APPENDIX

More on markup and XML

We learned in Chapter 4 about the extensible markup language, XML, and the family of open standards that have built up around it. XML is a flexible framework for describing document structure and metadata. In fact, there are other developments centered upon XML, which will probably also have far-reaching effects on digital libraries of the future. We describe some of them in this chapter. One concerns naming: how are we to name documents in this brave new Web world, where so much changes so quickly? Another concerns linking. The "hyperlink" has served us well as a central feature of the Web, but plain hyperlinks are rather rudimentary—far more can be done. Historically hypertext predates the Web, and researchers were busily pursuing the idea when the Web burst onto the scene in 1993. It was like a supernova exploding in the sky nearby: the effect was so dramatic that other work in the area was outshone, virtually obliterated. We were blinded! But the Web as we know it uses hyperlinks in a primitive way, and by now our eyes have accommodated to the point where we can see what to do next. A final issue is data types. In essence the document type description (DTD) describes a data type that represents a document, but more general data needs richer facilities. Also, DTDs, although they resemble it closely, are not written in proper XML, an inelegance that a later development has rectified.

The query is another important part of a digital library. Why should we have to learn different ways of querying when we move from one digital library, or Web search engine, to another? There are standard ways of expressing queries, and we have already met one of them in Chapter 3, the common command language (CCL), hat arose out of the library world, and variants have been in use for library catalog searching for years. Here we meet XQuery, part of the brave new world of XML and provides an exceptionally powerful framework for issuing queries and having the results composed into lists and even documents—documents that are generated dynamically on demand by accessing the contents of a digital library or any other information collection.

Chapter 5 introduced XML and discussed some extensions that allow XML documents to be presented in ways comparable to HTML: the stylesheet languages CSS and XSL. However, as we intimated there, there are other extensions to XML that provide more advanced data representation and manipulation facilities. Because of XML's importance for future digital libraries, we expand on these developments here. But first we need to say something about naming resources on the Internet and namespaces in XML.

Names and Namespaces

Most people are acquainted with URLs—indeed it is hard to talk to friends about the Internet without mentioning them. The acronym stands for "uniform resource locator," where a "resource" is a piece of information, typically a Web page. However, a URL is useless if the resource it identifies is unavailable—and we know from bitter experience that this happens all the time. The problem is that a "locator" is a kind of address, and things often move around on the Web when sites are reorganized or information changes hands. People lose touch when they change addresses frequently, and the same is true of information.

What is needed is a way of naming resources so that, wherever they are, they can be found. Since the early 1990s people have debated how to identify resources on the Internet in a way that is both independent of location and persistent over time. But naming is a difficult business, and although technical people prefer technical solutions, making a name "persistent" is really an institutional matter rather than a technical one. For example, a "persistent URL" or PURL is one that is backed up by an institutional commitment to availability over an extended period of time.[1]

The upshot of this debate has been a way of naming resources called a "universal resource name" or URN. Each URN includes within it a "naming authority" that is able to resolve the URN and provide the named information (or the address where it is currently stored).

Together URLs and URNs are types of URI, or "uniform resource identifier," and the term *URL* is now officially deprecated. In summary, there are three ways of naming resources.

> *URI (uniform resource identifier)*: the generic set of all names or addresses that are short strings referring to resources
>
> *URL (uniform resource locator)*: an informal term, no longer used in technical specifications, that is associated with popular URI schemes such as http, ftp, and mailto
>
> *URN (uniform resource name)*: either a URL that has an institutional commitment to persistence and availability (the PURL mentioned earlier), or a particular scheme intended to serve as a persistent, location-independent resource identifier

While URL is still widely used in practice, official standards increasingly use the term URI to specify such entities. As we said earlier, naming is a difficult business.

Namespaces help you to avoid mixing up XML tags that are designed for different purposes. We already encountered them in Chapter 5 (under "Basic XSL" in Section 5.3). For example, the style sheets in Figures 5.10 through 5.13 began with the lines

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:fo="http://www.w3.org/1999/XSL/Format"
```

The *xmlns* qualifier sets up a namespace, so this example creates two namespaces called *xsl* and *fo* (for XML stylesheet language and Formatting Object, respectively). The same tag can occur in each namespace and yet retain distinct meanings. For example, <*xsl:block*> specifies an XSL tag called *block*, while <*fo:block*> specifies a Formatting Object tag of the same name. Namespaces can be used in XML documents and DTDs, as well as in XSL style sheets.

Namespaces have further features, which are used in examples in this chapter. First, namespace declarations (such as the previous two lines) can be embedded in any node of the tree that represents the document, not just the root node as they were in Figures 5.10 through 5.15. The fact that tags are nested ensures that each namespace's scope is well defined. Thus different blocks can make use of different namespaces.

You can also define a *global* namespace by simply omitting the label that follows the colon, as in

```
xmlns="http://www.openarchives.org/"
```

Tags from this namespace do not need to be qualified with a prefix—in fact there is no prefix name to use! The global namespace can be redefined at different points in the document: unqualified tags use the current global namespace. This encourages brevity.

Namespace prefixes are used to qualify tags. They can also be used to qualify attributes. Within a single tag, different attributes may even use different namespaces: they can be freely mixed. For example,

```
<MyTag a:att="XXX" b:att="YYY">Some text</MyTag>
```

has one attribute from the *a* namespace and a second from the *b* namespace.

Links

XML files can define connections between resources using two supporting standards, XLink and XPointer. These provide a linking scheme that is far more powerful than the unidirectional hyperlinks of HTML. Of course power does not necessarily guarantee success. The Text Encoding Initiative mentioned at the end of Section 5.2 (Chapter 5) incorporates a very rich linking ability—in fact XLink is based on it—yet because of its complexity, it is not used as much as it might be.

We met the XPath mechanism in Chapter 5. It is a way of selecting parts of documents, and we used it in conjunction with transformations in the XML stylesheet language to manipulate parts of the document tree. XPointer is a development of XPath that provides a finer degree of contol over the part or parts of the document being selected. XLink is a general way of connecting selected resources. Together XPointer and XLink provide a foundation for bidirectional links, multiway links, and annotated links. For example, they might be used to specify an algorithm for picking out a destination such as "the third section heading in the Appendix."

In this chapter we will continue to use the UN example that was introduced in Chapter 5 to illustrate XML and style sheets (see Figures 5.3–5.13). Figure 8.1 augments it to demonstrate XLink usage. A new <*Intro*> element has been added to include introductory text, which will require corresponding updates to the DTD and any style sheets used to display the example. However, we do not elaborate on these because they are not the focus of the example.

Within the introductory text is an <*HrefLink*> element that provides a link to the United Nations Web site rather like the HTML anchor hyperlink <*a href*=…>. The tag name *HrefLink* is of no consequence; it is intended merely to convey the intention to anyone reading the source file (you!). What is important is the declaration of the XLink namespace within the tag, and the attributes and values that follow.

In Xlink every link has a *type*. In Figure 8.1 it has the value simple, which indicates an ordinary link. There are five other possibilities: *extended*, which is another main form of link, and *arc*, *locator*, *resource*, and *title*, which play supporting roles. *Extended links* allow labeled directed graphs to be described, in which nodes represent resources and directional links hold annotations. *Simple* links provide a shorthand for the HTML link tags such as <*a*> and <*img*> (or any HTML tags that use *href* and *src*). You will appreciate the value of the shorthand when we move on to the example in Figure 8.2.

The *href* attribute of the <*HrefLink*> tag in Figure 8.1 indicates the resource that the link addresses—the destination of the link. Here it is the home page of the UN, but the destination can combine a URI with an XPointer specification to select a subset of a document. The *actuate* attribute determines when the destination resource is accessed. Here it is set to *onRequest*, which means that access occurs whenever the link is clicked. Another commonly used value is *onLoad*, which means that the linked resource is accessed when the document containing the link is loaded. The *show* attribute determines where the destination resource will be displayed. Here it is set to *replace*, which means that the window used to display the UN agencies will be updated to show the UN's home page when the link is clicked. Alternatively you can set *show* to *open* the destination resource in a new window, or to *embed* it in the current page, at the current position.

If a document includes several links, the XLink namespace could be specified higher up the hierarchy—for example, at the root node. In many cases only the *href* attribute differs from one link to another, and values for the other attributes could be provided in a DTD. This would reduce the tag to

```
<HrefLink xlink:href="http://www.un.org/">
```

—just like HTML.

Figure 8.1 shows four XLink attributes. There are ten possibilities altogether. They are listed in the leftmost column of Table 8.1, which also indicates (with 3) those that can be used for various settings of the *type* attribute.

To illustrate the attributes and types, we extend the example to include a multiway link connecting the text *several agencies* to a list of Web sites, one for each agency. Here is what it will do. When the user

clicks on the words *several agencies*, a popup window will appear that displays an informative list of agency names. When one of these is clicked, a new window displaying the appropriate home page will be opened. This behavior can be achieved using extended links. If only simple links were available, each agency's link would have to be embedded in the document, needlessly consuming screen space.

Figure 8.2 shows what must be done. Not surprisingly it is far more complex than the previous example. Figure 8.2 also includes an internal link that uses an XPointer to make an extra entry in the multiway link that takes the user to the start of the agency information in the current document.

The link is created by the tag named <*MultiwayLink*>—the actual name is immaterial and is chosen purely for readability. The link's type is extended, and its scope ends with the tag </*MultiwayLink*> near the end of the example. Contained within it are elements of type *resource*, *locator*, and *arc*. There is only one resource here, <*SingleSource*>, and it represents the starting point of the multiway link. The content of this tag, "several agencies," is the anchor text for the link—what you click on in the document. The <*SingleSource*> element sets its *xlink:label* to *source*; we explain this shortly.

The <*SingleSource*> element is followed by a series of *locator* elements called <*Destination*>, one for each of the UN agencies (plus a further one that demonstrates the use of an XPointer). These are used to specify external resources—in our case the targets of the link—and they too make use of the *xlink:label* attribute. Following them is a series of *arc* elements called <*ExternalConnection*>, each of which have *from* and *to* attributes that reference the labels given in the resource and locator elements.

The structure defined by this multiway link is represented by the directed graph shown in Figure 8.3. Nodes represent the one *resource* element <*SingleSource*> and all the *locator* elements, of which several are called <*Destination*> and one is called <*Internal*>. These nodes correspond to a document or part of a document. The graph's edges are formed by matching the *from* and *to* attributes of *arc* elements with the *label* attributes used in the node definitions, and are annotated with the *title* attributes of the *arc* definitions. The title information that the *locator* elements provide is associated with the destination documents. Being attributes, these titles cannot contain markup. However, there is a second form of title (not shown in the example) that is specified using an element of type *title*, and this construct can include markup. It can accompany elements of type *arc*, *locator*, and *extended* and can be repeated several times within one element.

Within the *locator* and *arc* segments of the extended link are two tags that have not yet been discussed: <*Internal*> and <*InternalConnection*>. Again the names are immaterial, but are there to aid comprehension. These constructs demonstrate the use of XPointers. The first has an href attribute whose value is

```
#xpointer(/NGODoc/Agency[position()==1])
```

Unlike the *hrefs* that we have encountered so far, this is not a simple URI. It is an XPointer that specifies a hierarchical position within the document called *NGODoc*: namely, the first *Agency* node. XPointers are a superset of the XPaths that we met in Chapter 5 and contain extra features such as the ability to specify ranges that cross node boundaries.

This particular XPointer is used to select an internal portion of the document. However, it can also be paired up with a URI to link to part of an external resource, as in

www.un.org/index.html#xpointer(html/body/table)

Recall that in HTML a hyperlink can be directed at a particular position in a document by embedding an anchor name such as *<a name="here">* in the destination document and using *<a href="…#here">* in the source document to link to that point. The same effect can be achieved in XML by combining a URI with an XPointer that names an *ID* attribute in the destination document. However, an XPointer can be more expressive than this: it can indicate a particular internal position by specifying a location in the hierarchical structure that represents the target document. This allows a position to be specified in the destination document without actually having to edit explicit anchors into the destination document.

The *role* and *arcrole* attributes are not used in this example. Like *title*, they are semantic attributes that help communicate the meaning of the link. *Title* gives the description directly, but *role* and *arcrole* specify URIs that point to resources containing the description.

This example gives a flavor of what is possible. Far more complex graphs than that of Figure 8.3 can be built. The graph description can even be included in an external file—a type of database describing, for example, the linking structure for a network of pages.

Types

Document type definitions were originally introduced for use with the standard generalized markup language SGML. XML has been applied to a wider set of problems, and this has exposed limitations in the expressiveness of DTDs. For instance, they can include only limited information about data types, and certain structures are convoluted and do not scale well. XML Schema was designed to address these deficiencies.

In addition to describing what structure is allowed in an XML file, XML Schema provides a rich array of basic types, including year, date, and URI, as well as textual patterns and ways of subtyping and defining new types. Types can be applied to the data that appears between tag pairs, or to the fields of attributes. Everything is expressed using the basic XML syntax. Note that DTDs do not themselves adhere to XML syntax, but are expressed using an add-on notation. Strictly speaking they do not contain properly formed tags.

We introduced DTDs in Section 5.2 (Chapter 5) using the United Nations example (Figures 5.3 and 5.4). Figure 8.4 reworks the example of Figure 5.3 to give the same result using XML Schema. Like other members of the XML brotherhood, XML Schema is defined using namespaces to prevent any clash of tags with other elements of a document.

The file in Figure 8.4 has the same structure as other standards built on the basic XML foundation. The root node specifies the XML Schema namespace, enabling applications to interpret subsequent

nodes appropriately. Its children use *annotation*, *element*, and *complexType* elements. The first is a mechanism for embedding comments in the file in a more structured way than <!-- ... -->. The second defines an XML element: it performs the same function as <!*ELEMENT* ...> in a DTD. Structural information can be embedded inside the tag, as in the first occurrence of the *element* tag. Alternatively the definition can be deferred using the *type* attribute and the combined open/close tag syntax <…/> for empty elements, as in the second occurrence of the *element* tag. In the latter case the *complexType* element that provides the deferred definition need not follow the *element* node directly, although it does in our example.

The definition of the *NGODoc* element sets up a *sequence* comprising *Head* followed by *Body*. This construct forces elements to appear in the stated order. The attributes *minOccurs* and *maxOccurs* restrict the number of times each element may occur. In the *Head* element, both are set to 1. However, this is the default value and may be dropped—as it is in the definition of *Body*. Other numerical values can be used. For example, if *minOccurs* is 0 and *maxOccurs* is *unbounded*, any number of occurrences are allowed.

Within the scope of the *sequence* tag, two complementary tags may appear: *choice* and *all*. The last nesting group in Figure 8.4, which defines the *Agency* element, contains an example. In the lower half of the definition, *choice* is used to select an element from a list of potential candidates—in this case *Abbrev* and *Photo*. In the choice tag, *maxOccurs* is set to *unbounded*. This means that not only can *Abbrev* and *Photo* appear in any order, but any number of these element types can occur in any order. The *all* tag (not illustrated) means that all of the child elements must occur, but their order is immaterial. Constructs like these are difficult to define using DTDs.

To allow a mixture of parsed character data and tags within the *Body* element as the original DTD example did (Figure 5.4), *Body*'s definition sets the *mixed* attribute to *true*.

An element's attributes are defined using the *xsd:attribute* tag (equivalent to <!*ATTRIBUTE* ...> in a DTD). There are three examples in Figure 8.4. Defining attributes in XML Schema is decidedly self-referential because the construct uses attributes itself to define the name and type of the attribute being defined. However, it is fairly straightforward to deduce the meaning. For example, including

```
<xsd:attribute name="hq" type="xsd:string"/>
```

inside an *element* definition defines an attribute *hq* for that element. Here for the first time we encounter XML Schema's ability to specify type information. However, since this example matches an existing DTD, types are used in a simplistic way: they are all defined to be *string* to maintain compatibility with the character data (CDATA) used in the original DTD.

Attributes can be *optional* or *required*, and a *default* value can be supplied. These are handled within the construct using attributes of the same name.

A complication occurs when an element, defined to be a particular type, also has attributes. This occurs in the upper half of the *Agency* element definition (the last major nested group), when the *Name* element is defined. Its basic type is *string*, but it includes an *hq* attribute which is also defined

to be *string*. The syntax necessary to achieve this is convoluted and uses the new tag types *simpleContent* and *extension*; nevertheless it is not hard to follow. *Extension* permits the type of the element to be specified through the *base* attribute; then within the scope of the *extension* tag the attribute is defined as before. Finally, because XML Schema does not permit *extension* to be embedded directly in *complexType* tags, it is necessary to wrap *extension* up in a *simpleContent* tag.

The XML Schema specification in Figure 8.4 is far longer than the DTD equivalent in Figure 5.4. Of course the example is tutorial and could be abbreviated somewhat using such things as default values—but it would still be longer. The benefit of XML Schema is that it provides greater control over the structures and values that constitute a valid document. And documents like the specification in Figure 8.4 are clearly intended to be created and displayed with a structured editor rather than in raw text form. This transfers the emphasis from readability to "parseability": from ease of reading by a person to ease of manipulation by a computer. There are generic tools that allow all members of the XML family to be read, parsed, and edited.

XML Schema has extensive facilities for data typing. We have already encountered the type *string*, which was used in the first example to provide a counterpart to CDATA. This is a built-in type. There are over 40 others, reflecting the wide range of information handled by XML. The main categories are *Boolean*, *numeric*, *time*, *string*, and *binary*. *Binary* data is encoded into printable plain text to meet the XML character-set requirement. The *numeric* category includes signed and unsigned numbers, integer and floating point numbers, finite and infinite precision. Within the *time* category, dates, months, and years can all be represented individually. There is also a built-in type for URIs (subsuming URLs and URNs), and for XML notation such as entities (e.g., *"*) and tokens. New types can be constructed that expand or restrict the set of permissible values.

To demonstrate the data-typing abilities, we extend the XML Schema example to include the year in which each UN agency was founded, and we constrain more tightly the value types that attributes can assume. The result is shown in Figure 8.5. The *simpleType* construct is used to define new types. The first example is *UNYear*, in the lower half of Figure 8.5. It is based on the built-in time structure *gYear* and is restricted to values greater than the year 1850. (The United Nations was founded in 1945, but some of the specialized agencies it contains—such as the Universal Postal Union and the International Labor Organization—predate it by a considerable margin.)

The second *simpleType* in Figure 8.5 is *CityType*, which is used as the type for the *hqcity* attribute of the *Agency* element (near the top of the figure). This illustrates the use of patterns, which are couched in the form of regular expressions. The type definition restricts the basic *string* type to contain an uppercase letter (\p{*Lu*}) followed by a sequence of any letters (\p{*L*}*). Regular expressions follow standard practice, with the usual special characters: ? (meaning an optional character), + (meaning a sequence of one or more characters), * (a sequence of zero or more characters), . (any character), [A-Z] (the uppercase letters), and so on. The \p{...} in the example is an extension of the notation that expresses sets of Unicode symbols: letters, numbers, punctuation, currency symbols, and so on. We could have specified a capital letter as [A-Z], but \p{*Lu*} is more general. The *minLength* tag in the

type definition illustrates one way to specify bounds on string length. This could equally have been accomplished within the regular expression itself.

The last *simpleType* definition, *UNCountryType*, is an enumerated type. After specifying the base type to be *string*, a series of *enumeration tag*s are used to restrict the permissible countries to a given set.

Another main form of type declaration is *complexType*, which is used to define compound types that include other elements. We have already encountered this construct in the previous XML Schema example, although we did not discuss it explicitly. It was used in Figure 8.4 to make *NGODoc* contain a *Head* followed by a *Body*, to make *Head* contain a *Title*, to make *Body* contain an unbounded sequence of *Agencies*, and to make each *Agency* a *Name* followed by any number of *Abbrevs* and *Photos*.

Putting all this type information together, we see that Figure 8.5 adds these restrictions to the UN schema:

The *hq* attribute has been replaced with *hqcity* and *hqcountry*. The former must (unrealistically) contain just one word, starting with a capital letter and having at least one other letter. The latter must be one of a list of designated countries.

There is now a <*Founded*> element, used to express years after 1850.

The *src* attribute for *Photo* must now be a legal URI.

XML Query

XML Query (abbreviated to XQuery) is also designed for information retrieval without being tied to any specific implementation, but it takes a strikingly different approach, founded upon the extensible markup perspective. With the rider that all documents must be in XML, queries can be used to construct new documents, and no distinction is made between retrieval of documents and parts of documents.

XQuery is considerably more complex than CCL, involving its own functional programming language, and builds on XPath and XML Schema. The underlying data model (also shared by XPath) represents arbitrary sequences of documents, or parts of documents, as forests—lists of tree structures. The language consists of

- path expressions
- element constructors
- For-Let-Where-Return expressions
- quantified expressions
- operators and functions
- conditional expressions
- data types

The first two items are particular to XML; the third borrows heavily from the database query language SQL; and the last four are standard components of functional languages.

We demonstrate these parts through a series of examples based upon the resource outlined in Figure 8.14. Figure 8.14a represents the overall structure. The root node is <*Library*>, and its children represent publication categories (akin to the database names used in the CCL example). Within each category are the publications themselves, expressed as an XInclude statement that uses the *href* attribute to provide a URL for a particular publication. XInclude is a supplementary XML standard that acts just like the *include* statements of programming languages. It is self-explanatory. Triggered by the inclusion of its namespace, an XInclude-aware processing application interprets the *include* tags as textual substitution of the named resources.

Figure 8.14a embeds individual publications in an overall document type hierarchy. Figure 8.14b shows the first document, which reuses the Book Database schema from Section 8.2; the remaining publications follow the same pattern. In the example, a publication starts with a *Document* node, which includes *Metadata* and *Text* sections. *Metadata* includes *Title*, *Author*, and *Description* items, and the text is further marked up with *Poem*, *Chapter*, and other suitable elements. We could equally well have used the Open eBook standard, but this would unnecessarily complicate the XQuery illustrations.

Figure 8.15 shows XQuery statements that perform the same search as do the two examples of Figure 8.12: find documents with author Sam Hunt, and find documents authored by Sam Hunt, or whose abstract contains the phrase "Sam Hunt." The language's functional nature is very much in evidence, with keywords appearing in block capitals. It also has a scripting feel, variables being prefixed with $ and comments beginning with #. Both queries have the same structure: FOR . . . WHERE . . . RETURN, a subset of the For-Let-Where-Return or FLWR ("flower") expression. The first query uses three XPath expressions to select subtrees in the XML library structure. They are easily recognizable because they include slashes (/): the first is in the FOR line, the next in WHERE, and the last in RETURN. The result of executing the query is this: for every *Document* node in the *Library* (represented by *library.xml*), locate every grandchild *Author* (descending through *Metadata*) and check whether it contains the phrase "Sam Hunt"; if so, return its *Title* element.

SOME . . . IN . . . SATISFIES . . . is an example of a qualified expression; another form of qualified expression starts with EVERY. Our query finishes by calling the built-in function *contains*(). The language also allows new functions to be defined. Types are the same as the ones that XML Schema uses, augmented with operations for type casting (although none are present in the example).

Figure 8.16 shows two XQuery statements that show how elements can be constructed. The first example builds an XML document with the root *LibrarySummary* that contains a *DocumentSummary* element which summarizes each document by recording its title and the number of authors. To do this, programming constructs are embedded into XML tags using braces {…}. These in turn can specify XML tags, and so on. A LET statement is used on the second line to gather all the *Author* nodes,

which is later passed as an argument to the built-in function *count*(). This returns the number of items (authors) represented by the variable.

The second example builds an XML document that contains the poems that Sam Hunt published between 1980 and 1985. It uses a full FLWR statement, retrieving all documents *$d* with publication date *$p* that match the conditions expressed in the WHERE clause. The information that is extracted is packaged in *Document*, *Metadata*, and *Text* tags. Note the alternative method used to select the author. Instead of the protracted qualified clause *WHERE SOME . . . contains*(…) used in the Figure 8.15 example, the same effect is accomplished by the more concise XPath expression *$d/Metadata/[Author="Sam Hunt"]*. This is because we seek an exact match with the content of *Author*, not a substring match. FLWR expressions can be nested inside other expressions, including other FLWR expressions, although none of our examples requires this.

XQuery commands are considerably more lengthy than their CCL counterparts and require a more detailed knowledge of programming. A specialized environment might be set up for a particular community, so that members can achieve the results they need using simplified function calls defined in XQuery itself. For this to work, the various XML databases involved need to agree on the structures they use. That, of course, is what RDF, XML Schema, and DTDs are for.