
— *Vernon Poole and Kevin Hayes*

* This monograph will be also published in Spanish (full version printed; summary, abstracts, and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <<http://www.ati.es/novatica/>>, and in Italian (online edition only, containing summary, abstracts, and some articles) by the Italian CEPIS society ALSI (*Associazione nazionale Laureati in Scienze dell'informazione e Informatica*) and the Italian IT portal Tecnoteca at <<http://www.tecnoteca.it/>>.

Variations on XML

Carlos Delgado-Kloos

The following is a brief overview of the main ideas behind XML (eXtensible Markup Language) which are seen in relation to the classical concepts of programming, so helping this monograph to become an essentially self-contained block. The aim is to give readers a better understanding of the contributions and limitations of this language.

Keywords: Functional Languages, Trees, XML.

1 Introduction

What is XML (eXtensible Markup Language)? Among the many definitions to be found on the Web, we have chosen this one: "XML is the acronym for eXtensible Markup Language, the universal format for structured documents and data on the Web. It is a simplified dialect of SGML (Standard Generalized Markup Language) which provides a metalanguage containing rules for constructing specialised markup languages".

The above definition talks about documents, markup, metalanguage. We are tempted to ask ourselves what, if anything, has all this got to do with classic programming languages? And how far can we go with XML? And what lies at the heart of XML?

Let's go beyond the metaphors and parables and the specialised terminology of the world of documents. Let's leave the basic application aside and concentrate on the characteristics of the language. First of all we will be looking at its special syntax, before moving on to the type of expressions we can construct with XML.

2 Syntax

First of all, here is an XML document which we will be using as an illustration throughout the article. It is a document describing the properties of a book:

```
<book>
  <title>XML for Dummies</title>
  <persons>
    <author>Ed Tittel</author>
    <author>Norbert Mikula</author>
    <author>Ramesh Chandak</author>
  </persons>
  <published by publishing-house="Hungry Minds"
  covers="soft"/>
</book>
```

Here we find structures of the form `<f>a b c</f>`, in which there are pairs of tags (one start tag `<f>` and one end tag `</f>`) between which there is content, which can recursively

contain pairs (well formed) of tags or plain text.

Basically, writing

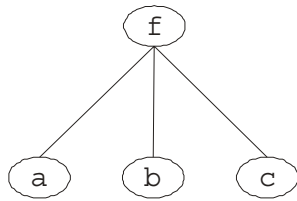
```
<f>
  a b c
</f>
```

in XML syntax is equivalent to this expression in the style of LaTeX

```
\begin{f}
  a b c
\end{f}
```

Carlos Delgado-Kloos got his degree in Telecommunications Engineering at the *Universidad Politécnica de Madrid* (UPM), Spain, in 1978, and a PhD in Computer Science at the *Technische Universität München*, Germany, in 1986. He is currently a full professor at the *Universidad Carlos III de Madrid* (Spain), where he also holds the positions of Director of the Department of Telematics Engineering, Director of the Master in e-commerce, and Director of the Nokia Chair. His current interests include design languages and techniques for hardware and software systems based on formal methods, as well as Internet based applications such as electronic publishing, tele-education and e-commerce. He has led many projects with European, national (Spanish Ministry) and bilateral (Spanish-German and Spanish-French) funding; among them it is worth mentioning the E-LANE e-learning project, coordinated by him. He has published over 120 articles in national and international conferences and journals, and has also written one book and co-edited four more. He holds or has held various positions in national and international bodies such as: vice-president of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*), vice-president of IFIP TC 10, secretary of IFIP WG 10.5, editor of the Springer journal 'Formal Aspects of Computing', deputy Director of Telecommunication Engineering at *Universidad Carlos III de Madrid*, and manager of the National Programme for Information and Communication Technologies at the Spanish Ministry for Science and Research. He has been a programme committee member or chair at more than 70 conferences and workshops, including vice-programme chair for the IFIP'92 World Computer Congress and programme chair for DATE 2002, Telecom I+D 2003, EduTech2004 and EUNICE2005. He belongs to several professional associations in Spain and abroad, ATI being one of them, and has published several papers in *Novática*, ATI's journal on whose Editorial Board he has served, and in *UPGRADE*. cdk@it.uc3m.es

The difference is "syntactic sugar". In either case this code can be represented by the tree



This notation, which could be called *circumfix*, includes the identifier before and after the arguments. Other variants to be found in LaTeX use *prefix* notation instead of the notation described above. For example:

```

\{a b c}
o
{\f a b c}
  
```

Which is very reminiscent of Lisp *s-expressions* (or symbolic expressions):

```
(f a b c)
```

As in Lisp both data and programmes are sequences, we can apply one Lisp programme (a sequence) to another (another sequence) and so obtain a new sequence (a programme or datum). Self-application made it possible for programmes to be automatically modified, which led to some interesting experiments. With XML we can play some similar games: We can apply an XSLT (eXtensible Stylesheet Language Transformations, an XML application) programme to a document written in another XML application to obtain a document, possibly in accordance with a third XML application. The advantage of having a single syntax is that the same tools can be used (for editing, analysing, manipulating, transforming, etc.) in different contexts.

There are those who have criticised XML notation as overly verbose. Using a prefix based variant, the notation shown below would have been sufficient. The need to repeat the element at the end of the line can be an aid to understanding and prevent errors, but it is actually rare for a document to be written manually.

```

<book>
  <title>XML for Dummies</>
  <persons>
    <author>Ed Tittel</>
    <author>Norbert Mikula</>
    <author>Ramesh Chandak</>
  </>
  <published by publishing-house="Hungry Minds"
  
```

```

    covers="soft"/>
  </>
  
```

3 Computing

So, is XML nothing more than Lisp with a new syntax? Certainly not. In the examples above, element **f** in XML is the mark we associate with the sequence **a b c**, and in Lisp it is the function we apply to the sequence of arguments **a b c**. In Lisp the symbol **f** (for function) is associated with a meaning by means of a function definition in the programme itself, so that we can directly interpret the application of that **f** in an expression. In XML that interpretation is different. The first thing to do is simply to create the tree, in order to associate it later with one or several interpretations. If we are thinking of a view for a user, it will be a display interpretation (for example, by means of an XSLT translation into HTML (HyperText Markup Language) for viewing on a browser).

XML syntax would not per se prevent meaning from being associated to elements in the document itself. This has in fact been done in a series of proposals, such as [1] and [6]. The latter defines the languages ConciseXML [2] and Water [7] which enable lambda abstractions to be associated to symbols so as to be able to perform computations. Let's see an example of this:

```

<!-- define method 'factorial' -->
<defmethod factorial n>
  <if> n.<is 0/> 1
  else n.<times <factorial n.<minus 1/> /> />
</if>
</defmethod>

<!-- call 'factorial' passing argument 7 -->
<factorial 7/>
  
```

In order to have a better understanding of the above expressions, we should bear in mind that in ConciseXML attribute values can be of any type, not just strings, and that it is not necessary to include attribute names, since position is used to know which attribute is being referred to.

The authors go one step further and define web services while still using the language:

```

<!-- serve 'factorial' web service at port 8383 -->
<server factorial port=8383/>

<!-- open browser and call web service 'factorial' -->
<open_browser_window "http://localhost:8383/?n=7"/>

<!-- define call to 'factorial' and call service -->
<web "http://localhost:8383/?n=7"/>.<execute/>
  
```

This is an opposite approach to the one used by SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language) and UDDI (Universal Description, Discovery and Integration), in which separate languages are defined for individual aspects.

What should become apparent at this point is that this verbose XML notation is not really ideal for representing computations, despite the fact that ConciseXML is the result of a huge simplification of XML syntax. That was the conclusion reached by the authors of Curl [3][4] when they propose a language similar to Lisp in terms of its syntax (but with curly braces instead of parentheses), which in addition to programming, also enables us to give formatting instructions in the manner of HTML or LaTeX. Thus the source text:

We can write a word in {italic italics} and do calculations like this one: 12+3 equals {+ 12 3}.

is interpreted as:

We can write a word in *italics* and do calculations like this one: 12+3 equals 15.

in which the *italic* function has the effect of displaying the content (the argument) in italics and the + function simply adds: presentation and computation are thus in the same uniform notation. HTML and JavaScript or other scripting languages achieve the same effect, but they require the use of two languages with very different characteristics and syntaxes.

4 Data Structures

Thus, XML could perfectly well have been a programming language, but it stayed as a language for defining data in the form of trees or nested lists. Our original example can be represented graphically by the tree shown below in Figure 1.

Or it can be displayed in the form of nested boxes, as shown in Figure 2.

It is a well known fact that functional languages, such as Haskell or Gofer, have a concise and easy to use notation for defining data types. Let us take a look at the relationship between an element declaration (called a Document Type Definition, or DTD, following the custom of SGML in the world of publishing) and a datatype declaration in Haskell [5]. A possible DTD for the example of the books could be as follows:

```

<!ELEMENT book (title, persons, published?)>
<!ELEMENT title #PCDATA>
<!ELEMENT persons (author* | publisher*)>
<!ELEMENT author #PCDATA>
<!ELEMENT publisher #PCDATA>
<!ELEMENT published EMPTY>
<!ATTLIST published
    publishing-house CDATA #IMPLIED
    covers (hard | soft) #REQUIRED>
    
```

For each element, there is a declaration of its content (!ELEMENT) and possibly its attribute list (!ATTLIST). The content is specified using sequences (,), alternatives (|), repetitions (* o +) and options (?). Nested elements are allowed and in the last instance we will have plain text

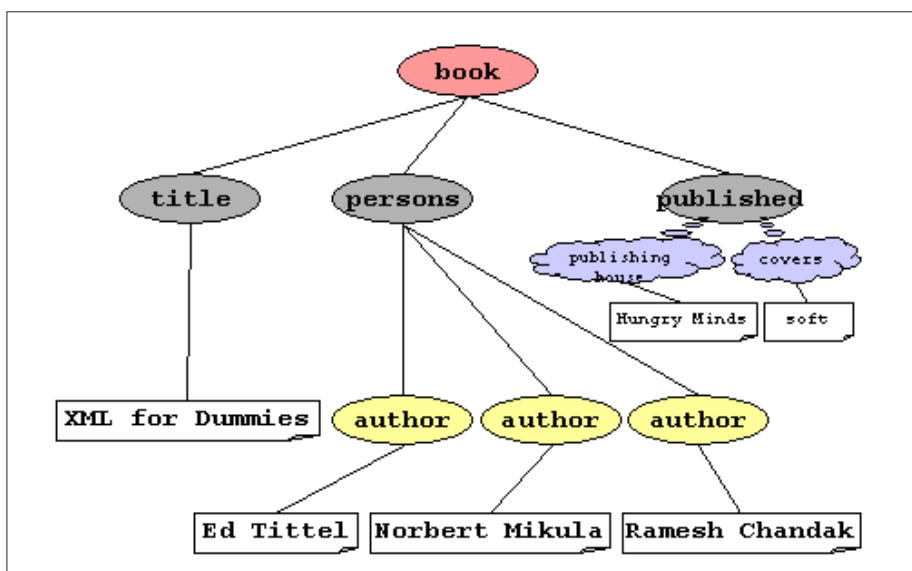


Figure 1. Tree Display.

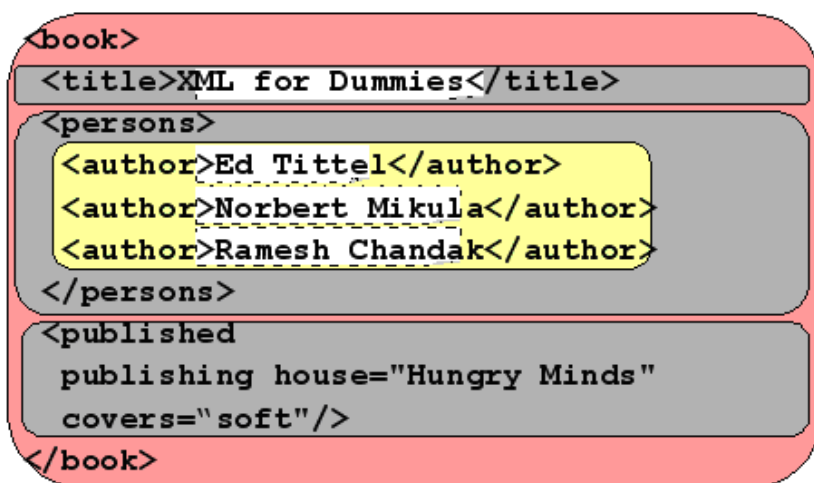


Figure 2. Nested Box Display.

(#PCDATA). Thus, the content of the element **persons** (that is, anything we find between a **<persons>** tag and a **</persons>** tag is either a sequence of **author** elements or a sequence of **publishing-house** elements. Attributes have a name and a value. Values can be strings of characters (**CDATA**) or belong to enumerative types defined in an ad hoc manner. Attributes can be obligatory (**#REQUIRED**) or optional (**#IMPLIED**).

The definition of a datatype in Haskell corresponding to the above DTD could have the following form:

```

data Book = Book Title Persons (Maybe Published)
data Title = Title String
data Persons = PersonsAuthor [Author]
               | PersonsPublishing-house [Publishing-house]
data Author = Author String
data Publishing-house = Publishing-house String
data Publishing-house = Publishing-house PublishedAt
data PublishedAt = PublishedAt {
  Publishing-house :: Maybe String,
  covers :: Hard | Soft }

```

As we can see, there could hardly be a more direct correspondence. Each element declaration corresponds to a new type declaration in Haskell. Sequence (,) corresponds to a product of types; i.e. values with a constructor, which is represented by the simple juxtaposition of types (there is no separating symbol). Alternative (|) corresponds to the union of types; i.e. values with several constructors. The corresponding symbol is also a vertical bar (|). Repetition (*) corresponds to list ([]) and option (?) to **Maybe**. Attribute lists are also easily translatable.

Attributes are not in sequential order as is the case with element sequences. Attributes are identified by a name rather

than their position. This means that Haskell field labels can be used. Optional attributes have **Maybe** in their type and if their values are restricted to a certain set, a corresponding enumerative type is defined (|).

The XML document we saw above corresponds to the following **Book** type value in Haskell:

```

Book (Title "XML for Dummies")
  PersonasAuthor [(Author "Ed Tittel"),
                 (Author "Norbert Mikula"),
                 (Author "Ramesh Chandak")]
  Published (Published At
    {Publishing-house="Hungry Minds",
     covers=Soft})

```

Haskell and DTDs do not correspond in their entirety; there are some differences. As well as the change in notation type (prefix/circumfix), in Haskell constructors are used to form the value, while in DTDs the concept of constructor does not exist per se, and type is used instead. This difference is especially important in the union of types (|).

For the sake of completeness we also include the corresponding XML schema:

```

<xs:element name="book">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="string"/>
      <xs:element name="persons"
type="personsType"/>
      <xs:element name="published"
type="publishedType"
minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

```

```
</xs:complexType>
</xs:element>
```

```
<xs:complexType name="personsType">
  <xs:choice>
    <xs:element name="author" type="string"
      minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="editor" type="string"
      minOccurs="0" maxOccurs="unbounded"/>
  </xs:choice>
</xs:complexType>
```

```
<xs:complexType name="publishedType">
  <xs:attribute name="publishing-house"
    type="string" use="optional"/>
  <xs:attribute name="covers"
    type="pastasType" use="required"/>
</xs:complexType>
```

```
<xs:simpleType name="coversType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="hard"/>
    <xs:enumeration value="soft"/>
  </xs:restriction>
</xs:simpleType>
```

We can see that the concision of DTDs is lost in XML schemas. Also regular expression notation is abandoned in favour of a more intuitive and verbose notation. However, gains are made in other areas: XML syntax, a richer datatype system, open content model, etc.

5 Conclusion

We can now finally provide a more technical definition of XML. We would define XML as a functional datatype definition language with a circumfix notation and without constructors as such (or, to be more precise, it uses type names as constructors). Types are defined separately from the values of those types (the first is done in a DTD and the second in a specific document). Semantics can also be associated with constructors, but, again, this is done separately. It is necessary to work first with the initial algebra and then, by using other mechanisms (normally XSLT transformations), the result can be translated into interpretable structures (HTML for example).

The aim of these brief reflections on XML has been to refer the new concepts and terminology that XML has introduced to the classical concepts of programming. Very often, when a new paradigm or language is defined, new metaphors and unknown terms are used, which makes it hard to tell what is new from what already existed. In the case of XML, terminology has also been inherited from the world of publishing, as it should not be forgotten that SGML, XML's forerunner, originally came from that world. With

these reflections we hope we have put XML in its proper context and have clarified some of its constructions along the way.

Acknowledgements

I would like to thank Bernhard Möller for his comments on previous versions of this article. This paper was prepared as part of the SIEMPRE project (TIC2002-03635), funded by the National ICT Programme of the Spanish Interministerial Committee of Science and Technology.

Translation by Steve Turpin

References

- [1] Peter T. Breuer, Carlos Delgado Kloos, Vicente Luque Centeno, Luis Sánchez Fernández. Higher Order Applicative XML Documents, Lecture Notes in Computer Science 2941, pp91-107. Heidelberg, Springer-Verlag, 2004.
- [2] ConciseXML. <<http://www.concisexml.org>> [consulted Nov. 2004].
- [3] Curl. <<http://www.curl.com>> [consulted Nov. 2004].
- [4] Bruce Mount, Gary Gray, Nikhil Damle. Curl Programming Bible. Wiley, 2002
- [5] Simon Peyton Jones. Haskell 98 Language and Libraries. Cambridge, Cambridge University Press, 2003.
- [6] Mike Plusch. Water: Simplified Web Services and XML Programming. Indianapolis, Indiana. Wiley. 2002.
- [7] Water, language for Web services. <<http://www.waterlanguage.com>> [Consulted Nov. 2004].
- [8] Malcolm Wallace, Colin Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? Proc. International Conference on Functional Programming, ACM, Paris, Sept 1999, <<http://www.cs.york.ac.uk/fp/HaXml/icfp99.html>> [consulted Nov. 2004].