A Technical Introduction to XML

By Norman Walsh

This introduction to XML presents the Extensible Markup Language at a reasonably technical level for anyone interested in learning more about structured documents. In addition to covering the XML 1.0 Specification, this article outlines related XML specifications, which are evolving. The article is organized in four main sections plus an appendix.

Start Here

Author's Note

It is somewhat remarkable to think that this article, which appeared initially in the Winter 1997 edition of the World Wide Web Journal was out of date by the time the final XML Recommendation was approved in February. And even as this update brings the article back into line with the final spec, a new series of recommendations are under development. When finished, these will bring namespaces, linking, schemas, stylesheets, and more to the table.

What is XML?

XML is a markup language for documents containing structured information.

Structured information contains both content (words, pictures, etc.) and some indication of what role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.). Almost all documents have some structure.

A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents.

What's a Document?

The number of applications currently being developed that are based on, or make use of, XML documents is truly amazing (particularly when you consider that XML is not yet a year old)! For our purposes, the word "document" refers not only to traditional documents, like this one, but also to the miriad of other XML "data formats". These include vector graphics, e-commerce transactions, mathematical equations, object meta-data, server APIs, and a thousand other kinds of structured information.

So XML is Just Like HTML?

No. In HTML, both the tag semantics and the tag set are fixed. An `<h1>` is always a first level heading and the tag `<ati.product.code>` is meaningless. The W3C, in conjunction with browser vendors and the WWW community, is constantly working to extend the definition of HTML to allow new tags to keep pace with changing technology and to bring variations in presentation (stylesheets) to the Web. However, these changes are always rigidly confined by what the browser vendors have implemented and by the fact that backward compatibility is paramount. And for people who want to disseminate information widely, features supported by only the latest releases of Netscape and Internet Explorer are not useful.

XML specifies neither semantics nor a tag set. In fact XML is really a meta-language for describing markup languages. In other words, XML provides a facility to define tags and the structural relationships between them. Since there's no predefined tag set, there can't be any preconceived semantics. All of the semantics of an XML document will either be defined by the applications that process them or by stylesheets.

So XML Is Just Like SGML?

No. Well, yes, sort of. XML is defined as an application profile of SGML. SGML is the Standard Generalized Markup Language defined by ISO 8879. SGML has been the standard, vendor-independent way to maintain repositories of structured documentation for more than a decade, but it is not well suited to serving documents over the web (for a number of technical reasons beyond the scope of this article). Defining XML as an application profile of SGML means that any fully conformant SGML system will be able to read XML documents. However, using and understanding XML documents *does not* require a system that is capable of understanding the full generality of SGML. XML is, roughly speaking, a restricted form of SGML.

For technical purists, it's important to note that there may also be subtle differences between documents as understood by XML systems and those same documents as understood by SGML systems. In particular, treatment of white space immediately adjacent to tags may be different.

Why XML?

In order to appreciate XML, it is important to understand why it was created. XML was created so that richly structured documents could be used over the web. The only viable alternatives, HTML and SGML, are not practical for this purpose.

HTML, as we've already discussed, comes bound with a set of semantics and does not provide arbitrary structure.

SGML provides arbitrary structure, but is too difficult to implement just for a web browser. Full SGML systems solve large, complex problems that justify their expense. Viewing structured documents sent over the web rarely carries such justification.

This is not to say that XML can be expected to completely replace SGML. While XML is being designed to deliver structured content over the web, some of the very features it lacks to make this practical, make SGML a more satisfactory solution for the creation and long-time storage of complex documents. In many organizations, filtering SGML to XML will be the standard procedure for web delivery.

XML Development Goals

The XML specification sets out the following goals for XML: [Section 1.1] (In this article, citations of the form [Section 1.1], these are references to the W3C Recommendation Extensible Markup Language (XML) 1.0. If you are interested in more technical detail about a particular topic, please consult the specification)

1. It shall be straightforward to use XML over the Internet. Users must be able to view XML documents as quickly and easily as HTML documents. In practice, this will only be possible when XML browsers are as robust and widely available as HTML browsers, but the principle remains.
2. XML shall support a wide variety of applications. XML should be beneficial to a wide variety of diverse applications: authoring, browsing, content analysis, etc. Although the initial focus is on serving structured documents over the web, it is not meant to narrowly define XML.
3. XML shall be compatible with SGML. Most of the people involved in the XML effort come from organizations that have a large, in some cases staggering, amount of material in SGML. XML was designed pragmatically, to be compatible with existing standards while solving the relatively new problem of sending richly structured documents over the web.
4. It shall be easy to write programs that process XML documents. The colloquial way of expressing this goal while the spec was being developed was that it ought to take about two weeks for a competent computer science graduate student to build a program that can process XML documents.
5. The number of optional features in XML is to be kept to an absolute minimum, ideally zero. Optional features inevitably raise compatibility problems when

users want to share documents and sometimes lead to confusion and frustration.

6. XML documents should be human-legible and reasonably clear. If you don't have an XML browser and you've received a hunk of XML from somewhere, you ought to be able to look at it in your favorite text editor and actually figure out what the content means.
7. The XML design should be prepared quickly. Standards efforts are notoriously slow. XML was needed immediately and was developed as quickly as possible.
8. The design of XML shall be formal and concise. In many ways a corollary to rule 4, it essentially means that XML must be expressed in EBNF and must be amenable to modern compiler tools and techniques.

   There are a number of technical reasons why the SGML grammar *cannot* be expressed in EBNF. Writing a proper SGML parser requires handling a variety of rarely used and difficult to parse language features. XML does not.
9. XML documents shall be easy to create. Although there will eventually be sophisticated editors to create and edit XML content, they won't appear immediately. In the interim, it must be possible to create XML documents in other ways: directly in a text editor, with simple shell and Perl scripts, etc.
10. Terseness in XML markup is of minimal importance. Several SGML language features were designed to minimize the amount of typing required to manually key in SGML documents. These features are not supported in XML. From an abstract point of view, these documents are indistinguishable from their more fully specified forms, but supporting these features adds a considerable burden to the SGML parser (or the person writing it, anyway). In addition, most modern editors offer better facilities to define shortcuts when entering text.

How Is XML Defined?

XML is defined by a number of related specifications:

Extensible Markup Language (XML) 1.0

Defines the syntax of XML. The XML specification is the primary focus of this article.

XML Pointer Language (XPointer) and **XML Linking Language (XLink)**

Defines a standard way to represent links between resources. In addition to simple links, like HTML's `<A>` tag, XML has mechanisms for links between multiple resources and links between read-only resources. XPointer describes how to address a resource, XLink describes how to associate two or more resources.

Extensible Style Language (XSL)

Defines the standard stylesheet language for XML.

As time goes on, additional requirements will be addressed by other specifications. Currently (Sep, 1998), namespaces (dealing with tags from multiple tag sets), a query language (finding out what's in a document or a collection of documents), and a schema language (describing the relationships between tags, DTDs in XML) are all being actively pursued.

Understanding the Specs

For the most part, reading and understanding the XML specifications does not require extensive knowledge of SGML or any of the related technologies.

One topic that may be new is the use of EBNF to describe the syntax of XML. Please consult the discussion of EBNF in the appendix of this article for a detailed description of how this grammar works.

What Do XML Documents Look Like?

If you are conversant with HTML or SGML, XML documents will look familiar. A simple XML document is presented in Example 1.

Example 1. A Simple XML Document

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

<oldjoke>

<burns>Say <quote>goodnight</quote>,
Gracie.</burns>

<allen><quote>Goodnight,
Gracie.</quote></allen>

<applause/>

</oldjoke>
```

A few things may stand out to you:

- The document begins with a processing instruction: `<?xml ...?>`. This is the *XML declaration* [Section 2.8]. While it is not required, its presence explicitly identifies the document as an XML document and indicates the version of XML to which it was authored.
- There's no document type declaration. Unlike SGML, XML does not require a document type declaration. However, a document type declaration can be supplied, and some documents will require one in order to be understood unambiguously.
- Empty elements (`<applause/>` in this example) have a modified syntax [Section 3.1]. While most elements in a document are wrappers around some content, empty elements are simply markers where something occurs (a horizontal rule for HTML's `<hr>` tag, for example, or a cross reference for DocBook's `<xref>` tag). The trailing /> in the modified syntax indicates to a program processing the XML document that the element is empty and no matching end-tag should be sought. Since XML documents do not require a document type declaration, without this clue it could be impossible for an XML parser to determine which tags were intentionally empty and which had been left empty by mistake.
  XML has softened the distinction between elements which are declared as `EMPTY` and elements which merely have no content. In XML, it is legal to use the empty-element tag syntax in either case. It's also legal to use a start-tag/end-tag pair for empty elements: `<applause></applause>`. If interoperability is of any concern, it's best to reserve empty-element tag syntax for elements which are

declared as `EMPTY` and to only use the empty-element tag form for those elements.

XML documents are composed of markup and content. There are six kinds of markup that can occur in an XML document: elements, entity references, comments, processing instructions, marked sections, and document type declarations. The following sections introduce each of these markup concepts.

Elements

Elements are the most common form of markup. Delimited by angle brackets, most elements identify the nature of the content they surround. Some elements may be empty, as seen above, in which case they have no content. If an element is not empty, it begins with a start-tag, `<element>`, and ends with an end-tag, `</element>`.

Attributes

Attributes are name-value pairs that occur inside start-tags after the element name. For example,

```
<div class="preface">
```

is a `div` element with the attribute `class` having the value `preface`. In XML, all attribute values must be quoted.

Entity References

In order to introduce markup into a document, some characters have been reserved to identify the start of markup. The left angle bracket, < , for instance, identifies the beginning of an element start- or end-tag. In order to insert these characters into your document as content, there must be an alternative way to represent them. In XML, entities are used to represent these special characters. Entities are also used to refer to often repeated or varying text and to include the content of external files.

Every entity must have a unique name. Defining your own entity names is discussed in the section on entity declarations. In order to use an entity, you simply reference it by name. Entity references begin with the ampersand and end with a semicolon.

For example, the `lt` entity inserts a literal < into a document. So the string `<element>` can be represented in an XML document as `&lt;element>`.

A special form of entity reference, called a character reference [Section 4.1], can be used to insert arbitrary Unicode characters into your document. This is a mechanism for inserting characters that cannot be typed directly on your keyboard.

Character references take one of two forms: decimal references, `&#8478;`, and hexadecimal references, `&#x211E;`. Both of these refer to character number U+211E from Unicode (which is the standard Rx prescription symbol, in case you were wondering).

Comments

Comments begin with `<!--` and end with `-->`. Comments can contain any data except the literal string `--`. You can place comments between markup anywhere in your document.

Comments are not part of the textual content of an XML document. An XML processor is not required to pass them along to an application.

Processing Instructions

Processing instructions (PIs) are an escape hatch to provide information to an application. Like comments, they are not textually part of the XML document, but the XML processor is required to pass them to an application.

Processing instructions have the form: `<?name pidata?>`. The name, called the PI target, identifies the PI to the application. Applications should process only the targets they recognize and ignore all other PIs. Any data that follows the PI target is optional, it is for the application that recognizes the target. The names used in PIs may be declared as notations in order to formally identify them.

PI names beginning with `xml` are reserved for XML standardization.

CDATA Sections

In a document, a `CDATA` section instructs the parser to ignore most markup characters.

Consider a source code listing in an XML document. It might contain characters that the XML parser would ordinarily recognize as markup (`<` and `&`, for example). In order to prevent this, a CDATA section can be used.

```
<![CDATA[

*p = &q;

b = (i <= 3);

]]>
```

Between the start of the section, `<![CDATA[` and the end of the section, `]]>`, all character data is passed directly to the application, without interpretation. Elements, entity references, comments, and processing instructions are all unrecognized and the characters that comprise them are passed literally to the application.

The only string that cannot occur in a CDATA section is `]]>`.

Document Type Declarations

A large percentage of the XML specification deals with various sorts of declarations that are allowed in XML. If you have experience with SGML, you will recognize these declarations from SGML DTDs (Document Type Definitions). If you have never seen them before, their significance may not be immediately obvious.

One of the greatest strengths of XML is that it allows you to create your own tag names. But for any given application, it is probably not meaningful for tags to occur in a completely arbitrary order. Consider the old joke example introduced earlier. Would this be meaningful?

```
<gracie><quote><oldjoke>Goodnight,
<applause/>Gracie</oldjoke></quote>

<burns><gracie>Say <quote>goodnight</quote>,
</gracie>Gracie.</burns></gracie>
```

It's so far outside the bounds of what we normally expect that it's nonsensical. It just doesn't *mean* anything.

However, from a strictly syntactic point of view, there's nothing wrong with that XML document. So, if the document is to have meaning, and certainly if you're writing a stylesheet or application to process it, there must be some constraint on the sequence and nesting of tags. Declarations are where these constraints can be expressed.

More generally, declarations allow a document to communicate meta-information to the parser about its content. Meta-information includes the allowed sequence and nesting of tags, attribute values and their types and defaults, the names of external files that may be referenced and whether or not they contain XML, the formats of some external (non-XML) data that may be referenced, and the entities that may be encountered.

There are four kinds of declarations in XML: element type declarations, attribute list declarations, entity declarations, and notation declarations.

Element Type Declarations

Element type declarations [Section 3.2] identify the names of elements and the nature of their content. A typical element type declaration looks like this:

```
<!ELEMENT oldjoke (burns+, allen, applause?)>
```

This declaration identifies the element named `oldjoke`. Its *content model* follows the element name. The content model defines what an element may contain. In this case, an `oldjoke` must contain `burns` and `allen` and may contain `applause`. The commas between element names indicate that they must occur in succession. The plus after `burns` indicates that it may be repeated more than once but must occur at least once. The question mark after `applause` indicates that it is optional (it may be absent, or it may occur exactly once). A name with no punctuation, such as `allen`, must occur exactly once.

Declarations for `burns`, `allen`, `applause` and all other elements used in any content model must also be present for an XML processor to check the validity of a document.

In addition to element names, the special symbol #PCDATA is reserved to indicate character data. The moniker PCDATA stands for parseable character data .

Elements that contain only other elements are said to have *element content* [Section 3.2.1]. Elements that contain both other elements and #PCDATA are said to have *mixed content* [Section 3.2.2].

For example, the definition for burns might be

```
<!ELEMENT burns (#PCDATA | quote)*>
```

The vertical bar indicates an or relationship, the asterisk indicates that the content is optional (may occur zero or more times); therefore, by this definition, burns may contain zero or more characters and quote tags, mixed in any order. All mixed content models must have this form: #PCDATA must come first, all of the elements must be separated by vertical bars, and the entire group must be optional.

Two other content models are possible: EMPTY indicates that the element has no content (and consequently no end-tag), and ANY indicates that *any* content is allowed. The ANY content model is sometimes useful during document conversion, but should be avoided at almost any cost in a production environment because it disables all content checking in that element.

Here is a complete set of element declarations for Example 1:

Example 2. Element Declarations for Old Jokes

```
<!ELEMENT oldjoke  (burns+, allen, applause?)>

<!ELEMENT burns    (#PCDATA | quote)*>

<!ELEMENT allen    (#PCDATA | quote)*>

<!ELEMENT quote    (#PCDATA)*>

<!ELEMENT applause EMPTY>
```

Attribute List Declarations

Attribute list declarations [Section 3.3] identify which elements may have attributes, what attributes they may have, what values the attributes may hold, and what value is the default. A typical attribute list declaration looks like this:

```
<!ATTLIST oldjoke

    name
ID
#REQUIRED

    label
CDATA
#IMPLIED

    status ( funny | notfunny ) 'funny'>
```

In this example, the oldjoke element has three attributes: name, which is an ID and is required; `label`, which is a string (character data) and is not required; and `status`, which must be either `funny` or `notfunny` and defaults to `funny`, if no value is specified.

Each attribute in a declaration has three parts: a name, a type, and a default value.

You are free to select any name you wish, subject to some slight restrictions [Section 2.3, production 5], but names cannot be repeated on the same element.

There are six possible attribute types:

`CDATA`

> CDATA attributes are strings, any text is allowed. Don't confuse CDATA attributes with CDATA sections, they are unrelated.

`ID`

> The value of an ID attribute must be a name [Section 2.3, production 5]. All of the ID values used in a document must be different. IDs uniquely identify individual elements in a document. Elements can have only a single ID attribute.

`IDREF`
or `IDREFS`

> An IDREF attribute's value must be the value of a single ID attribute on some element in the document. The value of an IDREFS attribute may contain multiple IDREF values separated by white space [Section 2.3, production 3].

`ENTITY`
or `ENTITIES`

> An ENTITY attribute's value must be the name of a single entity (see the discussion of entity declarations below). The value of an ENTITIES attribute may contain multiple entity names separated by white space.

`NMTOKEN`
or `NMTOKENS`

> Name token attributes are a restricted form of string attribute. In general, an NMTOKEN attribute must consist of a single word [Section 2.3, production 7], but there are no additional constraints on the word, it doesn't have to match another attribute or declaration. The value of an NMTOKENS attribute may contain multiple NMTOKEN values separated by white space.

A list of names

> You can specify that the value of an attribute must be taken from a specific list of names. This is frequently called an enumerated type because each of the possible values is explicitly enumerated in the declaration.
>
> Alternatively, you can specify that the names must match a notation name (see the discussion of notation declarations below).

There are four possible default values:

`#REQUIRED`

The attribute must have an explicitly specified value on every occurrence of the element in the document.

`#IMPLIED`

The attribute value is not required, and no default value is provided. If a value is not specified, the XML processor must proceed without one.

"value"

An attribute can be given any legal value as a default. The attribute value is not required on each element in the document, and if it is not present, it will appear to be the specified default.

`#FIXED`
"value"

An attribute declaration may specify that an attribute has a fixed value. In this case, the attribute is not required, but if it occurs, it must have the specified value. If it is not present, it will appear to be the specified default. One use for fixed attributes is to associate semantics with an element. A complete discussion is beyond the scope of this article, but you can find several examples of fixed attributes in the XLink specification.

The XML processer performs *attribute value normalization* [Section 3.3.3] on attribute values: character references are replaced by the referenced character, entity references are resolved (recursively), and whitespace is normalized.

Entity Declarations

Entity declarations [Section 4.2] allow you to associate a name with some other fragment of content. That construct can be a chunk of regular text, a chunk of the document type declaration, or a reference to an external file containing either text or binary data.

A few typical entity declarations are shown in Example 3.

Example 3. Typical Entity Declarations

```
<!ENTITY
ATI
"ArborText, Inc.">

<!ENTITY boilerplate     SYSTEM
"/standard/legalnotice.xml">

<!ENTITY ATIlogo
SYSTEM "/standard/logo.gif" NDATA GIF87A>
```

There are three kinds of entities:

Internal Entities

Internal entities [Section 4.2.1] associate a name with a string of literal text. The first entity in Example 3 is an internal entity. Using `&ATI;` anywhere in the document will insert ArborText, Inc. at that location. Internal entities allow you to define shortcuts for frequently typed text or text that is expected to change, such as the revision status of a document.

Internal entities can include references to other internal entities, but it is an error for them to be recursive.

The XML specification predefines five internal entities:

- `&lt;` produces the left angle bracket, `<`
- `&gt;` produces the right angle bracket, `>`
- `&amp;` produces the ampersand, `&`
- `&apos;` produces a single quote character (an apostrophe), `'`
- `&quot;` produces a double quote character, `"`

External Entities

External entities [Section 4.2.2] associate a name with the content of another file. External entities allow an XML document to refer to the contents of another file. External entities contain either text or binary data. If they contain text, the content of the external file is inserted at the point of reference and parsed as part of the referring document. Binary data is not parsed and may only be referenced in an attribute. Binary data is used to reference figures and other non-XML content in the document.

The second and third entities in Example 3 are external entities.

Using `&boilerplate;` will have insert the *contents* of the file `/standard/legalnotice.xml` at the location of the entity reference. The XML processor will parse the content of that file as if it occurred literally at that location.

The entity `ATIlogo` is also an external entity, but its content is binary. The `ATIlogo` entity can only be used as the value of an ENTITY (or ENTITIES) attribute (on a `graphic` element, perhaps). The XML processor will pass this information along to an application, but it does not attempt to process the content of `/standard/logo.gif`.

Parameter Entities

Parameter entities can only occur in the document type declaration. A parameter entity declaration is identified by placing `%` (percent-space) in front of its name in the declaration. The percent sign is also used in references to parameter entities, instead of the ampersand. Parameter entity references are immediately expanded in the document type declaration and their replacement text is part of the declaration, whereas normal entity references are not expanded. Parameter entities are not recognized in the body of a document.

Looking back at the element declarations in Example 2, you'll notice that two of them have the same content model:

```
<!ELEMENT burns    (#PCDATA | quote)*>

<!ELEMENT allen    (#PCDATA | quote)*>
```

At the moment, these two elements are the same only because they happen to have the same literal definition. In order to make more explicit the fact that these two elements are semantically the same, use a parameter entity to define their content model. The advantage of using a parameter entity is two-fold. First, it allows you to give a descriptive name to the content, and second it allows you to change the content model in only a single place, if you wish to update the element declarations, assuring that they always stay the same:

```
<!ENTITY % personcontent "#PCDATA | quote">

<!ELEMENT burns (%personcontent;)*>

<!ELEMENT allen (%personcontent;)*>
```

Notation Declarations

Notation declarations [Section 4.7] identify specific types of external binary data. This information is passed to the processing application, which may make whatever use of it it wishes. A typical notation declaration is:

```
<!NOTATION GIF87A SYSTEM "GIF">
```

Do I need a Document Type Declaration?

As we've seen, XML content can be processed without a document type declaration. However, there are some instances where the declaration is required:

Authoring Environments
: Most authoring environments need to read and process document type declarations in order to understand and enforce the content models of the document.

Default Attribute Values
: If an XML document relies on default attribute values, at least part of the declaration must be processed in order to obtain the correct default values.

White Space Handling
: The semantics associated with white space in element content differs from the semantics associated with white space in mixed content. Without a DTD, there is no way for the processor to distinguish between these cases, and all elements are effectively mixed content. For more detail, see the section called *White Space Handling*, later in this document.

In applications where a person composes or edits the data (as opposed to data that may be generated directly from a database, for example), a DTD is probably going to be required if any structure is to be guaranteed.

Including a Document Type Declaration

If present, the document type declaration must be the first thing in the document after optional processing instructions and comments [Section 2.8].

The document type declaration identifies the root element of the document and may contain additional declarations. All XML documents must have a single root element that contains all of the content of the document. Additional declarations may come from an external DTD, called the external subset, or be included directly in the document, the internal subset, or both:

```
<?XML version="1.0" standalone="no"?>

<!DOCTYPE chapter SYSTEM "dbook.dtd" [

<!ENTITY %ulink.module "IGNORE">

<!ELEMENT ulink (#PCDATA)*>

<!ATTLIST ulink

    xml:link       CDATA  #FIXED "SIMPLE"

    xml-attributes CDATA  #FIXED "HREF URL"

    URL            CDATA  #REQUIRED>

]>

<chapter>...</chapter>
```

This example references an external DTD, `dbook.dtd`, and includes element and attribute declarations for the `ulink` element in the internal subset. In this case, `ulink` is being given the semantics of a simple link from the XLink specification.

Note that declarations in the internal subset override declarations in the external subset. The XML processor reads the internal subset before the external subset and the *first* declaration takes precedence.

In order to determine if a document is valid, the XML processor must read the entire document type declaration (both internal and external subsets). But for some applications, validity may not be required, and it may be sufficient for the processor to read only the internal subset. In the example above, if validity is unimportant and the only reason to read the doctype declaration is to identify the semantics of `ulink`, reading the external subset is not necessary.

You can communicate this information in the *standalone document declaration* [Section 2.9]. The standalone document declaration, `standalone="yes"` or `standalone="no"` occurs in the XML declaration. A value of `yes` indicates that only internal declarations need to be processed. A value of `no` indicates that *both* the internal and external declarations must be processed.

Other Markup Issues

In addition to markup, there are a few other issues to consider: white space handling, attribute value normalization, and the language in which the document is written.

White Space Handling

White space handling [Section 2.10] is a subtle issue. Consider the following content fragment:

```
<oldjoke>

<burns>Say <quote>goodnight</quote>, Gracie.</burns>
```

Is the white space (the new line between `<oldjoke>` and `<burns>`) significant?

Probably not.

But how can you tell? You can only determine if white space is significant if you know the content model of the elements in question. In a nutshell, white space is significant in mixed content and is insignificant in element content.

The rule for XML processors is that they must pass all characters that are not markup through to the application. If the processor is a validating processor [Section 5.1], it must also inform the application about which whitespace characters are significant.

The special attribute `xml:space` may be used to indicate explicitly that white space is significant. On any element which includes the attribute specification `xml:space='preserve'`, all white space within that element (and within subelements that do not explicitly reset `xml:space`) is significant.

The only legal values for `xml:space` are `preserve` and `default`. The value `default` indicates that the default processing is desired. In a DTD, the `xml:space` attribute must be declared as an enumerated type with only those two values.

One last note about white space: in parsed text, XML processors are required to normalize all end-of-line markers to a single line feed character (&#A;) [Section 2.11]. This is rarely of interest to document authors, but it does eliminate a number of cross-platform portability issues.

Attribute Value Normalization

The XML processer performs attribute value normalization [Section 3.3.3] on attribute values: character references are replaced by the referenced character, entity references are resolved (recursively), and whitespace is normalized.

Language Identification

Many document processing applications can benefit from information about the natural language in which a document is written, XML defines the attribute `xml:lang` [Section 2.12] to identify the language. Since the purpose of this attribute is to standardize information across applications, the XML specification also describes how languages are to be identified.

Validity

Given the preceding discussion of type declarations, it follows that some documents are valid and some are not. There are two categories of XML documents: well-formed and valid.

Well-formed Documents

A document can only be well-formed [Section 2.1] if it obeys the syntax of XML. A document that includes sequences of markup characters that cannot be parsed or are invalid cannot be well-formed.

In addition, the document must meet all of the following conditions (understanding some of these conditions may require experience with SGML):

- The document instance must conform to the grammar of XML documents. In particular, some markup constructs (parameter entity references, for example) are only allowed in specific places. The document is not well-formed if they occur elsewhere, even if the document is well-formed in all other ways.
- The replacement text for all parameter entities referenced inside a markup declaration consists of zero or more complete markup declarations. (No parameter entity used in the document may consist of only part of a markup declaration.)
- No attribute may appear more than once on the same start-tag.
- String attribute values cannot contain references to external entities.
- Non-empty tags must be properly nested.
- Parameter entities must be declared before they are used.
- All entities except the following: `amp`, `lt`, `gt`, `apos`, and `quot` must be declared.
- A binary entity cannot be referenced in the flow of content, it can only be used in an attribute declared as `ENTITY` or `ENTITIES`.
- Neither text nor parameter entities are allowed to be recursive, directly or indirectly.

By definition, if a document is not well-formed, it is not XML. This means that there is no such thing as an XML document which is not well-formed, and XML processors are not required to do anything with such documents.

Valid Documents

A well-formed document is valid only if it contains a proper document type declaration and if the document obeys the constraints of that declaration (element sequence and nesting is valid, required attributes are provided, attribute values are of the correct type, etc.). The XML specification identifies all of the criteria in detail.

Pulling the Pieces Together

The XPointer and XLink specifications, currently under development, introduce a standard linking model for XML. In consideration of space, and the fact that the XLink draft is still developing, what follows is survey of the features of XLink, rather than a detailed description of the specification.

In the parlance of XLink, a link expresses a relationship between resources. A resource is any location (an element, or its content, or some part of its content, for example) that is addressed in a link. The exact nature of the relationship between resources depends on both the application that processes the link and semantic information supplied.

Some highlights of XLink are:

- XLink gives you control over the semantics of the link.
- XLink introduces Extended Links. Extended Links can involve more than two resources.
- XPointer introduces Extended Pointers (XPointers). XPointers provide a sophisticated method of locating resources. In particular, XPointers allow you to locate arbitrary resources in a document, without requiring that the resource be identified with an ID attribute.

Since XML does not have a fixed set of elements, the name of the linking element cannot be used to locate links. Instead, XML processors identify links by recognizing the `xml:link` attribute. Other attributes can be used to provide additional information to the XML processor. An attribute renaming facility exists to work around name collisions in existing applications.

Two of the attributes, `show` and `actuate` allow you to exert some control over the linking behavior. The `show` attribute determines whether the document linked-to is embeded in the current document, replaces the current document, or is displayed in a new window when the link is traversed. `actuate` determines how the link is traversed, either automatically or when selected by the user.

Some applications will require much finer control over linking behaviors. For those applications, standard places are provided where the additional semantics may be expressed.

Simple Links

A Simple Link strongly resembles an HTML `<A>` link:

```
<link xml:link="simple" href="locator">Link Text</link>
```

A Simple Link identifies a link between two resources, one of which is the content of the linking element itself. This is an in-line link.

The *locator* identifies the other resource. The locator may be a URL, a query, or an Extended Pointer.

Extended Links

Extended Links allow you to express relationships between more than two resources:

```
<elink xml:link="extended" role="annotation">

<locator xml:link="locator" href="text.loc">The Text</locator>

<locator xml:link="locator" href="annot1.loc">Annotations
        </locator>

<locator xml:link="locator" href="annot2.loc">
        More Annotations</locator>
```

```
<locator xml:link="locator" href="litcrit.loc">
        Literary Criticism</locator>

</elink>
```

This example shows how the relationships between a literary work, annotations, and literary criticism of that work might be expressed. Note that this link is separate from all of the resources involved.

Extended Links can be in-line, so that the content of the linking element (other than the locator elements), participates in the link as a resource, but that is not necessarily the case. The example above is an out-of-line link because it does not use its content as a resource.

Extended Pointers

Cross references with the XML ID/IDREF mechanism (which is similar to the `#fragment` mechanism in HTML) require that the document being linked-to has defined anchors where links are desired (and technically requires that both the ID and the IDREF occur in the same document). This may not always be the case and sometimes it is not possible to modify the document to which you wish to link.

XML XPointers borrow concepts from HyTime and the Text Encoding Initiative (TEI). XPointers offer a syntax that allows you to locate a resource by traversing the element tree of the document containing the resource.

For example,

```
child(2,oldjoke).(3,.)
```

locates the third child (whatever it may be) of the second `oldjoke` in the document.

XPointers can span regions of the tree. The XPointer

```
span(child(2,oldjoke),child(3,oldjoke))
```

selects the second and third `oldjoke` s in the document.

In addition to selecting by elements, XPointers allow for selection by ID, attribute value, and string matching. In this article, the XPointer

```
span(root()child(3,sect1)string(1,"Here",0),
        root()child(3,sect1)string(1,"Here",4))
```

selects the first occurance of the word "Here" in the What Do XML Documents Look Like? section of this article. The link can be established by an extended link *without modifying* the target document.

Note that an XPointer range can span a structurally invalid section of the document. The XLink specification does not specify how applications should deal with such ranges.

Extended Link Groups

Out-of-line links introduce the possibility that an XML processor may need to process several files in order to correctly display the hypertext document.

Following the annotated text example above, assuming that the actual text is read only, the XML processor must load at least the text and the document that contains the extended link.

XLink defines Extended Link Groups for this purpose. Loading an Extended Link Group communicates which documents must be loaded to the XML processor. Extended Link Groups can be used recursively, and a `steps` attribute is provided to limit the depth of recursion.

Understanding The Pieces

Some documents, particularly compound documents pulled together with XLinks, are likely to be composed of elements from multiple tag sets. For example, a technical article might be written using one DTD, but include mathematical equations written in MathML and vector graphics written in a third DTD.

In order for a processing application to associate the correct semantics with an element, it must know which tag set the element comes from. XML solves this problem with namespaces. Namespaces in XML describes this system in detail.

The principle is to allow a colon-delimited prefix to be associated with some external semantic via a URI. Then use of that prefix identifies the element as having the semantics described by the URI. For example:

```
<bk:para>The fraction 3/4 can be expressed in MathML as:

<ml:cn type="rational">3<ml:sep/>4</ml:cn>.</bk:para>
```

The `para` element in this example is explicitly identified as being in the namespace identified by the `bk` prefix, which must have been defined earlier in the document, and the `cn` and `sep` elements come from the `ml` namespace (presumably associated in some way with MathML).

Style and Substance

HTML browsers are largely hard-coded. Although some browsers can base their formatting on Cascading Style Sheets (CSS), they still contain hard-coded conventions for documents which do not provide a stylesheet. A first level heading appears the way that it does largely because the browser recognizes the `<h1>` tag.

Again, since XML documents have no fixed tag set, this approach will not work. The presentation of an XML document is *dependent* on a stylesheet.

The standard stylesheet language for XML documents is the Extensible Style Language (XSL). At the time of this writing, the XSL effort is well underway, but many questions remain unanswered. The XSL Working Group produced its first Working Draft on 18 Aug 1998.

Other stylesheet languages, like Cascading Style Sheets, are likely to be supported as well.

Conclusion

In this article, most of the major features of the XML Language have been discussed, and some of the concepts behind XLink, Namespaces, and XSL have been described. Although some things have been left out in the interest of the big picture (such as character encoding issues), hopefully you now have enough background to pick up and read the XML Specifications without difficulty.

Appendix: Extended Backus-Naur Form (EBNF)

One of the most significant design improvements in XML is to make it easy to use with modern compiler tools. Part of this improvement involves making it possible to express the syntax of XML in Extended Backus-Naur Form (EBNF) [Section 6]. If you've never seen EBNF before, think of it this way:

- EBNF is a set of rules, called productions
- Every rule describes a specific fragment of syntax
- A document is valid if it can be reduced to a single, specific rule, with no input left, by repeated application of the rules.

Let's take a simple example that has nothing to do with XML (or the real rules of language):

```
[1] Word      ::= Consonant Vowel+ Consonant

[2] Consonant ::= [^aeiou]

[3] Vowel     ::= [aeiou]
```

Rule 1 states that a word is a consonant followed by one or more vowels followed by another consonant. Rule 2 states that a consonant is any letter other than a, e, i, o, or u. Rule 3 states that a vowel is any of the letters a, e, i, o, or u. (The exact syntax of the rules, the meaning of square brackets and other special symbols, is laid out in the XML specification.)

Using the above example, is this red a `Word`? Yes.

1. red is the letter r followed by the letter e followed by the letter d: `'r' 'e' 'd'`.
2. `r` is a `Consonant` by rule 2, so red is: `Consonant 'e' 'd'`
3. `e` is a vowel by rule 3, so red is: `Consonant Vowel 'd'`.
4. By rule 2 again, red is: `Consonant Vowel Consonant` which, by rule 1, is a `Word`.

By the same analysis, reed , road , and xeaiioug are also words, but rate is not. There is no way to match `Consonant Vowel Consonant Vowel` using the EBNF above. XML is defined by an EBNF grammar of about 80 rules. Although the rules are more complex, the same sort of analysis allows an XML parser to determine that `<greeting>Hello World</greeting>` is a syntactically correct XML document while

`<greeting]Wrong Bracket!</greeting>` is not.

In very general terms, that's all there is to it. You'll find all the details about EBNF in *Compilers: Principles, Techniques, and Tools* by Aho, Sethi, and Ullman or in any modern compiler text book.

While EBNF isn't an efficient way to represent syntax for human consumption, there are programs that can automatically turn EBNF into a parser. This makes it a particularly efficient way to represent the syntax for a language that will be parsed by a computer.

Revision History

Revision 1.1.1 18 Sep 1998 Revised by: nwalsh

Draft of update with respect to the final W3C Recommendation of 10 Feb 1998.

Revision 1.1 18 Feb 1998 Revised by: nwalsh

The title of this article has been changed. The former title was simply An Introduction to XML. In preparing this article for publication on my own web site, I've added a couple of sections that were cut from the Journal version because the content overlapped with other articles. Note: this article has not yet been updated to reflect changes that occured between the XML working draft that was current in September, 1997 and the final recommendation of Feb, 1998.