Alessandro Stolfo

Summary: XML Syntax

2. XML Fundamentals

2.1 - 2.2 XML Documents and Elements, tags and Character Data

An XML document always contains text. The XML parser does not care about the file name, which could be any string, with any extension.
In an XML doc we can distinguish between **markup** and **character data** (normal text inside tags). Tags are the simplest form of markup. Unlike HTML, you are allowed to make up tags as you go along.
XML, unlike HTML, is case-sensitive. "<Person>" is not the same as "<PERSON>".

An element can contain **child elements**. A child element has exactly one **parent element**. Every XML document has one element that does not have a parent, the **root element**. It is the first element in the document and the one that contains all other elements. A well formed document has exactly one root element.

Elements that contain both child elements and non-whitespace character data are said to have **mixed content**.

2.3 Attributes

An attribute is a name-value pair attached to the element's start-tag. Names are separated from values by an equals sign and optional whitespace. Values are enclosed in single or double quotation marks. Attribute order inside a tag is not relevant. Element may have no more than one attribute with a given name.

Among informaticians a subject of debate is whether one should use child elements or attributes to hold information. Some argue that attributes are for metadata about the element while elements are for the information itself. Other argue that the distinction between data and metadata is not immediate.

2.4 Names

The rules for XML element names are also the rules for XML attribute names. These are referred to simply as XML names.

Element and other XML names may contain essentially any alphanumeric character. This includes the standard English letters A through Z and a through z as well as the digits 0 through 9. **XML names may also include non-English letters**, numbers, and ideograms, such as ö, ç, Ω, 串. They may also include these three punctuation characters: the underscore (_), the hypen (-), the period (.). The colon is allowed, but its use is reserved for namespaces. **XML names may only start with letters, ideograms, or the underscore character**. There is no limit to the length of an element or other XML name.

2.5 References

XML predefines exactly five entity references. These are:

< (<) (must always be escaped)

& (&) (must always be escaped)

> (>)

" (“)

' (')

The character data inside an element must not contain a raw unescaped opening angle bracket (<). This character is always interpreted as beginning a tag. Character data may not contain a raw unescaped ampersand (&) either. Only < and & must be used instead of the literal characters in element content. **The others are optional**. " and ' are useful inside attribute values where a raw " or ' might be misconstrued as ending the attribute value. Although there's no possibility of an unescaped greater-than sign (>) being misinterpreted as closing a tag it wasn't meant to close, > is allowed mostly for symmetry with <.

2.6 CDATA Section

A CDATA section is set off by <![CDATA[and]]>. Everything between the <![CDATA[and the]]> is treated as raw character data (not markup). This means that you do not have to use escape characters, since there is no chance of misinterpretation by the parser.

The only thing that cannot appear in a CDATA section is the CDATA section end delimiter,]]> .

2.7 Comments

Just as in HTML, comments begin with <!-- and end with the first occurrence of —>. The double hyphen -- must not appear anywhere inside the comment until the closing -->. In particular, a three-hyphen close like ---> is specifically forbidden.

Comments may appear anywhere in the character data of a document. They may also appear before or after the root element. However, comments may not appear inside a tag or inside another comment.

2.8 Processing Instructions

XML provides the processing instruction as an alternative means of passing information to particular applications that may read the document. A processing instruction begins with <? and ends with ?>. They are markup, but they're not elements.

Immediately following the <? is an XML name called the target , possibly the name of the application for which this processing instruction is intended or possibly just an identifier for this particular processing instruction.

Different processing instructions may have totally different syntaxes and semantics.

2.9 The XML Declaration

XML documents should (but do not have to) begin with an XML declaration. If an XML document does have an XML declaration, then that declaration must be the first thing in the document. It must not be preceded by any comments, whitespace, processing instructions, and so forth. The XML declaration looks like a processing instruction with the name xml and with version, standalone, and encoding pseudo-attributes:

- The **version** attribute should have the value 1.0. Under very unusual circumstances, it may also have the value 1.1.
- As far as regards the encoding, by default, XML documents are assumed to be encoded in the **UTF-8 variable-length encoding** of the Unicode character set. This is a strict superset of ASCII, so pure ASCII text files are also UTF-8 documents.
  However, XML also allows documents to specify their own character set with the (optional) encoding attribute inside the XML declaration. If it is omitted and no metadata is available, the Unicode character set is assumed.
- If the **standalone** attribute has the value no, then an application may be required to read an external DTD (that is, a DTD in a file other than the one it's reading now) to determine the proper values for parts of the document. The standalone attribute is optional in an XML declaration. If it is omitted, then the value no is assumed.

2.10 Well-Formedness

Every XML document, without exception, must be well-formed.
The simplest way to do this is by loading the document into a web browser that understands XML documents. An alternative is to use an XML parser directly.

4. Namespaces

Some documents combine markup from multiple XML applications. In some cases, these applications may use the same name to refer to different things. In other cases, there may not be any name conflicts, but it may still be important for software to determine quickly and decisively which XML application a given element or attribute belongs to.

4.2 Namespace Syntax

Namespaces distinguish between elements with different meanings but the same name by assigning each element a URI, called **namespace names**. Elements with the same name but different URIs are different kinds of elements. Elements with the same name and the same URI are the same kind of element.

Since URIs frequently contain characters such as /, %, and ~ that are not legal in XML names, short prefixes such as rdf and xsl stand in for them in element and attribute names. Each prefix is associated with a URI.

Prefixed elements and attributes in namespaces have names that contain exactly one colon. Everything before the colon is called the **prefix**. Everything after the colon is called the **local part**. The complete name, including the colon, is called the **qualified name** , **QName**, or raw name.
Prefixes are bound to namespace URIs by attaching an **xmlns:*prefix*** attribute to the

prefixed element or one of its ancestors. Bindings have scope within the element where they're declared and within its contents.

Namespace URIs do not necessarily point to any actual document or page. In fact, they don't have to use the http scheme. They might even use some other protocol like mailto in which URIs don't even point to documents.

You can indicate that an unprefixed element and all its unprefixed descendant elements belong to a particular namespace (called **default namespace**) by attaching an xmlns attribute with no prefix to the top element. Note that default namespaces only apply to elements, **not to attributes**. Unprefixed attributes are indeed in no namespace.

Of course, default namespace does not apply to any elements or attributes with prefixes. However, an unprefixed child element of a prefixed element still belongs to the default namespace.

4.3 How Parsers Handle Namespaces

A namespace-aware parser does add a couple of checks to the normal well-formedness checks that a parser performs. Specifically, it will reject documents that use unmapped prefixes and any element or attribute names that contain more than one colon.

4.4 Namespaces and DTDs

Namespaces are completely independent of DTDs and can be used in both valid and invalid documents. Namespaces do not in any way change DTD syntax nor do they change the definition of validity. For instance, the DTD of a valid document that uses an element named dc:title must include an ELEMENT declaration properly specifying the content of the dc:title element. The DTD cannot omit the prefix and simply declare a title element.

A problem that might occur is that changing the prefix requires changing all declarations that use that prefix in the DTD. However, defining the namespace prefix and the colon that separates the prefix from the local name as parameter entities (as follows) can make the job easier.

```
<!ENTITY % dc-prefix "dc">
<!ENTITY % dc-colon ":">
```

In this way, we can define the qualified names as more parameter entity references, like these:

```
<!ENTITY % dc-title      "%dc-prefix;%dc-colon;title">
<!ENTITY % dc-creator    “%dc-prefix;%dc-colon;creator">
```

Then you use the entity references for the qualified name in all declarations, like this:

```
<!ELEMENT %dc-title; (#PCDATA)>
<!ELEMENT %dc-creator; (#PCDATA)>
```

Now a document that needs to change the prefix simply changes the parameter entity definitions.

9. XPath

XPath is a non-XML language for identifying particular parts of XML documents, which can be selected by position, relative position, type, content, and several other criteria. The

W3C XML Schema Language uses XPath expressions to define uniqueness and identity constraints.

9.1 The structure of an XML Document

An XML document is a tree made up of nodes, which can be of seven kinds:

- The root node
- Element nodes
- Text nodes
- Attribute nodes
- Comment nodes
- Processing-instruction nodes
- Namespace nodes

One thing to note are the constructs **not included** in this list: CDATA sections, entity references, and document type declarations.

The XPath data model has several nonobvious features. First of all, the root node of the tree **is not the same as the root element**. The root node of the tree contains the entire document including the root element, as well as any comments and processing instructions that occur before the root element start-tag or after the root element end-tag.

Moreover, the XPath data model **does not include everything** in the document. In particular, the XML declaration, the DOCTYPE declaration, and the various parts of the DTD are not addressable via XPath.

9.2 Location Paths

The most useful XPath expression is a location path. A location path identifies a set of nodes in a document. This set may be empty, may contain a single node, or may contain several nodes. A location path is built out of successive **location steps**. Each location step is evaluated relative to a particular node in the document called the **context node**.

- The simplest location path is the one that selects the **root node** of the document, identified by the forward slash "/". The forward slash is also an absolute location path, because it always means the same thing (the root node), no matter what the context node is.
- The second simplest location path is a **single element name**. This path selects all child elements of the context node with the specified name. This path selects all child elements of the context node with the specified name. Exactly which elements these are depends on what the context node is.
- To select a particular **attribute of an element**, use an @ sign followed by the name of the attribute you want.
- The other three node types have special node tests to match them. These are as follows:
  - comment( )
  - text( )
  - processing-instruction( )

Wildcards match different element and node types at the same time. There are three wildcards: *, node( ) , and @*.

- The asterisk (*) matches any element node regardless of name.

- The node( ) wildcard matches not only all element types but also the root node, text nodes, processing-instruction nodes, namespace nodes, attribute nodes, and comment nodes.
- The @* wildcard matches all attribute nodes.

You often want to match more than one type of element or attribute but not all types. You can combine location paths and steps with the vertical bar (|) to indicate that you want to match any of the named elements.

9.3 Compound Location Paths

Location steps can be combined with a forward slash (/) to make a compound location path: `/people/person/name/first_name/text( )`.

A double forward slash (//) selects from all descendants of the context node, as well as the context node itself.

A double period (..) indicates the parent of the current node.

Finally, the single period (.) indicates the context node.

9.4 Predicates

Each step in a location path may (but does not have to) have a predicate that selects from the node-set current at that step in the expression. The predicate contains a Boolean expression, which is tested for each node in the context node list. If the expression is false, then that node is deleted from the list. Otherwise, it's retained. For example, suppose you want to find all `profession` elements whose value is "physicist". The XPath expression `//profession[. = "physicist"]` does this.

In some cases, the predicate may not be a Boolean, but it can be converted to one in a straightforward fashion. Predicates that evaluate to numbers are true if they're equal to the position of the context node; otherwise, they're false. For instance, `//name[2]` selects the second `name` element in the document.

9.5 Unabbreviated Location Paths

Up until this point, we've been using what are called abbreviated location paths. However, XPath also offers an unabbreviated syntax for location paths.

Exemples (abbreviated form => unabbreviated form):

- people/person/@id => child::people/child::person/attribute::id
- /people/person[@born<1950]/name[first_name="Alan"]
  =>
  /child::people/child::person[attribute::born < 1950] /
  child::name[child::first_name = "Alan"]

The term preceding the :: specifies the "axis" along with select the given element (specified after the ::).

The unabbreviated syntax adds eight more axes:

- ancestor
- following-sibling
- preceding-sibling

- following
- preceding
- namespace
- descendant
- ancestor-or-self

9.6 General XPath Expression

Location paths are not the only possible type of XPath expression. XPath expressions can also return numbers, Booleans, and strings.

- Numbers: All numbers are 8-byte, IEEE 754 floating-point doubles. No pure integers. Five basic operators (+, -, *, div, mod)
- Strings: sequence of characters. They may be enclosed in either single or double quotes. The only restriction XPath places on a string literal is that it must not contain the kind of quote that delimits it. You can use the = and != comparison operators to check whether two strings are the same.
- Booleans: XPath does not provide any Boolean literals. However, the XPath functions true() and false( ) can substitute for the missing literals quite easily. XPath provides also the and, or, not() operators.

9.7 XPath Functions

XPath provides many functions that you may find useful in predicates or raw expressions. Each XPath function returns one of these four types: Boolean, Number, Node-set, String. There are no void functions.

- **Node-Set Functions**: operate on or return information about node-sets. Examples are: count( ), last( ), position( ), id( ), local-name( ), namespace-uri(), name( ).
- **String Functions**: includes functions for basic string operations such as finding the length of a string or changing letters from upper- to lowercase. Examples: string( ), starts-with( ), string-length( ), substring(), substring-before( ).
- **Boolean Funtions**: are few in number and quite straightforward.
- **Number Functions**: XPath includes a few simple numeric functions for summing groups of numbers and finding the nearest integer to a number. Examples: number( ), round( ) , floor( ), ceiling( ), sum().