XML

Dr Peadar Grant

January 22, 2024

1 eXtensible Markup Language

Extensible Markup Language (XML) is a **markup** language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It's a subset of SGML, of which HTML is also an *application*. The tag names are up to you.

1.1 SGML

Standard Generalised Markup Language (SGML) originated as a means to store and transfer US government and military documents in machine-readable form, over a timespan of several decades. SGML uses **tags** to mark up a document. One of the best known examples of SGML is HTML.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Web page</title>
  </head>
  <body>
    <h1>Web page</h1>
    <p>This is a web page.</p>
  </body>
</html>
```

For many applications, SGML was too complicated, and so XML was designed. Apart from HTML and Document Type Declarations, you are unlikely to encounter SGML in its non-XML form very frequently. XML documents themselves are valid SGML, but have additional restrictions on their layout. (SGML documents are not valid

XML.)

1.2 Sample XML document

```
<?xml version="1.0"?>
<note>
  <to>Phil</to>
  <from>Celia</from>
  <heading>Weekend arrangements</heading>
  <body>I will be arriving on the 5 o clock train - see you then.</body>
</note>
```

1.3 Design Goals

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.

1.4 Benefits of XML

1. You define the **vocabulary**, so it provides flexibility.
2. XML is **simple** and used widely.
3. Machine and **human readable**.
4. **Lots of APIs** and parsers to handle XML.
5. XML Data **can be checked** for correctness.

1.5 Issues with XML

1. XML documents require more storage than fixed length or comma delimited files.
    - But, they compress well!
2. Can help but not does not solve the politics of competing formats (multiple vocabs).
3. Definitions can be overly nested and made too complex.

2 Structure of an XML doc

XML has a Directed Acyclic Graph tree structure, built from **nested** tree of **Elements** delimited by **tags**. Elements are **case-sensitive**.

```
<?xml version="1.0" ?>
<doc>

  <!-- CLOSED ELEMENTS -->
  <!-- a closed element's content is placed between start and end tags -->
  <element_name>Content</element_name>

  <!-- for example: -->
  <artist>Nine Inch Nails</artist>


  <!-- EMPTY ELEMENTS -->
  <!-- an empty element is composed of a single tag -->
  <element_name />
```

```
<!-- for example: the empty element -->
<artist />
<!-- is just a shorthand for -->
<artist></artist>


<!-- NESTING ELEMENTS -->

<!-- an element may contain one or more child elements -->
<parent>
  <child>...</child>
  <child />
  <anotherchild>...</anotherchild>
  <child>
    <grandchild>...</grandchild>
  </child>
</parent>
```

```
  <!-- for example: -->
  <band>
    <name>Pink Floyd</name>
    <member>David Gilmour</member>
    <member>Roger Waters</member>
    <member>Nick Mason</member>
    <member>Richard Wright</member>
  </band>

</doc>
```

2.1 Nesting rules

- Elements nested inside a *parent* element are called *child* elements.
- Elements must be nested correctly. Child elements must be enclosed within their parent elements.
- All elements must be nested within a single document or root element.
- There can be only one **root** element in a document.
- Indentation, while helpful, is just visual.

2.2 XML declaration

The XML declaration is always the first line of code in an XML document, and tells the **processor** that what follows is XML. It can also provide information about how the parser should interpret the document.

For now, the version is always 1.0. This attribute allows the program (or person) reading the document to read older versions if specifications change.

2.3 XML comments

Comments may appear anywhere after the declaration. The syntax for comments is:

```
<!-- comment text -->
```

Two or more dashes should not appear one after the other in a comment.

2.4 Free-format

XML documents are free-format. For example, the document:

```
<?xml version="1.0" ?><result><student>28191800</student><mark>20</mark></result>
```

is the exact same as:

```
<?xml version="1.0" ?>
<result>
  <student>28191800</student>
  <mark>20</mark>
</result>
```

3 Attributes

An attribute is a feature or characteristic of an element. Attributes are text strings and must be placed in single or double quotes. Use that which makes most sense in deciding whether to store the main data in an element or attribute. Often an element is a better choice.

```
<doc>

  <!-- attribute syntax -->
  <element_name attribute="value"> ... </element_name>

  <!-- example: -->
  <note id="12">This is a note...</note>

  <!-- often empty elements with attributes
       are used to store data -->
  <items>
    <item id="12" price="3.30" />
  </items>
```

```
</doc>
```

3.1 Mixing elements and attributes

```
<bookstore>
  <book category="CHILDREN">
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="WEB">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

4 Well-formed XML

When XML text satisfies the syntactic rules as laid out in the XML specification, we say that it is well-formed [Goldfarb, 2003].

4.1 Criteria

1. All XML elements must have a closing tag, except empty elements.
2. Tags are case sensitive.
3. All elements must be properly nested.
4. Documents must have one single root element.
5. Attribute values must be quoted.
6. Certain characters must be replaced with *Entity References*.

4.2 Entity references

Entity references replace certain characters as detailed in Table 1.

Normal symbol	Meaning	Replace with
<	less than	<
>	greater than	>
&	ampersand	&
’	apostrophe	'
”	quotation mark	"

Table 1: Entity References

4.3 Non well-formed XML

Non well-formed XML will cause errors in applications and parsers that expect strict XML syntax.

4.4 Line termination

By convention, different OSes use different delimiters to signal the end of a line in a text file, Table 2. XML specifies that new lines are **always** stored as LF, regardless of the operating system.

Operating system	Delimiters
Windows	CR LF
Unix, Mac OS X	LF
Mac (System 9 and below)	CR

Table 2: Newline characters

4.5 Checking for well-formed XML

The `xmllint` utility can check XML well-formedness.

```
# consult manpage
man xmllint

# basic usage
xmllint filename.xml
```

5 Vocabulary

Although the structure and syntax of XML documents is rigorously specified, you are free to choose a suitable vocabulary of element and attribute names for the application of interest. Some common vocabularies are listed in Table 3.

Vocabulary	Description
XHTML	a variant of HTML that complies with XML restrictions
SOAP	message format for web services
OpenOffice XML	Microsoft's current office word-processing format
Opendoc	standard formats for (non-Microsoft Office) office documents
Docbook XML	markup for writing (primarily) technical documentation

Table 3: Common standardised XML vocabularies

In addition, there are pre-existing vocabularies for many specialised applications. It's always good to use a pre-existing vocabulary if possible.

5.1 Developing an XML vocabulary

The process is:

1. List out the possible fields (tag names).
2. Draw a tree diagram.
3. Markup some sample XML data.
4. Factor in potential business rules and add them to our tree diagram.
5. Optionally formalise the vocabulary in a machine-readable form.

5.2 Worked example

As an example, we will develop a vocabulary that would let you specify address data for a person.

5.2.1 Possible fields

Start with:

- firstname
- secondname
- address1
- address2
- city
- country

Each of these can be specified as tags like

```
<firstname></firstname>
```

But: a person's name is not really part of an address. What happens if a person has multiple addresses? So we really need a nested structure:

```
<person>
  <address>
  </address>
</person>
```

5.2.2 Diagram

The nested structure of our document is then laid out as a **tree diagram**, Figure 1.

5.2.3 Sample data

```
<person>
  <firstname>Joe</firstname>
  <secondname>Murphy</secondname>
  <address>
    <street>23 Main Road</street>
    ... ...
```

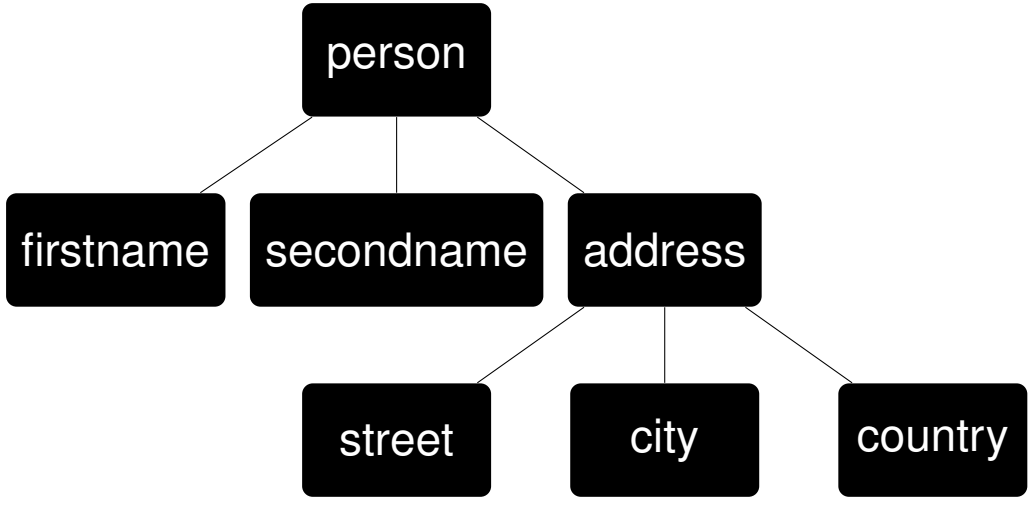

Figure 1: Tree diagram for address example

```
    <country>Ireland</country>
  </address>
</person>
```

5.2.4 Rules

We then decide on a number of business rules to determine if an element is optional, and if there are any constraints on its **multiplicity**:

- Person can have multiple first names.
- Person can have only one second names.
- Person can have multiple addresses.
- An address can have a few streets (lets say between 1 and 3).

These rules are then added to the tree diagram, Figure 2.

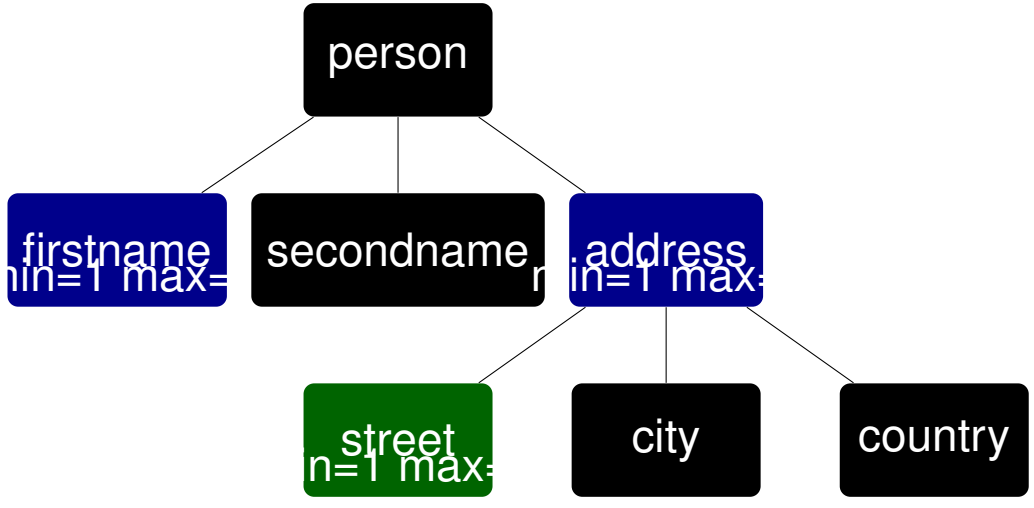

Figure 2: Tree diagram with business rules for address example

6 XPath

XPath expressions allow us to easily retrieve the contents of elements and attributes in an XML document, by writing expressions.

The XPath expression language is standardised and works similarly in many languages - Java, C#, C++/libxml/JavaSc etc.

XPath uses **expressions** and standard **functions** to return **nodes** that are **related** to other nodes.

6.1 Expressions

XPath uses path expressions to select nodes or node-sets in an XML document. These expressions are very similar in theory to file paths or URLs: Examples:

- `/Users/peadar/Intro.pdf`
- `/Users/peadar/Documents/Intro.pdf`
- `http://xyz.com/dir/file2.xml`
- `http://xyz.com/dir/sd/file2.xml`

6.2 Standard functions

XPath includes over 100 built-in functions. There are functions for string values, numeric values, date and time comparison, Boolean values, and more complex manipulations.

Functions usually **operate** on nodes or node-sets returned using an expression.

6.3 Nodes

In XPath, there are seven kinds of nodes:

1. **Element**
2. **Attribute**
3. **Text**
4. Namespace (see later)
5. Processing instruction
6. Comment
7. **Document**

6.4 Relationships

Effective use of XPath requires a firm understanding of the relationships present between various nodes, Table 4.

Relationship	Meaning
Parent	Each element and attribute has one parent.
Children	Element nodes may have zero, one or more children.
Siblings	Nodes that have the same parent.
Ancestors	A node's parent, parent's parent, etc.
Descendants	A node's children, children's children, etc.

Table 4: Important relationships between nodes

7 Basic selection expressions

XPath uses path expressions to select nodes or node-sets in an XML document. The node is selected by following a path or steps. The most useful path expressions are listed in Table 5.

Expression	Selects
elementname	all “elementname” elements
/	from root node / previous node
//	from anywhere in the document
..	parent of the current node
@attribname	attribute “attribname”
text()	text content of current node

Table 5: XPath expressions

7.1 Writing path expressions

A location path can be absolute or relative. An absolute location path starts with a slash (/) and a relative location path does not. In both cases the location path consists of one or more steps, each separated by a slash:

```
An absolute location path:

/step/step/...

A relative location path:

step/step/...
```

Each step is evaluated against the nodes in the current node-set.

7.2 Example

Using the bookstore document, a number of XPath expressions are shown in Table 6.

Expression	Selects
`bookstore`	all elements named “bookstore”
`/bookstore`	root element bookstore
`bookstore/book`	child books of bookstore
`//book`	all books in document
`bookstore//book`	descendant books of bookstore

Table 6: Expressions on `bookstore.xml` document

7.2.1 Selecting text content

Note that we must use the `text()` function to get just the text content of an element, rather than the element itself:

- `/bookstore/book/title` = title element of each book
- `/bookstore/book/title/text()` = title text content of each book

8 Tooling

8.1 Command-line

The `xmllint` tool can evaluate XPath expressions at the command-line.

```
xmllint --xpath '/route/station[amenity]' stations.xml
```

8.2 Python

Python's `lxml` library can work with XPath expressions:

```
import lxml.etree as ET

root = ET.parse('stations.xml')
for name in root.xpath('/route/station/name/text()'):
    print(name)
```

9 Predicates

Predicates are used to find a **specific node** or a **node containing a specific value**.

They are embedded in square brackets after the element name to which they refer. Examples:

- `/element/element[predicate]`
- `/element[predicate]/element`
- `/element[p1]/element[p2]`

9.1 Posessive predicates

Posessive predicates select elements that have matching children, Table 7.

Expression	Selects
//**parent[@attribute]**	all “parent” elements with “attribute”
//**gp/parent[child]**	all “parent” elements with “child”
//**gp/parent[child]/child2**	all “child2” of above...

Table 7: Posessive predicates

9.2 Value-based predicates

Value-based predicates select elements based on whether they or their child elements or attributes match specific values, Table 8.

Expression	Selects
`//parent[@attribute='value']`	all “parent” elements with “attribute”==value
`//gp/parent[child`$< v$`]`	all “parent” elements where “child” element has value $< v$
`//gp/parent[child`$< v$`]/child2`	all “child2” of above...

Table 8: Value-based predicates

9.3 Positional predicates

Positional predicates allow selections based on element ordering usually using XPath functions, Table 9.

Expression	Selects
/parent/child[*n*]	the *n*th child element of the parent.
/parent/child[last()]	the last child element of the parent.
/parent/child[last()-*n*]	the last but *n* child element of the parent.
/parent/child[position()$<$ *n*]	the first $n-1$ child elements of the parent.

Table 9: Positional predicates

9.4 Other predicates

There are a lot more things you can do with predicates — see online tutorials for more details W3 Schools [2014].

9.5 Combining two node sets

To combine two node sets, we use the | (vertical bar) operator between two separate XPath expressions. Example:

```
\texttt{/path/path[predicate] $\vert$ /path2/path2[predicate]}
```

9.6 Operators

When writing a predicate we can use any of the usual binary comparison operators, Table 10.

Operator	Meaning
=	equal
!=	not equal
<, >	less than, greater than
<=, >=	less/greater than or equal to
or, and	or/and

Table 10: XPath operators

You can use math operators in your comparisons, but **be careful of division**, Table 11.

Operator	Meaning
+	addition
−	subtraction
*	multiplication
div	division
mod	modulus / division remainder

Table 11: XPath operators

10 Wildcards

XPath wildcards can be used to select unknown XML elements, Table 12

Expression	Matches
`*`	any element node
`@*`	any attribute node
`node()`	any node of any kind

Table 12: XPath wildcards

11 Axes

An axis defines a node-set relative to the current node.

11.1 Direct axes

Direct axes specify the node itself, its parent or children, or its attributes, Table 13.

Axis	Selects
self	the current node
parent	the parent of the current node
child	all children of the current node
attribute	all attributes of the current node

Table 13: Direct axes

11.2 Cross-generational axes

Cross-generational axes allow us to specify nodes ancestral or descending nodes, Table 14.

Axis	Selects
ancestor	all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	current node and all of its ancestors
descendant	all descendants (children, grandchildren, etc.) of the current node
descendant-or-self	current node and all descendents

Table 14: Cross-generational axes

11.3 Positional axes

Positional axes allow us to select nodes before or after other nodes in the document, Table 15.

Axis	Selects
following	everything in the document after the closing tag of the current node
following-sibling	all siblings after the current node
preceding	all nodes before current node, except ancestors, attribute nodes and namespace nodes
preceeding-sibling	all siblings before the current node

Table 15: Positional axes

11.4 Writing location path expressions

With predicates and axes, a full step consists of:

- an axis (defines the tree-relationship between the selected nodes and the current node)
- a node-test (identifies a node within an axis)
- zero or more predicates (to further refine the selected node-set)

The syntax for a location step is:

```
axisname::nodetest[predicate]
```

11.5 Example

Here are some XPath expressions using axes for the bookstore.xml file:

Expression	Selects
child::book	all book nodes that are children of the current node
attribute::lang	the lang attribute of the current node
child::*	all element children of the current node
attribute::*	all attributes of the current node
child::text()	all text node children of the current node
child::node()	all children of the current node
descendant::book	all book descendants of the current node
ancestor::book	all book ancestors of the current node
ancestor-or-self::book	all book ancestors of the current node - and the current as well if it is a book node
child::*/child::price	all price grandchildren of the current node

Table 16: XPath axes expressions example

12 XQuery

XQuery is to XML what SQL is to database tables. XQuery was designed to **query** rather than *navigate* XML data.

12.1 Syntax

XQuery is built on top of **XPath**, and uses the **same expressions and predicates**, augmented by the queries made up of by the so-called **FLWOR** syntax, Table 17.

Component	Function
for	creates a sequence of nodes
let	binds a sequence to a variable
where	filters nodes on a boolean expression
order by	sorts the nodes
return	evaluated once per node

Table 17: FLWOR syntax

12.2 Rules

1. XQuery is case-sensitive
2. XQuery elements, attributes, and variables must be valid XML names
3. An XQuery string value can be in single or double quotes
4. Local variables (usually to represent the current node) are preceeded by the $ character.
5. XQuery comments are delimited by (: and :), e.g. (: XQuery Comment :)

12.3 Examples

We will use the following XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<bookstore>

  <book category="COOKING">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>

  <book category="CHILDREN">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
```

```
  <price>29.99</price>
</book>

<book category="WEB">
  <title lang="en">XQuery Kick Start</title>
  <author>James McGovern</author>
  <author>Per Bothner</author>
  <author>Kurt Cagle</author>
  <author>James Linn</author>
  <author>Vaidyanathan Nagarajan</author>
  <year>2003</year>
  <price>49.99</price>
</book>

<book category="WEB">
  <title lang="en">Learning XML</title>
  <author>Erik T. Ray</author>
  <year>2003</year>
```

```
    <price>39.95</price>
  </book>

</bookstore>
```

12.3.1 Basic selection

We are asked to:

> Select all the title elements under the book elements that are under the bookstore element that have a price element with a value that is higher than 30.

Write down a suitable XPath expression for this:

This same query could be expressed using the FLWOR syntax as follows:

```
for $x in /bookstore/book
where $x/price > 30.00
return $x/title
```

Obviously, the XPath expression is more concise. However, we have a problem if we want the returned element list to be ordered or otherwise modified:

```
for $x in /bookstore/book
where $x/price > 30.00
order by $x/title
return $x/title
```

We can simplify by using XPath predicates instead of (or in addition to) the WHERE clause:

```
for $x in /bookstore/book[price>30]
order by $x/title
return $x/title
```

12.4 Counting loop iterations

The at keyword can be used to count the iteration:

```
for $x at $i in /bookstore/book/title
return {$i}. {data($x)}
```

12.5 Multiple order by clauses

The order by clause is used to specify the sort order of the result. Here we want to order the result by category and title:

```
for $x in /bookstore/book
order by $x/@category, $x/title
return $x/title
```

References

Charles F Goldfarb. *The XML Handbook*. Prentice Hall, 2003.

W3 Schools. Xpath tutorial. `http://www.w3schools.com/xpath/default.asp`, 2014.