APPENDIX A

XML Reference Material

This appendix contains XML reference material. It is divided into three main parts:

1. XML BNF Grammar
2. Well-Formedness Constraints
3. Validity Constraints

The XML BNF grammar reference section shows you how to read a BNF Grammar and includes the BNF rules for XML 1.0 and examples of the XML 1.0 productions. The well-formedness constraints reference section explains what a well-formedness constraint is and lists the productions associated with the well-formedness constraints. The validity constraints reference section explains what a validity constraint is and lists and explains all of the validity constraints in the XML 1.0 Standard.

XML BNF Grammar

According to the XML 1.0 specification, an XML document is well-formed 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.

This section is designed to help you understand the first of those requirements and more quickly determine whether your documents meet that requirement.

Reading a BNF Grammar

BNF is an abbreviation for Backus-Naur-Form. BNF grammars are an outgrowth of compiler theory. A BNF grammar defines what is and is not a syntactically correct program or, in the case of XML, a syntactically correct document. It is possible to compare a document to a BNF grammar and determine precisely whether it does or does not meet the conditions of that grammar. There are no borderline cases. BNF grammars, properly written, have the advantage of leaving no room for interpretation. The advantage of this should be obvious to anyone who's had to struggle with HTML documents that display in one browser but not in another.

Technically, XML uses an Extended-Backus-Naur-Form grammar, which adds a few pieces not normally found in traditional, compiler-oriented BNF grammars.

Syntactical correctness is a necessary but not sufficient condition for XML documents. A document may strictly adhere to the BNF grammar, and yet fail to be well-formed or valid. For a document to be well-formed, it must also meet all the well-formedness constraints of the XML 1.0 specification. Well-formedness is the minimum level a document may achieve to be parsed. To be valid, a document must also meet all the validity constraints of the XML 1.0 specification. The well-formedness and validity constraints are discussed in the next two sections of this appendix, respectively.

BNF Grammar Parts

A BNF grammar has three parts:

1. A set of literal strings called *terminals*. For example, `CDATA`, `</`, `<`, `>`, `#REQUIRED`, and `<!ENTITY` are all terminals used in the XML 1.0 specification.
2. A set of *non-terminals* to ultimately be replaced by terminals.
3. A list of *productions* or rules that map non-terminals onto particular sequences of terminals and other non-terminals, including one specially identified as the *start* or *document* production.

If you're not a compiler theorist, that list probably could have been written in ancient Etruscan and made about as much sense. Let's see if we can make things clearer with a simple example, before we dive into the complexities of the XML 1.0 grammar.

Consider strings composed of non-negative integers added to or subtracted from each other, like these:

```
9+8+1+2+3
8-1-2-4-5
9+8-9-0+5+3
4
4+3
```

Notice a few things that are not in the list, and that we want to forbid in our grammar:

- ✦ Any character except the digits 0 through 9 and the plus and the minus signs
- ✦ Whitespace
- ✦ A string that begins with a + or a -
- ✦ Numbers less than 0 or greater than 9
- ✦ The empty string

Here's a BNF grammar that defines precisely those strings we want, and none of those we don't want:

```
[1] string ::= digit
[2] digit  ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7'
                   | '8' | '9'
[3] string ::= string '+' digit
[3] string ::= string '-' digit
```

Suppose you want to determine whether the string "9+3-2" satisfies this grammar. You begin by looking at the first production. This says that a string is the nonterminal digit. So you move to Production [2] which defines digit. Indeed 9 is one of the terminals listed as a digit. Thus the string "9" is a legitimate string. Production [3] says that a string followed by the plus sign and another digit is also a legitimate string. Thus "9+3" satisfies the grammar. Furthermore, it itself is a string. Production [4] says that a string followed by the minus sign and another digit is a legitimate string. Thus "9+3-2" is a legitimate string and satisfies the grammar.

Now consider the string "-9+1". By Production [1] a string must begin with a digit. This string doesn't begin with a digit, so it's illegal.

The XML 1.0 grammar is much larger and more complicated than this simple grammar. The next section lists its 89 productions. The following section elaborates on each of these productions in detail.

BNF Symbols

In XML's EBNF grammar the following basic symbols are used on the right-hand sides of productions:

`#xN`	N is a hexadecimal integer, and #xN is the Unicode character with the number N
`[a-zA-Z]`	matches any character in the specified range
`[#xN-#xN]`	matches any character in the specified range where N is the hexadecimal value of a Unicode character

`[^a-z]`	matches any character not in the specified range
`[^#xN-#xN]`	matches any character not in the specified range where N is the hexadecimal value of a Unicode character
`[^abc]`	matches any character not in the list
`[^#xN#xN#xN]`	matches any character whose value is not in the list
'*string*'	matches the literal string inside the single quotes
"*string*"	matches the literal string inside the double quotes

These nine basic patterns may be grouped to match more complex expressions:

(*contents*)	the contents of the parentheses are treated as a unit
A?	matches zero or one occurrences of 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 and does not match B
A+	matches one or more occurrences of A
A*	matches zero or more occurrences of A

The XML specification also uses three forms you probably won't encounter in non-XML-related specifications:

/* *text of comment* */	This is a comment, and any text inside the comment is ignored.
[WFC: *name*]	This names a well-formedness constraint associated with this production that documents must meet in order to qualify as well-formed. Well-formedness constraints will be found in the specification, but are not encapsulated in the BNF grammar.
[VC: *name*]	This names a validity constraint associated with this production that documents must meet in order to qualify as valid. Validity constraints will be found in the specification, but are not encapsulated in the BNF grammar.

The BNF Rules for XML 1.0

The complete BNF grammar for XML is given in the XML 1.0 specification, which you'll find in Appendix B of this book. However, if you're merely trying to match up your markup against productions in the grammar, it can be inconvenient to flip through the pages hunting for the necessary rules. For that purpose, the BNF rules and only the BNF rules for XML 1.0 are reproduced here.

Document

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

Character Range

```
[2] Char ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD]
                | [#x10000-#x10FFFF]
```

White Space

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

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)*
```

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]
                       | [-'()+,./:=?;!*#@$_%]
```

Character Data

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

Comments

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

Processing Instructions

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

CDATA Sections

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

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-ZO-9_.:] | '-')+
[27] Misc        ::= Comment | PI |  S
```

Document Type Definition

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

External Subset

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

Standalone Document Declaration

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

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])+
```

Element

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

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 ]
```

End Tag

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

Content of Elements

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

Tags for Empty Elements

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

Element Type Declaration

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

Element-content Models

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

Mixed-content Declaration

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

Attribute-list Declaration

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

Attribute Types

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

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 ]
```

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 ]
```

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*)
```

Character Reference

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

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 ]
```

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
```

External Entity Declaration

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

Text Declaration

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

Well-formed External Parsed Entity

```
[78] extParsedEnt ::= TextDecl? content
[79] extPE        ::= TextDecl? extSubsetDecl
```

Encoding Declaration

```
[80] EncodingDecl ::= S 'encoding' Eq ('"' EncName '"'
                    |  "'" EncName "'" )
[81] EncName      ::= [A-Za-z] ([A-Za-z0-9._] | '-')*
```

Notation Declarations

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

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]
```

Examples of the XML 1.0 Productions

This section shows you some instances of the productions to give you a better idea of what each one means.

Document

[1] document ::= prolog element Misc*

This rule says that an XML document is composed of a prolog (Production [22]), followed by a single root element (Production [39]), followed by any number of miscellaneous items (Production [27]). In other words, a typical structure looks like this:

```
<?xml version="1.0"?>
<!– a DTD might go here –>
<ROOT_ELEMENT>
  Content
</ROOT_ELEMENT>
<!– comments can go here –>
<?Reader, processing instructions can also go here?>
```

In practice, it's rare for anything to follow the close of the root element.

Production [1] rules out documents with more than one element as a root. For example,

```
<?xml version="1.0"?>
<ELEMENT1>
  Content
</ELEMENT1>
<ELEMENT2>
  Content
</ELEMENT2>
<ELEMENT1>
  Content
</ELEMENT1>
```

Character Range

[2] Char ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]

Production [2] defines the subset of Unicode characters which may appear in an XML document. The main items of interest here are the characters not included. Specifically, these are the non-printing ASCII control characters of which the most common are the bell, vertical tab, and formfeed; the surrogates block from #xD800 to #xDFFF, and the non-character #xFFFE.The control characters are not needed in XML and may cause problems in files displayed on old terminals or passed through old terminal servers and software.

The surrogates block will eventually be used to extend Unicode to support over one million different characters. However, none of these million plus are currently defined, and XML parsers are not allowed to support them.

The non-character #xFFFE is not defined in Unicode. Its appearance, especially at the start of a document, should indicate that you're reading the document with the wrong byte order; that is little endian instead of big endian or vice versa.

Whitespace

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

Production [3] defines whitespace as a run of one or more space characters (#x20), the horizontal tab (#x9), the carriage return (#xD), and the linefeed (#xA). Because of the +, 20 of these characters in a row are treated exactly the same as one.

Other ASCII whitespace characters like the vertical tab (#xB) are prohibited by production [2]. Other non-ASCII, Unicode whitespace characters like the non-breaking space (#A0) are not considered whitespace for the purposes of XML.

Names and Tokens

[4] NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender

Production [4] defines the characters that may appear in an XML name. XML names may only contain letters, digits, periods, hyphens, underscores, colons, combining characters, (Production [87]) and extenders (Production [89]).

[5] Name ::= (Letter | '_' | ':') (NameChar)*

Production [5] says an XML name must begin with a letter, an underscore, or a colon. It may not begin with a digit, a period, or a hyphen. Subsequent characters in an XML name may include any XML name character (Production [4]) including digits, periods, and hyphens. The following are acceptable XML names:

```
airplane
text.encoding
r
SEAT
Pilot
Pilot1
OscarWilde
BOOK_TITLE
:TITLE
_8ball
_______
```

These are not acceptable XML names:

```
air plane
.encoding
-r
Wilde,Oscar
BOOK TITLE
8ball
AHA!
```

[6] Names ::= Name (S Name)*

Production [6] defines a group of names as one or more XML names (Production [5]) separated by whitespace. This is a valid group of XML names:

```
BOOK AUTHOR TITLE PAGE EDITOR CHAPTER
```

This is not a valid group of XML names:

```
BOOK, AUTHOR, TITLE, PAGE, EDITOR, CHAPTER
```

[7] Nmtoken ::= (NameChar)+

Production [7] defines a name token as any sequence of one or more name characters. Unlike an XML name, a name token has no restrictions on what the first character is as long as it is a valid name character (Production [4]). In other words, XML name tokens may begin with a digit, a period, or a hyphen while an XML name may not. All valid XML names are valid XML name tokens, but not all valid name tokens are valid XML names.

The following are acceptable name tokens:

```
airplane
text.encoding
r
SEAT
```

```
Pilot
Pilot1
OscarWilde
BOOK_TITLE
:TITLE
_8ball
______
.encoding
-r
8ball
```

The following are not acceptable name tokens:

```
air plane
Wilde,Oscar
BOOK TITLE
AHA!
```

[8] Nmtokens ::= Nmtoken (S Nmtoken)*

Production [8] says a group of name tokens is one or more XML name tokens (Production [7]) separated by whitespace. This is a valid group of XML name tokens:

```
1POTATO 2POTATO 3POTATO 4POTATO
```

This is not a valid group of XML name tokens:

```
1POTATO, 2POTATO, 3POTATO, 4POTATO
```

Literals

[9] EntityValue ::= “‘“ ([^%&”] | PEReference | Reference)* “‘“ | “““ ([^%&’] | PEReference | Reference)* “““

Production [9] defines an entity value as any string of characters enclosed in double quotes or single quotes except for %, &, and the quote character (single or double) used to delimit the string. % and & may be used, however, if and only they’re the start of a parameter entity reference (Production [69]), general entity reference (Production [67]) or character reference. If you really need to include % and & in your entity values, you can escape them with the character references % and &, respectively.

These are legal entity values:

```
"This is an entity value"
'This is an entity value'
"75&#37; off"
"Ben &amp; Jerry's New York Super Fudge Chunk Ice Cream"
```

These are not legal entity values:

```
"This is an entity value'
'This is an entity value"
"75% off"
"Ben & Jerry's New York Super Fudge Chunk Ice Cream"
'Ben &amp; Jerry's New York Su
```

[10] AttValue ::= “‘“ ([^<&”] | Reference)* “‘“ | “‘“ ([^<&’] | Reference)* “‘“

Production [10] says that an attribute value may consist of any characters except, <, &, and " enclosed in double quotes or any characters except <, &, and ' enclosed in single quotes. The & may appear, however, only if it's used as the start of a reference (Production [67]) (either general or character).

These are legal attribute values:

```
"This is an attribute value"
'This is an attribute value'
'#FFCC33'
"75% off"
"Ben &amp; Jerry's New York Super Fudge Chunk Ice Cream"
"i &lt; j"
```

These are not legal attribute values:

```
"This is an attribute value'
'This is an attribute value"
"Ben & Jerry's New York Super Fudge Chunk Ice Cream"
'Ben &amp; Jerry's New York Super Fudge Chunk Ice Cream'
"i < j"
```

[11] SystemLiteral ::= (“‘“ [^”]* “‘“) | (“‘“ [^’]* “‘“)

Production [11] defines a system literal as any string of text that does not contain the double quote mark enclosed in double quotes. Alternately, a system literal may be any string of text that does not contain the single quote mark enclosed in single quotes. These are grammatical system literals:

```
"test"
" Hello there! "
' Hello
  there!'
"Embedded markup is <OK/> in system literals"
```

These are ungrammatical system literals:

```
" He said, "Get out of here!""
'Bailey's Cove'
```

[12] PubidLiteral ::= “‘“ PubidChar* “‘“ | “““ (PubidChar - “““)* “““

Production [12] says that a public ID literal is either zero or more public ID characters (Production [13]) enclosed in double quotes or zero or more public ID characters except the single quote mark enclosed in single quotes.

These are grammatical public ID literals:

```
"-//IETF//NONSGML Media Type application/pdf//EN"
'-//IETF//NONSGML Media Type application/pdf//EN'
"-//W3C//DTD XHTML 1.0 Strict + Math//EN"
```

These are ungrammatical public ID literals:

```
"{-//IETF//NONSGML Media Type application/pdf//EN}"
"-//IETF//NONSGML Media Type application/▼__//GR}"
```

[13] PubidChar ::= #x20 | #xD | #xA | [a-zA-Z0-9] | [-’()+,./:=?;!*#@$_%]

Production [13] lists the permissible public ID characters, essentially, the ASCII space, carriage return, and linefeed, the letters *a* through *z* and *A* through *Z*, the digits *0* through *9*, and the punctuation characters *-’()+,./:=?;!*#@$_%*.

Character Data

[14] CharData ::= [^<&]* - ([^<&]* ‘]]>’ [^<&]*)

Production [14] defines character data as any number of characters except for < and &. Furthermore the CDEnd string]]> may not appear as part of the character data. Character data may contain as few as zero characters.

Comments

[15] Comment ::= ‘<!—’ ((Char - ‘-’) | (‘-’ (Char - ‘-’)))* ‘—>’

Production [15] defines a comment as any string of text enclosed between <!— and —> marks with the single exception of the double hyphen —. These are all valid comments:

```
<!—Hello—>
<!—Hello there!—>
<!— Hello there! —>
<!— Hello
     there! —>
<!—<Hello/> <there/>!—>
<!—<Hello/> </there>!—>
```

This is not a legal comment:

```
<!— Hello—there! —>
```

Processing Instructions

[16] PI ::= '<?' PITarget (S (Char* - (Char* '?>' Char*)))? '?>'

Production [16] says that a processing instruction starts with the literal <?, followed by the name of the processing instruction target (Production [17]), optionally followed by whitespace followed by any number of characters except ?>. Finally, the literal ?> closes the processing instruction.

These are all legal processing instructions:

```
<?gcc version="2.7.2" options="-O4"?>
<?Terri Do you think this is a good example?>
```

These are not legal processing instructions:

```
<? I have to remember to fix this next part?>
<?Terri This is a good example!>
```

[17] PITarget ::= Name - (('X' | 'x') ('M' | 'm') ('L' | 'l'))

Production [17] says that a processing instruction target may be any XML name (Production [5]) except the string XML (in any combination of case). Thus, these are all acceptable processing instruction targets:

```
gcc
acrobat
Acrobat
Joshua
Acrobat_301
xml-stylesheet
XML_Whizzy_Writer_2000
```

These are not acceptable processing instruction targets:

```
xml
XML
xmL
```

CDATA Sections

[18] CDSect ::= CDStart CData CDEnd

Production [18] states that a CData section is composed of a CDStart (Production [19]), CData (Production [20]), and a CDEnd (Production [21]) in that order.

[19] CDStart ::= '<![CDATA['

Production [19] defines a CDStart as the literal string <![CDATA[and nothing else.

[20] CData ::= (Char* - (Char* ‘]]>’ Char*))

Production [20] says that a CData section may contain absolutely any characters except the CDEnd string]]>.

[21] CDEnd ::= ‘]]>’

Production [21] defines a CDEnd as the literal string]]> and nothing else.

These are correct CDATA sections:

```
<![CDATA[ The < character starts a tag in XML ]]>
<![CDATA[ CDATA sections begin with the literal <![CDATA[ ]]>
```

This is not a legal CDATA section:

```
<![CDATA[
  The three characters ]]> terminate a CDATA section
]]>
```

Prolog

[22] prolog ::= XMLDecl? Misc* (doctypedecl Misc*)?

Production [22] says that a prolog consists of an optional XML declaration, followed by zero or more miscellaneous items (Production [27]), followed by an optional document type declaration (Production [28]), followed by zero or more miscellaneous items. For instance, this is a legal prolog:

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

This is also a legal prolog:

```
<?xml version="1.0" standalone="yes"?>
<?xml:stylesheet type="text/css" href="greeting.css"?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
```

This is also a legal prolog:

```
<!—This strange document really doesn't have anything
    in its prolog! —>
```

This is not a legal prolog because a comment precedes the XML declaration:

```
<!—This is from the example in Chapter 8 —>
<?xml version="1.0" standalone="yes"?>
```

```
<?xml:stylesheet type="text/css" href="greeting.css"?>
<!DOCTYPE greeting [
  <!ELEMENT greeting (#PCDATA)>
]>
```

[23] XMLDecl ::= '<?xml' VersionInfo EncodingDecl? SDDecl? S? '?>'

Production [23] defines an XML declaration as the literal string `<?xml` followed by a mandatory version info string (Production [24]), optionally followed by an encoding declaration (Production [80]), optionally followed by a standalone document declaration (Production [32]), optionally followed by whitespace, followed by the literal string `?>`. These are legal XML declarations:

```
<?xml version="1.0"?>
<?xml version="1.0" encoding="Big5"?>
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<?xml version="1.0" standalone="no"? >
<?xml version="1.0" encoding="ISO-8859-5"?>
```

These are not legal XML declarations:

```
<?xml?>
<?xml encoding="Big5"?>
<?xml version="1.0" standalone="yes"? encoding="ISO-8859-1" >
<?xml version="1.0" standalone="no"? styles="poems.css">
```

[24] VersionInfo ::= S 'version' Eq (' VersionNum ' | " VersionNum ")

Production [24] defines the version info string as whitespace followed by the literal string `version`, followed by an equals sign (Production [25]), followed by a version number enclosed in either single or double quotes. These are legal version info strings:

```
version="1.0"
version='1.0'
version = '1.0'
```

These are ungrammatical version info strings:

```
version='1.0"
"1.0"=version
```

[25] Eq ::= S? '=' S?

Production [25] defines the string `Eq` in the grammar as a stand-in for the equals sign (`=`) in documents. Whitespace (Production [3]) may or may not appear on either side of the equals sign.

[26] VersionNum ::= ([a-zA-Z0-9_.:] | '-')+

Production [26] says that a version number consists of one or more of the letters a through z, the capital letters A through Z, the underscore, the period, and the hyphen. The following are grammatically correct version numbers:

```
1.0
1.x
1.1.3
1.5EA2
v1.5
EA_B
```

The following are ungrammatical version numbers:

```
version 1.5
1,5
1!1
1 5 3
v 1.5
_
```

The only version number currently used in XML documents is 1.0. This production might as well read:

```
VersionNum ::= "1.0"
```

[27] Misc ::= Comment | PI | S

Production [27] defines miscellaneous items in an XML document include comments (Production [15]), processing instructions (Production [16]), and whitespace (Production [3]).

Document Type Definition

[28] doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('[' (markupdecl | PEReference | S)* ']' S?)? '>'

Production [28] says that a document type declaration consists of the literal string `<!DOCTYPE`, followed by whitespace (Production [3]), followed by an XML name (Production [5]), optionally followed by whitespace and an external ID (Production [75]), optionally followed by more whitespace, followed by a left square bracket (`[`), followed by zero or more markup declarations (Production [29]), parameter entity references (Production [69]), and whitespace, followed by a right square bracket (`]`) and whitespace, followed by a closing angle bracket.

[29] markupdecl ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment

Production [29] says that a markup declaration may be either an element declaration (Production [45]), an attribute list declaration (Production [52]), an entity declaration (Production [70]), a notation declaration (Production [82]), a processing instruction (Production [16]), or a comment (Production [15]).

External Subset

[30] extSubset ::= TextDecl? extSubsetDecl

Production [30] says that an external subset consists of an optional text declaration (Production [77]), followed by an external subset declaration (Production [31]). Note that external subsets are merged into the document from the files they reside in before the syntax is checked against the BNF grammar.

[31] extSubsetDecl ::= (markupdecl | conditionalSect | PEReference | S)*

Production [31] says the external subset declaration contains any number of markup declarations (Production [29]), conditional sections (Production [61]), parameter entity references (Production [69]), and whitespace in any order. In essence, the external subset can contain anything the internal DTD can contain.

Standalone Document Declaration

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

Production [32] says that the standalone document declaration consists of the literal `standalone`, followed by an equals sign (which may be surrounded by whitespace), followed by one of the two values `yes` or `no` enclosed in single or double quotes. Legal standalone document declarations include:

```
standalone="yes"
standalone="no"
standalone='yes'
standalone='no'
standalone="yes"
standalone="no"
```

Language Identification

[33] LanguageID ::= Langcode ('-' Subcode)*

Production [33] defines a language ID as a language code (Production [34]), followed by zero or more hyphens and subcodes (Production [38]).

[34] Langcode ::= ISO639Code | IanaCode | UserCode

Production [34] defines a language code as either an ISO 639 code (Production [35]), an IANA code (Production [36]), or a user code (Production [37]).

[35] ISO639Code ::= ([a-z] | [A-Z]) ([a-z] | [A-Z])

Production [35] defines an ISO 639 code as exactly two small letters from the English alphabet. There are exactly 2704 (52 _ 52) grammatical ISO 639 codes including:

```
en
fr
jp
EN
jP
Fr
```

There are an infinite number of strings that aren't grammatical ISO 639 codes including:

```
English
French
Japanese
_______
```

[36] IanaCode ::= ('i' | 'I') '-' ([a-z] | [A-Z])+

Production [36] defines an IANA code as the small or capital letter *I* followed by a hyphen, followed by one or more letters from the English alphabet. These are grammatical IANA codes:

```
i-no-bok
i-no-nyn
i-navajo
i-mingo
```

These are not grammatical IANA codes:

```
no-bok
no-nyn
navajo
mingo
i-_______
```

[37] UserCode ::= ('x' | 'X') '-' ([a-z] | [A-Z])+

Production [37] defines a user code as the small or capital letter *X* followed by a hyphen, followed by one or more letters from the English alphabet. These are grammatical user codes:

```
x-klingon
X-Elvish
```

These are not grammatical IANA codes:

```
Elvish
xklingon
x-________
```

[38] Subcode ::= ([a-z] | [A-Z])+

Production [38] defines a subcode as one or more capital or small letters from the English alphabet. These are grammatical subcodes:

```
gb
GreatBritain
UK
uk
```

These are not grammatical subcodes:

```
Great Britain
_______
```

Element

[39] element ::= EmptyElemTag | STag content ETag

Production [39] defines an element as either an empty element tag (production [44]) or a start tag (production [40]), followed by content (production [43]), followed by an end tag (production [42]).

These are legal elements:

```
<P>Hello!</P>
<P/>
<P></P>
```

These are not legal elements:

```
<P>Hello!</p>
<P>
</Q>
```

Start Tag

[40] STag ::= '<' Name (S Attribute)* S? '>'

Production [40] says that a start tag begins with a < followed by an XML name (Production [5]), followed by any number of attributes (Production [41]) separated by whitespace, followed by a closing >. These are some legal start tags:

```
<DOCUMENT>
<_________>
<DOCUMENT  >
<DOCUMENT TITLE="The Red Badge of Courage" >
<DOCUMENT TITLE="The Red Badge of Courage" PAGES="129">
```

These are not legal start tags:

```
< DOCUMENT>
< >
<12091998>
```

[41] Attribute ::= Name Eq AttValue

Production [41] says that an attribute consists of an XML name (Production [5]), followed by an equals sign (which may be encased in whitespace) followed by an attribute value (Production [10]). Grammatical attributes include:

```
TITLE="The Red Badge of Courage"
PAGES="129"
TITLE = "The Red Badge of Courage"
PAGES = "129"
TITLE='The Red Badge of Courage'
PAGES='129'
SENTENCE='Jim said, "I didn&apos;t expect to see you here."'
```

Ungrammatical attributes include:

```
TITLE="The Red Badge of Courage'
PAGES=129
SENTENCE='Then Jim said, "I didn't expect to see you here."'
```

End Tag

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

Production [42] defines an end tag as the literal string </ immediately followed by an XML name, optionally followed by whitespace, followed by the > character. For example, these are grammatical XML end tags:

```
</PERSON>
</PERSON >
</AbrahamLincoln>
</_________>
```

These are not grammatical XML end tags:

```
</ PERSON>
</Abraham Lincoln>
</PERSON NAME="Abraham Lincoln">
</>
```

Content of Elements

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

Production [43] defines content as any number of elements (Production [39]), character data (Production [14]), references (Production [67]), CDATA sections (Production [18]), processing instructions (Production [16]), and comments (Production [15]) in any order. This production lists everything that can appear inside an element.

Tags for Empty Elements

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

Production [44] defines an empty element tag as the character <, followed by an XML name, followed by whitespace, followed by zero more attributes separated from each other by whitespace, optionally followed by whitespace, followed by the literal />. These are some grammatical empty tags:

```
<PERSON/>
<PERSON />
<Person/>
<person />
<AbrahamLincoln/>
<________/>
```

These are ungrammatical as empty tags:

```
< PERSON/>
<PERSON>
</Person>
</person/>
</>
```

(The second and third are grammatical start and end tags respectively.)

Element Type Declaration

[45] elementdecl ::= '<!ELEMENT' S Name S contentspec S? '>'

Production [45] says that an element declaration consists of the literal `<!ELEMENT`, followed by whitespace, followed by an XML name (Production [5]), followed by a content specification (Production [46]), optionally followed by whitespace, followed by the > character.

Grammatical element declarations include:

```
<!ELEMENT DOCUMENT ANY>
<!ELEMENT HR EMPTY>
<!ELEMENT DOCUMENT (#PCDATA | P | H)>
```

[46] contentspec ::= 'EMPTY' | 'ANY' | Mixed | children

Production [46] defines a content specification as either the literals `EMPTY` or `ANY`, a list of children (Production [47]) or mixed content (Production [51]).

Element-content Models

[47] children ::= (choice | seq) ('?' | '*' | '+')?

Production [47] says that a list of children consists of either a choice (Production [49]) or a sequence (Production [50]) optionally followed by one of the characters ?, *, or +.

[48] cp ::= (Name | choice | seq) ('?' | '*' | '+')?

Production [48] defines a content particle as an XML name (Production [5]), choice, (Production [49]), or sequence (Production [50], optionally suffixed with a ?, *, or +.

[49] choice ::= '(' S? cp (S? '|' S? cp)* S? ')'

Production [49] says that a choice is one or more content particles (Production [48]) enclosed in parentheses and separated from each other by vertical bars and optional whitespace. Grammatical choices include:

```
(P | UL | H1 | H2 | H3 | H4 | H5 | BLOCKQUOTE | PRE | HR | DIV)
(P|UL|H1|H2|H3|H4|H5|H6|BLOCKQUOTE|PRE|HR|DIV)
(SON | DAUGHTER)
( SON | DAUGHTER )
(ADDRESS | (NAME, STREET, APT, CITY, STATE, ZIP))
```

[50] seq ::= '(' S? cp (S? ',' S? cp)* S? ')'

Production [50] says that a sequence is one or more content particles (Production [48]) enclosed in parentheses and separated from each other by commas and optional whitespace. Grammatical sequences include:

```
(NAME, STREET, APT, CITY, STATE, ZIP)
(NAME , STREET , APT , CITY , STATE , ZIP)
(NAME,STREET,APT,CITY,STATE,ZIP)
( NAME,STREET,APT, CITY,STATE,ZIP )
(NAME, (STREET|BOX), (APT|SUITE), CITY, STATE, ZIP, COUNTRY?)
(NAME)
```

Mixed-content Declaration

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

Production [51] says that mixed content is either the literal `( #PCDATA)` (with allowances for optional whitespace) or a choice that includes the literal `#PCDATA` as its first content particle. These are some grammatical mixed-content models:

```
(#PCDATA)
( #PCDATA )
(#PCDATA | PERSON)
( #PCDATA | PERSON )
( #PCDATA | TITLE | JOURNAL | MONTH | YEAR | SERIES | VOLUME )
```

These are ungrammatical mixed content models:

```
(PERSON | #PCDATA)
(#PCDATA, TITLE, #PCDATA, JOURNAL, MONTH, YEAR, #PCDATA)
(#PCDATA | (NAME, STREET, APT, CITY, STATE, ZIP))
```

Attribute-list Declaration

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

Production [52] says that an attribute list declaration consists of the literal `<!ATTLIST`, followed by whitespace, followed by an XML name (Production [5]), followed by zero or more attribute definitions (Production [53]), optionally followed by whitespace, followed by the `>` character.

Grammatical attribute list declarations include:

```
<!ATTLIST IMG ALT CDATA #REQUIRED>
<!ATTLIST AUTHOR EXTENSION CDATA #IMPLIED>
<!ATTLIST AUTHOR COMPANY   CDATA #FIXED "TIC">
<!ATTLIST P VISIBLE (TRUE | FALSE) "TRUE">
<!ATTLIST ADDRESS STATE NMTOKEN #REQUIRED>
<!ATTLIST ADDRESS STATES NMTOKENS #REQUIRED>
<!ATTLIST P PNUMBER ID #REQUIRED>
<!ATTLIST PERSON FATHER IDREF #IMPLIED>
<!ATTLIST SLIDESHOW SOURCES ENTITIES #REQUIRED>
<!ATTLIST SOUND PLAYER NOTATION (MP) #REQUIRED>
```

[53] AttDef ::= S Name S AttType S DefaultDecl

Production [53] defines an attribute definition as whitespace, an XML name (Production [5]), more whitespace, an attribute type (Production [54]), more whitespace, and a default declaration (Production [60]). Grammatical attribute definitions include:

```
IMG ALT CDATA #REQUIRED
AUTHOR EXTENSION CDATA #IMPLIED
AUTHOR COMPANY   CDATA #FIXED "TIC"
P VISIBLE (TRUE | FALSE) "TRUE"
ADDRESS STATE NMTOKEN #REQUIRED
ADDRESS STATES NMTOKENS #REQUIRED
P PNUMBER ID #REQUIRED
PERSON FATHER IDREF #IMPLIED
SLIDESHOW SOURCES ENTITIES #REQUIRED
SOUND PLAYER NOTATION (MP) #REQUIRED
```

Attribute Types

[54] AttType ::= StringType | TokenizedType | EnumeratedType

Production [54] defines an attribute type as either a string type (Production [55]), a tokenized type (Production [56]), or an enumerated type (Production [57]).

[55] StringType ::= 'CDATA'

Production [55] defines a string type as the literal `CDATA`.

[56] TokenizedType ::= 'ID' | 'IDREF' | 'IDREFS' | 'ENTITY' | 'ENTITIES' | 'NMTOKEN' | 'NMTOKENS'

Production [56] defines TokenizedType as any one of theses seven literals:

```
ID
IDREF
IDREFS
```

```
ENTITY
ENTITIES
NMTOKEN
NMTOKENS
```

Enumerated Attribute Types

[57] EnumeratedType ::= NotationType | Enumeration

Production [57] defines an enumerated type as either a notation type (Production [58]) or an enumeration (Production [59]).

[58] NotationType ::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'

Production [58] defines a notation type as the literal `NOTATION`, followed by whitespace, followed by one or more XML names (Production [5]), separated by vertical bars, and enclosed in parentheses. These are some grammatical notation types:

```
NOTATION (MP)
NOTATION (MP | PDF)
NOTATION (mp | gcc | xv)
NOTATION (A | B | C)
```

These are some ungrammatical notation types:

```
NOTATION ("MP")
NOTATION (MP PDF)
NOTATION (mp, gcc, xv)
NOTATION ("A" "B" "C")
```

[59] Enumeration ::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'

Production [59] defines an enumeration as one or more XML name tokens (Production [7]) separated by vertical bars and enclosed in parentheses. These are some grammatical enumerations:

```
(airplane)
(airplane | train | car | horse)
( airplane | train | car | horse )
(cavalo | carro | trem |avi¯o)
(A | B | C | D | E | F | G | H)
```

The following are not acceptable enumerations:

```
()
(airplane train car horse)
(A, B, C, D, E, F, G, H)
airplane | train | car | horse
```

Attribute Defaults

[60] DefaultDecl ::= '#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)

Production [60] defines the default declaration as one of these four things:

- the literal `#REQUIRED`
- the literal `#IMPLIED`
- the literal `#FIXED` followed by whitespace (Production [3]), followed by an attribute value (Production [10])
- an attribute value (Production [10])

Conditional Section

[61] conditionalSect ::= includeSect | ignoreSect

Production [61] defines a conditional section as either an include section (Production [62]) or an ignore section (Production [63]).

[62] includeSect ::= '<![' S? 'INCLUDE' S? '[' extSubsetDecl ']]>'

Production [62] defines an include section as an external subset declaration (Production [31]) sandwiched between `<![INCLUDE[ ]]>`, modulo whitespace. These are grammatical include sections:

```
<![ INCLUDE [ ]]>
<![INCLUDE[ ]]>
<![ INCLUDE[ ]]>
```

[63] ignoreSect ::= '<![' S? 'IGNORE' S? '[' ignoreSectContents* ']]>'

Production [63] defines an ignore section as ignore section contents (Production [64]) sandwiched between `<![IGNORE[ ]]>`, modulo whitespace. These are grammatical ignore sections:

```
<![ IGNORE [ ]]>
<![IGNORE[ ]]>
<![ IGNORE[ ]]>
```

[64] ignoreSectContents ::= Ignore ('<![' ignoreSectContents ']]>' Ignore)*

Production [64] defines an ignore section contents as an ignore block (Production [65]), optionally followed by a block of text sandwiched between `<![` and `]]>` literals, followed by more text. This may be repeated as many times as desired. This allows ignore sections to nest.

[65] Ignore ::= Char* - (Char* ('<![' | ']]>') Char*)

Production 65 defines an ignore block as any run of text that contains neither the `<![` or `]]>` literals. This prevents any possible confusion about where an ignore block ends.

Character Reference

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

Production [66] defines two forms for character references. The first is the literal `&#` followed by one or more of the ASCII digits 0 through 9. The second form is the literal `&#x` followed by one or more of the hexadecimal digits 0 through F. The digits representing 10 through 16 (A through F) may be either lower- or uppercase.

Entity Reference

[67] Reference ::= EntityRef | CharRef

Production [67] defines a reference as either an entity reference (Production [68]) or a character reference (Production [66]).

[68] EntityRef ::= '&' Name ';'

Production [68] defines an entity reference as an XML name (Production [5]) sandwiched between the ampersand character and a semicolon. These are grammatical entity references:

```
&amp;
&agrave;
&my_abbreviation;
```

These are ungrammatical entity references:

```
&amp
& agrave ;
& my_abbreviation;
```

[69] PEReference ::= '%' Name ';'

Production [69] defines a parameter entity reference as an XML name (Production [5]) sandwiched between the percent character and a semicolon. These are grammatical parameter entity references:

```
%inlines;
%mathml;
%MyElements;
```

These are ungrammatical parameter entity references:

```
%inlines
% mathml ;
%my elements;
```

Entity Declaration

[70] EntityDecl ::= GEDecl | PEDecl

Production [70] defines an entity declaration as either a general entity declaration (Production [71]) or a parameter entity declaration (Production [71]).

[71] GEDecl ::= '<!ENTITY' S Name S EntityDef S? '>'

Production [71] defines a general entity declaration as the literal `<!ENTITY` followed by whitespace (Production [3]), followed by an XML name (Production [5]), followed by an entity definition (Production [73]), optionally followed by whitespace, followed by the `>` character. These are some grammatical general entity declarations:

```
<!ENTITY alpha "&#945;">
<!ENTITY Alpha "&#913;">
<!ENTITY SPACEMUSIC SYSTEM "/sounds/space.wav" NDATA MP >
<!ENTITY LOGO SYSTEM "logo.gif">
<!ENTITY COPY99 "Copyright 1999 %erh;">
```

These are some ungrammatical general entity declarations:

```
<!ENTITY alpha &#945;>
<!ENTITY Capital Greek Alpha "&#913;">
<!ENTITY LOGO SYSTEM logo.gif>
```

[72] PEDecl ::= '<!ENTITY' S '%' S Name S PEDef S? '>'

Production [72] defines a parameter entity declaration as the literal `<!ENTITY` followed by whitespace (Production [3]), followed by a percent sign and more whitespace, followed by an XML name (Production [5]), followed by an entity definition (Production [73]), optionally followed by whitespace, followed by the `>` character. In essence this says that parameter entity declarations are the same as general entity declarations except for the `%` between the `<!ENTITY` and the name. These are some grammatical parameter entity declarations:

```
<!ENTITY % fulldtd "IGNORE">
<!ENTITY % ERH "Elliotte Rusty Harold">
<!ENTITY % inlines
  "(person | degree | model | product | animal | ingredient)*">
```

These are some ungrammatical parameter entity declarations:

```
<!ENTITY %fulldtd; "IGNORE">
<!ENTITY % ERH  Elliotte Rusty Harold>
<!ENTITY % inlines
  "(person | degree | model | product | animal | ingredient)*'>
```

[73] EntityDef ::= EntityValue | (ExternalID NDataDecl?)

Production [73] says that an entity definition is either an entity value (Production [9]) or an external ID (Production [75]) followed by an NData declaration (Production [76]).

[74] PEDef ::= EntityValue | ExternalID

Production [74] says that the definition of a parameter entity may be either an entity value (Production [9]) or an external ID (Production [75]).

External Entity Declaration

[75] ExternalID ::= 'SYSTEM' S SystemLiteral | 'PUBLIC' S PubidLiteral S SystemLiteral

Production [75] defines an external ID as either the keyword `SYSTEM` followed by whitespace and a system literal (Production [11]) or the keyword `PUBLIC` followed by whitespace, a public ID literal (Production [12]), more whitespace, and a system literal (Production [11]). These are some grammatical external IDs:

```
SYSTEM "logo.gif"
SYSTEM "/images/logo.gif"
SYSTEM "http://www.idgbooks.com/logo.gif"
SYSTEM "../images/logo.gif"
PUBLIC "-//IETF//NONSGML Media Type image/gif//EN"
       "http://www.isi.edu/in-notes/iana/assignments/media-
types/image/gif"
```

These are some ungrammatical external IDs:

```
SYSTEM logo.gif
SYSTEM "/images/logo.gif'
SYSTEM http://www.idgbooks.com/logo.gif
PUBLIC "-//IETF//NONSGML Media Type image/gif//EN"
PUBLIC "http://www.isi.edu/in-notes/iana/assignments/media-
types/image/gif"
```

[76] NDataDecl ::= S 'NDATA' S Name

Production [76] defines an NData declaration as whitespace (Production [3]), followed by the `NDATA` literal, followed by whitespace, followed by an XML name (Production [5]). For example:

```
NDATA PDF
NDATA MIDI
```

Text Declaration

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

Production [77] says that a text declaration looks almost like an XML declaration (Production [23]) except that it may not have a standalone document declaration (Production [32]). These are grammatical text declarations:

```
<?xml version="1.0"?>
<?xml version="1.0" encoding="Big5"?>
<?xml version="1.0" encoding="ISO-8859-5"?>
```

These are not grammatical text declarations:

```
<?xml?>
<?xml encoding="Big5"?>
<?xml encoding="Big5" version="1.0" ?>
<?xml version="1.0" standalone="yes"? encoding="ISO-8859-1" >
<?xml version="1.0" styles="poems.css">
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<?xml version="1.0" standalone="no"? >
```

Well-formed External Parsed Entity

[78] extParsedEnt ::= TextDecl? content

Production [78] says that an external general parsed entity consists of an optional text declaration followed by content (Production [43]). The main point of this production is that the content may not include a DTD or any markup declarations.

[79] extPE ::= TextDecl? extSubsetDecl

Production [79] says that an external parameter entity consists of an optional text declaration followed by an external subset declaration (Production [31]).

Encoding Declaration

[80] EncodingDecl ::= S 'encoding' Eq ('"' EncName '"' | "'" EncName "'")

Production [80] defines an encoding declaration as whitespace (Production [3]), followed by the string "`encoding`" followed by an equals sign (Production [25]),

followed by the name of the encoding (Production [81]) enclosed in either single or double quotes. These are all legal encoding declarations:

```
encoding="Big5"
encoding="ISO-8859-5"
encoding = "Big5"
encoding = "ISO-8859-5"
encoding= 'Big5'
encoding= 'ISO-8859-5'
```

These are not legal encoding declarations:

```
encoding "Big5"
encoding="ISO-8859-51'
encoding = "Big5
encoding = 'ISO-8859-5"
```

[81] EncName ::= [A-Za-z] ([A-Za-z0-9._] | '-')*

Production [81] says the name of an encoding begins with one of the ASCII letters *A* through *Z* or *a* through *z*, followed by any number of ASCII letters, digits, periods, underscores, and hyphens. These are legal encoding names:

```
ISO-8859-1
Big5
GB2312
```

These are ungrammatical encoding names:

```
ISO 8859-1
Big5 Chinese
GB 2312
____851
```

Notation Declarations

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

Production [82] defines a notation declaration as the literal string "`<!NOTATION`", followed by whitespace (Production [3]), followed by an XML name (Production[5]) for the notation, followed by whitespace, followed by either an external ID (Production [75]) or a public ID (Production [83]), optionally followed by whitespace, followed by the literal string ">". These are grammatical notation declarations:

```
<!NOTATION GIF SYSTEM "image/gif">
<!NOTATION GIF SYSTEM "image/gif" >
<!NOTATION GIF PUBLIC
   "-//IETF//NONSGML Media Type image/gif//EN"
   "http://www.isi.edu/in-notes/iana/assignments/media-
types/image/gif">
```

These are not grammatical notation declarations:

```
<! NOTATION GIF SYSTEM "image/gif" >
< !NOTATION GIF SYSTEM "image/gif" >
<!NOTATION GIF "image/gif">
<!NOTATION GIF SYSTEM image/gif>
<!NOTATION GIF PUBLIC
   "http://www.isi.edu/in-notes/iana/assignments/media-
types/image/gif">
```

[83] PublicID ::= 'PUBLIC' S PubidLiteral

Production [83] defines a public ID as the literal string `PUBLIC`, followed by whitespace (Production [3]), followed by a public ID literal (Production [12]). These are grammatical public IDs:

```
PUBLIC "-//IETF//NONSGML Media Type image/gif//EN"
PUBLIC "ISO 8879:1986//ENTITIES Added Latin 1//EN//XML"
```

These are ungrammatical public IDs:

```
PUBLIC -//IETF//NONSGML Media Type image/gif//EN
PUBLIC 'ISO 8879:1986//ENTITIES Added Latin 1//EN//XML"
```

Characters

[84] Letter ::= BaseChar | Ideographic

Production [84] defines a letter as either a base character or an ideographic character.

[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]

Production [85] lists the base characters. These are the defined Unicode characters that are alphabetic but not punctuation marks or digits. For instance, `A-Z` and `a-z` are base characters but `0-9` and `!`, `"`, `#`, `$`, and so forth, are not. This list is so long because it contains characters from not only the English alphabet but also Greek, Hebrew, Arabic, Cyrillic, and all the other alphabetic scripts Unicode supports.

[86] Ideographic ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]

Production [86] lists the ideographic characters. `#x4E00-#x9FA5` are Unicode's Chinese-Japanese-Korean unified ideographs. `#x3007` is the ideographic number zero. Characters `#x3021` through `#x3029` are the Hangzhou style numerals.

[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

Production [87] lists the combining characters. These are characters that are generally combined with the preceding character to form the appearance of a single character. For example, character ̀ is the combining accent grave. The letter a (a) followed by a combining accent grave would generally be rendered as à and occupy only a single character width, even in a monospaced font.

[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]

Production [88] lists the characters that are considered to be digits. These include not only the usual European numerals 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, but also the Arabic-Indic digits used primarily in Egyptian Arabic, the Eastern Arabic Indic digits used in Persian and Urdu, and many more.

[89] Extender ::= #x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640 | #x0E46 | #x0EC6 | #x3005 | [#x3031-#x3035] | [#x309D-#x309E] | [#x30FC-#x30FE]

Production [89] lists the characters that are considered to be extenders. In order, these characters are the middle dot, the modifier letter triangular colon, the modifier letter half-triangular colon, the Greek middle dot, the Arabic tatweel, the Thai maiyamok, the Lao ko la, the ideographic iteration mark, five Japanese Kana repeat marks, the Japanese Hiragana iteration mark and voiced iteration mark, and the Japanese Katakana and Hiragana sound mark and prolonged sound mark. An extender is a character that's neither a letter nor a combining character, but that is nonetheless included in words as part of the word. The closest equivalent in English is perhaps the hyphen used in words like *mother-in-law* or *well-off*. However, the hyphen is not considered to be an extender in XML.

#x0387, the triangular colon, has been removed from the extender class in the latest Unicode errata sheet, but this has not yet trickled down into XML.

Well-formedness Constraints

According to the XML 1.0 specification, an XML document is well-formed 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.

This reference topic is designed to help you understand the second of those requirements and more quickly determine whether your documents meet that requirement.

What is a Well-formedness Constraint?

As you read the BNF grammar for XML 1.0, you should notice that some productions have associated well-formedness constraints, abbreviated WFC. For example, here's production [40]:

```
[40] STag ::= '<' Name (S Attribute)* S? '>'
          [ WFC: Unique Att Spec ]
```

What follows "WFC: " is the name of the well-formedness constraint, "Unique Att Spec" in this example. Generally, if you look a little below the production you'll find the constraint with the given name. For example, looking below Production [40] you'll find this:

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

This says that a given attribute may not appear more than once in a single element. For example, the following tag violates well-formedness:

```
<P COLOR="red" COLOR="blue">
```

Well-formedness constraints are used for requirements like this that are difficult or impossible to state in the form of a BNF grammar. As XML parsers read a document, they must not only check that the document matches the document production of

the BNF grammar, they must also check that it satisfies all well-formedness constraints.

There are also validity constraints that must be satisfied by valid documents. XML processors are not required to check validity constraints if they do not wish to, however. Most validity constraints deal with declarations in the DTD. Validity constraints are covered later in this appendix.

Productions Associated with Well-formedness Constraints

This section lists the productions associated with well-formedness constraints and explains those constraints. Most productions don't have any well-formedness constraints; so most productions are not listed here. The complete list of productions is found in the BNF Grammar portion of this appendix.

[29] markupdecl ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment

[Well-formedness Constraint: PEs in Internal Subset]

This well-formedness constraint states that parameter entity references defined in the *internal* DTD subset cannot be used inside a markup declaration. For example, the following is illegal:

```
<!ENTITY % INLINES SYSTEM "(I | EM | B | STRONG | CODE)*">
<!ELEMENT P %INLINES; >
```

On the other hand, the above would be legal in the *external* DTD subset.

[39] element ::= EmptyElemTag | STag content ETag

[Well-Formedness Constraint: Element Type Match]

This well-formedness constraint simply says that the name of the start tag must match the name of the corresponding end tag. For instance, these elements are well-formed:

```
<TEST>content</TEST>
<test>content</test>
```

However, these are not:

```
<TEST>content</test>
<Fred>content</Ethel>
```

[40] STag ::= '<' Name (S Attribute)* S? '>'

[Well-formedness Constraint: Unique Att Spec]

This constraint says that a given attribute may not appear more than once in a single element. For example, the following tags violates well-formedness:

```
<P COLOR="red" COLOR="blue">
<P COLOR="red" COLOR="red">
```

The problem is that the `COLOR` attribute appears twice in the same tag. In the second case, it doesn't matter that the value is the same both times. It's still malformed. The following two tags are well-formed because the attributes have slightly different names:

```
<P COLOR1="red" COLOR2="blue">
<P COLOR1="red" COLOR2="red">
```

[41] Attribute ::= Name Eq AttValue

[Well-formedness Constraint: No External Entity References]

This constraint says that attribute values may not contain entity references that point to data in other documents. For example, consider this attribute:

```
<BOX COLOR="&RED;" />
```

Whether this is well-formed depends on how the entity `RED` is defined. If it's completely defined in the DTD, either in the internal or external subset, this tag is acceptable. For example:

```
<!ENTITY RED "#FF0000">
```

However, if the `RED` entity is defined as an external entity that refers to a separate file, then it's not well defined. In that case, the `ENTITY` declaration would look something like this:

```
<!ENTITY RED SYSTEM "red.txt" NDATA COLOR>
```

Note that this constraint applies to parsed entities. It does not apply to unparsed entities given as the value of an attribute of type `ENTITY` or `ENTITIES`. For example, the following is legal even though `RED` is an external entity used as an attribute value.

```
<?xml version="1.0"?>
<!DOCTYPE EXAMPLE [
  <!ELEMENT EXAMPLE ANY>
  <!NOTATION COLOR SYSTEM "x-color">
```

```
  <!ENTITY RED SYSTEM "red.txt" NDATA COLOR>
  <!ATTLIST EXAMPLE HUE ENTITY #REQUIRED>
]>
<EXAMPLE HUE="RED">
testing 1 2 3
</EXAMPLE>
```

[Well-formedness Constraint: No < in Attribute Values]

This constraint is very simple. The less-than sign (<) cannot be part of an attribute value. For example, the following tags are malformed:

```
<BOX COLOR="<6699FF>" />
<HALFPLANE REGION="X < 8" />
```

Technically, these tags are already forbidden by Production [10]. The real purpose of this constraint is to make sure that a < doesn't slip in via an external entity reference. The correct way to embed a < in an attribute value is to use the `<` entity reference like this:

```
<BOX COLOR="&lt;6699FF>" />
<HALFPLANE REGION="X &lt; 8" />
```

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

[Well-formedness Constraint: Unique Att Spec]

This constraint says that a given attribute may not appear more than once in a single empty element. For example, the following tags violate well-formedness:

```
<P COLOR="red" COLOR="blue" />
<P COLOR="red" COLOR="red" />
```

Take a look at the second example. Even the purely redundant attribute violates well-formedness.

[60] DefaultDecl ::= '#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)

[Well-formedness Constraint: No < in Attribute Values]

This is the same constraint seen in Production [41]. This merely states that you can't place a < in a default attribute value in a `<!ATTLIST>` declaration. For example, these are malformed attribute declarations:

```
<!ATTLIST RECTANGLE COLOR  CDATA "<330033>">
<!ATTLIST HALFPLANE REGION CDATA "X < 0" />
```

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

[Well-formedness Constraint: Legal Character]

This constraint says that characters referred to by character references must be legal characters if they were simply typed in the document. Character references are convenience for inputting legal characters that are difficult to type on a particular system. They are not a means to input otherwise forbidden characters.

The definition of a legal character is given by Production [2]:

```
[2] Char ::= #x9 | #xA | #xD | [#x20-#xD7FF]
                | [#xE000-#xFFFD]  | [#x10000-#x10FFFF]
```

The main items of interest here are the characters not included. Specifically, these are the non-printing ASCII control characters of which the most common are the bell, vertical tab, and formfeed; the surrogates block from #xD800 to #xDFFF, and the non-character #xFFFE.

[68] EntityRef ::= '&' Name ';'

[Well-formedness Constraint: Entity Declared]

The intent of this well-formedness constraint is to make sure that all entities used in the document are declared in the DTD using `<!ENTITY>`. However, there are two loopholes:

1. The five predefined entities: `<`, `&apos;`, `>`, `&quot;`, and `&` are not required to be declared, although they may be.
2. A non-validating processor can allow undeclared entities if it's possible they may have been declared in the external DTD subset (which a non-validating processor is not required to read). Specifically, it's possible that entities were declared in an external DTD subset if:
   a. The standalone document declaration does not have `standalone="yes"`.
   b. The DTD contains at least one parameter entity reference.

If either of these conditions is violated, then undeclared entities (other than the five in loophole one) are not allowed.

This constraint also specifies that, if entities are declared, they must be declared before they're used.

[Well-formedness Constraint: Parsed Entity]

This constraint states that entity references may only contain the names of parsed entities. Unparsed entity names are only contained in attribute values of type `ENTITY` or `ENTITIES`. For example, this is a malformed document:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT DOCUMENT ANY>
  <!ENTITY LOGO SYSTEM "http://metalab.unc.edu/xml/logo.gif"
     NDATA GIF>
<!NOTATION GIF SYSTEM "image/gif">
]>
<DOCUMENT>
  &LOGO;
</DOCUMENT>
```

This is the correct way to embed the unparsed entity `LOGO` in the document:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DOCUMENT [

  <!ELEMENT DOCUMENT ANY>
  <!ENTITY LOGO SYSTEM "http://metalab.unc.edu/xml/logo.gif"
     NDATA GIF>
  <!NOTATION GIF SYSTEM "image/gif">
  <!ELEMENT IMAGE EMPTY>
  <!ATTLIST IMAGE SOURCE ENTITY #REQUIRED>

]>
<DOCUMENT>
  <IMAGE SOURCE="LOGO" />
</DOCUMENT>
```

[Well-formedness Constraint: No Recursion]

This well-formedness constraint states that a parsed entity cannot refer to itself. For example, this open source classic is malformed:

```
<!ENTITY GNU "&GNU;'s not Unix!">
```

Circular references are a little trickier to spot, but are equally illegal:

```
<!ENTITY LEFT  "Left &RIGHT; Left!">
<!ENTITY RIGHT "Right &LEFT; Right!">
```

Note that it's only the recursion that's malformed, not the mere use of one entity reference inside another. The following is perfectly fine because although the

COPY99 entity depends on the ERH entity, the ERH entity does not depend on the COPY99 entity.

```
<!ENTITY ERH    "Elliotte Rusty Harold">
<!ENTITY COPY99 "Copyright 1999 &ERH;">
```

[69] PEReference ::= '%' Name ';'

[Well-formedness Constraint: No Recursion]

This is the same constraint that applies to Production [68]. Parameter entities can't recurse any more than general entities can. For example, this entity declaration is also malformed:

```
<!ENTITY % GNU "%GNU;'s not Unix!">
```

And this is still illegal:

```
<!ENTITY % LEFT  "Left  %RIGHT; Left!">
<!ENTITY % RIGHT "Right %LEFT; Right!">
```

[Well-formedness Constraint: In DTD]

This well-formedness constraint requires that parameter entity references can only appear in the DTD. They may not appear in the content of the document or anywhere else that's not the DTD.

Validity Constraints

This reference topic is designed to help you understand what is required in order for an XML document to be *valid*. Validity is often useful, but is not always required. You can do a lot with simply well-formed documents, and such documents are often easier to write because there are fewer rules to follow. For valid documents, you must follow the BNF grammar, the well-formedness constraints, *and* the validity constraints discussed in this section.

What Is a Validity Constraint?

A validity constraint is a rule that must be adhered to by a valid document. Not all XML documents are, or need to be, valid. It is not necessarily an error for a document to fail to satisfy a validity constraint. Validating processors have the option of reporting violations of these constraints as errors, but they do not have to. All syntax (BNF) errors and well-formedness violations must still be reported however.

Only documents with DTDs may be validated. Almost all the validity constraints deal with the relationships between the content of the document and the declarations in the DTD.

Validity Constraints in XML 1.0

This section lists and explains all of the validity constraints in the XML 1.0 standard. These are organized according to the BNF rule each applies to.

[28] doctypedecl ::= '<!DOCTYPE' S Name (S ExternalID)? S? ('[' (markupdecl | PEReference | S)* ']' S?)? '>'

Validity Constraint: Root Element Type

This constraint simply states that the name given in the `DOCTYPE` declaration must match the name of the root element. In other words, the bold parts below have to all be the same.

```
<?xml version="1.0"?>
<!DOCTYPE ROOTNAME [
  <!ELEMENT ROOTNAME ANY>
]>
<ROOTNAME>
  content
</ROOTNAME>
```

It's also true that the root element must be declared — that's done by the line in italic — however that declaration is required by a different validity constraint, not this one.

[29] markupdecl ::= elementdecl | AttlistDecl | EntityDecl | NotationDecl | PI | Comment

Validity Constraint: Proper Declaration/PE Nesting

This constraint requires that a markup declaration contain or be contained in one or more parameter entities, but that it may not be split across a parameter entity. For example, consider this element declaration:

```
<!ELEMENT PARENT ( FATHER | MOTHER )>
```

The parameter entity declared by the following entity declaration is a valid substitute for the content model, because the parameter entity contains both the < and the >:

```
<!ENTITY % PARENT_DECL "<!ELEMENT PARENT ( FATHER | MOTHER )>">
```

Given that entity, you can rewrite the element declaration like this:

```
%PARENT_DECL;
```

This is valid because the parameter entity contains both the < and the >. Another option is to include only part of the element declaration in the parameter entity. For example, if you had many elements whose content model was `( FATHER | MOTHER)`, then it might be useful to do something like this:

```
<!ENTITY % PARENT_TYPES "( FATHER | MOTHER )">
<!ELEMENT PARENT %PARENT_TYPES;>
```

Here, neither the < or > is included in the parameter entity. You cannot enclose one of the angle brackets in the parameter entity without including its mate. The following, for example, is invalid, even though it appears to expand into a legal element declaration:

```
<!ENTITY % PARENT_TYPES "( FATHER | MOTHER )>">
<!ELEMENT PARENT %PARENT_TYPES;
```

Note that the problem is *not* that the parameter entity's replacement text contains a > character. That's legal (unlike the use of a < character, which would be illegal in an internal parameter entity declaration). The problem is how the > character is used to terminate an element declaration that began in another entity.

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

Validity Constraint: Standalone Document Declaration

In short, this constraint says that a document must have a standalone document declaration with the value no (`standalone="no"`) if any other files are required to process this file and determine its validity. Mostly this affects external DTD subsets linked in through parameter entities. This is the case if any of the following are true:

- An entity used in the document is declared in an external DTD subset.
- The external DTD subset provides default values for attributes that appear in the document without values.
- The external DTD subset changes how attribute values in the document may be normalized.
- The external DTD subset declares elements whose children are only elements (no character data or mixed content) when those children may themselves contain whitespace.

[39] element ::= EmptyElemTag | STag content ETag

Validity Constraint: Element Valid

This constraint simply states that this element matches an element declaration in the DTD. More precisely one of the following conditions must be true:

1. The element has no content and the element declaration declares the element `EMPTY`.
2. The element contains only child elements that match the regular expression in the element's content model.
3. The element is declared to have mixed content, and the element's content contains character data and child elements that are declared in the mixed-content declaration.
4. The element is declared `ANY`, and all child elements are declared.

[41] Attribute ::= Name Eq AttValue

Validity Constraint: Attribute Value Type

This constraint simply states that the attribute's name must have been declared in an `ATTLIST` declaration in the DTD. Furthermore, the attribute value must match the declared type in the `ATTLIST` declaration.

[45] elementdecl ::= '<!ELEMENT' S Name S contentspec S? '>'

Validity Constraint: Unique Element Type Declaration

An element cannot be declared more than once in the DTD, whether the declarations are compatible or not. For example, this is valid:

```
<!ELEMENT EM (#PCDATA)>
```

This, however, is not valid:

```
<!ELEMENT EM (#PCDATA)>
<!ELEMENT EM (#PCDATA | B)>
```

Neither is this valid:

```
<!ELEMENT EM (#PCDATA)>
<!ELEMENT EM (#PCDATA)>
```

This is most likely to cause problems when merging external DTD subsets from different sources that both declare some of the same elements. To a limited extent, namespaces can help resolve this.

[49] choice ::= '(' S? cp (S? '|' S? cp)* S? ')'

Validity Constraint: Proper Group/PE Nesting

This constraint states that a choice may contain or be contained in one or more parameter entities, but that it may not be split across a parameter entity. For example, consider this element declaration:

```
<!ELEMENT PARENT ( FATHER | MOTHER )>
```

The parameter entity declared by the following entity declaration is a valid substitute for the content model because the parameter entity contains both the (and the):

```
<!ENTITY % PARENT_TYPES "( FATHER | MOTHER )">
```

That is, you can rewrite the element declaration like this:

```
<!ELEMENT PARENT %PARENT_TYPES;>
```

This is valid because the parameter entity contains both the (and the). Another option is to include only the child elements, but leave out both parentheses. For example:

```
<!ENTITY % PARENT_TYPES " FATHER | MOTHER ">
<!ELEMENT PARENT ( %PARENT_TYPES; )>
```

The advantage here is that you can easily add additional elements not defined in the parameter entity. For example:

```
<!ELEMENT PARENT ( UNKNOWN | %PARENT_TYPES; ) >
```

What you cannot do, however, is enclose one of the parentheses in the parameter entity without including its mate. The following, for example, is invalid, even though it appears to expand into a legal element declaration.

```
<!ENTITY % FATHER " FATHER )">
<!ENTITY % MOTHER " ( MOTHER | ">
<!ELEMENT PARENT %FATHER; %MOTHER; ) >
```

The problem in this example is the ELEMENT declaration, not the ENTITY declarations. It is valid to declare the entities as done here. It's their use in the context of a choice that makes them invalid.

[50] seq ::= '(' S? cp (S? ',' S? cp)* S? ')'

Validity Constraint: Proper Group/PE Nesting

This is exactly the same constraint as above, except now it's applied to sequences rather than choices. It requires that a sequence may contain or be contained in one or more parameter entities, but it may not be split across a parameter entity. For example, consider this element declaration:

```
<!ELEMENT ADDRESS ( NAME, STREET, CITY, STATE, ZIP )>
```

The parameter entity declared by the following entity declaration is a valid substitute for the content model because the parameter entity contains both the (and the):

```
<!ENTITY % SIMPLE_ADDRESS "( NAME, STREET, CITY, STATE, ZIP )">
```

That is, you can rewrite the element declaration like this:

```
<!ELEMENT ADDRESS %SIMPLE_ADDRESS;>
```

This is valid because the parameter entity contains both the (and the). Another option is to include only the child elements, but leave out both parentheses. For example:

```
<!ENTITY % SIMPLE_ADDRESS " NAME, STREET, CITY, STATE, ZIP ">
<!ELEMENT ADDRESS( %SIMPLE_ADDRESS; )>
```

The advantage here is that you can easily add additional elements not defined in the parameter entity. For example:

```
<!ENTITY % INTERNATIONAL_ADDRESS " NAME, STREET, CITY,
  PROVINCE?, POSTAL_CODE?, COUNTRY ">
<!ELEMENT ADDRESS ( (%SIMPLE_ADDRESS;)
                  | (%INTERNATIONAL_ADDRESS;) ) >
```

What you cannot do, however, is enclose one of the parentheses in the parameter entity without including its mate. The following, for example, is invalid, even though it appears to expand into a legal element declaration:

```
<!ENTITY % SIMPLE_ADDRESS_1 "( NAME, STREET, ">
<!ENTITY % SIMPLE_ADDRESS_2 "CITY, STATE, ZIP)">
<!ELEMENT ADDRESS %SIMPLE_ADDRESS_1; %SIMPLE_ADDRESS_2; ) >
```

The problem in this example is the ELEMENT declaration, not the ENTITY declarations. It is valid to declare the entities like this. It's their use in the context of a sequence that makes them invalid.

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

Validity Constraint: Proper Group/PE Nesting

This is exactly the same constraint as above, except now it's applied to mixed content rather than choices or sequences. It requires that a mixed-content model may contain or be contained in a parameter entity, but it may not be split across a parameter entity. For example, consider this element declaration:

```
<!ELEMENT P ( #PCDATA | I | EM | B | STRONG )>
```

The parameter entity declared by the following entity declaration is a valid substitute for the content model because the parameter entity contains both the (and the):

```
<!ENTITY % INLINES "( #PCDATA | I | EM | B | STRONG )">
```

That is, you can rewrite the element declaration like this:

```
<!ELEMENT P %INLINES;>
```

This is valid because the parameter entity contains both the (and the). Another option is to include only the content particles, but leave out both parentheses. For example:

```
<!ENTITY % INLINES " #PCDATA | I | EM | B | STRONG ">
<!ELEMENT P ( %INLINES; ) >
```

The advantage here is that you can easily add additional elements not defined in the parameter entity. For example:

```
<!ELEMENT QUOTE ( %INLINES; | SPEAKER ) >
```

What you cannot do, however, is enclose one of the parentheses in the parameter entity without including its mate. The following, for example, is invalid, even though it appears to expand into a legal element declaration:

```
<!ENTITY % INLINES1 " I | EM | B | STRONG )">
<!ENTITY % INLINES2 " ( #PCDATA | SPEAKER | ">
<!ELEMENT QUOTE %INLINES1; %INLINES2; ) >
```

The problem in this example is the `ELEMENT` declaration, not the `ENTITY` declarations. It is valid to declare the entities as is done here. It's their use in the context of a choice (or sequence) that makes them invalid.

Validity Constraint: No Duplicate Types

No element can be repeated in a mixed-content declaration. For example, the follwing is invalid:

```
( #PCDATA | I | EM | I | EM )
```

There's really no reason to write a mixed-content declaration like this, but at the same time, it's not obvious what the harm is. Interestingly, pure choices do allow content models like this:

```
( I | EM | I | EM )
```

It only becomes a problem when `#PCDATA` gets mixed in.

This choice is ambiguous—that is, when the parser encounters an `I` or an `EM`, it doesn't know whether it matches the first or the second instance in the content model. So although legal, some parsers will report it as an error, and it should be avoided if possible.

[56] TokenizedType ::= 'ID' | 'IDREF' | 'IDREFS' | 'ENTITY' | 'ENTITIES' | 'NMTOKEN' | 'NMTOKENS'

Validity Constraint: ID

Attribute values of ID type must be valid XML names (Production [5]). Furthermore, a single name cannot be used more than once in the same document as the value of an ID type attribute. For example, this is invalid given that `ID` is declared to be ID:

```
<BOX ID="B1" WIDTH="50" HEIGHT="50" />
<BOX ID="B1" WIDTH="250" HEIGHT="250" />
```

This is also invalid because XML names cannot begin with numbers:

```
<BOX ID="1" WIDTH="50" HEIGHT="50" />
```

This is valid if `NAME` does not have type `ID`:

```
<BOX ID="B1" WIDTH="50" HEIGHT="50" />
<BOX NAME="B1" WIDTH="250" HEIGHT="250" />
```

On the other hand, that example is invalid if `NAME` does have type `ID`, even though the `NAME` attribute is different from the `ID` attribute. Furthermore, the following is invalid if `NAME` has type ID, even though two different elements are involved:

```
<BOX NAME="FRED" WIDTH="50" HEIGHT="50" />
<PERSON NAME="FRED" />
```

ID attribute values must be unique across all elements and ID attributes, not just a particular class of, or attributes of, a particular class of elements.

Validity Constraint: One ID per Element Type

Each element can have at most one attribute of type ID. For example, the following is invalid:

```
<!ELEMENT PERSON (ANY) >
<!ATTLIST PERSON SS_NUMBER   ID #REQUIRED>
<!ATTLIST PERSON EMPLOYEE_ID ID #REQUIRED>
```

Validity Constraint: ID Attribute Default

All attributes of ID type must be declared #IMPLIED or #REQUIRED. #FIXED is not allowed. For example, the following is invalid:

```
<!ATTLIST PERSON SS_NUMBER ID #FIXED "SS123-45-6789">
```

The problem is that if there's more than one PERSON element in the document, the ID validity constraint will automatically be violated.

Validity Constraint: IDREF

The IDREF validity constraint specifies that an attribute value of an IDREF type attribute must be the same as the value of an ID type attribute of an element in the document. Multiple IDREF attributes in the same or different elements may point to a single element. ID attribute values must be unique (at least among other ID attribute values in the same document), but IDREF attributes do not need to be.

Additionally, attribute values of type IDREFS must be a whitespace-separated list of ID attribute values from elements in the document.

Validity Constraint: Entity Name

The value of an attribute whose declared type is ENTITY must be the name of an unparsed general (non-parameter) entity declared in the DTD, whether in the internal or external subset.

The value of an attribute whose declared type is ENTITIES must be a whitespace-separated list of the names of unparsed general (non-parameter) entities declared in the DTD, whether in the internal or external subset.

Validity Constraint: Name Token

The value of an attribute whose declared type is NMTOKEN must match the NMTOKEN production of XML (Production [7]). That is, it must be composed of one or more name characters. It differs from an XML name in that it may start with a digit, a period, a hyphen, a combining character, or an extender.

The value of an attribute whose declared type is NMTOKENS must be a whitespace-separated list of name tokens. For example, this is a valid element with a COLORS attribute of type NMTOKENS:

```
<BOX WIDTH="50" HEIGHT="50" COLORS="red green blue" />
```

This is an invalid element with a COLORS attribute of type NMTOKENS:

```
<BOX WIDTH="50" HEIGHT="50" COLORS="red, green, blue" />
```

[58] NotationType ::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'

Validity Constraint: Notation Attributes

The value of an attribute whose declared type is NOTATION must be the name of a notation that's been declared in the DTD.

[59] Enumeration ::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'

Validity Constraint: Enumeration

The value of an attribute whose declared type is ENUMERATION must be a whitespace-separated list of name tokens. These name tokens do not necessarily have to be the names of anything declared in the DTD or elsewhere. They simply have to match the NMTOKEN production (Production [7]). For example, this is an invalid enumeration because commas rather than whitespace are used to separate the name tokens:

```
( red, green, blue)
```

This is an invalid enumeration because the name tokens are enclosed in quote marks:

```
( "red" "green" "blue")
```

Neither commas nor quote marks are valid name characters so there's no possibility for these common mistakes to be misinterpreted as a whitespace-separated list of unusual name tokens.

[60] DefaultDecl ::= '#REQUIRED' | '#IMPLIED' | (('#FIXED' S)? AttValue)

Validity Constraint: Required Attribute

If an attribute of an element is declared to be `#REQUIRED`, then it is a validity error for any instance of the element not to provide a value for that attribute.

Validity Constraint: Attribute Default Legal

This common-sense validity constraint merely states that any default attribute value provided in an `ATTLIST` declaration must satisfy the constraints for an attribute of that type. For example, the following is invalid because the default value, `UNKNOWN`, is not one of the choices given by the content model.

```
<!ATTLIST CIRCLE VISIBLE (TRUE | FALSE) "UNKNOWN">
```

`UNKNOWN` would be invalid for this attribute whether it was provided as a default value or in an actual element like the following:

```
<CIRCLE VISIBLE="UNKNOWN" />
```

Validity Constraint: Fixed Attribute Default

This common-sense validity constraint merely states that if an attribute is declared `#FIXED` in its `ATTLIST` declaration, then that same `ATTLIST` declaration must also provide a default value. For example, the following is invalid:

```
<!ATTLIST AUTHOR COMPANY CDATA #FIXED>
```

Here's a corrected declaration:

```
<!ATTLIST AUTHOR COMPANY CDATA #FIXED "TIC">
```

[68] EntityRef ::= '&' Name ';'

Validity Constraint: Entity Declared

This constraint expands on the well-formedness constraint of the same name. In a valid document, all referenced entities must be defined by `<!ENTITY>` declarations in the DTD. Definitions must precede any use of the entity they define.

The loophole for `standalone="no"` documents that applies to merely well-formed documents is no longer available. The loophole for the five predefined entities: `&lt;`, `&apos;`, `&gt;`, `&quot;`, and `&amp;` is still available. However, it is recom-

mended that you declare them, even though you don't absolutely have to. Those declarations would look like this:

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

[69] PEReference ::= '%' Name ';'

Validity Constraint: Entity Declared

This is the same constraint as the previous one, merely applied to parameter entity references instead of general entity references.

[76] NDataDecl ::= S 'NDATA' S Name

Validity Constraint: Notation Declared

The name used in a notation data declaration (which is in turn used in an entity definition for an unparsed entity) must be the name of a notation declared in the DTD. For example, the following document is valid. However, if you take away the line declaring the `GIF` notation (shown in bold) it becomes invalid.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE DOCUMENT [
  <!ELEMENT DOCUMENT ANY>
  <!ENTITY LOGO SYSTEM "http://metalab.unc.edu/xml/logo.gif"
     NDATA gif>
  <!NOTATION GIF SYSTEM "image/gif">
]>
<DOCUMENT>
  &LOGO;
</DOCUMENT>
```