mapping from namespace prefix to URI. This mapping is part of the lexical scope rather than the global scope. I think that is a better solution, as it it lets different modules use different short prefixes. Note that in Common Lisp resolving a name to a symbol is a read-type operation. I suggest it should be done at the same time as macros and special forms are recognized. This allows `let`-forms to define local namespace aliases (nicknames).

The Kawa keyword `define-namespace` defines a name as a prefix that aliases a namespace URI. At the same time it implicitly declares all the functions that use the same prefix to be constructors that when called create node objects. It is reasonable to also provide a `let-namespace` keyword, though Kawa does not currently do so.

```
(define-namespace xhtml
  "http://www.w3.org/1999/xhtml")

(xhtml:p "some text "
  (let-namespace
```

```
    (math "http://.../MathML")
    (math:xx))
```

If you print this using XML syntax you might get:

```
<xhtml:p
  xmlns:xhtml="http://www.w3.org/1999/xhtml">
  some text
  <math:xx
    xmlns:math="http://.../MathML"/>
</xhtml:p>
```

An imported module can be bound to a namespace prefix:

```
(import mat <matrix-functions>)
(mat:transpose (mat:zero 2 3))
```

5 Sequences

The XQuery 1.0 and XSLT 2.0 languages use *sequences* in an interesing way. Such a sequence differs from the Common Lisp concept of the same name in that you cannot directly nest sequences. Equivalently, there is no difference between a value and a singleton sequence consisting of just that value. In that respect a sequence is similar to Lisp/Scheme's "multiple values". However, a sequence is a first-class value in that a variable or parameter can be bound to a sequence. (Kawa represents an XQuery sequence and Scheme multiple values the same way.)

One might think that non-nestable sequences are too limiting, and of course a language needs a way to provide nested data structures. We'll discuss arrays and nodes later. Non-nestable sequences do provide a nice functional model for composing program fragments, as we'll see.

6 Statements are expressions

Scheme and Common Lisp are expression-oriented "mostly-functional" languages, but the looping and `prog` constructs don't fit very smoothly into this.

First consider the Scheme `<body>` syntax, which is one or more declarations or expressions, and whose result is that of the final expression. We can modify the definition so that the result is the sequence resulting from concatenating *all* the sequences resulting from the sub-expressions. We also define declarations, assignments, and similar statements to return zero values. Most existing code should work as is. When needed, a `discard` function can be used to ignore all its argument values, returning zero values. The concatenation operator becomes the same as the statement separator operator; in a non-Lisp-syntax language both might use semi-colon or line separators.

If evaluating a loop results in the concatenation of the values from each iteration, then the value of a loop is the same as unfolding the loop to yield a statement sequence, which is very intuitive behavior. Using sequence concatenation in this way yields pleasant and natural semantics for expression languages. As the XQuery language shows, it also it is very convenient for processing XML and similar data structures.

Here is an example:

```
(define x
  (let r ((i 0))
    (+ 100 i)
    (if (< i 5)
      (r (+ i 1)))
    i))
```

Each time `r`'s body it evaluated it yields two values: initially the result of `(+ 100 i)` and finally the parameter `i`. In between is a recursive call. All the values are appended, yielding a sequence of 10 values:

```
100 101 102 103 104 105 5 4 3 2 1 0
```

The FLWOR-expression of XQuery is a powerful and elegant way to map over sequences. Loosely, a FLWOR-expressions iterates over a sequence, and for each item in the sequence it binds a local variable, and evaluates an expression within that scope. The result is the concatenation of all the results. Common Lisp has a whole set of mapping functions, including `mapcar` which is the list of results of applying a function, and `mapcan` is the concatentation of lists resulting from applying a function. We don't need this if sequences are unnested. A simple Scheme syntax:

`(do-each (` *var* *sequence*)

body)

This evaluates *sequence*. Then each value yielded by *sequence* is bound to *var*, and *body* is evaluated. The value of the `do-each` is the concatenation of the result of each evaluation of *body*.

7 Arrays

Since sequences don't nest, we need a real data structure that supports nesting. An array is single value that contains a multi-dimensional mapping from integer tuples to sequences. Usually each component of an array is a single value, but there seems to be no reason to disallow sequences of other lengths, especially for modifiable arrays. If our language is like Scheme in supporting first-class functions using the same namespace as other values, then it seems reasonable to use function call syntax for array indexing. APL-like array operations correspond to higher-order functions.

The primary operations on sequences are concatenation and iteration; the primary operation on arrays is indexing. It follows that a string should be a sequence of characters, not an array of characters: Random access in a string is not a semantically meaningful operation.

8 Attributes and keywords

XML elements may have named string-valued attributes, which are useful for specifying optional properties. Such attributes are similar to keyword parameters, so it makes sense to use the same syntax for both. XML attributes come before the "body" or "children" of the element, while in Common Lisp (and many other languages with keyword parameters) the keyword parameters come after the unnamed parameters. Listing the attributes first makes sense when attributes tend to be shorter, or their value may influence the processing of the main contents. These concerns suggest we follow XML conventions.

XML attribute values are restricted to string values, while Lisp keyword parameters may be arbitrary values. However, it is worth noting that the "meaning" of an attribute may be defined by a schema as having a typed value ("hatsize" being the canonical example). In any case our XML-friendly language will of course allow arbitrary expressions yielding arbitrary values.

9 Patterns

In XQuery a parameter list is a tuple of parameters, each of which may be bound to a sequence. An alternative model is to make the entire parameter list be a sequence. This makes it easier for functions to have a variable number of parameters - essentially they have a single sequence parameter. Thus there is no need for Scheme's separate `apply` or `call-with-values` methods. On the other hand, you cannot have two parameters both of which takes a sequence.

If there is logically only a single parameter, then the function definition needs a way to split the sequence up. ML-style pattern matching is an elegant solution. Extending these to nested regular patterns, as in some XML-oriented functional langues like CDuce [CDuce], is very elegant and powerful. The syntax of such patterns is an open question, especially in a Lisp-like language, but here is one possibility:

```
(define (map-body fun
                  (xhtml:html
                   (xhtml:head h)
                   (xhtml:body b)))
  (xhtml:html (xhtml:head h)
              (xhtml:body (fun b))))
```

This assumes that the prefix `xhtml:` prefix has been decleared such that functions in that namespace are element constructors. The function `map-body` is defined to match against a sequence of two values, where the first value is a function that gets bound to the variable `fun`. An element constructor in a pattern matches against an actual parameter value constructed using that constructor, so the second argument value must match an `html` element that contains a `head` child followed by a `body` child. The formal parameter variables `h` and `b` are matched against the contents of that `head` and `body` elements. The

body of the function applies the function `fun` to the `b` value, and constructs modified `head`, `body`, and `html` elements.

Fitting keyword parameters into this model matching can be done different ways. We could follow Common Lisp in treating a keyword parameter as a two-element sequence consisting of a keyword and a value. However, taking apart a parameter list using a pattern is probably easier if we treat a keyword-value-pair as a combined "attribute value". This allows a keyword parameter to be a sequence. In this model:

```
(foo font: "Helvetica"
     style: (values 'bold 'italic))
```

is syntactic sugar for:

```
(foo (attribute 'font
                "Helvetica")
     (attribute 'style
                (values 'bold 'italic)))
```

10 Graphics: Models and Views

Mozilla's XUL and Microsoft's XAML languages are convenient ways to describe the graphical layout and structure of a GUI window as hierarchical structure, using XML syntax similar to expressing a web page in HTML.

```
<button label="Yes"
        image="yes-image.png"
        oncommand="yes-action" />
```

The behavior of the application has to be expressed using a different programming language, for example JavaScript. A better integrated language as described above could describe both display and behavior more conveniently:

```
(button label: "Yes"
        image: "yes-image.png"
        oncommand:
          (lambda ()
            (format #t
              "Yes button pressed! % !")))
```

Both XUL and XAML describe the "view" aspect of an application. but they don't support model-view separation. Using a real programming language with variables and functions can do that. The UI library defines two classes of "GUI objects": A *model value* is collection of data. It may have a default way it is displayed, but it can also be transformed by an affine transform, and it may be displayed multiple times at once. A *view value* represents actual "screen real estate": an actual window or sub-window. Views may be nested inside other views, but any given view only appears once. A model constructor is a function that returns a model, while a view constructor is a function that returns a view. The parameters to a view constructors may be other (usually nested) views, model values (to be displayed in the view), or other values. If the parameter of a view constructors is a model where is a view is expected, a model may be converted to a view using a default view constructor.

Here is a simple example, where an image (a model) is used twice, once transformed, in a taskbar:

```
(define left-arrow
        (image "left-arrow.png"))
(define right-arrow
        (flip-vertically left-arrow))
(taskbar
  (button label: "Back"
          image: left-arrow
          oncommand: back-command)
  (button label: "Next"
          image: right-arrow
          oncommand: next-command))
```

Notice how various optional properties are specified using keyword parameters. Typesetting an article like this can also use keyword parameters:

```
(paragraph slant: 'italic
  "This is important!")
```

Alternatively one can use functions:

```
(paragraph
  (italic "This")
  " is important!")
```

The function function `italic` returns an "italic version" of the argument, while the function `color` takes

a color followed by one or more aruments to be displayed in that color. All of these work on models, and so the results are values that can be displayed many times. What does this mean? Consider:

```
(define blue-this
  (color 'blue "This "))
(define warning
  (color 'red
    blue-this
    "is important!"))
blue-this
(italic warning)
```

The result should be a blue non-italic `This` followed by an blue italic `This` followed by a red italic `is important!`. That means a function like `color` should change the default color, but it should not change the color property of any characters that alread have a color property. (There might be a separate `force-color` function that does override any color properties in the arguments.) An easy way to implement `color` is that it just creates a data structure referencing the arguments. When that value is typeset or displayed using a "graphics context", we save the graphic context's current color, change the color, display/typeset the arguments, and then restore the color. To display/typeset the arguments may involve nested color or font changes.

11 Lexical structure

Most of these ideas are compatible with different lexical syntaxes, though above I've assumed a Scheme-like syntax. A C/Java-like syntax is also possible, with a few more changes, including adding keyword function arguments. I have also explored a more Haskell-like syntax, which uses juxtaposition for function calls, and structure using white space and indentation. It is also appealing to use juxtaposition for tuple concatenation, though using juxtaposition for both application and concatenation might be confusing.

12 Links and more information

I name "Q2" refers to the language and implemenation where I'm trying out these and other ideas. For more on Q2 see (http://gnu.org/software/kawa/q2). The implementation, such as it is, included in the Kawa (http://gnu.org/software/kawa) source tree.

References

[CDuce] Benzaken, Castagna, and Frisch. *CDuce: an XML-Centric General-Purpose Language* ICFP SIGPLAN 38(9). 2003. (http://www.cduce.org/).

[Kawa] Per Bothner. *Kawa: Compiling Scheme to Java* Lisp Users Conference (Berkeley). 1998. (http://www.gnu.org/software/kawa/).

[XAML] Microsoft. *"Longhorn" Markup Language (code-named "XAML") Overview* (http://longhorn.msdn.microsoft.com/lhsdk/core/overviews/about%20xaml.aspx).

[XQuery] *XQuery 1.0: An XML Query Language* (http://www.w3c.org/XML/Query).

[XUL] Mozilla. *XML User Interface Language (XUL)* (http://www.mozilla.org/projects/xul/).