Using XML to Build the Annotated XML Specification

by Tim Bray

The design of XML 1.0 stretched over 20 months ending in February 1998, with input from a couple of hundred of the world's best experts in the area of markup, publishing, and Web design. The result of that work, the XML 1.0 Specification, is a highly condensed document that contains little or no information about how it came to read the way it does.

Even before the release of XML 1.0, it became obvious that some parts of the spec were self-explanatory, while others were causing headaches for its users.

The Annotated XML Specification addresses both of these problems. It supplements the basic specification, first with historical background and explanation of how things came to be the way they are, and second with detailed explanations of the portions of the spec that have proved difficult. Commercially, it has been a success; in its first month on the Web, it had over 100,000 page views from over 26,000 unique Internet addresses. It remains, by a substantial margin, the most popular item available at the XML.com site.

This article explains how I created the Annotated XML Specification. If you haven't looked at it, you might want to give it a glance before reading about it, or even better, open it in another browser window while you read about it here.

How the Annotated XML Specification Works
Tim describes the architecture of the AXML system and the design decisions he made.

Flipping the Links
How Java was used to convert the XML to HTML.

SISTER SITES
Seybold Seminars
Web Review
Perl.com
WebCoder
WebFonts
W3 Journal

A Songline PACE Production

Conclusion: How Much Work Was It?

The conclusion of Tim Bray's explanation of how he created the Annotated XML Specification.

Next: How the Annotated XML Specification Works

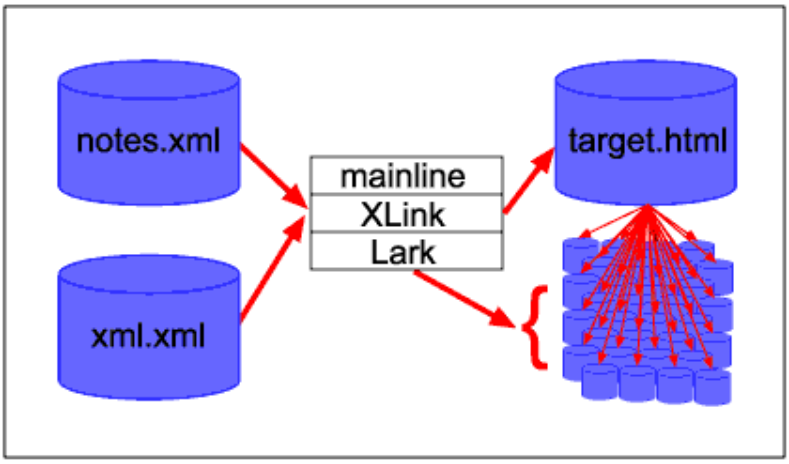
Sept. 12, 1998

Building the Annotated XML Specification

How the Annotated XML Specification Works

by Tim Bray

The architecture of the system, illustrated below, is simple enough:



The XML 1.0 specification is accessed in read-only mode. For convenience, I keep a local copy in a file called *xml.xml*. All the annotations live in another single file (about 25% larger than the XML specification itself) named *notes.xml*. A Java program, based on my Lark processor, reads both *xml.xml* and *notes.xml* and builds in-memory tree-structured representations of both. After processing all the links in *notes.xml*, the program writes out a file called *target.html*, which is the annotated version of the spec, and a large number of small files, each containing one of the annotations. The rest of this article outlines what's in those files and what the program does.

Design Choices for the Annotation

The first question I had to face in constructing the annotation was whether or not to use the highly-unfinished XLink and XPointer technologies, currently under development in the World Wide Web Consortium (W3C) XML Activity. XLink and XPointer have two large advantages: they are built for XML and can point at arbitrary locations inside a document.

On the other hand, neither spec is nearly finished, and they have changed (in syntax, if not at a conceptual level) from draft to draft. Furthermore, there were (in early 1998) neither commercial nor freeware implementations available. So if I were going to use this technology, I was going to have to write all the software myself. I decided to go ahead with XLink and XPointer, and while the syntax described here is about a year behind the latest drafts, I believe that what I've done is conceptually in tune with the current thinking, so the syntax will be easy to upgrade once the spec settles down.

XLink: A Quick Review

The XLink spec is concerned with recognizing which elements are being used as links, and with giving those links some useful structure and properties. HTML has a very simple solution to this set of problems: all linking elements have to be named **A**, and they have to point at one "resource", and the resource's address is found in the **HREF=** attribute.

```
&LT;A HREF="resource_address"&GT;
```

XLink tries to be more general than HTML: any element can serve as a link, and is identified as such by using a magic reserved attribute, **`xml:link=`**. There are two kinds of XLinks, identified by the values **`xml:link="simple"`** and **`xml:link="extended"`**. In the annotation, I only used the **`extended`** flavor, so I won't discuss **`simple`** XLinks here. Extended XLinks can have a bunch of useful associated information:

in-line

> If a linking element is "in-line", the element itself is one of the ends of the link. All HTML links are in-line, as are all the links in the Annotated Spec. Out-of-line links are really the Wild Blue Yonder of hypertext theory and practice.

labels

XLinks can have labels, both machine-readable (provided in the **`role=`** attribute) and human-readable (in the **`title=`** attribute).

behavior

XLink includes some tools for controlling the behavior of a link when it's being followed. In HTML, links really have only one behavior; you are at one page, you follow a link, then you're somewhere else. Since I had to use the Web as it is today to deliver the annotations, I wasn't able to get fancy with behaviors.

The `x` Element

In the Annotated Spec, I used an element named **`x`** as the chief linking element. If you were to dress it up with all the necessary attributes, one of these elements would look like this:

```
<x
  xml:link="extended"
  inline="true"
  content-role="commentary"
  content-title="Annotation" >
  ... contents of the linking element go here ...
</x>
```

If every one of the 312 annotations had to carry around all those attributes, the Annotated Spec would be hard to write and to work with. Fortunately, XML has attribute defaulting, so I could provide all these attributes just once, in the document header:

```
<!DOCTYPE Annotations [
 <!ELEMENT x (here|spec)+>
 <!ATTLIST x
   xml:link       CDATA  #FIXED "extended"
   inline         CDATA  #FIXED "true"
   content-role   CDATA  #FIXED "commentary"
   content-title  CDATA  #FIXED "Annotation"
   id             ID     #REQUIRED> ]>
<Annotations>
<x id="first-link-id"> ... content of first link ... </x>
<x id="second-link-id"> ... content of second link ... </x>
...
</Annotations>
```

The real role of the **`x`** element is to hold one **`here`** element and a bunch of **`spec`** elements. The **`here`** element actually contains the text of the annotation, while each of the **`spec`** elements points at a location in the XML spec where the annotation applies. Most of the annotations apply to only one location, but there are a few that attach to many places. For example, there is an annotation saying that DTD keywords (such as `DOCTYPE` , `SYSTEM`, `ELEMENT`, and `ATTLIST`) must be in upper-case; this annotation is attached to the first definition of each of these keywords.

The `here` Element

This element is what XLink calls a "Locator" - it serves as one end of the extended link, and contains the annotation. It has a lot of attributes, most of which are defaulted and don't actually appear in the body of the document. Here's the declaration, with all those attributes:

```
 <!ELEMENT here ANY>
 <!ATTLIST here
  xml:link CDATA #FIXED "locator"
  actuate  CDATA #FIXED "auto"
  show     CDATA #FIXED "replace"
  role     CDATA #FIXED "annotation"
  title    CDATA #REQUIRED
  href     CDATA #FIXED "here()"
  index    CDATA #IMPLIED">
```

Here's what all those attributes mean:

`xml:link="locator"`

tells the processing program that this is a locator

`actuate="auto"`

specifies that the link should be processed as soon as it's found - this differs from the

Web browser behavior of just displaying the link (as underlined blue text) then waiting for the user activate it.

`show`

tells the processor that the result of following this link should replace the target of the previous one that was followed.

`role`

tells the processor that this **`here`** element contains the annotation, not a pointer into the spec.

`title`

provides a human-readable label for this link - this is required to be present since it is used to generate the title in the Web implementation.

`href`

points to the annotation; which in the Annotated Spec is just the content of this element.

`index`

not part of the XLink apparatus - used to build an index of all the annotations; if it's not provided, the **`title`** value is used.

The content is declared as **`ANY`** and contains text marked up with HTML tags. Since this annotation was designed for Web delivery, and this content was designed to be read by humans, I felt that HTML was adequate to meet my formatting needs. HTML also had the advantage that I didn't have to write code to convert it for delivery. So far, I've found HTML perfectly satisfactory for this particular application. However, I do *not* draw the conclusion that HTML is going to be the right presentation solution for every, or even most, hypertext applications. Since the annotation is an XML document, the HTML has to be well-formed, to allow processing with XML-processor based tools.

Here's one of the **`here`** elements:

```
<here title='The Document Entity is Special'
 index='Document Entity, Special Status Of'>
<p>The differences between the document entity and
any other external parsed entity are:</p>
<ol><li>The document entity can begin with an
<Sref href='&h;dt-xmldecl'>XML declaration</Sref>,
other external parsed entities with a
<Sref href='&h;NT-TextDecl'>text declaration</Sref>.
</li>
<li>The document entity can contain a
<Sref href='&h;dt-doctype'>document type
declaration</Sref>.</li></ol></here>
```

Note that there are a few magic non-HTML elements mixed in; in this case, the **`Sref`** element, which is used to encode a pointer back into the XML specification. The Annotated Spec also uses **`Xref`** (external reference) and **`Nref`** (reference to another annotation) elements. That pointer (in the **`href`** attribute) uses an entity reference, **`&h;`**, which contains the URL for the XML spec; this is a good idea since there are hundreds of these URLs in the Annotated spec, and the location I read the XML spec from might change.

The spec Element

This is another XLink "Locator", which contains a pointer into the XML spec, indicating what part of the spec the annotation is there to annotate. Here's its declaration:

```
<!ELEMENT spec EMPTY>
<!ATTLIST spec
  xml-link CDATA #FIXED "locator"
  actuate  CDATA "user"
  show     CDATA "replace"
  role     (Using|History|Tech|Misc|Example) "Misc"
  title    CDATA "Into XML Specification"
  href     CDATA #REQUIRED>
```

The only really interesting attributes are **`role`**, saying which kind of annotation it is, and **`href`**, which contains the URL pointing into the spec. The allowed values of **`role`** correspond to the Ⓤ, Ⓗ, Ⓣ, Ⓜ, and Ⓔ symbols that mark the annotations in the spec.

Here's an example, not just of a **`spec`** element, but of a whole **`x`** element with its **`spec`** and **`here`** children:

```
<x id='RfC1808URI'>
<spec role='Using' href='&s;id(RFC1808)'/>
<here title='RFC 1818 URL'>
<p><Xref href='ftp://ds.internic.net/rfc/rfc1808.txt'>
ftp://ds.internic.net/rfc/rfc1808.txt</Xref></p>
</here></x>
```

In this example, the target of the reference is easily identified, since it is just a bibliographic entry that has an **id** attribute.

XPointer: A Quick Review

In an XLink Locator, there is an **href=** attribute that gives the URI identifying an end of the link. An XPointer is a string of characters that is used after the **#** "fragment separator" character in that **href=** value. It points into the XML document by treating it as a tree structure and identifying numbered **child** and **descendant** nodes.

The best XPointers are those that are based on "ID Attributes", that is to say attributes that have been declared to have a unique value. These are easy for an XML Processor to find and traverse to, but there is a problem in that you don't know which attributes are so declared unless you are prepared to read the whole DTD. This means, if you read the XML spec carefully, that to be sure, you have to use a validating XML processor. In my case, I was able to get away with using Lark, my non validating processor, simply by assuming that any attribute whose name was **id** was an ID attribute.

XPointer provides quite a few different verbs for selecting objects inside the document tree; in the Annotated Spec I was able to get away with using only the **id**, **descendant**, **child**, and **string** verbs. Furthermore, I could have got by without using the **string** operator. This raises the question: if something as complex as the Annotated Spec can be constructed with just these operators, do we really need all the others?

Here are some interesting examples of XPointers from the Annotated Spec; they have been set up to point into the indicated part of the HTML version of the XML spec.

```
id(NT-Mixed)child(1,rhs)string(1,"ATA",4)
id(sec-xml-wg)descendant(18,name)string(1,"gue",4)
id(sec-guessing)child(8,p)string(1,XML.,4)
id(sec-prolog-dtd)descendant(2,vcnote)
```

I actually authored each of the 312 XPointers in the Annotated Spec by hand, finding IDs, counting children, and matching strings. At the summer 1998 XML Developers' Day conference, David Megginson showed how I could have programmed GNU Emacs (which is what I use for editing anyhow) to construct these automatically at the touch of a key. The lesson is that any sensible XML editing environment ought to make it easy to construct this kind of hyperlink.

Next: **Flipping the Links**
Prev: **Building the Annotated XML Specification**

Sept. 12, 1998

Building the Annotated XML Specification

Flipping the Links

by Tim Bray

Once I had authored all these hyperlinks, I faced the problem of how to display them. Unfortunately, in 1998 there weren't any Web browsers that could do the job, and in fact, I wanted this to be usable by anyone with a basic HTML browser. So my program had to turn the links around; instead of one XML file containing hundreds of links into another, I needed one HTML file (the annotated spec) containing hundreds of links, each to one little annotation file. All this was done in Java, as noted above. The rest of this article gives a step-by-step description of the program, and may be a bit challenging if you're not a Java programmer.

Step 1: Parse the Annotations

First, the program creates an instance of the Lark parser, and in one step, reads the annotations file:

```
annot = new Lark();
r = new XmlInputStream(new FileInputStream(args[0]));
System.err.println("Parsing Annotations...");
annot.buildTree(true);
annot.saveText(true);
aroot = annot.readXML(h, r);
```

After this operation, the variable **aroot** points to the root of the annotations document tree.

Step 2: Put the XLinks in a Vector

This code runs through the annotations tree and puts all the XLinks that it finds in the variable **xlinks**:

```
Vector xlinks = new Vector();
findXLinks(aroot, xlinks);
System.out.println("xlinks: "+xlinks.size());
...
private static void findXLinks(Element e, Vector v)
...
    XLink link = XLink.isLink(e, sIDs);
    if (link != null) v.addElement(link);
...
    children = e.children();
    for (i = 0; i < children.size(); i++)
    {
      child = children.elementAt(i);
      if (child instanceof Element)
        findXLinks((Element) child, v);
    }
```

This code looks at each element, then calls itself recursively on that element's children. It uses the routine **isLink** to determine whether an element is an XLink:

```
public static XLink isLink(Element e, Hashtable ids)
...
  String form = e.attributeValue("xml:link");
  if (form == null) return null;
  else if (form.equals("simple") || form.equals("extended"))
  {
    XLink link = new XLink(ids);
    link.loadFromElement(e, ids);
    return link;
  }
  ...
```

An element is an XLink if it has an attribute **xml:link=** whose value is either **simple** or **extended**.

Each XLink, once identified, is stored into an XLink object for later re-use; these objects are what populate

the **xlinks** vector mentioned above. The function which loads up the XLink data structures is the same for the extended and locator linking elements, since their makeup is almost identical.:

```
private void loadFromElement(Element e, Hashtable ids)
  ...
  // load up role, content-role, etc. fields from attribute values
  mRole = e.attributeValue("role");
  ..
  splitHref(); // save the HRef and build an XPointer if there is one
  Vector children = e.children();
  for (i = 0; i < children.size(); i++)
  {
      if ( // check xml:link att to see if this child is a locator
      {
        XLink link = new XLink(ids);
        link.loadFromElement((Element) child, ids);
        mLocators.addElement(link);
```

Step 3: Traverse the XLinks

Once the **xlinks** vector is filled up, we run through it, traversing those whose role is not "annotation" - in effect, this means we traverse only the **spec** XPointers:

```
findXLinks(aroot, xlinks);
...
System.err.println("Traversing...");
for (i = 0; i < xlinks.size(); i++)
{
  XLink link = (XLink) xlinks.elementAt(i);
  targets = link.traverseExcept("annotation");
  if (targets.size() == 0)
  {
    System.out.println("Dangling Annotation!");
    link.dump(System.out);
  }
```

The routine **traverseExcept** returns the list of targets that the corresponding XPointer points at. If that list is of size 0, it means that the XPointer is broken and doesn't point at anything. Since I was constructing the XPointers by hand, I saw this message a lot during the construction of the Annotated Spec.

The example above doesn't show the actual traversal of a single link; here's the code which does that:

```
private void doTraverse(Vector ret, Element linkingElement)
  ...
  if (// I haven't already parsed this instance
  {
    // parse the instance
    System.err.println("Parsing target...");
    Lark xml = new Lark();
    ...
    sTargetRoot = xml.readXML(h, r);
    System.err.println("Done.");
    }
  }
  ret.addElement(sTargetRoot);
  // if there's an XPointer, traverse that
  if (mXP != null)
    mXP.traverse(ret);
```

This code first looks to see whether the URL in the XLink has already been parsed. If it hasn't, it makes a new parser and parses that document. Since in this case, *all* the links are into the XML spec, this code only gets executed once, causing the parsing of the XML spec when the first **spec** locator element is traversed.

The code above is putting the results of each traversal in the vector **ret**. First of all, it inserts a pointer to the root of the target XML document, then checks to see if the XLink contains an XPointer (in the Annotated Spec, they all do) and if so, traverses that. The code for that is here:

```
public void traverse(Vector ret)
  ...
  // an XPointer is stored as an array of steps in an obvious way
```

```
for (i = 0; i < mSteps.size(); i++)
{
  step = (XPStep) mSteps.elementAt(i);
  kw = step.kw();
  switch (kw)
  {
  case sDescendant:
    for (j = 0; j < old.size(); j++)
    {
      start = (Element) old.elementAt(j);
      inOrder(start, ret, step, 0);
    }
    break;

  case sString:
    for (j = 0; j < old.size(); j++)
    {
      start = (Element) old.elementAt(j);
      searchForString(start, ret, step, 0);
    }
    break;

  case sID:
    ret.addElement(mIDs.get(step.type()));
    break;

  case sChild:
    ...
        for (k = 0; k < children.size() && !done; k++)
        {
          o = children.elementAt(k);
          if (step.match(o))
    ...
  case sRoot: case sHere: case sDitto: case sHTML:
  case sAncestor: case sPreceding: case sPSibling:
  case sFollowing: case sFSibling: default:
    if (kw < sMaxKW)
      throw new Exception("Can't do '" + sKeywords[kw] + "' yet");
    else
      throw new Exception("Bogus KW value "+kw);
```

As the comment says, each XPointer object contains an array with each element being one of the XPointer's steps. First, look for a moment at the **`sChild`** code above. It uses the method **`step.match()`** to check whether one step matches a particular child.

The code for **`sDescendant`** also uses that code, but of course has to do an in-order traversal of the whole subtree. Note how few of the XPointer verbs are implemented.

This part of the code, following all the links and making the connections between the annotation and the spec, takes almost no time to run. Some of the elapsed time in running this program is spent in parsing the annotations file and the XML spec, but most of it goes into the task of loading these fairly large complex documents into trees in memory. This burns immense amounts of memory; the two files are together less than 500K in size, but the two trees in memory use well over 10 megabytes. There are some tricks that could be used to make the trees more compact, but even so, the lesson is that fully-parsed XML documents burn a lot of memory. This is one reason why it's a good idea to do as much work as possible through a stream API such as SAX. However, the annotation in particular and XPointer processing in general are examples of jobs that it would be really hard to do without having a tree in memory.

Step 4: Decorate the Target

At the end of all this work, the vector **`targets`** contains a list of all the locations in the XML spec that have annotations pointing at them. Next, the code runs through the XML spec and, for each element that is the target of a link, it adds a new child which is an HTML **`A`** element with an **`href=`** pointer pointing at the appropriate annotation file. This element is added as the *last* child of the annotated element, so the annotation symbol will appear at the end of the target. This is an arbitrary choice that worked fine for the Annotated Spec, but may not be appropriate for many other applications. If the XPointer points into the middle of a chunk of text, more work is required; the text has to be split into two nodes, and the HTML **`A`** element inserted between them:

```
for (k = 0; k < targets.size(); k++)
{
  Object o = targets.targetAt(k);
  Element lE = link.linkingElement();
  if (o instanceof Element)
  {
    ((Element) o).addChild(makeAnchor(lE, targets.roleAt(k)));
  }
  else if (o instanceof Text)
  {
    // OK, we're going to have to split this text node
    splitText((Text) o, targets.offsetAt(k),
      lE, targets.roleAt(k));
  }
```

Once all these extra elements have been spliced into the XML spec, writing out the annotated version is pretty easy. I already had a large amount of Java code that converts the XML to HTML, which is used to generate the HTML versions of the spec that you can find at the W3C site and elsewhere. This code works by having a Java class for each element; all I had to do was to add a new class to handle the new spliced-in **A** elements, which involved very little work aside from putting in the Ⓗ-style decorations depending on the **role** attribute:

```
System.err.println("Done.  Writing target...");

xroot = XLink.currentRoot();
out = new PrintStream(new FileOutputStream("target.html"));
if (xroot != null)
  printer.writeHTML(xroot, out, false);
out.close();
System.err.println("Done.  Writing notes...");
```

Step 5: Write the Annotations

Writing out the annotations is pretty easy too. They'd been placed in a vector named **notes** constructed earlier in the process. For each one, we open a file, dump in the HTML text (with a bit of extra decoration) and the job's done.

```
System.err.println("Done.  Writing notes...");

for (i = 0; i < notes.size(); i++)
{
  if (notes.elementAt(i) instanceof Element)
  {
    child = (Element) notes.elementAt(i);
    title = child.attributeValue("id");
    out = new PrintStream(new FileOutputStream(
                "notes/" + title + ".html"));
    out.println("<HTML><HEAD><TITLE>"
          + title + "</TITLE>");
    ...
```

We use the value of the **id** attribute for the filename; that's why it is **#REQUIRED**.

How the Annotated XML Specification Works

Flipping the Links

Conclusion: How Much Work Was It?

Next: Conclusion: How Much Work Was It?
Prev: How the Annotated XML Specification Works

XML From The Inside Out

Sept. 12, 1998

Java, Publishing, Tools

Books about XML Java, Publishing, Tools

Amazon.com Book Search

FIND
Current Issue
Article Search
Previous Issues

FEATURES
Annotated XML 1.0
What is XML?

GUIDES
XML Resource Guide
The Standards List
NEW:
The Submissions List
Authoring Tools Guide
Content Mgmt. Tools
XML Events Calendar

FORUMS
View Messages
Register
Login
Change your profile

COLUMNS
Puzzlin' Evidence
XML Q&A
XML:Geek

LIBRARY
Seybold Publications
W3 Journal: XML

TOOLBOX
RUWF?
The XML Syntax Checker
XML Testbed

ABOUT XML.COM
Who We Are
Our Mission
Become a Sponsor

Building the Annotated XML Specification

Conclusion

How Much Work Was It?

by Tim Bray

Building the Annotated Spec required writing quite a lot of Java code. Here's a summary of the lines of code required, which is a poor way to measure the amount of work. I didn't keep track of the amount of time it took, because I was writing and debugging the code as I wrote the annotations, and doing both of these things in parallel with a lot of traveling and lecturing and consulting.

Lines of code	Java File	Function
403	Annotate.java	Mainline, bookkeeping
52	Link.java	All the information describing a link from the annotation file into the XML spec
276	XLink.java	XLink processing
82	XPStep.java	XPointer step processing
303	XPointer.java	XPointer processing
1116		Total

This is nowhere near being a complete implementation of XLink and XPointer. It contains just enough logic to solve this particular application's problems.

Lessons

From a commercial point of view, the Annotated Spec is a huge success. We should be cautious in drawing conclusions from this exercise, since the average hypertext creator is not going to regard writing 1100-plus lines of Java as a normal part of the authoring process. I think, though, that some useful lessons do emerge from this work:

- Hypertext annotation is a useful technique for adding value to complex reference texts.

- The basic design of XLink and XPointer seem, in practice, to be sound, in terms of providing the machinery necessary to build a sophisticated, usable hypertext.
- While this project did not involve a general-purpose implementation of XLink/XPointer, a very large part of the necessary logic was roughed-in without uncovering any unforeseen engineering problems.
- Implementing XPointer, at the moment, requires loading all of a parsed document into an in-memory tree. This consumes excessive memory for large documents, so a "virtual tree" facility, allowing tree-walking without actual memory loading, will likely be essential for successful industrial implementations of XML hypertexts.
- We need a solution to the problem of identifying **`ID`** attributes without having to use a validating processor.
- We need some serious debate on the selection of verbs to be included in the XPointer specification. Quite likely, there is a case for having one or two more verbs than I used in putting together the Annotated Spec. On the other hand, it seems probable that XPointers would be improved by removing one or two of the many existing verbs.
- The task of generating XPointers should be automated, not done by hand.
- The Annotated spec had to be drastically transformed into conventional HTML for delivery, since there was no widely-deployed software available with the capability of displaying this type of hypertext. There is an important open question as to whether, in general, complex hypertexts will require extensive processing for delivery.

How the Annotated XML Specification Works

Flipping the Links

Conclusion: How Much Work Was It?

Prev: Flipping the Links

Introduction to the Annotated XML Specification

by Tim Bray

The other window contains the XML specification; this window the commentary on it. The content and appearance of the XML spec are exactly as in the official version; it has not been edited in any way to generate this presentation. The commentary is contained in external XML files, with XML hyperlinks into the (entirely unaltered) XML version of the spec. The footnoted HTML version that you see on the screen is program-generated. The annotations are flagged as follows:

Historical or cultural commentary; some entertainment value.

Technical explanations, including amplifications, corrections, and answers to Frequently Asked Questions.

Advice on how to use this specification.

Examples to illustrate what the spec is saying.

Annotations that it's hard to find a category for.

REC-xml-19980210

Extensible Markup Language (XML) 1.0

W3C Recommendation 10-February-1998

This version:

http://www.w3.org/TR/1998/REC-xml-19980210

http://www.w3.org/TR/1998/REC-xml-19980210.xml

http://www.w3.org/TR/1998/REC-xml-19980210.html

http://www.w3.org/TR/1998/REC-xml-19980210.pdf

http://www.w3.org/TR/1998/REC-xml-19980210.ps

Latest version:

http://www.w3.org/TR/REC-xml

Previous version:

http://www.w3.org/TR/PR-xml-971208

Editors:

Tim Bray (Textuality and Netscape) <tbray@textuality.com>

Jean Paoli (Microsoft) <jeanpa@microsoft.com>

C. M. Sperberg-McQueen (University of Illinois at Chicago) <cmsmcq@uic.edu>

Abstract

The Extensible Markup Language (XML) is a subset of SGML that is completely described in this document. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

Status of this document

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document specifies a syntax created by subsetting an existing, widely used international text processing standard (Standard Generalized Markup Language, ISO 8879:1986(E) as amended and corrected) for use on the World Wide Web. It is a product of the W3C XML Activity, details of which can be found at http://www.w3.org/XML. A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

This specification uses the term URI, which is defined by [Berners-Lee et al.], a work in progress expected to update [IETF RFC1738] and [IETF RFC1808].

The list of known errors in this specification is available at http://www.w3.org/XML/xml-19980210-errata.

Please report errors in this document to xml-editor@w3.org.

Extensible Markup Language (XML) 1.0

Table of Contents

Appendices

---

1. Introduction

Extensible Markup Language, abbreviated XML, describes a class of data objects(M) called XML documents(U) and partially describes the behavior of computer programs which process them.(U) XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]. By construction, XML documents are conforming SGML documents. (U)

XML documents are made up of storage units called entities, which contain either parsed or unparsed data.(U) Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

[Definition:] A software module called an **XML processor**(H) is used to read XML documents and provide access to their content and structure. [Definition:] It is assumed that an XML processor is doing its work on behalf of another module, called the **application**. This specification describes the required behavior of an XML processor in terms of how it must read XML data and the information it must provide to the application.(T)

1.1 Origin and Goals

XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996.(H) It was chaired by Jon Bosak(H) of Sun Microsystems with the active participation of an XML Special Interest Group (previously known as the SGML Working Group)(H) also organized by the W3C. The membership of the XML Working Group is given in an appendix. Dan Connolly served as the WG's contact with the W3C. (H)

The design goals(H) for XML are:

1. XML shall be straightforwardly usable over the Internet.(H)

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.

This specification, together with associated standards (Unicode and ISO/IEC 10646 for characters, Internet RFC 1766 for language identification tags, ISO 639 for language name codes, and ISO 3166 for country name codes), provides all the information necessary to understand XML Version 1.0 and construct computer programs to process it.

This version of the XML specification may be distributed freely, as long as all text and legal notices remain intact.

1.2 Terminology

The terminology used to describe XML documents is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of an XML processor:

may

[Definition:] Conforming documents and XML processors are permitted to but need not behave as described.

must

Conforming documents and XML processors are required to behave as described; otherwise they are in error.

error

[Definition:] A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it.

fatal error

[Definition:] An error which a conforming XML processor must detect and report to the application. After encountering a fatal error, the processor may continue processing the data to search for further errors and may report such errors to the application. In order to support correction of errors, the processor may make unprocessed data from the document (with intermingled character data and markup) available to the application. Once a fatal error is detected, however, the processor must not continue normal processing (i.e., it must not continue to pass character data and information about the document's logical structure to the application in the normal way).

at user option

Conforming software may or must (depending on the modal verb in the sentence) behave as described; if it does, it must provide users a means to enable or disable the behavior described.

validity constraint

A rule which applies to all valid XML documents. Violations of validity constraints are errors; they must, at user option, be reported by validating XML processors.

well-formedness constraint

A rule which applies to all well-formed XML documents. Violations of well-formedness constraints are fatal errors.

match

[Definition:] (Of strings or names:) Two strings or names being compared must be identical. Characters with multiple possible representations in ISO/IEC 10646 (e.g. characters with both precomposed and base+diacritic forms) match only if they have the same representation in both strings. At user option, processors may normalize such characters to some canonical form. No case folding is performed. (Of strings and rules in the grammar:) A string matches a grammatical production if it belongs to the language generated by that production. (Of content and content models:) An element matches its declaration when it conforms in the fashion described in the constraint "Element Valid".

for compatibility

[Definition:] A feature of XML included solely to ensure that XML remains compatible with SGML.

for interoperability

[Definition:] A non-binding recommendation included to increase the chances that XML documents can be processed by the existing installed base of SGML processors which predate the WebSGML Adaptations Annex to ISO 8879.

2. Documents

[Definition:] A data object is an **XML document** if it is well-formed, as defined in this specification. A well-formed XML document may in addition be valid if it meets certain further constraints.

Each XML document has both a logical and a physical structure. Physically, the document is composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. A document begins in a "root" or document entity. Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup. The logical and physical structures must nest properly, as described in "4.3.2 Well-Formed Parsed Entities".

2.1 Well-Formed XML Documents

[Definition:] A textual object is a well-formed XML document if:

1. Taken as a whole, it matches the production labeled `document`.
2. It meets all the well-formedness constraints given in this specification.
3. Each of the parsed entities which is referenced directly or indirectly within the document is well-formed.

Document
[1] `document ::= prolog element Misc*`

Matching the `document` production implies that:

1. It contains one or more elements.
2. [Definition:] There is exactly one element, called the **root**, or document element, no part of which appears in the content of any other element. For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. More simply stated, the elements, delimited by start- and end-tags, nest properly within each other.

[Definition:] As a consequence of this, for each non-root element C in the document, there is one other element P in the document such that C is in the content of P, but is not in the content of any other element that is in the content of P. P is referred to as the **parent** of C, and C as a **child** of P.

2.2 Characters

[Definition:] A parsed entity contains **text**, a sequence of characters, which may represent markup or character data. [Definition:] A **character** is an atomic unit of text as specified by ISO/IEC 10646 [ISO/IEC 10646]. Legal characters are tab, carriage return, line feed, and the legal graphic characters of Unicode and ISO/IEC 10646. The use of "compatibility characters", as defined in section 6.8 of [Unicode], is discouraged.

Character Range

```
[2] Char ::= #x9 | #xA | #xD | [#x20-#xD7FF]     /* any Unicode
           | [#xE000-#xFFFD]                       character,
           | [#x10000-#x10FFFF]                    excluding the
                                                   surrogate blocks,
                                                     FFFE, and
                                                   FFFF. */
```

The mechanism for encoding character code points into bit patterns may vary from entity to entity. All XML processors must accept the UTF-8 and UTF-16 encodings of 10646; the mechanisms for signaling which of the two is in use, or for bringing other encodings into play, are discussed later, in "4.3.3 Character Encoding in Entities".

2.3 Common Syntactic Constructs

This section defines some symbols used widely in the grammar.

S (white space) consists of one or more space (#x20) characters, carriage returns, line feeds, or tabs.

White Space

```
[3] S ::= (#x20 | #x9 | #xD | #xA)+
```

Characters are classified for convenience as letters, digits, or other characters. Letters consist of an alphabetic or syllabic base character possibly followed by one or more combining characters, or of an ideographic character. Full definitions of the specific characters in each class are given in "B. Character Classes".

[Definition:] A **Name** is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters. Names beginning with the string "`xml`", or any string which would match `(('X'|'x') ('M'|'m') ('L'|'l'))`, are reserved for standardization in this or future versions of this specification.

Note: The colon character within XML names is reserved for experimentation with name spaces. Its meaning is expected to be standardized at some future point, at which point those documents using the colon for experimental purposes may need to be updated. (There is no guarantee that any name-space mechanism adopted for XML will in fact use the colon as a name-space delimiter.) In practice, this means that authors should not use the colon in XML names except as part of name-space experiments, but that XML processors should accept the colon as a name character.

An `Nmtoken` (name token) is any mixture of name characters.

Names and Tokens

```
[4] NameChar ::= Letter | Digit | '.' | '-' | '_' | ':'
                 | CombiningChar | Extender
[5]     Name ::= (Letter | '_' | ':') (NameChar)*
[6]    Names ::= Name (S Name)*
[7]  Nmtoken ::= (NameChar)+
[8] Nmtokens ::= Nmtoken (S Nmtoken)*
```

Literal data is any quoted string not containing the quotation mark used as a delimiter for that string. Literals are used for specifying the content of internal entities (EntityValue), the values of attributes (AttValue), and external identifiers (SystemLiteral). Note that a SystemLiteral can be parsed without scanning for markup.

Literals

```
 [9]   EntityValue ::= '"' ([^%&"] | PEReference | Reference)* '"'
                     |  "'" ([^%&'] | PEReference | Reference)*
                     "'"
[10]     AttValue ::= '"' ([^<&"] | Reference)* '"'
                     |  "'" ([^<&'] | Reference)* "'"
[11] SystemLiteral ::= ('"' [^"]* '"') | ("'" [^']* "'")
[12]  PubidLiteral ::= '"' PubidChar* '"' | "'" (PubidChar - "'")*
                     "'"
[13]     PubidChar ::= #x20 | #xD | #xA | [a-zA-Z0-9]
                     | [-'()+,./:=?;!*#@$_%]
```

2.4 Character Data and Markup

Text consists of intermingled character data and markup. [Definition:] **Markup** takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, and processing instructions.

[Definition:] All text that is not markup constitutes the **character data** of the document.

The ampersand character (&) and the left angle bracket (<) may appear in their literal form *only* when used as markup delimiters, or within a comment, a processing instruction, or a CDATA section. They are also legal within the literal entity value of an internal entity declaration; see "4.3.2 Well-Formed Parsed Entities". If they are needed elsewhere, they must be escaped using either numeric character references or the strings "`&`" and "`<`" respectively. The right angle bracket (>) may be represented using the string "`>`", and must, for compatibility, be escaped using "`>`" or a character reference when it appears in the string "`]]>`" in content, when that string is not marking the end of a CDATA section.

In the content of elements, character data is any string of characters which does not contain the start-delimiter of any markup. In a CDATA section, character data is any string of characters not including the CDATA-section-close delimiter, "`]]>`".

To allow attribute values to contain both single and double quotes, the apostrophe or single-quote character (') may be represented as "`&apos;`", and the double-quote character (") as "`&quot;`".

Character Data

```
[14] CharData ::= [^<&]* - ([^<&]* ']]>' [^<&]*)
```

2.5 Comments

[Definition:] **Comments** may appear anywhere in a document outside other markup; in addition, they may appear within the document type declaration at places allowed by the grammar. They are not part of the document's character data; an XML processor may, but need not, make it possible for an application to retrieve the text of comments. For compatibility, the string "`--`" (double-hyphen) must not occur within comments.

Comments

```
[15] Comment ::= '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
```

An example of a comment:

```
<!-- declarations for <head> & <body> -->
```

2.6 Processing Instructions

[Definition:] **Processing instructions** (PIs) allow documents to contain instructions for applications.

Processing Instructions

```
[16]       PI ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))?
                  '?>'
[17] PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
```

PIs are not part of the document's character data, but must be passed through to the application. The PI begins with a target (`PITarget`) used to identify the application to which the instruction is directed. The target names "`XML`", "`xml`", and so on are reserved for standardization in this or future versions of this specification. The XML Notation mechanism may be used for formal declaration of PI targets.

2.7 CDATA Sections

[Definition:] **CDATA sections** may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string "`<![CDATA[`" and end with the string "`]]>`":

CDATA Sections

```
[18]  CDSect ::= CDStart CData CDEnd
[19] CDStart ::= '<![CDATA['
[20]   CData ::= (Char* - (Char* ']]>' Char*))
[21]   CDEnd ::= ']]>'
```

Within a CDATA section, only the `CDEnd` string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form; they need not (and cannot) be escaped using "`<`" and "`&`". CDATA sections cannot nest.

An example of a CDATA section, in which "`<greeting>`" and "`</greeting>`" are recognized as character data, not markup:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

2.8 Prolog and Document Type Declaration

[Definition:] XML documents may, and should,(H) begin with an **XML declaration** which specifies the version of XML being used.(T) For example, the following is a complete XML document, well-formed but not valid:(T)

```
<?xml version="1.0"?>
<greeting>Hello, world!</greeting>
```

and so is this:

```
<greeting>Hello, world!</greeting>
```

The version number "`1.0`" should be used to indicate conformance to this version of this specification; it is an error for a document to use the value "`1.0`" if it does not conform to this version of this specification. It is the intent of the XML working group to give later versions of this specification numbers other than "`1.0`", but this intent does not indicate a commitment(U) to produce any future versions of XML, nor if any are produced, to use any particular numbering scheme. Since future versions are not ruled out, this construct is provided as a means to allow the possibility of automatic version recognition, should it become necessary. Processors may signal an error if they receive documents labeled with versions they do not support.

The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute-value pairs with its logical structures. XML provides a mechanism, the document type declaration(U), to define constraints on the logical structure and to support the use of predefined storage units. [Definition:] An XML document is **valid** if it has an associated document type declaration(T) and if the document complies with the constraints expressed in it.(T)

The document type declaration must appear before the first element in the document.

Prolog

```
[22]      prolog ::= XMLDecl?(T) Misc* (doctypedecl Misc*)?
[23]    XMLDecl ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S?
                    '?>'(T)
[24] VersionInfo ::= S 'version' Eq (' VersionNum ' | " VersionNum
                    ")(U)
[25]          Eq ::= S? '=' S?
[26]  VersionNum ::= ([a-zA-Z0-9_.:] | '-')+
[27]        Misc ::= Comment | PI |  S
```

[Definition:] The XML **document type declaration** contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition,(U) or **DTD**. The document type declaration can point to an external subset (a special kind of external entity)(T) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together.

[Definition:] A **markup declaration** is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration. These declarations may be contained in whole or in part within parameter entities, as described in the well-formedness and validity constraints below. For fuller information, see "4. Physical Structures".

Document Type Definition

```
[28] doctypedecl ::= '<!DOCTYPE' S Name (S [ VC: Root Element Type ]
                     ExternalID)? S? ('['
                     (markupdecl
                     | PEReference | S)* ']'
                     S?)? '>'
[29]  markupdecl ::= elementdecl                [ VC: Proper
                     | AttlistDecl                Declaration/PE Nesting
                     | EntityDecl                 ]
                     | NotationDecl | PI
                     | Comment
                                                 [ WFC: PEs in Internal
                                                  Subset ]
```

The markup declarations may be made up in whole or in part of the replacement text of parameter entities. The productions later in this specification for individual nonterminals (`elementdecl`, `AttlistDecl`, and so on) describe the declarations *after* all the parameter entities have been included. U

Validity Constraint: Root Element Type
The `Name` in the document type declaration must match the element type of the root element.

Validity Constraint: Proper Declaration/PE Nesting
Parameter-entity replacement text must be properly nested with markup declarations. That is to say, if either the first character or the last character of a markup declaration (`markupdecl` above) is contained in the replacement text for a parameter-entity reference, both must be contained in the same replacement text. E

Well-Formedness Constraint: PEs in Internal Subset
In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.) T

Like the internal subset, the external subset and any external parameter entities referred to in the DTD must consist of a series of complete markup declarations of the types allowed by the non-terminal symbol `markupdecl`, interspersed with white space or parameter-entity references. However, portions of the contents of the external subset or of external parameter entities may conditionally be ignored by using the conditional section construct; this is not allowed T in the internal subset.

External Subset

```
[30]     extSubset ::= TextDecl? extSubsetDecl T
[31] extSubsetDecl ::= ( markupdecl | conditionalSect | PEReference
                       | S )*
```

The external subset and external parameter entities also differ from the internal subset in that in them, parameter-entity references are permitted *within* markup declarations, not only *between* markup declarations.

An example of an XML document with a document type declaration:

```
<?xml version="1.0"?>
<!DOCTYPE greeting SYSTEM "hello.dtd">
<greeting>Hello, world!</greeting>
```

The system identifier "`hello.dtd`" gives the URI of a DTD for the document. T

The declarations can also be given locally, as in this example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

If both the external and internal subsets are used, the internal subset is considered to occur before the external subset. This has the effect that entity and attribute-list declarations in the internal subset take precedence over those in the external subset.

2.9 Standalone Document Declaration

Markup declarations can affect the content of the document, as passed from an XML processor to an application; examples are attribute defaults and entity declarations. The standalone document declaration, which may appear as a component of the XML declaration, signals whether or not there are such declarations which appear external to the document entity.

Standalone Document Declaration

```
[32] SDDecl ::= S 'standalone' Eq (("'"    [ VC: Standalone Document
             ('yes' | 'no') "'") | ('"'   Declaration ]
             ('yes' | 'no') '"'))
```

In a standalone document declaration, the value "`yes`" indicates that there are no markup declarations external to the document entity (either in the DTD external subset, or in an external parameter entity referenced from the internal subset) which affect the information passed from the XML processor to the application. The value "`no`" indicates that there are or may be such external markup declarations. Note that the standalone document declaration only denotes the presence of external *declarations*; the presence, in a document, of references to external *entities*, when those entities are internally declared, does not change its standalone status.

If there are no external markup declarations, the standalone document declaration has no meaning. If there are external markup declarations but there is no standalone document declaration, the value "`no`" is assumed.

Any XML document for which `standalone="no"` holds can be converted algorithmically to a standalone document, which may be desirable for some network delivery applications.

Validity Constraint: Standalone Document Declaration
The standalone document declaration must have the value "`no`" if any external markup declarations contain declarations of:

- attributes with default values, if elements to which these attributes apply appear in the document without specifications of values for these attributes, or
- entities (other than `amp`, `lt`, `gt`, `apos`, `quot`), if references to those entities appear in the document, or
- attributes with values subject to normalization, where the attribute appears in the document with a value which will change as a result of normalization, or
- element types with element content, if white space occurs directly within any instance of those types.

An example XML declaration with a standalone document declaration:

```
<?xml version="1.0" standalone='yes'?>
```

2.10 White Space Handling

In editing XML documents, it is often convenient to use "white space" (spaces, tabs, and blank lines, denoted by the nonterminal S in this specification) to set apart the markup for greater readability. Such white space is typically not intended for inclusion in the delivered version of the document. On the other hand, "significant" white space that should be preserved in the delivered version is common, for example in poetry and source code.

An XML processor must always pass all characters in a document that are not markup through to the application. A validating XML processor must also inform the application which of these characters constitute white space appearing in element content.

A special attribute named `xml:space` may be attached to an element to signal an intention that in that element, white space should be preserved by applications. In valid documents, this attribute, like any other, must be declared if it is used. When declared, it must be given as an enumerated type whose only possible values are "`default`" and "`preserve`". For example:

```
<!ATTLIST poem   xml:space (default|preserve) 'preserve'>
```

The value "`default`" signals that applications' default white-space processing modes are acceptable for this element; the value "`preserve`" indicates the intent that applications preserve all the white space. This declared intent is considered to apply to all elements within the content of the element where it is specified, unless overriden with another instance of the `xml:space` attribute.

The root element of any document is considered to have signaled no intentions as regards application space handling, unless it provides a value for this attribute or the attribute is declared with a default value.

2.11 End-of-Line Handling

XML parsed entities are often stored in computer files which, for editing convenience, are organized into lines. These lines are typically separated by some combination of the characters carriage-return (#xD) and line-feed (#xA).

To simplify the tasks of applications, wherever an external parsed entity or the literal entity value of an internal parsed entity contains either the literal two-character sequence "#xD#xA" or a standalone literal #xD, an XML processor must pass to the application the single character #xA. (This behavior can conveniently be produced by normalizing all line breaks to #xA on input, before parsing.)

2.12 Language Identification

In document processing, it is often useful to identify the natural or formal language in which the content is written. A special attribute named `xml:lang` may be inserted in documents to specify the language used in the contents and attribute values of any element in an XML document. In valid documents, this attribute, like any other, must be declared if it is used. The values of the attribute are language identifiers as defined by [IETF RFC 1766], "Tags for the Identification of Languages":

Language Identification

```
[33] LanguageID ::= Langcode ('-' Subcode)*
[34]   Langcode ::= ISO639Code |  IanaCode |  UserCode
[35] ISO639Code ::= ([a-z] | [A-Z]) ([a-z] | [A-Z])
[36]   IanaCode ::= ('i' | 'I') '-' ([a-z] | [A-Z])+
[37]   UserCode ::= ('x' | 'X') '-' ([a-z] | [A-Z])+
[38]    Subcode ::= ([a-z] | [A-Z])+
```

The Langcode may be any of the following:

- a two-letter language code as defined by [ISO 639], "Codes for the representation of names of languages"
- a language identifier registered with the Internet Assigned Numbers Authority [IANA]; these begin with the prefix "i-" (or "I-")
- a language identifier assigned by the user, or agreed on between parties in private use; these must begin with the prefix "x-" or "X-" in order to ensure that they do not conflict with names later standardized or registered with IANA

There may be any number of Subcode segments; if the first subcode segment exists and the Subcode consists of two letters, then it must be a country code from [ISO 3166], "Codes for the representation of names of countries." If the first subcode consists of more than two letters, it must be a subcode for the language in question registered with IANA, unless the Langcode begins with the prefix "x-" or "X-".

It is customary to give the language code in lower case, and the country code (if any) in upper case. Note that these values, unlike other names in XML documents, are case insensitive.

For example:

```
<p xml:lang="en">The quick brown fox jumps over the lazy dog.</p>
<p xml:lang="en-GB">What colour is it?</p>
<p xml:lang="en-US">What color is it?</p>
<sp who="Faust" desc='leise' xml:lang="de">
  <l>Habe nun, ach! Philosophie,</l>
  <l>Juristerei, und Medizin</l>
  <l>und leider auch Theologie</l>
  <l>durchaus studiert mit heißem Bemüh'n.</l>
  </sp>
```

The intent declared with xml:lang is considered to apply to all attributes and content of the element where it is specified, unless overridden with an instance of xml:lang on another element within that content.

A simple declaration for xml:lang might take the form

```
xml:lang  NMTOKEN  #IMPLIED
```

but specific default values may also be given, if appropriate. In a collection of French poems for English students, with glosses and notes in English, the xml:lang attribute might be declared this way:

```
    <!ATTLIST poem   xml:lang NMTOKEN 'fr'>
    <!ATTLIST gloss  xml:lang NMTOKEN 'en'>
    <!ATTLIST note   xml:lang NMTOKEN 'en'>
```

3. Logical Structures

[Definition:] Each XML document contains one or more **elements**,(E) the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type,(U) identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications. Each attribute specification has a name and a value.

Element

```
[39] element ::= EmptyElemTag
               | STag content ETag [WFC: Element Type Match ]
                                   [VC: Element Valid ]
```

This specification does not constrain the semantics, use, or (beyond syntax) names of the element types and attributes, except that names beginning with a match to `(('X'|'x')('M'|'m')('L'|'l'))` are reserved for standardization in this or future versions of this specification.

Well-Formedness Constraint: Element Type Match
The Name in an element's end-tag must match the element type in the start-tag.

Validity Constraint: Element Valid
An element is valid if there is a declaration matching `elementdecl` where the Name matches the element type, and one of the following holds:

1. The declaration matches `EMPTY` and the element has no content.
2. The declaration matches `children` and the sequence of child elements belongs to the language generated by the regular expression in the content model, with optional white space (characters matching the nonterminal S) between each pair of child elements.
3. The declaration matches `Mixed` and the content consists of character data and child elements whose types match names in the content model.
4. The declaration matches `ANY`, and the types of any child elements have been declared.

3.1 Start-Tags, End-Tags, and Empty-Element Tags

[Definition:] The beginning of every non-empty XML element is marked by a **start-tag**.

Start-tag

```
[40]      STag ::= '<' Name (S Attribute)*   [WFC: Unique Att Spec ]
                   S? '>'
[41] Attribute ::= Name Eq AttValue          [VC: Attribute Value
                                               Type ]
                                             [WFC: No External
                                               Entity References ]
                                             [WFC: No < in Attribute
                                               Values ]
```

The Name in the start- and end-tags gives the element's **type**. [Definition:] The Name-AttValue pairs are referred to as the **attribute specifications** of the element, [Definition:] with the Name in each pair referred to as the **attribute name** and [Definition:] the content of the AttValue (the text between the ' or " delimiters) as the **attribute value**.

Well-Formedness Constraint: Unique Att Spec
No attribute name may appear more than once in the same start-tag or empty-element tag.

Validity Constraint: Attribute Value Type
The attribute must have been declared; the value must be of the type declared for it. (For attribute types, see "3.3 Attribute-List Declarations".)

Well-Formedness Constraint: No External Entity References
Attribute values cannot contain direct or indirect entity references to external entities.

Well-Formedness Constraint: No < in Attribute Values
The replacement text of any entity referred to directly or indirectly in an attribute value (other than "<") must not contain a <.

An example of a start-tag:

```
<termdef id="dt-dog" term="dog">
```

[Definition:] The end of every element that begins with a start-tag must be marked by an **end-tag** containing a name that echoes the element's type as given in the start-tag:

End-tag

```
[42] ETag ::= '</' Name S? '>'
```

An example of an end-tag:

```
</termdef>
```

[Definition:] The text between the start-tag and end-tag is called the element's **content**:

Content of Elements

```
[43] content ::= (element | CharData | Reference | CDSect | PI
                 | Comment)*
```

[Definition:] If an element is **empty**, it must be represented either by a start-tag immediately followed by an end-tag or by an empty-element tag. [Definition:] An **empty-element tag** takes a special form:

Tags for Empty Elements

```
[44] EmptyElemTag ::= '<' Name (S Attribute)* S?    [ WFC: Unique Att
                      '/>'                           Spec ]
```

Empty-element tags may be used for any element which has no content, whether or not it is declared using the keyword EMPTY. For interoperability, the empty-element tag must be used, and can only be used, for elements which are declared EMPTY.

Examples of empty elements:

```
<IMG align="left"
 src="http://www.w3.org/Icons/WWW/w3c_home" />
<br></br>
<br/>
```

3.2 Element Type Declarations

The element structure of an XML document may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element's content.

Element type declarations often constrain which element types can appear as children of the element. At user option, an XML processor may issue a warning when a declaration mentions an element type for which no declaration is provided, but this is not an error.

[Definition:] An **element type declaration** takes the form:

Element Type Declaration

```
[45]   elementdecl ::= '<!ELEMENT' S Name S [ VC: Unique Element
                       contentspec S? '>'      Type Declaration ]
[46] contentspec ::= 'EMPTY' | 'ANY'
                     | Mixed | children
```

where the Name gives the element type being declared.

Validity Constraint: Unique Element Type Declaration
No element type may be declared more than once.

Examples of element type declarations:

```
<!ELEMENT br EMPTY>
<!ELEMENT p (#PCDATA|emph)* >
<!ELEMENT %name.para; %content.para; >
<!ELEMENT container ANY>
```

3.2.1 Element Content

[Definition:] An element type has **element content** when elements of that type must contain only child elements (no character data), optionally separated by white space (characters matching the nonterminal S). In this case, the constraint includes a content model, a simple grammar governing the allowed types of the child elements and the order in which they are allowed to appear. The grammar is built on content particles (cps), which consist of names, choice lists of content particles, or sequence lists of content particles:

Element-content Models

```
[47] children ::= (choice | seq) ('?'
                  | '*' | '+')?
[48]       cp ::= (Name | choice | seq)
                  ('?' | '*' | '+')?
[49]   choice ::= '(' S? cp ( S? '|' S? cp [ VC: Proper Group/PE
                  )* S? ')'                    Nesting ]
[50]      seq ::= '(' S? cp ( S? ',' S? cp [ VC: Proper Group/PE
                  )* S? ')'                    Nesting ]
```

where each Name is the type of an element which may appear as a child. Any content particle in a choice list may appear in the element content at the location where the choice list appears in the grammar; content particles occurring in a sequence list must each appear in the element content in the order given in the list. The optional character following a name or list governs whether the element or the content particles in the list may occur one or more (+), zero or more (*), or zero or one times (?). The absence of such an operator means that the element or content particle must appear exactly once. This syntax and meaning are identical to those used in the productions in this specification.

The content of an element matches a content model if and only if it is possible to trace out a path through the content model, obeying the sequence, choice, and repetition operators and matching each element in the content against an element type in the content model. For compatibility, it is an error if an element in the document can match more than one occurrence of an element type in the content model. For more information, see "E. Deterministic Content

Models". U

Validity Constraint: Proper Group/PE Nesting
Parameter-entity replacement text must be properly nested with parenthetized groups. That is to say, if either of the opening or closing parentheses in a `choice`, `seq`, or `Mixed` construct is contained in the replacement text for a parameter entity, both must be contained in the same replacement text. T For interoperability, if a parameter-entity reference appears in a `choice`, `seq`, or `Mixed` construct, its replacement text should not be empty, and neither the first nor last non-blank character of the replacement text should be a connector (`|` or `,`).

Examples of element-content models:

```
<!ELEMENT spec (front, body, back?)>
<!ELEMENT div1 (head, (p | list | note)*, div2*)>
<!ELEMENT dictionary-body (%div.mix; | %dict.mix;)*>
```

3.2.2 Mixed Content

[Definition:] An element type has **mixed content** when elements of that type may contain character data, optionally interspersed with child elements. In this case, the types of the child elements may be constrained, but not their order or their number of occurrences: T

Mixed-content Declaration

```
[51] Mixed ::= '(' S? '#PCDATA' U (S? '|'
               S? Name)* S? ')*'
               | '(' S? '#PCDATA' S? ')'  [VC: Proper Group/PE
                                           Nesting ]
                                          [VC: No Duplicate Types ]
```

where the `Name`s give the types of elements that may appear as children.

Validity Constraint: No Duplicate Types
The same name must not appear more than once in a single mixed-content declaration.

Examples of mixed content declarations:

```
<!ELEMENT p (#PCDATA|a|ul|b|i|em)*>
<!ELEMENT p (#PCDATA | %font; | %phrase; | %special; | %form;)* >
<!ELEMENT b (#PCDATA)>
```

3.3 Attribute-List Declarations

Attributes are used to associate name-value pairs with elements. Attribute specifications may appear only within start-tags and empty-element tags; thus, the productions used to recognize them appear in "3.1 Start-Tags, End-Tags, and Empty-Element Tags". Attribute-list declarations may be used:

- To define the set of attributes pertaining to a given element type.
- To establish type constraints for these attributes.
- To provide default values for attributes.

[Definition:] **Attribute-list declarations** specify the name, data type, and default value (if any) of each attribute associated with a given element type: T

Attribute-list Declaration

```
[52] AttlistDecl ::= '<!ATTLIST' S Name AttDef* S? '>'
[53]      AttDef ::= S Name S AttType S DefaultDecl
```

The Name in the AttlistDecl rule is the type of an element. At user option, an XML processor may issue a warning if attributes are declared for an element type not itself declared, but this is not an error. The Name in the AttDef rule is the name of the attribute.

When more than one AttlistDecl is provided for a given element type, the contents of all those provided are merged. When more than one definition is provided for the same attribute of a given element type, the first declaration is binding and later declarations are ignored. For interoperability, writers of DTDs may choose to provide at most one attribute-list declaration for a given element type, at most one attribute definition for a given attribute name, and at least one attribute definition in each attribute-list declaration. For interoperability, an XML processor may at user option issue a warning when more than one attribute-list declaration is provided for a given element type, or more than one attribute definition is provided for a given attribute, but this is not an error.

3.3.1 Attribute Types

XML attribute types are of three kinds: a string type, a set of tokenized types, and enumerated types. The string type may take any literal string as a value; the tokenized types have varying lexical and semantic constraints, as noted:

Attribute Types

```
[54]       AttType ::= StringType | TokenizedType
                     | EnumeratedType
[55]    StringType ::= 'CDATA'
[56] TokenizedType ::= 'ID'                    [ VC: ID ]
                                               [ VC: One ID per
                                                 Element Type ]
                                               [ VC: ID
                                                 Attribute
                                                 Default ]
                     | 'IDREF'                 [ VC: IDREF ]
                     | 'IDREFS'                [ VC: IDREF ]
                     | 'ENTITY'                [ VC: Entity Name
                                                 ]
                     | 'ENTITIES'              [ VC: Entity Name
                                                 ]
                     | 'NMTOKEN'               [ VC: Name Token
                                                 ]
                     | 'NMTOKENS'              [ VC: Name Token
                                                 ]
```

Validity Constraint: ID
Values of type ID must match the Name production. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must uniquely identify the elements which bear them.

Validity Constraint: One ID per Element Type
No element type may have more than one ID attribute specified.

Validity Constraint: ID Attribute Default

An ID attribute must have a declared default of #IMPLIED or #REQUIRED.

Validity Constraint: IDREF
Values of type IDREF must match the Name production, and values of type IDREFS must match Names; each Name must match the value of an ID attribute on some element in the XML document; i.e. IDREF values must match the value of some ID attribute.

Validity Constraint: Entity Name
Values of type ENTITY must match the Name production, values of type ENTITIES must match Names; each Name must match the name of an unparsed entity declared in the DTD.

Validity Constraint: Name Token
Values of type NMTOKEN must match the Nmtoken production; values of type NMTOKENS must match Nmtokens.

[Definition:] **Enumerated attributes** can take one of a list of values provided in the declaration. There are two kinds of enumerated types:

Enumerated Attribute Types

```
[57] EnumeratedType ::= NotationType
                      | Enumeration
[58]   NotationType ::= 'NOTATION' S '(' S?     [VC: Notation
                      Name (S? '|' S? Name)*     Attributes ]
                      S? ')'
[59]    Enumeration ::= '(' S? Nmtoken (S? '|'  [VC: Enumeration ]
                      S? Nmtoken)* S? ')'
```

A NOTATION attribute identifies a notation, declared in the DTD with associated system and/or public identifiers, to be used in interpreting the element to which the attribute is attached.

Validity Constraint: Notation Attributes
Values of this type must match one of the notation names included in the declaration; all notation names in the declaration must be declared.

Validity Constraint: Enumeration
Values of this type must match one of the Nmtoken tokens in the declaration.

For interoperability, the same Nmtoken should not occur more than once in the enumerated attribute types of a single element type.

3.3.2 Attribute Defaults

An attribute declaration provides information on whether the attribute's presence is required, and if not, how an XML processor should react if a declared attribute is absent in a document.

Attribute Defaults

```
[60] DefaultDecl ::= '#REQUIRED' | '#IMPLIED'
                   | (('#FIXED' S)? AttValue) [ VC: Required
                                                Attribute ]
                                              [ VC: Attribute
                                                Default Legal ]
                                              [ WFC: No < in
                                                Attribute Values ]
                                              [ VC: Fixed Attribute
                                                Default ]
```

In an attribute declaration, #REQUIRED means that the attribute must always be provided, #IMPLIED that no default value is provided. [Definition:] If the declaration is neither #REQUIRED nor #IMPLIED, then the AttValue value contains the declared **default** value; the #FIXED keyword states that the attribute must always have the default value. If a default value is declared, when an XML processor encounters an omitted attribute, it is to behave as though the attribute were present with the declared default value.

Validity Constraint: Required Attribute
If the default declaration is the keyword #REQUIRED, then the attribute must be specified for all elements of the type in the attribute-list declaration.

Validity Constraint: Attribute Default Legal
The declared default value must meet the lexical constraints of the declared attribute type.

Validity Constraint: Fixed Attribute Default
If an attribute has a default value declared with the #FIXED keyword, instances of that attribute must match the default value.

Examples of attribute-list declarations:

```
<!ATTLIST termdef
          id      ID      #REQUIRED
          name    CDATA   #IMPLIED>
<!ATTLIST list
          type    (bullets|ordered|glossary)  "ordered">
<!ATTLIST form
          method  CDATA   #FIXED "POST">
```

3.3.3 Attribute-Value Normalization

Before the value of an attribute is passed to the application or checked for validity, the XML processor must normalize it as follows:

- a character reference is processed by appending the referenced character to the attribute value
- an entity reference is processed by recursively processing the replacement text of the entity
- a whitespace character (#x20, #xD, #xA, #x9) is processed by appending #x20 to the normalized value, except that only a single #x20 is appended for a "#xD#xA" sequence that is part of an external parsed entity or the literal entity value of an internal parsed entity
- other characters are processed by appending them to the normalized value

If the declared value is not CDATA, then the XML processor must further process the normalized attribute value by discarding any leading and trailing space (#x20) characters, and by replacing sequences of space (#x20) characters by a single space (#x20) character.

All attributes for which no declaration has been read should be treated by a non-validating parser as if declared

CDATA.

3.4 Conditional Sections

[Definition:] **Conditional sections** are portions of the document type declaration external subset which are included in, or excluded from, the logical structure of the DTD based on the keyword which governs them.

Conditional Section		
[61]	conditionalSect	::= includeSect | ignoreSect
[62]	includeSect	::= '<![' S? 'INCLUDE' S? '[' extSubsetDecl ']]>'
[63]	ignoreSect	::= '<![' S? 'IGNORE' S? '[' ignoreSectContents* ']]>'
[64]	ignoreSectContents	::= Ignore ('<![' ignoreSectContents ']]>' Ignore)*
[65]	Ignore	::= Char* - (Char* ('<![' | ']]>') Char*)

Like the internal and external DTD subsets, a conditional section may contain one or more complete declarations, comments, processing instructions, or nested conditional sections, intermingled with white space.

If the keyword of the conditional section is `INCLUDE`, then the contents of the conditional section are part of the DTD. If the keyword of the conditional section is `IGNORE`, then the contents of the conditional section are not logically part of the DTD. Note that for reliable parsing, the contents of even ignored conditional sections must be read in order to detect nested conditional sections and ensure that the end of the outermost (ignored) conditional section is properly detected. If a conditional section with a keyword of `INCLUDE` occurs within a larger conditional section with a keyword of `IGNORE`, both the outer and the inner conditional sections are ignored.

If the keyword of the conditional section is a parameter-entity reference, the parameter entity must be replaced by its content before the processor decides whether to include or ignore the conditional section.

An example:

```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >

<![%draft;[
<!ELEMENT book (comments*, title, body, supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>
```

4. Physical Structures

[Definition:] An XML document may consist of one or many storage units. These are called **entities**; they all have **content** and are all (except for the document entity, see below, and the external DTD subset) identified by **name**. Each XML document has one entity called the document entity, which serves as the starting point for the XML processor and may contain the whole document.

Entities may be either parsed or unparsed. [Definition:] A **parsed entity's** contents are referred to as its replacement text; this text is considered an integral part of the document.

[Definition:] An **unparsed entity** is a resource whose contents may or may not be text, and if text, may not be XML.

Each unparsed entity has an associated notation, identified by name. Beyond a requirement that an XML processor make the identifiers for the entity and notation available to the application, XML places no constraints on the contents of unparsed entities.

Parsed entities are invoked by name using entity references; unparsed entities by name, given in the value of `ENTITY` or `ENTITIES` attributes.

[Definition:] **General entities** are entities for use within the document content. In this specification, general entities are sometimes referred to with the unqualified term *entity* when this leads to no ambiguity. [Definition:] Parameter entities are parsed entities for use within the DTD. These two types of entities use different forms of reference and are recognized in different contexts. Furthermore, they occupy different namespaces; a parameter entity and a general entity with the same name are two distinct entities. Ⓣ

4.1 Character and Entity References

[Definition:] A **character reference** refers to a specific character in the ISO/IEC 10646 character set, for example one not directly accessible from available input devices.Ⓣ

Character Reference

```
[66] CharRef ::= '&#' [0-9]+ ';'
                 | '&#x' [0-9a-fA-F]+ ';' [ WFC: Legal Character ]
```

Well-Formedness Constraint: Legal Character
Characters referred to using character references must match the production for Char.

If the character reference begins with "`&#x`", the digits and letters up to the terminating `;` provide a hexadecimal representation of the character's code point in ISO/IEC 10646.Ⓣ If it begins just with "`&#`", the digits up to the terminating `;` provide a decimal representation of the character's code point.

[Definition:] An **entity reference** refers to the content of a named entity. [Definition:] References to parsed general entities use ampersand (`&`) and semicolon (`;`) as delimiters.Ⓤ [Definition:] **Parameter-entity references** use percent-sign (`%`) and semicolon (`;`) as delimiters.

Entity Reference

```
[67]   Reference ::= EntityRef | CharRef
[68]   EntityRef ::= '&' Name ';'        [ WFC: Entity Declared ]
                                         [ VC: Entity Declared ]
                                         [ WFC: Parsed Entity ]
                                         [ WFC: No Recursion ]
[69] PEReference ::= '%' Name ';'        [ VC: Entity Declared ]
                                         [ WFC: No Recursion ]
                                         [ WFC: In DTD ]
```

Well-Formedness Constraint: Entity Declared
In a documentⓉ without any DTD, a document with only an internal DTD subset which contains no parameter entity references, or a document with "`standalone='yes'`", the Name given in the entity reference must match that in an entity declaration, except that well-formed documents need not declare any of the following entities: `amp`, `lt`, `gt`, `apos`, `quot`. The declaration of a parameter entity must precede any reference to it. Similarly, the declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration. Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be

declared is a well-formedness constraint only if standalone='yes'.

Validity Constraint: Entity Declared
In a document (T) with an external subset or external parameter entities with "`standalone='no'`", the `Name` given in the entity reference must match that in an entity declaration. For interoperability, valid documents should declare the entities `amp`, `lt`, `gt`, `apos`, `quot`, in the form specified in "4.6 Predefined Entities". The declaration of a parameter entity must precede any reference to it. Similarly, the declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration.

Well-Formedness Constraint: Parsed Entity
An entity reference must not contain the name of an unparsed entity. (T) Unparsed entities may be referred to only in attribute values declared to be of type `ENTITY` or `ENTITIES`.

Well-Formedness Constraint: No Recursion
A parsed entity must not contain a recursive reference to itself, either directly or indirectly. (T)

Well-Formedness Constraint: In DTD
Parameter-entity references may only appear in the DTD. (T)

Examples of character and entity references:

```
Type <key>less-than</key> (&#x3C;) to save options.
This document was prepared on &docdate; and
is classified &security-level;.
```

Example of a parameter-entity reference:

```
<!-- declare the parameter entity "ISOLat2"... -->
<!ENTITY % ISOLat2
         SYSTEM "http://www.xml.com/iso/isolat2-xml.entities" >
<!-- ... now reference it. -->
%ISOLat2;
```

4.2 Entity Declarations

[Definition:] Entities are declared thus:

Entity Declaration

```
[70] EntityDecl ::= GEDecl | PEDecl
[71]     GEDecl ::= '<!ENTITY' (U) S Name S EntityDef S? '>'
[72]     PEDecl ::= '<!ENTITY' S '%' S Name S PEDef S? '>'
[73]  EntityDef ::= EntityValue | (ExternalID NDataDecl?)
[74]      PEDef ::= EntityValue | ExternalID
```

The `Name` identifies the entity in an entity reference or, in the case of an unparsed entity, in the value of an `ENTITY` or `ENTITIES` attribute. If the same entity is declared more than once, the first declaration encountered is binding; at user option, an XML processor may issue a warning if entities are declared multiple times. (T)

4.2.1 Internal Entities

[Definition:] If the entity definition is an `EntityValue`, the defined entity is called an **internal entity**. There is no separate physical storage object, and the content of the entity is given in the declaration. (T) Note that some

processing of entity and character references in the literal entity value may be required to produce the correct replacement text: see "4.5 Construction of Internal Entity Replacement Text".

An internal entity is a parsed entity.

Example of an internal entity declaration:

```
<!ENTITY Pub-Status "This is a pre-release of the
 specification.">
```

4.2.2 External Entities

[Definition:] If the entity is not internal, it is an **external entity**, declared as follows:

External Entity Declaration	
[75] ExternalID	::= 'SYSTEM' S SystemLiteral | 'PUBLIC' S PubidLiteral S SystemLiteral
[76] NDataDecl	::= S 'NDATA' S Name [VC: Notation Declared]

If the NDataDecl is present, this is a general unparsed entity; otherwise it is a parsed entity.

Validity Constraint: Notation Declared
The Name must match the declared name of a notation.

[Definition:] The SystemLiteral is called the entity's **system identifier**. It is a URI, which may be used to retrieve the entity. Note that the hash mark (#) and fragment identifier frequently used with URIs are not, formally, part of the URI itself; an XML processor may signal an error if a fragment identifier is given as part of a system identifier. Unless otherwise provided by information outside the scope of this specification (e.g. a special XML element type defined by a particular DTD, or a processing instruction defined by a particular application specification), relative URIs are relative to the location of the resource within which the entity declaration occurs. A URI might thus be relative to the document entity, to the entity containing the external DTD subset, or to some other external parameter entity.

An XML processor should handle a non-ASCII character in a URI by representing the character in UTF-8 as one or more bytes, and then escaping these bytes with the URI escaping mechanism (i.e., by converting each byte to %HH, where HH is the hexadecimal notation of the byte value).

[Definition:] In addition to a system identifier, an external identifier may include a **public identifier**. An XML processor attempting to retrieve the entity's content may use the public identifier to try to generate an alternative URI. If the processor is unable to do so, it must use the URI specified in the system literal. Before a match is attempted, all strings of white space in the public identifier must be normalized to single space characters (#x20), and leading and trailing white space must be removed.

Examples of external entity declarations:

```
<!ENTITY open-hatch
         SYSTEM "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY open-hatch
         PUBLIC "-//Textuality//TEXT Standard open-hatch boilerplate//EN"
         "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY hatch-pic
         SYSTEM "../grafix/OpenHatch.gif"
         NDATA gif >
```

4.3 Parsed Entities

4.3.1 The Text Declaration

External parsed entities may each begin with a **text declaration**.

Text Declaration
[77] TextDecl ::= '<?xml' VersionInfo? EncodingDecl S? '?>'

The text declaration must be provided literally, not by reference to a parsed entity. No text declaration may appear at any position other than the beginning of an external parsed entity.

4.3.2 Well-Formed Parsed Entities

The document entity is well-formed if it matches the production labeled document. An external general parsed entity is well-formed if it matches the production labeled extParsedEnt. An external parameter entity is well-formed if it matches the production labeled extPE.

Well-Formed External Parsed Entity
[78] extParsedEnt ::= TextDecl? content [79] extPE ::= TextDecl? extSubsetDecl

An internal general parsed entity is well-formed if its replacement text matches the production labeled content. All internal parameter entities are well-formed by definition.

A consequence of well-formedness in entities is that the logical and physical structures in an XML document are properly nested; no start-tag, end-tag, empty-element tag, element, comment, processing instruction, character reference, or entity reference can begin in one entity and end in another.

4.3.3 Character Encoding in Entities

Each external parsed entity in an XML document may use a different encoding for its characters. All XML processors must be able to read entities in either UTF-8 or UTF-16.

Entities encoded in UTF-16 must begin with the Byte Order Mark described by ISO/IEC 10646 Annex E and Unicode Appendix B (the ZERO WIDTH NO-BREAK SPACE character, #xFEFF). This is an encoding signature, not part of either the markup or the character data of the XML document. XML processors must be able to use this character to differentiate between UTF-8 and UTF-16 encoded documents.

Although an XML processor is required to read only entities in the UTF-8 and UTF-16 encodings, it is recognized that other encodings are used around the world, and it may be desired for XML processors to read entities that use them. Parsed entities which are stored in an encoding other than UTF-8 or UTF-16 must begin with a text declaration containing an encoding declaration:

Encoding Declaration

```
[80] EncodingDecl ::= S 'encoding' Eq ('"'
                      EncName '"' |  "'"
                      EncName "'" )
[81]      EncName ::= [A-Za-z] ([A-Za-z0-9._]   /* Encoding name
                      | '-')*                    contains only
                                                 Latin characters
                                                 */
```

In the document entity, the encoding declaration is part of the XML declaration. The `EncName` is the name of the encoding used.

In an encoding declaration, the values "`UTF-8`", "`UTF-16`", "`ISO-10646-UCS-2`", and "`ISO-10646-UCS-4`" should be used for the various encodings and transformations of Unicode / ISO/IEC 10646, the values "`ISO-8859-1`", "`ISO-8859-2`", ... "`ISO-8859-9`" should be used for the parts of ISO 8859, and the values "`ISO-2022-JP`", "`Shift_JIS`", and "`EUC-JP`" should be used for the various encoded forms of JIS X-0208-1997. XML processors may recognize other encodings; it is recommended that character encodings registered (as *charsets*) with the Internet Assigned Numbers Authority [IANA], other than those just listed, should be referred to using their registered names. Note that these registered names are defined to be case-insensitive, so processors wishing to match against them should do so in a case-insensitive way.(T)

In the absence of information provided by an external transport protocol (e.g. HTTP or MIME), it is an error(T) for an entity including an encoding declaration to be presented to the XML processor in an encoding other than that named in the declaration, for an encoding declaration to occur other than at the beginning of an external entity, or for an entity which begins with neither a Byte Order Mark nor an encoding declaration to use an encoding other than UTF-8. Note that since ASCII is a subset of UTF-8, ordinary ASCII entities do not strictly need an encoding declaration.(T)

It is a fatal error when an XML processor encounters an entity with an encoding that it is unable to process.

Examples of encoding declarations:

```
<?xml encoding='UTF-8'?>
<?xml encoding='EUC-JP'?>
```

4.4 XML Processor Treatment of Entities and References

The table below(U) summarizes the contexts in which character references, entity references, and invocations of unparsed entities might appear and the required behavior of an XML processor in each case. The labels in the leftmost column describe the recognition context:

Reference in Content

as a reference anywhere after the start-tag and before the end-tag of an element; corresponds to the nonterminal `content`.(E)

Reference in Attribute Value

as a reference within either the value of an attribute in a start-tag, or a default value in an attribute declaration; corresponds to the nonterminal `AttValue`.(E)

Occurs as Attribute Value

as a `Name`, not a reference, appearing either as the value of an attribute which has been declared as type `ENTITY`, or as one of the space-separated tokens in the value of an attribute which has been declared as type `ENTITIES`.(E)

Reference in Entity Value

as a reference within a parameter or internal entity's literal entity value in the entity's declaration; corresponds to the nonterminal `EntityValue`. (E)

Reference in DTD

as a reference within either the internal or external subsets of the DTD, but outside of an `EntityValue` or `AttValue`. (E)

	Entity Type				Character
	Parameter	Internal General	External Parsed General	Unparsed	
Reference in Content	Not recognized	Included	Included if validating	Forbidden	Included
Reference in Attribute Value	Not recognized	Included in literal	Forbidden	Forbidden	Included
Occurs as Attribute Value	Not recognized	Forbidden	Forbidden	Notify	Not recognized
Reference in EntityValue	Included in literal	Bypassed	Bypassed	Forbidden	Included
Reference in DTD	Included as PE	Forbidden	Forbidden	Forbidden	Forbidden

4.4.1 Not Recognized

Outside the DTD, the % character has no special significance; thus, what would be parameter entity references in the DTD are not recognized as markup in `content`. Similarly, the names of unparsed entities are not recognized except when they appear in the value of an appropriately declared attribute.

4.4.2 Included

[Definition:] An entity is **included** when its replacement text is retrieved and processed, in place of the reference itself, as though it were part of the document at the location the reference was recognized. The replacement text may contain both character data and (except for parameter entities) markup, which must be recognized in the usual way, except that the replacement text of entities used to escape markup delimiters (the entities `amp`, `lt`, `gt`, `apos`, `quot`) is always treated as data. (T) (The string "`AT&T;`" expands to "`AT&T;`" and the remaining ampersand is not recognized as an entity-reference delimiter.) A character reference is **included** when the indicated character is processed in place of the reference itself.

4.4.3 Included If Validating

When an XML processor recognizes a reference to a parsed entity, in order to validate the document, the processor must include its replacement text. If the entity is external, and the processor is not attempting to validate the XML document, the processor may, but need not, include the entity's replacement text. If a non-validating parser does not include the replacement text, it must inform the application that it recognized, but did not read, the entity.

This rule is based on the recognition that the automatic inclusion provided by the SGML and XML entity mechanism, primarily designed to support modularity in authoring, is not necessarily appropriate for other applications, in particular document browsing. (H) Browsers, for example, when encountering an external parsed entity reference, might choose to provide a visual indication of the entity's presence and retrieve it for display only

on demand.

4.4.4 Forbidden

The following are forbidden, and constitute fatal errors:

- the appearance of a reference to an unparsed entity.
- the appearance of any character or general-entity reference in the DTD except within an `EntityValue` or `AttValue`.(H)
- a reference to an external entity in an attribute value.(H)

4.4.5 Included in Literal

When an entity reference appears in an attribute value, or a parameter entity reference appears in a literal entity value, its replacement text is processed in place of the reference itself as though it were part of the document at the location the reference was recognized, except that a single or double quote character in the replacement text is always treated as a normal data character and will not terminate the literal. For example, this is well-formed:(T)

```
<!ENTITY % YN '"Yes"' >
<!ENTITY WhatHeSaid "He said &YN;" >
```

while this is not:

```
<!ENTITY EndAttr "27'" >
<element attribute='a-&EndAttr;>
```

4.4.6 Notify

When the name of an unparsed entity appears as a token in the value of an attribute of declared type `ENTITY` or `ENTITIES`, a validating processor must inform the application of the system and public (if any) identifiers for both the entity and its associated notation.(T)

4.4.7 Bypassed

When a general entity reference appears in the `EntityValue` in an entity declaration, it is bypassed and left as is. (T)

4.4.8 Included as PE

Just as with external parsed entities, parameter entities need only be included if validating. When a parameter-entity reference is recognized in the DTD and included, its replacement text is enlarged by the attachment of one leading and one following space (#x20) character; the intent is to constrain the replacement text of parameter entities to contain an integral number of grammatical tokens in the DTD. (U)

4.5 Construction of Internal Entity Replacement Text

In discussing the treatment of internal entities, it is useful to distinguish two forms of the entity's value. [Definition:] The **literal entity value** is the quoted string actually present in the entity declaration, corresponding to the non-terminal `EntityValue`. [Definition:] The **replacement text** is the content of the entity, after replacement of character references and parameter-entity references.

The literal entity value as given in an internal entity declaration (`EntityValue`) may contain character, parameter-entity, and general-entity references. Such references must be contained entirely within the literal entity value. The actual replacement text that is included as described above must contain the *replacement text* of any

parameter entities referred to, and must contain the character referred to, in place of any character references in the literal entity value; however, general-entity references must be left as-is, unexpanded. For example, given the following declarations:

```
<!ENTITY % pub    "&#xc9;ditions Gallimard" >
<!ENTITY   rights "All rights reserved" >
<!ENTITY   book   "La Peste: Albert Camus,
&#xA9; 1947 %pub;. &rights;" >
```

then the replacement text for the entity "book" is:

```
La Peste: Albert Camus,
© 1947 Éditions Gallimard. &rights;
```

The general-entity reference "&rights;" would be expanded should the reference "&book;" appear in the document's content or an attribute value.

These simple rules may have complex interactions; for a detailed discussion of a difficult example, see "D. Expansion of Entity and Character References".

4.6 Predefined Entities

[Definition:] Entity and character references can both be used to **escape** the left angle bracket, ampersand, and other delimiters. A set of general entities (amp, lt, gt, apos, quot) is specified for this purpose. Numeric character references may also be used; they are expanded immediately when recognized and must be treated as character data, so the numeric character references "<" and "&" may be used to escape < and & when they occur in character data.

All XML processors must recognize these entities whether they are declared or not. For interoperability, valid XML documents should declare these entities, like any others, before using them. If the entities in question are declared, they must be declared as internal entities whose replacement text is the single character being escaped or a character reference to that character, as shown below.

```
<!ENTITY lt     "&#38;#60;">
<!ENTITY gt     "&#62;">
<!ENTITY amp    "&#38;#38;">
<!ENTITY apos   "&#39;">
<!ENTITY quot   "&#34;">
```

Note that the < and & characters in the declarations of "lt" and "amp" are doubly escaped to meet the requirement that entity replacement be well-formed.

4.7 Notation Declarations

[Definition:] **Notations** identify by name the format of unparsed entities, the format of elements which bear a notation attribute, or the application to which a processing instruction is addressed.

[Definition:] **Notation declarations** provide a name for the notation, for use in entity and attribute-list declarations and in attribute specifications, and an external identifier for the notation which may allow an XML processor or its client application to locate a helper application capable of processing data in the given notation.

Notation Declarations

```
[82] NotationDecl ::= '<!NOTATION' S Name S (ExternalID |
                      PublicID) S? '>'
[83]     PublicID ::= 'PUBLIC' S PubidLiteral
```

XML processors must provide applications with the name and external identifier(s) of any notation declared and referred to in an attribute value, attribute definition, or entity declaration. They may additionally resolve the external identifier into the system identifier, file name, or other information needed to allow the application to call a processor for data in the notation described. (It is not an error, however, for XML documents to declare and refer to notations for which notation-specific applications are not available on the system where the XML processor or application is running.)

4.8 Document Entity

[Definition:] The **document entity** serves as the root of the entity tree and a starting-point for an XML processor. This specification does not specify how the document entity is to be located by an XML processor; unlike other entities, the document entity has no name and might well appear on a processor input stream without any identification at all.

5. Conformance

5.1 Validating and Non-Validating Processors

Conforming XML processors fall into two classes: validating and non-validating.

Validating and non-validating processors alike must report violations of this specification's well-formedness constraints in the content of the document entity and any other parsed entities that they read.

[Definition:] **Validating processors** must report violations of the constraints expressed by the declarations in the DTD, and failures to fulfill the validity constraints given in this specification. To accomplish this, validating XML processors must read and process the entire DTD and all external parsed entities referenced in the document.

Non-validating processors are required to check only the document entity, including the entire internal DTD subset, for well-formedness. [Definition:] While they are not required to check the document for validity, they are required to **process** all the declarations they read in the internal DTD subset and in any parameter entity that they read, up to the first reference to a parameter entity that they do *not* read; that is to say, they must use the information in those declarations to normalize attribute values, include the replacement text of internal entities, and supply default attribute values. They must not process entity declarations or attribute-list declarations encountered after a reference to a parameter entity that is not read, since the entity may have contained overriding declarations.

5.2 Using XML Processors

The behavior of a validating XML processor is highly predictable; it must read every piece of a document and report all well-formedness and validity violations. Less is required of a non-validating processor; it need not read any part of the document other than the document entity. This has two effects that may be important to users of XML processors:

- Certain well-formedness errors, specifically those that require reading external entities, may not be detected by a non-validating processor. Examples include the constraints entitled Entity Declared, Parsed Entity, and No Recursion, as well as some of the cases described as forbidden in "4.4 XML Processor Treatment of Entities and References".
- The information passed from the processor to the application may vary, depending on whether the processor reads parameter and external entities. For example, a non-validating processor may not normalize attribute

values, include the replacement text of internal entities, or supply default attribute values, where doing so depends on having read declarations in external or parameter entities.

For maximum reliability in interoperating between different XML processors, applications which use non-validating processors should not rely on any behaviors not required of such processors. Applications which require facilities such as the use of default attributes or internal entities which are declared in external entities should use validating XML processors.

6. Notation

The formal grammar of XML is given in this specification using a simple Extended Backus-Naur Form (EBNF) notation. Each rule in the grammar defines one symbol, in the form

```
symbol ::= expression
```

Symbols are written with an initial capital letter if they are defined by a regular expression, or with an initial lower case letter otherwise. Literal strings are quoted.

Within the expression on the right-hand side of a rule, the following expressions are used to match strings of one or more characters:

#xN

where N is a hexadecimal integer, the expression matches the character in ISO/IEC 10646 whose canonical (UCS-4) code value, when interpreted as an unsigned binary number, has the value indicated. The number of leading zeros in the #xN form is insignificant; the number of leading zeros in the corresponding code value is governed by the character encoding in use and is not significant for XML.

[a-zA-Z], [#xN-#xN]

matches any character with a value in the range(s) indicated (inclusive).

[^a-z], [^#xN-#xN]

matches any character with a value *outside* the range indicated.

[^abc], [^#xN#xN#xN]

matches any character with a value not among the characters given.

"string"

matches a literal string matching that given inside the double quotes.

'string'

matches a literal string matching that given inside the single quotes.

These symbols may be combined to match more complex patterns as follows, where A and B represent simple expressions:

(expression)

expression is treated as a unit and may be combined as described in this list.

A?

matches A or nothing; optional A.

A B

matches A followed by B.

A | B

matches A or B but not both.

A - B

matches any string that matches A but does not match B.

A+

matches one or more occurrences of A.

A*

matches zero or more occurrences of A.

Other notations used in the productions are:

/* ... */

comment.

[wfc: ...]

well-formedness constraint; this identifies by name a constraint on well-formed documents associated with a production.

[vc: ...]

validity constraint; this identifies by name a constraint on valid documents associated with a production.

Appendices

A. References

A.1 Normative References

IANA

(Internet Assigned Numbers Authority) *Official Names for Character Sets*, ed. Keld Simonsen et al. See ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

IETF RFC 1766

IETF (Internet Engineering Task Force). *RFC 1766: Tags for the Identification of Languages*, ed. H. Alvestrand. 1995.

ISO 639

(International Organization for Standardization). *ISO 639:1988 (E). Code for the representation of names of languages.* [Geneva]: International Organization for Standardization, 1988.

ISO 3166

(International Organization for Standardization). *ISO 3166-1:1997 (E). Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes* [Geneva]: International Organization for Standardization, 1997.

ISO/IEC 10646

ISO (International Organization for Standardization). *ISO/IEC 10646-1993 (E). Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane.* [Geneva]: International Organization for Standardization, 1993 (plus amendments AM 1 through AM 7).

Unicode

The Unicode Consortium. *The Unicode Standard, Version 2.0.* Reading, Mass.: Addison-Wesley Developers Press, 1996.

A.2 Other References

Aho/Ullman

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading: Addison-Wesley, 1986, rpt. corr. 1988.

Berners-Lee et al.

Berners-Lee, T., R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax and*

Semantics. 1997. (Work in progress; see updates to RFC1738.)

Brüggemann-Klein

Brüggemann-Klein, Anne. *Regular Expressions into Finite Automata*. Extended abstract in I. Simon, Hrsg., LATIN 1992, S. 97-98. Springer-Verlag, Berlin 1992. Full Version in Theoretical Computer Science 120: 197-213, 1993.

Brüggemann-Klein and Wood

Brüggemann-Klein, Anne, and Derick Wood. *Deterministic Regular Languages*. Universität Freiburg, Institut für Informatik, Bericht 38, Oktober 1991.

Clark

James Clark. Comparison of SGML and XML. See http://www.w3.org/TR/NOTE-sgml-xml-971215.

IETF RFC1738

IETF (Internet Engineering Task Force). *RFC 1738: Uniform Resource Locators (URL)*, ed. T. Berners-Lee, L. Masinter, M. McCahill. 1994.

IETF RFC1808

IETF (Internet Engineering Task Force). *RFC 1808: Relative Uniform Resource Locators*, ed. R. Fielding. 1995.

IETF RFC2141

IETF (Internet Engineering Task Force). *RFC 2141: URN Syntax*, ed. R. Moats. 1997.

ISO 8879

ISO (International Organization for Standardization). *ISO 8879:1986(E). Information processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML).* First edition -- 1986-10-15. [Geneva]: International Organization for Standardization, 1986.

ISO/IEC 10744

ISO (International Organization for Standardization). *ISO/IEC 10744-1992 (E). Information technology -- Hypermedia/Time-based Structuring Language (HyTime).* [Geneva]: International Organization for Standardization, 1992. *Extended Facilities Annexe.* [Geneva]: International Organization for Standardization, 1996.

B. Character Classes

Following the characteristics defined in the Unicode standard, characters are classed as base characters (among others, these contain the alphabetic characters of the Latin alphabet, without diacritics), ideographic characters, and combining characters (among others, this class contains most diacritics); these classes combine to form the class of letters. Digits and extenders are also distinguished.

Characters
[84] Letter ::= BaseChar | Ideographic

```
[85]      BaseChar ::= [#x0041-#x005A] | [#x0061-#x007A]Ⓤ
                     | [#x00C0-#x00D6] | [#x00D8-#x00F6]
                     | [#x00F8-#x00FF]Ⓤ | [#x0100-#x0131]
                     | [#x0134-#x013E] | [#x0141-#x0148]
                     | [#x014A-#x017E] | [#x0180-#x01C3]
                     | [#x01CD-#x01F0] | [#x01F4-#x01F5]
                     | [#x01FA-#x0217]Ⓤ | [#x0250-#x02A8]Ⓤ
                     | [#x02BB-#x02C1]Ⓤ | #x0386
                     | [#x0388-#x038A] | #x038C | [#x038E-#x03A1]
                     | [#x03A3-#x03CE] | [#x03D0-#x03D6] | #x03DA
                     | #x03DC | #x03DE | #x03E0 | [#x03E2-#x03F3]
                      | [#x0401-#x040C] | [#x040E-#x044F]
                    Ⓤ
                     | [#x0451-#x045C] | [#x045E-#x0481]
                     | [#x0490-#x04C4] | [#x04C7-#x04C8]
                     | [#x04CB-#x04CC] | [#x04D0-#x04EB]
                     | [#x04EE-#x04F5] | [#x04F8-#x04F9]Ⓤ
                     | [#x0531-#x0556] | #x0559 | [#x0561-#x0586]
                      | [#x05D0-#x05EA] | [#x05F0-#x05F2]Ⓤ
                    Ⓤ
                     | [#x0621-#x063A] | [#x0641-#x064A]
                     | [#x0671-#x06B7] | [#x06BA-#x06BE]
                     | [#x06C0-#x06CE] | [#x06D0-#x06D3] | #x06D5
                     | [#x06E5-#x06E6]Ⓤ | [#x0905-#x0939]
                     | #x093D | [#x0958-#x0961]Ⓤ
                     | [#x0985-#x098C] | [#x098F-#x0990]
                     | [#x0993-#x09A8] | [#x09AA-#x09B0] | #x09B2
                     | [#x09B6-#x09B9] | [#x09DC-#x09DD]
                     | [#x09DF-#x09E1] | [#x09F0-#x09F1]Ⓤ
                     | [#x0A05-#x0A0A] | [#x0A0F-#x0A10]
                     | [#x0A13-#x0A28] | [#x0A2A-#x0A30]
                     | [#x0A32-#x0A33] | [#x0A35-#x0A36]
                     | [#x0A38-#x0A39] | [#x0A59-#x0A5C] | #x0A5E
                     | [#x0A72-#x0A74]Ⓤ | [#x0A85-#x0A8B]
                     | #x0A8D | [#x0A8F-#x0A91] | [#x0A93-#x0AA8]
                     | [#x0AAA-#x0AB0] | [#x0AB2-#x0AB3]
                     | [#x0AB5-#x0AB9] | #x0ABD | #x0AE0Ⓤ
                     | [#x0B05-#x0B0C] | [#x0B0F-#x0B10]
                     | [#x0B13-#x0B28] | [#x0B2A-#x0B30]
                     | [#x0B32-#x0B33] | [#x0B36-#x0B39] | #x0B3D
                     | [#x0B5C-#x0B5D] | [#x0B5F-#x0B61]Ⓤ
                     | [#x0B85-#x0B8A] | [#x0B8E-#x0B90]
                     | [#x0B92-#x0B95] | [#x0B99-#x0B9A] | #x0B9C
                     | [#x0B9E-#x0B9F] | [#x0BA3-#x0BA4]
                     | [#x0BA8-#x0BAA] | [#x0BAE-#x0BB5]
                     | [#x0BB7-#x0BB9]Ⓤ | [#x0C05-#x0C0C]
                     | [#x0C0E-#x0C10] | [#x0C12-#x0C28]
                     | [#x0C2A-#x0C33] | [#x0C35-#x0C39]
                     | [#x0C60-#x0C61]Ⓤ | [#x0C85-#x0C8C]
                     | [#x0C8E-#x0C90] | [#x0C92-#x0CA8]
                     | [#x0CAA-#x0CB3] | [#x0CB5-#x0CB9] | #x0CDE
                     | [#x0CE0-#x0CE1]Ⓤ | [#x0D05-#x0D0C]
                     | [#x0D0E-#x0D10] | [#x0D12-#x0D28]
```

```
                      | [#x0D2A-#x0D39] | [#x0D60-#x0D61]Ⓤ
                      | [#x0E01-#x0E2E] | #x0E30 | [#x0E32-#x0E33]
                      | [#x0E40-#x0E45]Ⓤ | [#x0E81-#x0E82]
                      | #x0E84 | [#x0E87-#x0E88] | #x0E8A | #x0E8D
                      | [#x0E94-#x0E97] | [#x0E99-#x0E9F]
                      | [#x0EA1-#x0EA3] | #x0EA5 | #x0EA7
                      | [#x0EAA-#x0EAB] | [#x0EAD-#x0EAE] | #x0EB0
                      | [#x0EB2-#x0EB3] | #x0EBD | [#x0EC0-#x0EC4]
                       | [#x0F40-#x0F47] | [#x0F49-#x0F69]Ⓤ
                     Ⓤ
                      | [#x10A0-#x10C5] | [#x10D0-#x10F6]Ⓤ
                      | #x1100 | [#x1102-#x1103] | [#x1105-#x1107]
                      | #x1109 | [#x110B-#x110C] | [#x110E-#x1112]
                      | #x113C | #x113E | #x1140 | #x114C | #x114E
                      | #x1150 | [#x1154-#x1155] | #x1159
                      | [#x115F-#x1161] | #x1163 | #x1165 | #x1167
                      | #x1169 | [#x116D-#x116E] | [#x1172-#x1173]
                      | #x1175 | #x119E | #x11A8 | #x11AB
                      | [#x11AE-#x11AF] | [#x11B7-#x11B8] | #x11BA
                      | [#x11BC-#x11C2] | #x11EB | #x11F0 | #x11F9
                       | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9]Ⓤ
                     Ⓤ
                      | [#x1F00-#x1F15] | [#x1F18-#x1F1D]
                      | [#x1F20-#x1F45] | [#x1F48-#x1F4D]
                      | [#x1F50-#x1F57] | #x1F59 | #x1F5B | #x1F5D
                      | [#x1F5F-#x1F7D] | [#x1F80-#x1FB4]
                      | [#x1FB6-#x1FBC] | #x1FBE | [#x1FC2-#x1FC4]
                      | [#x1FC6-#x1FCC] | [#x1FD0-#x1FD3]
                      | [#x1FD6-#x1FDB] | [#x1FE0-#x1FEC]
                      | [#x1FF2-#x1FF4] | [#x1FF6-#x1FFC]Ⓤ
                      | #x2126 | [#x212A-#x212B] | #x212EⓊ
                      | [#x2180-#x2182]Ⓤ | [#x3041-#x3094]Ⓤ
                      | [#x30A1-#x30FA]Ⓤ | [#x3105-#x312C]Ⓤ
                      | [#xAC00-#xD7A3]Ⓤ
[86]   Ideographic ::= [#x4E00-#x9FA5]Ⓤ | #x3007 | [#x3021-#x3029]
                     Ⓤ
[87] CombiningChar ::= [#x0300-#x0345] | [#x0360-#x0361]Ⓤ
                      | [#x0483-#x0486]Ⓤ | [#x0591-#x05A1]
                      | [#x05A3-#x05B9] | [#x05BB-#x05BD] | #x05BF
                      | [#x05C1-#x05C2] | #x05C4Ⓤ
                      | [#x064B-#x0652] | #x0670 | [#x06D6-#x06DC]
                      | [#x06DD-#x06DF] | [#x06E0-#x06E4]
                      | [#x06E7-#x06E8] | [#x06EA-#x06ED]Ⓤ
                      | [#x0901-#x0903] | #x093C | [#x093E-#x094C]
                      | #x094D | [#x0951-#x0954] | [#x0962-#x0963]
                       | [#x0981-#x0983] | #x09BC | #x09BE
                     Ⓤ
                      | #x09BF | [#x09C0-#x09C4] | [#x09C7-#x09C8]
                      | [#x09CB-#x09CD] | #x09D7 | [#x09E2-#x09E3]
                       | #x0A02 | #x0A3C | #x0A3E | #x0A3F
                     Ⓤ
                      | [#x0A40-#x0A42] | [#x0A47-#x0A48]
                      | [#x0A4B-#x0A4D] | [#x0A70-#x0A71]Ⓤ
```

```
                      | [#x0A81-#x0A83] | #x0ABC | [#x0ABE-#x0AC5]
                      | [#x0AC7-#x0AC9] | [#x0ACB-#x0ACD]U
                      | [#x0B01-#x0B03] | #x0B3C | [#x0B3E-#x0B43]
                      | [#x0B47-#x0B48] | [#x0B4B-#x0B4D]
                      | [#x0B56-#x0B57]U | [#x0B82-#x0B83]
                      | [#x0BBE-#x0BC2] | [#x0BC6-#x0BC8]
                      | [#x0BCA-#x0BCD] | #x0BD7U
                      | [#x0C01-#x0C03] | [#x0C3E-#x0C44]
                      | [#x0C46-#x0C48] | [#x0C4A-#x0C4D]
                      | [#x0C55-#x0C56]U | [#x0C82-#x0C83]
                      | [#x0CBE-#x0CC4] | [#x0CC6-#x0CC8]
                      | [#x0CCA-#x0CCD] | [#x0CD5-#x0CD6]U
                      | [#x0D02-#x0D03] | [#x0D3E-#x0D43]
                      | [#x0D46-#x0D48] | [#x0D4A-#x0D4D] | #x0D57
                        | #x0E31 | [#x0E34-#x0E3A]
                     U
                      | [#x0E47-#x0E4E]U | #x0EB1
                      | [#x0EB4-#x0EB9] | [#x0EBB-#x0EBC]
                      | [#x0EC8-#x0ECD]U | [#x0F18-#x0F19]
                      | #x0F35 | #x0F37 | #x0F39 | #x0F3E | #x0F3F
                      | [#x0F71-#x0F84] | [#x0F86-#x0F8B]
                      | [#x0F90-#x0F95] | #x0F97 | [#x0F99-#x0FAD]
                      | [#x0FB1-#x0FB7] | #x0FB9U
                      | [#x20D0-#x20DC] | #x20E1U
                      | [#x302A-#x302F]U | #x3099 | #x309AU
[88]         Digit ::=[#x0030-#x0039]U | [#x0660-#x0669]U
                      | [#x06F0-#x06F9]U | [#x0966-#x096F]U
                      | [#x09E6-#x09EF]U | [#x0A66-#x0A6F]U
                      | [#x0AE6-#x0AEF]U | [#x0B66-#x0B6F]U
                      | [#x0BE7-#x0BEF]U | [#x0C66-#x0C6F]U
                      | [#x0CE6-#x0CEF]U | [#x0D66-#x0D6F]U
                      | [#x0E50-#x0E59]U | [#x0ED0-#x0ED9]U
                      | [#x0F20-#x0F29]U
[89]      Extender ::=#x00B7U | #x02D0 | #x02D1U | #x0387U
                      | #x0640U | #x0E46U | #x0EC6U | #x3005U
                      | [#x3031-#x3035]U | [#x309D-#x309E]U
                      | [#x30FC-#x30FE]U
```

The character classes defined here can be derived from the Unicode character database as follows:U

- Name start characters must have one of the categories Ll, Lu, Lo, Lt, Nl.U
- Name characters other than Name-start characters must have one of the categories Mc, Me, Mn, Lm, or Nd.
- Characters in the compatibility area (i.e. with character code greater than #xF900 and less than #xFFFE) are not allowed in XML names.T
- Characters which have a font or compatibility decomposition (i.e. those with a "compatibility formatting tag" in field 5 of the database -- marked by field 5 beginning with a "<") are not allowed.
- The following characters are treated as name-start characters rather than name characters, because the property file classifies them as Alphabetic: [#x02BB-#x02C1], #x0559, #x06E5, #x06E6.
- Characters #x20DD-#x20E0 are excluded (in accordance with Unicode, section 5.14).

- Character #x00B7 is classified as an extender, because the property list so identifies it.
- Character #x0387 is added as a name character, because #x00B7 is its canonical equivalent.
- Characters ':' and '_' are allowed as name-start characters.
- Characters '-' and '.' are allowed as name characters.

C. XML and SGML (Non-Normative)

XML is designed to be a subset of SGML, in that every valid XML document should also be a conformant SGML document. For a detailed comparison of the additional restrictions that XML places on documents beyond those of SGML, see [Clark].

D. Expansion of Entity and Character References (Non-Normative)

This appendix contains some examples illustrating the sequence of entity- and character-reference recognition and expansion, as specified in "4.4 XML Processor Treatment of Entities and References".

If the DTD contains the declaration

```
<!ENTITY example "<p>An ampersand (&#38;#38;) may be escaped
numerically (&#38;#38;#38;) or with a general entity
(&amp;amp;).</p>" >
```

then the XML processor will recognize the character references when it parses the entity declaration, and resolve them before storing the following string as the value of the entity "example":

```
<p>An ampersand (&#38;) may be escaped
numerically (&#38;#38;) or with a general entity
(&amp;amp;).</p>
```

A reference in the document to "&example;" will cause the text to be reparsed, at which time the start- and end-tags of the "p" element will be recognized and the three references will be recognized and expanded, resulting in a "p" element with the following content (all data, no delimiters or markup):

```
An ampersand (&) may be escaped
numerically (&#38;) or with a general entity
(&amp;).
```

A more complex example will illustrate the rules and their effects fully. In the following example, the line numbers are solely for reference.

```
1 <?xml version='1.0'?>
2 <!DOCTYPE test [
3 <!ELEMENT test (#PCDATA) >
4 <!ENTITY % xx '&#37;zz;'>
5 <!ENTITY % zz '&#60;!ENTITY tricky "error-prone" >' >
6 %xx;
7 ]>
8 <test>This sample shows a &tricky; method.</test>
```

This produces the following:

- in line 4, the reference to character 37 is expanded immediately, and the parameter entity "xx" is stored in the

symbol table with the value "`%zz;`". Since the replacement text is not rescanned, the reference to parameter entity "`zz`" is not recognized. (And it would be an error if it were, since "`zz`" is not yet declared.)

- in line 5, the character reference "`&#60;`" is expanded immediately and the parameter entity "`zz`" is stored with the replacement text "`<!ENTITY tricky "error-prone" >`", which is a well-formed entity declaration.
- in line 6, the reference to "`xx`" is recognized, and the replacement text of "`xx`" (namely "`%zz;`") is parsed. The reference to "`zz`" is recognized in its turn, and its replacement text ("`<!ENTITY tricky "error-prone" >`") is parsed. The general entity "`tricky`" has now been declared, with the replacement text "`error-prone`".
- in line 8, the reference to the general entity "`tricky`" is recognized, and it is expanded, so the full content of the "`test`" element is the self-describing (and ungrammatical) string *This sample shows a error-prone method*.

E. Deterministic Content Models (Non-Normative)

For compatibility, it is required that content models in element type declarations be deterministic.

SGML requires deterministic content models (it calls them "unambiguous"); XML processors built using SGML systems may flag non-deterministic content models as errors.

For example, the content model `((b, c) | (b, d))` is non-deterministic, because given an initial `b` the parser cannot know which `b` in the model is being matched without looking ahead to see which element follows the `b`. In this case, the two references to `b` can be collapsed into a single reference, making the model read `(b, (c | d))`. An initial `b` now clearly matches only a single name in the content model. The parser doesn't need to look ahead to see what follows; either `c` or `d` would be accepted.

More formally: a finite state automaton may be constructed from the content model using the standard algorithms, e.g. algorithm 3.5 in section 3.9 of Aho, Sethi, and Ullman [Aho/Ullman]. In many such algorithms, a follow set is constructed for each position in the regular expression (i.e., each leaf node in the syntax tree for the regular expression); if any position has a follow set in which more than one following position is labeled with the same element type name, then the content model is in error and may be reported as an error.

Algorithms exist which allow many but not all non-deterministic content models to be reduced automatically to equivalent deterministic models; see Brüggemann-Klein 1991 [Brüggemann-Klein].

F. Autodetection of Character Encodings (Non-Normative)

The XML encoding declaration functions as an internal label on each entity, indicating which character encoding is in use. Before an XML processor can read the internal label, however, it apparently has to know what character encoding is in use--which is what the internal label is trying to indicate. In the general case, this is a hopeless situation. It is not entirely hopeless in XML, however, because XML limits the general case in two ways: each implementation is assumed to support only a finite set of character encodings, and the XML encoding declaration is restricted in position and content in order to make it feasible to autodetect the character encoding in use in each entity in normal cases. Also, in many cases other sources of information are available in addition to the XML data stream itself. Two cases may be distinguished, depending on whether the XML entity is presented to the processor without, or with, any accompanying (external) information. We consider the first case first.

Because each XML entity not in UTF-8 or UTF-16 format *must* begin with an XML encoding declaration, in which the first characters must be '`<?xml`', any conforming processor can detect, after two to four octets of input, which of the following cases apply. In reading this list, it may help to know that in UCS-4, '<' is "`#x0000003C`" and '?' is "`#x0000003F`", and the Byte Order Mark required of UTF-16 data streams is "`#xFEFF`".

- `00 00 00 3C`: UCS-4, big-endian machine (1234 order)

- `3C 00 00 00`: UCS-4, little-endian machine (4321 order)
- `00 00 3C 00`: UCS-4, unusual octet order (2143)
- `00 3C 00 00`: UCS-4, unusual octet order (3412)
- `FE FF`: UTF-16, big-endian
- `FF FE`: UTF-16, little-endian
- `00 3C 00 3F`: UTF-16, big-endian, no Byte Order Mark (and thus, strictly speaking, in error)
- `3C 00 3F 00`: UTF-16, little-endian, no Byte Order Mark (and thus, strictly speaking, in error)
- `3C 3F 78 6D`: UTF-8, ISO 646, ASCII, some part of ISO 8859, Shift-JIS, EUC, or any other 7-bit, 8-bit, or mixed-width encoding which ensures that the characters of ASCII have their normal positions, width, and values; the actual encoding declaration must be read to detect which of these applies, but since all of these encodings use the same bit patterns for the ASCII characters, the encoding declaration itself may be read reliably
- `4C 6F A7 94`: EBCDIC (in some flavor; the full encoding declaration must be read to tell which code page is in use)
- other: UTF-8 without an encoding declaration, or else the data stream is corrupt, fragmentary, or enclosed in a wrapper of some kind

This level of autodetection is enough to read the XML encoding declaration and parse the character-encoding identifier, which is still necessary to distinguish the individual members of each family of encodings (e.g. to tell UTF-8 from 8859, and the parts of 8859 from each other, or to distinguish the specific EBCDIC code page in use, and so on).

Because the contents of the encoding declaration are restricted to ASCII characters, a processor can reliably read the entire encoding declaration as soon as it has detected which family of encodings is in use. Since in practice, all widely used character encodings fall into one of the categories above, the XML encoding declaration allows reasonably reliable in-band labeling of character encodings, even when external sources of information at the operating-system or transport-protocol level are unreliable.

Once the processor has detected the character encoding in use, it can act appropriately, whether by invoking a separate input routine for each case, or by calling the proper conversion function on each character of input.

Like any self-labeling system, the XML encoding declaration will not work if any software changes the entity's character set or encoding without updating the encoding declaration. Implementors of character-encoding routines should be careful to ensure the accuracy of the internal and external information used to label the entity.

The second possible case occurs when the XML entity is accompanied by encoding information, as in some file systems and some network protocols. When multiple sources of information are available, their relative priority and the preferred method of handling conflict should be specified as part of the higher-level protocol used to deliver XML. Rules for the relative priority of the internal label and the MIME-type label in an external header, for example, should be part of the RFC document defining the text/xml and application/xml MIME types. In the interests of interoperability, however, the following rules are recommended.

- If an XML entity is in a file, the Byte-Order Mark and encoding-declaration PI are used (if present) to determine the character encoding. All other heuristics and sources of information are solely for error recovery.
- If an XML entity is delivered with a MIME type of text/xml, then the `charset` parameter on the MIME type determines the character encoding method; all other heuristics and sources of information are solely for error recovery.
- If an XML entity is delivered with a MIME type of application/xml, then the Byte-Order Mark and encoding-declaration PI are used (if present) to determine the character encoding. All other heuristics and sources of information are solely for error recovery.

These rules apply only in the absence of protocol-level documentation; in particular, when the MIME types text/xml and application/xml are defined, the recommendations of the relevant RFC will supersede these rules.

G. W3C XML Working Group (Non-Normative)

This specification was prepared and approved for publication by the W3C XML Working Group (WG). WG approval of this specification does not necessarily imply that all WG members voted for its approval. The current and former members of the XML WG are:

Jon Bosak, Sun (Chair); James Clark (Technical Lead); Tim Bray, Textuality and Netscape (XML Co-editor); Jean Paoli, Microsoft (XML Co-editor); C. M. Sperberg-McQueen, U. of Ill. (XML Co-editor); Dan Connolly, W3C (W3C Liaison); Paula Angerstein, Texcel; Steve DeRose, INSO; Dave Hollander, HP; Eliot Kimber, ISOGEN; Eve Maler, ArborText; Tom Magliery, NCSA; Murray Maloney, Muzmo and Grif; Makoto Murata, Fuji Xerox Information Systems; Joel Nava, Adobe; Conleth O'Connell, Vignette; Peter Sharpe, SoftQuad; John Tigue, DataChannel

Versions of the XML 1.0 Specification

XML

The XML version of the spec is the one that Michael and I actually authored, and should be considered authoritative. It contains substantially more information (including a partial revision log, and some amusing comments) than appear in any of the HTML or print versions. It was edited mostly in GNU Emacs.

HTML

The version you are reading, and the one at the W3C site, were generated by a Java program I wrote that uses my XML processor (Lark) to read the specification, and a lot of custom code to generate the corresponding HTML. The HTML version is the best version to use if you are a programmer and want to drill down into the spec and check up on the details of definitions and productions.

PostScript

The PostScript version was created by using Jade, a public-domain DSSSL engine; this was driven by a stylesheet that created an RTF version of the spec. The RTF version was hand-edited in Microsoft Word to fix up pagination and so on, and to generate the PostScript. All this work was done by Jon Bosak.

PDF

The PDF version was created from the PostScript version using the Adobe Distiller product. It is probably the one to get if you want to print out the spec for easy reading. It is interesting to note that the HTML version is slightly larger than the XML version, while the PDF version is slightly smaller.

Back-link to spec

Jon Bosak

Jon Bosak is the single person without whose efforts XML would most likely have failed to happen. He had come to appreciate the power and flexibility of SGML in his days running Novell's (excellent) on-line documentation repository at http://www.novell.com, and had acquired a conviction that HTML was not a suitable base on which to build the next layer of Web infrastructure.

Jon's stewardship of the XML process has been marked by a combination of deft political maneuvering with steadfast insistence on the principle of doing things based on principle, not expediency.

Back-link to spec

Copyright Details

Previous Releases

This one-liner glosses over a whole lot of history. Follow the link and look at *its* "Previous Releases" link to get the full story.

Back-link to spec

Tim Bray

I am a Canadian; I graduated from the University of Guelph in 1981 in math and CS. After on-the-job training from Digital (R.I.P.) and GTE, I got the best job in the world in 1987, managing the *New Oxford English Dictionary Project* at the University of Waterloo. This project led to the founding of Open Text Corporation in 1989, and an Internet IPO. XML was my first project after I became an independent consultant in 1996.

I had never really done any serious standards work, let alone written a specification, before this project. My main motivations in the XML project were:

- To produce something that programmers could implement,
- To make sure the internationalization was thorough and usable, and
- To make sure that XML would provide good support for search and retrieval applications.

Back-link to spec

Jean Paoli

Jean Paoli has been a leader in the SGML community since its early days; he was chief technologist at Grif S.A., for many years the leading European vendor of SGML authoring technology.

Well-connected to the European research establishment that was the birthplace of the WWW, he was aware of its importance before most people knew it existed.

He joined Microsoft in 1996 and, shortly thereafter, was a founding member of the XML effort. He deserves a very large part of the credit for waking Microsoft up to the importance of descriptive markup in general and XML in particular.

Jean's contributions to the XML WG debate got careful attention not only because of who he represented, but because he knows what he's talking about. He never actually did any editorial work on the XML specification; the appearance of his name in the list of editors achieved two important political goals:

1. Helping ensure the acceptance of XML by getting Microsoft's name on the cover, and
2. Defusing a political brouhaha that blew up shortly after I, after having served as a WG member and co-editor on a pro bono basis for some 8 months, signed a consulting contract with Netscape whereby I acted as their eyes, ears, and voice on the XML WG. Microsoft, unable to tolerate their competitor being in an apparently favored position, demanded (and temporarily got) my dismissal as co-editor. Jean's appointment was part of the bargain that restored me to the co-editorship. In fairness to Jean, it should be pointed out that he never asked for the posting, and rumor has it that he actively resisted it, but it was such an obviously good idea that it got quick consensus support.

Back-link to spec

C.M. (Michael) Sperberg-McQueen

Michael Sperberg-McQueen is justly famous as Editor-in-Chief of the Text Encoding Initiative, and also as the perennial keynote speaker at the annual SGML conferences.

He is authoritative on SGML and XML, and erudite on a frightening number of other subjects. He and I split the actual work of creating this specification along approximately equal lines; I used to say that I wrote 40% and Michael wrote 80%, then I chopped. He deserves most of the credit for the grammar engineering.

Back-link to spec

The Correct Title

The correct title of this specification, and the correct full name of XML, is "Extensible Markup Language". "e**X**tensible Markup Language" is just a spelling error. However, the abbreviation "XML" is not only correct but, appearing as it does in the title of the specification, an official name of the Extensible Markup Language.

The name and abbreviation were invented by James Clark; other options under consideration had included MGML, for Minimal Generalized Markup Language. Here is an excerpt from an email from James dated August 19, 1996:

I agree that GM isn't vey catchy. The other problem with "generalized" is that I suspect many, even quite technical people, don't know what a generalized markup language is. Nonetheless it seems to me that the fact that our markup language is generalized is something that should be tremendously appealing to users: "it's the markup language where *you*, not W3C or Netscape or Microsoft, choose how to mark up your data". I think what we need is a word that gets across the idea of generalized markup who don't know what it is. Perhaps something like "unrestricted", "unlimited", "extensible", "user-controlled".
I think putting "standard" in the name of the standard is a bit vacuous, so I would favour a name like UML or XML.

And here's a reply from Jon Bosak, dated August 20th:

In my opinion, the U-combinations won't fly, but if we allow "X" to stand for "extensible", then I could live with (and even come to love) XML as an acronym for "extensible markup language", and I hereby now throw it into the list of current proposals.

And here, finally, are the results of the committee vote:

Votes	Acronym	Full Name
5	XML	Extensible Markup Language
4	MAGMA	Minimal Architecture for Generalized Markup Applications
3	SLIM	Structured Language for Internet Markup
1	MGML	Minimal Generalized Markup Language

Back-link to spec

SGML

SGML, for Standard Generalized Markup Language, is ISO Standard 8879. SGML has a record of successful application in large, complex, sophisticated, publishing applications. However, it had never (as of mid-1996) really taken off on the World-Wide Web, despite the fact that HTML was advertised as being an application of SGML.

XML is an attempt to package up the important virtues and most-used features of SGML in a compact, easily-implemented package that is optimized for delivery on the WWW.

Back-link to spec

What "Recommendation" Means

The World Wide Web Consortium (W3C) is not a democracy. This sentence means exactly what it says: that the Director, Tim Berners-Lee, having reviewed this specification as well as the votes and commentary submitted by W3C member organizations, decided to bless this document as a Recommendation.

Back-link to spec

Citation Of The XML Spec

Presumably, the most common method of citing this document is by URL; use the first one appearing above under the heading "This Version".

A correct bibliographic reference, for use in paper publications, would be:

"Extensible Markup Language (XML) 1.0", Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, 10 February 1998. Available at http://www.w3.org/TR/REC-xml

Back-link to spec

URIs, URNs, and URLs

The use of URI in this specification, as opposed to URL, was a matter of W3C policy; W3C is actively pursuing a policy of generalizing pointers and popularizing the use of the term URI, with the goal of actually popularizing their use.

This was somewhat controversial, since at the time of the creation of the XML spec, virtually every computer in the world was equipped with a URL resolver, whereas virtually none were properly equipped to deal with URIs in the general case. Furthermore, the number of people who really understand what "URI" means is infinitesimal, compared to the number with a good working understanding of URLs.

Concretely, a URI (Uniform Resource Identifier) is either a URN (Uniform Resource Name) or URL (Uniform Resource Locator). URLs are well-known, but contain more subtleties than one might think. URNs are an attempt to give something on the Web a longer-lasting and more robust name than a URL offers. A good introduction to all this stuff may be found here.

Note the consistent usage of the word *Resource*; a resource is a key concept in the architecture of the Web: any addressable unit of information of service. In practical terms, the definition (although useful) is somewhat circular: a resource is anything that can have a URI, and a URI is a short piece of text that identifies a resource.

Back-link to spec

Send Mail, Please

One of the people who get mail sent to this address is me, and we all take it very seriously. Real problems that get reported will go on the errata page, and will be fixed if there's ever another release of XML.

Back-link to spec

What Do You Mean By "Data Object?"

Good question. The point is that an XML document is sometimes a file, sometimes a record in a relational database, sometimes an object delivered by an Object Request Broker, and sometimes a stream of bytes arriving at a network socket.

These can all be described as "data objects".

Back-link to spec

Why All This Blue Text?

Note that the words "XML Document" are a hyperlink in one place and bold-face in another. That means that this is a formally defined term in the XML document; the hyperlink points to the definition of the term. In this version of the spec, such definitions are marked with the string "[Definition:]" and (if your browser supports it) color.

When you follow the hyperlink and arrive at the definition of the term, you'll notice that it appears in bold-face; in fact, anything presented in bold-face is a defined term that will have pointers to it from here and there around the document.

These were designed to help out in reading the spec, but have also proved handy in external documents, such as this one. See, we can point at XML Document from here, too!

Back-link to spec

Syntax, Processors, and APIs

This specification gives a rather thorough description of the syntax rules that make a data object into an XML document. It uses the *XML processor* idea, introduced here, to help explain some of this syntax, and to constrain some aspects of the behavior of programs that process XML documents.

However, the specification does not define an Application Programming Interface which an application can use for interaction with an XML Processor. In fact, one of the specification's weak points is the discussion of exactly what information an application can expect to receive from an XML document.

While this shortcoming is acknowledged, it has not caused me any great loss of sleep. I have seen immense amounts of work invested by very smart people in an attempt to create truly interoperable APIs; examples would include SQL, Posix, and the X Window System. Achieving real interoperability at the API level has proved so difficult in practice as to be open to question as a design goal. Furthermore, it is hard to be sure of coming up with an API that is of equal utility for all aspects of interacting with an XML document; the needs of an authoring system, of a browser, and of a full-text indexer are dramatically different.

The real saving grace is the syntax. It is commonplace, even fashionable, to belittle the importance of syntax. But a document format which can unambiguously express complex hierarchical data structures, and which can reliably be parsed in a variety of computing environments, is in itself something of a rare and special achievement. If this is the only level of interoperability that XML ever achieves, it will still prove to have been a significant step forward in the history of distributed computing.

Back-link to spec

By Construction

What this sentence means is that an XML document which follows the rules in this specification is a conforming SGML document. However, this specification does not have a normative reference to the SGML standard, and it should not be necessary to read or understand that document in order to construct and use XML documents and software. This is a benefit, since the SGML standard is a large document and difficult to read. This should not be seen as a criticism of SGML, since that standard is trying to achieve much higher, more complex, and more generalized goals than XML is.

Back-link to spec

Parsed and Unparsed

The use of the terms "parsed" and "unparsed" for entities may seem a bit on the obscure side. SGML uses the terms "text" and "data" for the same purposes, but we found that misleading, because it's all, once you get right down to it, data, and furthermore, some of the "data" entities might contain text. The only real difference between the two kinds of entities is whether an XML processor has to try and parse them or not; hence the names.

Another benefit of using "parsed" and "unparsed" is that this frees up the useful word text, which really ought to mean something in the XML context.

Back-link to spec

The History and Utility of XML Processors

When we invented the idea of the "XML Processor", it was essentially a theoretical construct; some aspects of the language are easier to explain in terms of the actions of a program than with purely syntactic techniques. In particular, it's hard to see how to define error-handling behavior without having a system to describe the behavior of.

By the time the first draft of the spec was out, it was starting to look like an XML processor might be useful in real-life, not just in theory. So I sat down and wrote one, and so did lots of other people, and it has turned out that the use of a real XML processor is *essential* for the serious XML developer.

If you use a conforming XML processor, the things you won't have to worry about include:

1. dealing with characters in any format other than native Unicode,
2. picking apart the syntax of tags and attributes and entities, and
3. worrying about how your operating system happens to deal with line-ends.

Another nice thing about XML processors is that they are free, and pretty easy to use, so there's really no good reason not to use one.

Back-link to spec

The Processor and the Application

While this spec constrains some behaviors of an XML processor, it places no constraints on the application. This is an important point; it would be inappropriate (not to mention futile) for this document to try to enforce what other people *do* with XML; our job is limited to ensuring that they can interchange XML documents and agree on what they contain.

Back-link to spec

History

The "SGML Editorial Review Board" named here was born in the summer of 1996. To quote an email from Jon Bosak, the ringleader:

I've been bugging the W3C for some time to start up a group in parallel with their html-erb to foster the early development of SGML and DSSSL on the Web. They finally responded by saying, basically, that if I wanted a group like that, I could start it myself. After discussing this with some key people at WWW5 in Paris and SGML Europe in Munich, and getting the approval of my management to sink significant time into this, I have decided to take them up on their offer.

The first email exchanged among the members of the then-SGML ERB is dated Monday July 22, 1996. The email exchanges in that body were hosted by Dave Hollander at Hewlett-Packard, since a combination of infrastructure problems and lack of interest in the W3C made it impossible for them to host.

One of the advantages of Jon's approach (going ahead even though there was little enthusiasm evident at the time in the W3C community) was that he was able to recruit members based on what he thought they had to offer, not what their employers wanted to get accomplished.

Back-link to spec

The Working Group/SIG

The "SGML WG", which eventually became the "XML Special Interest Group", came into existence on August 28, 1996. In practice, the smaller group (WG) marshaled issues for debate, then the larger group (SIG) debated, then the WG resolved things by voting. In practice, this often failed to settle important issues; on several, the SIG simply refused to accept the results of the vote and bullied the WG until it reversed itself.

The important thing about the SIG is the breadth of its membership (including, crucially, SGML luminaries such as Charles Goldfarb, Dave Peterson, and James Mason), and the quite frankly astounding amount of time they were able to put into the process.

The important thing about the WG is how well the process worked. Its discussions and votes, which are a matter of public record, reveal that while the WG often failed to achieve unanimity, it did achieve *consensus* in the important meaning of the word, as evidenced by the fact that members of the WG are often willing to defend aspects of XML which they personally voted against.

Back-link to spec

Dan Connolly and the W3C

The W3C was, paradoxically, something of a late arrival to the W3C party. While they authorized Jon Bosak to found and run the activity, providing that he made no call on W3C resources, the W3C staff did not perceive that XML had the potential for really high impact.

The concrete effect of this was that the staff in general and Dan Connolly in particular essentially ignored the progress of XML until it suddenly started to gain wide industry acceptance in the spring of 1997. This had an upside in that the XML process was relatively untroubled by the kind of time-wasting industry politics and bureaucratic infighting that are inevitable in an organization such as W3C. It had a downside in that the XML process was deprived of Dan Connolly's considerable expertise and experience. In particular, after Dan became interested and involved, he made several suggestions which would, if implemented, probably have have constituted real improvements in this specification. Unfortunately, they would have required major document re-engineering, and the required time and editorial cycles were simply not present at that late stage of the process.

Once XML became visible and public, Dan became an invaluable resource and deserves considerable credit for its eventual arrival at "Recommendation" status.

Back-link to spec

Design Goals

These were hashed out by the smaller Working Group and published in August 1996, before debate started in the larger SIG. The debate that led to their creation was very useful, and their existence proved of great value in guiding the detailed debate.

The commentary attached here to each of the design goals is based on large part on the more extended published version mentioned above.

Back-link to spec

Usable Over the Internet

This was not taken to mean that you could feed XML to the browsers of the day, but that the design would have regard at all times to the needs of distributed applications working on large-scale networks.

Back-link to spec

A Wide Variety of Applications

This can be seen as something of a counterweight to the first design goal. Whereas XML is built for the Net, it is not limited to the Net. Specifically, we wanted to enable the creation of applications including authoring tools, simple filters, formatting engines, and translators.

Back-link to spec

SGML Compatibility

Of our design goals, this proved to be most troublesome in practice. Our subgoals included:

1. Existing SGML tools will be able to read and write XML data.
2. XML instances are SGML documents as they are, without changes to the instance.
3. For any XML document, a DTD can be generated such that SGML will produce "the same parse" as would an XML processor.
4. XML should have essentially the same expressive power as SGML.

That we largely succeeded in accomplishing these goals is due to two factors. First, the determination of many members of the WG and SIG to stick with SGML compatibility, even when there were good design reasons for abandoning it. Second, the active participation and co-operation of the SGML community, led by Charles Goldfarb, in ensuring that the load of ensuring that XML and SGML remained compatible was shared by both parties.

Back-link to spec

Easy-to-Write Programs

Specifically, we wanted it to be straightforward to create useful XML programs that did not depend on reading the DTD. We were ambitious, and declared that "easy" meant that the holder of a CS bachelor's degree ought to be able to write basic XML processing machinery in less than a week.

The motivation here is obvious - data formats succeed or fail depending on whether there are good tools available. If the format is easy to process, good tools will be available; otherwise not.

The processor-in-a-week goal has proved elusive, so in that respect we failed to meet this design goal at a quantitative level. However, the fact that within months of the appearance of the first drafts of the XML spec, there were a substantial number of freeware XML processors on the market serves as evidence that perhaps we did meet our goal, qualitatively.

Back-link to spec

No Optional Features

One of the problems with SGML has historically been that in its (laudable) attempt to maximize generality, it adopted a large number of optional features. This meant, in practice that if I had a conforming SGML document and you had another, we might not be able to interchange them.

This design goal was met; XML has no options, and every XML processor in the world should be able to read every XML document in the world, assuming that it can decode the characters.

Back-link to spec

Legible and Clear

This goal was motivated simply by the perception that textual formats are more open, more useful, and more pleasant to work with than binary formats. One of the substantial benefits of XML is that no matter how bad a day your tools are having, you can always pull an XML document into Emacs or Notepad or whatever your favorite editor is, and get useful work done.

Back-link to spec

Quick Design Process

This goal was motivated largely by fear. We perceived that many of the Net's powers-that-be did not share our desire for widespread use of open, nonproprietary, textual data formats. We believed that if we didn't toss XML's hat into the ring soon, the Web's obvious need for extensibility would be met by some combination of binary gibberish and proprietary kludges.

Back-link to spec

Formal and Concise Design

This is closely related to design goal #4 above; a data format is programmer-friendly if programmers can read and use the defining documents; otherwise not. Too many other standards and specifications have relied too heavily on prose and not enough on formalisms.

This is one area where some, in particular Dan Connolly, have argued that the actual XML spec falls short of the goal; that a version could have been created which was substantially more formal and concise than the current document.

Back-link to spec

Easy to Create

There is more at work here than the simple realization that if the documents are easy to create, then they will be created. The main goal was in fact to design XML in such a way that it would be tractable to design and build XML authoring systems. Our success in meeting this design goal remains to be established in the marketplace.

Back-link to spec

Terseness is of Minimal Importance

The historical reason for this goal is that the complexity and difficulty of SGML was greatly increased by its use of minimization, i.e. the omission of pieces of markup, in the interest of terseness. In the case of XML, whenever there was a conflict between conciseness and clarity, clarity won.

Back-link to spec

Other Goals

There are two other principles which, while they don't appear in this list as design goals, continually surfaced during the XML discussion. The first was that of internationalization. It was an inexcusable oversight to miss this in the list above. We bent over backward to make XML work properly in all the world's scripts, with a fair degree of success.

The second undeclared design principle is embodied in the acronym "DPH". This stands for "Desperate Perl Hacker" - the luckless subordinate who is informed that some global change is required in a large, complex document inventory at short notice, and who is able to deliver by applying a scripting facility such as perl to cleanly-structured data such as XML.

Back-link to spec

A Little Contradiction

This paragraph, which makes it clear that XML outsources some of its problems to other specifications (a good idea, and one with which no-one disagrees), is in fairly stark contrast to the claim in the abstract that XML is "completely described in this document". Oops.

Back-link to spec

Who Owns The XML Spec?

In effect, nobody. Common copyright law would suggest that some interest in the spec vests in the individuals who wrote it and in the W3C (due to language in the contract it signs with its members).

Fortunately, all of those who have any claim to a piece of the copyright share a common set of goals: that the specification be distributed as widely as possible, and that its integrity be preserved.

Back-link to spec

May

The use of this word is controversial, as it suggests that XML has optional features, which is not true. There are, however, some actions which an XML processor is allowed but not required to take, particularly in the case when it is a non-validating processor. This will absolutely not weaken XML's claim that any processor can read any document (subject to character encoding issues), and is not expected to cause operational problems; furthermore the spec describes how an application may avoid any unpredictability in results.

Back-link to spec

Must

The spec should probably say, but does not, that when it says "in error", it is using the term "error" in precisely the sense described in the following paragraph. That is to say, this is a class of errors which you can't always be sure of detecting. An obvious example would be an external entity which says it's encoded in Shift-JIS but is actually in ISO-Latin-1; in some circumstances, particularly if it didn't contain any of the magic XML syntax characters, there would be no way for an XML processor to detect this, but it would still be an error.

Back-link to spec

Draconian Error Handling

This innocent-looking definition embodies one of the most important and unprecedented aspects of XML: "Draconian" error-handling. Dracon (c.659-c.601 B.C.E.) introduced the first written legislation to Athens. His code was consistent in that it decreed the death penalty for crimes both low and high. Similarly, a conforming XML processor must "not continue normal processing" once it detects a fatal error. Phrases used to amplify this wording have included "halt and catch fire", "barf", "flush the document down the toilet", and "penalize innocent end-users".

The motivation for this policy is simple. We want XML to empower programmers to write code that can be transmitted across the Web and execute on a large number of desktops. However, if this code must include error-handling for all sorts of sloppy end-user practices, it will of necessity balloon in size to the point where it, like Netscape Navigator, or Microsoft Internet Explorer, is tens of megabytes in size, thus defeating the purpose.

The circumstances that produce the Draconian behavior - "fatal errors" - are all failures to attain the condition of well-formedness. Well-formedness doesn't cost much; the tags have to be balanced, the entities have to be declared, and attributes have to be quoted; that's about it. The benefits of well-formedness - the empowering of programmers discussed above - are high. Thus Draconian error-handling is arguably a good trade-off.

However, it is also arguably not. It is directly contradictory to the spirit of HTML, where tool vendors compete on their ability to handle egregiously broken pages. It is also counter to SGML practice, where tools often make a best-effort attempt to continue in the face of errors. These facts led to a debate on this particular subject that was as intense and prolonged as any in the history of the XML project. The final majority in favor of the Draconian policy was large but by no means 100%.

Increasing the level of interest in this debate was the fact that the Draconian policy was requested, separately and independently, by Netscape and Microsoft, who were bitterly aware of the consequences of trying to work around user error in the extreme case. I don't think anybody really would retroactively have changed the policy of forgiveness as regards HTML; it is one of the reasons for the success of the Web. But in the case of XML, we have it as an explicit goal that this should be fodder for programs other than screen-painters; given this, the Draconian policy eventually managed to build what amounted to consensus support.

Back-link to spec

At User Option

The point of the phrase "at user option" is that there are some things that the XML processor really *should* warn you about; but equally, sometimes you may choose not to listen to these warnings, and you should be able to tell the processor to shut up. Note that this is always a *negative* option; a behavior which the user can disable, not enable.

Few people know what a modal verb is; I certainly didn't, though Michael Sperberg-McQueen claimed, after the fact, that he did. A "modal verb" is "must" or "may" - the amusing thing about this usage is that an earlier draft just said "verb", but Murata Makoto, a WG member and the chief translator of the XML spec into Japanese, exhibiting an intimidating grasp of English grammar, gently corrected the editors on this point.

Back-link to spec

Constraints

Validity and well-formedness constraints are rules that the editors weren't smart enough to figure out how to build into the grammar of the spec, and were thus forced to resort to verbiage, in defiance of two of XML's design goals.

Back-link to spec

String Matching

Most people think string matching should be easy, but that doesn't include people who work with Unicode (of which much more elsewhere). The problem is that in Unicode, a character such as "é" has two representations; one as the single character whose number is hexadecimal #xe9 (decimal 233), another as the ordinary character "e" (#x65, decimal 101) followed by the accent character (#x301, decimal 769). On the screen or on paper, though, there's no way to tell these apart. The Unicode Standard has all sorts of good advice as to how to deal with these situations.

To keep things simple, XML doesn't require a processor to try any of these combining-character tricks; that is to say, it is free to regard to the single character #e9 as different from the two-character sequence #65,#301. It is *allowed* to try, which might be a desirable feature in a commercial product; but if this behavior is causing problems for users, they can (note the "at user option phrase") turn it off.

Back-link to spec

XML is Case-Sensitive!

This is really different from what we've all become used to with HTML and SGML (well, SGML allows case-sensitivity, but nobody actually uses it). Most people who've worked with either SGML or HTML find XML's case-sensitivity unnerving at first, but you get used to it quickly, just as we've all become used to case-sensitivity in our programming languages and file names.

The reason for the case-sensitivity is simple: internationalization. English is one of the few languages in the world where it's easy and straightforward to map upper- and lower-case letters together. The majority of the world's population uses languages that don't even have case, and don't see why one class of characters should arbitrarily be regarded as the same as another.

But once you've left the safe bounds of English, you don't have to go all the way to Asia to get in trouble with case-folding. What is the upper-case version of the character "é"? It turns out the answer is sometimes different depending on whether you're in Québec or France. Then there are the problems with the German "ß" and the dotless Turkish "i", and in general... well, you want to stay away from case-folding. So XML does.

Back-link to spec

Matching Productions

Anyone who's taken an undergraduate Computer Science course that covers Compilers will understand the gibberish in the preceding sentence. If you haven't, don't worry about it; it says that the text in XML documents has to match, in a formal sense, the grammar in this document.

Back-link to spec

For Compatibility

The somewhat strained tone in this definition evidences the somewhat strained quality of debate that created it. The problem is simple. Being compatible with SGML is a real virtue, for several reasons:

- It allows the use of the rich inventory of existing SGML tools on XML documents.
- It protects document owners from vendor rape; a demand for standards-compliance is one of the few defenses an information producer has against the natural desires of the software vendor to lock his data into the vendor's products.
- It helps sell products to government agencies and the like who like to buy standards-conformant stuff.

On the other hand, SGML was designed in the early-to-mid 1980s, and with experience, we have come to be keenly aware of some design errors. It was very tempting merely to eliminate them from XML, and some of us thought that this was the way to go. Eventually, we decided to include some rather questionable constraints on XML simply for the sake of compatibility.

One of the reasons why this was easy was the eagerness of the SGML community to work with us, and in fact to modify SGML to remove some of these irritants, to an extent that was very gratifying.

Back-link to spec

For Interoperability

The SGML community was very forthcoming in working with us to avoid the existence of any real or perceived split between SGML and XML. To that end, they designed, proposed, and voted into the ISO standard a set of changes that removed many potential irritants. This was a good thing all around.

Fortunately or unfortunately, there is a large installed base of SGML products that don't know about these changes to the standard and will thus (quite properly) complain about some things that can appear in XML documents. Wherever this can happen, the XML spec contains a "for interoperability" warning, and if you think you might need to feed your XML documents to existing SGML products, you should take these seriously.

Back-link to spec

The Term "Well-Formed"

The design of XML was a truly creative effort, and my contribution as co-editor was far from the largest. The idea of well-formedness was originally, however, mine. It never got voted on; I just wrote the terms "validity" and "well-formedness" into the first draft; Michael Sperberg-McQueen demurred briefly, as the term had, for him, some other semantic baggage, but it survived in the absence of other suggestions, and when the draft became public, nobody ever challenged the idea.

It should be noted that the term "well-formed" does have another meeting in formal mathematical logic; an assertion in formal logic is well-formed if it meets certain grammatical rules; whether it is "true" or not in some framework is another question entirely. I found the analogy very compelling, and nobody else ever thought up a better name.

In SGML, they have adopted the terms "tag-valid" and "type-valid" corresponding respectively to XML's "well-formed" and "valid".

Back-link to spec

Well-Formed and Valid

Suppose that I have an XML document, but I don't claim that it's valid. This could mean any of the following:

1. There's no DTD for this document, or
2. There is a DTD for this document, but it's off on a server somewhere and I don't want to get it, or
3. There's a DTD available right here, and in fact if I were to check, the document *would* be valid, but all I'm doing is displaying or indexing it, so I don't care about the DTD.

Back-link to spec

Nonterminal

Note that the word "document" is not only a hyperlink, but it's in `computer` typeface. This means that it's a "nonterminal", one of the names defined and used in the grammar of the XML specification. Follow that pointer and you'll go to the production that defines it.

Back-link to spec

Trailing "Misc"

The fact that you're allowed some trailing junk after the root element, I decided (but unfortunately too late) is a real design error in XML. If I'm writing a network client, I'm probably going to close the link as soon as a I see the root element end-tag, and not depend on the other end closing it down properly.

Furthermore, if I want to send a succession of XML documents over a network link, if I find a processing instruction after a root element, is it a trailer on the previous document, or part of the prolog of the next?

Back-link to spec

Roots and Nests

This paragraph (which, strictly speaking is unnecessary - it is merely amplifying the consequences of the grammar) describes the essence of well-formedness. Simply stated, there has to be one element that contains everything else, and all the elements have to nest nicely within each other - no overlapping! All this "root" and "nest" terminology suggests trees, which is just fine.

The fact that XML requires a single root element is more important than you might think; given that we expect to be transmitting these documents over network links which, we all know, are sometimes slow, flaky, and unreliable, it's a really good idea if the beginning and (especially) the end of every document is clearly marked, so that even if the guy on the other end is slow in closing down the link, you know when you've got the whole message.

Back-link to spec

Parent and Child

Some reviewers of the XML spec grumbled darkly about this un-called-for (they said) side trip into mathe-magic. Well, while one of your co-editors will confess to a math degree, we do bandy the terms "parent element" and "child element" around an awful lot, and it's good to have it written down somewhere, with great precision, exactly what that means. Anyhow, that's our story, and we're sticking to it.

Back-link to spec

Unicode

If all you're ever going to do is read and write ASCII documents, you can safely ignore all this material about Unicode and 10646. On the other hand, if you're in the other 98% of the computing profession, then:

You need to know about this stuff!

SGML had an even more general notion of what a character is than XML does; so general in fact, that it was hard to be interoperable. XML takes a very simple view: characters are just numbers, and the numbers mean what the Unicode (and thankfully-identical ISO 10646) standard says they mean.

If you *don't* know about this stuff but think you ought to, relax, there's good news; all you need to do is get the Unicode spec (available at good technical bookstores or from the Unicode Web site). OK, so it's a little expensive; go ahead and get it anyhow.

Once you've got it, you may blanch at its huge size and weight and wonder what you've got yourself into. Relax, 80% of it is tables of Chinese, Japanese, and Korean characters, which you can safely ignore (unless you're a user of one of those languages who is working with electronic texts, in which case you'll be glad of having them). The first 60 pages or so contain everything you really need to know to do international character processing, presented sanely, logically, and with really great typography.

If you're a bibliophile, you'll enjoy reading this. If your friends and loved ones are too, then it's a coffee table book.

Back-link to spec

Legal Characters

XML is thorougly internationalized, compared to most other document formats in common use, because it signs up to support each and every Unicode character. So the problem's solved, right? Well, not really, especially if you're a mathematician or textbook publisher. Because it turns out, there are lots of useful characters that are in occasional use (particularly in math textbooks, almost never in your weekly report to your boss) but just aren't there in Unicode.

There are a few solutions to this problem. One would be to use SGML, which has a trick called SDATA entities that can be used to talk about any old character you might want to dream up, whether or not they actually exist. Secondly, Unicode has a block of 6,400 characters called the "Private Use Area" (#e000 to #f8ff, decimal 57,344 to 63,743) for precisely this purpose; in your own application, you can use these characters to mean anything you want. Of course, if you want to interchange your XML documents with anyone else, you'd better have agreed in advance on what you're up to in this area.

Back-link to spec

Compatibility Characters

Unicode makes a heroic effort to include all the real, useful, characters from all the world's writing systems. But when it was invented, it had to deal with the fact that there was a *lot* of existing text in the world. As a result, it had to adopt some fairly unwanted children.

For example, in English, when a lower-case "i" follows a lower-case "f", it is common to print them in a combined form called a ligature. It's probably wrong to think of this thing as a "character", but nonetheless, there were a lot of typesetting systems around that had a code for it. So there is a Unicode "compatibility" character (#xfb01, decimal 64527, if you must know) for this and for quite a few similar misshapen beasts.

There are quite a few Japanese compatibility characters, in particular the dreaded "half-width katakana". These arose from a misguided attempt, in the early days of computing, to handle Japanese texts by using only one of the three Japanese alphabets and storing these things in one byte apiece. These "half-width katakana" were also printed out at the same width as a Latin character, which is, well, half as wide as is really comfortable. The problem is, although Japanese computing is now done properly with 16-bit characters, there were some old systems that depended on these half-width mistakes, and thus they made it into Unicode.

However, the Unicode people were smart enough to provide an index which makes it easy to spot all these compromises with history, and it would be a really good idea if you were never under any circumstances to use them in a modern XML document. But if you must, this discouraging note won't actually stop you.

Back-link to spec

Unicode Surrogates

Unicode characters are made up of 16 bits, which allows for 65,536 different characters. At this point, not all of them have been used, but they will be one day. There are a very large number of Japanese and Chinese characters that have not yet been included. None of these are in common use, but they include a lot of characters of serious interest to scholars and historians, so the problem won't go away.

Also, Japanese and Chinese people can (and do) invent new characters all the time.

But there are only so many characters you can store in 16 bits, so how to make the problem go away? Previous international-text standards, in particular the much-hated ISO 2022, used a trick where you would switch modes, i.e. have a magic sequence that said "shift into Korean" or "shift into Arabic"; while this was straightforward enough to render on a screen, it was pure hell for programmers. I'll skip the details to avoid boring the non-programmers, but for the geeks in the crowd, two words: pointer arithmetic.

Unicode has a clever solution to this, called the "surrogate blocks". They call the 65,536 Unicode characters the "Basic Multilingual Plane" (BMP for short), and it contains almost all of the characters most people will ever need. In the BMP, two chunks of 1,024 characters have been reserved and will never be used to represent ordinary characters. These are called the Surrogate Blocks; the first extends from #d800 to #dbff (decimal 55,296 to 56,319) inclusive, the second from #dc00 to #dfff (decimal 56,320 to 57,343). The trick is that for any new characters that get assigned outside of the BMP, they have to be made up of a 16-bit character from the first block, then another from the second. This way you get about a million (1,048,576) new characters; although to keep things simple, the Unicode people prefer to think of them as 16 new planes, each of 65,536 characters.

The beauty of the surrogate technique is that a program that doesn't understand it would just render such a character as two blobs on the screen. A program that understands the basic idea, but doesn't know that exact character, would render it as one blob on the scren. And a program that knows the character would be just fine.

And programmers love it, because by looking at one 16-bit quantity they can always tell whether they're looking at an ordinary character, or at the top or bottom half of a non-BMP character.

The practical effect is that if you have a character from one of the surrogate that's not part of a 2-character low-surrogate high-surrogate combo, then you have a fatal error.

By the way, the surrogates are going to be used not only for lots of Chinese-style characters, but for the characters of Egyptian hieroglyphics, Mayan, other dead languages, and Klingon.

Back-link to spec

Encodings

Unicode, as we discussed above, identifies characters using 16-bit numbers. But that's not everything you need to know, because there are a variety of different ways of storing those 16-bit numbers in computer files, and XML can hardly get away with telling everyone they have to use the same one.

Unicode Encodings

Unicode itself defines a variety of ways of doing this; UTF-16 is the simplest; it just stores each 16-bit character in 16 bits in the obvious way, and larger characters using the Surrogate block trick described above.

UTF-8 is a trick, originally invented by the Unix gang at Bell Labs, which stores old-fashioned 7-bit ASCII characters as themselves, and anything else as anywhere from 2 to 5 bytes, each with a value greater than 127. UTF-8 saves disk space if your text is mostly ASCII, but wastes it for other texts.

UTF-8 also has the important virtue that pre-Unicode computer programs that process text a byte at a time can usually, if they're not doing anything too fancy, get away with processing UTF-8. On the other hand, if you're programming in Java, where all characters are 16 bits, you'd rather stay away from UTF-8.

Non-Unicode Encodings

Very little of the world's text is stored in Unicode encodings. Most of it's in ASCII or ISO-Latin (European) or JIS (Japanese) or one of many Chinese encoding schemes. However, since Unicode was built by combining the contents of all those schemes, all the characters in those files are in fact Unicode characters; so each of these things can be regarded just as encodings of 16-bit Unicode characters. And in fact, converting from any of these to real Unicode values is actually fairly straightforward. XML allows you to use any of these encodings, and has some tricks for communicating which one is in use.

Back-link to spec

White Space

White space is important in documents. In some languages, it's used to separate words. In computer files, it's commonly used to break things up and make text more readable. In typography and presentation, the effective use of white space is tremendously important.

Watch out; not all the characters that you might think of spaces count as spaces in XML. In particular, Unicode defines a few special kinds of spaces, including one called "Ideographic Space" designed for use in between Chinese, Japanese, and Korean characters. But in XML, none of them are white space in the sense this specification uses the word.

Back-link to spec

Letters

This sentence only makes sense if you read it in the Unicode context; words such as "syllabic", "base character", "combining character", and "ideographic character" are used very carefully and have important meanings in non-English languages.

Back-link to spec

Legal Name

The basic idea of XML is that you divide documents up into physical and logical structures, and then you give the structures names. There are some rules about these names; first of all, they can contain non-Latin (for example Hebrew or Korean) characters, but they can't begin with the three letters "X" "M" and "L".

The following are legal names:

- `Tim.Bray`
- `société`
- `w3-c:Parsed_Entity`

The following are not legal names:

- `xmlu`
- `2pac`
- `P=NP`

Back-link to spec

Namespaces

The namespace problem arises when you want to build an XML document using pieces from one or more other documents created by others. When you do this, you can't very well rely on the others to avoid using the same names; for example, a document describing cooking supplies and a document describing inventory control might both the use tag `<Stock>` but mean very different things; and if you wanted to build an on-line store for cooks, this might be a real problem.

The reason the colon comes into play is that the leading candidate for a solution to the namespace problem relies on using colons; in the example above, you might want to distinguish inventories as `<Retail:Stock>` and soup-base as `<Food:Stock>`.

Back-link to spec

Markup

Markup, as you can see from this list, is a word that covers a whole lot of territory; in fact, most of the rest of the spec is little more than a discussion of the different kinds of markup and how they are used.

One of the nice things about XML is markup can easily be distinguished from text because it always begins either with the character < (in which case it ends with the character >) or the character & (in which case it ends with the character ;).

Back-link to spec

Escaping Delimiters

The spec goes into a lot of agonizing detail, but it all amounts to: always use `<` for < (the "lt" stands for less-than, by the way) and `&` for &.

Back-link to spec

Dealing with ">"

If you need to put a greater-than sign in an XML document, that's OK, unless the previous two characters happened to be `]]`. What this means is that if you're generating XML in a computer program, you might as well always use `>`, just to be sure.

Back-link to spec

XML Comments

For those of you familiar with SGML, watch out; XML comments are really different. SGML allows comments to show up in the middle of declarations in a variety of contexts; in XML, they have to stand alone. Also, the trick, popular among some of SGML users, of inserting empty declarations (<!>) is not legal in XML.

Back-link to spec

No Comment Abuse

It's important to note that processors are allowed, if they wish, simply to ignore the comments in a file. This means that if you're building an XML application, you should *never* rely on anything that shows up in a comment (this sleazy trick is far too common in HTML). If you need to put material in a document that is not part of a document, but is available to programs that read it, that's what Processing Instructions, described in the next section, are for.

Back-link to spec

The Example Comment

The point to this example is that the tags such as <head> and the free-floating & cause no trouble, because they're hidden inside a comment.

Back-link to spec

Processing Instructions

XML, just like SGML, is *descriptive markup*; the idea is that you mark things up with labels that say only what they are, not how they should be processed. This buys you all sorts of good things such as flexibility and application-independence.

On the other hand, in the real world, it is often very useful to post a yellow-sticky on a document saying "this is where the background music comes in" or "last time they edited, this was at the top of the screen", or the like.

Processing Instructions (usually called PIs) are the solution. They are in the document, but are not part of the data. They have to begin with a Name, which is supposed to identify the application that might be interested in this PI.

Of course, finding names for things is tricky; that's why the spec recommends that you use a Notation name, which has an associated URI.

The spec says nothing about the content of a PI, aside from the fact that it extends up to the string ?>. However, XML itself uses some PI-like constructs, for example the XML declaration, and in those it uses a syntax much like that of a start-tag. While not compulsory, this seems like a good idea just for consistency.

Back-link to spec

Names With "XML" in Them

This production rules out a PI target of "XML", "xml", or anything in between. Of course this technically allows the use of something like `<?XmlStyle?>`, but that would be unwise.

It might even be illegal; in an outburst of sloppiness, the spec, in its definition of Name, says that names beginning with the letters x, m, l, are "reserved", whatever that means.

Anyhow, don't use 'em.

Back-link to spec

CDATA Sections

CDATA sections don't *mean* anything; they are strictly a convenience to make XML document authors' lives easier. Suppose I want to include an example of some XML in an XML document (this isn't that uncommon, after all I'm about to do it right now). There are two ways I could do this:

```
&lt;tag>Hello, world!&lt;/tag>
```

and

```
<![CDATA[<tag>Hello, world!</tag>]]>
```

These encode strictly identical data, and there is nothing wrong with either. The CDATA technique, though, is probably a lot more convenient for an author, because she can cut chunks of XML out of other documents and drop them right into the CDATA section, without having to worry about escaping all those < and & characters.

Back-link to spec

CDATA Sections and Binary Data

A lot of people would like a way to package up any old binary data and include it in an XML file. The conventional XML answer to this would be to store it separately and point at it with an unparsed entity. Which is fine, but that's not what people want; they want to include the data right in the file, which is a reasonable way to go if you're going to transmit it over the network.

When you look at CDATA, you might get the impression that you could maybe jam your binary data in a CDATA section. You'd be right, but you'd have to guarantee that it never included a byte sequence that looks like `]]>`. There is a trick you can use to get around that, but it's awkward:

```
<![CDATA[Use *two* CDATA sections when you need to embed a
"]]]><![CDATA[>" in the data ]]>
```

Another way to go would be to encode the binary data in base64 or some other technique that's guaranteed never to contain a <; but if you're going to do that, you don't need a CDATA section; any old element would do. Perhaps this is a good use for XML's notation attributes.

Back-link to spec

Why Aren't XML Declarations Compulsory?

They should be, but history got in the way. An XML declaration adds a lot to the value of a document; it makes it self-identifying, and in the internationalized context, self-unpacking. We couldn't bring ourselves to make them compulsory, though, because quite a lot of existing SGML and HTML documents either are, or could easily be made, well-formed XML documents. This wouldn't be the case, though, if we'd required the XML declaration.

However, in all your new XML documents, you should definitely use an XML declaration unless you have a *really* good reason not to.

Back-link to spec

Contents of the XML Declaration

The XML declaration does a lot more than specify the version of XML. To start with, it identifies the document as being XML; this is in the fine tradition of the Unix "magic number" and #! constructs (pipe down purists: I know that #! *is* really a magic number).

This idea, that documents should be self-identifying has also been taken up in recent times by VRML.

Note that the XML declaration also provides a home for the encoding declaration and standalone declarations, discussed a little later.

Back-link to spec

Why Can't The Example Be Valid?

Because there's no `<!DOCTYPE` declaration, which means there's no DTD, and you can't be valid unless there's a DTD to check against.

Back-link to spec

Future Versions of XML?

Will there be future versions of XML? Maybe, maybe not. Toward the end of the development of XML 1.0, the XML WG tossed out a lot of pretty reasonable requests for improvements and enhancements on the grounds that we had to get this job done, and that they could be dealt with in version 1.1.

On the other hand, immediately after the birth of XML 1.0, the parties involved had a massive attack of conservatism and fear. Since the industry acceptance of XML 1.0 had been astoundingly broad and fast, it seemed unreasonable to do any more fiddling with the spec. First of all, it would create confusion and uncertainty among those who were betting on this technology. Second, it seems foolish to charge ahead "making improvements" and "fixing problems", when in a year or so, we are going to have an immense amount of industry experience under our belts, and really know where the improvements need to be made and the problems need to be fixed.

On the other hand, it is absolutely 100% certain that there will be other specifications that are layered on top of XML 1.0. There will definitely include specifications for namespaces, for hyperlinks, and for stylesheeting. It is also a pretty safe bet that some others will exist that we haven't yet begun to dream of.

For the moment, it's safe to base implementations on XML 1.0, and highly unsound to put off developments waiting for some future version.

Back-link to spec

Document Type DECLARATION vs. DEFINITION

Just to make things confusing, there are two things for which you *could* use the abbreviation "DTD". The document type *declaration* is everything between the string `<!DOCTYPE` and the matching `>`. It contains the Document Type *Definition*, which is what is abbreviated "DTD". When people talk about the Document Type *Declaration*, they usually say "doctype declaration" for short, which while not official, is certainly clear.

Back-link to spec

Just Because There's a DTD...

... doesn't mean you have to check for validity. There is a whole class of XML processors, called non-validating processors which, as their name suggest, do not check the document against the contents of the DTD, whether or not it is provided. In particular, such processors are completely free to ignore any parts of the DTD (such as the external subset, but there are others) that are in other entities.

It turns out that *all* processors have to read the internal subset, and on top of that, have to use some of the declarations in it. But the existence of the `<!DOCTYPE` declaration is not a signal to validate.

Back-link to spec

Are The Constraints In The DTD Or The DTD?

This sentence is misleading, since the constraints are really contained in the document type definition (DTD). OK, the DTD is contained in the document type declaration, so the spec is correct, but since the function of the DTD is simply to hold these constraints, it would have been better to say "definition" or DTD.

Back-link to spec

The XML Declaration Must Come First

As the text makes clear, the XML declaration or the text declaration (if you have one) has to be the first thing in the entity (if you're not up on entities yet, assume for the moment that an entity is a file). This does not mean the first non-blank thing in the entity, it means *right at the front* of the entity; the first bytes a program gets when it opens the file and starts reading.

Back-link to spec

XML and Text Declarations are Ordered!

Although the text doesn't mention it, the grammar makes it clear that the things appearing in XML and Text declaration must be in one and only one order. Which is to say that neither

```
<?xml encoding='UTF-8' version='1.0'?>
```

nor

```
<?xml standalone='no' encoding='ASCII'?>
```

are well-formed.

Back-link to spec

Error In Production 24

The quotation marks around the version number are supposed to be literals; thus this should really say:

```
VersionInfo ::= S 'version' Eq ("'" VersionNum "'" |
                '"' VersionNum '"')
```

Nobody noticed this boo-boo until the spec had been out for a month.

Back-link to spec

Document Type Definition (DTD)

Why isn't "document type definition" in bold-face? This is its official definition, and this is what people mean when they bandy about the acronym "DTD". Definitely an editing error here.

Back-link to spec

The External Subset is an Entity

As the spec doesn't make nearly clear enough, the DTD's external subset is an entity; to be precise, an external parsed entity. Bear this carefully in mind whenever we are discussing the rules and behaviors associated with external entities and parsed entities.

Back-link to spec

Keywords Are Upper-Case

One of the consequences of XML's case-sensitivity is that there's no flexibility about the keywords; they must always be in upper-case letters just as they appear in the grammar. That means you have to use `<!DOCTYPE` and `<!ELEMENT` and `<!ATTLIST` and `<!NOTATION`. This also applies to keywords like `SYSTEM` and `PUBLIC`, and those such as `ID`, `CDATA` and `NMTOKEN` that appear in attribute-list declarations.

Back-link to spec

The Grammar and Parameter Entities

This little paragraph is awfully important. Parameter entities (PEs) are things that show up in the DTD as references beginning with `%` and ending with `;`. They are kind of hideous and hard to use, and especially, they are hard to read when you're looking at someone else's DTD. But for the moment, they are really the only way we have to build DTDs in a modular and maintainable way.

The problem in the spec is that the grammar has to try describe both where PE references can appear, and what the document has to look like after they're expanded. A validating XML processor has to expand PEs (according to a whole bunch of rules provided later in the spec); in this case, the grammar describes what the text looks like *after* they have all been expanded.

This attempt to do two things at once leads to some awkwardnesses in the grammar, and to the existence of paragraphs such the one this note is attached to.

Back-link to spec

Proper Nesting of Parameter Entities

This is probably not as clear as it could be. Here is an example of a PE reference that is disallowed by this rule:

```
<!ENTITY
 % pe1 "EMPTY> <!ELEMENT e2 EMPTY>"
 >
<!ELEMENT e1 %pe1;
```

Back-link to spec

Parameter Entity Reference Restrictions

It took a lot of work to get to this policy. The idea is that although PEs are useful (essential, even) for DTD construction and maintenance, that is the *only* place they are useful, and they shouldn't complicate the lives of non-validating processors. So to achieve this, in the internal subset (the part between the `[` and `]` in the doctype declaration), you can only use PEs at the "top level". that is to say, this is legal in the internal subset:

```
<!ENTITY % e1 "<!ELEMENT e1 ANY>"> %e1;
```

while this is not (but would be in the external subset):

```
<!ENTITY % e2 "(e3|e4)">
<!ELEMENT e2 %e2;>
```

Then, to make things even easier, non-validating processors are allowed to skip PE references completely. This, combined, with the fact that in the internal subset, they can only exist at the top level, makes the internal subset very easy indeed to parse.

Back-link to spec

Keeping the Internal Subset Simple

Conditional sections are another item that is absolutely required for basic DTD housekeeping, but too hairy for consumption by a (hopefully lightweight) non-validating processor. So, you'll never see any in the internal subset.

Back-link to spec

The External Subset Can Have a Text Declaration

As you'll see later in the spec, different external parsed entities can be in different encodings of Unicode, and can have Text Declarations to help handle these encodings. Since the external subset is (as the spec doesn't make quite clear enough) an external parsed entity, it can have one too.

Back-link to spec

Point at DTDs with URIs

It is not an accident that the system identifier is a URI; in fact, *all* system identifiers (which are used by Entities and Notations as well) are URIs; this the only kind of external pointer that is used in XML.

Back-link to spec

Local DTDs

The example below is perfectly well-formed and valid, and it might make sense, in a network setting, to *transmit* the whole DTD in the internal subset as in this example.

However, it would almost never make sense to *author* or *maintain* the document in this form. Typically, a DTD is used to describe and control the authoring of many documents, not just one; for this reason it makes sense to store it separately and point at it from each of those documents.

Back-link to spec

Internal vs. External Subsets

It turns out that several kinds of things in XML can have multiple declarations. When this happens, the one that appears first "wins" and subsequent declarations are ignored.

In this case, when one of the declarations is in the internal, and one in the external, subset, the internal subset wins. Instead of just saying this, the spec is playing it a bit cute here and letting this policy fall out of the fact that the internal subset is "considered to appear before" the other. This semantic tap-dance is inherited from SGML.

Back-link to spec

Why the Standalone Declaration?

The problem is that declarations in the DTD can actually affect how an application sees the content of the document. Which declarations can do this is described in exhaustive detail in a list a couple of paragraphs down.

In most cases, this probably isn't a problem; most XML applications will pretty well know whether or not the documents' consumers will be using the DTD or not, and if this matters. But if you want to be sure, you can povide a Standalone Document Declaration (SDD), which tips a receiving application off that reading the DTD might change the document.

Important Note

Neither the presence of the SDD nor its value has any effect on the required behavior of the XML processor or the application. (In fact, due to what is probably an error in the specification, the processor is not even required to *inform* the app as to its value). It is merely a statement of fact about the document, so that if any downstream application needs to be really sure that it is seeing the document exactly the same way as another application that used the DTD, it will know when it has to go fetch and read the DTD.

Back-link to spec

How Be Sure You've Got the Standalone Declaration Right

Don't have a DTD.

If you do, pack whatever you need right into the internal subset. Then put `standalone='yes'` in your XML declaration, and you won't need to worry.

Back-link to spec

White Space that Isn't Really There

The kind of white-space the spec is talking about is that used to "pretty-print" XML documents; you've probably seen this in displays of HTML source code. For example:

```
<p>Little boys, ingredients for:
  <ol>
    <li>Snips,</li>
    <li>snails,</li>
    <li>puppy dogs' tails.</li>
  </ol>
</p>
```

Pretty clearly, the leading white space used to indent the middle lines is really not "part of" the document.

In SGML, there are a whole lot of rules that are designed to allow you to do this kind of pretty-printing without having the white space leak into the document. The rule the SGML-ers use is "White space caused by markup is not significant." This makes sense on the face of it, but when you try to write down precise rules to describe what "caused by markup" means, it turns out that you get into a horrible rat's-nest of complexity (and this is really being unkind to rats).

We in the XML WG decided that this was a good idea, and that the SGML guys just hadn't been smart enough to write the rules down simply and clearly. We discovered that we were wrong, and that it's an impossibly hairy problem; one that would be nice to solve, but nobody has cracked the nut yet.

Back-link to spec

XML's White Space Policy

XML has an incredibly simple rule about how to handle white space, that is contained in this one sentence: "If it ain't markup, it's data." Under no circumstances will an XML processor discard some white space because, in the processor's opinion, it is not "significant".

Let's look at our white space example again:

```
<p>Little boys, ingredients for:
  <ol>
    <li>Snips,</li>
    <li>snails,</li>
    <li>puppy dogs' tails.</li>
  </ol>
</p>
```

An XML processor will pass the application not just the title and the ingredients, but all the white space characters you can see before the `<ol>` and `<li>` tags, and also the line-end characters you *can't* see; in this case, 7 of them. (But note that an XML processor will clean up the line-ends as described in the next section, so while apps are going to have to wrestle with white space, they won't have to deal with CR-NL on windows and CR on Mac and NL on Unix.)

This behavior is going to cause some surprises and problems for XML users and programmers, because we've come to expect (as a result of working with SGML and HTML) "insignificant" white space to auto-magically vanish.

On the other hand, those who've actually worked with real SGML tools will generally approve of XML's behavior, because it has an important virtue, namely that the rule is simple and anyone can understand it: all white space gets passed through, always.

Back-link to spec

White Space in Element Content

Let's look at our white-space example again, but let's attach a simple DTD to the top of the document:

```
<!DOCTYPE p [
<!ELEMENT p (#PCDATA|ol)*>
<!ELEMENT ol (li+)>
<!ELEMENT li (#PCDATA)>
]>
<p>Little boys, ingredients for:
  <ol>
    <li>Snips,</li>
    <li>snails,</li>
    <li>puppy dogs' tails.</li>
  </ol>
</p>
```

If you look at the declaration for the element `p`, you'll see that it can contain text mixed up among its child elements. That means that the leading spaces before the `ol` elements might well be part of the `p`; the XML processor can't do anything special with them.

On the other hand, the declaration for the `ol` element makes it clear that it's not really intended to contain anything but `li` elements; thus any white space that appears between them is probably decorative, intended for pretty-printing. This is what's called element content. In this case, the XML processor is required to tell the application about the special status of this white space.

Important Note: The application isn't required to *do* anything in particular because this white space appears in element content; it's just that the information might well be useful in some situations. Note also that only a validating XML processor can provide this information.

Back-link to spec

Why Does "xml:space" Exist?

`xml:space` exists because the XML WG was influenced by the very handy `<PRE>` element in HTML. This tells the HTML processor not to chew up any of the white space, but to leave it as-is (it normally also has the side-effect of causing the content to be rendered in a monospace font).

Which is a useful capability, and so XML got the `xml:space` attribute.

xml:space Has No Required Effect

That is to say, the existence of this attribute has no effect whatsoever on the required behavior of XML processors or applications. It is simply a convention that authors can use to pass a message along to whoever's downstream, requesting that the white space in some element be preserved.

Back-link to spec

Line End Trade-Offs

The idea here is that a programmer need never have to wrestle with the fact that Windows boxes, Macintoshes, and Unix systems all use different characters to separate lines. Since XML documents will be stored in files on all these systems, and will often be broken up into lines, it's absolutely certain that these documents will use all these different combinations of carriage-return and line-feed.

But as a programmer using an XML processor, you can count on never seeing anything but a single line-feed character separating lines. This means your code will run anywhere.

Since the publication of the spec, we have received a certain number of complaints from Microsoft Windows programmers, who find it surprising and disturbing that the data they receive from the XML processor has "weird, unconventional" line separation. Given the relative number of Windows programmers, it might have been a good idea to adopt the Windows-standard CR-LF as the line separator signal, as opposed to the single LF; but it's too late for that now.

Back-link to spec

Language Identification Is Vital

XML documents are made up mostly of text, which in most cases, is used to carry a message encoded in human language. It turns out that in practical terms, you can do very little that's useful with text if you don't know what language it's written in. Things you can't do properly in a language-oblivious way include:

- Render it on a screen or on paper
- Spell-check it
- Make a full-text index of it
- Support an author properly in an editing program

This may be, on the face of it, rather surprising; doesn't XML use Unicode, and doesn't Unicode encode all the world's characters in an unambiguous way? That's true, and in fact the use of Unicode is one of XML's big advantages; but nonetheless, experience has shown that you still need to know about languages. That's why XML has the `xml:lang` attribute.

Back-link to spec

Language Tagging

There's a potentially tricky terminology problem. People who care about processing multilingual text (which is a lot of people, these days) generally refer to the process of identifying a text's language as "language tagging". This is pretty obvious, but there's a problem, because in XML terms, `xml:lang` isn't a tag, it's an attribute, which can be attached to any old XML tag.

So to summarize, you can language-tag tags, but the tags you tag the tags with are attributes, not tags. So to speak.

Back-link to spec

xml:lang Has No Required Effect

Like the `xml:space` attribute and the Standalone Document Declaration, the `xml:lang` attribute doesn't have any effect on the required behavior of XML processors or applications. It's just a pre-cooked way we provide for authors to provide this information to any downstream applications that might care.

Back-link to spec

Ignore All This Grammar

To figure out how to use `xml:lang`, just go look at the examples at the bottom of this section.

Back-link to spec

Example: Elements

The spec should *really* have an example, to keep this grounded in reality. So here are some example elements:

```
<a href="http://www.textuality.com">Tim's home</a>
<img src='http://www.textuality.com/tbray.gif'></img>
<br/>
```

Back-link to spec

Element Type

The innocent-sounding phrase "element type" is awfully useful. The type, of course, is the name that appears in the start- and end-tags. But there's a big (and useful) difference between "element" and "element type". Here's an example:

```
<p>XML doesn't <em>mean</em> anything, it's just a syntax. But
syntax can be <em>very</em> useful.</p>
```

In the example, there are three elements, but only two element types: `p` and `em`.

If, when we're talking about XML, we're careful to respect the distinction between elements and their types, we'll save ourselves a lot of time and avoid considerable confusion.

Back-link to spec

Markup Has No Semantics

Writers discussing XML and its parent SGML have called upon a wide range of rhetoric and a menagerie of metaphors to try to explain what elements and attributes mean.

Elements and attributes don't mean anything. All they do is break up documents into cleanly identified chunks, and give those chunks names. This is a useful and good thing to do, but trying to figure out what it all means is best reserved for barroom discussions at conferences (in which context it is an extremely worthwhile pursuit.)

Back-link to spec

When An Element Is Valid

This tiny paragraph and four-piece list is central to the concept of validity, and thus central to XML. In prose terms, a document is valid when there's a DTD and the document matches the DTD. A large part of validity amounts to matching element declarations, and this constraint tries to formalize exactly what it means to do so.

Back-link to spec

Element Types Must Be Declared

The first basic ingredient in validity is that all the elements that appear in the document must have been declared in the DTD.

This seems obvious but actually it is becoming controversial; a lot of people would like to be able to do "partial validation" of documents, just checking that the elements that have been declared are valid. There's no way to do this in XML as it stands now.

Back-link to spec

Empty Elements

This means that an element declared EMPTY has to look either like this: `<img src='madonna.gif'></img>` or like this: `<img src='madonna.gif'/>`.

Back-link to spec

Element Content

An element is said to have "element content" if its declaration matches children. The description of what this means only makes sense if you know enough computer science to be comfy with the concept of a "language generated by a regular expression." In English, it means that if, in the DTD, you've given rules for an element saying what other elements that can appear inside it and what order they have to appear in, then the document follows those rules.

There is an important point about white space. If I have a declaration saying, for example that a `OL` element is allowed to contain only `LI` elements, then it's OK for there to be white space between them; this makes it possible, among other things, for the `LI` elements to appear on separate lines.

Back-link to spec

Mixed Content

In XML, you can *either* control the order that child elements of some element appear in (this is called element content), *or* you can say that an element contains both other elements and text (called mixed content).

To make this more obvious, consider the HTML `DL` element, which can really only contain `DT` and `DD` elements, and you have to have a `DT` before each `DD`; this is an example of element content. On the other hand, HTML's `P` element can contain ordinary text mixed up with all sorts of other elements, such as `A`, `B`, `I`, and `IMG`, in any old order.

In XML, if you're going to do mixed content, that's fine, but you can't control what order the child elements, mixed in among the text, appear in. This is based on years of experience with SGML, which allows (while discouraging) such control.

Back-link to spec

ANY

The second part of this rule (saying that the child element types must have been declared) is not really necessary, since we already said that in the paragraph at the beginning of this list. ANY means exactly what it sounds like it means; the element can contain any old thing in any old order, as long as it doesn't break any of the other rules of XML.

Should ANY Ever Be Used?

Back when we were working through the design of XML, I couldn't see any excuse for having ANY, and voted against it. I lost that vote, and that was probably a good thing, because I have since learned of a very useful way to use ANY. Suppose you're given an existing well-formed XML document and you want to build a DTD for it. One way to do this is as follows:

1. Make a list of all the element types that actually appear in the document, and build a simple DTD which declares each and every one of them as ANY. Now you've got a DTD (not a very useful one) and a valid document.
2. Pick one of the elements, and work out how it's actually used in the document. Design a rule, and replace the ANY declaration with an EMPTY or Mixed or Element content declaration. This, of course, is the tricky part, particularly in a large document.
3. Repeat step 2, working through the elements one by one, until you have a useful DTD.

Back-link to spec

Names Can Be In Any Language

Bear in mind that these Names, like all names in XML, are fully internationalized. So it would be perfectly reasonable, in XML, to have tagging such as `<Français>Medoc</Français>` or `<Español>Rioja</Español>`.

Back-link to spec

The Buddha-Nature of Element Types

One can have an immensely amusing argument as to whether the Name that appears in start- and end-tags (and also empty-element tags, as the spec (tsk, tsk) doesn't say) *is* the type of the element, or whether the element's type is an abstract metaphysical what-not which is *named* by the type. This is reminiscent of the debate in Lewis Carroll about a song, its name, what it's called, what its name is called, and so on ad infinitum.

I am pleased by the fact that the wording in the spec ("The name ... gives the element's type") can be used to support either interpretation.

Back-link to spec

Entity References in Attribute Values

The production for Attribute refers to AttValue, which is inconveniently defined elsewhere. But if you look closely, note that you can have entity references in attribute values. In fact, this turns out to be very useful. In the document you are now reading, there are a *lot* of hyperlinks into the XML spec; these are done with an ordinary href attribute, for example:

```
<x href='&spec;id(sec-starttags)'>
```

Then we can use the official copy of the spec at the W3C site, or a local copy on disk, just by changing the definition of the `spec` attribute. Also it saves a lot of typing and disk space.

Back-link to spec

No Doubling Up Attributes

What this says, in case it's not obvious, is that you can't get away with
`<img src='madonna.jpg' src='child.jpg'>`
One of the nice side-effects of this rule, for a programmer, is that all the XML processors and libraries have a function that, given an element and the name of an attribute, can return the value of the attribute, without having to worry about multiple occurrences. Of course, such a function can also signal that there was no such attribute on that element.

Back-link to spec

Valid Attributes

The word "type" seems to have crept unnecessarily into the title of this constraint; it's doing the same work as the much longer and hairier "Element Valid" constraint just above. However, attribute validity is such a rambling and detail-laden affair that there is no attempt to summarize it here as was done for elements.

Do note the important first phrase; an attribute must have been declared to be used in a valid document.

Back-link to spec

No External Entity References In Attribute Values

There's a good reason for this rule. XML goes to a lot of work to make external entities self-identifying and allowing them to use different character encodings; this is the reason for the text declaration. But an efficient XML processor is probably going to treat attribute values in quite a different way, and it's hard to figure out what the right way would be to deal with a text declaration in the middle of an attribute value.

So in XML, it's internal entities only in attribute values.

Back-link to spec

Banishing the <

This rule might seem a bit unnecessary, on the face of it. Since you can't have tags in attribute values, having an < can hardly be confusing, so why ban it?

This is another attempt to make life easy for the DPH. The rule in XML is simple: when you're reading text, and you hit a <, then that's a markup delimiter. Not just sometimes, always. When you want one in the data, you have to use `<`. Not just sometimes, always. In attribute values too.

This rule has another unintended beneficial side-effect; it makes the catching of certain errors much easier. Suppose you have a chunk of XML as follows:

```
<a href="notes.html> <img src='notes.gif'></a>
```

Notice that the `notes.html` is missing its closing quote. Without the no-`<` rule, it would be really hard to detect this problem and issue a reasonable error message. Since attribute values can contain almost anything, no error would be detected until the processor finds the next quotation mark. Instead, you get an error message the first time you hit a <, which in the example above, as in many cases, is almost immediately.

Back-link to spec

Content

There are a couple of things about this one-liner that might not be obvious. First, the "text" that makes up an element's content isn't just character data, but also tags and any other kind of markup.

Secondly, there is potential for confusion between the content of an element, which is whatever tags and text that it contains, and the very specific term element content.

Back-link to spec

Two Ways to be Empty

So, is this: `<img src='madonna.gif'></img>` really exactly the same as `<img src='madonna.gif'/>`? As far as XML is concerned, they are. This decision was the occasion of much religious debate, with some feeling that there is an essential element between "point" and "container" type elements. And as the "for interoperability" note below makes clear, if you are using pre-1998 SGML software to process XML, there is a big difference.

Back-link to spec

Why Does The Empty-Element Tag Exist?

The existence of the empty-element tag is sort of an accident of history. In SGML, there is no difference at all, so you can't tell whether `<IMG src='madonna.jpg'>` is a start-tag or an empty element without looking at the DTD. In XML, this is unacceptable, since you have to be able to parse documents received over the network without first fetching a DTD or any other external help. So we invented the empty-element tag syntax as a way to make such elements obvious and parsing easy.

Later on in the process, we realized that there was no difference between the two forms of empty elements, and thus no real strong need for the empty-element tag; in fact, it's just a shorthand, space-saving abbreviation. Had we been keeping strictly to our design principles, we would have tossed it out of the language; but by that time we'd had it for a few months, and people really seemed to like it.

Back-link to spec

Empty HTML Elements In XML

Suppose you want to write some text which is at the same time both HTML and XML. Well, this isn't too hard; all you have to do is put in all the tags you usually leave out (like for example `<BODY>`), quote your attribute values (no more `<P align=right>`), and you should be OK.

There are some problems, specifically HTML's empty elements such as `IMG`, `HR`, and `BR`. Of these, you can crack the `IMG` and `HR` nuts simply by putting in end-tags, for example `<IMG SRC='madonna.jpg'></IMG>`; which looks a little weird, and will not be valid for some SGML parsers, but won't cause a browser any heartburn.

`BR` is a real problem, though. Sure, you can double it up like so: `<BR></BR>`, but most browsers, when they see this, will stupidly put in *two* blank lines, which really uglifies your page. However, it turns out that if you use `<BR />` (there's a space between the `BR` and the `/`) then everything works just fine. The guys at Netscape once explained to me *why* this works, but thank goodness I had signed a non-disclosure agreement, because it's not a pretty story.

Back-link to spec

The Future Of DTDs

Declarations, as this paragraph says, constrain elements' content. But not as much as we'd really like. All they really control is which types of elements can be children, what order they appear, and whether they have text in between them.

A lot of people would like to do a lot more constraining; to declare, for an element named `<DateOfBirth>`, that it must contain a valid date string, that must be earlier than the current date.

At another level, people would like to say that the element `<Golfer>` inherits all the attributes and declarations from the element `<Person>`, then adds one or two more such as "Handicap".

There is quite a lot of work under way to figure out how to build a document declaration facility that is more advanced than what today's DTDs offer.

Back-link to spec

You Can Use Undeclared Elements?

What this means is that I can declare that an `E1` element can have child elements of type `E2`, even if I haven't declared element type `E2`. Mind you, if an `E2` actually shows up in the document without having been declared, that would still make the document invalid. This rather contradictory-smelling rule's existence is due to high-intensity demands from co-editor Michael Sperberg-McQueen, who, in another life, has used this underhanded trick to very good effect in the design of large, complex, modular DTDs.

Back-link to spec

Understanding Element Declarations

This section, you'll discover, doesn't contain any explanation of what EMPTY and ANY mean. You can find that above in the discussion of what validity means.

Mixed is discussed in the section after next, which is entitled (logically enough) *Mixed Content*. Finally, children is discussed in the immediately following section, entitled *Element Content*. Obvious, right?

Back-link to spec

Element Declarations Once Only

This is in contrast with both entities and attributes, for which multiple declarations are allowed, and in fact commonly used. There are arguments both for and against this policy, but SGML does it this way, and it doesn't seem to cause problems.

Back-link to spec

Element Content Example

This example is more than a little puzzling; at least to those who are reading the spec in order and haven't boned up on parameter-entity (PE) references. PE references are the things between the % and ;; they are a special kind of entity reference, i.e. macro of no arguments, for use in the DTD. Because PE references are expanded before they are actually parsed, this doesn't really show an example of an element declaration. On the other hand, if you look at a real declaration in a real commercial DTD, this kind of thing is what you'll mostly see.

Also, it's worth noting that this example would be flat-out illegal in the internal subset; it would have to occur in the external subset or an external parameter entity.

Back-link to spec

Explaining Element Content

The verbiage surrounding the productions here (47-50) doesn't, I find, help very much. Neither do the productions; they are mostly there to help programmers understand the way these things fit together. To learn how to use element content models, look at lots of examples; there are two or three a little later in this section.

One note of real-world caution: although the content-model grammar allows you to build insanely complicated rules, most people don't, most times, thank goodness.

Back-link to spec

Deterministic Grammars

This stuff is not worth worrying about. This rule was inherited from SGML; its inclusion in SGML was actually a design error. This was retained in XML not only for compatibility with SGML (not quite a good enough reason; I voted against it) but because some of the most popular existing SGML tools actually rely on it for certain internal optimizations.

It's likely that quite a few XML products will never bother checking for violations of this rule, because it's hard; *if* you're writing a DTD *and* you get a complaint about a nondeterministic content model, *then* you might find it worthwhile to read the appendix.

Back-link to spec

Well-Behaved Parameter Entities

This entity rules out things like:

```
<!ENTITY % p1 "(A|B">
<!ENTITY % p2 "|C|D)">
<!ELEMENT X %p1;%p2;>
```

This makes life easier for people who write XML editors and other authoring software tools; managing parameter entities is tough enough for these people without having them spill all over the place.

Back-link to spec

#PCDATA Means What?

An observant reader wrote in to say that while the grammar makes clear that the keyword #PCDATA is used to signal mixed content, the spec says nothing about where that magic word comes from. The # is there because it can't be used in a Name, so #PCDATA could never be the type of a child element.

The string PCDATA itself stands for "Parsed Character Data". It is another inheritance from SGML; in this usage, "parsed" means that the XML processor will read this text looking for markup signaled by < and & characters.

Back-link to spec

Attribute List Declarations

If these were being designed today, we probably would try to avoid gluing together the attribute name, type, and default information in a package with the element type in this way. It's easy to imagine a situation where we have a pool of attribute names, types, and defaults, and then we assign these to particular elements, but that's not what we inherited from SGML

One way in which SGML is ahead of XML is that it has recently added a facility where you can declare an attribute and say it can attach to *all* the element types in the document. This would be useful in all sorts of situations, and maybe we should consider adding it to some future version of XML.

Back-link to spec

Undeclared Elements in Attribute-List Declarations

Of course, if an element of this undeclared type actually showed up in the document, that *would* be an error - a validity error, to be precise.

Back-link to spec

Multiple Attribute-List Declarations

Why would you want multiple declarations? Here's one scenario: normally, all your attribute-list declarations are placed (quite properly) in the external subset of the DTD, so that you can use those declarations to validate lots of different documents.

But these declarations are not just used for validation; they are also very useful for giving attributes default values, which increases maintainability and decreases document size. The key point is that you can do this even when you're not validating. But a non-validating processor is not required to read anything but the internal subset, so if you have any attributes you want to declare defaults for, you might want to insert copies of those declarations in the internal subset for transmission over the network. But doing this would make the document invalid, because the declarations would be duplicated in the internal and external subsets.

Rather than work around this, XML decided, after consultation with the SGML community, simply to drop SGML's existing rule against duplicate attribute lists. While the SGML standard has been revised to make this legal in SGML as well, the pre-1998 inventory of commercial SGML tools absolutely won't allow it.

Back-link to spec

Some Non-Interoperable Declarations

This sentence perhaps tries to pack too much into too little space. Suppose you have a declaration as follows:

```
<!ATTLIST e a1 CDATA #IMPLIED>
```

Then, none of the following would meet the conditions here for interoperability:

```
<!-- 2nd attribute-list declaration for element type "e" -->
<!ATTLIST e a2 CDATA #IMPLIED>
<!-- 2nd declaration for attribute "a2" -->
<!ATTLIST e a2 IDREF #REQUIRED>
<!-- empty attribute-list declaration -->
<!ATTLIST e >
```

Back-link to spec

Attribute Declarations and Proust

I have taught half-day and whole-day courses in XML on many occasions. One of the big problems with the full-day course is that after you've done some history and introduction and goal-setting, then talked about elements, you get to this point (attribute declarations) around lunch time.

The consequence of this is that when the class comes back after lunch, to listen the discussion of attribute types and their declarations, most of them go to sleep. The fact of the matter is that there are a lot of attribute types (I voted against a few of them), there are lots of relevant details, and it is pretty tedious. Bear in mind also that since all this stuff lives in the DTD, it is really of interest only in the context of validation, and if you're working with downstream non-validating applications, you can pretty well ignore all this attribute type stuff.

Back-link to spec

ID and IDREF Attributes

ID and IDREF attributes together provide a simple inside-the-document linking mechanism, with every IDREF attribute required to point at an ID attribute. ID attributes, though, are important, whether or not you have any IDREFs, because they give elements unique addresses. They have a very important use in the future as the targets of hyperlinks coming into the document from outside.

In general, it's a good idea to attach ID attributes to as many elements as possible in every document, because, later, if you decide you need to point at anything, you'll be happy if it has an ID attribute ready for use.

However, there's a real problem here. In principle, you can't know whether an attribute is of type ID unless you're a validating processor and thus required to read the DTD, where the type is declared. On the other hand, we know that there are many non-validating applications which could make very good use of IDs.

Thus there is a real requirement that XML figure out a way to identify ID attributes without relying on the DTD. One of the simplest ways to do this would simply be to assert that attributes named `id` are of type ID unless declared otherwise. Quite a few applications (including the one that generated this annotated spec) are already doing this, de facto.

Back-link to spec

One ID Per Element Type

In my opinion, this rule should really have a "for compatibility" attached to it, because while it is a rule in SGML, I've never understood why an element shouldn't have two different IDs.

Back-link to spec

No Defaults for ID Attributes

It doesn't make any sense to give an ID attribute a default value. The whole point of an ID attribute is to have a one-time-only address for a particular element in the document.

Back-link to spec

Unparsed Entities

Unparsed entities are a way to add pointers to your document that point at different kinds of data; the most common application would be for nontextual objects like graphics, video, and other multimedia. To use an unparsed entity, you have to do four things:

1. Declare a notation, which describes the format. Examples would be JPEG, PNG, and VRML.
2. Declare the entity; you have to identify its notation by name and its location by URI.
3. Declare an attribute for some element to be of type ENTITY.
4. Insert one of those elements, provide the attribute, and use the entity as its value.

Here's an example laid out step-by-step:

```
<!-- 1 -->
<!NOTATION vrml  PUBLIC "VRML 2">
<!-- 2 -->
<!ENTITY Antarctica SYSTEM "http://www.antarctica.net" NDATA vrml>
<!-- 3 -->
<!ATTLIST World src ENTITY #REQUIRED> <!-- 4 -->
<World src='Antarctica'>
```

Back-link to spec

Notation Attributes

The idea is, sometimes, although an element contains ordinary-looking (as far as the XML processor can tell) text, that text needs to be interpreted in some particular way. Suppose for example, that it is PostScript, or VRML, or Base64-encoded image data. A good way to handle this situation (when you are validating) is to declare notations for these kinds of text, and then an attribute of type Notation to identify them when they occur.

Here's an example:

```
<!NOTATION ps PUBLIC "PostScript Level 3">
<ATTLIST Rendered In NOTATION (ps) #IMPLIED>
<Rendered In='ps'>gsave
112 75 moveto 112 300 lineto
showpage grestore
</Rendered>
```

Back-link to spec

Enumerated Attributes

The spec has almost no advertising for one of the most useful things in XML, enumerated attributes. Suppose I have a declaration saying:

```
<!ATTLIST List Type (numbered|bullets) #REQUIRED>
```

Then every `List` element has to have a `Type` attribute saying whether it's numbered or bulleted; that is to say, the attribute has only two possible values, which are given right there in the declaration. You'll see a lot of this in real-world XML documents; an authoring system can make good use of this; when the author puts in such an element, she gets to pick from a menu of the possible attribute values.

Back-link to spec

Multiple Enumerated Attributes With The Same Value

What this "for interoperability" is saying is not obvious, and to understand why it's here requires a little bit of backtracking. SGML was designed with immense concern for saving keystrokes and disk space. For this reason, it has a lot of "minimization" features, allowing you to leave out pieces of markup. One of the things you can minimize away is the attribute name; so suppose you had a declaration like this:

```
<!ATTLIST List Ordered (yes|no) #REQUIRED>
```

Then, using minimization, I could do the following: `<List no>`; there's no reason to put in the `Ordered=`, you see, because it's "obvious." That's OK, until I expand that attribute list a little bit:

```
<!ATTLIST List
 Ordered (yes|no) "yes"
 Secret (yes|no) "no">
```

Then, what does `<List no>` mean? To get around this problem, SGML made that declaration illegal; you couldn't have two of these enumerated attributes that shared a common value. Since XML doesn't have minimization, it just isn't a problem. And since most users of SGML don't *do* minimization, it hadn't been a problem there either; but the inability to have declarations like this certainly had been.

So as with other such problems in XML, we worked with the SGML committee and got agreement to relax this restriction in SGML, allowing us to bypass it in XML. But remember, as with all the other "for interoperability" rules, there is a large installed base of SGML tools that are not forgiving on this point.

Back-link to spec

If An Attribute is #IMPLIED In The Forest, Does Anybody Care?

All #IMPLIED means is what it says here, that this attribute, while it's allowed for this element type, doesn't always have to be there, and that if it isn't, there's no default value, it just isn't there.

The term #IMPLIED is inherited from SGML, which explains that in this case the application should "imply" the attribute's existence or nonexistence; the discussion is somewhat metaphysical in tone and has puzzled generations of students of SGML.

The real reason #IMPLIED exists, of course, is to make declarations easy to parse; every attribute declaration has a name part, a type part, and a default part (which may, if it's #IMPLIED, be effectively absent).

Back-link to spec

#REQUIRED, #IMPLIED, and Real Defaults

This is called the default declaration, but it doesn't always declare defaults. #REQUIRED and #IMPLIED are escape-hatches, the first saying that defaults are irrelevant because the attribute has to be there, the second that defaults are irrelevant because if the attribute isn't there, it just isn't there. Note that #REQUIRED only applies in valid documents (a missing #REQUIRED attribute is a *validity* error), but #IMPLIED often applies, because even a non-validating processor is required to do attribute defaulting for any declarations it actually reads.

If the default declaration is #REQUIRED or #IMPLIED, there is no default; this may sound obvious, but people have gotten confused on the basis that because there's a default declaration, then #REQUIRED must in some sense be a "default."

Back-link to spec

Why #FIXED?

The idea of an attribute that has a value that's fixed and can't be changed strikes many as odd, the first time they encounter it. But there are common applications; suppose, for example, that I insert a pointer here to my favorite piece of the XML spec. When I do this, I use an element named `Sref` (for Spec-ref), which becomes an `A` element in the HTML version. However, in all cases, I want that pointer to get another attribute `target='spec'` ", so that when you use it, it takes effect in the left-hand frame, which is named `spec`.

So that I don't have to enter that attribute in each `Sref` element, the internal subset contains this:

```
<!ATTLIST Sref target CDATA #FIXED "spec">
```

There are lots of places where #FIXED attributes are useful, and you can expect to run across them regularly.

Back-link to spec

How Attribute Defaults Work

This last sentence troubles many. Is it really necessary to introduce the behavior of the XML Processor in order to explain how default attributes work? Some have argued that the spec would be better if it simply defined carefully what the data in an XML document is; one of the things that affects this is whether or not attributes are being defaulted.

This argument has something to recommend it, particularly in terms of precision and conciseness. On the other hand, many find the explanation here, which in fact does describe what any reasonable program has to do about attributes, an easier explanation to grasp. This remains an open issue.

Back-link to spec

Valid Attribute Defaults

This constraint is really an example of belt-and-suspenders let's-be-really-sure thinking. What it means is that you can't do this:

```
<!ATTLIST El At NMTOKEN "$$$">
```

Back-link to spec

Normalizing Attribute Values

As the four-part list suggests, this is actually a bit on the tricky side. But it's another area where all the tricky bits have been assigned to the XML Processor so as to make the application's life easier.

Attribute values are normally short and simple, except when they're not. The idea here is that you can generate XML that looks like this:

```
<!ENTITY home 'xml.com'>
<El At="
found at
&home;
">
```

Then all the application will see for that attribute value is
`found at xml.com`.

Back-link to spec

Complexity Of Conditional Sections

Conditional sections are another SGML invention that got into XML because the people on the committee who had experience writing DTDs claimed that life would just be impossible without them.

Unless you're going to be either writing a validating parser, or designing a lot of DTDs, you really can skip this section. In the latter case, a better way to learn about this stuff is to read one of the excellent books on the subject.

Back-link to spec

Conditional Section Keywords

This little paragraph, and the example just below, document the key trick that makes conditional sections useful. Rather than having the actual words INCLUDE or IGNORE as the keyword, normally you'd have a parameter-entity reference, so that you could, by redeclaring that entity in the internal subset, switch back and forth between conditional sections.

Back-link to spec

The Document Entity

The document entity is an important concept, particularly in networked applications. On the network, it's typically expensive to send documents around in multiple pieces; in most cases, you'd prefer to send just one chunk down the wire. That chunk is the document entity, and it's all that a non-validating XML processor is required to read. Even if you *author* your documents in a distributed modular fashion in multiple independent pieces, you probably need to compose them so that for delivery, each document entity is a more or less complete package.

Back-link to spec

An Entity Taxonomy

The preceding four paragraphs talk about a lot of different kinds of entities; and another classification, internal vs. external, will show up in another section or two.

One might conclude that this profusion of entity categories is a symptom of a rather ad-hoc design, and this may well be true; entities are used for a lot of different purposes, and their use involves three distinct syntaxes.

To summarize, there are the following ways to categorize entities:

- Parsed vs. Unparsed
- General vs. Parameter
- Internal vs. External

This makes a total of eight possible combinations; but it turns out that neither internal/unparsed nor parameter/unparsed can happen; so in fact the number of different entity categories is five:

1. Internal parsed general
2. Internal parsed parameter
3. External parsed general
4. External parsed parameter
5. External unparsed general

This is why it's fine to refer simply to an "unparsed entity" or to an "internal parameter entity". It is important to note, though, that unparsed entities are general entities in that they share a name-space with other non-parameter entities.

Back-link to spec

Character References

These are very handy indeed for dealing with (slightly) multilingual texts in your basic monoglot-American computing environment. ASCII, remember, qualifies as Unicode, in its UTF-8 flavor. Suppose you live in an ASCII world, and your name is Tim Bray, and you really need to include the Arabic version of your name in some document. Nothing to it, just say:

```
&#x62a&#x645; &#x628;&#x631;&#x627;&#x64a;
```

Of course, numeric character references do not provide a real solution, for example, to the problem of writing an all-Arabic document. For that, you'd need an XML-enabled Arabic word processor.

Back-link to spec

Why Hexadecimal?

In SGML, this kind of character reference uses decimal numbers. XML added the kind beginning with &#x (in co-operation with the SGML committee, which has also added them to SGML) not because we are programming geeks who believe in torturing the innocent, but because in the Unicode book all the character tables are labelled in hexadecimal. This way you can refer back and forth between XML documents and the Unicode spec without having to do any conversions.

Back-link to spec

What Does "As Delimiters" Mean?

It means that an entity reference begins with `&` and ends with `;`, for example `&home;` or `&copy;` or `<`.

Back-link to spec

Undeclared Entities as Well-Formedness Errors

All this is just trying to say that when a document contains an entity reference (i.e. a & followed by a Name, followed by a ;), the entity has to be declared, *EXCEPT*...

The "except" is all about the difference between well-formedness and validity. Notice that we are currently in a *well-formedness* rule. This means that we're not validating; and non-validating processors don't have to read any external entities, including the external subset of the DTD.

So it's perfectly possible that some entity might have been declared, but that the processor has (quite rightly) not read that declaration. So this list of conditions basically lists all the conditions in which a non-validating processor can be confident of having read all the declarations; in these cases, *then* if you hit a reference to an entity that wasn't declared, it's an error. In fact, since this is a well-formedness rule, it's a fatal error, which means you have to stop processing the document.

Back-link to spec

Undeclared Entities as Validity Errors

If you read (and understand) the note just above, you might reasonably ask why this set of rules also has to be complex. This is a *validity* constraint, right, and if you're validating, and find an undeclared entity, that should *always* be an error, right?

Well yes, but sometimes it's a validity error (which you can survive, and keep processing) and sometimes it's a well-formedness (i.e. fatal) error. The reason all this verbiage is here is to make that the "undeclared-ness" of some entity isn't both a validity and well-formedness error. So all this language is simply the mirror-image of that in the well-formedness constraint above, and restricts this rule to the cases where that one doesn't apply.

Back-link to spec

No &-References to Unparsed Entities

In other words, you can't do the following:

```
<!ENTITY Madonna SYSTEM 'Madonna.jpg' NDATA Gif>
...
<p>The picture: &Madonna;</p>
```

The reason is that nobody really understands what it means to embed unparsed (possibly binary, as in the example) data in the middle of some text.

Back-link to spec

Recursive Entity References Are Death

But in case it isn't, this rule means you can't get away with:

```
<!ENTITY oops 'He said: "&oops;"'>
```

Or this:

```
<!ENTITY oops 'He said: "&WhatHeSaid;"'>
<!ENTITY WhatHeSaid '&oops;'>
```

Back-link to spec

PE References Outside the DTD

This constraint is not actually wrong, but it is rather misleading. Suppose I have a parameter entity named `Fred`, then if the string `%Fred;` appears somewhere in the document, outside of the DTD, that's not an error as this suggests; it's just the string `%Fred;`.

Back-link to spec

Multiple Entity Declarations

The first-declaration-wins rule is found surprising by many, but it has proved to work pretty well over the years of SGML experience. The practical effect is, that since the internal subset of the DTD is considered to appear "first", you can put entity declarations there to override those in the external part of the DTD.

Back-link to spec

Views of Internal Entities

Computer programmers tend to think of internal entities as macros of zero arguments. Publishers tend to think of them as "boilerplate".

One of the nice consequences of the rules for conformance is that every XML processor in the world is required to do internal entity processing, *if* the entities are declared in the internal subset. Internal entities are incredibly useful in many situations; here are a few examples:

- Little chunks of boilerplate text that you're going to be using all over the place, which it's nice to store and manage in only one place, thus improving accuracy and saving network bandwidth. For example: `<!ENTITY Copyright ';&copy; 1998, Tim Bray'>`
- In URLs; many XML documents contain lots of URLs; as we all know, URLs tend to move around and are painfully difficult to maintain. It is really useful to use entities to help make this easier: `<a href='&home;/bin/wr.pl'>`.
- Naming Unicode characters; of course, you can refer to them by number, but it's much nicer to say `<!ENTITY mdash '&#x2014;'>`

Back-link to spec

Recognizing External Entity Declarations

Yes, if it's not internal, it's external; but in practice, it's easy to recognize an external entity because in the declaration, the entity name is followed, not by a quoted string, but by the keyword `SYSTEM` (which must be in upper-case, remember).

Back-link to spec

What's NDATA?

This keyword is inherited from SGML, where it used to stand for "Notation Data"; but in general, a lot of people have come to call these things "NDATA Entities". "Unparsed" is really a much better name.

Back-link to spec

What's a Notation?

It's just a name, with an associated public identifier, which can be applied to unparsed entities and also, when used in a `NOTATION` attribute, to the content of elements.

The name itself doesn't mean anything, but (with luck) it or the public ID will tell how to deal with the data to which it's attached.

Back-link to spec

Entities and their Identifiers

If you are going to use any external entities, you have to declare them, and in that declaration, you have to give a "system identifier", and that system identifier has to be a URI, and whoever is using your document has to be able to use that URI to retrieve the contents of the entity.

A URI, of course, is (in the year 1998, in practical terms) just a URL with a party face.

Since you can't tell if a URL is any good by looking at it, the only way to check it out is to try it, using (if you're a programmer) your handy local library (e.g. *java.net* for Java geeks), or (if you're a real person) your handy local Web browser.

This is quite a bit different from SGML, where you can stay away from the fragility and general icky-ness of URLs by using "public identifiers", which are just a label that you look up in a local catalogue, or just auto-magically know about.

Public identifiers are a really good idea, and it would be good if XML had them. However, at the time we designed XML, there was really no generally-agreed-on way to resolve public identifiers to actually find the data. And URLs are certainly not perfect, but nearly everyone knows what they mean, more or less, and pretty well everybody has machinery on their desktop that they can use to retrieve them.

So, in XML you can use public identifiers, but you still have to provide a system identifier. If you do something silly like try using an empty identifier, or one that doesn't work, anyone to whom you send the document has every right to complain that it's broken.

Back-link to spec

What Is and Isn't in a URI

You have to have read the RFCs that define URLs with a pretty sharp eye to have spotted this one. What it means is that if you have `<a href='Start#part1'>`, then `part1` isn't part of the URL. But if you have `<a href='Start?q=a23'>`, then `q=a23` *is* part of the URL.

This means you can do things like `<a href='Start?q=a23#part1'>`, but it means that you really can't use an address containing a # to refer to an external entity.

Back-link to spec

Non-ASCII Characters In URIs

What this says is that if you're pointing at an entity and the URL in your system identifier contains some non-ASCII characters, for example `société.html`, the processor, before trying to use the URI, should convert the string to UTF-8 (which will yield 2 or more bytes for each non-ASCII character) and then for each of those bytes, express it as the character "%" followed by two hex digits. In the case of `société.html`, the character `é` is `#xe9`, which in UTF-8 would be the two bytes `#xdd`, `#x81`; thus the processor should encode this URL as `soci%dd%81t%dd%81.html`.

In fact, if you want to be really safe and follow the letter of the law, you (or your editing software) should do this before you store the URI in the XML document, because the letter of the law says that URLs really aren't supposed to contain any non-ASCII characters at all.

This may seem a bit clumsy, but it's consistent with the basic rules that are supposed to be followed by other Web software.

Back-link to spec

Public Identifiers Are Non-Portable

Once again, public identifiers are a trick inherited from SGML that are probably only useful to people who already have working SGML software installed. Remember that if you use public identifiers within your own organization, that's perfectly OK, but if you want to interchange XML documents with anybody external, they have the right to demand, and you have the obligation to provide, a working system identifier (URI) for each external entity.

Back-link to spec

The Text Declaration

The reason this exists, of course, is to provide a place for the encoding declaration so that each parsed entity can signal its own character encoding.

Back-link to spec

The Document Entity is Special

The differences between the document entity and any other external parsed entity are:

1. The document entity can begin with an XML declaration, other external parsed entities with a text declaration.
2. The document entity can contain a document type declaration.

Back-link to spec

Well-Formed Internal Entities

Which means that you can't have something like `<!ENTITY Bad "<a href=">`

Back-link to spec

The Byte Order Mark

The BOM is a really helpful idea. To start with, it helps avoid confusing 8-bit and 16-bit character encodings. If you try to read a BOM-flagged 16-bit file in 7-bit or 8-bit mode, the first two characters are going to look like either -1 and -2 or 254 and 255 depending on how you look at them; either case is a good signal that something is seriously wrong.

Once you know you're in 16-bit mode, the BOM is still helpful. This is because computers internally often treat 16-bit numbers as pairs of 8-bit numbers. Then, when they transmit a 16-bit number, sometimes they transmit the two 8-bit halves low-half-first, sometimes high-half-first. The rules say they should transmit the BOM (#fffe) as (#ff, #fe). But if you look at the first two bytes of a file and see (#fe, #ff), this is a really reliable hint that the bytes are being swapped, and will in many cases allow the processor to read the file.

Back-link to spec

Using the Encoding Declaration

This rule is here because it allows the use of tricks such as those described in the appendix on Autodetection of Character Encodings. We recognize that although the Web provides a method for a server to tell the client what kind of encoding is being used, sometimes it breaks down, and sometimes there's no server (like when you're reading something straight off a disk).

In these situations, everything works much better if entities give the processor some help in figuring out how things are encoded.

Back-link to spec

Other Encodings

If you want to use an encoding that isn't listed in this paragraph, you should do as it says and go look in the IANA encodings list. Mind you, that list is far from easy to use.

It would seem that most commercial products are likely to be able to handle ASCII, ISO-8859-1, and a couple of JIS dialects, along with the UTF encodings. So it would be a very good idea to stick with one of these.

Back-link to spec

Is A Broken Encoding Declaration An Error?

This sentence represents the outcome of a whole lot of passionate debate. Here's the problem: often, when you transmit something over the network (the two best-known recipients are email and a web browser), you normally send along some extra information saying how it's encoded. This is usually packaged up in something called a MIME header (the "HTTP headers" used on the Web are very similar).

This means that an XML entity might signal its encoding in two different places; internally, with a Byte Order Mark or an encoding declaration, and externally, with one of these headers. While these should normally be in agreement, sometimes they aren't; for example, some Web servers actually change the encoding of some pages as they transmit them; if this (it's called "transcoding") happens, then the encoding declaration would probably become wrong.

Also, some Web servers are just broken, and do stupid things like claiming that everything is in ISO-8859-1 without even checking to see whether this is true. In these cases, the encoding declaration would be right, the header wrong.

Thus this sentence. What it says is that while an incorrect encoding declaration is an error, if an entity comes down the pipe with an external label that allows a processor to read it, then the processor is not allowed to toss it on the floor because of a broken encoding declaration. In other words, the receiver of the entity shouldn't be penalized because an upstream Web server did something stupid.

While this rule is sensible, it can lead to some problems. Suppose some document is in "Shift-JIS" and has an encoding declaration saying so. Then it gets served out by some Web server that translates it into "EUC-JIS" and sends along a header saying (correctly) that it has done so. In this case, presumably the XML processor reads the entity correctly using the header, ignores the (now incorrect) encoding declaration, and everything is fine. Fine, that is, until the user saves the file to the disk. Now it has a broken encoding declaration, but there's no header to help work around the problem. The moral of this story is that the program that read the entity should probably adjust the encoding declaration before saving it.

You can get the same kind of error for an entity which was in pure ASCII and (legally) has no encoding declaration; if it gets transcoded into something else, the absence of the declaration becomes an error.

There is another problem, even more pernicious. It turns out that there are some encoding declaration errors that just can't be detected. For example, if you had a general entity stored in EBCDIC which didn't declare itself, and it was referenced in the middle of an ASCII XML document, if the EBCDIC didn't have any bytes that looked like `<` or `&`, then there would be no way for the XML processor to spot the error.

To summarize, the lesson is that the processor should work with whatever information is available to figure out how to decode external entities; and system architects should bear in mind that on the World Wide Web, you can never be absolutely 100% sure of avoiding being bitten by encoding errors.

Back-link to spec

The Trouble With ASCII

As the spec says, pure ASCII files are UTF-8 as they sit, and thus don't require an encoding declaration. The problem is that a lot of ASCII files are not quite pure. Modern Microsoft operating systems, in particular, make it easy to type in words like "Español", with the "ñ" encoded according to some "code page" that may or may not line up with ISO 8859-1 or any other standard.

In the document you are now reading, the "ñ" is encoded as #f1, which happens to be correct per 8859; but is definitely *not* legal UTF-8. Thus this document, to be well-formed, really needs to have an XML declaration like this:

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

Back-link to spec

The Big Entity Processing Table

Well, we could say this is complex because structured texts are rich, complex, objects, and any system that can capture their essence will of necessity have some complexity.

Or we could claim that the design of SGML used the `&...;` syntax for at least three different purposes, and furthermore, introduced more complexity (and the need for two types of entities) by using a different syntax for the DTD and the document itself.

In any case, the design processes for SGML and then for XML led to a situation where there are a lot of different things that can happen. This table brings the descriptions of them all together in one place; it has received good reviews from XML implementers.

Back-link to spec

Example: Reference In Content

```
<p>He said: &WhatHeSaid;</p>
```

Back-link to spec

Example: Reference In Attribute Value

```
<A HREF='&home;/start.html'>
```

Back-link to spec

Example: Occurs As Attribute Value

```
<!ENTITY Antarctica SYSTEM "http://www.antarctica.net NDATA vrml>
<!ATTLIST World src ENTITY #REQUIRED> ...
<World src='Antarctica'>
```

Back-link to spec

Example: Reference in Entity Value

```
<!ENTITY PLX "Perl &heart; XML!">
```

Back-link to spec

Example: Reference in DTD

```
<!ELEMENT %Para; (#PCDATA|%ParaBits;)*>
```

Back-link to spec

Entities are Processed Recursively

This means that the replacement text of an entity (except when in an attribute value or entity declaration, see below) is processed as if it were part of the document; in particular, this means that it can contain other entity references, which are processed recursively. This means that in the following:

```
<!ENTITY AC "The &W3C; Advisory Council">
<!ENTITY W3C "WWW Consortium">
```

The reference `&AC;` would result in the string `The WWW Consortium Advisory Council` appearing as character data in the document.

Back-link to spec

Why Are External Entities Included Optionally?

In discussion of external entities, we realized that the semantics of external text entities (compulsory inclusion at the point where they are encountered) are deeply incompatible with the desired behavior of Web browsers. Consider the following example of the beginning of an XML document:

```
<?xml version='1.0'?>
<!DOCTYPE doc [ <!ENTITY MSA SYSTEM
"http://www.microsoft.com/press/311.xml">
<!ENTITY NSA SYSTEM "http://home.netscape.com/PR/x27.xml">
]>
<doc>Netscape today announced that &NSA;. In response, Microsoft
issued the following statement: &MSA;.
...
```

A Web browser is typically making an aggressive effort to display text to the user as soon as possible, in parallel with fetching it from the network. In the example above, if a browser were required to fetch and process all external entities, it could only display the first four words before starting another network fetch operation. To make things worse, bear in mind that the replacement text for the entity NSA could well include other external entities which in turn would need to be fetched.

This type of situation is unacceptable. Hence the rule that non-validating parsers need not fetch external entities if they don't want to.

Back-link to spec

Entity Reference Syntax

You can't use character or general entities in the DTD, you have to use parameter entities. This is a consequence of SGML's basic design choice that DTDs use a different syntax from that of documents. It is fairly likely that future-generation schema replacements will use XML, not DTD, syntax; thus there will be no necessity for parameter entities.

Back-link to spec

Error In Example In Section 4.4.5

This example, inserted one of the very last editing operations performed on the XML 1.0 specification, is wrong. In fact, the statement is correct and the text in the example is well-formed. However, the real point is that it would be well-formed if the second line read as follows, with a % instead of a &:

```
<!ENTITY WhatHeSaid "He said %YN;" >
```

Back-link to spec

Notify, But That's All

The processor has to inform the application. But, the spec doesn't say anything about what (if anything) the application has to do about this.

Back-link to spec

Bypassing

The practical effect of this is that entities don't have to be declared before being used, and the following is well-formed:

```
<!ENTITY AC "The &W3C; Advisory Council">
<!ENTITY W3C "WWW Consortium">
```

Back-link to spec

Included As PE

SGML expresses this same constraint using a bunch of grammar engineering and some formal constraints. Which, at a formal level, is perhaps more elegant; but the results are awfully complex and hard to understand. XML's trick of having the processor insert extra spaces seems a bit ad-hoc, but it's at least easy to understand.

Back-link to spec

Don't Use Parameter Entities In Other Entity Values

Eve Maler, one of the best DTD experts I know, told me that using a parameter entity in this way, to help build the value of another internal entity, is a really bad idea. So much so that she wanted to suppress this example in the spec. I can't actually imagine why you'd do this, but the rule did need illustration, so she lost that argument. But I'd take her advice seriously.

Back-link to spec

Names Are Better Than Character References

Yes, you can encode a < as <, but in general, < is much better, just because names are better than numbers. That's why the named entity declarations in the example below are actually quite typical constructs; it's really pretty unfriendly to users to have things like Շ floating around in documents.

Back-link to spec

Use Of Notations

This paragraph is really too specific. A notation is a name and an external identifier; the idea is that it's supposed to be helpful in figuring out how to deal with the data to which it's attached. The suggestion that the XML processor itself might do something with the notation is really bogus; that's not its job. So if anyone is interested, it would have to be the application, not the XML processor. Furthermore, the application might be able to handle the data itself without any help, once it knows the notation.

Back-link to spec

The XML Processor Should Not Deal With Unparsed Entities

I know it's a part of the spec, but I think this sentence is completely bogus. The job of the XML processor is to process XML, not to deal with attachments or be a general multimedia dispatcher.

Back-link to spec

Examples Of Document Entities

If the XML document is in a file, the document entity is that file. These annotations started life in a file called `notes.xml`; that was the document entity. If you're fetching an XML document via a URL, then the stream of bytes that you get by calling a function like `java.net.URL.openStream()` is the document entity.

Back-link to spec

Validity Is Not An Option

XML evangelists, such as myself, take great glee in pointing out that XML, unlike SGML, has no optional features; the result, we claim triumphantly, is that any XML processor in the world should be able to read any XML document in the world (well, modulo character encoding issues).

"Aha!" claim some ungrateful doubting Thomases; "XML distinguishes well-formedness and validity, and *that's an option!*"

Wrong. Anything that's well-formed is an XML document, and any XML processor has to be able to read any well-formed document. If a document wants to aspire to the higher karmic plane of validity, well good on it, but that's an extra, not an optional feature of XML.

Back-link to spec

Both Validating And Non-Validating

Some members of the first generation of XML processors actually are both validating and non-validating. Usually, they have some way to turn the validating behavior on and off. Nothing in the spec rules this out, and it seems to be useful.

In a similar vein, note that while a non-validating processor doesn't have to read anything but the document entity, nothing forbids this, and in fact, it's pretty easy to do it, from the programming point of view. So many of those first-generation processors will read external entities, even if not validating; the good ones also include a switch so you can turn this behavior on and off; sometimes you absolutely don't want to read external entities, for performance reasons.

Back-link to spec

What An XML Processor Must Do

This little definition is awfully important to XML. What it means that if anything claims to accept XML, then you can send it documents that have an internal subset, and if that internal subset uses handy tricks like attribute defaults and internal entities, then you can be confident that they'll be processed correctly.

Note, however, that you *can't* safely use external entities unless the use of validating processors is advertised.

The reason this paragraph is here is that during the development of XML, it became apparent that some people, given the chance, would claim XML support for anything that processed text that contained something that looked like a tag. The result of this would be that there would be legal XML documents that wouldn't be processed correctly by these people; while we can't make everyone do the right thing, we can (we hope) keep these sort of half-baked efforts from being awarded an "XML" label.

Back-link to spec

Effect Of Bypassing External Parameter Entities

Suppose I have the following short document:

```
<!DOCTYPE doc [
 <!ATTLIST doc a CDATA "D4A">
 <!ENTITY % x SYSTEM "x.ent">
 %x;
 <!ATTLIST doc b CDATA "D4B">
 ]>
<doc>Hello.</doc>
```

If a non-validating processor reads this, and chooses not to read the external entity `x` (which is just fine), then when it reads the document, it must tell the application that it saw something like:

```
<doc a="D4A">
```

That is, it is required to use the default attribute information for the attribute `a`, but *forbidden* to do so for `b`.

This makes perfect sense, if you consider that `x.ent` might have contained the following:

```
<!ATTLIST doc b CDATA "Tantric">
```

In this case, giving the attribute `b` the default value `D4B` would simply be wrong.

Back-link to spec

Predictable Behavior In XML Processors

This whole section is evidence of near-fanatical concern on the part of some members of the Working Group for exactness and predictability. In my opinion, since the spec makes it clear that non-validating parsers don't have to read external entities, anybody who's building an app that doesn't require a validating parser just won't use them, if they have a brain in their head. Thus, in operational terms, I expect no nasty surprises as a result of the fact that a non-validating parser might or might not read an external entity.

But some among us feel that optional behavior is dangerous and highly undesirable; therefore, this section explains in agonizing detail exactly what consequences optional entity reading can have, and gives some motherhood advice on how to avoid it.

Back-link to spec

RFC 1766 URL

ftp://ds.internic.net/rfc/rfc1766.txt

Back-link to spec

ISO 639

Technical contents of ISO 639:1988 (E/F) "Code for the representation of names of languages".

Typed by Keld.Simonsen@dkuug.dk 1990-11-30 Minor corrections, 1992-09-08 by Keld Simonsen Sundanese corrected, 1992-11-11 by Keld Simonsen

Two-letter lower-case symbols are used.

The Registration Authority for ISO 639 is Infoterm, Osterreiches Normungsinstitut (ON), Postfach 130, A-1021 Vienna, Austria.

```
aa Afar
ab Abkhazian
af Afrikaans
am Amharic
ar Arabic
as Assamese
ay Aymara
az Azerbaijani

ba Bashkir
be Byelorussian
bg Bulgarian
bh Bihari
bi Bislama
bn Bengali; Bangla
bo Tibetan
br Breton

ca Catalan
co Corsican
cs Czech
cy Welsh

da Danish
de German
dz Bhutani

el Greek
en English
eo Esperanto
es Spanish
et Estonian
eu Basque

fa Persian
fi Finnish
fj Fiji
```

fo Faeroese
fr French
fy Frisian

ga Irish
gd Scots Gaelic
gl Galician
gn Guarani
gu Gujarati

ha Hausa
hi Hindi
hr Croatian
hu Hungarian
hy Armenian

ia Interlingua
ie Interlingue
ik Inupiak
in Indonesian
is Icelandic
it Italian
iw Hebrew

ja Japanese
ji Yiddish
jw Javanese

ka Georgian
kk Kazakh
kl Greenlandic
km Cambodian
kn Kannada
ko Korean
ks Kashmiri
ku Kurdish
ky Kirghiz

la Latin
ln Lingala
lo Laothian
lt Lithuanian
lv Latvian, Lettish

mg Malagasy
mi Maori
mk Macedonian
ml Malayalam
mn Mongolian

mo Moldavian
mr Marathi
ms Malay
mt Maltese
my Burmese

na Nauru
ne Nepali
nl Dutch
no Norwegian

oc Occitan
om (Afan) Oromo
or Oriya

pa Punjabi
pl Polish
ps Pashto, Pushto
pt Portuguese

qu Quechua

rm Rhaeto-Romance
rn Kirundi
ro Romanian
ru Russian
rw Kinyarwanda

sa Sanskrit
sd Sindhi
sg Sangro
sh Serbo-Croatian
si Singhalese
sk Slovak
sl Slovenian
sm Samoan
sn Shona
so Somali
sq Albanian
sr Serbian
ss Siswati
st Sesotho
su Sundanese
sv Swedish
sw Swahili

ta Tamil
te Tegulu
tg Tajik

```
th Thai
ti Tigrinya
tk Turkmen
tl Tagalog
tn Setswana
to Tonga
tr Turkish
ts Tsonga
tt Tatar
tw Twi

uk Ukrainian
ur Urdu
uz Uzbek

vi Vietnamese
vo Volapuk

wo Wolof

xh Xhosa

yo Yoruba

zh Chinese
zu Zulu
```

Back-link to spec

ISO 3166 URL

This is an ISO standard which, unusually, has to change all the time. So if you want to be sure, you'll have to go to ISO and spend some real money.

However, you can generally look these up on the Web by going to any search engine and searching for the phrase "ISO 3166".

I usually look in ftp://ftp.ripe.net/iso3166-countrycodes.

Back-link to spec

Unicode URL

I've never actually *seen* a copy of ISO 10646; one reason is that the Unicode version, which is identical, comes in a really beautiful book. The easy way to get it is to go to http://www.unicode.org and order it on-line; I did, and it showed up very quickly.

Back-link to spec

The Dragon Book

This book, versions of which have been in print for many years, is known as "the dragon book" because of the picture on the cover. It must be one of the most widely-used computer science reference works ever written; in particular, anyone who's ever taken a compiler course or written a serious parser very likely has a copy on their shelves.

Back-link to spec

URI URL

This one is still a major moving target; I asked Dan Connolly what the best on-line reference was, and he said:

Here's the path:
http://www.w3.org/
http://www.w3.org/Architecture/
http://www.w3.org/Architecture/#Research
http://www.w3.org/Addressing/
http://www.w3.org/Addressing/#hotlist
http://www.ics.uci.edu/pub/ietf/uri/
http://www.ics.uci.edu/pub/ietf/uri/draft-fielding-uri-syntax-02.txt

Back-link to spec

Regular Languages Into Finite Automata

ftp://ftp.informatik.uni-freiburg.de/documents/reports/report33/

Back-link to spec

Deterministic Regular Languages

ftp://ftp.informatik.uni-freiburg.de/documents/reports/report38/

Back-link to spec

RFC 1738 URL

ftp://ds.internic.net/rfc/rfc1738.txt

Back-link to spec

RFC 1818 URL

ftp://ds.internic.net/rfc/rfc1808.txt

Back-link to spec

RFC 2141 URL

ftp://ds.internic.net/rfc/rfc2141.txt

Back-link to spec

HyTime URL

http://www.ornl.gov/sgml/wg8/docs/n1920/

Back-link to spec

XML's Notion Of A Letter

The Working Group (in particular James Clark) labored mightily over these definitions, and I think that we hit the mark very well. Unicode itself contains a set of properties that are supposed to define what a "letter" is, and Java has used these to make rules as to what can be an "identifier". I think that the XML rules provide a much more reasonable and less surprising notion of what it's reasonable to put in a name.

Back-link to spec

ASCII Latin

Basic ASCII Latin

Back-link to spec

ISO Latin

What used to be the top half of ISO 8859 Latin-1; European accented characters.

Back-link to spec

Accented Latin

Lots of different accented and otherwise marked Latin characters, for handling everything from Welsh to Serbo-Croat.

Back-link to spec

IPA

IPA is the International Phonetic Alphabet, much used in dictionaries.

Back-link to spec

Spacing Modifiers

Reversed commas and such-like.

Back-link to spec

Greek

Greek, based on ISO-8859-7.

Back-link to spec

Cyrillic

Cyrillic (for Russian and related languages), based on ISO-8859-5 and other sources.

Back-link to spec

Armenian

Back-link to spec

Hebrew

Hebrew, based on ISO 8859-8 and other sources.

Back-link to spec

Arabic

Arabic, based on ISO 8859-6 and other sources.

Back-link to spec

Devanagari

Devanagari, based on ISCII 1988.

Back-link to spec

Bengali

Bengali, based on ISCII 1988.

Back-link to spec

Gurmukhi

Gurmukhi, based on ISCII 1988.

Back-link to spec

Gujarati

Gujarati, based on ISCII 1988.

Back-link to spec

Oriya

Oriya, based on ISCII 1988.

Back-link to spec

Tamil

Tamil, based on ISCII 1988.

Back-link to spec

Telugu

Telugu, based on ISCII 1988.

Back-link to spec

Kannada

Kannada, based on ISCII 1988.

Back-link to spec

Malayalam

Malayalam, based on ISCII 1988.

Back-link to spec

Thai

Thai, based on TIS 620-2529.

Back-link to spec

Lao

Lao, based on TIS 620-2529.

Back-link to spec

Tibetan

Back-link to spec

Georgian

Georgian, both archaic and modern.

Back-link to spec

Hangul Jamo

The Korean combining alphabet.

Back-link to spec

Latin Extended Additional

Exotic letter-diacritic combinations, plus a block used to support Vietnamese.

Back-link to spec

Greek Extended

Mostly used for polytonic Greek texts.

Back-link to spec

Letterlike Symbols

Ohms, Kelvins, etc.

Back-link to spec

Number Forms

Three exotic Roman-Numeral combinations.

Back-link to spec

Hiragana

The Japanese Hiragana syllabics, based on JIS X 0208.

Back-link to spec

Katakana

The Japanese Katakana syllabics, based on JIS X 0208.

Back-link to spec

Bopomofo

Bopomofo, for teaching the phonetics of Mandarin, based on GB 2312.

Back-link to spec

Hangul Syllables

The standard Korean Hangul syllables, based on KS C 5601-1992.

Back-link to spec

Unified CJK Ideographs

The set of unified "Han" ideographs used in writing Chinese, Japanese, and Korean.

Back-link to spec

CJK Symbols and Punctuation

Used with CJK ideographs; these are ideographic numerals.

Back-link to spec

Combining Diacritics

Accents and diacritics by themselves, in case you want to combine them with a character that does come with a precooked diacritic version.

Back-link to spec

Cyrillic Combining Diacritics

Back-link to spec

Hebrew Combining Diacritics

Back-link to spec

Arabic Combining Diacritics

Back-link to spec

Devanagari Combining Diacritics

Back-link to spec

Bengali Combining Diacritics

Back-link to spec

Gurmukhi Combining Diacritics

Back-link to spec

Gujarati Combining Diacritics

Back-link to spec

Oriya Combining Diacritics

Back-link to spec

Tamil Combining Diacritics

Back-link to spec

Telugu Combining Diacritics

Back-link to spec

Kannada Combining Diacritics

Back-link to spec

Malayalam Combining Diacritics

Back-link to spec

Thai Combining Diacritics

Back-link to spec

Lao Combining Diacritics

Back-link to spec

Tibetan Combining Diacritics

Back-link to spec

Math/Technical Combining Diacritics

Back-link to spec

Ideographic Diacritics

Back-link to spec

Hiragana/Katakana Combining Diacritics

Back-link to spec

Basic ASCII digits 0-9

Back-link to spec

Arabic Digits 0-9

Back-link to spec

Eastern Arabic-Indic Digits 0-9

Used in Persian and Urdu.

Back-link to spec

Devanagari Digits 0-9

Back-link to spec

Bengali Digits 0-9

Back-link to spec

Gurmukhi Digits 0-9

Back-link to spec

Gujarati Digits 0-9

Back-link to spec

Oriya Digits 0-9

Back-link to spec

Tamil Digits 0-9

Back-link to spec

Telugu Digits 0-9

Back-link to spec

Kannada Digits 0-9

Back-link to spec

Malayalam Digits 0-9

Back-link to spec

Thai Digits 0-9

Back-link to spec

Lao Digits 0-9

Back-link to spec

Tibetan Digits 0-9

Back-link to spec

Middle Dot

Back-link to spec

Triangular Colon and Half Colon

Back-link to spec

Greek Ano Teleia

Back-link to spec

Arabic Tatweel

Back-link to spec

Thai Maiyamok

Back-link to spec

Lao Ko La

Back-link to spec

Ideographic Iteration Mark

Back-link to spec

Japanese Kana Repetition Marks

Back-link to spec

Japanese Hiragana Iteration Marks

Back-link to spec

Japanese Kana Iteration Marks

Back-link to spec

XML Rules For Character Classification

Although the Working Group emphatically *did* argue over the inclusion and exclusion of individual characters, we (well, mostly James Clark) were able to work out a set of rules whereby you can extract the XML lists from those in the standard automatically.

Back-link to spec

Character Categories

The Unicode standard is accompanied by a set of tables, available from http://www.unicode.org or on a CD-ROM that you get with the book. These tables, in their electronic form, assign a bunch of categories to each character; that's where these names come from.

Back-link to spec

It's Not Necessary To Read the ISO SGML Standard

ISO 8879, the SGML standard, is designed for maximum flexibility and generality, not for readability. One of the main benefits of XML is that you can write software without having to read and understand this rather intimidating document.

Back-link to spec

The Ugliness Of Entity Expansion

Fortunately, you don't have to worry about it unless you're going to be doing some heavy DTD engineering. And this is far from being the most challenging aspect of that job; so if you *are* going to, I recommend taking a course, or at the very least reading a book on the subject, before you start.

Back-link to spec

Encodings: Details for Programmers

If you're not a computer programmer, you can skip the next few paragraphs, down to the one that begins "Like any self-labeling system..." For non-programmers, the message is that even if normal character encoding signalling mechanisms break down, in XML you have an excellent chance of figuring out what's going on and thus interchanging documents, using the techniques described (elegantly, by my co-editor Michael Sperberg-McQueen) in the following section.

If you are a computer programmer, these techniques are actually pretty easy to implement. The really disgusting code is when you have to pick apart UTF-8; but that's documented fairly well in the Unicode standard, and the CD-ROM comes with example code in C.

Back-link to spec

External vs. Internal Encoding Evidence

There was a lot of passionate debate around this point. Many people feel very strongly that if you're in the Web environment and there is an HTTP header, that's the only thing that should be used to determine character encoding; that the BOMs and encoding declarations and so on are just extras in those cases. They felt this so strongly that they wanted to write that rule right into the main body of the spec.

The majority felt, though, that XML was probably going to be used regularly in non-Web applications, so there was really no place for this kind of language in the main body of the spec. This text in an appendix represents the compromise. Lesson: the best experts in this area really think you ought to use external headers, when you have them. They're probably right.

Back-link to spec

James Clark

James Clark is an Englishman who has been lucky in life to the extent that he can turn his formidable talents to whatever most interests him. Fortunately for the worlds of publishing technology, what mosts interest him is software, typesetting, and structured documents. He is the author of groff, SP, and Jade; SP has come, in a practical way, to serve as the real-world definition of what SGML is and is not. He has already, in XML's short life, made some major technical contributions and, as usual, made them freely available for everyone.

His position in the XML activity as "technical lead" meant, in practice, that whenever he said something, the rest of the group took it very seriously. His contributions to the design of XML included its name, the empty-element tag syntax, and many other crucial aspects.

Back-link to spec

Paula Angerstein

Paula was a late arrival to the Working Group, and thus had very little direct input to this version of the specification. However, her name is a good one to have on the spec; she has served in leadership positions for many years in the publishing industry (now at Texcel), and put in considerable time on the ISO standards committees.

Back-link to spec

Steve DeRose

Steve DeRose's name is one to conjure with in both the academic hypertext and commercial electronic publishing fields. He authored and co-authored many seminal papers, including one in the CACM in 1987 that was for many people (including me) the real in-depth introduction to the advantages of descriptive markup in general and SGML in particular.

Steve was in at the birth of Electronic Book Technologies (now Inso), the company that with the "Dyna" products more or less defined the SGML-electronic-publishing industry.

Back-link to spec

Dave Hollander

Dave put in quite a few years in Bell Labs, then quite a few more at Hewlett-Packard. At HP, he presided first over a heroic effort to put SGML to work in an industrial publishing environment, then over one of the first and still largest-scale corporate web-site and intranet roll-outs.

In the course of the XML discussions, Dave often served as the representative of the industrial information provider, arguing at high volume and with great persuasiveness for design choices that would maximize, above all else, interoperability in XML documents.

Back-link to spec

Eliot Kimber

Eliot came to the notice of the SGML world when, as an employee of IBM, he (in co-operation with Wayne Wohler, who deserves some mention in these notes simply for being the first person that I know of to have proposed a minimal SGML) engineered the IBM IDDOC document architecture, a system for automating technical pubs that probably remains unmatched in its ambition and comprehensiveness.

Eliot has moved on to employment with smaller companies than IBM, and to editorship of the ISO HyTime (10744) standard. In the XML discussions, his voice was often raised to point out with brutal clarity what we would lose by discarding this feature or that from SGML. Despite his enthusiastic championing of the utility of the most arcane features of SGML, he has been a strong supporter of the basic XML idea both in theory and in practice.

Back-link to spec

Eve Maler

Eve Maler, a long time DEC veteran before taking up her current work at Arbortext, is arguably the world's foremost expert on the construction of SGML DTDs.

While there is probably room for argument on the subject of world DTD supremacy, I can attest personally to Eve's frightening precision and thoroughness as a copy editor; a huge number of glaring boo-boos and subtly-shaded misstatements formerly in the XML spec owe their deaths to her tireless combing of draft after draft.

Back-link to spec

Tom Magliery

Tom works in the team at NCSA that originally gave the world Mosaic, the graphical Web browser that turned the Internet from geek-pasture to investment-banker territory. Pressure of other commitments forced him to leave the WG about halfway through the work of building XML 1.0.

Back-link to spec

Murray Maloney

Murray was a founding member of the WG and, due to industry churn, holds the record for representing the largest number of different organizations: SoftQuad, Grif, Muzmo, CNGroup, and Veo.

Murray's involvement with XML extends well back before its birth; he was involved in the W3C in its very early days, and was a leader among those lobbying the W3C to bless some sort of SGML-on-the-Web effort. He gets quite a bit of credit for XML's making a prominent splash at the WWW6 conference in April 1997.

The history is even longer than that; Murray was heavily involved in the Davenport group, a clear ancestor of the XML WG in that it included Eve Maler, Dave Hollander, and Jon Bosak as chair.

Finally, although Murray would probably demur, his long and close relationship with Yuri Rubinsky makes me think of him as as representing Yuri's voice in the process.

Back-link to spec

Murata Makoto

The name is in the correct order above, incorrect in the spec; Japanese names should be written surname-first. While Makoto arrived in the WG relatively late in its working life, he had very substantial input into the spec, and was a diligent and effective reviewer. Obviously, he had special expertise to offer in the area of internationalization, but his contributions extended well beyond this area; he was particularly concerned with achieving extreme rigor in the specification and minimizing the chances for nondeterministic behavior.

Back-link to spec

Joel Nava

Joel (of Adobe), like many others, joined the WG rather late in its work process. His voice in the process was raised consistently in favor of greater and still greater simplicity; he consistently argued for the omission of features and the reduction of complexity; while he did not win all those arguments, the spec is better for them having been made.

Back-link to spec

Conleth O'Connell

Con O'Connell goes *way* back in the history of SGML. He joined the XML WG too late to have been a major force in the development of XML 1.0, but his name, and Vignette's, are good ones to have associated with XML.

Back-link to spec

Peter Sharpe

Peter started working on SGML software while the standard was still being built. He is the primary architect of Author/Editor, one of the earliest and perhaps the largest-selling of all the SGML editors. In recent years, he has had an even greater impact on the world as the primary architect of HoTMetaL, one of the earliest HTML editors, and still a popular product.

In the XML process, Peter spoke for those who had to write the software that authors would use. In particular, he was a force pushing for many of the simplifications in the rules governing XML DTDs.

Back-link to spec

John Tigue

John is a Microsoft veteran who went to work for a small company located near Seattle named DataChannel. DataChannel was perhaps the first private sector player to leap on the XML idea and define itself as an XML vendor; John himself implemented one of the early XML parsers.

Back-link to spec

Extensible Markup Language (XML) 1.0

Dies ist die deutsche Übersetzung der »XML 1.0 Recommendation« vom 10. Februar 1998 des W3C. Bitte beachten Sie, daß dieses Dokument Übersetzungsfehler enthalten kann. Die normative, englische Version steht im Web unter http://www.w3.org/TR/1998/REC-xml-19980210 zur Verfügung. Diese deutsche Übersetzung ist unter http://www.mintert.com/xml/trans/REC-xml-19980210-de.html zu finden. Weitere Teile der XML-Übersetzung sind (jetzt oder in Zukunft) unter http://www.mintert.com/xml/trans/ zu finden.

Übersetzer

- Henning Behme (iX) <hb@ix.heise.de>
- Stefan Mintert (Universität Dortmund) <stefan@mintert.com>

Wir haben uns bemüht, die Übersetzung mit höchster Sorgfalt durchzuführen. Einige englische Begriffe haben wir jedoch nicht übersetzt, wenn dadurch ein schlechteres Verständnis zu erwarten wäre. Auf einzelne problematische Stellen haben wir im Text hingewiesen.

Die Übersetzer danken allen Personen, die sich mit Hinweisen, Korrekturen oder Anregungen an dieser Übersetzung beteiligt haben[1].

REC-xml-19980210

Extensible Markup Language (XML) 1.0

Empfehlung des W3C, 10. Februar 1998

Diese Version

- http://www.w3.org/TR/1998/REC-xml-19980210
- http://www.w3.org/TR/1998/REC-xml-19980210.xml
- http://www.w3.org/TR/1998/REC-xml-19980210.html
- http://www.w3.org/TR/1998/REC-xml-19980210.pdf
- http://www.w3.org/TR/1998/REC-xml-19980210.ps

Aktuelle Version

http://www.w3.org/TR/1998/REC-xml

Vorherige Version

http://www.w3.org/TR/PR-xml-971208

Herausgeber

- Tim Bray (Textuality und Netscape) <tbray@textuality.com>

- Jean Paoli (Microsoft) <jeanpa@microsoft.com>
- C. M. Sperberg-McQueen (Universität Chicago) <cmsmcq@uic.edu>

Zusammenfassung

Die Extensible Markup Language (XML) ist eine Teilmenge von SGML, die vollständig in diesem Dokument beschrieben ist. Das Ziel ist es, zu ermöglichen, generic SGML in der Weise über das Web auszuliefern, zu empfangen und zu verarbeiten, wie es jetzt mit HTML möglich ist. XML wurde entworfen, um eine einfache Implementierung und Zusammenarbeit sowohl mit SGML als auch mit HTML zu gewährleisten.

Status dieses Dokuments

Dieses Dokument wurde von Mitgliedern des W3C und anderen Interessierten geprüft und vom Direktor als W3C-Empfehlung gebilligt. Es ist ein stabiles Dokument und darf als Referenzmaterial verwendet oder als normative Referenz von einem anderen Dokument zitiert werden. Die Rolle des W3C bei der Erstellung dieser Empfehlung ist es, die Spezifikation bekanntzumachen und ihre breite Anwendung zu fördern. Dies erhöht die Funktionsfähigkeit und Interoperabilität des Web.

Dieses Dokument definiert eine Syntax, die als Teilmenge eines bereits vorhandenen und weithin eingesetzten internationalen Standards (Standard Generalized Markup Language, ISO 8879:1986(E), in der berichtigten und korrigierten Fassung) für die Anwendung im World Wide Web geschaffen wurde. Es ist ein Ergebnis des XML-Engagements des W3C; Einzelheiten sind unter http://www.w3.org/XML/ zu finden. Eine Liste aktueller W3C-Empfehlungen und anderer technischer Dokumente ist unter http://www.w3.org/TR/ zu finden.

Diese Spezifikation verwendet den von [Berners-Lee et al.] definierten Begriff URI; die Arbeit daran ist noch im Gange und wird voraussichtlich [IETF RFC1738] und [IETF RFC1808] aktualisieren.

Die Liste der bekannten Fehler in dieser Spezifikation ist unter http://www.w3.org/XML/xml-19980210-errata verfügbar.

Bitte melden Sie Fehler in diesem Dokument an <xml-editor@w3.org>.

Inhaltsverzeichnis

1 Einleitung

Die Extensible Markup Language, abgekürzt XML, beschreibt eine Klasse von Datenobjekten, genannt XML-Dokumente, und beschreibt teilweise das Verhalten von Computer-Programmen, die solche Dokumente verarbeiten. XML ist ein Anwendungsprofil (application profile) oder eine eingeschränkte Form von SGML, der Standard Generalized Markup Language [ISO 8879]. Durch ihre Konstruktion sind XML-Dokumente konforme SGML-Dokumente.

XML-Dokumente sind aus Speicherungseinheiten aufgebaut, genannt Entities, die entweder analysierte (parsed) oder nicht analysierte (unparsed) Daten enthalten. Analysierte Daten bestehen aus Zeichen, von denen einige Zeichendaten und andere Markup darstellen. Markup ist eine Beschreibung der Aufteilung auf Speicherungseinheiten und der logischen Struktur des Dokuments. XML bietet einen Mechanismus an, um Beschränkungen der Aufteilung und logischen Struktur zu formulieren.

Ein Software-Modul, genannt **XML-Prozessor**, dient dazu, XML-Dokumente zu lesen und den Zugriff auf ihren Inhalt und ihre Struktur zu erlauben. Es wird angenommen, daß ein XML-Prozessor seine Arbeit als Teil eines anderen Moduls, genannt **Anwendung**, erledigt. Diese Spezifikation beschreibt das notwendige Verhalten eines XML-Prozessors soweit es die Frage betrifft, wie er XML-Daten einlesen muß und welche Informationen er an die Anwendung weiterreichen muß.

1.1 Herkunft und Ziele

XML wurde von einer XML-Arbeitsgruppe (ursprünglich bekannt als das **SGML Editorial Review Board**) entwickelt, die 1996 unter der Schirmherrschaft des World Wide Web Consortium (W3C) gegründet wurde. Den Vorsitz hatte Jon Bosak von Sun Microsystems inne, unter aktiver Beteiligung einer XML Special Interest Group (ehemals SGML-Arbeitsgruppe), die ebenfalls vom W3C organisiert wurde. Die Mitglieder der XML-Arbeitsgruppe sind in einem der Anhänge genannt. Dan Connolly fungierte als Kontaktperson zwischen der Arbeitsgruppe und dem W3C.

Die Entwurfsziele für XML sind:

1. XML soll sich im Internet auf einfache Weise nutzen lassen.
2. XML soll ein breites Spektrum von Anwendungen unterstützen.
3. XML soll zu SGML kompatibel sein.
4. Es soll einfach sein, Programme zu schreiben, die XML-Dokumente verarbeiten.
5. Die Zahl optionaler Merkmale in XML soll minimal sein, idealerweise Null.
6. XML-Dokumente sollten für Menschen lesbar und angemessen verständlich sein.
7. Der XML-Entwurf sollte zügig abgefaßt sein.
8. Der Entwurf von XML soll formal und präzise sein.
9. XML-Dokumente sollen leicht zu erstellen sein.
10. Knappheit von XML-Markup ist von minimaler Bedeutung.

Diese Spezifikation enthält, zusammen mit den verwandten Standards (Unicode und ISO/IEC 10646 für Zeichen, Internet-RFC 1766 für Codes zur Identifikation der Sprache, ISO 639 für Codes von Sprachnamen und ISO 3166 für Ländernamen-Codes), alle Informationen, die notwendig sind, um XML in der Version 1.0 zu verstehen und um Programme zu schreiben, die sie verarbeiten.

Diese Version der XML-Spezifikation darf frei weitergegeben werden, solange der gesamte Text und alle rechtlichen Hinweise unversehrt bleiben.

1.2 Terminologie

Die Terminologie, die zur Beschreibung von XML-Dokumenten verwendet wird, ist innerhalb dieser Spezifikation definiert. Die in der folgenden Liste definierten Begriffe werden benutzt, um jene Definitionen zu formulieren und die Aktionen eines XML-Prozessors zu beschreiben.

darf (may)

Konforme Dokumente und XML-Prozessoren dürfen sich so verhalten, wie beschrieben, müssen es aber nicht.

müssen/nicht dürfen (must)

Konforme Dokumente und XML-Prozessoren müssen sich/dürfen sich nicht so verhalten, wie beschrieben, andernfalls sind sie fehlerhaft.

Fehler (error)

Eine Verletzung der Regeln der Spezifikation; die Konsequenzen sind nicht definiert. Konforme Software darf Fehler erkennen und anzeigen und anschließend weiterarbeiten.

Kritischer Fehler (fatal error)

Ein Fehler, den ein konformer XML-Prozessor erkennen und an das Anwendungsprogramm melden muß. Nach der Erkennung eines kritischen Fehlers darf der Prozessor die Verarbeitung fortsetzen, um nach weiteren Fehlern zu suchen und diese dem Anwendungsprogramm zu melden. Um die Fehlerkorrektur zu unterstützen, darf der Prozessor nichtverarbeitete Daten des Dokuments (mit vermischtem Text und Markup) dem Anwendungsprogramm zur Verfügung stellen. Wenn ein kritischer Fehler aufgetreten ist, darf ein Prozessor die normale Verarbeitung nicht fortsetzen. Dies heißt insbesondere, daß er keine weiteren Daten oder Informationen über die logische Struktur des Dokuments an das Anwendungsprogramm weiterleiten darf, wie es normalerweise geschieht.

benutzeroptional (at user option)

Konforme Software darf oder muß (in Abhängigkeit von dem Modalverb im Satz) sich so verhalten wie beschrieben. Sie muß dem Benutzer eine Möglichkeit einräumen, das beschriebene Verhalten ein- oder auszuschalten.

Gültigkeitsbeschränkung (validity constraint)

Eine Regel, die für alle gültigen XML-Dokumente gilt. Verletzungen von Gültigkeitsbeschränkungen sind Fehler. Sie müssen benutzeroptional von validierenden XML-Prozessoren gemeldet werden.

Wohlgeformtheitsbeschränkung (well-formedness constraint)

Eine Regel, die für alle wohlgeformten XML-Dokumente gilt. Verletzungen von Wohlgeformtheitsbeschränkungen sind kritische Fehler.

passen (match)

- (Strings oder Namen:) Zwei Zeichenketten oder Namen, die verglichen werden, müssen identisch sein. Zeichen mit mehreren möglichen Repräsentationen in ISO/IEC 10646 (zum Beispiel Zeichen in einer geschlossenen Form und einer Form, die aus einem Grundzeichen und einem diakritischen Zeichen zusammengesetzt wird) passen nur, wenn sie die gleiche Repräsentation in beiden Zeichenketten haben. Benutzeroptional darf ein XML-Prozessor solche Zeichen in eine kanonische Form normalisieren. Es wird keine Umwandlung zwischen Groß- und Kleinschreibung durchgeführt.
- (Strings und Regeln in der Grammatik:) Ein String paßt zu einer grammatikalischen Produktion, wenn er zur Sprache gehört, die durch die Produktion erzeugt wird.
- (Inhalt und Inhaltsmodelle:) Ein Element paßt zu seiner Deklaration, wenn es in der Weise konform ist, die in der Gültigkeitsbeschränkung »gültiges Element« beschrieben ist.

zwecks Kompatibilität (for compatibility)

Eine Eigenschaft von XML, die einzig zu dem Zweck eingeführt wurde, daß XML mit SGML kompatibel bleibt.

zwecks Zusammenarbeit (for interoperability)

Eine unverbindliche Empfehlung, die dazu dient, die Wahrscheinlichkeit zu erhöhen, daß XML-Dokumente von existierenden SGML-Prozessoren verarbeitet werden können, die älter sind als der »WebSGML Adaptions Annex«.

2 Dokumente

Ein Datenobjekt ist ein **XML-Dokument**, wenn es im Sinne dieser Spezifikation wohlgeformt ist. Ein wohlgeformtes XML-Dokument kann darüber hinaus gültig sein, sofern es bestimmten weiteren Einschränkungen genügt.

Jedes XML-Dokument hat sowohl eine »logische« als auch eine »physikalische Struktur«. Physikalisch besteht das Dokument aus einer Reihe von Einheiten, genannt Entities. Ein Entity kann auf andere Entities verweisen, um sie in das Dokument einzubinden. Jedes Dokument beginnt in einer »Wurzel-« oder »Dokument-Entity«. Aus logischer Sicht besteht das Dokument aus Deklarationen, Elementen, Kommentaren, Zeichenreferenzen und Processing Instructions, die

innerhalb des Dokuments durch *explizites Markup* ausgezeichnet sind. Logische und physikalische Strukturen müssen korrekt verschachtelt sein, wie in 4.3.2 beschrieben.

2.1 Wohlgeformte XML-Dokumente

Ein textuelles Objekt ist ein **wohlgeformtes** XML-Dokument, wenn

1. es als Gesamtheit betrachtet zu der mit document bezeichneten Produktion paßt,
2. es alle Wohlgeformtheitsbeschränkungen dieser Spezifikation erfüllt und
3. jedes seiner parsed Entities, welches direkt oder indirekt referenziert wird, wohlgeformt ist.

Dokument
[1] document ::= prolog element Misc*

Daß ein Dokument zur document-Produktion paßt, impliziert:

1. Es enthält ein oder mehrere Elemente.
2. Es existiert genau ein Element, genannt **Wurzel** oder **Dokument-Element**, von dem kein Teil im Inhalt eines anderen Elements enthalten ist. Für alle anderen Elemente gilt: Wenn das Start-Tag im Inhalt eines anderen Elements ist, dann auch das End-Tag. Einfacher ausgedrückt: Die Elemente, begrenzt durch Start- und End-Tag, sind korrekt *ineinander verschachtelt*.

Als eine Konsequenz daraus gilt, daß für jedes Element k, das nicht die Wurzel ist, ein anderes Element v existiert, so daß sich k im Inhalt von v befindet, aber nicht im Inhalt eines anderen Elements, das sich im Inhalt von v befindet. V heißt **Vater**[2] von k und k heißt **Kind** von v.

2.2 Zeichen

Ein analysiertes Entity enthält **Text**, eine Folge von Zeichen, die entweder Markup oder Zeichendaten darstellen. Ein **Zeichen** (**character**) ist eine atomare Einheit von Text gemäß der Spezifikation in ISO/IEC 10646. Gültige Zeichen sind Tab (Tabulator), Carriage Return (Wagenrücklauf), Line Feed (Zeilenvorschub) sowie die Grafikzeichen von Unicode und ISO/IEC 10646. Von der Verwendung von »Kompatibilitätszeichen« (compatibility characters), wie sie in Abschnitt 6.8 von [Unicode] definiert werden, wird abgeraten.

Zeichenbereich	
[2] Char ::= #x9 | #xA | #xD | [#x20-#D7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]	/* jedes Unicode-Zeichen, ausgenommen die Ersatzblöcke FFFE und FFFF. */

Der Mechanismus zur Kodierung von Zeichen in Bitmustern darf von Entity zu Entity variieren. Alle XML-Prozessoren müssen die Kodierungen UTF-8 und UTF-16 aus ISO/IEC 10646 akzeptieren. Die Möglichkeiten zur Deklaration, welche der beiden Kodierungen verwendet wird, sowie die Verwendung von anderen Kodierungen werden später, in 4.3.3, behandelt.

2.3 Allgemeine syntaktische Konstrukte

Dieser Abschnitt definiert einige Symbole, die innerhalb der Grammatik häufig Verwendung finden.

S (Leerraum, White Space) besteht aus einem oder mehreren Leerzeichen (#x20), Wagenrückläufen, Zeilenvorschüben oder Tabulatoren.

Leerraum
[3] S ::= (#x20 | #x9 | #xD | #xA)+

Zeichen sind aus Gründen der Bequemlichkeit als Buchstaben, Ziffern und andere Zeichen klassifiziert. Buchstaben bestehen aus einem führenden alphabetischen oder Silbenzeichen, möglicherweise gefolgt von einem oder mehreren kombinierenden Zeichen oder von einem ideographischen Zeichen. Die vollständige Definition der einzelnen Zeichen in

jeder Klasse ist in 8 aufgeführt.

Ein **Name** ist ein Token[3], das mit einem Buchstaben (letter) oder einem erlaubten Interpunktionszeichen beginnt, woran sich Buchstaben, Ziffern (digit), Bindestriche, Unterstriche, Doppelpunkte oder Punkte anschließen. Letztere sind als Name-Zeichen bekannt. Namen, die mit »xml« oder mit einer Zeichenkette beginnen, die zu (('X'|'x') ('M'|'m') ('L'|'l')) paßt, sind für die Standardisierung in dieser oder einer zukünftigen Version dieser Spezifikation reserviert.

Hinweis: Der Doppelpunkt ist innerhalb von XML-Namen für Experimente mit Namensräumen reserviert. Es ist zu erwarten, daß seine Bedeutung irgendwann in der Zukunft standardisiert wird. Es könnte dann notwendig sein, Dokumente, die mit dem Doppelpunkt experimentieren, zu aktualisieren. (Es gibt keine Garantie, daß ein Mechanismus für Namensräume in XML tatsächlich den Doppelpunkt als Trennzeichen für Namensräume verwendet.) Praktisch bedeutet das, daß Autoren den Doppelpunkt in XML-Namen außer zu Namensraum-Versuchen nicht einsetzen sollten, daß aber XML-Prozessoren den Doppelpunkt als Name-Zeichen akzeptieren sollten.

Ein **Nmtoken** (Name Token) ist eine beliebige Kombination von Name-Zeichen.

Namen und Token
[4] NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender
[5] Name ::= (Letter | '_' | ':') (NameChar)*
[6] Names ::= Name (S Name)*
[7] Nmtoken ::= (NameChar)+
[8] Nmtokens ::= Nmtoken (S Nmtoken)*

Ein Literal ist eine beliebige, in Anführungszeichen[4] eingeschlossene Zeichenkette, die nicht das Anführungszeichen enthält, das zur Begrenzung der Zeichenkette verwendet wird. Literale werden verwendet, um den Inhalt von internen Entities (EntityValue), die Werte von Attributen (AttValue) und um externe Bezeichner (SystemLiteral) anzugeben. Beachten Sie, daß ein SystemLiteral analysiert (parsed) werden kann, ohne nach Markup zu suchen.

Literale
[9] EntityValue ::= '"' ([^%&"] | PEReference | Reference)* '"' | "'" ([^%&'] | PEReference | Reference)* "'"
[10] AttValue ::= '"' ([^<&"] | Reference)* '"' | "'" ([^<&'] | Reference)* "'"
[11] SystemLiteral ::= ('"' [^"]* '"') | ("'" [^']* "'")
[12] PubidLiteral ::= '"' PubidChar* '"' | "'" (PubidChar - "'")* "'"
[13] PubidChar ::= #x20 | #xD | #xA | [a-zA-Z0-9] | [-'()+,./:=?;!*#@$_%]

2.4 Zeichendaten und Markup

Text besteht aus miteinander vermengten Zeichendaten und Markup. **Markup** besteht aus Start-Tags, End-Tags, Tags für leere Elemente, Entity-Referenzen, Zeichenreferenzen, Kommentaren, Begrenzungen für CDATA-Abschnitte, Dokumenttyp-Deklarationen und Processing Instructions.

Sämtlicher Text, der kein Markup ist, bildet die **Zeichendaten** des Dokuments.

Das et-Zeichen (&) und die öffnende spitze Klammer (<) dürfen in ihrer literalen Form *ausschließlich* als Markup-Begrenzungen, innerhalb eines Kommentars, einer Processing Instruction oder eines CDATA-Abschnitts benutzt werden. Sie sind außerdem innerhalb des literalen Werts einer internen Entity-Deklaration zulässig, siehe 4.3.2. Falls sie an anderer Stelle benötigt werden, müssen sie geschützt (escaped) werden. Dies kann durch eine numerische Zeichenreferenz oder die Zeichenketten & (Ampersand, et-Zeichen) bzw. < (kleiner-als, less-than) geschehen. Die schließende spitze Klammer (>) kann durch die Zeichenkette > (größer-als, greater-than) dargestellt werden. Sie muß zwecks Kompatibilität durch > oder eine Zeichenreferenz geschützt werden, falls sie in der Zeichenkette]]> an einer Stelle auftritt, an der diese Zeichenkette nicht das Ende eines CDATA-Abschnitts markiert.

Innerhalb des Inhalts eines Elements ist jede Folge von Zeichen, die keine Anfangsbegrenzung von irgendeiner Form von Markup enthalten, Teil der Zeichendaten. Innerhalb eines CDATA-Abschnitts ist jede Folge von Zeichen, die nicht den CDATA-Abschluß]]> enthält, Teil der Zeichendaten.

Um Attributwerten zu erlauben, sowohl das einfache als auch das doppelte Anführungszeichen zu enthalten, kann das

Apostroph (') als ' und das doppelte Anführungszeichen (") als " dargestellt werden.

Zeichendaten
[14] CharData ::= [^<&]* - ([^<&]* ']]>' [^<&]*)

2.5 Kommentare

Kommentare dürfen innerhalb des Dokuments an beliebiger Stelle außerhalb des übrigen Markup stehen. Darüber hinaus dürfen sie innerhalb der Dokumenttyp-Deklaration an den von der Grammatik erlaubten Stellen stehen. Sie sind kein Bestandteil der Zeichendaten eines Dokuments. Ein XML-Prozessor kann, muß aber nicht, der Anwendung eine Möglichkeit einräumen, den Text eines Kommentars zu lesen. Zwecks Kompatibilität darf die Zeichenkette »--« (zwei Trennstriche) nicht innerhalb eines Kommentars erscheinen.

Kommentare
[15] Comment ::= <!--' ((Char - '-') | ('-' (Char - '-')))* '-->'

Ein Beispiel für einen Kommentar

```
<!-- Deklaration für <head> & <body> -->
```

2.6 Processing Instructions

Processing Instructions (PIs) erlauben Dokumenten, Anweisungen für Anwendungsprogramme zu enthalten.

Processing Instructions
[16] PI ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
[17] PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))

PIs sind kein Bestandteil der Zeichendaten des Dokuments, müssen jedoch an die Anwendung weitergereicht werden. Die PIs beginnen mit einem Ziel (PITarget), das dazu dient, die Anwendung zu identifizieren, an die sich die Anweisung richtet. Die Zielnamen XML, xml und so weiter sind für die Standardisierung in dieser oder einer zukünftigen Version dieser Spezifikation reserviert. Der XML-Notationsmechanismus darf für die formale Deklaration von PI-Zielen benutzt werden.

2.7 CDATA-Abschnitte

CDATA-Abschnitte dürfen überall dort stehen, wo auch Zeichendaten erlaubt sind. Sie dienen dazu, ganze Textblöcke zu schützen, die Zeichen enthalten, die normalerweise als Markup interpretiert würden. CDATA-Abschnitte beginnen mit der Zeichenkette <![CDATA[und enden mit]]>.

CDATA-Abschnitte
[18] CDSect ::= CDStart CData CDEnd
[19] CDStart ::= '<![CDATA['
[20] CData ::= (Char* - (Char* ']]>' Char*))
[21] CDEnd ::= ']]>'

Innerhalb eines CDATA-Abschnittes wird *ausschließlich* CDEnd erkannt. Das heißt die öffnende spitze Klammer und das et-Zeichen dürfen in ihrer literalen Form erscheinen. Sie müssen (und können) nicht mit < bzw. & geschützt werden. CDATA-Abschnitte können nicht verschachtelt werden.

Folgendes Beispiel zeigt einen CDATA-Abschnitt, in dem <gruss> und </gruss> als Zeichendaten und nicht als Markup erkannt werden.

```
<![CDATA[<gruss>Hallo Welt!</gruss>]]>
```

2.8 Prolog und Dokumenttyp-Deklaration

XML-Dokumente können und sollten mit einer **XML-Deklaration** beginnen, die die verwendete XML-Version spezifiziert. Beispielsweise sind die folgenden Zeilen ein vollständiges XML-Dokument, wohlgeformt, aber nicht gültig.

```
<?xml version="1.0"?>
<gruss>Hallo Welt!</gruss>
```

Gleiches gilt hierfür:

```
<gruss>Hallo Welt!</gruss>
```

Die Versionsnummer »1.0« sollte benutzt werden, um Konformität zu dieser Version der Spezifikation anzuzeigen. Es ist ein Fehler, wenn ein Dokument den Wert »1.0« benutzt und nicht konform zu dieser Version der Spezifikation ist. Es ist die Absicht der XML-Arbeitsgruppe, späteren Versionen dieser Spezifikation andere Nummern als »1.0« zu geben. Diese Absichtserklärung ist jedoch kein Versprechen, irgendwelche weiteren Versionen von XML zu erstellen, noch, falls es weitere gibt, irgendein bestimmtes Numerierungsschema zu verwenden. Da zukünftige Versionen nicht ausgeschlossen sind, wurde die Deklaration eingeführt, um die Möglichkeit der automatischen Versionserkennung zu erlauben, sofern dies notwendig werden sollte. Prozessoren dürfen einen Fehler anzeigen, wenn sie ein Dokument einer Version bekommen, die sie nicht unterstützen.

Die Funktion des Markup in einem XML-Dokument ist es, die Aufteilung auf Speicherungseinheiten und die logische Struktur zu beschreiben sowie Attribut-Wert-Paare mit der logischen Struktur zu verbinden. XML stellt einen Mechanismus zur Verfügung, die Dokumenttyp-Deklaration, um Beschränkungen der logischen Struktur zu definieren und die Verwendung von vordefinierten Speichereinheiten zu unterstützen. Ein XML-Dokument ist **gültig**, wenn es eine dazugehörige Dokumenttyp-Deklaration besitzt und wenn sich das Dokument an die darin formulierten Beschränkungen hält.

Die Dokumenttyp-Deklaration muß vor dem ersten Element im Dokument stehen.

Prolog

[22] prolog	::= XMLDecl? Misc* (doctypedecl Misc*)?
[23] XMLDecl	::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24] VersionInfo	::= S 'version' Eq (' VersionNum ' | " VersionNum ")
[25] Eq	::= S? '=' S?
[26] VersionNum	::= ([a-zA-Z0-9_.:] | '-')+
[27] Misc	::= Comment | PI | S

Die XML-Dokumenttyp-Deklaration enthält oder verweist auf Markup-Deklarationen, die eine Grammatik für eine Klasse von Dokumenten bilden. Diese Grammatik ist bekannt als **Dokumenttyp-Definition**, kurz **DTD**. Die Dokumenttyp-Deklaration kann entweder auf eine externe Teilmenge (eine besondere Art eines externen Entity) verweisen, die Markup-Deklarationen enthält, oder sie kann Markup-Deklarationen direkt in einer internen Teilmenge enthalten oder beides. Die DTD für ein Dokument besteht aus beiden Teilmengen zusammen.

Eine **Markup-Deklaration** ist eine Elementtyp-Deklaration, eine Attributlisten-Deklaration oder eine Notation-Deklaration. Diese Deklarationen dürfen ganz oder teilweise innnerhalb von Parameter-Entities stehen, wie in den Wohlgeformtheits- und Gültigkeitsbeschränkungen unten beschrieben. Für vollständigere Informationen siehe Abschnitt 4.

Dokumenttyp-Definition

[28] doctypedecl	::= '<!DOCTYPE' S Name (S ExternalID)? S? ('[' (markupdecl | PEReference | S)* ']' S?)? '>'	[GKB: Wurzel-Elementtyp]
[29] markupdecl	::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment	[GKB: Ordentliche Deklaration/PE-Verschachtelung] [WGB: PEs in interner Teilmenge]

Die Markup-Deklarationen dürfen ganz oder teilweise durch den Ersetzungstext von Parameter-Entities gebildet werden. Die Produktionen für einzelne Nicht-Terminale (elementdecl, AttlistDecl usw.), die später in dieser Spezifikation folgen, beschreiben die Deklarationen, *nachdem* sämtliche Parameter-Entities aufgelöst wurden.

Gültigkeitsbeschränkung: Wurzel-Elementtyp
Der Name in der Dokumenttyp-Deklaration muß mit dem Elementtyp des Wurzel-Elements übereinstimmen.

Gültigkeitsbeschränkung: Ordentliche Deklaration/PE-Verschachtelung
Der Ersetzungstext von Parameter-Entities muß ordentlich mit Markup-Deklarationen verschachtelt sein. Das heißt: wenn entweder das erste oder das letzte Zeichen einer Markup-Deklaration (markupdecl, siehe oben) im Ersetzungstext für eine Parameter-Entity-Referenz enthalten ist, dann müssen beide im selben Ersetzungstext enthalten sein.

Wohlgeformtheitsbeschränkung: PEs in interner Teilmenge
Im internen Teil der DTD können Parameter-Entity-Referenzen nur dort stehen, wo auch Markup-Deklarationen stehen können, nicht jedoch innerhalb von Markup-Deklarationen. (Dies gilt nicht für Referenzen, die in externen Parameter-Entities erscheinen oder für die externe Teilmenge.

Wie die interne Teilmenge so muß auch die externe Teilmenge und jedes externe Parameter-Entity, auf das die DTD Bezug nimmt, aus einer Folge von vollständigen Markup-Deklarationen des Typs bestehen, der durch das Nicht-Terminal markupdecl erlaubt wird, vermengt mit Leerraum (White Space) oder Parameter-Entity-Referenzen. Allerdings dürfen Teile des Inhalts der externen Teilmenge oder von externen Parameter-Entities unter Verwendung von bedingten Abschnitten ignoriert werden. Dies ist innerhalb der internen Teilmenge nicht erlaubt.

Externe Teilmenge		
[30] extSubset	::= TextDecl? extSubsetDecl	
[31] extSubsetDecl	::= (markupdecl | conditionalSect | PEReference | S)*	

Die externe Teilmenge und externe Parameter-Entities unterscheiden sich darüber hinaus von der internen Teilmenge dadurch, daß Parameter-Entity-Referenzen *innerhalb* von Markup-Deklarationen erlaubt sind, nicht nur *zwischen* Markup-Deklarationen.

Ein Beispiel:

```
<?xml version="1.0"?>
<!DOCTYPE gruss SYSTEM "hallo.dtd">
<gruss>Hallo Welt!</gruss>
```

Der System-Identifier hallo.dtd gibt den URI einer DTD für das Dokument an.

Die Deklaration kann auch lokal angegeben werden, wie in folgendem Beispiel.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE gruss [
    <!ELEMENT gruss (#PCDATA)>
]>
<gruss>Hallo Welt!</gruss>
```

Wenn sowohl die externe als auch die interne Teilmenge verwendet werden, sollte die interne Teilmenge vor der externen Teilmenge stehen. Dies hat den Effekt, daß Entity- und Attributlisten-Deklarationen in der internen Teilmenge

Vorrang vor jenen in der externen Teilmenge haben.

2.9 Standalone-Dokumentdeklaration

Markup-Deklarationen können den Inhalt eines Dokuments beeinflussen, wie er von einem XML-Prozessor an eine Anwendung weitergereicht wird. Beispiele sind Vorgaben für Attributwerte sowie Entity-Deklarationen. Die **Standalone-Dokumentdeklaration** (engl. alleinstehend), die als Teil der XML-Deklaration erscheinen darf, zeigt an, ob es Deklarationen gibt, die außerhalb des Dokumentes stehen.

Standalone-Dokumentdeklaration	
[32] SDDecl ::= S 'standalone' Eq (("'" ('yes' | 'no') "'") | ('"' ('yes' | 'no') '"'))	[GKB: Standalone-Dokumentdeklaration]

In einer Standalone-Dokumentdeklaration zeigt der Wert »yes« (Ja), daß es keine Markup-Deklarationen außerhalb des Dokuments gibt (weder in der externen Teilmenge der DTD noch in einem externen Parameter-Entity, auf die von der internen Teilmenge verwiesen wird), die die Informationen beeinflussen, die vom XML-Prozessor an die Anwendung weitergereicht werden. Der Wert »no« (Nein) zeigt an, daß möglicherweise solche externen Markup-Deklarationen vorhanden sind. Beachten Sie, daß die Standalone-Dokumentdeklaration lediglich anzeigt, ob es externe *Deklarationen* gibt. Das Vorhandensein von Referenzen auf externe *Entities* in einem Dokument, die intern deklariert sind, ändert den Standalone-Status nicht.

Wenn es keine externen Markup-Deklarationen gibt, hat die Standalone-Dokumentdeklaration keine Bedeutung. Wenn es externe Markup-Deklarationen gibt, jedoch keine Standalone-Dokumentdeklaration vorhanden ist, wird der Wert »no« angenommen.

Ein XML-Dokument, für das standalone="no" gilt, kann algorithmisch in ein Standalone-Dokument konvertiert werden, was für netzweite Anwendungen wünschenswert sein kann.

Gültigkeitsbeschränkung: Standalone-Dokumentdeklaration
Die Standalone-Dokumentdeklaration muß den Wert »no« haben, falls eine externe Markup-Deklaration eine der folgenden Deklarationen enthält:

- Attribute mit einem Vorgabewert (Default), falls Elemente, für die diese Attribute gelten, im Dokument erscheinen, ohne daß Werte für diese Attribute angegeben wurden
- andere Entities als amp, lt, gt, apos, quot, falls Referenzen zu solchen Entities im Dokument erscheinen
- Attribute mit Werten, die normalisiert werden, wobei das Attribut im Dokument mit einem Wert erscheint, der sich durch die Normalisierung ändert
- Elementtypen, deren Inhalt weitere Elemente enthält, falls Leerraum (White Space) direkt innerhalb einer Instanz solcher Typen erscheint

```
<?xml version="1.0" standalone='yes'?>
```

2.10 Behandlung von Leerraum

Bei dem Editieren von XML-Dokumenten ist es oft angenehm, Leerraum (White Space; Leerzeichen (spaces), Tabulatoren, Leerzeilen; innerhalb dieser Spezifikation mit dem Nicht-Terminal S bezeichnet) zu verwenden, um das Markup für eine bessere Lesbarkeit voneinander abzusetzen. Dieser Leerraum soll üblicherweise beim Versenden des Dokuments nicht erhalten bleiben. Auf der anderen Seite gibt es häufig »signifikanten« Leerraum, der auch beim Versenden erhalten bleiben soll, beispielsweise in Gedichten und Quellcode.

Ein XML-Prozessor muß stets alle Zeichen in einem Dokument, die nicht zum Markup gehören, an die Anwendung weiterreichen. Ein validierender XML-Prozessor muß die Anwendung außerdem darüber informieren, welche Leerraumzeichen im Inhalt eines Elements stehen.

Ein besonderes Attribut namens **xml:space** kann einem Element zugewiesen werden, um anzuzeigen, daß Leerraum in diesem Element von Anwendungen erhalten werden soll. In gültigen Dokumenten muß dieses Attribut, wie jedes andere, deklariert werden. Bei der Deklaration muß es als Aufzählungstyp, dessen einzige Werte »default« (Vorgabe) und »preserve« (Erhalten) sind, geschrieben werden. Zum Beispiel:

```
<!ATTLIST gedicht   xml:space (default|preserve) 'preserve'>
```

Der Wert »default« zeigt an, daß die normale Leerraumbehandlung einer Anwendung für dieses Element akzeptabel ist. Der Wert »preserve« zeigt die Absicht an, daß Anwendungen sämtlichen Leerraum erhalten. Diese erklärte Absicht gilt für alle Elemente innerhalb des Elements, für das »preserve« angegeben wurde, es sei denn, es ist für ein eingebettetes Element explizit mit dem Attribut xml:space überschrieben worden.

Vom Wurzelelement eines beliebigen Dokuments wird angenommen, daß es keine Leerraumbehandlung der Anwendung vorsieht, es sei denn, es gibt einen Wert für dieses Attribut vor -- oder das Attribut ist mit einem Vorgabewert deklariert.

2.11 Behandlung des Zeilenendes

Analysierte (parsed) XML-Entities sind oft in Dateien abgelegt, die, zur leichteren Änderbarkeit, in Zeilen unterteilt sind. Diese Zeilen sind üblicherweise durch Kombinationen der Zeichen Wagenrücklauf (carriage-return, #xD) und Zeilenvorschub (line-feed, #xA) getrennt.

Um die Aufgabe von Anwendungen zu erleichtern, muß ein XML-Prozessor das einzelne Zeichen #xA an die Anwendung weiterreichen, wo immer ein externes analysiertes Entity oder der literale Entity-Wert eines internen analysierten Entity entweder die literale Zweizeichenfolge #xD#xA oder ein einzelnes Literal #xD enthält. (Dieses Verhalten läßt sich angenehmerweise erreichen, indem man alle Zeilenumbrüche beim Lesen vor dem Analysieren zu #xA normalisiert.)

2.12 Identifikation der Sprache

In der Dokumentenverarbeitung ist es oft nützlich, die natürliche oder formale Sprache, in der der Inhalt geschrieben ist, zu identifizieren. Ein besonderes Attribut namens **xml:lang** kann in Dokumente eingefügt werden, um die für den Inhalt oder für die Attributwerte von beliebigen Elementen verwendete Sprache anzugeben. In gültigen Dokumenten muß dieses Attribut, wie jedes andere, deklariert werden. Die Werte dieses Attributs sind Sprachencodes gemäß Definition in [IETF RFC 1766]:

Identifikation der Sprache	
[33] LanguageID	::= Langcode ('-' Subcode)*
[34] Langcode	::= ISO639Code | IanaCode | UserCode
[35] ISO639Code	::= ([a-z] | [A-Z]) ([a-z] | [A-Z])
[36] IanaCode	::= ('i' | 'I') '-' ([a-z] | [A-Z])+
[37] UserCode	::= ('x' | 'X') '-' ([a-z] | [A-Z])+
[38] Subcode	::= ([a-z] | [A-Z])+

Der Langcode ist etwas folgender Art:

- ein Zwei-Buchstaben Sprachencode gemäß [ISO 639]
- ein Sprachencode, der bei der Internet Assigned Numbers Authority [IANA] registriert ist; sie beginnen mit dem Präfix »i-« oder »I-«.
- ein Sprachencode, den der Benutzer selbst vergeben hat oder der zwischen zwei Parteien zum privaten Gebrauch vereinbart wurde; diese Codes müssen mit dem Präfix »x-« oder »X-« beginnen, um Konflikte mit Namen zu vermeiden, die später standardisiert oder bei der IANA registriert werden.

Es dürfen beliebig viele Subcode-Segmente existieren. Falls das erste Subcode-Segment vorhanden ist und aus zwei Buchstaben besteht, dann muß es ein Ländercode aus [ISO 3166] sein. Falls der erste Subcode aus mehr als zwei Buchstaben besteht, dann muß es ein bei der IANA registrierter Subcode für die fragliche Sprache sein; es sei denn, Langcode beginnt mit einem »x-« oder »X-«.

Es ist üblich, den Sprachencode in Kleinbuchstaben und den Ländercode (falls vorhanden) in Großbuchstaben anzugeben. Beachten Sie, daß diese Werte, anders als andere Namen in XML-Dokumenten, von der Groß/Kleinschreibung unabhängig sind.

Ein Beispiel:

```
<p xml:lang="en">The quick brown fox jumps over the lazy dog.</p>
<p xml:lang="de-DE">In welcher Straße hast du geparkt?</p>
<p xml:lang="de-CH">In welcher Strasse hast du parkiert?</p>
<sp who="Faust" desc='leise' xml:lang="de">
    <l>Habe nun, ach! Philosophie,</l>
    <l>Juristerei, und Medizin</l>
    <l>und leider auch Theologie</l>
    <l>durchaus studiert mit heißem Bemüh'n.</l>
    </sp>
```

Die mit xml:lang gemachte Deklaration wird auf alle Attribute und den Inhalt des Elementes angewandt, für das xml:lang spezifiziert wurde. Innerhalb des Inhalts kann ein weiteres Attribut xml:lang dieses Verhalten für ein eingebettetes Element überschreiben.

Eine einfache Deklaration für xml:lang kann folgende Form annehmen:

```
xml:lang  NMTOKEN  #IMPLIED
```

Es können auch Vorgabewerte -- falls sinnvoll -- angegeben werden. In einer Sammlung von französischen Gedichten für deutschsprachige Studenten mit Erläuterungen und Bemerkungen in Deutsch kann die Deklaration für das xml:lang-Attribut etwa folgendermaßen aussehen:

```
<!ATTLIST gedicht       xml:lang NMTOKEN 'fr'>
<!ATTLIST erlaeuterung  xml:lang NMTOKEN 'de'>
<!ATTLIST bemerkung     xml:lang NMTOKEN 'de'>
```

3 Logische Strukturen

Jedes XML-Dokument enthält ein oder mehrere **Elemente**, die entweder durch Start- und End-Tags oder, im Falle eines leeren Elements, durch ein Leeres-Element-Tag begrenzt sind. Jedes Element hat einen Typ, der durch einen Namen, auch »**generic identifier**« (**GI**) genannt, identifiziert wird, und es kann auch eine Menge von Attributspezifikationen haben. Jede Attributspezifikation hat einen Namen und einen Wert.

Element
[39] element ::= EmptyElemTag | STag content ETag [WGB: Elementtyp-Übereinstimmung] [GKB: gültiges Element]

Diese Spezifikation macht keine Einschränkungen hinsichtlich Semantik, Verwendung, Benennung (abgesehen von der Syntax) von Elementtypen und Attributen, mit der Ausnahme, daß Namen, deren Anfang zu (('X'|'x')('M'|'m')('L'|'l')) paßt, für die Standardisierung in dieser oder einer zukünftigen Verision dieser Spezifikation reserviert sind.

Wohlgeformtheitsbeschränkung: Elementtyp-Übereinstimmung
Der Name im End-Tag eines Elements muß mit dem Elementtyp im Start-Tag übereinstimmen.

Gültigkeitsbeschränkung: gültiges Element
Ein Element ist gültig, wenn es eine Deklaration gibt, die zu elementdecl paßt, wobei der Name zu dem Elementtyp paßt und eine der folgenden Bedingungen gilt:

1. Die Deklaration paßt zu EMPTY, und das Element hat keinen Inhalt.
2. Die Deklaration paßt zu children, und die Folge der Kind-Elemente gehört zu der Sprache, die durch den regulären Ausdruck im Inhaltsmodell generiert wird, mit optionalem Leerraum (Zeichen, die zu dem Nicht-Terminal S passen) zwischen jedem Paar von Kind-Elementen.
3. Die Deklaration paßt zu Mixed, und der Inhalt besteht aus Zeichendaten und Kind-Elementen, deren Typen zu den Namen im Inhaltsmodell passen.
4. Die Deklaration paßt zu Any, und die Typen aller Kind-Elemente sind deklariert worden.

3.1 Start-Tags, End-Tags und Leeres-Element-Tags

Der Beginn jedes nicht-leeren XML-Elements ist durch ein **Start-Tag** markiert.

Start-Tag		
[40] STag	::= '<' Name (S Attribute)* S? '>'	[WGB: Eindeutige Attributspezifikation]
[41] Attribute	::= Name Eq AttValue	[GKB: Typ des Attributwertes] [WGB: Keine externen Entity-Referenzen] [WGB: Kein < in Attribut-Werten]

Der Name in den Start- und End-Tags ist der **Typ** des Elements. Die Name-AttValue-Paare werden als **Attribut-Spezifikation** des Elements bezeichnet, wobei der Name in jedem Paar der **Attribut-Name** und der Inhalt des AttValue (der Text zwischen den '- oder "-Zeichen) der **Attribut-Wert** ist.

Wohlgeformtheitsbeschränkung: Eindeutige Attributspezifikation
Kein Attributname darf mehr als einmal in demselben Start-Tag oder Leeres-Element-Tag erscheinen.

Gültigkeitsbeschränkung: Typ des Attributwertes
Das Attribut muß deklariert worden sein. Der Wert muß von dem Typ sein, der für ihn deklariert wurde (siehe dazu 3.3).

Wohlgeformtheitsbeschränkung: Keine externen Entity-Referenzen
Attributwerte können weder direkte noch indirekte Entity-Referenzen auf externe Entities enthalten.

Wohlgeformtheitsbeschränkung: Kein < in Attribut-Werten
Der Ersetzungstext eines Entities, auf das direkt oder indirekt in einem Attributwert verwiesen wird, darf kein < enthalten (außer <).

Ein Beispiel für einen Start-Tag:

```
<termdef id="dt-hund" term="hund">
```

Das Ende jedes Elements, das mit einem Start-Tag beginnt, muß mit einem **End-Tag** markiert werden. Dieses muß einen Namen enthalten, der den Elementtyp wiederholt, wie er vom Start-Tag vorgegeben wurde.

End-Tag
[42] ETag ::= '</' Name S? '>'

Ein Beispiel eines End-Tags:

```
</termdef>
```

Der Text zwischen Start- und End-Tag wird der **Inhalt** des Elements genannt.

Inhalt von Elementen
[43] content ::= (element | CharData | Reference | CDSect | PI | Comment)*

Wenn ein Element **leer** (empty) ist, dann muß es entweder durch einen Start-Tag, auf den unmittelbar ein End-Tag folgt, oder durch ein Leeres-Element-Tag dargestellt werden. Ein **Leeres-Element-Tag** hat eine besondere Form:

Leeres-Element-Tag
[44] EmptyElemTag ::= '<' Name (S Attribute)* S? '/>' [WGB: Eindeutige Attributspezifikation]

Leeres-Element-Tags können für jedes Element benutzt werden, das keinen Inhalt hat, unabhängig davon, ob es mit dem Schlüsselwort EMPTY deklariert wurde. Zwecks Zusammenarbeit muß (und kann nur) das Leeres-Element-Tag nur für Elemente verwendet werden, die als EMPTY deklariert wurden.

Beispiele für leere Elemente:

```
<IMG align="left" src="http://www.w3.org/Icons/WWW/w3c_home" />
<br></br>
<br/>
```

3.2 Elementtyp-Deklarationen

Die Element-Struktur eines XML-Dokuments kann zum Zwecke der Validierung durch Elementtyp- und Attributlisten-Deklarationen beschränkt werden. Eine Elementtyp-Deklaration beschränkt den Inhalt eines Elements.

Elementtyp-Deklarationen beschränken oft, welche Elementtypen als Kinder des Elements erscheinen können. Benutzeroptional kann ein XML-Prozessor eine Warnung ausgeben, falls eine Deklaration einen Elementtyp nennt, für den es keine Deklaration gibt. Dies ist aber kein Fehler.

Eine **Elementtyp-Deklaration** hat folgende Form:

Elementtyp-Deklaration		
[45] elementdecl	::= '<!ELEMENT' S Name S contentspec S? '>'	[GKB: Eindeutige Elementtyp-Deklarationen]
[46] contentspec	::= 'EMPTY' | 'ANY' | Mixed | children	

Hierbei bezeichnet Name den zu deklarierenden Elementtyp.

Gültigkeitsbeschränkung: Eindeutige Elementtyp-Deklarationen
Kein Elementtyp darf mehr als einmal deklariert werden.

Beispiel für Elementtyp-Deklarationen:

```
<!ELEMENT br EMPTY>
<!ELEMENT p (#PCDATA|emph)* >
<!ELEMENT %name.para; %content.para; >
<!ELEMENT container ANY>
```

3.2.1 Element-Inhalt

Ein Elementtyp hat **Element-Inhalt**, wenn Elemente dieses Typs ausschließlich Kindelemente (keine Zeichendaten) enthalten dürfen, die optional durch Leerraum (Zeichen, die zu dem Nicht-Terminal S passen) unterteilt sind. In diesem Fall enthält die Beschränkung ein Inhaltsmodell, eine einfache Grammatik, die die erlaubten Typen der Kindelemente und die Reihenfolge, in der sie erscheinen dürfen, festlegt. Die Grammatik ist auf Inhaltsteilen (content particles, cps) aufgebaut, die aus Namen, Auswahllisten (choice) und Folgen (sequence) von Inhaltsteilen bestehen.

Modelle für Element-Inhalt		
[47] children	::= (choice | seq) ('?' | '*' | '+')?	
[48] cp	::= (Name | choice | seq) ('?' | '*' | '+')?	
[49] choice	::= '(' S? cp (S? '|' S? cp)* S? ')'	[GKB: Ordentliche Gruppierung/PE-Verschachtelung]
[50] seq	::= '(' S? cp (S? ',' S? cp)* S? ')'	[GKB: Ordentliche Gruppierung/PE-Verschachtelung]

Hierbei ist Name der Typ eines Elements, das als Kind vorkommen darf. Jeder Inhaltsteil einer Auswahlliste kann im Elementinhalt an der Stelle stehen, an der die Auswahlliste in der Grammatik steht. Inhaltsteile in einer Folge müssen alle im Elementinhalt und in der vorgegebenen Reihenfolge erscheinen. Das optionale Zeichen, das auf einen Namen oder eine Liste folgt, legt fest, ob der Inhalt oder die Inhaltsteile in der Liste einmal oder mehrmals (+), keinmal oder mehrmals (*) oder keinmal oder einmal (?) auftreten dürfen. Das Fehlen eines solchen Operators bedeutet, daß das Element oder der Inhaltsteil genau einmal erscheinen muß. Diese Syntax und Bedeutung sind identisch zu denen, die in den Produktionen dieser Spezifikation Verwendung finden.

Der Inhalt eines Elements paßt zu einem Inhaltsmodell, dann und nur dann, wenn es möglich ist, unter Befolgung der Sequenz-, Auswahl- und Wiederholungsoperatoren einen Pfad durch das Inhaltsmodell zu finden, wobei jedes Element im Inhalt zu einem Elementtyp im Inhaltsmodell paßt. Zwecks Kompatibilität ist es ein Fehler, wenn ein Element im Dokument zu mehr als einem Elementtyp im Inhaltsmodell paßt. Für weitere Informationen siehe Abschnitt 11.

Gültigkeitsbeschränkung: Ordentliche Gruppierung/PE-Verschachtelung
Der Ersetzungstext von Parameter-Entities muß ordentlich mit geklammerten Gruppen verschachtelt sein. Das heißt: Wenn entweder die öffnende oder die schließende Klammer in einem choice-, seq- oder Mixed-Konstrukt im Ersetzungstext eines Parameter-Entity enthalten ist, dann müssen beide im selben Ersetzungstext enthalten sein. Zwecks Zusammenarbeit gilt: Wenn eine Parameter-Entity-Referenz in einem choice-, seq- oder Mixed-Konstrukt erscheint, dann sollte dessen Ersetzungstext nicht leer sein und weder das erste noch das letzte nicht-leere Zeichen (non-blank) des Ersetzungstextes sollte ein Konnektor (| oder ,) sein.

Beispiele für Modelle mit Element-Inhalt:

```
<!ELEMENT spec (front, body, back?)>
<!ELEMENT div1 (head, (p | list | note)*, div2*)>
<!ELEMENT dictionary-body (%div.mix; | %dict.mix;)*>
```

3.2.2 Gemischter Inhalt

Ein Element hat **gemischten Inhalt**, wenn Elemente dieses Typs Zeichendaten enthalten dürfen, die optional mit Kindelementen gemischt sind. In diesem Fall können die Typen der Kindelemente beschränkt werden, nicht jedoch ihre Reihenfolge oder ihre Anzahl.

Deklaration von gemischtem Inhalt		
[51] Mixed	::= '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*' | '(' S? '#PCDATA' S? ')'	[GKB: Ordentliche Gruppierung/PE-Verschachtelung] [GKB: Keine doppelten Typen]

wobei Name den Elementtyp bezeichnet, der als Kind erscheinen darf.

Gültigkeitsbeschränkung: Keine doppelten Typen
Derselbe Name darf nicht mehr als einmal in einer Deklaration von gemischtem Inhalt erscheinen.

Beispiel für Deklarationen von gemischtem Inhalt:

```
<!ELEMENT p (#PCDATA|a|ul|b|i|em)*>
<!ELEMENT p (#PCDATA | %font; | %phrase; | %special; | %form;)* >
<!ELEMENT b (#PCDATA)>
```

3.3 Attributlisten-Deklaration

Attribute werden verwendet, um Name-Wert-Paare mit Elementen zu verknüpfen. Attribut-Spezifikationen dürfen ausschließlich innerhalb von Start-Tags und Leeres-Element-Tags erscheinten. Folglich ist die Produktion zu deren Erkennung im Abschnitt 3.1 zu finden. Deklarationen von Attribut-Listen dürfen benutzt werden, um

- die Menge der Attribute zu definieren, die zu einem gegebenen Elementtyp gehören,
- Typbeschränkungen für diese Attribute aufzustellen und
- Vorgabewerte für Attribute anzugeben.

Attributlisten-Deklarationen spezifizieren den Namen, Datentyp und ggf. den Vorgabewert jedes Attributs, das zu einem gegebenen Element gehört:

Attributlisten-Deklaration		
[52] AttlistDecl	::= '<!ATTLIST' S Name AttDef* S? '>'	
[53] AttDef	::= S Name S AttType S DefaultDecl	

Der Name in der AttlistDecl-Regel ist der Typ eines Elements. Benutzeroptional kann ein XML-Prozessor eine Warnung ausgeben, wenn Attribute für einen nicht deklarierten Elementtyp deklariert werden; dies ist jedoch kein Fehler. Der Name in der AttDef-Regel ist der Name des Attributs.

Falls mehr als eine AttlistDecl für ein gegebenes Element angegeben wird, werden sie alle vereinigt. Falls mehr als eine Definition für dasselbe Attribut eines gegebenen Elementtyps angegeben wird, so gilt zwingend die erste Deklaration, und alle weiteren werden ignoriert. Zwecks Zusammenarbeit können sich Verfasser von DTDs entscheiden, maximal eine Attributlisten-Deklaration für einen gegebenen Elementtyp, maximal eine Attributdefinition für einen gegebenen Attributnamen und mindestens eine Attributdefinition in jeder Attributlisten-Deklaration vorzusehen. Zwecks Zusammenarbeit kann ein XML-Prozessor benutzeroptional eine Warnung ausgeben, wenn mehr als eine Attributlisten-Deklaration für einen gegeben Elementtyp angegeben ist oder wenn mehr als eine Attributdefinition für ein gegebenes Attribut angegeben ist; dies ist jedoch kein Fehler.

3.3.1 Attribut-Typen

Es gibt drei Arten von XML-Attribut-Typen: einen Zeichenketten-Typ (string), eine Menge von Token-Typen (tokenized) und einen Aufzählungstyp (enumerated). Der Zeichenketten-Typ kann jede literale Zeichenkette als Wert aufnehmen. Der Token-Typ hat verschiedene lexikalische und semantische Beschränkungen:

Attributtypen		
[54] AttType	::= StringType | TokenizedType | EnumeratedType	
[55] StringType	::= 'CDATA'	
[56] TokenizedType	::= 'ID'	[GKB: ID] [GKB: Eine ID pro Elementtyp] [GKB: Vorgabe für ID-Attribute]
	| 'IDREF'	[GKB: IDREF]
	| 'IDREFS'	[GKB: IDREF]
	| 'ENTITY'	[GKB: Entity-Name]
	| 'ENTITIES'	[GKB: Entity-Name]
	| 'NMTOKEN'	[GKB: Name-Token]
	| 'NMTOKENS'	[GKB: Name-Token]

Gültigkeitsbeschränkung: ID
Werte des Typs ID müssen zur Produktion Name passen. Ein Name darf nicht mehr als einmal als Wert dieses Typs in einem XML-Dokument erscheinen. Das heißt, ID-Werte müssen die Elemente, die sie tragen, eindeutig identifizieren.

Gültigkeitsbeschränkung: Eine ID pro Elementtyp
Kein Elementtyp darf mehr als ein ID-Attribut besitzen.

Gültigkeitsbeschränkung: Vorgabe für ID-Attribute
Ein ID-Attribut muß einen deklarierten Vorgabewert von #IMPLIED oder #REQUIRED haben.

Gültigkeitsbeschränkung: IDREF
Werte des Typs IDREF müssen zur Produktion Name passen, und Werte des Typs IDREFS müssen zur Produktion Names passen. Jeder Name muß mit dem Wert eines ID-Attributs eines Elements im XML-Dokument übereinstimmen. Das heißt, IDREF-Werte müssen zum Wert irgendeines ID-Attributs passen.

Gültigkeitsbeschränkung: Entity-Name
Werte des Typs ENTITY müssen zur Produktion Name passen, Werte des Typs ENTITIES müssen zur Produktion Names passen. Jeder Name muß mit dem Namen eines in der DTD deklarierten, nicht analysierten (unparsed) Entity übereinstimmen.

Gültigkeitsbeschränkung: Name-Token
Werte des Typs NMTOKEN müssen zur Produktion Nmtoken passen, Werte des Typs NMTOKENS müssen zur Produktion Nmtokens passen.

Aufzählungs-Attributtypen können einen Wert aus einer deklarierten Liste von Werten annehmen. Es gibt zwei Arten

von Aufzählungstypen:

Aufzählungs-Attributtypen		
[57] EnumeratedType	::= NotationType | Enumeration	
[58] NotationType	::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'	[GKB: Notation-Attribute]
[59] Enumeration	::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'	[GKB: Aufzählung]

Ein NOTATION-Attribut identifiziert eine Notation, die in der DTD mit einem zugehörigen System- und/oder Public-Identifier deklariert ist. Die Notation dient dazu, das Element mit diesem Attribut zu interpretieren.

Gültigkeitsbeschränkung: Notation-Attribute
Werte dieses Typs müssen zu einem der Notation-Namen passen, die in der Deklaration enthalten sind. Alle Notation-Namen in der Deklaration müssen deklariert sein.

Gültigkeitsbeschränkung: Aufzählung
Werte dieses Typs müssen zu einem der Nmtoken-Token aus der Deklaration passen.

Zwecks Zusammenarbeit sollte jedes Nmtoken nicht mehr als einmal in den Aufzählungsattributen eines einzelnen Elementes vorkommen.

3.3.2 Attribut-Vorgaben

Eine Attribut-Deklaration enthält Informationen, ob ein Attribut vorkommen muß und, falls nicht, wie ein XML-Prozessor bei einem im Dokument fehlenden Attribut reagieren sollte.

Attribut-Vorgaben		
[60] DefaultDecl	::= '#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)	[GKB: Notwendiges Attribut] [GKB: korrekte Attribut-Vorgabe] [WGB: Kein < in Attribut-Werten] [GKB: Feste Attribut-Vorgabe]

In einer Attribut-Deklaration bedeutet #REQUIRED (notwendig), daß das Attribut immer angegeben werden muß und #IMPLIED (impliziert), daß es keinen Vorgabewert gibt. Wenn die Deklaration weder #REQUIRED noch #IMPLIED ist, enthält der AttValue-Wert den deklarierten **Vorgabe**-Wert. Das Schlüsselwort #FIXED (fest) zeigt an, daß das Attribut stets den Vorgabewert haben muß. Falls ein Vorgabewert deklariert ist, verhält sich ein XML-Prozessor bei einem weggelassenen Attribut so, als ob das Attribut mit dem Vorgabewert im Dokument stünde.

Gültigkeitsbeschränkung: Notwendiges Attribut
Falls die Attribut-Vorgabe das Schlüsselwort #REQUIRED ist, so muß das Attribut für alle Elemente des in der Attributlisten-Deklaration genanten Typs angegeben werden.

Gültigkeitsbeschränkung: korrekte Attribut-Vorgabe
Der deklarierte Vorgabewert muß den lexikalischen Beschränkungen des deklarierten Attribut-Typs genügen.

Gültigkeitsbeschränkung: Feste Attribut-Vorgabe
Wenn ein Attribut einen mit dem Schlüsselwort #FIXED deklarierten Vorgabewert besitzt, müssen alle Instanzen des Attributes den Vorgabewert besitzen.

Beispiel für Attributlisten-Deklarationen:

```
<!ATTLIST termdef
          id      ID      #REQUIRED
          name    CDATA   #IMPLIED>
<!ATTLIST list
          type    (bullets|ordered|glossary)  "ordered">
<!ATTLIST form
          method  CDATA   #FIXED "POST">
```

3.3.3 Normalisierung von Attribut-Werten

Bevor der Wert eines Attributs an die Anwendung weitergereicht oder auf Gültigkeit geprüft wird, muß der XML-Prozessor ihn folgendermaßen normalisieren:

- Im Falle einer Zeichenreferenz wird das referenzierte Zeichen an den Attribut-Wert angehängt.
- Im Falle einer Entity-Referenz wird der Ersetzungstext des Entity rekursiv verarbeitet.
- Im Falle eines Leerraum-Zeichens (whitespace character: #x20, #xD, #xA, #x9) wird #x20 an den normalisierten Wert angehängt. Eine Ausnahme ist die Sequenz »#xD#xA«, die Teil eines extern-analysierten (external parsed) Entity oder des literalen Werts eines intern analysierten (internal parsed) Entity ist. Für sie wird nur ein einziges #x20 angehängt.
- Andere Zeichen werden dem normalisierten Wert angehängt.

Falls der deklarierte Wert nicht CDATA ist, muß der XML-Prozessor den normalisierten Wert in folgender Weise weiterverarbeiten: Er entfernt alle führenden und abschließenden Leerzeichen (#x20) und ersetzt Folgen von Leerzeichen (#x20) durch ein einzelnes Leerzeichen (#x20).

Alle Attribute, für die keine Deklaration gelesen wurde, sollten von einem nicht-validierenden Parser behandelt werden, als ob sie als CDATA deklariert wären.

3.4 Bedingte Abschnitte

Bedingte Abschnitte sind Teile der externen Teilmenge der Dokumenttyp-Deklaration, die in Abhängigkeit von einem Schlüsselwort in die logische Struktur der DTD eingebunden oder von ihr ausgeschlossen sind.

Bedingte Abschnitte	
[61] conditionalSect	::= includeSect | ignoreSect
[62] includeSect	::= '<![' S? 'INCLUDE' S? '[' extSubsetDecl ']]>'
[63] ignoreSect	::= '<![' S? 'IGNORE' S? '[' ignoreSectContents* ']]>'
[64] ignoreSectContents	::= Ignore ('<![' ignoreSectContents ']]>' Ignore)*
[65] Ignore	::= Char* - (Char* ('<![' | ']]>') Char*)

So wie die internen und externen Teilmengen der DTD darf auch ein bedingter Abschnitt eine oder mehrere vollständige Deklarationen, Kommentare, Processing Instructions oder verschachtelte bedingte Abschnitte, vermischt mit Leerraum, enthalten.

Wenn das Schlüsselwort des bedingten Abschnitts INCLUDE heißt, dann wird der Inhalt des bedingten Abschnitts als Teil der DTD angesehen. Wenn das Schlüsselwort des bedingten Abschnitts IGNORE heißt, dann ist der Inhalt des bedingten Abschnitts kein logischer Teil der DTD. Beachten Sie, daß selbst der Inhalt von ignorierten bedingten Abschnitten für eine zuverlässige Analyse gelesen werden muß, um verschachtelte bedingte Abschnitte zu erkennen und um sicherzustellen, daß das Ende des äußeren (ignorierten) bedingten Abschnitts korrekt erkannt wird. Wenn ein bedingter Abschnitt mit dem Schlüsselwort INCLUDE innerhalb eines größeren bedingten Abschnitts mit dem Schlüsselwort IGNORE vorkommt, dann werden sowohl der innere als auch der äußere ignoriert.

Wenn das Schlüsselwort des bedingten Abschnitts ein Parameter-Entity ist, dann muß das Entity durch seinen Inhalt ersetzt werden, bevor der Prozessor entscheiden kann, ob er den bedingten Abschnitt ignoriert oder einbindet.

Ein Beispiel:

```
<!ENTITY % entwurf 'INCLUDE' >
<!ENTITY % fertig  'IGNORE' >

<![%entwurf;[
<!ELEMENT buch (kommentare*, titel, rumpf, anhaenge?)>
]]>
<![%fertig;[
<!ELEMENT buch (titel, rumpf, anhaenge?)>
]]>
```

4 Physikalische Strukturen

Ein XML-Dokument kann aus einer oder mehreren Speicherungseinheiten bestehen. Diese werden **Entities** genannt. Sie haben alle **Inhalt** und sind alle (abgesehen vom Dokument-Entity, s.u., und der externen Teilmenge der DTD) durch einen **Namen** identifiziert. Jedes XML-Dokument besitzt ein Entity namens **Dokument-Entity**, welches als Ausgangspunkt für den XML-Prozessor dient und das gesamte Dokument enthalten darf.

Entities dürfen entweder analysiert (parsed) oder nicht-analysiert (unparsed) sein. Der Inhalt eines **analysierten Entity** wird als sein **Ersetzungstext** bezeichnet. Dieser Text gilt als integraler Bestandteil des Dokuments.

Ein **nicht-analysiertes Entity** ist eine Ressource, deren Inhalt Text sein kann oder auch nicht, und falls es sich um Text handelt, nicht XML sein muß. Jedes nicht-analysierte Entity hat eine zugeordnete **Notation**, die durch ihren Namen identifiziert wird. XML erlegt dem Inhalt eines nicht-analysierten Entity keine Beschränkungen auf, es muß lediglich gewährleistet sein, daß der XML-Prozessor der Anwendung die Bezeichner für das Entity und für die Notation zur Verfügung stellt.

Analysierte Entities werden mit ihrem Namen durch Entity-Referenzen aufgerufen. Nicht-analysierte Entities werden mit ihrem Namen, der als Wert von ENTITY oder ENTITIES angegeben ist, aufgerufen.

Allgemeine Entities dienen der Verwendung innerhalb des Dokumentinhalts. In dieser Spezifikation werden allgemeine Entities oft unpräzise *Entities* genannt, sofern dies nicht zu Mehrdeutigkeit führt. Parameter-Entities sind analysierte Entities für die Benutzung innerhalb der DTD. Diese beiden Arten von Entities verwenden verschiedene Formen der Referenzierung und werden in unterschiedlichen Kontexten erkannt. Darüber hinaus belegen sie verschiedene Namensräume, d.h. ein Parameter-Entity und ein allgemeines Entity mit demselben Namen sind zwei verschiedenen Entities.

4.1 Zeichen- und Entity-Referenzen

Eine **Zeichenreferenz** verweist auf ein spezifisches Zeichen im Zeichensatz ISO/IEC 10646, etwa ein Zeichen, das auf dem Eingabegerät nicht direkt verfügbar ist.

Zeichenreferenz
[66] CharRef ::= '&#' [0-9]+ ';' | '&#x' [0-9a-fA-F]+ ';' [WGB: Erlaubtes Zeichen]

Wohlgeformtheitsbeschränkung: Erlaubtes Zeichen
Zeichen, auf die mittels einer Zeichenreferenz verwiesen wird, müssen zu der Produktion für Char passen.

Wenn die Zeichenreferenz mit »&#x« beginnt, stellen die Ziffern und Buchstaben bis zum abschließenden ; die hexadezimale Darstellung des Zeichencodes in ISO/IEC 10646 dar. Wenn sie nur mit »&#« beginnt, stellen die Ziffern bis zum abschließenden ; die dezimale Darstellung des Zeichencodes dar.

Eine **Entity-Referenz** verweist auf den Inhalt eines benannten Entity. Referenzen zu analysierten allgemeinen Entities verwenden das et-Zeichen (&) und das Semikolon (;) als Begrenzungszeichen. **Parameter-Entity-Referenzen** verwenden das Prozentzeichen (%) und das Semikolon (;) als Begrenzungszeichen.

Entity-Referenz		
[67] Reference	::= EntityRef | CharRef	
[68] EntityRef	::= '&' Name ';'	[WGB: Entity deklariert] [GKB: Entity deklariert] [WGB:Analysiertes Entity] [WGB: Keine Rekursion]
[69] PEReference	::= '%' Name ';'	[GKB: Entity deklariert] [WGB: Keine Rekursion] [WGB: In der DTD]

Wohlgeformtheitsbeschränkung: Entity deklariert
In einem Dokument ohne DTD, einem Dokument mit nur einer internen DTD-Teilmenge ohne Parameter-Entity-Referenzen oder einem Dokument mit »standalone='yes'« muß der Name in der Entity-Referenz zu dem in einer Entity-Deklaration passen. Eine Ausnahme ist, daß wohlgeformte Dokumente keine der folgenden Entities

deklarieren müssen: amp, lt, gt, apos, quot. Die Deklaration eines Parameter-Entity muß vor einer Referenz darauf erfolgen. Auf ähnliche Weise muß die Deklaration eines allgemeinen Entity vor einer Referenz erfolgen, die im Vorgabewert in einer Attributlisten-Deklaration steht. Beachten Sie, daß im Fall von Entities, die in der externen Teilmenge oder in externen Parameter-Entities deklariert sind, ein nicht-validierender Parser nicht verpflichtet ist, deren Deklarationen zu lesen und zu verarbeiten. Für diese Dokumente gilt die Regel, daß ein Entity nur dann deklariert sein muß, wenn eine Wohlgeformtheitsbeschränkung standalone='yes' gilt.

Gültigkeitsbeschränkung: Entity deklariert
In einem Dokument mit einer externen Teilmenge oder externen Parameter-Entities mit »standalone='no'« muß der in der Entity-Referenz angegebene Name mit dem in der Deklaration übereinstimmen. Zwecks Zusammenarbeit sollten gültige Dokumente die Entities amp, lt, gt, apos, quot wie in 4.6 definieren. Die Deklaration eines Parameter-Entity muß vor einer Referenz darauf erfolgen. Auf ähnliche Weise muß die Deklaration eines allgemeinen Entity vor einer Referenz erfolgen, die im Vorgabewert in einer Attributlisten-Deklaration steht.

Wohlgeformtheitsbeschränkung: Analysiertes Entity
Eine Entity-Referenz darf keinen Namen eines nicht-analysierten Entity enthalten. Auf nicht-analysierte Entities darf nur in solchen Attribut-Werten verwiesen werden, die vom Typ ENTITY oder ENTITIES sind.

Wohlgeformtheitsbeschränkung: Keine Rekursion
Ein analysiertes Entity darf weder direkt noch indirekt eine rekursive Referenz auf sich selbst enthalten.

Wohlgeformtheitsbeschränkung: In der DTD
Parameter-Entity-Referenzen dürfen nur in der DTD erscheinen.

Beispiele für Zeichen- und Entity-Referenzen:

```
Drücke <taste>kleiner-als</taste> (&#x3C;), um die
Optionen zu speichern.
Dieses Dokument wurde am &dokdatum; erstellt und
ist als &sicherheits-stufe; klassifiziert.
```

Beispiel einer Parameter-Entity-Referenz:

```
<!-- Deklariere Parameter-Entity "ISOLat2"... -->
<!ENTITY % ISOLat2
         SYSTEM "http://www.xml.com/iso/isolat2-xml.entities" >
<!-- ... und nun verweise darauf. -->
%ISOLat2;
```

4.2 Entity-Deklarationen

Entities werden auf folgende Weise deklariert:

Entity-Deklarationen	
[70] EntityDecl	::= GEDecl | PEDecl
[71] GEDecl	::= '<!ENTITY' S Name S EntityDef S? '>'
[72] PEDecl	::= '<!ENTITY' S '%' S Name S PEDef S? '>'
[73] EntityDef	::= EntityValue | (ExternalID NDataDecl?)
[74] PEDef	::= EntityValue | ExternalID

Der Name bezeichnet das Entity in einer Entity-Referenz oder, im Falle eines nicht-analysierten Entity, im Wert eines ENTITY- oder ENTITIES-Attributs. Wenn dasselbe Entity mehr als einmal deklariert wird, ist die erste Deklaration verbindlich. Benutzeroptional kann ein XML-Prozessor eine Warnung ausgeben, wenn Entities mehrfach deklariert sind.

4.2.1 Interne Entities

Wenn die Entity-Definition ein EntityValue ist, wird das definierte Entity **internes Entity** genannt. Es gibt keine separate Speicherungseinheit, der Inhalt des Entity ist in der Deklaration angegeben. Beachten Sie, daß eine gewisse

Verarbeitung von Entity- und Zeichenreferenzen im literalen Entity-Wert notwendig sein kann, um den korrekten Ersetzungstext zu erzeugen; siehe 4.5.

Ein internes Entity ist ein analysiertes (parsed) Entity.

Ein Beispiel für eine interne Entity-Deklaration:

```
<!ENTITY Pub-Status "Dies ist eine Vorabveröffentlichung dieser Spezifikation">
```

4.2.2 Externe Entities

Ein Entity, das nicht intern ist, ist ein **externes Entity**, das folgendermaßen deklariert wird:

Deklaration von Externen Entities	
[75] ExternalID ::= 'SYSTEM' S SystemLiteral | 'PUBLIC' S PubidLiteral S SystemLiteral	
[76] NDataDecl ::= S 'NDATA' S Name	[GKB: Deklarierte Notation]

Wenn NDataDecl vorhanden ist, handelt es sich um ein allgemeines nicht-analysiertes Entity, sonst ist es ein analysiertes Entity.

Gültigkeitsbeschränkung: Deklarierte Notation
Der Name muß mit dem deklarierten Namen einer Notation übereinstimmen.

Das SystemLiteral wird der **System-Identifier** des Entity genannt. Es handelt sich um einen URI, der dazu benutzt werden kann, das Entity aufzufinden. Beachten Sie, daß das Hash-Zeichen (#) und das bei URIs häufig benutzte Identifier-Fragment kein formaler Bestandteil des URIs selbst sind. Ein XML-Prozessor kann einen Fehler anzeigen, wenn ein Identifier-Fragment als Teil eines System-Identifier angegeben wird. Soweit keine Informationen, die außerhalb des Rahmens dieser Spezifikation liegen, etwas anders aussagen (z.B. ein besonderes XML-Element, das von einer bestimmten DTD definiert wird, oder eine Processing Instruction, die durch eine bestimmte Anwendungsspezifikation definiert wird), beziehen sich relative URIs auf die Adresse der Quelle, in der die Entity-Deklaration steht. Ein URI kann damit relativ zum Dokument-Entity, zum Entity, das die externe DTD-Teilmenge enthält, oder zu irgendeinem anderen externen Parameter-Entity sein.

Ein XML-Prozessor sollte ein Nicht-ASCII-Zeichen in einem URI in folgender Weise behandeln:

1. Darstellung des Zeichens in UTF-8 als ein oder mehrere Bytes
2. Anschließendes Schützen (escape) dieser Bytes mit dem entsprechenden URI-Verfahren (d.h. Umwandeln eines Bytes in %HH, wobei HH die hexadezimale Notation des Byte-Wertes ist)

Zusätzlich zu einen System-Identifier darf ein externer Identifier auch einen **Public-Identifier** enthalten. Ein XML-Prozessor, der versucht, den Inhalt des Entity zu laden, kann den Public-Identifier verwenden, um einen alternativen URI zu erzeugen. Falls der Prozessor dazu nicht in der Lage ist, muß er den als System Identifier angegebenen URI verwenden. Vor der Abbildung des Public-Identifier auf einen System-Identifier müssen alle Folgen von Leerraum (White Space) auf ein einzelnes Leerzeichen (#x20) normalisiert und führende und abschließende Leerzeichen entfernt werden.

Beispiele für Deklarationen von externen Entities:

```
<!ENTITY open-hatch
         SYSTEM "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY open-hatch
         PUBLIC "-//Textuality//TEXT Standard open-hatch boilerplate//EN"
         "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY hatch-pic
         SYSTEM "../grafix/OpenHatch.gif"
         NDATA gif >
```

4.3 Analysierte Entities

4.3.1 Die Text-Deklaration

Extern-analysierte Entities können mit einer **Text-Deklaration** beginnen.

Text-Deklaration
[77] TextDecl ::= '<?xml' VersionInfo? EncodingDecl S? '?>'

Die Text-Deklaration muß literal angegeben werden, nicht als Referenz auf ein analysiertes Entity. An keiner Stelle außer am Anfang eines externen analysierten Entity darf eine Text-Deklaration vorkommen.

4.3.2 Wohlgeformte, analysierte Entities

Das Dokument-Entity ist wohlgeformt, wenn es zur Produktion document paßt. Ein externes, allgemeines, analysiertes Entity ist wohlgeformt, wenn es zur Produktion extParsedEnt paßt. Ein externes Parameter-Entity ist wohlgeformt, wenn es zur Produktion extPE paßt.

Wohlgeformte, extern-analysierte Entities
[78] extParsedEnt ::= TextDecl? content [79] extPE ::= TextDecl? extSubsetDecl

Ein internes, allgemeines, analysiertes Entity ist wohlgeformt, wenn sein Ersetzungstext zur Produktion content paßt. Alle internen Parameter-Entities sind per definitionem wohlgeformt.

Eine Konsequenz der Wohlgeformtheit von Entities ist, daß die logische und physikalische Struktur eines XML-Dokuments korrekt verschachtelt ist. Kein Start-Tag, End-Tag, Leeres-Element-Tag, Element, Kommentar, Processing Instruction, Zeichen- oder Entityreferenz kann in einem Entity beginnen und in einem anderen enden.

4.3.3 Zeichenkodierung in Entities

Jedes extern-analysierte Entity in einem XML-Dokument kann eine andere Zeichenkodierung verwenden. Alle XML-Prozessoren müssen in der Lage sein, Entities in UTF-8 oder UTF-16 zu lesen.

Entities in UTF-16-Kodierung müssen mit einer Byte-Order-Markierung beginnen, die in ISO/IEC 10646, Annex E, und in Unicode Appendix B (das ZERO WIDTH NO-BREAK SPACE-Zeichen, #xFEFF) beschrieben ist. Dies ist eine Kodierungssignatur, die weder Teil des Markup noch der Zeichendaten des XML-Dokumentes ist. XML-Prozessoren müssen in der Lage sein, dieses Zeichen zu verwenden, um zwischen UTF-8 und UTF-16 zu unterscheiden.

Obwohl ein XML-Prozessor lediglich Entities in den Kodierungen UTF-8 und UTF-16 lesen muß, ist es offensichtlich, daß andere Kodierungen überall in der Welt benutzt werden. Deshalb kann es wünschenswert sein, daß ein XML-Prozessor solche Entities lesen und benutzen kann. Analysierte Entities, die in einer anderen Kodierung als UTF-8 und UTF-16 gespeichert sind, müssen mit einer Text-Deklaration beginnen, die eine Kodierungsdeklaration enthält.

Kodierungsdeklaration
[80] EncodingDecl ::= S 'encoding' Eq ('"' EncName '"' | "'" EncName "'") [81] EncName ::= [A-Za-z] ([A-Za-z0-9._]| '-')* /* Kodierungsnamen enthalten ausschließlich lateinische Buchstaben */

Im Dokument-Entity ist die Kodierungsdeklaration ein Teil der XML-Deklaration. Der EncName ist der Name der verwendeten Kodierung.

In einer Kodierungsdeklaration sollten die Werte »UTF-8«, »UTF-16«, »ISO-10646-UCS-2« und »ISO-10646-UCS-4« für die verschiedenen Kodierungen und Transformationen von Unicode und ISO/IEC 10646 benutzt werden. Die Werte »ISO-8859-1«, »ISO-8859-2«, ... »ISO-8859-9« sollten für die Teile von ISO 8859 benutzt werden. Die Werte »ISO-2022-JP«, »Shift_JIS« und »EUC-JP« sollten für die verschiedenen Kodierungsformen von JIS X-0208-1997 benutzt werden. XML-Prozessoren dürfen andere Kodierungen erkennen. Es wird empfohlen, daß andere als die genannten Zeichenkodierungen, die bei der Internet Assigned Numbers Authority [IANA] (als *Zeichensatz*, *charsets*)

registriert sind, mit ihrem registrierten Namen genannt werden. Beachten Sie, daß diese registrierten Namen als case-insensitive definiert sind. Prozessoren sollten einen Vergleich also auch case-insensitive durchführen.

Bei Abwesenheit von Informationen, die durch ein externes Transportprotokoll (z.B. HTTP oder MIME-Typ) geliefert werden, ist es ein Fehler, wenn ein Entity, welches eine Kodierungsdeklaration enthält, in einer anderen Kodierung an den XML-Prozessor übergeben wird. Ebenso ist es ein Fehler, wenn eine Kodierungsdeklaration an einer anderen Stelle als dem Anfang des externen Entity steht oder auch, wenn ein Entity, das weder mit einer Byte-Order-Markierung noch mit einer Kodierungsdeklaration beginnt, eine andere Kodierung als UTF-8 benutzt. Beachten Sie, daß wegen der Tatsache, daß ASCII eine Teilmenge von UTF-8 ist, ASCII-Entities nicht unbedingt eine Kodierungsdeklaration brauchen.

Es ist ein kritischer Fehler, wenn ein XML-Prozessor ein Entity in einer Kodierung bekommt, die er nicht verarbeiten kann.

Beispiel für Kodierungsdeklarationen:

```
<?xml encoding='UTF-8'?>
<?xml encoding='EUC-JP'?>
```

4.4 Behandlung von Entities und Referenzen durch einen XML-Prozessor

Die folgende Tabelle faßt zusammen, in welchem Kontext Zeichenreferenzen, Entity-Referenzen und der Aufruf von nicht-analysierten Entities stehen dürfen und wie sich ein XML-Prozessor in jedem Fall zu verhalten hat. Die Einträge in der linken Spalte beschreiben den Kontext:

Referenz im Inhalt

ist eine Referenz zwischen Start-Tag und End-Tag eines Elements. Korrespondiert mit dem Nicht-Terminal content.

Referenz im Attribut-Wert

ist eine Referenz entweder im Wert eines Attributes (im Start-Tag) oder ein Vorgabewert in einer Attributdeklaration. Korrespondiert mit dem Nicht-Terminal AttValue.

Auftreten als Attribut-Wert

als ein Name, nicht als Referenz, entweder als Wert eines Attributs, das als Typ ENTITY deklariert wurde, oder als ein Wert einer Liste von Token (durch Leerzeichen getrennt) in einem Attribut des Typs ENTITIES.

Referenz im Entity-Wert

ist eine Referenz innerhalb des literalen Entity-Werts in der Deklaration eines Parameter- oder internen Entity. Korrespondiert mit dem Nicht-Terminal EntityValue.

Referenz in der DTD

ist eine Referenz in der internen oder externen Teilmenge der DTD, aber außerhalb eines EntityValue oder AttValue.

	Entity-Typ				Zeichen
	Parameter	**intern, allgemein**	**extern-analysiert, allgemein**	**nicht-analysiert**	
Referenz im Inhalt	Nicht erkannt	Inkludiert	Inkludiert, falls validierend	Verboten	Inkludiert
Referenz im Attribut-Wert	Nicht erkannt	in Literal inkludiert	Verboten	Verboten	Inkludiert
Auftreten als Attribut-Wert	Nicht erkannt	Verboten	Verboten	Informieren	Nicht erkannt
Referenz im Entity-Wert	in Literal inkludiert	Durchgereicht	Durchgereicht	Verboten	Inkludiert
Referenz in der DTD	Als PE inkludiert	Verboten	Verboten	Verboten	Verboten

4.4.1 Nicht erkannt

Außerhalb der DTD hat das Prozent-Zeichen (%) keine besondere Bedeutung. Folglich wird das, was in der DTD ein Parameter-Entity wäre, im Inhalt nicht als Markup erkannt. Ebenso werden die Namen von nicht-analysierten Entities nicht erkannt, es sei denn, sie erscheinen als Wert eines entsprechend deklarierten Attributs.

4.4.2 Inkludiert

Ein Entity wird inkludiert, wenn sein Ersetzungstext an der Stelle der Referenz selbst geladen und verarbeitet wird, so als ob es Teil des Dokumentes wäre und zwar an der Stelle, an der die Referenz steht. Der Ersetzungstext kann sowohl Zeichendaten als auch Markup enthalten (mit Ausnahme der Parameter-Entities), die in der üblichen Weise behandelt werden, abgesehen davon, daß Ersetzungstext, der dazu dient, Markup-Zeichen zu schützen (die Entities amp, lt, gt, apos, quot), stets als Zeichendaten behandelt wird. (Die Zeichenkette »AT&T;« expandiert zu »AT&T;« und das übriggebliebene et-Zeichen wird nicht als Begrenzung einer Entity-Referenz angesehen.) Eine Zeichenreferenz wird inkludiert, wenn das referenzierte Zeichen an Stelle der Referenz verarbeitet wird.

4.4.3 Inkludiert, falls validierend

Wenn der XML-Prozessor bei dem Versuch, ein Dokument zu validieren, auf eine Referenz zu einem analysierten Entity stößt, dann muß der Prozessor dessen Ersetzungstext inkludieren. Wenn es sich um ein externes Entity handelt und der Prozessor gar nicht versucht, das XML-Dokument zu validieren, ist es dem Prozessor freigestellt, den Ersetzungtext zu inkludieren. Wenn ein nicht-validierender Parser einen Ersetzungtext nicht inkludiert, muß er die Anwendung darüber informieren, daß er auf ein Entity gestoßen ist, dieses aber nicht eingelesen hat.

Diese Regel basiert auf der Erkenntnis, daß die automatische Inkludierung, die durch den Entity-Mechanismus von SGML und XML zur Verfügung steht und dazu gedacht war, Modularität bei der Texterstellung zu ermöglichen, nicht unbedingt für andere Anwendungen geeignet ist, insbesondere für das Dokument-Browsing. Beispielsweise könnten Browser sich dafür entscheiden, eine Referenz auf ein extern-analysiertes Entity visuell anzuzeigen und das Entity erst auf Anfrage zu laden und darzustellen.

4.4.4 Verboten

Das folgende ist verboten und stellt einen kritischen Fehler dar:

- das Erscheinen einer Referenz auf ein nicht-analysiertes Entity
- das Erscheinen einer Zeichen- oder allgemeinen Entity-Referenz in der DTD, es sei denn sie steht innerhalb eines EntityValue oder AttValue
- eine Referenz auf ein externes Entity in einem Attribut-Wert

4.4.5 In Literal inkludiert

Wenn eine Entity-Referenz in einem Attribut-Wert erscheint oder wenn eine Parameter-Entity-Referenz in einem literalen Entity-Wert erscheint, wird dessen Ersetzungstext an Stelle der Referenz selbst verarbeitet; so, als ob er Teil des Dokuments an der Stelle wäre, an der die Referenz steht. Eine Ausnahme ist, daß ein einfaches (Apostroph) oder doppeltes Anführungszeichen im Ersetzungstext stets als normales Zeichen behandelt wird und das Literal nicht beendet. Zum Beispiel ist folgendes wohlgeformt:

```
<!ENTITY % JN '"Ja"' >
<!ENTITY WasErSagte "Er sagte &JN;" >
```

Dieses jedoch nicht:

```
<!ENTITY EndAttr "27'" >
<element attribute='a-&EndAttr;>
```

4.4.6 Informieren

Wenn der Name eines nicht-analysierten Entity als ein Token im Wert eines Attributs erscheint, dessen deklarierter Typ ENTITY oder ENTITIES ist, dann muß ein validierender Prozessor die Anwendung über den System- und Public-Identifier (falls vorhanden) des Entity und dessen Notation informieren.

4.4.7 Durchgereicht

Wenn eine Referenz auf ein allgemeines Entity im EntityValue in einer Entity-Deklaration erscheint, wird es unverändert durchgereicht.

4.4.8 Als PE inkludiert

Genau wie extern-analysierte Entities müssen auch Parameter-Entities nur dann inkludiert werden, wenn das Dokument validiert wird. Wenn eine Parameter-Entity-Referenz in der DTD erkannt und inkludiert wird, wird dessen Ersetzungstext um ein führendes und abschließendes Leerzeichen (#x20) erweitert. Die Absicht ist es, den Ersetzungstext so zu beschränken, daß er in sich geschlossene grammatikalische Token der DTD enthält.

4.5 Konstruktion des Ersetzungstextes von internen Entities

Bei der Diskussion der Behandlung von internen Entities ist es nützlich, zwei Formen von Entity-Werten zu unterscheiden:

1. Der **literale Entity-Wert** ist die in Anführungszeichen eingeschlossene Zeichenkette in der Entity-Deklaration, korrespondierend zu dem Nicht-Terminal EntityValue.
2. Der **Ersetzungstext** ist der Inhalt des Entity nach Ersetzen von Zeichen- und Parameter-Entity-Referenzen.

Der literale Entity-Wert, wie er in einer internen Entity-Deklaration (EntityValue) angegeben ist, darf Zeichen-, Parameter-Entity- und allgemeine Entity-Referenzen enthalten. Solche Referenzen müssen vollständig innerhalb des literalen Entity-Werts enthalten sein. Der tatsächliche Ersetzungstext, der wie oben beschrieben inkludiert wird, muß den Ersetzungstext von jedem Parameter-Entity, das referenziert wird, enthalten. Im Falle von Zeichenreferenzen muß er das referenzierte Zeichen im literalen Entity-Wert enthalten. Allgemeine Entity-Referenzen müssen jedoch bleiben wie sie sind, nicht expandiert. Gegeben sei beispielsweise folgende Deklaration:

```
<!ENTITY % pub    "&#xc9;ditions Gallimard" >
<!ENTITY   rights "All rights reserved" >
<!ENTITY   book   "La Peste: Albert Camus,
&#xA9; 1947 %pub;. &rights;" >
```

Dann ist der Ersetzungstext für das Entity »book« folgendes:

```
La Peste: Albert Camus,
© 1947 Éditions Gallimard. &rights;
```

Die allgemeine Entity-Referenz »&rights;« würde expandiert, wenn die Referenz »&book;« im Inhalt des Dokuments oder in einem Attributwert stehen würde.

Diese einfachen Regeln können komplexe Wechselwirkungen haben. Für eine detaillierte Diskussion komplizierterer Beispiele siehe 10.

4.6 Vordefinierte Entities

Entity- und Zeichenreferenzen können beide benutzt werden, um die öffnende spitze Klammer, das et-Zeichen und andere Begrenzungen zu **schützen**. Zu diesem Zweck ist eine Menge von allgemeinen Entities (amp, lt, gt, apos, quot) spezifiert worden. Außerdem können numerische Zeichenreferenzen verwendet werden. Diese werden unmittelbar expandiert, sobald sie erkannt werden, und müssen als Zeichendaten behandelt werden. So können die numerischen Zeichenreferenzen »<« und »&« verwendet werden, um die Zeichen < und & innerhalb von Zeichendaten zu schützen.

Alle XML-Prozessoren müssen diese Entities erkennen, unabhängig davon, ob sie deklariert sind oder nicht. Zwecks Zusammenarbeit sollten gültige XML-Dokumente diese Entities vor der Benutzung wie andere deklarieren. Wenn die fraglichen Entities deklariert sind, müssen sie als interne Entities deklariert werden, deren Ersetzungstext das einzelne zu schützende Zeichen oder eine Zeichenreferenz darauf ist, wie unten gezeigt.

```
<!ENTITY lt     "&#38;#60;">
<!ENTITY gt     "&#62;">
<!ENTITY amp    "&#38;#38;">
```

```
<!ENTITY apos   "&#39;">
<!ENTITY quot   "&#34;">
```

Beachten Sie, daß die Zeichen < und & in der Deklaration von »lt« und »amp« doppelt geschützt sind, um die Anforderung zu erfüllen, daß Entity-Ersetzungen wohlgeformt sind.

4.7 Notation-Deklarationen

Notationen identifizieren durch einen Namen das Format von nicht-analysierten Entities, das Format von Elementen, die ein Notation-Attribut tragen oder die Anwendung, an die sich eine Processing Instruction richtet.

Notation-Deklarationen geben einer Notation einen Namen für die Verwendung in Entity- und Attributlisten-Deklarationen und in Attribut-Spezifikationen sowie einen externen Identifier für die Notation, der es dem XML-Prozessor oder seinem Anwendungsprogramm erlaubt, ein Hilfsprogramm zu finden, das in der Lage ist, Daten in der gegebenen Notation zu verarbeiten.

Notation-Deklarationen
[82] NotationDecl ::= '<!NOTATION' S Name S (ExternalID | PublicID) S? '>'
[83] PublicID ::= 'PUBLIC' S PubidLiteral

XML-Prozessoren müssen Anwendungen mit dem Namen und dem/den externen Identifier(n) jeder Notation versorgen, die deklariert werden und auf die in einem Attribut-Wert, einer Attributdefinition oder einer Entity-Deklaration verwiesen wird. Sie dürfen außerdem den externen Identifier zu einem System-Identifier, einem Dateinamen oder einer anderen benötigten Information auflösen, die es der Anwendung erlaubt, einen Prozessor für die Daten in der beschriebenen Notation aufzurufen. (Es ist kein Fehler, wenn ein XML-Dokument Notationen deklariert und referenziert, für die keine notationsspezifischen Anwendungen auf dem System verfügbar sind, auf dem der XML-Prozessor oder dessen Anwendung läuft.)

4.8 Dokument-Entity

Das **Dokument-Entity** dient als Wurzel des Entity-Baumes und als Startpunkt für einen XML-Prozessor. Diese Spezifikation gibt nicht an, wie das Dokument-Entity von einem XML-Prozessor gefunden wird. Anders als andere Entities hat das Dokument-Entity keinen Namen und kann im Eingabestrom des Prozessors ganz ohne Identifikation erscheinen.

5 Konformität

5.1 Validierende und nicht-validierende Prozessoren

Konforme XML-Prozessoren fallen in zwei Kategorien: validierend und nicht-validierend.

Sowohl validierende als auch nicht-validierende Prozessoren müssen Verletzungen der in dieser Spezifikation genannten Wohlgeformtheitsbeschränkung, die im Inhalt des Dokument-Entity und jedes anderen analysierten Entity das sie lesen, melden.

Validierende Prozessoren müssen Verletzungen der Beschränkungen, die durch die Deklarationen in der DTD formuliert werden, sowie jedes Nicht-Erfüllen der in dieser Spezifikation formulierten Gültigkeitsbeschränkung melden. Um dies zu erreichen, müssen validierende XML-Prozessoren die gesamte DTD und alle extern-analysierten Entities, die im Dokument referenziert werden, einlesen und verarbeiten.

Nicht-validierende Prozessoren müssen lediglich das Dokument-Entity, einschließlich der gesamten internen DTD-Teilmenge auf Wohlgeformtheit prüfen. Während sie das Dokument nicht auf Gültigkeit prüfen müssen, müssen sie alle Deklarationen, die sie in der internen DTD-Teilmenge lesen, und alle Parameter-Entities, die sie lesen, verarbeiten, bis zur ersten Referenz auf ein Parameter-Entity, das sie *nicht* lesen. Das heißt, sie müssen die Information in solchen Deklarationen nutzen, um Attribut-Werte zu normalisieren, sie müssen den Ersetzungstext von internen Entities einfügen, und sie müssen Vorgabewerte (defaults) liefern. Sie dürfen keine Entity-Deklarationen oder Attributlisten-Deklarationen verarbeiten, die sich hinter einer Referenz auf ein nicht gelesenes Parameter-Entity befinden, da das Entity überschreibende Deklarationen enthalten könnte.

5.2 Benutzen von XML-Prozessoren

Das Verhalten von validierenden XML-Prozessoren ist hochgradig vorhersagbar; er muß jeden Teil eines Dokuments lesen und alle Verletzungen von Wohlgeformtheit und Gültigkeit melden. Geringer sind die Ansprüche an einen nicht validierenden Prozessor; er muß keinen anderen Teil als das Dokument-Entity lesen. Dies hat zwei Effekte, die für den Benutzer eines XML-Prozessors wichtig sein könnten:

- Bestimmte Wohlgeformtheitsfehler, insbesondere solche, die das Lesen von externen Entities erfordern, werden von einem nicht-validierenden XML-Prozessor möglicherweise nicht entdeckt. Beispiele sind sowohl in den Beschränkungen mit den Titeln »Entity deklariert«, »Analysiertes Entity« und »Keine Rekursion« zu finden als auch in einigen der als verboten beschriebenen Fälle im Abschnitt 4.4.
- Die Information, die vom Prozessor an die Anwendung gereicht wird, kann variieren, abhängig davon, ob der Prozessor Parameter- und externe Entities liest. Zum Beispiel darf es ein nicht-validierender Prozessor unterlassen, Attributwerte zu normalisieren, Ersetzungstext von externen Entities einzufügen oder Vorgabewerte für Attribute zu liefern. In welchen Fällen er es macht, hängt davon ab, ob er Deklarationen in externen oder Parameter-Entities gelesen hat.

Für die maximale Verläßlichkeit bei der Zusammenarbeit mit verschiedenen XML-Prozessoren sollten sich Anwendungen, die nicht-validierende Prozessoren benutzen, auf kein Verhalten verlassen, das solche Prozessoren nicht zeigen müssen. Anwendungen, die gewisse Dinge benötigen, wie die Verwendung von Vorgabewerten oder interne Entities, die in externen Entities deklariert sind, sollten validierende XML-Prozessoren benutzen.

6 Notation

Die formale Grammatik von XML ist in dieser Spezifikation unter Verwendung einer einfachen **Erweiterten Backus-Naur-Form** (**EBNF**) notiert. Jede Regel der Gramamtik definiert ein Symbol in der folgenden Form:

```
Symbol ::= Ausdruck
```

Symbole beginnen mit einem großen Anfangsbuchstaben, falls sie durch einen regulären Ausdruck definiert sind, sonst mit einem kleinen Anfangsbuchstaben. Literale Zeichenketten sind in Anführungszeichen eingeschlossen.

Innerhalb des Ausdrucks auf der rechten Seite der Regel werden die folgenden Ausdrücke verwendet, um Muster von einem oder mehr Zeichen anzugeben:

#xN

wobei N eine ganze Hexadezimalzahl ist. Dieser Ausdruck paßt zu dem Zeichen aus ISO/IEC 10646, dessen kanonischer (UCS-4-) Code bei Interpretation als vorzeichenlose (unsigned) Binärzahl den Wert N hat. Die Anzahl der führenden Nullen in #xN ist unwichtig. Die Anzahl der führenden Nullen im korrespondierenden Code ist durch die Zeichenkodierung vorgegeben und für XML unerheblich.

[a-zA-Z], [#xN-#xN]

paßt zu jedem Zeichen mit einem Code innerhalb und inklusive des/der angegebenen Intervalle(s).

[^a-z], [^#xN-#xN]

paßt zu jedem Zeichen außerhalb des Intervalls.

[^abc], [^#xN#xN#xN]

paßt zu jedem Zeichen, das nicht zu den genannten gehört.

"zeichenkette"

paßt zu der literalen Zeichenkette, die innerhalb der doppelten Anführungszeichen angegeben ist.

'zeichenkette'

paßt zu der literalen Zeichenkette, die innerhalb der einfachen Anführungszeichen (Apostroph) angegeben ist.

Diese Symbole können auf folgende Weise kombiniert werden, um komplexere Muster zu bilden. A und B stellen jeweils einfach Ausdrücke dar.

(ausdruck)

ausdruck wird als Einheit betrachtet und kann, wie in dieser Liste beschrieben, kombiniert werden.

A?

paßt zu A oder nichts; optionales A.

A B

paßt zu A, gefolgt von B.

A | B

paßt zu A oder B, aber nicht zu beidem.

A - B

paßt zu jeder Zeichenkette, die zu A paßt, aber nicht zu B.

A+

paßt zu einfachem oder mehrfachem Vorkommen von A.

A*

paßt zu null-, ein- oder mehrfachem Vorkommen von A.

Weitere in den Produktionen benutzte Notationen sind:

/* ... */

Kommentar

[WGB: ...]

Wohlgeformtheitsbeschränkung. Diese identifiziert durch einen Namen eine mit einer Produktion verknüpfte Beschränkung eines wohlgeformten Dokuments.

[GKB: ...]

Gültigkeitsbeschränkung. Diese identifiziert durch einen Namen eine mit einer Produktion verknüpfte Beschränkung eines gültigen Dokuments.

7 Anhang A: Referenzen

7.1 Normative Referenzen

IANA

(Internet Assigned Numbers Authority) Official Names for Character Sets, ed. Keld Simonsen et al. Siehe ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

IETF RFC 1766

IETF (Internet Engineering Task Force). RFC 1766: Tags for the Identification of Languages, ed. H. Alvestrand. 1995.

ISO 639

(International Organization for Standardization). ISO 639:1988 (E). Code for the representation of names of languages. [Geneva]: International Organization for Standardization, 1988.

ISO 3166

(International Organization for Standardization). ISO 3166-1:1997 (E). Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes [Geneva]: International Organization for Standardization, 1997.

ISO/IEC 10646

ISO (International Organization for Standardization). ISO/IEC 10646-1993 (E). Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane. [Geneva]: International Organization for Standardization, 1993 (plus amendments AM 1 through AM 7).

Unicode

The Unicode Consortium. The Unicode Standard, Version 2.0. Reading, Mass.: Addison-Wesley Developers Press, 1996.

7.2 Weitere Referenzen

Aho/Ullman

Aho, Alfred V., Ravi Sethi und Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Reading: Addison-Wesley, 1986, rpt. corr. 1988.

Berners-Lee et al.

Berners-Lee, T., R. Fielding und L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax and Semantics. 1997. (Work in progress; see updates to RFC 1738.)

Brüggemann-Klein

Brüggemann-Klein, Anne. Regular Expressions into Finite Automata. Extended abstract in I. Simon, Hrsg., LATIN 1992, S. 97-98. Springer-Verlag, Berlin 1992. Full Version in Theoretical Computer Science 120: 197-213, 1993.

Brüggemann-Klein und Wood

Brüggemann-Klein, Anne, und Derick Wood. Deterministic Regular Languages. Universität Freiburg, Institut für Informatik, Bericht 38, Oktober 1991.

Clark

James Clark. Comparison of SGML and XML. Siehe http://www.w3.org/TR/NOTE-sgml-xml-971215.

IETF RFC 1738

IETF (Internet Engineering Task Force). RFC 1738: Uniform Resource Locators (URL), ed. T. Berners-Lee, L. Masinter, M. McCahill. 1994.

IETF RFC 1808

IETF (Internet Engineering Task Force).RFC 1808: Relative Uniform Resource Locators, ed. R. Fielding. 1995.

IETF RFC 2141

IETF (Internet Engineering Task Force). RFC 2141: URN Syntax, ed. R. Moats. 1997.

ISO 8879

ISO (International Organization for Standardization). ISO 8879:1986(E). Information processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML). First edition -- 1986-10-15. [Geneva]: International Organization for Standardization, 1986.

ISO/IEC 10744

ISO (International Organization for Standardization). ISO/IEC 10744-1992 (E). Information technology -- Hypermedia/Time-based Structuring Language (HyTime). [Geneva]: International Organization for Standardization, 1992. Extended Facilities Annexe. [Geneva]: International Organization for Standardization, 1996.

8 Anhang B: Zeichenklassen

Den im Unicode-Standard definierten Charakteristika zufolge sind Zeichen klassifiziert als **Grundzeichen** (base characters; unter anderem gehören dazu die alphabetischen Zeichen des lateinischen Alphabets ohne diakritische Zeichen), **ideographische Zeichen** sowie kombinierte Zeichen (unter anderem gehören dazu diakritische Zeichen). Zusammen bilden diese Klassen die Klasse der **Buchstaben** (letters). Außerdem werden **Ziffern** (digits) und **Erweiterungen** (extenders) unterschieden.

[84] Letter ::= BaseChar | Ideographic

[85] BaseChar ::= [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6] | [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131] | [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E] | [#x0180-#x01C3] | [#x01CD-#x01F0] | [#x01F4-#x01F5] | [#x01FA-#x0217] | [#x0250-#x02A8] | [#x02BB-#x02C1] | #x0386 | [#x0388-#x038A] | #x038C | [#x038E-#x03A1] | [#x03A3-#x03CE] | [#x03D0-#x03D6] | #x03DA | #x03DC | #x03DE | #x03E0 | [#x03E2-#x03F3] | [#x0401-#x040C] | [#x040E-#x044F] | [#x0451-#x045C] | [#x045E-#x0481] | [#x0490-#x04C4] | [#x04C7-#x04C8] | [#x04CB-#x04CC] | [#x04D0-#x04EB] | [#x04EE-#x04F5] | [#x04F8-#x04F9] | [#x0531-#x0556] | #x0559 | [#x0561-#x0586] | [#x05D0-#x05EA] | [#x05F0-#x05F2] | [#x0621-#x063A] | [#x0641-#x064A] | [#x0671-#x06B7] | [#x06BA-#x06BE] | [#x06C0-#x06CE] | [#x06D0-#x06D3] | #x06D5 | [#x06E5-#x06E6] | [#x0905-#x0939] | #x093D | [#x0958-#x0961] | [#x0985-#x098C] | [#x098F-#x0990] | [#x0993-#x09A8] | [#x09AA-#x09B0] | #x09B2 | [#x09B6-#x09B9] | [#x09DC-#x09DD] | [#x09DF-#x09E1] | [#x09F0-#x09F1] | [#x0A05-#x0A0A] | [#x0A0F-#x0A10] | [#x0A13-#x0A28] | [#x0A2A-#x0A30] | [#x0A32-#x0A33] | [#x0A35-#x0A36] | [#x0A38-#x0A39] | [#x0A59-#x0A5C] | #x0A5E | [#x0A72-#x0A74] | [#x0A85-#x0A8B] | #x0A8D | [#x0A8F-#x0A91] | [#x0A93-#x0AA8] | [#x0AAA-#x0AB0] | [#x0AB2-#x0AB3] | [#x0AB5-#x0AB9] | #x0ABD | #x0AE0 | [#x0B05-#x0B0C] | [#x0B0F-#x0B10] | [#x0B13-#x0B28] | [#x0B2A-#x0B30] | [#x0B32-#x0B33] | [#x0B36-#x0B39] | #x0B3D | [#x0B5C-#x0B5D] | [#x0B5F-#x0B61] | [#x0B85-#x0B8A] | [#x0B8E-#x0B90] | [#x0B92-#x0B95] | [#x0B99-#x0B9A] | #x0B9C | [#x0B9E-#x0B9F] | [#x0BA3-#x0BA4] | [#x0BA8-#x0BAA] | [#x0BAE-#x0BB5] | [#x0BB7-#x0BB9] | [#x0C05-#x0C0C] | [#x0C0E-#x0C10] | [#x0C12-#x0C28] | [#x0C2A-#x0C33] | [#x0C35-#x0C39] | [#x0C60-#x0C61] | [#x0C85-#x0C8C] | [#x0C8E-#x0C90] | [#x0C92-#x0CA8] | [#x0CAA-#x0CB3] | [#x0CB5-#x0CB9] | #x0CDE | [#x0CE0-#x0CE1] | [#x0D05-#x0D0C] | [#x0D0E-#x0D10] | [#x0D12-#x0D28] | [#x0D2A-#x0D39] | [#x0D60-#x0D61] | [#x0E01-#x0E2E] | #x0E30 | [#x0E32-#x0E33] | [#x0E40-#x0E45] | [#x0E81-#x0E82] | #x0E84 | [#x0E87-#x0E88] | #x0E8A | #x0E8D | [#x0E94-#x0E97] | [#x0E99-#x0E9F] | [#x0EA1-#x0EA3] | #x0EA5 | #x0EA7 | [#x0EAA-#x0EAB] | [#x0EAD-#x0EAE] | #x0EB0 | [#x0EB2-#x0EB3] | #x0EBD | [#x0EC0-#x0EC4] | [#x0F40-#x0F47] | [#x0F49-#x0F69] | [#x10A0-#x10C5] | [#x10D0-#x10F6] | #x1100 | [#x1102-#x1103] | [#x1105-#x1107] | #x1109 | [#x110B-#x110C] | [#x110E-#x1112] | #x113C | #x113E | #x1140 | #x114C | #x114E | #x1150 | [#x1154-#x1155] | #x1159 | [#x115F-#x1161] | #x1163 | #x1165 | #x1167 | #x1169 | [#x116D-#x116E] | [#x1172-#x1173] | #x1175 | #x119E | #x11A8 | #x11AB | [#x11AE-#x11AF] | [#x11B7-#x11B8] | #x11BA | [#x11BC-#x11C2] | #x11EB | #x11F0 | #x11F9 | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9] | [#x1F00-#x1F15] | [#x1F18-#x1F1D] | [#x1F20-#x1F45] | [#x1F48-#x1F4D] | [#x1F50-#x1F57] | #x1F59 | #x1F5B | #x1F5D | [#x1F5F-#x1F7D] | [#x1F80-#x1FB4] | [#x1FB6-#x1FBC] | #x1FBE | [#x1FC2-#x1FC4] | [#x1FC6-#x1FCC] | [#x1FD0-#x1FD3] | [#x1FD6-#x1FDB] | [#x1FE0-#x1FEC] | [#x1FF2-#x1FF4] | [#x1FF6-#x1FFC] | #x2126 | [#x212A-#x212B] | #x212E | [#x2180-#x2182] | [#x3041-#x3094] | [#x30A1-#x30FA] | [#x3105-#x312C] | [#xAC00-#xD7A3]

[86] Ideographic ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]

[87] CombiningChar	::=	[#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486] | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD] | #x05BF | [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652] | #x0670 | [#x06D6-#x06DC] | [#x06DD-#x06DF] | [#x06E0-#x06E4] | [#x06E7-#x06E8] | [#x06EA-#x06ED] | [#x0901-#x0903] | #x093C | [#x093E-#x094C] | #x094D | [#x0951-#x0954] | [#x0962-#x0963] | [#x0981-#x0983] | #x09BC | #x09BE | #x09BF | [#x09C0-#x09C4] | [#x09C7-#x09C8] | [#x09CB-#x09CD] | #x09D7 | [#x09E2-#x09E3] | #x0A02 | #x0A3C | #x0A3E | #x0A3F | [#x0A40-#x0A42] | [#x0A47-#x0A48] | [#x0A4B-#x0A4D] | [#x0A70-#x0A71] | [#x0A81-#x0A83] | #x0ABC | [#x0ABE-#x0AC5] | [#x0AC7-#x0AC9] | [#x0ACB-#x0ACD] | [#x0B01-#x0B03] | #x0B3C | [#x0B3E-#x0B43] | [#x0B47-#x0B48] | [#x0B4B-#x0B4D] | [#x0B56-#x0B57] | [#x0B82-#x0B83] | [#x0BBE-#x0BC2] | [#x0BC6-#x0BC8] | [#x0BCA-#x0BCD] | #x0BD7 | [#x0C01-#x0C03] | [#x0C3E-#x0C44] | [#x0C46-#x0C48] | [#x0C4A-#x0C4D] | [#x0C55-#x0C56] | [#x0C82-#x0C83] | [#x0CBE-#x0CC4] | [#x0CC6-#x0CC8] | [#x0CCA-#x0CCD] | [#x0CD5-#x0CD6] | [#x0D02-#x0D03] | [#x0D3E-#x0D43] | [#x0D46-#x0D48] | [#x0D4A-#x0D4D] | #x0D57 | #x0E31 | [#x0E34-#x0E3A] | [#x0E47-#x0E4E] | #x0EB1 | [#x0EB4-#x0EB9] | [#x0EBB-#x0EBC] | [#x0EC8-#x0ECD] | [#x0F18-#x0F19] | #x0F35 | #x0F37 | #x0F39 | #x0F3E | #x0F3F | [#x0F71-#x0F84] | [#x0F86-#x0F8B] | [#x0F90-#x0F95] | #x0F97 | [#x0F99-#x0FAD] | [#x0FB1-#x0FB7] | #x0FB9 | [#x20D0-#x20DC] | #x20E1 | [#x302A-#x302F] | #x3099 | #x309A
[88] Digit	::=	[#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9] | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F] | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF] | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F] | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29]
[89] Extender	::=	#x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640 | #x0E46 | #x0EC6 | #x3005 | [#x3031-#x3035] | [#x309D-#x309E] | [#x30FC-#x30FE]

Die hier definierten Klassen könnten aus der Unicode-Zeichendatenbank in folgender Weise abgeleitet werden:

- Anfangszeichen von Namen müssen in einer der Kategorien Ll, Lu, Lo, Lt, Nl sein.
- Andere Namen-Zeichen als die Anfangszeichen müssen in einer der Kategorien Mc, Me, Mn, Lm, Nd sein.
- Zeichen im Kompatibilitätsbereich (compatibility area; das sind Zeichen mit einem Code größer #xF900 als und kleiner als #xFFFE) sind in XML-Namen nicht erlaubt.
- Zeichen, die eine Zeichensatz- oder Kompatibilitätszerlegung (font or compatibility decomposition) haben (d.h. solche, die mit einem »compatibility formatting tag« in Feld 5 der Datenbank -- durch ein mit einem »<« beginnendes Feld 5 gekennzeichnet) sind nicht erlaubt.
- Die folgenden Zeichen werden als Anfangszeichen von Namen, im Gegensatz zu Namen-Zeichen, behandelt, da die Property-Datei sie als alphabetisch klassifiziert: [#x02BB-#x02C1], #x0559, #x06E5, #x06E6.
- Die Zeichen #x20DD-#x20E0 sind ausgeschlossen (in Übereinstimmung mit Unicode, Abschnitt 5.14)
- Zeichen #x00B7 ist als Erweiterung klassifiziert, da die Property-Liste es so einstuft.
- Zeichen #x0387 wurde als Namen-Zeichen hinzugefügt, da #x00B7 sein kanonisches Äquivalent ist.
- Zeichen »:« und »_« sind als Anfangszeichen für Namen erlaubt.
- Zeichen »-« und ».« sind als Namen-Zeichen erlaubt.

9 Anhang C: XML und SGML (nicht normativ)

XML wurde als Teilmenge von SGML entworfen, so daß jedes gültige XML-Dokument auch ein konformes SGML-Dokument ist. Für einen detaillierten Vergleich der weitergehenden Beschränkungen, die XML über SGML hinaus einem Dokument auferlegt, siehe [Clark].

10 Anhang D: Expansion von Entity- und Zeichenreferenzen (nicht normativ)

Dieser Anhang enthält einige Beispiele, die die Abfolge der Erkennung und Expansion von Entity- und Zeichenreferenzen, wie in 4.4 beschrieben, illustrieren.

Wenn die DTD folgende Deklaration enthält

```
<!ENTITY beispiel "<p>Ein et-Zeichen (&#38;#38;) kann
numerisch (&#38;#38;#38;) oder mit einem allgemeinen
Entity (&amp;amp;) geschützt werden.</p>" >
```

dann wird der XML-Prozessor die Zeichenreferenzen erkennen, sobald er die Entity-Deklaration analysiert, und wird sie auflösen, um schließlich folgende Zeichenkette als den Wert des Entity »beispiel« zu speichern:

```
<p>Ein et-Zeichen (&#38;) kann
numerisch (&#38;#38;) oder mit einem allgemeinen
Entity (&amp;amp;) geschützt werden.</p>
```

Eine Referenz auf »&beispiel;« im Dokument verursacht eine erneute Analyse, in der die Start- und End-Tags des »p«-Elements erkannt und die drei Referenzen erkannt und expandiert werden. Das Ergebnis ist ein »p«-Element mit folgendem Inhalt (alles Zeichendaten, keine Begrenzungen oder Markup):

```
Ein et-Zeichen (&) kann
numerisch (&#38;) oder mit einem allgemeinen
Entity (&amp;) geschützt werden.
```

Ein komplexeres Beispiel wird die Regeln und ihre Auswirkungen vollständig illustrieren. Im folgenden Beispiel dienen die Zeilennummern einzig zur Referenzierung.

```
 1 <?xml version='1.0'?>
 2 <!DOCTYPE test [
 3 <!ELEMENT test (#PCDATA) >
 4 <!ENTITY % xx '&#37;zz;'>
 5 <!ENTITY % zz '&#60;!ENTITY trickreiche "fehler-anfällig" >' >
 6 %xx;
 7 ]>
 8 <test>Dieses Beispiel zeigt eine &trickreiche; Methode.</test>
```

Dieses führt zu folgendem:

- In Zeile 4 wird die Referenz auf das Zeichen 37 sofort aufgelöst, und das Parameter-Entity »xx« wird in der Symboltabelle mit dem Wert »%zz;« abgelegt. Da der Ersetzungstext nicht noch einmal analysiert wird, wird die Referenz auf das Parameter-Entity »zz« nicht erkannt. (Und das wäre auch ein Fehler, denn »zz« ist noch nicht deklariert.)
- In Zeile 5 wird die Zeichenreferenz »<« sofort expandiert, und das Parameter-Entity »zz« wird mit dem Ersetzungstext »<!ENTITY trickreiche "fehler-anfällige" >« abgelegt, was eine wohlgeformte Entity-Deklaration ist.
- In Zeile 6 wird die Referenz auf »xx« erkannt, und der Ersetzungstext von »xx«, nämlich »%zz;«, wird analysiert. Die Referenz auf »zz« wird seinerseits erkannt und dessen Ersetzungstext (»<!ENTITY trickreiche "fehler-anfällige" >«) wird analysiert. Das allgemeine Entity »trickreiche« wurde nun mit dem Ersetzungstext »fehler-anfällige« deklariert.
- In Zeile 8 wird die Referenz auf das allgemeine Entity »trickreiche« erkannt und expandiert, so daß der volle Inhalt des Elementes »test« nun die folgende selbstbeschreibende (und grammatikalisch falsche) Zeichenkette ist: »Dieses Beispiel zeigt eine fehler-anfällig Methode.«

11 Anhang E: Deterministische Inhaltsmodelle (nicht normativ)

Zwecks Kompatibilität ist es notwendig, daß Inhaltsmodelle in Elementtyp-Deklarationen deterministisch sind.

SGML benötigt deterministische Inhaltsmodelle (dort »unzweideutig« (unambiguous) genannt). XML-Prozessoren, die mit SGML-Systemen konstruiert wurden, können nicht-deterministische Inhaltsmodelle als Fehler anzeigen.

Zum Beispiel ist das Inhaltsmodell ((b, c) | (b, d)) nicht deterministisch, da der Parser bei einem gegebenen initialen b nicht wissen kann, welches b im Inhaltsmodell dazu paßt, ohne vorauszuschauen und nachzusehen, welches Element dem b folgt. In diesem Fall können die zwei Referenzen auf b auf folgende Weise zu einer einzigen Referenz zusammengefaßt werden: (b, (c | d)). Ein einleitendes b paßt nun eindeutig zu einem Namen im Inhaltsmodell. Der Parser braucht nicht mehr vorauszuschauen, um zu sehen, was folgt. Sowohl c als auch d würden akzeptiert.

Etwas formaler: Ein endlicher Automat kann aus dem Inhaltsmodell unter Verwendung von Standardalgorithmen, etwa Algorithmus 3.5 in Abschnitt 3.9 von Aho, Sethi und Ullman [Aho/Ullman], konstruiert werden. In vielen solcher Algorithmen wird eine Folgemenge für jeden Zustand im regulären Ausdruck (d.h. jedes Blatt im Syntaxbaum des regulären Ausdrucks) konstruiert. Falls ein Zustand eine Folgemenge besitzt, in der mehr als ein Folgezustand mit dem gleichen Elementtyp-Namen markiert ist, dann ist das Inhaltsmodell fehlerhaft und kann als Fehler angezeigt werden.

Es existieren Algorithmen, die es erlauben, viele (aber nicht alle) nicht-deterministischen Inhaltsmodelle automatisch auf äquivalente deterministische Modelle zurückzuführen; siehe Brüggemann-Klein 1991 [Brüggemann-Klein].

12 Anhang F: Automatische Erkennung von Zeichenkodierungen (nicht normativ)

Die XML-Kodierungsdeklaration fungiert als eine interne Markierung in jedem Entity, die anzeigt, welche Zeichenkodierung benutzt wird. Bevor ein XML-Prozessor die interne Markierung lesen kann, muß er offensichtlich wissen, welche Zeichenkodierung benutzt wird -- was genau das ist, was die interne Markierung anzeigen will. Im allgemeinen Fall ist das eine hoffnungslose Situation. In XML ist es aber nicht völlig hoffnungslos, weil XML den allgemeinen Fall auf zwei Arten einschränkt: Es wird angenommen, daß jede Implementation nur eine endliche Menge von Zeichenkodierungen unterstützt, und die XML-Kodierungsdeklaration ist in ihrer Position und ihrem Inhalt eingeschränkt, um eine automatische Erkennung der benutzten Zeichenkodierung in jedem Entity im Normalfall zu ermöglichen. Außerdem sind in vielen Fällen zusätzlich zum XML-Datenstrom andere Informationsquellen verfügbar. Zwei Fälle können unterschieden werden, abhängig davon, ob das XML-Entity dem Prozessor ohne oder mit weiteren (externen) Informationen zur Verfügung steht. Wir nehmen zunächst den ersten Fall an.

Da jedes XML-Entity, das nicht im UTF-8- oder UTF-16-Format vorliegt, mit einer XML-Kodierungsdeklaration beginnen muß, in der die ersten Zeichen »<?xml« sind, kann jeder konforme Prozessor nach Einlesen von zwei bis vier Oktetten entscheiden, welcher der folgenden Fälle vorliegt. Beim Lesen dieser Liste kann es hilfreich sein, zu wissen, daß in UCS-4 das »<« die Kodierung »#x0000003C« und »?« die Kodierung »#x0000003F« hat und daß die Byte-Order-Markierung, die von UTF-16-Datenströmen benötigt wird, die Kodierung »#xFEFF« hat.

- 00 00 00 3C: UCS-4, Big-endian-Maschine (Reihenfolge 1234)
- 3C 00 00 00: UCS-4, Little-endian-Maschine (Reihenfolge 4321)
- 00 00 3C 00: UCS-4, ungewöhnliche Oktett-Reihenfolge (2143)
- 00 3C 00 00: UCS-4, ungewöhnliche Oktett-Reihenfolge (3412)
- FE FF: UTF-16, Big-endian
- FF FE: UTF-16, Little-endian
- 00 3C 00 3F: UTF-16, Big-endian, keine Byte-Order-Markierung (und damit -- streng genommen -- fehlerhaft)
- 3C 00 3F 00: UTF-16, Little-endian, keine Byte-Order-Markierung (und damit -- streng genommen -- fehlerhaft)
- 3C 3F 78 6D: UTF-8, ISO 646, ASCII, einige Teile von ISO 8859, Shift-JIS, EUC oder jede andere 7-Bit-, 8-Bit-, oder Kodierung mit gemischter Breite, die sicherstellt, daß die ASCII-Zeichen ihre normale Position, Breite und Werte haben. Die tatsächliche Kodierungsdeklaration muß gelesen werden, aber da jede dieser Kodierungen die gleichen Bitmuster für ASCII-Zeichen verwendet, kann die Kodierungsdeklaration selbst sicher gelesen werden.

- 4C 6F A7 94: EBCDIC (In einigen Fällen muß die vollständige Kodierungsdeklaration gelesen werden, um sagen zu können, welche Codepage verwendet wird.)
- andere: UTF-8 ohne Kodierungsdeklaration; andernfalls ist der Datenstrom fehlerhaft, unvollständig oder in irgendeinen Wrapper eingepackt.

Dieses Niveau der automatischen Erkennung ist ausreichend, um die XML-Kodierungsdeklaration zu lesen und den Identifier für die Zeichenkodierung zu analysieren, was immer noch notwendig ist, um zwischen einzelnen Mitgliedern einer Kodierungsfamilie zu unterscheiden (z.B. um UTF-8 von 8859 und die Teile von 8859 voneinander zu unterscheiden oder um die genaue EBCDIC-Codepage zu identifizieren).

Da der Inhalt der Kodierungsdeklaration auf ASCII-Zeichen beschränkt ist, kann ein Prozessor sicher die gesamte Kodierungsdeklaration lesen, sobald er erkannt hat, welche Kodierungsfamilie verwendet wird. Da praktisch alle weit verbreiteten Zeichenkodierungen in eine der obengenannten Kategorien fallen, erlaubt die XML-Kodierungsdeklaration eine hinreichend zuverlässige Selbstbeschreibung der Zeichenkodierungen, selbst wenn externe Informationsquellen auf Ebene des Betriebssystems oder Transport-Protokolls unzuverlässig sind.

Hat der Prozessor einmal die verwendete Zeichenkodierung erkannt, kann er angemessen handeln, sei es durch den Aufruf einer für jeden Fall separaten Eingaberoutine oder den Aufruf einer geeigneten Konvertierungsfunktion für jedes Eingabezeichen.

Wie jedes selbstkennzeichnende System wird auch die XML-Kodierungsdeklaration nicht funktionieren, falls irgendeine Software den Zeichensatz oder die Kodierung des Entity ändert, ohne die Kodierungsdeklaration zu aktualisieren. Programmierer von Routinen zur Zeichenkodierung sollten mit Sorgfalt die Korrektheit von internen und externen Informationen zur Kennzeichnung eines Entity sicherstellen.

Der zweite mögliche Fall tritt ein, wenn das XML-Entity durch Kodierungsinformationen ergänzt wird, wie in einigen Filesystemen und Netzwerkprotokollen. Wenn mehrere Informationsquellen verfügbar sind, sollten ihre relative Priorität und die bevorzugte Methode zur Handhabung von Konflikten als Teil des Übertragungsprotokolls, das zum Versenden von XML benutzt wird, spezifiziert werden. Regeln für die relative Priorität der internen Kennzeichnung und zum Beispiel der **MIME-Typ**-Kennzeichnung in einem externen Protokollkopf sollten Teil des RFC-Dokumentes sein, das die MIME-Typen **text/xml** und **application/xml** definiert. Im Interesse der Zusammenarbeit werden aber folgende Regeln empfohlen:

- Falls ein XML-Entity in einer Datei steht, dann werden die Byte-Order-Markierung und die Processing Instruction mit Kodierungsdeklaration verwendet (falls vorhanden), um die Zeichenkodierung zu bestimmen. Alle anderen Heuristiken und Informationsquellen dienen einzig dazu, im Fehlerfall weiterzuarbeiten.
- Falls ein XML-Entity mit dem MIME-Typ »text/xml« ausgeliefert wird, dann bestimmt der »charset«-Parameter des MIME-Typs die Zeichenkodierung. Alle anderen Heuristiken und Informationsquellen dienen einzig dazu, im Fehlerfall weiterzuarbeiten.
- Falls ein XML-Entity mit dem MIME-Typ »application/xml« ausgeliefert wird, dann werden die Byte-Order-Markierung und die Processing Instruction mit Kodierungsdeklaration verwendet (falls vorhanden), um die Zeichenkodierung zu bestimmen. Alle anderen Heuristiken und Informationsquellen dienen einzig zur Fehlerbehebung.

Diese Regeln gelten nur bei Fehlen von Dokumentation der Protokoll-Ebene. Insbesondere sobald die MIME-Typen »text/xml« und »application/xml« definiert sind, werden die Empfehlungen des relevanten RFC diese Regeln ersetzen.

13 Anhang G: XML-Arbeitsgruppe des W3C (nicht normativ)

Diese Spezifikation wurde von der W3C-XML-Arbeitsgruppe (AG) erstellt und deren Veröffentlichung gebilligt. Die Billigung der AG bedeutet nicht zwingend, daß alle AG-Mitglieder dafür gestimmt haben. Die momentanen und ehemaligen Mitglieder der XML-AG sind:

Jon Bosak, Sun (Vorsitz); James Clark (Technische Leitung); Tim Bray, Textuality und Netscape (XML-Co-Herausgeber); Jean Paoli, Microsoft (XML-Co-Herausgeber); C. M. Sperberg-McQueen, U. of Ill. (XML-Co-Herausgeber); Dan Connolly, W3C (Verbindung zum W3C); Paula Angerstein, Texcel; Steve DeRose, INSO; Dave Hollander, HP; Eliot Kimber, ISOGEN; Eve Maler, ArborText; Tom Magliery, NCSA; Murray Maloney, Muzmo and Grif; Makoto Murata, Fuji Xerox Information Systems; Joel Nava, Adobe; Conleth O'Connell, Vignette; Peter Sharpe, SoftQuad; John Tigue, DataChannel

Anmerkungen der Übersetzer

1 Wir bedanken uns bei (in alphabetischer Reihenfolge): Martin J. Dürst, Dirk Gouders, Ingo Macherius, C. M. Sperberg-McQueen, Karsten Tinnefeld.

2 Der im Englischen verwendete geschlechtsneutrale Begriff »parent« konnte nicht übersetzt werden, da im Deutschen eine Singularform von »Eltern« fehlt. Die hier benutzte Übersetzung in der männlichen Form wurde gewählt, da sie in der Informatikliteratur üblich ist.

3 Ein Token ist ein Zeichen bzw. eine Zeichenfolge, das bzw. die für einen Parser eine syntaktische Einheit darstellt.

4 Der Begriff des »Anführungszeichens« (double quotes) bezeichnet innerhalb der übersetzten Spezifikation die auf Computer-Tastaturen zu findenden hochgestellten Anführungszeichen, die nicht wie im Deutschen zwischen tiefgestellter *Anführung* (öffnendes Zeichen) und hochgestellter *Abführung* (schließendes Zeichen) unterscheiden. Gleiches gilt für das »einfache Anführungszeichen« (single quotes) im englischen Originaltext. Dies haben wir in der Übersetzung gelegentlich als »Apostroph« bezeichnet, da es sich um das gleiche Zeichen (auf der Tastatur) handelt, wenngleich es nicht die Funktion des Apostroph (Auslassung) erfüllt. Wir haben hier dem Pragmatismus Vorrang vor typographischer Exaktheit gegeben.

Extensible Markup Language (XML) 1.0 (Second Edition)

W3C Recommendation 6 October 2000

This version:

http://www.w3.org/TR/2000/REC-xml-20001006 (XHTML, XML, PDF, XHTML review version with color-coded revision indicators)

Latest version:

http://www.w3.org/TR/REC-xml

Previous versions:

http://www.w3.org/TR/2000/WD-xml-2e-20000814

http://www.w3.org/TR/1998/REC-xml-19980210

Editors:

Tim Bray, Textuality and Netscape <tbray@textuality.com>

Jean Paoli, Microsoft <jeanpa@microsoft.com>

C. M. Sperberg-McQueen, University of Illinois at Chicago and Text Encoding Initiative <cmsmcq@uic.edu>

Eve Maler, Sun Microsystems, Inc. <eve.maler@east.sun.com> - Second Edition

---

Abstract

The Extensible Markup Language (XML) is a subset of SGML that is completely described in this document. Its goal is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML. XML has been designed for ease of implementation and for interoperability with both SGML and HTML.

Status of this Document

This document has been reviewed by W3C Members and other interested parties and has been endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited as a normative reference from another document. W3C's role in making the Recommendation is to draw attention to the specification and to promote its widespread deployment. This enhances the functionality and interoperability of the Web.

This document specifies a syntax created by subsetting an existing, widely used international text processing standard (Standard Generalized Markup Language, ISO 8879:1986(E) as amended and corrected) for use on the World Wide Web. It is a product of the W3C XML Activity, details of which can be found at http://www.w3.org/XML. The English version of this specification is the only normative version. However, for translations of this document, see http://www.w3.org/XML/#trans. A list of current W3C Recommendations and other technical documents can be found at http://www.w3.org/TR.

This second edition is *not* a new version of XML (first published 10 February 1998); it merely incorporates the changes dictated by the first-edition errata (available at http://www.w3.org/XML/xml-19980210-errata) as a convenience to readers. The errata list for this second edition is available at http://www.w3.org/XML/xml-V10-2e-errata.

Please report errors in this document to xml-editor@w3.org; archives are available.

Note:

C. M. Sperberg-McQueen's affiliation has changed since the publication of the first edition. He is now at the World Wide Web Consortium, and can be contacted at cmsmcq@w3.org.

Table of Contents

Appendices

---

1 Introduction

Extensible Markup Language, abbreviated XML, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. XML is an application profile or restricted form of SGML, the Standard Generalized Markup Language [ISO 8879]. By construction, XML documents are conforming SGML documents.

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

[Definition: A software module called an **XML processor** is used to read XML documents and provide access to their content and structure.] [Definition: It is assumed that an XML processor is doing its work on behalf of another module, called the **application**.] This specification describes the required behavior of an XML processor in terms of how it must read XML data and the information it must provide to the application.

1.1 Origin and Goals

XML was developed by an XML Working Group (originally known as the SGML Editorial Review Board) formed under the auspices of the World Wide Web Consortium (W3C) in 1996. It was chaired by Jon Bosak of Sun Microsystems with the active participation of an XML Special Interest Group (previously known as the SGML Working Group) also organized by the W3C. The membership of the XML Working Group is given in an appendix. Dan Connolly served as the WG's contact with the W3C.

The design goals for XML are:

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.

This specification, together with associated standards (Unicode and ISO/IEC 10646 for characters, Internet RFC 1766 for language identification tags, ISO 639 for language name codes, and ISO 3166 for country name codes), provides all the information necessary to understand XML Version 1.0 and construct computer programs to process it.

This version of the XML specification may be distributed freely, as long as all text and legal notices remain intact.

1.2 Terminology

The terminology used to describe XML documents is defined in the body of this specification. The terms defined in the following list are used in building those definitions and in describing the actions of an XML processor:

may

[Definition: Conforming documents and XML processors are permitted to but need not behave as described.]

must

[Definition: Conforming documents and XML processors are required to behave as described; otherwise they are in error.]

error

[Definition: A violation of the rules of this specification; results are undefined. Conforming software may detect and report an error and may recover from it.]

fatal error

[Definition: An error which a conforming XML processor must detect and report to the application. After encountering a fatal error, the processor may continue processing the data to search for further errors and may report such errors to the application. In order to support correction of errors, the processor may make unprocessed data from the document (with intermingled character data and markup) available to the application. Once a fatal error is detected, however, the processor must not continue normal processing (i.e., it must not continue to pass character data and information about the document's logical structure to the application in the normal way).]

at user option

[Definition: Conforming software may or must (depending on the modal verb in the sentence) behave as described; if it does, it must provide users a means to enable or disable the behavior described.]

validity constraint

[Definition: A rule which applies to all valid XML documents. Violations of validity constraints are errors; they must, at user option, be reported by validating XML processors.]

well-formedness constraint

[Definition: A rule which applies to all well-formed XML documents. Violations of well-formedness constraints are fatal errors.]

match

[Definition: (Of strings or names:) Two strings or names being compared must be identical. Characters with multiple possible representations in ISO/IEC 10646 (e.g. characters with both precomposed and base+diacritic forms) match only if they have the same representation in both strings. No case folding is performed. (Of strings and rules in the grammar:) A string matches a grammatical production if it belongs to the language generated by that production. (Of content and content models:) An element matches its declaration when it conforms in the fashion described in the constraint **[VC: Element Valid]**.]

for compatibility

[Definition: Marks a sentence describing a feature of XML included solely to ensure that XML remains compatible with SGML.]

for interoperability

[Definition: Marks a sentence describing a non-binding recommendation included to increase the chances that XML documents can be processed by the existing installed base of SGML processors which predate the WebSGML Adaptations Annex to ISO 8879.]

2 Documents

[Definition: A data object is an **XML document** if it is well-formed, as defined in this specification. A well-formed XML document may in addition be valid if it meets certain further constraints.]

Each XML document has both a logical and a physical structure. Physically, the document is composed of units called entities. An entity may refer to other entities to cause their inclusion in the document. A document begins in a "root" or document entity. Logically, the document is composed of declarations, elements, comments, character references, and processing instructions, all of which are indicated in the document by explicit markup. The logical and physical structures must nest properly, as described in **4.3.2 Well-Formed Parsed Entities**.

2.1 Well-Formed XML Documents

[Definition: A textual object is a **well-formed** XML document if:]

1. Taken as a whole, it matches the production labeled document.
2. It meets all the well-formedness constraints given in this specification.
3. Each of the parsed entities which is referenced directly or indirectly within the document is well-formed.

Document

```
[1] document ::= prolog element Misc*
```

Matching the document production implies that:

1. It contains one or more elements.

2. [Definition: There is exactly one element, called the **root**, or document element, no part of which appears in the content of any other element.] For all other elements, if the start-tag is in the content of another element, the end-tag is in the content of the same element. More simply stated, the elements, delimited by start- and end-tags, nest properly within each other.

[Definition: As a consequence of this, for each non-root element `C` in the document, there is one other element `P` in the document such that `C` is in the content of `P`, but is not in the content of any other element that is in the content of `P`. `P` is referred to as the **parent** of `C`, and `C` as a **child** of `P`.]

2.2 Characters

[Definition: A parsed entity contains **text**, a sequence of characters, which may represent markup or character data.] [Definition: A **character** is an atomic unit of text as specified by ISO/IEC 10646 [ISO/IEC 10646] (see also [ISO/IEC 10646-2000]). Legal characters are tab, carriage return, line feed, and the legal characters of Unicode and ISO/IEC 10646. The versions of these standards cited in **A.1 Normative References** were current at the time this document was prepared. New characters may be added to these standards by amendments or new editions. Consequently, XML processors must accept any character in the range specified for Char. The use of "compatibility characters", as defined in section 6.8 of [Unicode] (see also D21 in section 3.6 of [Unicode3]), is discouraged.]

Character Range

```
[2]  Char  ::=  #x9 | #xA | #xD | [#x20-#xD7FF] |
                [#xE000-#xFFFD] | [#x10000-#x10FFFF]
```

/ any Unicode character, excluding the surrogate blocks, FFFE, and FFFF. */*

The mechanism for encoding character code points into bit patterns may vary from entity to entity. All XML processors must accept the UTF-8 and UTF-16 encodings of 10646; the mechanisms for signaling which of the two is in use, or for bringing other encodings into play, are discussed later, in **4.3.3 Character Encoding in Entities**.

2.3 Common Syntactic Constructs

This section defines some symbols used widely in the grammar.

S (white space) consists of one or more space (#x20) characters, carriage returns, line feeds, or tabs.

White Space

```
[3]  S  ::=  (#x20 | #x9 | #xD | #xA)+
```

Characters are classified for convenience as letters, digits, or other characters. A letter consists of an alphabetic or syllabic base character or an ideographic character. Full definitions of the specific characters in each class are given in **B Character Classes**.

[Definition: A **Name** is a token beginning with a letter or one of a few punctuation characters, and continuing with letters, digits, hyphens, underscores, colons, or full stops, together known as name characters.] Names beginning with the string "`xml`", or any string which would match `(('X'|'x') ('M'|'m') ('L'|'l'))`, are reserved for standardization in this or future versions of this specification.

Note:

The Namespaces in XML Recommendation [XML Names] assigns a meaning to names containing colon characters. Therefore, authors should not use the colon in XML names except for namespace purposes, but XML processors must accept the colon as a name character.

An Nmtoken (name token) is any mixture of name characters.

Names and Tokens

```
[4]  NameChar  ::=  Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar |
                    Extender
[5]  Name      ::=  (Letter | '_' | ':') (NameChar)*
[6]  Names     ::=  Name (S Name)*
[7]  Nmtoken   ::=  (NameChar)+
[8]  Nmtokens  ::=  Nmtoken (S Nmtoken)*
```

Literal data is any quoted string not containing the quotation mark used as a delimiter for that string. Literals are used for specifying the content of internal entities (EntityValue), the values of attributes (AttValue), and external identifiers (SystemLiteral). Note that a SystemLiteral can be parsed without scanning for markup.

Literals

```
[9]   EntityValue    ::=  '"' ([^%&"] | PEReference | Reference)* '"'
                        |  "'" ([^%&'] | PEReference | Reference)* "'"
[10]  AttValue       ::=  '"' ([^<&"] | Reference)* '"'
                        |  "'" ([^<&'] | Reference)* "'"
[11]  SystemLiteral  ::=  ('"' [^"]* '"') | ("'" [^']* "'")
[12]  PubidLiteral   ::=  '"' PubidChar* '"' | "'" (PubidChar - "'")* "'"
[13]  PubidChar      ::=  #x20 | #xD | #xA | [a-zA-Z0-9] | [-'()+,./:=?;!*#@$_%]
```

Note:

Although the EntityValue production allows the definition of an entity consisting of a single explicit < in the literal (e.g., `<!ENTITY mylt "<">`), it is strongly advised to avoid this practice since any reference to that entity will cause a well-formedness error.

2.4 Character Data and Markup

Text consists of intermingled character data and markup. [Definition: **Markup** takes the form of start-tags, end-tags, empty-element tags, entity references, character references, comments, CDATA section delimiters, document type declarations, processing instructions, XML declarations, text declarations, and any white space that is at the top level of the document entity (that is, outside the document element and not inside any other markup).]

[Definition: All text that is not markup constitutes the **character data** of the document.]

The ampersand character (&) and the left angle bracket (<) may appear in their literal form *only* when used as markup delimiters, or within a comment, a processing instruction, or a CDATA section. If they are needed elsewhere, they must be escaped using either numeric character references or the strings "`&`" and "`<`" respectively. The right angle bracket (>) may be represented using the string "`>`", and must, for compatibility, be escaped using "`>`" or a character reference when it appears in the string "`]]>`" in content, when that string is not marking the end of a CDATA section.

In the content of elements, character data is any string of characters which does not contain the start-delimiter of any markup. In a CDATA section, character data is any string of characters not including the CDATA-section-close delimiter, "`]]>`".

To allow attribute values to contain both single and double quotes, the apostrophe or single-quote character (') may be represented as "`&apos;`", and the double-quote character (") as "`&quot;`".

Character Data

```
[14]  CharData  ::=  [^<&]* - ([^<&]* ']]>' [^<&]*)
```

2.5 Comments

[Definition: **Comments** may appear anywhere in a document outside other markup; in addition, they may appear within the document type declaration at places allowed by the grammar. They are not part of the document's character data; an XML processor may, but need not, make it possible for an application to retrieve the text of comments. For compatibility, the string "`--`" (double-hyphen) must not occur within comments.] Parameter entity references are not recognized within comments.

Comments

```
[15] Comment ::= '<!--' ((Char - '-') | ('-' (Char - '-')))* '-->'
```

An example of a comment:

```
<!-- declarations for <head> & <body> -->
```

Note that the grammar does not allow a comment ending in `--->`. The following example is *not* well-formed.

```
<!-- B+, B, or B--->
```

2.6 Processing Instructions

[Definition: **Processing instructions** (PIs) allow documents to contain instructions for applications.]

Processing Instructions

```
[16] PI       ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'
[17] PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))
```

PIs are not part of the document's character data, but must be passed through to the application. The PI begins with a target (PITarget) used to identify the application to which the instruction is directed. The target names "`XML`", "`xml`", and so on are reserved for standardization in this or future versions of this specification. The XML Notation mechanism may be used for formal declaration of PI targets. Parameter entity references are not recognized within processing instructions.

2.7 CDATA Sections

[Definition: **CDATA sections** may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup. CDATA sections begin with the string "`<![CDATA[`" and end with the string "`]]>`":]

CDATA Sections

```
[18] CDSect  ::= CDStart CData CDEnd
[19] CDStart ::= '<![CDATA['
[20] CData   ::= (Char* - (Char* ']]>' Char*))
[21] CDEnd   ::= ']]>'
```

Within a CDATA section, only the CDEnd string is recognized as markup, so that left angle brackets and ampersands may occur in their literal form; they need not (and cannot) be escaped using "`<`" and "`&`". CDATA sections cannot nest.

An example of a CDATA section, in which "`<greeting>`" and "`</greeting>`" are recognized as character data, not markup:

```
<![CDATA[<greeting>Hello, world!</greeting>]]>
```

2.8 Prolog and Document Type Declaration

[Definition: XML documents should begin with an **XML declaration** which specifies the version of XML being used.] For example, the following is a complete XML document, well-formed but not valid:

```
<?xml version="1.0"?> <greeting>Hello, world!</greeting>
```

and so is this:

```
<greeting>Hello, world!</greeting>
```

The version number "`1.0`" should be used to indicate conformance to this version of this specification; it is an error for a document to use the value "`1.0`" if it does not conform to this version of this specification. It is the intent of the XML working group to give later versions of this specification numbers other than "`1.0`", but this intent does not indicate a commitment to produce any future versions of XML, nor if any are produced, to use any particular numbering scheme. Since future versions are not ruled out, this construct is provided as a means to allow the possibility of automatic version recognition, should it become necessary. Processors may signal an error if they receive documents labeled with versions they do not support.

The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute-value pairs with its logical structures. XML provides a mechanism, the document type declaration, to define constraints on the logical structure and to support the use of predefined storage units. [Definition: An XML document is **valid** if it has an associated document type declaration and if the document complies with the constraints expressed in it.]

The document type declaration must appear before the first element in the document.

Prolog

```
[22]  prolog        ::=  XMLDecl? Misc* (doctypedecl Misc*)?
[23]  XMLDecl       ::=  '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24]  VersionInfo   ::=  S 'version' Eq ("'" VersionNum "'" | '"' VersionNum
                         '"')/* */
[25]  Eq            ::=  S? '=' S?
[26]  VersionNum    ::=  ([a-zA-Z0-9_.:] | '-')+
[27]  Misc          ::=  Comment | PI | S
```

[Definition: The XML **document type declaration** contains or points to markup declarations that provide a grammar for a class of documents. This grammar is known as a document type definition, or **DTD**. The document type declaration can point to an external subset (a special kind of external entity) containing markup declarations, or can contain the markup declarations directly in an internal subset, or can do both. The DTD for a document consists of both subsets taken together.]

[Definition: A **markup declaration** is an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration.] These declarations may be contained in whole or in part within parameter entities, as described in the well-formedness and validity constraints below. For further information, see **4 Physical Structures**.

Document Type Definition

[28]	doctypedecl	::=	'<!DOCTYPE' S Name (S ExternalID)? S? ('[' (markupdecl | DeclSep)* ']' S?)? '>'	[VC: Root Element Type] [WFC: External Subset] /* */
[28a]	DeclSep	::=	PEReference | S	[WFC: PE Between Declarations] /* */
[29]	markupdecl	::=	elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment	[VC: Proper Declaration/PE Nesting] [WFC: PEs in Internal Subset]

Note that it is possible to construct a well-formed document containing a doctypedecl that neither points to an external subset nor contains an internal subset.

The markup declarations may be made up in whole or in part of the replacement text of parameter entities. The productions later in this specification for individual nonterminals (elementdecl, AttlistDecl, and so on) describe the declarations *after* all the parameter entities have been included.

Parameter entity references are recognized anywhere in the DTD (internal and external subsets and external parameter entities), except in literals, processing instructions, comments, and the contents of ignored conditional sections (see **3.4 Conditional Sections**). They are also recognized in entity value literals. The use of parameter entities in the internal subset is restricted as described below.

Validity constraint: Root Element Type

The Name in the document type declaration must match the element type of the root element.

Validity constraint: Proper Declaration/PE Nesting

Parameter-entity replacement text must be properly nested with markup declarations. That is to say, if either the first character or the last character of a markup declaration (markupdecl above) is contained in the replacement text for a parameter-entity reference, both must be contained in the same replacement text.

Well-formedness constraint: PEs in Internal Subset

In the internal DTD subset, parameter-entity references can occur only where markup declarations can occur, not within markup declarations. (This does not apply to references that occur in external parameter entities or to the external subset.)

Well-formedness constraint: External Subset

The external subset, if any, must match the production for extSubset.

Well-formedness constraint: PE Between Declarations

The replacement text of a parameter entity reference in a DeclSep must match the production extSubsetDecl.

Like the internal subset, the external subset and any external parameter entities referenced in a DeclSep must consist of a series of complete markup declarations of the types allowed by the non-terminal symbol markupdecl, interspersed with white space or parameter-entity references. However, portions of the contents of the external subset or of these external parameter entities may conditionally be ignored by using the conditional section construct; this is not allowed in the internal subset.

External Subset

```
[30] extSubset     ::= TextDecl? extSubsetDecl
[31] extSubsetDecl ::= ( markupdecl | conditionalSect | DeclSep)* /* */
```

The external subset and external parameter entities also differ from the internal subset in that in them, parameter-entity references are permitted *within* markup declarations, not only *between* markup declarations.

An example of an XML document with a document type declaration:

```
<?xml version="1.0"?> <!DOCTYPE greeting SYSTEM "hello.dtd"> <greeting>Hello,
world!</greeting>
```

The system identifier "`hello.dtd`" gives the address (a URI reference) of a DTD for the document.

The declarations can also be given locally, as in this example:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

If both the external and internal subsets are used, the internal subset is considered to occur before the external subset. This has the effect that entity and attribute-list declarations in the internal subset take precedence over those in the external subset.

2.9 Standalone Document Declaration

Markup declarations can affect the content of the document, as passed from an XML processor to an application; examples are attribute defaults and entity declarations. The standalone document declaration, which may appear as a component of the XML declaration, signals whether or not there are such declarations which appear external to the document entity or in parameter entities. [Definition: An **external markup declaration** is defined as a markup declaration occurring in the external subset or in a parameter entity (external or internal, the latter being included because non-validating processors are not required to read them).]

Standalone Document Declaration

```
[32] SDDecl ::= S 'standalone' Eq (("'" ('yes' |
                'no') "'") | ('"' ('yes' | 'no')
                '"'))
```

[VC: Standalone Document Declaration]

In a standalone document declaration, the value "yes" indicates that there are no external markup declarations which affect the information passed from the XML processor to the application. The value "no" indicates that there are or may be such external markup declarations. Note that the standalone document declaration only denotes the presence of external *declarations*; the presence, in a document, of references to external *entities*, when those entities are internally declared, does not change its standalone status.

If there are no external markup declarations, the standalone document declaration has no meaning. If there are external markup declarations but there is no standalone document declaration, the value "no" is assumed.

Any XML document for which `standalone="no"` holds can be converted algorithmically to a standalone document, which may be desirable for some network delivery applications.

Validity constraint: Standalone Document Declaration

The standalone document declaration must have the value "no" if any external markup declarations contain declarations of:

- attributes with default values, if elements to which these attributes apply appear in the document without

specifications of values for these attributes, or

- entities (other than `amp`, `lt`, `gt`, `apos`, `quot`), if references to those entities appear in the document, or
- attributes with values subject to normalization, where the attribute appears in the document with a value which will change as a result of normalization, or
- element types with element content, if white space occurs directly within any instance of those types.

An example XML declaration with a standalone document declaration:

```
<?xml version="1.0" standalone='yes'?>
```

2.10 White Space Handling

In editing XML documents, it is often convenient to use "white space" (spaces, tabs, and blank lines) to set apart the markup for greater readability. Such white space is typically not intended for inclusion in the delivered version of the document. On the other hand, "significant" white space that should be preserved in the delivered version is common, for example in poetry and source code.

An XML processor must always pass all characters in a document that are not markup through to the application. A validating XML processor must also inform the application which of these characters constitute white space appearing in element content.

A special attribute named `xml:space` may be attached to an element to signal an intention that in that element, white space should be preserved by applications. In valid documents, this attribute, like any other, must be declared if it is used. When declared, it must be given as an enumerated type whose values are one or both of "default" and "preserve". For example:

```
<!ATTLIST poem  xml:space (default|preserve) 'preserve'>

<!-- -->
<!ATTLIST pre xml:space (preserve) #FIXED 'preserve'>
```

The value "default" signals that applications' default white-space processing modes are acceptable for this element; the value "preserve" indicates the intent that applications preserve all the white space. This declared intent is considered to apply to all elements within the content of the element where it is specified, unless overriden with another instance of the `xml:space` attribute.

The root element of any document is considered to have signaled no intentions as regards application space handling, unless it provides a value for this attribute or the attribute is declared with a default value.

2.11 End-of-Line Handling

XML parsed entities are often stored in computer files which, for editing convenience, are organized into lines. These lines are typically separated by some combination of the characters carriage-return (#xD) and line-feed (#xA).

To simplify the tasks of applications, the characters passed to an application by the XML processor must be as if the XML processor normalized all line breaks in external parsed entities (including the document entity) on input, before parsing, by translating both the two-character sequence #xD #xA and any #xD that is not followed by #xA to a single #xA character.

2.12 Language Identification

In document processing, it is often useful to identify the natural or formal language in which the content is written. A special attribute named `xml:lang` may be inserted in documents to specify the language used in the contents and attribute values of any element in an XML document. In valid documents, this attribute, like any other, must be

declared if it is used. The values of the attribute are language identifiers as defined by [IETF RFC 1766], Tags for the Identification of Languages, or its successor on the IETF Standards Track.

Note:

[IETF RFC 1766] tags are constructed from two-letter language codes as defined by [ISO 639], from two-letter country codes as defined by [ISO 3166], or from language identifiers registered with the Internet Assigned Numbers Authority [IANA-LANGCODES]. It is expected that the successor to [IETF RFC 1766] will introduce three-letter language codes for languages not presently covered by [ISO 639].

(Productions 33 through 38 have been removed.)

For example:

```
<p xml:lang="en">The quick brown fox jumps over the lazy dog.</p>
<p xml:lang="en-GB">What colour is it?</p>
<p xml:lang="en-US">What color is it?</p>
<sp who="Faust" desc='leise' xml:lang="de">
  <l>Habe nun, ach! Philosophie,</l>
  <l>Juristerei, und Medizin</l>
  <l>und leider auch Theologie</l>
  <l>durchaus studiert mit heißem Bemüh'n.</l>
</sp>
```

The intent declared with `xml:lang` is considered to apply to all attributes and content of the element where it is specified, unless overridden with an instance of `xml:lang` on another element within that content.

A simple declaration for `xml:lang` might take the form

```
xml:lang NMTOKEN #IMPLIED
```

but specific default values may also be given, if appropriate. In a collection of French poems for English students, with glosses and notes in English, the `xml:lang` attribute might be declared this way:

```
<!ATTLIST poem   xml:lang NMTOKEN 'fr'>
<!ATTLIST gloss  xml:lang NMTOKEN 'en'>
<!ATTLIST note   xml:lang NMTOKEN 'en'>
```

3 Logical Structures

[Definition: Each XML document contains one or more **elements**, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications.] Each attribute specification has a name and a value.

Element

[39] `element` ::= `EmptyElemTag`
| `STag` `content` `ETag` [WFC: Element Type Match]
[VC: Element Valid]

This specification does not constrain the semantics, use, or (beyond syntax) names of the element types and attributes, except that names beginning with a match to `(('X'|'x')('M'|'m')('L'|'l'))` are reserved for standardization in this or future versions of this specification.

Well-formedness constraint: Element Type Match

The Name in an element's end-tag must match the element type in the start-tag.

Validity constraint: Element Valid

An element is valid if there is a declaration matching elementdecl where the Name matches the element type, and one of the following holds:

1. The declaration matches **EMPTY** and the element has no content.
2. The declaration matches children and the sequence of child elements belongs to the language generated by the regular expression in the content model, with optional white space (characters matching the nonterminal S) between the start-tag and the first child element, between child elements, or between the last child element and the end-tag. Note that a CDATA section containing only white space does not match the nonterminal S, and hence cannot appear in these positions.
3. The declaration matches Mixed and the content consists of character data and child elements whose types match names in the content model.
4. The declaration matches **ANY**, and the types of any child elements have been declared.

3.1 Start-Tags, End-Tags, and Empty-Element Tags

[Definition: The beginning of every non-empty XML element is marked by a **start-tag**.]

Start-tag

```
[40] STag      ::= '<' Name (S Attribute)* S? '>' [WFC: Unique Att Spec]
[41] Attribute ::= Name Eq AttValue               [VC: Attribute Value Type]
                                                  [WFC: No External Entity References]
                                                  [WFC: No < in Attribute Values]
```

The Name in the start- and end-tags gives the element's **type**. [Definition: The Name-AttValue pairs are referred to as the **attribute specifications** of the element], [Definition: with the Name in each pair referred to as the **attribute name**] and [Definition: the content of the AttValue (the text between the ' or " delimiters) as the **attribute value**.]Note that the order of attribute specifications in a start-tag or empty-element tag is not significant.

Well-formedness constraint: Unique Att Spec

No attribute name may appear more than once in the same start-tag or empty-element tag.

Validity constraint: Attribute Value Type

The attribute must have been declared; the value must be of the type declared for it. (For attribute types, see **3.3 Attribute-List Declarations**.)

Well-formedness constraint: No External Entity References

Attribute values cannot contain direct or indirect entity references to external entities.

Well-formedness constraint: No < in Attribute Values

The replacement text of any entity referred to directly or indirectly in an attribute value must not contain a <.

An example of a start-tag:

```
<termdef id="dt-dog" term="dog">
```

[Definition: The end of every element that begins with a start-tag must be marked by an **end-tag** containing a name that echoes the element's type as given in the start-tag:]

End-tag

```
[42]  ETag  ::=  '</' Name S? '>'
```

An example of an end-tag:

```
</termdef>
```

[Definition: The text between the start-tag and end-tag is called the element's **content**:]

Content of Elements

```
[43]  content  ::=  CharData? ((element | Reference | CDSect | PI | Comment)   /* */
                    CharData?)*
```

[Definition: An element with no content is said to be **empty**.] The representation of an empty element is either a start-tag immediately followed by an end-tag, or an empty-element tag. [Definition: An **empty-element tag** takes a special form:]

Tags for Empty Elements

```
[44]  EmptyElemTag  ::=  '<' Name (S Attribute)* S? '/>' [WFC: Unique Att Spec]
```

Empty-element tags may be used for any element which has no content, whether or not it is declared using the keyword **EMPTY**. For interoperability, the empty-element tag should be used, and should only be used, for elements which are declared EMPTY.

Examples of empty elements:

```
<IMG align="left"
 src="http://www.w3.org/Icons/WWW/w3c_home" />
<br></br>
<br/>
```

3.2 Element Type Declarations

The element structure of an XML document may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element's content.

Element type declarations often constrain which element types can appear as children of the element. At user option, an XML processor may issue a warning when a declaration mentions an element type for which no declaration is provided, but this is not an error.

[Definition: An **element type declaration** takes the form:]

Element Type Declaration

```
[45]  elementdecl  ::=  '<!ELEMENT' S Name S contentspec S? '>'   [VC: Unique Element Type Declaration]
[46]  contentspec  ::=  'EMPTY' | 'ANY' | Mixed | children
```

where the Name gives the element type being declared.

Validity constraint: Unique Element Type Declaration

No element type may be declared more than once.

Examples of element type declarations:

```
<!ELEMENT br EMPTY>
<!ELEMENT p (#PCDATA|emph)* >
<!ELEMENT %name.para; %content.para; >
<!ELEMENT container ANY>
```

3.2.1 Element Content

[Definition: An element type has **element content** when elements of that type must contain only child elements (no character data), optionally separated by white space (characters matching the nonterminal S).][Definition: In this case, the constraint includes a **content model**, a simple grammar governing the allowed types of the child elements and the order in which they are allowed to appear.] The grammar is built on content particles (cps), which consist of names, choice lists of content particles, or sequence lists of content particles:

Element-content Models

```
[47]  children  ::=  (choice | seq) ('?' | '*' | '+')?
[48]  cp        ::=  (Name | choice | seq) ('?' | '*' |
                     '+')?
[49]  choice    ::=  '(' S? cp ( S? '|' S? cp )+ S? ')'      /* */
                                                             /* */
                                                             [VC: Proper Group/PE Nesting]
[50]  seq       ::=  '(' S? cp ( S? ',' S? cp )* S? ')'      /* */
                                                             [VC: Proper Group/PE Nesting]
```

where each Name is the type of an element which may appear as a child. Any content particle in a choice list may appear in the element content at the location where the choice list appears in the grammar; content particles occurring in a sequence list must each appear in the element content in the order given in the list. The optional character following a name or list governs whether the element or the content particles in the list may occur one or more (+), zero or more (*), or zero or one times (?). The absence of such an operator means that the element or content particle must appear exactly once. This syntax and meaning are identical to those used in the productions in this specification.

The content of an element matches a content model if and only if it is possible to trace out a path through the content model, obeying the sequence, choice, and repetition operators and matching each element in the content against an element type in the content model. For compatibility, it is an error if an element in the document can match more than one occurrence of an element type in the content model. For more information, see **E Deterministic Content Models**.

Validity constraint: Proper Group/PE Nesting

Parameter-entity replacement text must be properly nested with parenthesized groups. That is to say, if either of the opening or closing parentheses in a choice, seq, or Mixed construct is contained in the replacement text for a parameter entity, both must be contained in the same replacement text.

For interoperability, if a parameter-entity reference appears in a choice, seq, or Mixed construct, its replacement text should contain at least one non-blank character, and neither the first nor last non-blank character of the replacement text should be a connector (| or ,).

Examples of element-content models:

```
<!ELEMENT spec (front, body, back?)>
<!ELEMENT div1 (head, (p | list | note)*, div2*)>
<!ELEMENT dictionary-body (%div.mix; | %dict.mix;)*>
```

3.2.2 Mixed Content

[Definition: An element type has **mixed content** when elements of that type may contain character data, optionally interspersed with child elements.] In this case, the types of the child elements may be constrained, but not their order or their number of occurrences:

Mixed-content Declaration

```
[51]  Mixed  ::=  '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
                | '(' S? '#PCDATA' S? ')'                    [VC: Proper Group/PE Nesting]
                                                             [VC: No Duplicate Types]
```

where the Names give the types of elements that may appear as children. The keyword **#PCDATA** derives historically from the term "parsed character data."

Validity constraint: No Duplicate Types

The same name must not appear more than once in a single mixed-content declaration.

Examples of mixed content declarations:

```
<!ELEMENT p (#PCDATA|a|ul|b|i|em)*>
<!ELEMENT p (#PCDATA | %font; | %phrase; | %special; | %form;)* >
<!ELEMENT b (#PCDATA)>
```

3.3 Attribute-List Declarations

Attributes are used to associate name-value pairs with elements. Attribute specifications may appear only within start-tags and empty-element tags; thus, the productions used to recognize them appear in **3.1 Start-Tags, End-Tags, and Empty-Element Tags**. Attribute-list declarations may be used:

- To define the set of attributes pertaining to a given element type.
- To establish type constraints for these attributes.
- To provide default values for attributes.

[Definition: **Attribute-list declarations** specify the name, data type, and default value (if any) of each attribute associated with a given element type:]

Attribute-list Declaration

```
[52]  AttlistDecl  ::=  '<!ATTLIST' S Name AttDef* S? '>'
[53]  AttDef       ::=  S Name S AttType S DefaultDecl
```

The Name in the AttlistDecl rule is the type of an element. At user option, an XML processor may issue a warning if attributes are declared for an element type not itself declared, but this is not an error. The Name in the AttDef rule is the name of the attribute.

When more than one AttlistDecl is provided for a given element type, the contents of all those provided are merged. When more than one definition is provided for the same attribute of a given element type, the first declaration is binding and later declarations are ignored. For interoperability, writers of DTDs may choose to provide at most one

attribute-list declaration for a given element type, at most one attribute definition for a given attribute name in an attribute-list declaration, and at least one attribute definition in each attribute-list declaration. For interoperability, an XML processor may at user option issue a warning when more than one attribute-list declaration is provided for a given element type, or more than one attribute definition is provided for a given attribute, but this is not an error.

3.3.1 Attribute Types

XML attribute types are of three kinds: a string type, a set of tokenized types, and enumerated types. The string type may take any literal string as a value; the tokenized types have varying lexical and semantic constraints. The validity constraints noted in the grammar are applied after the attribute value has been normalized as described in **3.3 Attribute-List Declarations**.

Attribute Types

```
[54]  AttType         ::=  StringType | TokenizedType |
                           EnumeratedType
[55]  StringType      ::=  'CDATA'
[56]  TokenizedType   ::=  'ID'                          [VC: ID]
                                                         [VC: One ID per
                                                         Element Type]
                                                         [VC: ID Attribute
                                                         Default]
                           | 'IDREF'                     [VC: IDREF]
                           | 'IDREFS'                    [VC: IDREF]
                           | 'ENTITY'                    [VC: Entity Name]
                           | 'ENTITIES'                  [VC: Entity Name]
                           | 'NMTOKEN'                   [VC: Name Token]
                           | 'NMTOKENS'                  [VC: Name Token]
```

Validity constraint: ID

Values of type **ID** must match the Name production. A name must not appear more than once in an XML document as a value of this type; i.e., ID values must uniquely identify the elements which bear them.

Validity constraint: One ID per Element Type

No element type may have more than one ID attribute specified.

Validity constraint: ID Attribute Default

An ID attribute must have a declared default of **#IMPLIED** or **#REQUIRED**.

Validity constraint: IDREF

Values of type **IDREF** must match the Name production, and values of type **IDREFS** must match Names; each Name must match the value of an ID attribute on some element in the XML document; i.e. **IDREF** values must match the value of some ID attribute.

Validity constraint: Entity Name

Values of type **ENTITY** must match the Name production, values of type **ENTITIES** must match Names; each Name must match the name of an unparsed entity declared in the DTD.

Validity constraint: Name Token

Values of type **NMTOKEN** must match the Nmtoken production; values of type **NMTOKENS** must match

Nmtokens.

[Definition: **Enumerated attributes** can take one of a list of values provided in the declaration]. There are two kinds of enumerated types:

Enumerated Attribute Types

[57]	EnumeratedType	::=	NotationType | Enumeration	
[58]	NotationType	::=	'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'	[VC: Notation Attributes] [VC: One Notation Per Element Type] [VC: No Notation on Empty Element]
[59]	Enumeration	::=	'(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'	[VC: Enumeration]

A **NOTATION** attribute identifies a notation, declared in the DTD with associated system and/or public identifiers, to be used in interpreting the element to which the attribute is attached.

Validity constraint: Notation Attributes

Values of this type must match one of the notation names included in the declaration; all notation names in the declaration must be declared.

Validity constraint: One Notation Per Element Type

No element type may have more than one **NOTATION** attribute specified.

Validity constraint: No Notation on Empty Element

For compatibility, an attribute of type **NOTATION** must not be declared on an element declared **EMPTY**.

Validity constraint: Enumeration

Values of this type must match one of the Nmtoken tokens in the declaration.

For interoperability, the same Nmtoken should not occur more than once in the enumerated attribute types of a single element type.

3.3.2 Attribute Defaults

An attribute declaration provides information on whether the attribute's presence is required, and if not, how an XML processor should react if a declared attribute is absent in a document.

Attribute Defaults

[60]	DefaultDecl	::=	'#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)	[VC: Required Attribute] [VC: Attribute Default Legal] [WFC: No < in Attribute Values] [VC: Fixed Attribute Default]

In an attribute declaration, **#REQUIRED** means that the attribute must always be provided, **#IMPLIED** that no default value is provided. [Definition: If the declaration is neither **#REQUIRED** nor **#IMPLIED**, then the AttValue value contains the declared **default** value; the **#FIXED** keyword states that the attribute must always have the default value. If a default value is declared, when an XML processor encounters an omitted attribute, it is to behave as though

the attribute were present with the declared default value.]

Validity constraint: Required Attribute

If the default declaration is the keyword **#REQUIRED**, then the attribute must be specified for all elements of the type in the attribute-list declaration.

Validity constraint: Attribute Default Legal

The declared default value must meet the lexical constraints of the declared attribute type.

Validity constraint: Fixed Attribute Default

If an attribute has a default value declared with the **#FIXED** keyword, instances of that attribute must match the default value.

Examples of attribute-list declarations:

```
<!ATTLIST termdef
          id      ID      #REQUIRED
          name    CDATA   #IMPLIED>
<!ATTLIST list
          type    (bullets|ordered|glossary)  "ordered">
<!ATTLIST form
          method  CDATA   #FIXED "POST">
```

3.3.3 Attribute-Value Normalization

Before the value of an attribute is passed to the application or checked for validity, the XML processor must normalize the attribute value by applying the algorithm below, or by using some other method such that the value passed to the application is the same as that produced by the algorithm.

1. All line breaks must have been normalized on input to #xA as described in **2.11 End-of-Line Handling**, so the rest of this algorithm operates on text normalized in this way.
2. Begin with a normalized value consisting of the empty string.
3. For each character, entity reference, or character reference in the unnormalized attribute value, beginning with the first and continuing to the last, do the following:
   - For a character reference, append the referenced character to the normalized value.
   - For an entity reference, recursively apply step 3 of this algorithm to the replacement text of the entity.
   - For a white space character (#x20, #xD, #xA, #x9), append a space character (#x20) to the normalized value.
   - For another character, append the character to the normalized value.

If the attribute type is not CDATA, then the XML processor must further process the normalized attribute value by discarding any leading and trailing space (#x20) characters, and by replacing sequences of space (#x20) characters by a single space (#x20) character.

Note that if the unnormalized attribute value contains a character reference to a white space character other than space (#x20), the normalized value contains the referenced character itself (#xD, #xA or #x9). This contrasts with the case where the unnormalized value contains a white space character (not a reference), which is replaced with a space character (#x20) in the normalized value and also contrasts with the case where the unnormalized value contains an entity reference whose replacement text contains a white space character; being recursively processed, the white space character is replaced with a space character (#x20) in the normalized value.

All attributes for which no declaration has been read should be treated by a non-validating processor as if declared **CDATA**.

Following are examples of attribute normalization. Given the following declarations:

```
<!ENTITY d "&#xD;">
<!ENTITY a "&#xA;">
<!ENTITY da "&#xD;&#xA;">
```

the attribute specifications in the left column below would be normalized to the character sequences of the middle column if the attribute a is declared **NMTOKENS** and to those of the right columns if a is declared **CDATA**.

Attribute specification	a is NMTOKENS	a is CDATA
a=" xyz"	x y z	#x20 #x20 x y z
a="&d;&d;A&a;&a;B&da;"	A #x20 B	#x20 #x20 A #x20 #x20 B #x20 #x20
a= "

A

B
" | #xD #xD A #xA #xA B #xD
#xA | #xD #xD A #xA
#xA B #xD #xD |

Note that the last example is invalid (but well-formed) if a is declared to be of type **NMTOKENS**.

3.4 Conditional Sections

[Definition: **Conditional sections** are portions of the document type declaration external subset which are included in, or excluded from, the logical structure of the DTD based on the keyword which governs them.]

Conditional Section

```
[61] conditionalSect    ::= includeSect | ignoreSect
[62] includeSect        ::= '<![' S? 'INCLUDE' S? '['
                            extSubsetDecl ']]>'          /* */
                                                         [VC: Proper
                                                         Conditional
                                                         Section/PE Nesting]
[63] ignoreSect         ::= '<![' S? 'IGNORE' S? '['
                            ignoreSectContents* ']]>'    /* */
                                                         [VC: Proper
                                                         Conditional
                                                         Section/PE Nesting]
[64] ignoreSectContents ::= Ignore ('<![' ignoreSectContents
                            ']]>' Ignore)*
[65] Ignore             ::= Char* - (Char* ('<![' | ']]>')
                            Char*)
```

Validity constraint: Proper Conditional Section/PE Nesting

If any of the "<![", "[", or "]]>" of a conditional section is contained in the replacement text for a parameter-entity reference, all of them must be contained in the same replacement text.

Like the internal and external DTD subsets, a conditional section may contain one or more complete declarations, comments, processing instructions, or nested conditional sections, intermingled with white space.

If the keyword of the conditional section is **INCLUDE**, then the contents of the conditional section are part of the DTD. If the keyword of the conditional section is **IGNORE**, then the contents of the conditional section are not logically part of the DTD. If a conditional section with a keyword of **INCLUDE** occurs within a larger conditional section with a keyword of **IGNORE**, both the outer and the inner conditional sections are ignored. The contents of an ignored conditional section are parsed by ignoring all characters after the "[" following the keyword, except conditional section starts "<![" and ends "]]>", until the matching conditional section end is found. Parameter entity references are not recognized in this process.

If the keyword of the conditional section is a parameter-entity reference, the parameter entity must be replaced by its content before the processor decides whether to include or ignore the conditional section.

An example:

```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >

<![%draft;[
<!ELEMENT book (comments*, title, body, supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>
```

4 Physical Structures

[Definition: An XML document may consist of one or many storage units. These are called **entities**; they all have **content** and are all (except for the document entity and the external DTD subset) identified by entity **name**.] Each XML document has one entity called the document entity, which serves as the starting point for the XML processor and may contain the whole document.

Entities may be either parsed or unparsed. [Definition: A **parsed entity's** contents are referred to as its replacement text; this text is considered an integral part of the document.]

[Definition: An **unparsed entity** is a resource whose contents may or may not be text, and if text, may be other than XML. Each unparsed entity has an associated notation, identified by name. Beyond a requirement that an XML processor make the identifiers for the entity and notation available to the application, XML places no constraints on the contents of unparsed entities.]

Parsed entities are invoked by name using entity references; unparsed entities by name, given in the value of **ENTITY** or **ENTITIES** attributes.

[Definition: **General entities** are entities for use within the document content. In this specification, general entities are sometimes referred to with the unqualified term *entity* when this leads to no ambiguity.] [Definition: **Parameter entities** are parsed entities for use within the DTD.] These two types of entities use different forms of reference and are recognized in different contexts. Furthermore, they occupy different namespaces; a parameter entity and a general entity with the same name are two distinct entities.

4.1 Character and Entity References

[Definition: A **character reference** refers to a specific character in the ISO/IEC 10646 character set, for example one not directly accessible from available input devices.]

Character Reference

```
[66]  CharRef  ::=  '&#' [0-9]+ ';'
                  | '&#x' [0-9a-fA-F]+ ';' [WFC: Legal Character]
```

Well-formedness constraint: Legal Character

Characters referred to using character references must match the production for Char.

If the character reference begins with "&#x", the digits and letters up to the terminating ; provide a hexadecimal representation of the character's code point in ISO/IEC 10646. If it begins just with "&#", the digits up to the terminating ; provide a decimal representation of the character's code point.

[Definition: An **entity reference** refers to the content of a named entity.] [Definition: References to parsed general entities use ampersand (&) and semicolon (;) as delimiters.] [Definition: **Parameter-entity references** use percent-sign (%) and semicolon (;) as delimiters.]

Entity Reference

[67]	Reference	::=	EntityRef | CharRef	
[68]	EntityRef	::=	'&' Name ';'	[WFC: Entity Declared] [VC: Entity Declared] [WFC: Parsed Entity] [WFC: No Recursion]
[69]	PEReference	::=	'%' Name ';'	[VC: Entity Declared] [WFC: No Recursion] [WFC: In DTD]

Well-formedness constraint: Entity Declared

In a document without any DTD, a document with only an internal DTD subset which contains no parameter entity references, or a document with "standalone='yes'", for an entity reference that does not occur within the external subset or a parameter entity, the Name given in the entity reference must match that in an entity declaration that does not occur within the external subset or a parameter entity, except that well-formed documents need not declare any of the following entities: amp, lt, gt, apos, quot. The declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration.

Note that if entities are declared in the external subset or in external parameter entities, a non-validating processor is not obligated to read and process their declarations; for such documents, the rule that an entity must be declared is a well-formedness constraint only if standalone='yes'.

Validity constraint: Entity Declared

In a document with an external subset or external parameter entities with "standalone='no'", the Name given in the entity reference must match that in an entity declaration. For interoperability, valid documents should declare the entities amp, lt, gt, apos, quot, in the form specified in **4.6 Predefined Entities**. The declaration of a parameter entity must precede any reference to it. Similarly, the declaration of a general entity must precede any attribute-list declaration containing a default value with a direct or indirect reference to that general entity.

Well-formedness constraint: Parsed Entity

An entity reference must not contain the name of an unparsed entity. Unparsed entities may be referred to only in attribute values declared to be of type **ENTITY** or **ENTITIES**.

Well-formedness constraint: No Recursion

A parsed entity must not contain a recursive reference to itself, either directly or indirectly.

Well-formedness constraint: In DTD

Parameter-entity references may only appear in the DTD.

Examples of character and entity references:

```
Type <key>less-than</key> (&#x3C;) to save options.
This document was prepared on &docdate; and
is classified &security-level;.
```

Example of a parameter-entity reference:

```
<!-- declare the parameter entity "ISOLat2"... -->
<!ENTITY % ISOLat2
         SYSTEM "http://www.xml.com/iso/isolat2-xml.entities" >
<!-- ... now reference it. -->
%ISOLat2;
```

4.2 Entity Declarations

[Definition: Entities are declared thus:]

Entity Declaration

```
[70]  EntityDecl  ::=  GEDecl | PEDecl
[71]  GEDecl      ::=  '<!ENTITY' S Name S EntityDef S? '>'
[72]  PEDecl      ::=  '<!ENTITY' S '%' S Name S PEDef S? '>'
[73]  EntityDef   ::=  EntityValue | (ExternalID NDataDecl?)
[74]  PEDef       ::=  EntityValue | ExternalID
```

The Name identifies the entity in an entity reference or, in the case of an unparsed entity, in the value of an **ENTITY** or **ENTITIES** attribute. If the same entity is declared more than once, the first declaration encountered is binding; at user option, an XML processor may issue a warning if entities are declared multiple times.

4.2.1 Internal Entities

[Definition: If the entity definition is an EntityValue, the defined entity is called an **internal entity**. There is no separate physical storage object, and the content of the entity is given in the declaration.] Note that some processing of entity and character references in the literal entity value may be required to produce the correct replacement text: see **4.5 Construction of Internal Entity Replacement Text**.

An internal entity is a parsed entity.

Example of an internal entity declaration:

```
<!ENTITY Pub-Status "This is a pre-release of the
 specification.">
```

4.2.2 External Entities

[Definition: If the entity is not internal, it is an **external entity**, declared as follows:]

External Entity Declaration

```
[75]  ExternalID  ::=  'SYSTEM' S SystemLiteral
                     | 'PUBLIC' S PubidLiteral S SystemLiteral
[76]  NDataDecl   ::=  S 'NDATA' S Name                    [VC: Notation Declared]
```

If the NDataDecl is present, this is a general unparsed entity; otherwise it is a parsed entity.

Validity constraint: Notation Declared

The Name must match the declared name of a notation.

[Definition: The SystemLiteral is called the entity's **system identifier**. It is a URI reference (as defined in [IETF RFC 2396], updated by [IETF RFC 2732]), meant to be dereferenced to obtain input for the XML processor to construct the entity's replacement text.] It is an error for a fragment identifier (beginning with a # character) to be part of a system identifier. Unless otherwise provided by information outside the scope of this specification (e.g. a special XML element type defined by a particular DTD, or a processing instruction defined by a particular application specification), relative URIs are relative to the location of the resource within which the entity declaration occurs. A URI might thus be relative to the document entity, to the entity containing the external DTD subset, or to some other external parameter entity.

URI references require encoding and escaping of certain characters. The disallowed characters include all non-ASCII characters, plus the excluded characters listed in Section 2.4 of [IETF RFC 2396], except for the number sign (#) and percent sign (%) characters and the square bracket characters re-allowed in [IETF RFC 2732]. Disallowed characters must be escaped as follows:

1. Each disallowed character is converted to UTF-8 [IETF RFC 2279] as one or more bytes.
2. Any octets corresponding to a disallowed character are escaped with the URI escaping mechanism (that is, converted to %HH, where HH is the hexadecimal notation of the byte value).
3. The original character is replaced by the resulting character sequence.

[Definition: In addition to a system identifier, an external identifier may include a **public identifier**.] An XML processor attempting to retrieve the entity's content may use the public identifier to try to generate an alternative URI reference. If the processor is unable to do so, it must use the URI reference specified in the system literal. Before a match is attempted, all strings of white space in the public identifier must be normalized to single space characters (#x20), and leading and trailing white space must be removed.

Examples of external entity declarations:

```
<!ENTITY open-hatch
         SYSTEM "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY open-hatch
         PUBLIC "-//Textuality//TEXT Standard open-hatch boilerplate//EN"
         "http://www.textuality.com/boilerplate/OpenHatch.xml">
<!ENTITY hatch-pic
         SYSTEM "../grafix/OpenHatch.gif"
         NDATA gif >
```

4.3 Parsed Entities

4.3.1 The Text Declaration

External parsed entities should each begin with a **text declaration**.

Text Declaration

```
[77]  TextDecl  ::=  '<?xml' VersionInfo? EncodingDecl S? '?>'
```

The text declaration must be provided literally, not by reference to a parsed entity. No text declaration may appear at any position other than the beginning of an external parsed entity. The text declaration in an external parsed entity is not considered part of its replacement text.

4.3.2 Well-Formed Parsed Entities

The document entity is well-formed if it matches the production labeled document. An external general parsed entity is well-formed if it matches the production labeled extParsedEnt. All external parameter entities are well-formed by definition.

Well-Formed External Parsed Entity

```
[78]  extParsedEnt  ::=  TextDecl? content
```

An internal general parsed entity is well-formed if its replacement text matches the production labeled content. All internal parameter entities are well-formed by definition.

A consequence of well-formedness in entities is that the logical and physical structures in an XML document are properly nested; no start-tag, end-tag, empty-element tag, element, comment, processing instruction, character reference, or entity reference can begin in one entity and end in another.

4.3.3 Character Encoding in Entities

Each external parsed entity in an XML document may use a different encoding for its characters. All XML processors must be able to read entities in both the UTF-8 and UTF-16 encodings. The terms "UTF-8" and "UTF-16" in this specification do not apply to character encodings with any other labels, even if the encodings or labels are very similar to UTF-8 or UTF-16.

Entities encoded in UTF-16 must begin with the Byte Order Mark described by Annex F of [ISO/IEC 10646], Annex H of [ISO/IEC 10646-2000], section 2.4 of [Unicode], and section 2.7 of [Unicode3] (the ZERO WIDTH NO-BREAK SPACE character, #xFEFF). This is an encoding signature, not part of either the markup or the character data of the XML document. XML processors must be able to use this character to differentiate between UTF-8 and UTF-16 encoded documents.

Although an XML processor is required to read only entities in the UTF-8 and UTF-16 encodings, it is recognized that other encodings are used around the world, and it may be desired for XML processors to read entities that use them. In the absence of external character encoding information (such as MIME headers), parsed entities which are stored in an encoding other than UTF-8 or UTF-16 must begin with a text declaration (see **4.3.1 The Text Declaration**) containing an encoding declaration:

Encoding Declaration

```
[80]  EncodingDecl  ::=  S 'encoding' Eq ('"' EncName '"' |
                         "'" EncName "'" )
[81]  EncName       ::=  [A-Za-z] ([A-Za-z0-9._] | '-')*      /* Encoding name contains
                                                               only Latin characters */
```

In the document entity, the encoding declaration is part of the XML declaration. The EncName is the name of the encoding used.

In an encoding declaration, the values "`UTF-8`", "`UTF-16`", "`ISO-10646-UCS-2`", and "`ISO-10646-UCS-4`" should be used for the various encodings and transformations of Unicode / ISO/IEC 10646, the values "`ISO-8859-1`", "`ISO-8859-2`", ... "`ISO-8859-`n" (where n is the part number) should be used for the parts of ISO 8859, and the values "`ISO-2022-JP`", "`Shift_JIS`", and "`EUC-JP`" should be used for the various encoded forms of JIS X-0208-1997. It is recommended that character encodings registered (as *charset*s) with the Internet Assigned Numbers Authority [IANA-CHARSETS], other than those just listed, be referred to using their registered names; other encodings should use names starting with an "x-" prefix. XML processors should match character encoding names in a case-insensitive way and should either interpret an IANA-registered name as the encoding registered at IANA for that name or treat it as unknown (processors are, of course, not required to support all IANA-registered encodings).

In the absence of information provided by an external transport protocol (e.g. HTTP or MIME), it is an error for an entity including an encoding declaration to be presented to the XML processor in an encoding other than that named

in the declaration, or for an entity which begins with neither a Byte Order Mark nor an encoding declaration to use an encoding other than UTF-8. Note that since ASCII is a subset of UTF-8, ordinary ASCII entities do not strictly need an encoding declaration.

It is a fatal error for a TextDecl to occur other than at the beginning of an external entity.

It is a fatal error when an XML processor encounters an entity with an encoding that it is unable to process. It is a fatal error if an XML entity is determined (via default, encoding declaration, or higher-level protocol) to be in a certain encoding but contains octet sequences that are not legal in that encoding. It is also a fatal error if an XML entity contains no encoding declaration and its content is not legal UTF-8 or UTF-16.

Examples of text declarations containing encoding declarations:

```
<?xml encoding='UTF-8'?>
<?xml encoding='EUC-JP'?>
```

4.4 XML Processor Treatment of Entities and References

The table below summarizes the contexts in which character references, entity references, and invocations of unparsed entities might appear and the required behavior of an XML processor in each case. The labels in the leftmost column describe the recognition context:

Reference in Content

> as a reference anywhere after the start-tag and before the end-tag of an element; corresponds to the nonterminal content.

Reference in Attribute Value

> as a reference within either the value of an attribute in a start-tag, or a default value in an attribute declaration; corresponds to the nonterminal AttValue.

Occurs as Attribute Value

> as a Name, not a reference, appearing either as the value of an attribute which has been declared as type **ENTITY**, or as one of the space-separated tokens in the value of an attribute which has been declared as type **ENTITIES**.

Reference in Entity Value

> as a reference within a parameter or internal entity's literal entity value in the entity's declaration; corresponds to the nonterminal EntityValue.

Reference in DTD

> as a reference within either the internal or external subsets of the DTD, but outside of an EntityValue, AttValue, PI, Comment, SystemLiteral, PubidLiteral, or the contents of an ignored conditional section (see **3.4 Conditional Sections**).

.

	Entity Type				Character
	Parameter	Internal General	External Parsed General	Unparsed	
Reference in Content	Not recognized	Included	Included if validating	Forbidden	Included
Reference in Attribute Value	Not recognized	Included in literal	Forbidden	Forbidden	Included

Occurs as Attribute Value	Not recognized	Forbidden	Forbidden	Notify	Not recognized
Reference in EntityValue	Included in literal	Bypassed	Bypassed	Forbidden	Included
Reference in DTD	Included as PE	Forbidden	Forbidden	Forbidden	Forbidden

4.4.1 Not Recognized

Outside the DTD, the % character has no special significance; thus, what would be parameter entity references in the DTD are not recognized as markup in content. Similarly, the names of unparsed entities are not recognized except when they appear in the value of an appropriately declared attribute.

4.4.2 Included

[Definition: An entity is **included** when its replacement text is retrieved and processed, in place of the reference itself, as though it were part of the document at the location the reference was recognized.] The replacement text may contain both character data and (except for parameter entities) markup, which must be recognized in the usual way. (The string "`AT&T;`" expands to "`AT&T;`" and the remaining ampersand is not recognized as an entity-reference delimiter.) A character reference is **included** when the indicated character is processed in place of the reference itself.

4.4.3 Included If Validating

When an XML processor recognizes a reference to a parsed entity, in order to validate the document, the processor must include its replacement text. If the entity is external, and the processor is not attempting to validate the XML document, the processor may, but need not, include the entity's replacement text. If a non-validating processor does not include the replacement text, it must inform the application that it recognized, but did not read, the entity.

This rule is based on the recognition that the automatic inclusion provided by the SGML and XML entity mechanism, primarily designed to support modularity in authoring, is not necessarily appropriate for other applications, in particular document browsing. Browsers, for example, when encountering an external parsed entity reference, might choose to provide a visual indication of the entity's presence and retrieve it for display only on demand.

4.4.4 Forbidden

The following are forbidden, and constitute fatal errors:

- the appearance of a reference to an unparsed entity.
- the appearance of any character or general-entity reference in the DTD except within an EntityValue or AttValue.
- a reference to an external entity in an attribute value.

4.4.5 Included in Literal

When an entity reference appears in an attribute value, or a parameter entity reference appears in a literal entity value, its replacement text is processed in place of the reference itself as though it were part of the document at the location the reference was recognized, except that a single or double quote character in the replacement text is always treated as a normal data character and will not terminate the literal. For example, this is well-formed:

```
<!--  -->
<!ENTITY % YN '"Yes"' >
<!ENTITY WhatHeSaid "He said %YN;" >
```

while this is not:

```
<!ENTITY EndAttr "27'" >
<element attribute='a-&EndAttr;>
```

4.4.6 Notify

When the name of an unparsed entity appears as a token in the value of an attribute of declared type **ENTITY** or **ENTITIES**, a validating processor must inform the application of the system and public (if any) identifiers for both the entity and its associated notation.

4.4.7 Bypassed

When a general entity reference appears in the EntityValue in an entity declaration, it is bypassed and left as is.

4.4.8 Included as PE

Just as with external parsed entities, parameter entities need only be included if validating. When a parameter-entity reference is recognized in the DTD and included, its replacement text is enlarged by the attachment of one leading and one following space (#x20) character; the intent is to constrain the replacement text of parameter entities to contain an integral number of grammatical tokens in the DTD. This behavior does not apply to parameter entity references within entity values; these are described in **4.4.5 Included in Literal**.

4.5 Construction of Internal Entity Replacement Text

In discussing the treatment of internal entities, it is useful to distinguish two forms of the entity's value. [Definition: The **literal entity value** is the quoted string actually present in the entity declaration, corresponding to the non-terminal EntityValue.] [Definition: The **replacement text** is the content of the entity, after replacement of character references and parameter-entity references.]

The literal entity value as given in an internal entity declaration (EntityValue) may contain character, parameter-entity, and general-entity references. Such references must be contained entirely within the literal entity value. The actual replacement text that is included as described above must contain the *replacement text* of any parameter entities referred to, and must contain the character referred to, in place of any character references in the literal entity value; however, general-entity references must be left as-is, unexpanded. For example, given the following declarations:

```
<!ENTITY % pub    "&#xc9;ditions Gallimard" >
<!ENTITY   rights "All rights reserved" >
<!ENTITY   book   "La Peste: Albert Camus,
&#xA9; 1947 %pub;. &rights;" >
```

then the replacement text for the entity "`book`" is:

```
La Peste: Albert Camus,
© 1947 Éditions Gallimard. &rights;
```

The general-entity reference "`&rights;`" would be expanded should the reference "`&book;`" appear in the document's content or an attribute value.

These simple rules may have complex interactions; for a detailed discussion of a difficult example, see **D Expansion**

of Entity and Character References.

4.6 Predefined Entities

[Definition: Entity and character references can both be used to **escape** the left angle bracket, ampersand, and other delimiters. A set of general entities (`amp`, `lt`, `gt`, `apos`, `quot`) is specified for this purpose. Numeric character references may also be used; they are expanded immediately when recognized and must be treated as character data, so the numeric character references "`&#60;`" and "`&#38;`" may be used to escape `<` and `&` when they occur in character data.]

All XML processors must recognize these entities whether they are declared or not. For interoperability, valid XML documents should declare these entities, like any others, before using them. If the entities `lt` or `amp` are declared, they must be declared as internal entities whose replacement text is a character reference to the respective character (less-than sign or ampersand) being escaped; the double escaping is required for these entities so that references to them produce a well-formed result. If the entities `gt`, `apos`, or `quot` are declared, they must be declared as internal entities whose replacement text is the single character being escaped (or a character reference to that character; the double escaping here is unnecessary but harmless). For example:

```
<!ENTITY lt     "&#38;#60;">
<!ENTITY gt     "&#62;">
<!ENTITY amp    "&#38;#38;">
<!ENTITY apos   "&#39;">
<!ENTITY quot   "&#34;">
```

4.7 Notation Declarations

[Definition: **Notations** identify by name the format of unparsed entities, the format of elements which bear a notation attribute, or the application to which a processing instruction is addressed.]

[Definition: **Notation declarations** provide a name for the notation, for use in entity and attribute-list declarations and in attribute specifications, and an external identifier for the notation which may allow an XML processor or its client application to locate a helper application capable of processing data in the given notation.]

Notation Declarations

[82]	`NotationDecl`	::=	`'<!NOTATION' S Name S (ExternalID | PublicID) S? '>'`	[VC: Unique Notation Name]
[83]	`PublicID`	::=	`'PUBLIC' S PubidLiteral`	

Validity constraint: Unique Notation Name

Only one notation declaration can declare a given Name.

XML processors must provide applications with the name and external identifier(s) of any notation declared and referred to in an attribute value, attribute definition, or entity declaration. They may additionally resolve the external identifier into the system identifier, file name, or other information needed to allow the application to call a processor for data in the notation described. (It is not an error, however, for XML documents to declare and refer to notations for which notation-specific applications are not available on the system where the XML processor or application is running.)

4.8 Document Entity

[Definition: The **document entity** serves as the root of the entity tree and a starting-point for an XML processor.] This specification does not specify how the document entity is to be located by an XML processor; unlike other entities, the document entity has no name and might well appear on a processor input stream without any identification at all.

5 Conformance

5.1 Validating and Non-Validating Processors

Conforming XML processors fall into two classes: validating and non-validating.

Validating and non-validating processors alike must report violations of this specification's well-formedness constraints in the content of the document entity and any other parsed entities that they read.

[Definition: **Validating processors** must, at user option, report violations of the constraints expressed by the declarations in the DTD, and failures to fulfill the validity constraints given in this specification.] To accomplish this, validating XML processors must read and process the entire DTD and all external parsed entities referenced in the document.

Non-validating processors are required to check only the document entity, including the entire internal DTD subset, for well-formedness. [Definition: While they are not required to check the document for validity, they are required to **process** all the declarations they read in the internal DTD subset and in any parameter entity that they read, up to the first reference to a parameter entity that they do *not* read; that is to say, they must use the information in those declarations to normalize attribute values, include the replacement text of internal entities, and supply default attribute values.] Except when `standalone="yes"`, they must not process entity declarations or attribute-list declarations encountered after a reference to a parameter entity that is not read, since the entity may have contained overriding declarations.

5.2 Using XML Processors

The behavior of a validating XML processor is highly predictable; it must read every piece of a document and report all well-formedness and validity violations. Less is required of a non-validating processor; it need not read any part of the document other than the document entity. This has two effects that may be important to users of XML processors:

- Certain well-formedness errors, specifically those that require reading external entities, may not be detected by a non-validating processor. Examples include the constraints entitled Entity Declared, Parsed Entity, and No Recursion, as well as some of the cases described as forbidden in **4.4 XML Processor Treatment of Entities and References**.
- The information passed from the processor to the application may vary, depending on whether the processor reads parameter and external entities. For example, a non-validating processor may not normalize attribute values, include the replacement text of internal entities, or supply default attribute values, where doing so depends on having read declarations in external or parameter entities.

For maximum reliability in interoperating between different XML processors, applications which use non-validating processors should not rely on any behaviors not required of such processors. Applications which require facilities such as the use of default attributes or internal entities which are declared in external entities should use validating XML processors.

6 Notation

The formal grammar of XML is given in this specification using a simple Extended Backus-Naur Form (EBNF) notation. Each rule in the grammar defines one symbol, in the form

```
symbol ::= expression
```

Symbols are written with an initial capital letter if they are the start symbol of a regular language, otherwise with an initial lower case letter. Literal strings are quoted.

Within the expression on the right-hand side of a rule, the following expressions are used to match strings of one or

more characters:

`#xN`

where `N` is a hexadecimal integer, the expression matches the character in ISO/IEC 10646 whose canonical (UCS-4) code value, when interpreted as an unsigned binary number, has the value indicated. The number of leading zeros in the `#xN` form is insignificant; the number of leading zeros in the corresponding code value is governed by the character encoding in use and is not significant for XML.

`[a-zA-Z]`, `[#xN-#xN]`

matches any Char with a value in the range(s) indicated (inclusive).

`[abc]`, `[#xN#xN#xN]`

matches any Char with a value among the characters enumerated. Enumerations and ranges can be mixed in one set of brackets.

`[^a-z]`, `[^#xN-#xN]`

matches any Char with a value *outside* the range indicated.

`[^abc]`, `[^#xN#xN#xN]`

matches any Char with a value not among the characters given. Enumerations and ranges of forbidden values can be mixed in one set of brackets.

`"string"`

matches a literal string matching that given inside the double quotes.

`'string'`

matches a literal string matching that given inside the single quotes.

These symbols may be combined to match more complex patterns as follows, where `A` and `B` represent simple expressions:

(`expression`)

`expression` is treated as a unit and may be combined as described in this list.

`A?`

matches `A` or nothing; optional `A`.

`A B`

matches `A` followed by `B`. This operator has higher precedence than alternation; thus `A B | C D` is identical to `(A B) | (C D)`.

`A | B`

matches `A` or `B` but not both.

`A - B`

matches any string that matches `A` but does not match `B`.

`A+`

matches one or more occurrences of `A`.Concatenation has higher precedence than alternation; thus `A+ | B+` is identical to `(A+) | (B+)`.

`A*`

matches zero or more occurrences of `A`. Concatenation has higher precedence than alternation; thus `A* | B*` is identical to `(A*) | (B*)`.

Other notations used in the productions are:

`/* ... */`

comment.

`[ wfc: ... ]`

well-formedness constraint; this identifies by name a constraint on well-formed documents associated with a production.

`[ vc: ... ]`

validity constraint; this identifies by name a constraint on valid documents associated with a production.

A References

A.1 Normative References

IANA-CHARSETS

(Internet Assigned Numbers Authority) Official Names for Character Sets, ed. Keld Simonsen et al. See ftp://ftp.isi.edu/in-notes/iana/assignments/character-sets.

IETF RFC 1766

IETF (Internet Engineering Task Force). RFC 1766: Tags for the Identification of Languages, ed. H. Alvestrand. 1995. (See http://www.ietf.org/rfc/rfc1766.txt.)

ISO/IEC 10646

ISO (International Organization for Standardization). ISO/IEC 10646-1993 (E). Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane. [Geneva]: International Organization for Standardization, 1993 (plus amendments AM 1 through AM 7).

ISO/IEC 10646-2000

ISO (International Organization for Standardization). ISO/IEC 10646-1:2000. Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane. [Geneva]: International Organization for Standardization, 2000.

Unicode

The Unicode Consortium. *The Unicode Standard, Version 2.0.* Reading, Mass.: Addison-Wesley Developers Press, 1996.

Unicode3

The Unicode Consortium. *The Unicode Standard, Version 3.0.* Reading, Mass.: Addison-Wesley Developers Press, 2000. ISBN 0-201-61633-5.

A.2 Other References

Aho/Ullman

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. Reading: Addison-Wesley, 1986, rpt. corr. 1988.

Berners-Lee et al.

Berners-Lee, T., R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax and Semantics. 1997. (Work in progress; see updates to RFC1738.)

Brüggemann-Klein

Brüggemann-Klein, Anne. Formal Models in Document Processing. Habilitationsschrift. Faculty of Mathematics at the University of Freiburg, 1993. (See ftp://ftp.informatik.uni-freiburg.de/documents/papers/brueggem/habil.ps.)

Brüggemann-Klein and Wood

Brüggemann-Klein, Anne, and Derick Wood. Deterministic Regular Languages. Universität Freiburg, Institut für Informatik, Bericht 38, Oktober 1991. Extended abstract in A. Finkel, M. Jantzen, Hrsg., STACS 1992, S. 173-184. Springer-Verlag, Berlin 1992. Lecture Notes in Computer Science 577. Full version titled

One-Unambiguous Regular Languages in Information and Computation 140 (2): 229-253, February 1998.

Clark

James Clark. Comparison of SGML and XML. See http://www.w3.org/TR/NOTE-sgml-xml-971215.

IANA-LANGCODES

(Internet Assigned Numbers Authority) Registry of Language Tags, ed. Keld Simonsen et al. (See http://www.isi.edu/in-notes/iana/assignments/languages/.)

IETF RFC2141

IETF (Internet Engineering Task Force). *RFC 2141: URN Syntax*, ed. R. Moats. 1997. (See http://www.ietf.org/rfc/rfc2141.txt.)

IETF RFC 2279

IETF (Internet Engineering Task Force). RFC 2279: UTF-8, a transformation format of ISO 10646, ed. F. Yergeau, 1998. (See http://www.ietf.org/rfc/rfc2279.txt.)

IETF RFC 2376

IETF (Internet Engineering Task Force). RFC 2376: XML Media Types. ed. E. Whitehead, M. Murata. 1998. (See http://www.ietf.org/rfc/rfc2376.txt.)

IETF RFC 2396

IETF (Internet Engineering Task Force). RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. T. Berners-Lee, R. Fielding, L. Masinter. 1998. (See http://www.ietf.org/rfc/rfc2396.txt.)

IETF RFC 2732

IETF (Internet Engineering Task Force). RFC 2732: Format for Literal IPv6 Addresses in URL's. R. Hinden, B. Carpenter, L. Masinter. 1999. (See http://www.ietf.org/rfc/rfc2732.txt.)

IETF RFC 2781

IETF (Internet Engineering Task Force). *RFC 2781: UTF-16, an encoding of ISO 10646*, ed. P. Hoffman, F. Yergeau. 2000. (See http://www.ietf.org/rfc/rfc2781.txt.)

ISO 639

(International Organization for Standardization). ISO 639:1988 (E). Code for the representation of names of languages. [Geneva]: International Organization for Standardization, 1988.

ISO 3166

(International Organization for Standardization). ISO 3166-1:1997 (E). Codes for the representation of names of countries and their subdivisions -- Part 1: Country codes [Geneva]: International Organization for Standardization, 1997.

ISO 8879

ISO (International Organization for Standardization). ISO 8879:1986(E). Information processing -- Text and Office Systems -- Standard Generalized Markup Language (SGML). First edition -- 1986-10-15. [Geneva]: International Organization for Standardization, 1986.

ISO/IEC 10744

ISO (International Organization for Standardization). ISO/IEC 10744-1992 (E). Information technology -- Hypermedia/Time-based Structuring Language (HyTime). [Geneva]: International Organization for Standardization, 1992. *Extended Facilities Annexe.* [Geneva]: International Organization for Standardization, 1996.

WEBSGML

ISO (International Organization for Standardization). ISO 8879:1986 TC2. Information technology -- Document Description and Processing Languages. [Geneva]: International Organization for Standardization, 1998. (See http://www.sgmlsource.com/8879rev/n0029.htm.)

XML Names

Tim Bray, Dave Hollander, and Andrew Layman, editors. Namespaces in XML. Textuality, Hewlett-Packard, and Microsoft. World Wide Web Consortium, 1999. (See http://www.w3.org/TR/REC-xml-names/.)

B Character Classes

Following the characteristics defined in the Unicode standard, characters are classed as base characters (among others, these contain the alphabetic characters of the Latin alphabet), ideographic characters, and combining characters (among others, this class contains most diacritics) Digits and extenders are also distinguished.

Characters

```
[84]  Letter        ::=  BaseChar | Ideographic
[85]  BaseChar      ::=  [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6]
                       | [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131]
                       | [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E]
                       | [#x0180-#x01C3] | [#x01CD-#x01F0] | [#x01F4-#x01F5]
                       | [#x01FA-#x0217] | [#x0250-#x02A8] | [#x02BB-#x02C1]
                       | #x0386 | [#x0388-#x038A] | #x038C | [#x038E-#x03A1]
                       | [#x03A3-#x03CE] | [#x03D0-#x03D6] | #x03DA | #x03DC
                       | #x03DE | #x03E0 | [#x03E2-#x03F3] | [#x0401-#x040C]
                       | [#x040E-#x044F] | [#x0451-#x045C] | [#x045E-#x0481]
                       | [#x0490-#x04C4] | [#x04C7-#x04C8] | [#x04CB-#x04CC]
                       | [#x04D0-#x04EB] | [#x04EE-#x04F5] | [#x04F8-#x04F9]
                       | [#x0531-#x0556] | #x0559 | [#x0561-#x0586]
                       | [#x05D0-#x05EA] | [#x05F0-#x05F2] | [#x0621-#x063A]
                       | [#x0641-#x064A] | [#x0671-#x06B7] | [#x06BA-#x06BE]
                       | [#x06C0-#x06CE] | [#x06D0-#x06D3] | #x06D5
                       | [#x06E5-#x06E6] | [#x0905-#x0939] | #x093D
                       | [#x0958-#x0961] | [#x0985-#x098C] | [#x098F-#x0990]
                       | [#x0993-#x09A8] | [#x09AA-#x09B0] | #x09B2
                       | [#x09B6-#x09B9] | [#x09DC-#x09DD] | [#x09DF-#x09E1]
                       | [#x09F0-#x09F1] | [#x0A05-#x0A0A] | [#x0A0F-#x0A10]
                       | [#x0A13-#x0A28] | [#x0A2A-#x0A30] | [#x0A32-#x0A33]
                       | [#x0A35-#x0A36] | [#x0A38-#x0A39] | [#x0A59-#x0A5C]
                       | #x0A5E | [#x0A72-#x0A74] | [#x0A85-#x0A8B] | #x0A8D
                       | [#x0A8F-#x0A91] | [#x0A93-#x0AA8] | [#x0AAA-#x0AB0]
                       | [#x0AB2-#x0AB3] | [#x0AB5-#x0AB9] | #x0ABD | #x0AE0
                       | [#x0B05-#x0B0C] | [#x0B0F-#x0B10] | [#x0B13-#x0B28]
                       | [#x0B2A-#x0B30] | [#x0B32-#x0B33] | [#x0B36-#x0B39]
                       | #x0B3D | [#x0B5C-#x0B5D] | [#x0B5F-#x0B61]
                       | [#x0B85-#x0B8A] | [#x0B8E-#x0B90] | [#x0B92-#x0B95]
                       | [#x0B99-#x0B9A] | #x0B9C | [#x0B9E-#x0B9F]
                       | [#x0BA3-#x0BA4] | [#x0BA8-#x0BAA] | [#x0BAE-#x0BB5]
                       | [#x0BB7-#x0BB9] | [#x0C05-#x0C0C] | [#x0C0E-#x0C10]
                       | [#x0C12-#x0C28] | [#x0C2A-#x0C33] | [#x0C35-#x0C39]
                       | [#x0C60-#x0C61] | [#x0C85-#x0C8C] | [#x0C8E-#x0C90]
                       | [#x0C92-#x0CA8] | [#x0CAA-#x0CB3] | [#x0CB5-#x0CB9]
                       | #x0CDE | [#x0CE0-#x0CE1] | [#x0D05-#x0D0C]
                       | [#x0D0E-#x0D10] | [#x0D12-#x0D28] | [#x0D2A-#x0D39]
                       | [#x0D60-#x0D61] | [#x0E01-#x0E2E] | #x0E30
                       | [#x0E32-#x0E33] | [#x0E40-#x0E45] | [#x0E81-#x0E82]
                       | #x0E84 | [#x0E87-#x0E88] | #x0E8A | #x0E8D
                       | [#x0E94-#x0E97] | [#x0E99-#x0E9F] | [#x0EA1-#x0EA3]
                       | #x0EA5 | #x0EA7 | [#x0EAA-#x0EAB] | [#x0EAD-#x0EAE]
                       | #x0EB0 | [#x0EB2-#x0EB3] | #x0EBD | [#x0EC0-#x0EC4]
                       | [#x0F40-#x0F47] | [#x0F49-#x0F69] | [#x10A0-#x10C5]
                       | [#x10D0-#x10F6] | #x1100 | [#x1102-#x1103]
                       | [#x1105-#x1107] | #x1109 | [#x110B-#x110C]
```

```
                        | [#x110E-#x1112] | #x113C | #x113E | #x1140 | #x114C
                        | #x114E | #x1150 | [#x1154-#x1155] | #x1159
                        | [#x115F-#x1161] | #x1163 | #x1165 | #x1167 | #x1169
                        | [#x116D-#x116E] | [#x1172-#x1173] | #x1175 | #x119E
                        | #x11A8 | #x11AB | [#x11AE-#x11AF] | [#x11B7-#x11B8]
                        | #x11BA | [#x11BC-#x11C2] | #x11EB | #x11F0 | #x11F9
                        | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9] | [#x1F00-#x1F15]
                        | [#x1F18-#x1F1D] | [#x1F20-#x1F45] | [#x1F48-#x1F4D]
                        | [#x1F50-#x1F57] | #x1F59 | #x1F5B | #x1F5D
                        | [#x1F5F-#x1F7D] | [#x1F80-#x1FB4] | [#x1FB6-#x1FBC]
                        | #x1FBE | [#x1FC2-#x1FC4] | [#x1FC6-#x1FCC]
                        | [#x1FD0-#x1FD3] | [#x1FD6-#x1FDB] | [#x1FE0-#x1FEC]
                        | [#x1FF2-#x1FF4] | [#x1FF6-#x1FFC] | #x2126
                        | [#x212A-#x212B] | #x212E | [#x2180-#x2182]
                        | [#x3041-#x3094] | [#x30A1-#x30FA] | [#x3105-#x312C]
                        | [#xAC00-#xD7A3]
[86] Ideographic    ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]
[87] CombiningChar  ::= [#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
                        | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD]
                        | #x05BF | [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652]
                        | #x0670 | [#x06D6-#x06DC] | [#x06DD-#x06DF]
                        | [#x06E0-#x06E4] | [#x06E7-#x06E8] | [#x06EA-#x06ED]
                        | [#x0901-#x0903] | #x093C | [#x093E-#x094C] | #x094D
                        | [#x0951-#x0954] | [#x0962-#x0963] | [#x0981-#x0983]
                        | #x09BC | #x09BE | #x09BF | [#x09C0-#x09C4]
                        | [#x09C7-#x09C8] | [#x09CB-#x09CD] | #x09D7
                        | [#x09E2-#x09E3] | #x0A02 | #x0A3C | #x0A3E | #x0A3F
                        | [#x0A40-#x0A42] | [#x0A47-#x0A48] | [#x0A4B-#x0A4D]
                        | [#x0A70-#x0A71] | [#x0A81-#x0A83] | #x0ABC
                        | [#x0ABE-#x0AC5] | [#x0AC7-#x0AC9] | [#x0ACB-#x0ACD]
                        | [#x0B01-#x0B03] | #x0B3C | [#x0B3E-#x0B43]
                        | [#x0B47-#x0B48] | [#x0B4B-#x0B4D] | [#x0B56-#x0B57]
                        | [#x0B82-#x0B83] | [#x0BBE-#x0BC2] | [#x0BC6-#x0BC8]
                        | [#x0BCA-#x0BCD] | #x0BD7 | [#x0C01-#x0C03]
                        | [#x0C3E-#x0C44] | [#x0C46-#x0C48] | [#x0C4A-#x0C4D]
                        | [#x0C55-#x0C56] | [#x0C82-#x0C83] | [#x0CBE-#x0CC4]
                        | [#x0CC6-#x0CC8] | [#x0CCA-#x0CCD] | [#x0CD5-#x0CD6]
                        | [#x0D02-#x0D03] | [#x0D3E-#x0D43] | [#x0D46-#x0D48]
                        | [#x0D4A-#x0D4D] | #x0D57 | #x0E31 | [#x0E34-#x0E3A]
                        | [#x0E47-#x0E4E] | #x0EB1 | [#x0EB4-#x0EB9]
                        | [#x0EBB-#x0EBC] | [#x0EC8-#x0ECD] | [#x0F18-#x0F19]
                        | #x0F35 | #x0F37 | #x0F39 | #x0F3E | #x0F3F
                        | [#x0F71-#x0F84] | [#x0F86-#x0F8B] | [#x0F90-#x0F95]
                        | #x0F97 | [#x0F99-#x0FAD] | [#x0FB1-#x0FB7] | #x0FB9
                        | [#x20D0-#x20DC] | #x20E1 | [#x302A-#x302F] | #x3099
                        | #x309A
[88] Digit          ::= [#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9]
                        | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F]
                        | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF]
                        | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F]
                        | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29]
[89] Extender       ::= #x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640 | #x0E46
                        | #x0EC6 | #x3005 | [#x3031-#x3035] | [#x309D-#x309E]
                        | [#x30FC-#x30FE]
```

The character classes defined here can be derived from the Unicode 2.0 character database as follows:

- Name start characters must have one of the categories Ll, Lu, Lo, Lt, Nl.
- Name characters other than Name-start characters must have one of the categories Mc, Me, Mn, Lm, or Nd.
- Characters in the compatibility area (i.e. with character code greater than #xF900 and less than #xFFFE) are not allowed in XML names.
- Characters which have a font or compatibility decomposition (i.e. those with a "compatibility formatting tag" in field 5 of the database -- marked by field 5 beginning with a "<") are not allowed.
- The following characters are treated as name-start characters rather than name characters, because the property file classifies them as Alphabetic: [#x02BB-#x02C1], #x0559, #x06E5, #x06E6.
- Characters #x20DD-#x20E0 are excluded (in accordance with Unicode 2.0, section 5.14).
- Character #x00B7 is classified as an extender, because the property list so identifies it.
- Character #x0387 is added as a name character, because #x00B7 is its canonical equivalent.
- Characters ':' and '_' are allowed as name-start characters.
- Characters '-' and '.' are allowed as name characters.

C XML and SGML (Non-Normative)

XML is designed to be a subset of SGML, in that every XML document should also be a conforming SGML document. For a detailed comparison of the additional restrictions that XML places on documents beyond those of SGML, see [Clark].

D Expansion of Entity and Character References (Non-Normative)

This appendix contains some examples illustrating the sequence of entity- and character-reference recognition and expansion, as specified in **4.4 XML Processor Treatment of Entities and References**.

If the DTD contains the declaration

```
<!ENTITY example "<p>An ampersand (&#38;#38;) may be escaped
numerically (&#38;#38;#38;) or with a general entity
(&amp;amp;).</p>" >
```

then the XML processor will recognize the character references when it parses the entity declaration, and resolve them before storing the following string as the value of the entity "`example`":

```
<p>An ampersand (&#38;) may be escaped
numerically (&#38;#38;) or with a general entity
(&amp;amp;).</p>
```

A reference in the document to "`&example;`" will cause the text to be reparsed, at which time the start- and end-tags of the `p` element will be recognized and the three references will be recognized and expanded, resulting in a `p` element with the following content (all data, no delimiters or markup):

```
An ampersand (&) may be escaped
numerically (&#38;) or with a general entity
(&amp;).
```

A more complex example will illustrate the rules and their effects fully. In the following example, the line numbers are solely for reference.

```
1 <?xml version='1.0'?>
2 <!DOCTYPE test [
3 <!ELEMENT test (#PCDATA) >
4 <!ENTITY % xx '&#37;zz;'>
5 <!ENTITY % zz '&#60;!ENTITY tricky "error-prone" >' >
6 %xx;
7 ]>
8 <test>This sample shows a &tricky; method.</test>
```

This produces the following:

- in line 4, the reference to character 37 is expanded immediately, and the parameter entity "`xx`" is stored in the symbol table with the value "`%zz;`". Since the replacement text is not rescanned, the reference to parameter entity "`zz`" is not recognized. (And it would be an error if it were, since "`zz`" is not yet declared.)
- in line 5, the character reference "`&#60;`" is expanded immediately and the parameter entity "`zz`" is stored with the replacement text "`<!ENTITY tricky "error-prone" >`", which is a well-formed entity declaration.
- in line 6, the reference to "`xx`" is recognized, and the replacement text of "`xx`" (namely "`%zz;`") is parsed. The reference to "`zz`" is recognized in its turn, and its replacement text ("`<!ENTITY tricky "error-prone" >`") is parsed. The general entity "`tricky`" has now been declared, with the replacement text "`error-prone`".
- in line 8, the reference to the general entity "`tricky`" is recognized, and it is expanded, so the full content of the `test` element is the self-describing (and ungrammatical) string *This sample shows a error-prone method.*

E Deterministic Content Models (Non-Normative)

As noted in **3.2.1 Element Content**, it is required that content models in element type declarations be deterministic. This requirement is for compatibility with SGML (which calls deterministic content models "unambiguous"); XML processors built using SGML systems may flag non-deterministic content models as errors.

For example, the content model `((b, c) | (b, d))` is non-deterministic, because given an initial `b` the XML processor cannot know which `b` in the model is being matched without looking ahead to see which element follows the `b`. In this case, the two references to `b` can be collapsed into a single reference, making the model read `(b, (c | d))`. An initial `b` now clearly matches only a single name in the content model. The processor doesn't need to look ahead to see what follows; either `c` or `d` would be accepted.

More formally: a finite state automaton may be constructed from the content model using the standard algorithms, e.g. algorithm 3.5 in section 3.9 of Aho, Sethi, and Ullman [Aho/Ullman]. In many such algorithms, a follow set is constructed for each position in the regular expression (i.e., each leaf node in the syntax tree for the regular expression); if any position has a follow set in which more than one following position is labeled with the same element type name, then the content model is in error and may be reported as an error.

Algorithms exist which allow many but not all non-deterministic content models to be reduced automatically to equivalent deterministic models; see Brüggemann-Klein 1991 [Brüggemann-Klein].

F Autodetection of Character Encodings (Non-Normative)

The XML encoding declaration functions as an internal label on each entity, indicating which character encoding is in use. Before an XML processor can read the internal label, however, it apparently has to know what character encoding is in use--which is what the internal label is trying to indicate. In the general case, this is a hopeless situation. It is not entirely hopeless in XML, however, because XML limits the general case in two ways: each

implementation is assumed to support only a finite set of character encodings, and the XML encoding declaration is restricted in position and content in order to make it feasible to autodetect the character encoding in use in each entity in normal cases. Also, in many cases other sources of information are available in addition to the XML data stream itself. Two cases may be distinguished, depending on whether the XML entity is presented to the processor without, or with, any accompanying (external) information. We consider the first case first.

F.1 Detection Without External Encoding Information

Because each XML entity not accompanied by external encoding information and not in UTF-8 or UTF-16 encoding *must* begin with an XML encoding declaration, in which the first characters must be '`<?xml`', any conforming processor can detect, after two to four octets of input, which of the following cases apply. In reading this list, it may help to know that in UCS-4, '<' is "`#x0000003C`" and '?' is "`#x0000003F`", and the Byte Order Mark required of UTF-16 data streams is "`#xFEFF`". The notation ## is used to denote any byte value except that two consecutive ##s cannot be both 00.

With a Byte Order Mark:

`00 00 FE FF`	UCS-4, big-endian machine (1234 order)
`FF FE 00 00`	UCS-4, little-endian machine (4321 order)
`00 00 FF FE`	UCS-4, unusual octet order (2143)
`FE FF 00 00`	UCS-4, unusual octet order (3412)
`FE FF ## ##`	UTF-16, big-endian
`FF FE ## ##`	UTF-16, little-endian
`EF BB BF`	UTF-8

Without a Byte Order Mark:

`00 00 00 3C`
`3C 00 00 00`
`00 00 3C 00`
`00 3C 00 00`
	UCS-4 or other encoding with a 32-bit code unit and ASCII characters encoded as ASCII values, in respectively big-endian (1234), little-endian (4321) and two unusual byte orders (2143 and 3412). The encoding declaration must be read to determine which of UCS-4 or other supported 32-bit encodings applies.

`00 3C 00 3F`	UTF-16BE or big-endian ISO-10646-UCS-2 or other encoding with a 16-bit code unit in big-endian order and ASCII characters encoded as ASCII values (the encoding declaration must be read to determine which)
`3C 00 3F 00`	UTF-16LE or little-endian ISO-10646-UCS-2 or other encoding with a 16-bit code unit in little-endian order and ASCII characters encoded as ASCII values (the encoding declaration must be read to determine which)
`3C 3F 78 6D`	UTF-8, ISO 646, ASCII, some part of ISO 8859, Shift-JIS, EUC, or any other 7-bit, 8-bit, or mixed-width encoding which ensures that the characters of ASCII have their normal positions, width, and values; the actual encoding declaration must be read to detect which of these applies, but since all of these encodings use the same bit patterns for the relevant ASCII characters, the encoding declaration itself may be read reliably
`4C 6F A7 94`	EBCDIC (in some flavor; the full encoding declaration must be read to tell which code page is in use)
Other	UTF-8 without an encoding declaration, or else the data stream is mislabeled (lacking a required encoding declaration), corrupt, fragmentary, or enclosed in a wrapper of some kind

Note:

In cases above which do not require reading the encoding declaration to determine the encoding, section 4.3.3 still requires that the encoding declaration, if present, be read and that the encoding name be checked to match the actual encoding of the entity. Also, it is possible that new character encodings will be invented that will make it necessary to use the encoding declaration to determine the encoding, in cases where this is not required at present.

This level of autodetection is enough to read the XML encoding declaration and parse the character-encoding identifier, which is still necessary to distinguish the individual members of each family of encodings (e.g. to tell UTF-8 from 8859, and the parts of 8859 from each other, or to distinguish the specific EBCDIC code page in use, and so on).

Because the contents of the encoding declaration are restricted to characters from the ASCII repertoire (however encoded), a processor can reliably read the entire encoding declaration as soon as it has detected which family of encodings is in use. Since in practice, all widely used character encodings fall into one of the categories above, the XML encoding declaration allows reasonably reliable in-band labeling of character encodings, even when external sources of information at the operating-system or transport-protocol level are unreliable. Character encodings such as UTF-7 that make overloaded usage of ASCII-valued bytes may fail to be reliably detected.

Once the processor has detected the character encoding in use, it can act appropriately, whether by invoking a separate input routine for each case, or by calling the proper conversion function on each character of input.

Like any self-labeling system, the XML encoding declaration will not work if any software changes the entity's character set or encoding without updating the encoding declaration. Implementors of character-encoding routines should be careful to ensure the accuracy of the internal and external information used to label the entity.

F.2 Priorities in the Presence of External Encoding Information

The second possible case occurs when the XML entity is accompanied by encoding information, as in some file systems and some network protocols. When multiple sources of information are available, their relative priority and the preferred method of handling conflict should be specified as part of the higher-level protocol used to deliver XML. In particular, please refer to [IETF RFC 2376] or its successor, which defines the `text/xml` and `application/xml` MIME types and provides some useful guidance. In the interests of interoperability, however, the following rule is recommended.

- If an XML entity is in a file, the Byte-Order Mark and encoding declaration are used (if present) to determine the character encoding.

G W3C XML Working Group (Non-Normative)

This specification was prepared and approved for publication by the W3C XML Working Group (WG). WG approval of this specification does not necessarily imply that all WG members voted for its approval. The current and former members of the XML WG are:

- Jon Bosak, Sun (*Chair*)
- James Clark (*Technical Lead*)
- Tim Bray, Textuality and Netscape (*XML Co-editor*)
- Jean Paoli, Microsoft (*XML Co-editor*)
- C. M. Sperberg-McQueen, U. of Ill. (*XML Co-editor*)
- Dan Connolly, W3C (*W3C Liaison*)
- Paula Angerstein, Texcel
- Steve DeRose, INSO
- Dave Hollander, HP
- Eliot Kimber, ISOGEN
- Eve Maler, ArborText
- Tom Magliery, NCSA
- Murray Maloney, SoftQuad, Grif SA, Muzmo and Veo Systems
- MURATA Makoto (FAMILY Given), Fuji Xerox Information Systems
- Joel Nava, Adobe
- Conleth O'Connell, Vignette
- Peter Sharpe, SoftQuad

- John Tigue, DataChannel

H W3C XML Core Group (Non-Normative)

The second edition of this specification was prepared by the W3C XML Core Working Group (WG). The members of the WG at the time of publication of this edition were:

- Paula Angerstein, Vignette
- Daniel Austin, Ask Jeeves
- Tim Boland
- Allen Brown, Microsoft
- Dan Connolly, W3C (*Staff Contact*)
- John Cowan, Reuters Limited
- John Evdemon, XMLSolutions Corporation
- Paul Grosso, Arbortext (*Co-Chair*)
- Arnaud Le Hors, IBM (*Co-Chair*)
- Eve Maler, Sun Microsystems (*Second Edition Editor*)
- Jonathan Marsh, Microsoft
- MURATA Makoto (FAMILY Given), IBM
- Mark Needleman, Data Research Associates
- David Orchard, Jamcracker
- Lew Shannon, NCR
- Richard Tobin, University of Edinburgh
- Daniel Veillard, W3C
- Dan Vint, Lexica
- Norman Walsh, Sun Microsystems
- François Yergeau, Alis Technologies (*Errata List Editor*)
- Kongyi Zhou, Oracle

I Production Notes (Non-Normative)

This Second Edition was encoded in the XMLspec DTD (which has documentation available). The HTML versions were produced with a combination of the xmlspec.xsl, diffspec.xsl, and REC-xml-2e.xsl XSLT stylesheets. The PDF version was produced with the html2ps facility and a distiller program.

XML Blueberry Requirements

W3C Working Draft 21 September 2001

This version:
http://www.w3.org/TR/2001/WD-xml-blueberry-req-20010921
Latest version:
http://www.w3.org/TR/xml-blueberry-req
Previous Version:
http://www.w3.org/TR/2001/WD-xml-blueberry-req-20010620
Editor:
John Cowan, Reuters *(jcowan@reutershealth.com)*

Abstract

This document lists the design principles and requirements for the Blueberry revision of the XML Recommendation, a limited revision of XML 1.0 being developed by the World Wide Web Consortium's XML Core Working Group solely to address character set issues.

Status of this document

This is a W3C Working Draft produced as a deliverable of the XML Core WG according to its charter and the current XML Activity process. A list of current W3C working drafts and notes can be found at http://www.w3.org/TR .

This document is a work in progress representing the current consensus of the W3C XML Core Working Group. It is published for review by W3C members and other interested parties. Publication as a Working Draft does not imply endorsement by the W3C membership. Comments should be sent to www-xml-blueberry-comments@w3.org , which is an automatically and publicly archived email list .

Table of Contents

1. Introduction

The W3C's XML 1.0 Recommendation [XML] was first issued in 1998, and despite the issuance of many errata culminating in a Second Edition of 2001, has remained (by intention) unchanged with respect to what is well-formed XML and what is not. This stability has been extremely useful for interoperability. However, the Unicode Standard [Unicode] on which XML 1.0 relies has not remained static, evolving from version 2.0 to version 3.1. Characters present in Unicode 3.1 but not in Unicode 2.0 may be used in XML character data. However, they are not allowed in XML names such as element type names, attribute names, enumerated attribute values, processing instruction targets, and so on. In addition, some characters that should have been permitted in XML names were not, due to oversights and inconsistencies in Unicode 2.0.

As a result, fully native-language XML markup is not possible in *at least* the following languages: Amharic, Burmese, Canadian aboriginal languages, Cherokee, Dhivehi, Hakka Chinese (Bopomofo script), Khmer, Minnan Chinese (Bopomofo script), Mongolian (traditional script), Oromo, Syriac, Tigre, and Yi, because the characters required to write these languages did not exist in Unicode 2.0. In addition, Chinese (particularly as used in Hong Kong) and Japanese can make use in XML names of only a subset of their complete character repertoires.

The point has been made that many of these languages can be written using other scripts, notably the Latin script, which makes transliterated native markup possible. However, exactly the same argument applies to many languages (for example, Greek) that were already fully encoded in Unicode 2.0. Discriminating against languages simply because their scripts were not encoded in Unicode 2.0 is inherently unjust. In addition, working with transliteration is far more painful for native readers and writers than working with the native script.

In addition, XML 1.0 attempts to adapt to the line-end conventions of various modern operating systems, but discriminates against the conventions used on IBM and IBM-compatible mainframes. As a result, XML documents on mainframes are not plain text files according to the local conventions. XML 1.0 documents generated on mainframes must either violate the local line-end conventions, or employ otherwise unnecessary translation phases before parsing and after generation. Allowing straightforward interoperability is particularly important when data stores are shared between mainframe and non-mainframe systems (as opposed to being copied from one to the other).

A new XML version, rather than a set of errata to XML 1.0, is being created because the change affects the definition of well-formed documents. XML 1.0 processors must continue to reject documents that contain new characters in XML names or new line-end conventions. It is presumed that the distinction between XML 1.0 and XML Blueberry will be indicated by the XML declaration.

2. Design Principles

1. The XML 1.0 goals listed in section 1.1 of the XML Recommendation are reaffirmed.

2. XML Blueberry documents shall permit the full and straightforward use of writing systems supported by Unicode 3.1.

3. XML Blueberry documents shall permit the full and straightforward use of operating environments that support Unicode 3.1.

4. The changes required for XML 1.0 processors to also process XML Blueberry shall be as few and as small as possible.

3. Requirements

1. XML Blueberry documents shall allow the use within XML names of all Unicode 3.1 characters, insofar as appropriate for XML.

2. XML Blueberry documents shall support the line-end conventions associated with Unicode 3.1, insofar as appropriate for XML.

3. The working group shall consider the issue of future updates to Unicode.

4. The working group shall consider the issue of W3C normalization as expressed in the W3C Character Model [CharMod].

5. In creating XML Blueberry, the working group shall not consider any revisions to XML 1.0 except those needed to accomplish these requirements.

4. References

CharMod
W3C (World Wide Web Consortium). *Character Model for the World Wide Web* (work in progress). [Cambridge, MA]. http://www.w3.org/TR/charmod

XML
W3C (World Wide Web Consortium). *Extensible Markup Language (XML) Recommendation.* Version 1.0, 2nd edition. [Cambridge, MA]. http://www.w3.org/TR/REC-xml

Unicode
The Unicode Consortium. *The Unicode Standard, Version 3.1.* [Reading, MA: Addison-Wesley Developers Press, 2000]. http://www.unicode.org