Extensible Markup Language (XML) 1.1 (Second Edition)

W3C Recommendation 16 August 2006

This version:

http://www.w3.org/TR/2006/REC-xml11-20060816

Latest version:

http://www.w3.org/TR/xml11

Previous version:

http://www.w3.org/TR/2006/PER-xml11-20060614

Authors and Contributors:

Tim Bray (Textuality and Netscape) <tbray@textuality.com>
Jean Paoli (Microsoft) <jeanpa@microsoft.com>
C. M. Sperberg-McQueen (W3C) <cmsmcq@w3.org>
Eve Maler (Sun Microsystems, Inc.) <eve.maler@east.sun.com>
François Yergeau
John Cowan <cowan@ccil.org>

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 section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

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 XML Core Working Group as part of the XML Activity.

On 29 September 2006 this document was edited in place to remove a number of spurious and potentially misleading spaces.

The English version of this specification is the only normative version. However, for translations of this document, see http://www.w3.org/2003/03/Translations/byTechnology?technology=xml11.

This document is a W3C Recommendation. This second edition is *not* a new version of XML. As a convenience to readers, it incorporates the changes dictated by the accumulated errata (available at http://www.w3.org/XML/xml-V11-1e-errata) to the First Edition of XML 1.1, dated 4 February 2004. In addition, the markup introduced to clarify when prescriptive keywords are used in the formal sense defined in [IETF RFC 2119], has been modified to better match the intent of [IETF RFC 2119]. This edition supersedes the previous W3C Recommendation of 4 February 2004.

Please report errors in this document to the public xml-editor@w3.org mailing list; archives are available. For the convenience of readers, an XHTML version with color-coded revision indicators is also provided; this version highlights each change due to an erratum published in the errata list, together with a link to the particular erratum in that list. Most of the errata in the list provide a rationale for the change. The errata list for this second edition is available at http://www.w3.org/XML/xml-V11-2e-errata.

An implementation report is available at http://www.w3.org/XML/2006/06/xml11-2e-implementation.html. A Test Suite is maintained to help assessing conformance to this specification.

This document has been reviewed by W3C Members, by software developers, and by other W3C groups and interested parties, and is endorsed by the Director as a W3C Recommendation. It is a stable document and may be used as reference material or cited 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 is governed by the 24 January 2002 CPP as amended by the W3C Patent Policy Transition Procedure. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

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.

A software module called an *XML processor* is used to read XML documents and provide access to their content and structure. 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 Working Group'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 [Unicode] and ISO/IEC 10646 [ISO/IEC 10646] for characters, Internet RFC 3066 [IETF RFC 3066] for language identification tags, ISO 639 [ISO 639] for language name codes, and ISO 3166 [ISO 3166] for country name codes), provides all the information necessary to understand XML Version 1.1 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 key words MUST, MUST NOT, REQUIRED, SHALL, SHALL NOT, SHOULD, SHOULD NOT, RECOMMENDED, MAY, and OPTIONAL, when EMPHASIZED, are to be interpreted as described in [IETF RFC 2119]. In addition, the terms defined in the following list are used in building those definitions and in describing the actions of an XML processor:

error

A violation of the rules of this specification; results are undefined. Unless otherwise specified, failure to observe a prescription of this specification indicated by one of the keywords MUST, REQUIRED, MUST NOT, SHALL and SHALL NOT is an error. Conforming software MAY detect and report an error and MAY recover from it.

fatal error

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

(Of strings or names:) Two strings or names being compared are identical. Characters with multiple possible representations in Unicode (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 "Element Valid".

for compatibility

Marks a sentence describing a feature of XML included solely to ensure that XML remains compatible with SGML.

for interoperability

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.

1.3. Rationale and list of changes for XML 1.1

The W3C's XML 1.0 Recommendation was first issued in 1998, and despite the issuance of many errata culminating in a Third Edition of 2004, 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 on which XML 1.0 relies for character specifications has not remained static, evolving from version 2.0 to version 4.0 and beyond. Characters not present in Unicode 2.0 may already be used in XML 1.0 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.

The overall philosophy of names has changed since XML 1.0. Whereas XML 1.0 provided a rigid definition of names, wherein everything that was not permitted was forbidden, XML 1.1 names are designed so that everything that is not forbidden (for a specific reason) is permitted. Since Unicode will continue to grow past version 4.0, further changes to XML can be avoided by allowing almost any character, including those not yet assigned, in names.

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). Therefore XML 1.1 adds NEL (#x85) to the list of line-end characters. For completeness, the Unicode line separator character, #x2028, is also supported.

Finally, there is considerable demand to define a standard representation of arbitrary Unicode characters in XML documents. Therefore, XML 1.1 allows the use of character references to the control characters #x1 through #x1F, most of which are forbidden in XML 1.0. For reasons of robustness, however, these characters still cannot be used directly in documents. In order to improve the robustness of character encoding detection, the additional control characters #x7F through #x9F, which were freely allowed in XML 1.0 documents, now must also appear only as character references. (Whitespace characters are of course exempt.) The minor sacrifice of backward compatibility is considered not significant. Due to potential problems with APIs, #x0 is still forbidden both directly and as a character reference.

Finally, XML 1.1 defines a set of constraints called "full normalization" on XML documents, which document creators SHOULD adhere to, and document processors SHOULD verify. Using fully normalized documents ensures that identity comparisons of names, attribute values, and character content can be made correctly by simple binary comparison of Unicode strings.

A new XML version, rather than a set of errata to XML 1.0, is being created because the changes affect the definition of well-formed documents. XML 1.0 processors must continue to reject documents that contain new characters in XML names, new line-end conventions, and references to control characters. The distinction between XML 1.0 and XML 1.1 documents is indicated by the version number information in the XML declaration at the start of each document.

2. Documents

A data object is an *XML document* if it is well-formed, as defined in this specification. In addition, the XML document is 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 on page 28.

2.1. Well-Formed XML Documents

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***) - (**Char*** **RestrictedChar Char***)

Matching the **document** production implies that:

1. It contains one or more elements.
2. 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.

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

A parsed entity contains *text*, a sequence of characters, which may represent markup or character data. 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 characters of Unicode and ISO/IEC 10646. The versions of these standards cited in Appendix A.1 – Normative References on page 37 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**.

Character Range

[2] Char ::= [#x1-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF] /* any Unicode character, excluding the surro-

gate
blocks,
FFFE,
and
FFFF.
*/

[3] RestrictedChar ::= [#x1-#x8] | [#xB-#xC] | [#xE-#x1F] | [#x7F-#x84] | [#x86-#x9F]

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 Unicode [Unicode]; 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 on page 28.

Document authors are encouraged to avoid “compatibility characters”, as defined in Unicode [Unicode]. The characters defined in the following ranges are also discouraged. They are either control characters or permanently undefined Unicode characters:

```
[#x1-#x8], [#xB-#xC], [#xE-#x1F], [#x7F-#x84], [#x86-#x9F], [#xFDD0-#xFDDF],
[#x1FFFE-#x1FFFF], [#x2FFFE-#x2FFFF], [#x3FFFE-#x3FFFF],
[#x4FFFE-#x4FFFF], [#x5FFFE-#x5FFFF], [#x6FFFE-#x6FFFF],
[#x7FFFE-#x7FFFF], [#x8FFFE-#x8FFFF], [#x9FFFE-#x9FFFF],
[#xAFFFE-#xAFFFF], [#xBFFFE-#xBFFFF], [#xCFFFE-#xCFFFF],
[#xDFFFE-#xDFFFF], [#xEFFFE-#xEFFFF], [#xFFFFE-#xFFFFF],
[#x10FFFE-#x10FFFF].
```

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

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

The presence of #xD in the above production is maintained purely for backward compatibility with the First Edition. As explained in § 2.11 – End-of-Line Handling on page 12, all #xD characters literally present in an XML document are either removed or replaced by #xA characters before any other processing is done. The only way to get a #xD character to match this production is to use a character reference in an entity value literal.

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 with any string which would match `(('X'|'x') ('M'|'m') ('L'|'l'))`, are reserved for standardization in this or future versions of this specification.

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.

The first character of a Name MUST be a NameStartChar, and any other characters MUST be NameChars; this mechanism is used to prevent names from beginning with European (ASCII) digits or with basic combining characters. Almost all characters are permitted in names, except those which either are or reasonably could be used as delimiters. The intention is to be inclusive rather than exclusive, so that writing systems not yet encoded in Unicode can be used in XML names. See Appendix I – Suggestions for XML Names on page 44 for suggestions on the creation of names.

Document authors are encouraged to use names which are meaningful words or combinations of words in natural languages, and to avoid symbolic or white space characters in names. Note that COLON, HYPHEN-MINUS, FULL STOP (period), LOW LINE (underscore), and MIDDLE DOT are explicitly permitted.

The ASCII symbols and punctuation marks, along with a fairly large group of Unicode symbol characters, are excluded from names because they are more useful as delimiters in contexts where XML names are used outside XML documents; providing this group gives those contexts hard guarantees about what *cannot* be part of an XML name. The character #x037E, GREEK QUESTION MARK, is excluded because when normalized it becomes a semicolon, which could change the meaning of entity references.

Names and Tokens

```
[5]     NameStartChar ::= ":" | [A-Z] | "_" | [a-z] | [#xC0-#xD6] | [#xD8-#xF6] | [#xF8-#x2FF] |
                          [#x370-#x37D] | [#x37F-#x1FFF] | [#x200C-#x200D] | [#x2070-#x218F]
                          | [#x2C00-#x2FEF] | [#x3001-#xD7FF] | [#xF900-#xFDCF] | [#xFDF0-
                          #xFFFD] | [#x10000-#xEFFFF]
[6]          NameChar ::= NameStartChar | "-" | "." | [0-9] | #xB7 | [#x0300-#x036F] | [#x203F-
                          #x2040]
[7]              Name ::= NameStartChar (NameChar)*
[8]             Names ::= Name (#x20 Name)*
[9]           Nmtoken ::= (NameChar)+
[10]         Nmtokens ::= Nmtoken (#x20 Nmtoken)*
```

☞ The **Names** and **Nmtokens** productions are used to define the validity of tokenized attribute values after normalization (see § 3.3.1 – Attribute Types on page 19).

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

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

☞ Although the **EntityValue** production allows the definition of a general 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. *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).

All text that is not markup constitutes the *character data* of the document.

The ampersand character (&) and the left angle bracket (<) MUST NOT appear in their literal form, except 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 "`&amp;`" and "`&lt;`" respectively. The right angle bracket (>) may be represented using the string "`&gt;`", and MUST, for compatibility, be escaped using either "`&gt;`" 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 or the CDATA-section-close delimiter, "`]]>`". 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

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

2.5. Comments

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 MUST NOT be recognized within comments.

Comments

[17]	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

Processing instructions (PIs) allow documents to contain instructions for applications.

Processing Instructions

[18]	PI	::=	'<?' **PITarget** (**S** (**Char*** - (**Char*** '?>' **Char***)))? '?>'
[19]	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 MUST NOT be recognized within processing instructions.

2.7. CDATA Sections

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

[20]	CDSect	::=	**CDStart** **CData** **CDEnd**
[21]	CDStart	::=	'<![CDATA['
[22]	CData	::=	(**Char*** - (**Char*** ']]>' **Char***))
[23]	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 “`&lt;`” and “`&amp;`”. 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

XML 1.1 documents MUST begin with an *XML declaration* which specifies the version of XML being used. For example, the following is a complete XML 1.1 document, well-formed but not valid:

```
<?xml version="1.1"?>
<greeting>Hello, world!</greeting>
```

but the following is an XML 1.0 document because it does not have an XML declaration:

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

The function of the markup in an XML document is to describe its storage and logical structure and to associate attribute name-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. 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

```
[24]        prolog ::= XMLDecl Misc* (doctypedecl Misc*)?
[25]       XMLDecl ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[26]   VersionInfo ::= S 'version' Eq ("'" VersionNum "'" | '"' VersionNum '"')
[27]            Eq ::= S? '=' S?
[28]    VersionNum ::= '1.1'
[29]          Misc ::= Comment | PI | S
```

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.

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 on page 24.

Document Type Definition

```
[30]   doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('[' intSubset ']' S?)? '>'
[31]      DeclSep ::= PEReference | S
[32]    intSubset ::= (markupdecl | DeclSep)*
[33]   markupdecl ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment
```

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 on page 23). 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 MUST NOT occur within markup declarations; they may occur where markup declarations can occur. (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 but is allowed in external parameter entities referenced in the internal subset.

External Subset

[34]	extSubset	::=	**TextDecl**? **extSubsetDecl**
[35]	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="1.1"?>
<!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="1.1" encoding="UTF-8" ?>
<!DOCTYPE greeting [
<!ELEMENT greeting (#PCDATA)>
]>
<greeting>Hello, world!</greeting>
```

If both the external and internal subsets are used, the internal subset MUST be 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.

If a document is well-formed or valid XML 1.0, and provided it does not contain any control characters in the range [#x7F-#x9F] other than as character escapes, it may be made well-formed or valid XML 1.1 respectively simply by changing the version number.

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. 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

[36]	SDDecl	::=	**S** 'standalone' **Eq** (("'" ('yes' | 'no') "'") | ('"' ('yes' | 'no') '"'))

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 tokenized types, where the attribute appears in the document with a value such that *normalization* will produce a different value from that which would be produced in the absence of the declaration, 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="1.1" 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 overridden with another instance of the xml:space attribute. This specification does not give meaning to any value of xml:space other than default and preserve. It is an error for other values to be specified; the XML processor MAY report the error or MAY recover by ignoring the attribute specification or by reporting the (erroneous) value to the application. Applications may ignore or reject erroneous values.

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 XML processor MUST behave as if it normalized all line breaks in external parsed entities (including the document entity) on input, before parsing, by translating all of the following to a single #xA character:

1. the two-character sequence #xD #xA
2. the two-character sequence #xD #x85
3. the single character #x85
4. the single character #x2028
5. any #xD character that is not immediately followed by #xA or #x85.

The characters #x85 and #x2028 cannot be reliably recognized and translated until an entity's encoding declaration (if present) has been read. Therefore, it is a fatal error to use them within the XML declaration or text declaration.

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 3066], *Tags for the Identification of Languages*, or its successor; in addition, the empty string may be specified.

(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 hei&#xDF;em Bem&#xFC;h'n.</l>
</sp>
```

The language specified by xml:lang applies to the element where it is specified (including the values of its attributes), and to all elements in its content unless overridden with another instance of xml:lang. In particular, the empty value of xml:lang is used on an element B to override a specification of xml:lang on an enclosing element A, without specifying another language. Within B, it is considered that there is no language information available, just as if xml:lang had not been specified on B or any of its ancestors. Applications determine which of an element's attribute values and which parts of its character content, if any, are treated as language-dependent values described by xml:lang.

☞ Language information may also be provided by external transport protocols (e.g. HTTP or MIME). When available, this information may be used by XML applications, but the more local information provided by xml:lang should be considered to override it.

A simple declaration for xml:lang might take the form

```
xml:lang CDATA #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 CDATA 'fr'>
<!ATTLIST gloss  xml:lang CDATA 'en'>
<!ATTLIST note   xml:lang CDATA 'en'>
```

2.13. Normalization Checking

All XML parsed entities (including document entities) SHOULD be fully normalized as per the definition of Appendix B – Definitions for Character Normalization on page 39 supplemented by the following definitions of *relevant constructs* for XML:

1. The replacement text of all parsed entities

2. All text matching, in context, one of the following productions:

   A. **CData**

   B. **CharData**

   C. **content**

   D. **Name**

   E. **Nmtoken**

However, a document is still well-formed even if it is not fully normalized. XML processors SHOULD provide a user option to verify that the document being processed is in fully normalized form, and report to the application whether it is or not. The option to not verify SHOULD be chosen only when the input text is certified, as defined by Appendix B – Definitions for Character Normalization on page 39.

The verification of full normalization MUST be carried out as if by first verifying that the entity is in include-normalized form as defined by Appendix B – Definitions for Character Normalization on page 39 and by then verifying that none of the relevant constructs listed above begins (after character references are expanded) with a composing character as defined by Appendix B – Definitions for Character Normalization on page 39. Non-validating processors MUST ignore possible denormalizations that would be caused by inclusion of external entities that they do not read.

The composing character are all Unicode characters of non-zero combining class, plus a small number of class-zero characters that nevertheless take part as a non-initial character in certain Unicode canonical decompositions. Since these characters are meant to follow base characters, restricting relevant constructs (including content) from beginning with a composing character does not meaningfully diminish the expressiveness of XML.

If, while verifying full normalization, a processor encounters characters for which it cannot determine the normalization properties (i.e., characters introduced in a version of Unicode [Unicode] later than the one used in the implementation of the processor), then the processor MAY, at user option, ignore any possible denormalizations caused by these characters. The option to ignore those denormalizations SHOULD NOT be chosen by applications when reliability or security are critical.

XML processors MUST NOT transform the input to be in fully normalized form. XML applications that create XML 1.1 output from either XML 1.1 or XML 1.0 input SHOULD ensure that the output is fully normalized; it is not necessary for internal processing forms to be fully normalized.

The purpose of this section is to strongly encourage XML processors to ensure that the creators of XML documents have properly normalized them, so that XML applications can make tests such as identity comparisons of strings without having to worry about the different possible "spellings" of strings which Unicode allows.

When entities are in a non-Unicode encoding, if the processor transcodes them to Unicode, it SHOULD use a normalizing transcoder.

3. Logical Structures

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

[37]	element	::=	**EmptyElemTag**
			| **STag content ETag**

This specification does not constrain the application 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 (not even entity references, comments, PIs or white space).
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, comments and PIs (i.e. markup matching production [27] **Misc**) 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 or a reference to an entity whose replacement text is character references expanding to white space do not match the nonterminal **S**, and hence cannot appear in these positions; however, a reference to an internal entity with a literal value consisting of character references expanding to white space does match **S**, since its replacement text is the white space resulting from expansion of the character references.
3. The declaration matches **Mixed** , and the content (after replacing any entity references with their replacement text) consists of character data (including CDATA sections), comments, PIs and child elements whose types match names in the content model.
4. The declaration matches `ANY`, and the content (after replacing any entity references with their replacement text) consists of character data , CDATA sections, comments, PIs and child elements whose types have been declared.

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

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

Start-tag

[38]	STag	::=	'<' **Name** (**S** **Attribute**)* **S**? '>'
[39]	Attribute	::=	**Name** **Eq** **AttValue**

The **Name** in the start- and end-tags gives the element's *type*. The **Name**-**AttValue** pairs are referred to as the *attribute specifications* of the element, with the **Name** in each pair referred to as the *attribute name* and 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

An attribute name MUST NOT 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 on page 18.)

Well-Formedness Constraint: No External Entity References

Attribute values MUST NOT 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">
```

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

[40]	ETag	::=	'</' **Name** **S**? '>'

An example of an end-tag:

```
</termdef>
```

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

Content of Elements

[41]	content	::=	**CharData**? ((**element** | **Reference** | **CDSect** | **PI** | **Comment**) **CharData**?)*

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. An *empty-element tag* takes a special form:

Tags for Empty Elements

[42]	EmptyElemTag	::=	'<' **Name** (**S** **Attribute**)* **S**? '/>'

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.

An *element type declaration* takes the form:

Element Type Declaration

[43]	elementdecl	::=	'<!ELEMENT' **S Name S contentspec S**? '>'
[44]	contentspec	::=	'EMPTY' | 'ANY' | **Mixed** | **children**

where the **Name** gives the element type being declared.

Validity Constraint: Unique Element Type Declaration
An element type MUST NOT 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

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 (**cp**s), which consist of names, choice lists of content particles, or sequence lists of content particles:

Element-content Models

[45]	children	::=	(**choice** | **seq**) ('?' | '*' | '+')?
[46]	cp	::=	(**Name** | **choice** | **seq**) ('?' | '*' | '+')?
[47]	choice	::=	'(' **S**? **cp** (**S**? '|' **S**? **cp**)+ **S**? ')'
[48]	seq	::=	'(' **S**? **cp** (**S**? ',' **S**? **cp**)* **S**? ')'

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 the content model allows an element to match more than one occurrence of an element type in the content model. For more information, see Appendix D – Deterministic Content Models on page 41.

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

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

```
[49]    Mixed ::= '(' S? '#PCDATA' (S? '|' S? Name)* S? ')*'
                  | '(' S? '#PCDATA' S? ')'
```

where the **Name**s 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 MUST NOT appear outside of 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 on page 15. 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.

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

[50]	AttlistDecl	::=	'<!ATTLIST' **S** **Name** **AttDef*** **S**? '>'
[51]	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 are more constrained. The validity constraints noted in the grammar are applied after the attribute value has been normalized as described in § 3.3.3 – Attribute-Value Normalization on page 22.

Attribute Types

[52]	AttType	::=	**StringType** | **TokenizedType** | **EnumeratedType**
[53]	StringType	::=	'CDATA'
[54]	TokenizedType	::=	'ID' | 'IDREF' | 'IDREFS' | 'ENTITY' | 'ENTITIES' | 'NMTOKEN' | 'NMTOKENS'

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

An element type MUST NOT 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**.

Enumerated attributes have a list of allowed values in their declaration . They MUST take one of those values. There are two kinds of enumerated attribute types:

Enumerated Attribute Types

[55]	EnumeratedType	::=	**NotationType** | **Enumeration**
[56]	NotationType	::=	'NOTATION' **S** '(' **S**? **Name** (**S**? '|' **S**? **Name**)* **S**? ')'
[57]	Enumeration	::=	'(' **S**? **Nmtoken** (**S**? '|' **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: One Notation Per Element Type

An element type MUST NOT 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: No Duplicate Tokens

The notation names in a single **NotationType** attribute declaration, as well as the **NmTokens** in a single **Enumeration** attribute declaration, MUST all be distinct.

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 is to react if a declared attribute is absent in a document.

Attribute Defaults

```
[58]    DefaultDecl  ::=  '#REQUIRED' | '#IMPLIED'
                          | (('#FIXED' S)? AttValue)                    /* */
```

In an attribute declaration, `#REQUIRED` means that the attribute MUST always be provided, `#IMPLIED` that no default value is provided. 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. When an XML processor encounters an element without a specification for an attribute for which it has read a default value declaration, it MUST report the attribute with the declared default value to the application.

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 Value Syntactically Correct

The declared default value MUST meet the syntactic constraints of the declared attribute type. That is, the default value of an attribute:

- of type IDREF or ENTITY must match the **Name** production;
- of type IDREFS or ENTITIES must match the **Names** production;
- of type NMTOKEN must match the **Nmtoken** production;
- of type NMTOKENS must match the **Nmtokens** production;
- of an **enumerated type** (either a **NOTATION** type or an **enumeration**) must match one of the enumerated values.

Note that only the syntactic constraints of the type are required here; other constraints (e.g. that the value be the name of a declared unparsed entity, for an attribute of type ENTITY) will be reported by a validating parser only if an element without a specification for this attribute actually occurs.

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 on page 12, 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`.

It is an error if an attribute value contains a reference to an entity for which no declaration has been read.

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;&#x20;&a;B&da;"`	`A #x20 B`	`#x20 #x20 A #x20 #x20 #x20 B #x20 #x20`

`a=`
`"&#xd;&#xd;A&#xa;&#xa;B&#xd;&#xa;"`
	`#xD #xD A #xA #xA B #xD #xA`	`#xD #xD A #xA #xA B #xD #xA`

Note that the last example is invalid (but well-formed) if a is declared to be of type `NMTOKENS`.

3.4. Conditional Sections

Conditional sections are portions of the document type declaration external subset or of external parameter entities which are included in, or excluded from, the logical structure of the DTD based on the keyword which governs them.

Conditional Section

[59]	conditionalSect	::=	**includeSect** | **ignoreSect**
[60]	includeSect	::=	'<![' **S**? 'INCLUDE' **S**? '[' **extSubsetDecl** ']]>'
[61]	ignoreSect	::=	'<![' **S**? 'IGNORE' **S**? '[' **ignoreSectContents*** ']]>'
[62]	ignoreSectContents	::=	**Ignore** ('<![' **ignoreSectContents** ']]>' **Ignore**)*
[63]	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 MUST be processed as part of the DTD. If the keyword of the conditional section is `IGNORE`, then the contents of the conditional section MUST NOT be processed as 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 MUST be ignored. The contents of an ignored conditional section MUST be 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 MUST NOT be 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

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. The contents of a *parsed entity* are referred to as its replacement text; this text is considered an integral part of the document.

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.

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. *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

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

```
[64]    CharRef ::= '&#' [0-9]+ ';'
                    | '&#x' [0-9a-fA-F]+ ';'
```

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.

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

Entity Reference

[65]	Reference	::=	**EntityRef** | **CharRef**
[66]	EntityRef	::=	'&' **Name** ';'
[67]	PEReference	::=	'%' **Name** ';'

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 non-validating processors are *not obligated to* to read and process entity declarations occurring in parameter entities or in the external subset; 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 parameter entity references 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 on page 33. 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 MUST NOT appear outside 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

Entities are declared thus:

Entity Declaration

[68]	EntityDecl	::=	**GEDecl** | **PEDecl**
[69]	GEDecl	::=	'<!ENTITY' **S Name S EntityDef S**? '>'
[70]	PEDecl	::=	'<!ENTITY' **S** '%' **S Name S PEDef S**? '>'
[71]	EntityDef	::=	**EntityValue** | (**ExternalID NDataDecl**?)
[72]	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

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 Entity Replacement Text on page 32.

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

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

External Entity Declaration

[73]	ExternalID	::=	'SYSTEM' **S** **SystemLiteral** | 'PUBLIC' **S** **PubidLiteral** **S** **SystemLiteral**
[74]	NDataDecl	::=	**S** 'NDATA' **S** **Name**

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.

The **SystemLiteral** is called the entity's *system identifier*. It is meant to be converted to a URI reference (as defined in [IETF RFC 3986]), as part of the process of dereferencing it 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. This is defined to be the external entity containing the '<' which starts the declaration, at the point when it is parsed as a declaration. 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. Attempts to retrieve the resource identified by a URI may be redirected at the parser level (for example, in an entity resolver) or below (at the protocol level, for example, via an HTTP `Location:` header). In the absence of additional information outside the scope of this specification within the resource, the base URI of a resource is always the URI of the actual resource returned. In other words, it is the URI of the resource retrieved after all redirection has occurred.

System identifiers (and other XML strings meant to be used as URI references) may contain characters that, according to [IETF RFC 3986], must be escaped before a URI can be used to retrieve the referenced resource. The characters to be escaped are the control characters #x0 to #x1F and #x7F (most of which cannot appear in XML), space #x20, the delimiters '<' #x3C, '>' #x3E and '"' #x22, the *unwise* characters '{' #x7B, '}' #x7D, '|' #x7C, '\' #x5C, '^' #x5E and '`' #x60, as well as all characters above #x7F. Since escaping is not always a fully reversible process, it MUST be performed only when absolutely necessary and as late as possible in a processing chain. In particular, neither the process of converting a relative URI to an absolute one nor the process of passing a URI reference to a process or software component responsible for dereferencing it SHOULD trigger escaping. When escaping does occur, it MUST be performed as follows:

1. Each character to be escaped is represented in UTF-8 [Unicode] as one or more bytes.
2. The resulting bytes 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.

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 any combination of the public and system identifiers as well as additional information outside the scope of this specification 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

[75] TextDecl ::= '<?xml' **VersionInfo**? **EncodingDecl** **S**? '?>'

The text declaration MUST be provided literally, not by reference to a parsed entity. The text declaration MUST NOT 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.

☞ Only parsed entities that are referenced directly or indirectly within the document are required to be well-formed.

Well-Formed External Parsed Entity

[76] extParsedEnt ::= (**TextDecl**? **content**) - (**Char*** **RestrictedChar** **Char***)

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 general 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 and entities encoded in UTF-8 MAY begin with the Byte Order Mark described in ISO/IEC 10646 [ISO/IEC 10646] or Unicode [Unicode] (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 on page 28) containing an encoding declaration:

Encoding Declaration

[77]	EncodingDecl	::=	**S** 'encoding' **Eq** ('"' **EncName** '"' | "'" **EncName** "'")	
[78]	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 a fatal 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 byte sequences that are not legal in that encoding. Specifically, it is a fatal error if an entity encoded in UTF-8 contains any irregular code unit sequences, as defined in

Unicode [Unicode]. Unless an encoding is determined by a higher-level protocol, 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'?>
```

4.3.4. Version Information in Entities

Each entity, including the document entity, can be separately declared as XML 1.0 or XML 1.1. The version declaration appearing in the document entity determines the version of the document as a whole. An XML 1.1 document may invoke XML 1.0 external entities, so that otherwise duplicated versions of external entities, particularly DTD external subsets, need not be maintained. However, in such a case the rules of XML 1.1 are applied to the entire document.

If an entity (including the document entity) is not labeled with a version number, it is treated as if labeled as version 1.0.

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 **Entity-Value**, **AttValue**, **PI**, **Comment**, **SystemLiteral**, **PubidLiteral**, or the contents of an ignored conditional section (see § 3.4 – Conditional Sections on page 23).

.

	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*	*Error*	*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

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&amp;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, except in the **EntityValue** in an entity declaration.
- 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 MUST be 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 MUST always be treated as a normal data character and MUST 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 MUST be 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 MUST be 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 MUST NOT apply to parameter entity references within entity values; these are described in § 4.4.5 – Included in Literal on page 32.

4.4.9. Error

It is an error for a reference to an unparsed entity to appear in the **EntityValue** in an entity declaration.

4.5. Construction of Entity Replacement Text

In discussing the treatment of entities, it is useful to distinguish two forms of the entity's value. For an internal entity, the *literal entity value* is the quoted string actually present in the entity declaration, corresponding to the non-terminal **EntityValue**. For an external entity, the *literal entity value* is the exact text contained in the entity. For an internal entity, the *replacement text* is the content of the entity, after replacement of character references and parameter-entity references. For an external entity, the *replacement text* is the content of the entity, after stripping the text declaration (leaving any surrounding white space) if there is one but without any replacement of character references or 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 (or) 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 Appendix C – Expansion of Entity and Character References on page 40.

4.6. Predefined Entities

Entity and character references may 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 OPTIONAL 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

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.

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

[79]	NotationDecl	::=	'<!NOTATION' **S** **Name** **S** (**ExternalID** | **PublicID**) **S**? '>'
[80]	PublicID	::=	'PUBLIC' **S** **PubidLiteral**

Validity Constraint: Unique Notation Name

A given **Name** MUST NOT be declared in more than one notation declaration.

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

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.

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. 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; when `standalone="yes"`, processors MUST process these declarations.

Note that when processing invalid documents with a non-validating processor the application may not be presented with consistent information. For example, several requirements for uniqueness within the document may not be met, including more than one element with the same id, duplicate declarations of elements or notations with the same name, etc. In these cases the behavior of the parser with respect to reporting such information to the application is undefined.

XML 1.1 processors MUST be able to process both XML 1.0 and XML 1.1 documents. Programs which generate XML SHOULD generate XML 1.0, unless one of the specific features of XML 1.1 is required.

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 fail to 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 on page 30.
- 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 fail to *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 DTD facilities not related to validation (such as the declaration of default attributes and internal entities that are or may be specified 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 lowercase 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 whose number (code point) in ISO/IEC 10646 is `N`. The number of leading zeros in the `#xN` form is insignificant.

[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`.

`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.

Appendix A. References

A.1. Normative References

IANA-CHARSETS

(Internet Assigned Numbers Authority) *Official Names for Character Sets*, ed. Keld Simonsen et al. Available at `http://www.iana.org/assignments/character-sets`.

IETF RFC 2119

IETF (Internet Engineering Task Force). *RFC 2119: Key words for use in RFCs to Indicate Requirement Levels*. Scott Bradner, 1997. Available at `http://www.ietf.org/rfc/rfc2119.txt`.

IETF RFC 3066

IETF (Internet Engineering Task Force). *RFC 3066: Tags for the Identification of Languages*, ed. H. Alvestrand. 2001. Available at `http://www.ietf.org/rfc/rfc3066.txt`.

IETF RFC 3986

IETF (Internet Engineering Task Force). *RFC 3986: Uniform Resource Identifier (URI): Generic Syntax*. T. Berners-Lee, R. Fielding, L. Masinter. 2005. Available at `http://www.ietf.org/rfc/rfc3986.txt`.

ISO/IEC 10646

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* and *ISO/IEC 10646-2:2001. Information technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 2: Supplementary Planes*, as, from time to time, amended, replaced by a new edition or expanded by the addition of new parts. [Geneva]: International Organization for Standardization. (See http://www.iso.ch for the latest version.)

Unicode

The Unicode Consortium. *The Unicode Standard, Version 4.0.* Reading, Mass.: Addison-Wesley, 2003, as updated from time to time by the publication of new versions. (See http://www.unicode.org/unicode/standard/versions for the latest version and additional information on versions of the standard and of the Unicode Character Database).

XML-1.0

W3C. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, François Yergeau (editors) Available at `http://www.w3.org/TR/xml`.

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.

Brüggemann-Klein

Brüggemann-Klein, Anne. *Formal Models in Document Processing*. Habilitationsschrift. Faculty of Mathematics at the University of Freiburg, 1993. Available at `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.

Charmod

W3C Working Draft. *Character Model for the World Wide Web 1.0*. Martin J. Dürst, François Yergeau, Richard Ishida, Misha Wolf, Tex Texin. Available at `http://www.w3.org/TR/2003/WD-charmod-20030822/`.

Clark

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

IANA-LANGCODES

(Internet Assigned Numbers Authority) *Registry of Language Tags*, ed. Keld Simonsen et al. Available at `http://www.iana.org/assignments/language-tags`.

IETF RFC 2141

IETF (Internet Engineering Task Force). *RFC 2141: URN Syntax*, ed. R. Moats. 1997. Available at `http://www.ietf.org/rfc/rfc2141.txt`.

IETF RFC 3023

IETF (Internet Engineering Task Force). *RFC 3023: XML Media Types*. eds. M. Murata, S. St.Laurent, D. Kohn. 2001. Available at `http://www.ietf.org/rfc/rfc3023.txt`.

IETF RFC 2781

IETF (Internet Engineering Task Force). *RFC 2781: UTF-16, an encoding of ISO 10646*, ed. P. Hoffman, F. Yergeau. 2000. Available at `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. Available at `http://www.sgmlsource.com/8879/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. Available at `http://www.w3.org/TR/REC-xml-names/`.

Appendix B. Definitions for Character Normalization

This appendix contains the necessary definitions for character normalization. For additional background information and examples, see [Charmod].

Text is said to be in a *Unicode encoding form* if it is encoded in UTF-8, UTF-16 or UTF-32.

Legacy encoding is taken to mean any character encoding not based on Unicode.

A *normalizing transcoder* is a transcoder that converts from a legacy encoding to a Unicode encoding form and ensures that the result is in Unicode Normalization Form C (see UAX #15 [Unicode]).

A *character escape* is a syntactic device defined in a markup or programming language that allows one or more of:

1. expressing syntax-significant characters while disregarding their significance in the syntax of the language, or
2. expressing characters not representable in the character encoding chosen for an instance of the language, or
3. expressing characters in general, without use of the corresponding character codes.

Certified text is text which satisfies at least one of the following conditions:

1. it has been confirmed through inspection that the text is in normalized form
2. the source text-processing component is identified and is known to produce only normalized text.

Text is, for the purposes of this specification, *Unicode-normalized* if it is in a Unicode encoding form and is in Unicode Normalization Form C, according to a version of Unicode Standard Annex #15: Unicode Normalization Forms [Unicode] at least as recent as the oldest version of the Unicode Standard that contains all the characters actually present in the text, but no earlier than version 3.2.

Text is *include-normalized* if:

1. the text is Unicode-normalized and does not contain any character escapes or includes whose expansion would cause the text to become no longer Unicode-normalized; or
2. the text is in a legacy encoding and, if it were transcoded to a Unicode encoding form by a normalizing transcoder, the resulting text would satisfy clause 1 above.

A *composing character* is a character that is one or both of the following:

1. the second character in the canonical decomposition mapping of some primary composite (as defined in D3 of UAX #15 [Unicode]), or
2. of non-zero canonical combining class (as defined in Unicode [Unicode]).

Text is *fully-normalized* if:

1. the text is in a Unicode encoding form, is include-normalized and none of the comprising the text begin with a composing character or a character escape representing a composing character; or
2. the text is in a legacy encoding and, if it were transcoded to a Unicode encoding form by a normalizing transcoder, the resulting text would satisfy clause 1 above.

Appendix C. 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 on page 30.

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='1.1'?>
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.*

Appendix D. Deterministic Content Models (Non-Normative)

As noted in § 3.2.1 – Element Content on page 17, 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].

Appendix E. 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.

E.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`	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.
`3C 00 00 00`	
`00 00 3C 00`	
`00 3C 00 00`	
`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

☞ 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.

E.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 3023] 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.

Appendix F. 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 participants voted for its approval. The current and former members in 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

Appendix G. W3C XML Core Working Group (Non-Normative)

The second edition of this specification was prepared by the W3C XML Core Working Group (WG). The participants in the WG at the time of publication of this edition were:

Leonid Arbouzov, Sun Microsystems; John Cowan; Andrew Fang, PTC-Arbortext; Paul Grosso, PTC-Arbortext (Co-Chair); Konrad Lanz, A-SIT; Philippe Le Hégaret, W3C (Staff Contact); Glenn Marcy, IBM; Sandra Martinez, NIST; Ravindrakumar R, CDAC; Lew Shannon; Henry Thompson, W3C (Staff Contact); Richard Tobin, University of Edinburgh; Daniel Veillard; Norman Walsh, Sun Microsystems (Co-Chair); François Yergeau

Appendix H. Production Notes (Non-Normative)

This edition was encoded in a slightly modified version of the XMLspec DTD, 2.10. The XHTML versions were produced with a combination of the xmlspec.xsl, diffspec.xsl, and REC-xml.xsl XSLT stylesheets.

Appendix I. Suggestions for XML Names (Non-Normative)

The following suggestions define what is believed to be best practice in the construction of XML names used as element names, attribute names, processing instruction targets, entity names, notation names, and the values of attributes of type ID, and are intended as guidance for document authors and schema designers. All references to Unicode are understood with respect to a particular version of the Unicode Standard greater than or equal to 3.0; which version should be used is left to the discretion of the document author or schema designer.

The first two suggestions are directly derived from the rules given for identifiers in the Unicode Standard, version 3.0, and exclude all control characters, enclosing nonspacing marks, non-decimal numbers, private-use characters, punctuation characters (with the noted exceptions), symbol characters, unassigned codepoints, and white space characters. The other suggestions are mostly derived from [XML-1.0] Appendix B.

1. The first character of any name should have a Unicode General Category of Ll, Lu, Lo, Lm, Lt, or Nl, or else be '_' #x5F.

2. Characters other than the first should have a Unicode General Category of Ll, Lu, Lo, Lm, Lt, Mc, Mn, Nl, Nd, Pc, or Cf, or else be one of the following: '-' #x2D, '.' #x2E, ':' #x3A or '·' #xB7 (middle dot). Since Cf characters are not directly visible, they should be employed with caution and only when necessary, to avoid creating names which are distinct to XML processors but look the same to human beings.
3. Ideographic characters which have a canonical decomposition (including those in the ranges [#xF900-#xFAFF] and [#x2F800-#x2FFFD], with 12 exceptions) should not be used in names.
4. Characters which have a compatibility decomposition (those with a "compatibility formatting tag" in field 5 of the Unicode Character Database -- marked by field 5 beginning with a "<") should not be used in names. This suggestion does not apply to #x0E33 THAI CHARACTER SARA AM or #x0EB3 LAO CHARACTER AM, which despite their compatibility decompositions are in regular use in those scripts.
5. Combining characters meant for use with symbols only (including those in the ranges [#x20D0-#x20EF] and [#x1D165-#x1D1AD]) should not be used in names.
6. The interlinear annotation characters ([#xFFF9-#xFFFB]) should not be used in names.
7. Variation selector characters should not be used in names.
8. Names which are nonsensical, unpronounceable, hard to read, or easily confusable with other names should not be employed.

This page is intentionally left blank.