REC-xml-19980210

Extensible Markup Language (XML) 1.0

W3C Recommendation 10-February-1998

This version:
http://www.w3.org/TR/1998/REC-xml-19980210
http://www.w3.org/TR/1998/REC-xml-19980210.xml
http://www.w3.org/TR/1998/REC-xml-19980210.html
http://www.w3.org/TR/1998/REC-xml-19980210.pdf
http://www.w3.org/TR/1998/REC-xml-19980210.ps
Latest version:
http://www.w3.org/TR/REC-xml
Previous version:
http://www.w3.org/TR/PR-xml-971208
Editors:
Tim Bray (Textuality and Netscape) <tbray@textuality.com>
Jean Paoli (Microsoft) <jeanpa@microsoft.com>
C. M. Sperberg-McQueen (University of Illinois at Chicago) <cmsmcq@uic.edu>

Abstract

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

Status of this document

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

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

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

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

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

Extensible Markup Language (XML) 1.0

Table of Contents

Appendices

---

1. Introduction

Extensible Markup Language, abbreviated XML, describes a class of data objects 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 WG's contact with the W3C.

The design goals for XML are:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.
10. Terseness in XML markup is of minimal importance.

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

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

1.2 Terminology

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

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

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

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

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

for compatibility
: A feature of XML included solely to ensure that XML remains compatible with SGML.

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

2. Documents

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

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

2.1 Well-Formed XML Documents

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

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

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

Matching the `document` production implies that:

1. It contains one or more elements.
2. 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 graphic characters of Unicode and ISO/IEC 10646. The use of "compatibility characters", as defined in section 6.8 of [Unicode], is discouraged.

Character Range

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

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

2.3 Common Syntactic Constructs

This section defines some symbols used widely in the grammar.

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

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

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

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

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

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

Names and Tokens

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

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

Literals

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

2.4 Character Data and Markup

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

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

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

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

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

Character Data			
[14]	`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.

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

An example of a comment:

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

2.6 Processing Instructions

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

Processing Instructions

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

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

2.7 CDATA Sections

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

CDATA Sections

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

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

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

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

2.8 Prolog and Document Type Declaration

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

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

and so is this:

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

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

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

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

Prolog

```
[22]      prolog ::= XMLDecl? Misc* (doctypedecl Misc*)?
[23]     XMLDecl ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'
[24] VersionInfo ::= S 'version' Eq (' VersionNum ' | " VersionNum ")
[25]          Eq ::= S? '=' S?
[26]  VersionNum ::= ([a-zA-Z0-9_.:] | '-')+
[27]        Misc ::= Comment | PI | S
```

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 fuller information, see "4. Physical Structures".

Document Type Definition

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

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

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

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

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

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

External Subset

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

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

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

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

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

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

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

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

2.9 Standalone Document Declaration

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

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

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

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

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

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

contain declarations of:

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

An example XML declaration with a standalone document declaration:

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

2.10 White Space Handling

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

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

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

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

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

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

2.11 End-of-Line Handling

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

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

2.12 Language Identification

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

Language Identification

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

The `Langcode` may be any of the following:

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

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

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

For example:

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

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

A simple declaration for `xml:lang` might take the form

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

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

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

3. Logical Structures

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

Element

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

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

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

Validity Constraint: Element Valid

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

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

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

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

Start-tag

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

The Name in the start- and end-tags gives the element's **type**. 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**.

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

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

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

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

An example of a start-tag:

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

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

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

An example of an end-tag:

```
</termdef>
```

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

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

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

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

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

Examples of empty elements:

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

3.2 Element Type Declarations

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

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

An **element type declaration** takes the form:

<table>
<tr><th colspan="4">Element Type Declaration</th></tr>
<tr><td>[45]</td><td>elementdecl ::=</td><td>'<!ELEMENT' S Name S contentspec S? '>'</td><td>[VC: Unique Element Type Declaration]</td></tr>
<tr><td>[46]</td><td>contentspec ::=</td><td>'EMPTY' | 'ANY' | Mixed | children</td><td></td></tr>
</table>

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

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

Examples of element type declarations:

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

3.2.1 Element Content

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:

<table>
<tr><th colspan="4">Element-content Models</th></tr>
<tr><td>[47]</td><td>children ::=</td><td>(choice | seq) ('?' | '*' | '+')?</td><td></td></tr>
<tr><td>[48]</td><td>cp ::=</td><td>(Name | choice | seq) ('?' | '*' | '+')?</td><td></td></tr>
<tr><td>[49]</td><td>choice ::=</td><td>'(' S? cp (S? '|' S? cp)* S? ')'</td><td>[VC: Proper Group/PE Nesting]</td></tr>
<tr><td>[50]</td><td>seq ::=</td><td>'(' S? cp (S? ',' S? cp)* S? ')'</td><td>[VC: Proper Group/PE Nesting]</td></tr>
</table>

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

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

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

Examples of element-content models:

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

3.2.2 Mixed Content

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

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

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

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

Examples of mixed content declarations:

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

3.3 Attribute-List Declarations

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

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

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

Attribute-list Declaration

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

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

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

3.3.1 Attribute Types

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

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

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

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

Validity Constraint: ID Attribute Default
An ID attribute must have a declared default of `#IMPLIED` or `#REQUIRED`.

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

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

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

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

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

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

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

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

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

3.3.2 Attribute Defaults

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

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

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

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

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

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

Examples of attribute-list declarations:

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

3.3.3 Attribute-Value Normalization

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

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

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

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

3.4 Conditional Sections

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

Conditional Section

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

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

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

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

An example:

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

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

4. Physical Structures

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

Entities may be either parsed or unparsed. A **parsed entity's** contents 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 not be XML. Each unparsed entity has an associated notation, identified by name. Beyond a requirement that an XML processor make the identifiers for the entity and notation available to the application, XML places no constraints on the contents of unparsed entities.

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

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

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

If the character reference begins with "`&#x`", the digits and letters up to the terminating `;` provide a hexadecimal representation of the character's code point in ISO/IEC 10646. If it begins just with "`&#`", the digits up to the terminating `;` provide a decimal representation of the character's code point.

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

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

Validity Constraint: Entity Declared
In a document with an external subset or external parameter entities with "`standalone='no'`", the `Name` given in the entity reference must match that in an entity declaration. For interoperability, valid documents should declare the entities `amp`, `lt`, `gt`, `apos`, `quot`, in the form specified in "4.6 Predefined Entities". The declaration of a parameter entity must precede any reference to it. Similarly, the declaration of a general entity must precede any reference to it which appears in a default value in an attribute-list declaration.

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

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

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

Examples of character and entity references:

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

Example of a parameter-entity reference:

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

4.2 Entity Declarations

Entities are declared thus:

Entity Declaration

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

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

4.2.1 Internal Entities

If the entity definition is an `EntityValue`, the defined entity is called an **internal entity**. There is no separate physical storage object, and the content of the entity is given in the declaration. Note that some processing of entity and character references in the literal entity value may be required to produce the correct replacement text: see "4.5 Construction of Internal Entity Replacement Text".

An internal entity is a parsed entity.

Example of an internal entity declaration:

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

4.2.2 External Entities

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

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

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

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

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

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

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

Examples of external entity declarations:

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

4.3 Parsed Entities

4.3.1 The Text Declaration

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

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

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

4.3.2 Well-Formed Parsed Entities

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

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

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

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

4.3.3 Character Encoding in Entities

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

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

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

Encoding Declaration

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

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

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

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

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

Examples of encoding declarations:

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

4.4 XML Processor Treatment of Entities and References

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

Reference in Content
: as a reference anywhere after the start-tag and before the end-tag of an element; corresponds to the nonterminal `content`.

Reference in Attribute Value
: as a reference within either the value of an attribute in a start-tag, or a default value in an attribute declaration; corresponds to the nonterminal `AttValue`.

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

Reference in Entity Value
: as a reference within a parameter or internal entity's literal entity value in the entity's declaration; corresponds to the nonterminal `EntityValue`.

Reference in DTD
: as a reference within either the internal or external subsets of the DTD, but outside of an `EntityValue` or `AttValue`.

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

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, except that the replacement text of entities used to escape markup delimiters (the entities `amp`, `lt`, `gt`, `apos`, `quot`) is always treated as data. (The string "`AT&T;`" expands to "`AT&T;`" and the remaining ampersand is not recognized as an entity-reference delimiter.) A character reference is **included** when the indicated character is processed in place of the reference itself.

4.4.3 Included If Validating

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

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

4.4.4 Forbidden

The following are forbidden, and constitute fatal errors:

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

4.4.5 Included in Literal

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

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

while this is not:

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

4.4.6 Notify

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

4.4.7 Bypassed

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

4.4.8 Included as PE

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

4.5 Construction of Internal Entity Replacement Text

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

The literal entity value as given in an internal entity declaration (`EntityValue`) may contain character, parameter-entity, and general-entity references. Such references must be contained entirely within the literal entity value. The actual replacement text that is included as described above must contain the *replacement text* of any parameter entities referred to, and must contain the character referred to, in place of any character references in the literal entity value; however, general-entity references must be left as-is, unexpanded. For example, given the following declarations:

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

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

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

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

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

4.6 Predefined Entities

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

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

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

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

4.7 Notation Declarations

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

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

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

4.8 Document Entity

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 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. They must not process entity declarations or attribute-list declarations encountered after a reference to a parameter entity that is not read, since the entity may have contained overriding declarations.

5.2 Using XML Processors

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

- Certain well-formedness errors, specifically those that require reading external entities, may not be detected by a non-validating processor. Examples include the constraints entitled Entity Declared, Parsed Entity, and No Recursion, as well as some of the cases described as forbidden in "4.4 XML Processor Treatment of Entities and References".

- The information passed from the processor to the application may vary, depending on whether the processor reads parameter and external entities. For example, a non-validating processor may not normalize attribute values, include the replacement text of internal entities, or supply default attribute values, where doing so depends on having read declarations in external or parameter entities.

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

6. Notation

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

```
symbol ::= expression
```

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

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

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

`[a-zA-Z], [#xN-#xN]`
: matches any character with a value in the range(s) indicated (inclusive).

`[^a-z], [^#xN-#xN]`
: matches any character with a value *outside* the range indicated.

`[^abc], [^#xN#xN#xN]`
: matches any character with a value not among the characters given.

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

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

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

`(expression)`

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

`A?`
: matches `A` or nothing; optional `A`.

`A B`
: matches `A` followed by `B`.

`A | B`
: matches `A` or `B` but not both.

`A - B`
: matches any string that matches `A` but does not match `B`.

`A+`
: matches one or more occurrences of `A`.

`A*`
: matches zero or more occurrences of `A`.

Other notations used in the productions are:

`/* ... */`
: comment.

`[ wfc: ... ]`
: well-formedness constraint; this identifies by name a constraint on well-formed documents associated with a production.

`[ vc: ... ]`
: validity constraint; this identifies by name a constraint on valid documents associated with a production.

Appendices

A. References

A.1 Normative References

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

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

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

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

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

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

A.2 Other References

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

Berners-Lee et al.
Berners-Lee, T., R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax and Semantics*. 1997. (Work in progress; see updates to RFC1738.)

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

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

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

IETF RFC1738
IETF (Internet Engineering Task Force). *RFC 1738: Uniform Resource Locators (URL)*, ed. T. Berners-Lee, L. Masinter, M. McCahill. 1994.

IETF RFC1808
IETF (Internet Engineering Task Force). *RFC 1808: Relative Uniform Resource Locators*, ed. R. Fielding. 1995.

IETF RFC2141
IETF (Internet Engineering Task Force). *RFC 2141: URN Syntax*, ed. R. Moats. 1997.

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

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

B. Character Classes

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

Characters

[84] `Letter ::= BaseChar | Ideographic`

[85]
```
BaseChar ::= [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6]
           | [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131]
           | [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E]
           | [#x0180-#x01C3] | [#x01CD-#x01F0] | [#x01F4-#x01F5]
           | [#x01FA-#x0217] | [#x0250-#x02A8] | [#x02BB-#x02C1]
           | #x0386 | [#x0388-#x038A] | #x038C | [#x038E-#x03A1]
           | [#x03A3-#x03CE] | [#x03D0-#x03D6] | #x03DA | #x03DC
           | #x03DE | #x03E0 | [#x03E2-#x03F3] | [#x0401-#x040C]
           | [#x040E-#x044F] | [#x0451-#x045C] | [#x045E-#x0481]
           | [#x0490-#x04C4] | [#x04C7-#x04C8] | [#x04CB-#x04CC]
           | [#x04D0-#x04EB] | [#x04EE-#x04F5] | [#x04F8-#x04F9]
           | [#x0531-#x0556] | #x0559 | [#x0561-#x0586]
           | [#x05D0-#x05EA] | [#x05F0-#x05F2] | [#x0621-#x063A]
           | [#x0641-#x064A] | [#x0671-#x06B7] | [#x06BA-#x06BE]
           | [#x06C0-#x06CE] | [#x06D0-#x06D3] | #x06D5
           | [#x06E5-#x06E6] | [#x0905-#x0939] | #x093D
           | [#x0958-#x0961] | [#x0985-#x098C] | [#x098F-#x0990]
           | [#x0993-#x09A8] | [#x09AA-#x09B0] | #x09B2
           | [#x09B6-#x09B9] | [#x09DC-#x09DD] | [#x09DF-#x09E1]
           | [#x09F0-#x09F1] | [#x0A05-#x0A0A] | [#x0A0F-#x0A10]
           | [#x0A13-#x0A28] | [#x0A2A-#x0A30] | [#x0A32-#x0A33]
           | [#x0A35-#x0A36] | [#x0A38-#x0A39] | [#x0A59-#x0A5C]
           | #x0A5E | [#x0A72-#x0A74] | [#x0A85-#x0A8B] | #x0A8D
           | [#x0A8F-#x0A91] | [#x0A93-#x0AA8] | [#x0AAA-#x0AB0]
           | [#x0AB2-#x0AB3] | [#x0AB5-#x0AB9] | #x0ABD | #x0AE0
           | [#x0B05-#x0B0C] | [#x0B0F-#x0B10] | [#x0B13-#x0B28]
           | [#x0B2A-#x0B30] | [#x0B32-#x0B33] | [#x0B36-#x0B39]
           | #x0B3D | [#x0B5C-#x0B5D] | [#x0B5F-#x0B61]
           | [#x0B85-#x0B8A] | [#x0B8E-#x0B90] | [#x0B92-#x0B95]
           | [#x0B99-#x0B9A] | #x0B9C | [#x0B9E-#x0B9F]
           | [#x0BA3-#x0BA4] | [#x0BA8-#x0BAA] | [#x0BAE-#x0BB5]
           | [#x0BB7-#x0BB9] | [#x0C05-#x0C0C] | [#x0C0E-#x0C10]
           | [#x0C12-#x0C28] | [#x0C2A-#x0C33] | [#x0C35-#x0C39]
           | [#x0C60-#x0C61] | [#x0C85-#x0C8C] | [#x0C8E-#x0C90]
           | [#x0C92-#x0CA8] | [#x0CAA-#x0CB3] | [#x0CB5-#x0CB9]
           | #x0CDE | [#x0CE0-#x0CE1] | [#x0D05-#x0D0C]
           | [#x0D0E-#x0D10] | [#x0D12-#x0D28] | [#x0D2A-#x0D39]
           | [#x0D60-#x0D61] | [#x0E01-#x0E2E] | #x0E30
           | [#x0E32-#x0E33] | [#x0E40-#x0E45] | [#x0E81-#x0E82]
           | #x0E84 | [#x0E87-#x0E88] | #x0E8A | #x0E8D
           | [#x0E94-#x0E97] | [#x0E99-#x0E9F] | [#x0EA1-#x0EA3]
           | #x0EA5 | #x0EA7 | [#x0EAA-#x0EAB] | [#x0EAD-#x0EAE]
           | #x0EB0 | [#x0EB2-#x0EB3] | #x0EBD | [#x0EC0-#x0EC4]
           | [#x0F40-#x0F47] | [#x0F49-#x0F69] | [#x10A0-#x10C5]
           | [#x10D0-#x10F6] | #x1100 | [#x1102-#x1103]
           | [#x1105-#x1107] | #x1109 | [#x110B-#x110C]
           | [#x110E-#x1112] | #x113C | #x113E | #x1140 | #x114C
           | #x114E | #x1150 | [#x1154-#x1155] | #x1159
           | [#x115F-#x1161] | #x1163 | #x1165 | #x1167 | #x1169
           | [#x116D-#x116E] | [#x1172-#x1173] | #x1175 | #x119E
```

```
                         | #x11A8 | #x11AB | [#x11AE-#x11AF] | [#x11B7-#x11B8]
                         | #x11BA | [#x11BC-#x11C2] | #x11EB | #x11F0 | #x11F9
                         | [#x1E00-#x1E9B] | [#x1EA0-#x1EF9] | [#x1F00-#x1F15]
                         | [#x1F18-#x1F1D] | [#x1F20-#x1F45] | [#x1F48-#x1F4D]
                         | [#x1F50-#x1F57] | #x1F59 | #x1F5B | #x1F5D
                         | [#x1F5F-#x1F7D] | [#x1F80-#x1FB4] | [#x1FB6-#x1FBC]
                         | #x1FBE | [#x1FC2-#x1FC4] | [#x1FC6-#x1FCC]
                         | [#x1FD0-#x1FD3] | [#x1FD6-#x1FDB] | [#x1FE0-#x1FEC]
                         | [#x1FF2-#x1FF4] | [#x1FF6-#x1FFC] | #x2126
                         | [#x212A-#x212B] | #x212E | [#x2180-#x2182]
                         | [#x3041-#x3094] | [#x30A1-#x30FA] | [#x3105-#x312C]
                         | [#xAC00-#xD7A3]
[86]      Ideographic ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]
[87]    CombiningChar ::= [#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
                         | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD]
                         | #x05BF | [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652]
                         | #x0670 | [#x06D6-#x06DC] | [#x06DD-#x06DF]
                         | [#x06E0-#x06E4] | [#x06E7-#x06E8] | [#x06EA-#x06ED]
                         | [#x0901-#x0903] | #x093C | [#x093E-#x094C] | #x094D
                         | [#x0951-#x0954] | [#x0962-#x0963] | [#x0981-#x0983]
                         | #x09BC | #x09BE | #x09BF | [#x09C0-#x09C4]
                         | [#x09C7-#x09C8] | [#x09CB-#x09CD] | #x09D7
                         | [#x09E2-#x09E3] | #x0A02 | #x0A3C | #x0A3E | #x0A3F
                         | [#x0A40-#x0A42] | [#x0A47-#x0A48] | [#x0A4B-#x0A4D]
                         | [#x0A70-#x0A71] | [#x0A81-#x0A83] | #x0ABC
                         | [#x0ABE-#x0AC5] | [#x0AC7-#x0AC9] | [#x0ACB-#x0ACD]
                         | [#x0B01-#x0B03] | #x0B3C | [#x0B3E-#x0B43]
                         | [#x0B47-#x0B48] | [#x0B4B-#x0B4D] | [#x0B56-#x0B57]
                         | [#x0B82-#x0B83] | [#x0BBE-#x0BC2] | [#x0BC6-#x0BC8]
                         | [#x0BCA-#x0BCD] | #x0BD7 | [#x0C01-#x0C03]
                         | [#x0C3E-#x0C44] | [#x0C46-#x0C48] | [#x0C4A-#x0C4D]
                         | [#x0C55-#x0C56] | [#x0C82-#x0C83] | [#x0CBE-#x0CC4]
                         | [#x0CC6-#x0CC8] | [#x0CCA-#x0CCD] | [#x0CD5-#x0CD6]
                         | [#x0D02-#x0D03] | [#x0D3E-#x0D43] | [#x0D46-#x0D48]
                         | [#x0D4A-#x0D4D] | #x0D57 | #x0E31 | [#x0E34-#x0E3A]
                         | [#x0E47-#x0E4E] | #x0EB1 | [#x0EB4-#x0EB9]
                         | [#x0EBB-#x0EBC] | [#x0EC8-#x0ECD] | [#x0F18-#x0F19]
                         | #x0F35 | #x0F37 | #x0F39 | #x0F3E | #x0F3F
                         | [#x0F71-#x0F84] | [#x0F86-#x0F8B] | [#x0F90-#x0F95]
                         | #x0F97 | [#x0F99-#x0FAD] | [#x0FB1-#x0FB7] | #x0FB9
                         | [#x20D0-#x20DC] | #x20E1 | [#x302A-#x302F] | #x3099
                         | #x309A
[88]            Digit ::= [#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9]
                         | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F]
                         | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF]
                         | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F]
                         | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29]
[89]         Extender ::= #x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640 | #x0E46
                         | #x0EC6 | #x3005 | [#x3031-#x3035] | [#x309D-#x309E]
                         | [#x30FC-#x30FE]
```

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

- Name start characters must have one of the categories Ll, Lu, Lo, Lt, Nl.
- Name characters other than Name-start characters must have one of the categories Mc, Me, Mn, Lm, or Nd.
- Characters in the compatibility area (i.e. with character code greater than #xF900 and less than

#xFFFE) are not allowed in XML names.
- Characters which have a font or compatibility decomposition (i.e. those with a "compatibility formatting tag" in field 5 of the database -- marked by field 5 beginning with a "<") are not allowed.
- The following characters are treated as name-start characters rather than name characters, because the property file classifies them as Alphabetic: [#x02BB-#x02C1], #x0559, #x06E5, #x06E6.
- Characters #x20DD-#x20E0 are excluded (in accordance with Unicode, section 5.14).
- Character #x00B7 is classified as an extender, because the property list so identifies it.
- Character #x0387 is added as a name character, because #x00B7 is its canonical equivalent.
- Characters ':' and '_' are allowed as name-start characters.
- Characters '-' and '.' are allowed as name characters.

C. XML and SGML (Non-Normative)

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

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

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

If the DTD contains the declaration

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

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

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

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

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

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

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

This produces the following:

- in line 4, the reference to character 37 is expanded immediately, and the parameter entity "xx" is stored in the symbol table with the value "%zz;". Since the replacement text is not rescanned, the reference to parameter entity "zz" is not recognized. (And it would be an error if it were, since "zz" is not yet declared.)
- in line 5, the character reference "<" is expanded immediately and the parameter entity "zz" is stored with the replacement text "<!ENTITY tricky "error-prone" >", which is a well-formed entity declaration.
- in line 6, the reference to "xx" is recognized, and the replacement text of "xx" (namely "%zz;") is parsed. The reference to "zz" is recognized in its turn, and its replacement text ("<!ENTITY tricky "error-prone" >") is parsed. The general entity "tricky" has now been declared, with the replacement text "error-prone".
- in line 8, the reference to the general entity "tricky" is recognized, and it is expanded, so the full content of the "test" element is the self-describing (and ungrammatical) string *This sample shows a error-prone method.*

E. Deterministic Content Models (Non-Normative)

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

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

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

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

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

F. Autodetection of Character Encodings (Non-Normative)

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

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

- `00 00 00 3C`: UCS-4, big-endian machine (1234 order)
- `3C 00 00 00`: UCS-4, little-endian machine (4321 order)
- `00 00 3C 00`: UCS-4, unusual octet order (2143)
- `00 3C 00 00`: UCS-4, unusual octet order (3412)
- `FE FF`: UTF-16, big-endian
- `FF FE`: UTF-16, little-endian
- `00 3C 00 3F`: UTF-16, big-endian, no Byte Order Mark (and thus, strictly speaking, in error)
- `3C 00 3F 00`: UTF-16, little-endian, no Byte Order Mark (and thus, strictly speaking, in error)
- `3C 3F 78 6D`: UTF-8, ISO 646, ASCII, some part of ISO 8859, Shift-JIS, EUC, or any other 7-bit, 8-bit, or mixed-width encoding which ensures that the characters of ASCII have their normal positions, width, and values; the actual encoding declaration must be read to detect which of these applies, but since all of these encodings use the same bit patterns for the ASCII characters, the encoding declaration itself may be read reliably
- `4C 6F A7 94`: EBCDIC (in some flavor; the full encoding declaration must be read to tell which code page is in use)
- other: UTF-8 without an encoding declaration, or else the data stream is corrupt, fragmentary, or enclosed in a wrapper of some kind

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

Because the contents of the encoding declaration are restricted to ASCII characters, a processor can reliably read the entire encoding declaration as soon as it has detected which family of encodings is in use. Since in practice, all widely used character encodings fall into one of the categories above, the

XML encoding declaration allows reasonably reliable in-band labeling of character encodings, even when external sources of information at the operating-system or transport-protocol level are unreliable.

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

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

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

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

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

G. W3C XML Working Group (Non-Normative)

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

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