# XML Linking Language(XLink

- Linking and addressing
- Simple links
- Extended links

Chapter

# 34

Hypertext links are the backbone of the World Wide Web. Documents were shuffled around the Internet long before today's Web existed, but it was the ease of moving from page to page with hypertext links that made the Web into the mass market phenomenon it is today.

However, despite their centrality, Web links have many weaknesses. The linking system that we use today is essentially unchanged from the earliest version of the Web. Unfortunately, market inertia has prevented anything more powerful from coming along ... until now.

The second specification in the XML family is XLink. It allows links that go far beyond those provided by HTML. XLinks can have multiple end points, be traversed in multiple directions, and be stored in databases and groups independently of the documents they refer to.

XLink is exciting, but we cannot take full advantage of it yet. It is still being refined by the World Wide Web consortium.

***Note*** *The current version of XLink, and its companion specification, XPointer, are working drafts and will change before they are completed. The basic concepts are well understood and will not change, but the specifics may change between now and then. We will cover only the parts we consider most stable.*

## 34.1 | Basic concepts

The most important (and sometimes subtle) distinction in any discussion of hyperlinking is that of *linking* versus *addressing*. Linking is simply declaring a relationship between two things. If we say "George Washington and Booker T. Washington share a last name" then we have linked those two people in some way.

Addressing, on the other hand, is about describing how to find the two things being linked. There are many kinds of addresses, such as mail addresses, email addresses and URLs. When you create a link in XML, you declare a relationship between two objects referred to by their addresses (URIs). We refer to these objects as *resources*. We will discuss the addresses (URIs) more in 34.3, "Addressing", on page 511.

If you have created Web pages before, you are probably familiar with HTML's simple A element. Whether or not you are familiar with HTML, that link is a good starting point for understanding hyperlinking in general.

The A stands for anchor. Anchor is essentially the HTML term for resource. An HTML link has two ends, termed the *source* and *destination*. When you click on the source end, (designated with an A element and HREF attribute), the Web browser transports you to the other end. Example 34-1 shows how this works.

**Example 34-1. An HTML (not XML) link**

```
<A HREF="http://www.mysite.com">Go to my site!</A>
```

In this case, the A element itself describes a link, and its HREF attribute points to one of the resources (the destination). As we know, links connect resources, so there must be at least one more resource involved. The other resource, the source, is actually the text that forms the content of the A element. As we will see, XML *simple* links also use the content of the link as one of the resources.

The destination of the link in Example 34-1 does not necessarily know that it is a destination. If you want to link to the Disney home page, you do not need to inform Disney. If a particular document has fifty A elements with HREF attributes, then you know that it has fifty links out. But the Web provides no way to know how many links into it there are.

In the more general *extended* link case, we will link two things such that *neither* end will "know" that it is being linked. The link exists in some third location (or fourth, or fifth, for multi-ended links). This is intuitive if you go back to the definition of linking as defining a relationship. In a real-world sense, I can "link" Jenny Jones and Oprah Winfrey just by speaking of them in the same sentence. Unless they are interested in careers as webmasters, they will probably never know. XLink provides a standardized way to express this in markup.

We might even want to link something that is not explicitly labeled. For instance, we might want to link the third paragraph of the fourth sub-point of the second section of a legal document to the transcript of a relevant court case.

This is analogous to the real world situation where you can either send something to a labeled location ("Please take this to the White House.") or you can give directions to the destination. In hyperlinking terms, we would consider either one of these to be an "address." Obviously there must be some way of locating a resource from a link, but it could be either an address, a label or a combination of the two: "The building is 5 blocks down the street from the White House."

### 34.1.1 *Simple links*

Although XLink allows more flexible links than does HTML, it also offers simple links that are not much more complicated than HTML's links are. This sort of link is referred to as a simple link. A simple link has two ends, a source and a destination, just like an HTML link. One end has content

that represents a resource (usually the source) and it refers to the other end through a URI.

**Example 34-2. XML Simple Link**

```
. . . for more information, consult
<citation xml:link="simple" href="http://www.uw.ca/paper.xml">
     Biemans(1997)
</mylink>
```

The biggest difference between this link and the HTML link is that this element is not designated a link by its element type name. It is not called A or any special element type name specified in the XML specification. You can call your linking elements whatever you want to. This is an important feature, because it allows you to have many different types of linking elements in a document, perhaps with different declarations, attributes and behaviors. Just as XML allows you to use any element-type name for paragraphs or figures, it allows you flexibility in your linking element-type names.

The link is actually designated an XML link by its `xml:link` attribute. The `xml:` prefix indicates that this attribute's allowed values and semantics are specified by the World Wide Web consortium. This attribute describes what kind of link it is. In this case, it is a simple link.

## 34.1.2 *Link roles*

In HTML, link resources are either sources or destinations. The element that describes the link is always the source. The resource referred to is always the destination. In XML, this rigid distinction is not hard-wired. An application can make either or both links into sources or destinations. Consider, for instance, if a Web browser made it possible to create notes about someone else's Web site and "stick" them on to it like Post-It notes. These annotations might be represented as XLink *extended* links.

In this case, we actually want the application to make some form of clickable "hotspot" at the other end, on the newspaper's Web page. Of course we don't want them to have control of the actual linking element, or else they might just choose not to show our link. So we want the link to exist in one

**Example 34-3. XLink annotations.**

```
<annotation xml:link="extended"
        href="http://www.mynewspaper.com">
As usual, your editorial is filled with the kind of claptrap and
willywag that gives me the heebie-jeebies!
</annotation>
```

spot and create a "hotspot" at another. This is the opposite of traditional HTML links.

In order to reverse linking roles, we must somehow tell the application that we want it to do so. One way would be to use an element-type name that the application is hard-coded to understand as having that semantic. For instance an "annotation server" might only deal with `annotation` elements, or perhaps a few different variants, and would thus know exactly how to handle it.

Another way would be to use some form of stylesheet. But you would still need to have something special in the document that would differentiate annotations from other links (perhaps the annotation element-type name). The stylesheet would provide an extra level of translation to allow your private element-type names to be interpreted as annotations by software.

Yet another way to solve this problem would be to provide an attribute that describes the role of the link in the document and hypertext system. Any of these are valid implementation approaches, and the XLink specification provides a special `role` attribute to handle the last case. Example 34-4 is an example of that attribute in action.

**Example 34-4. Role attribute**

```
<hlink role="annotation"
        xml:link="extended"
href="http://www.mynewspaper.com">
As usual, your editorial is filled with the kind of claptrap and
willywag that gives me the heebie-jeebies!
</hlink>
```

In this case, the role designation has moved from the element-type name (now `hlink` instead of `annotation`) to the role attribute. Which is more appropriate will depend on your DTD, your software and your taste. XLink could perhaps dictate one style or the other, but real world usage is not that simple. For instance you might need to use an industry standard DTD and

thus have no control over element-type names. In another application, you might need to constrain the occurrence of certain kinds of linking elements, and thus need to use element-type names and content models.

### 34.1.3 *Is this for real?*

You might well ask whether all of this annotation stuff is likely to happen. After all, there are all sorts of social, technical and financial difficulties related to being able to annotate someone else's Web page. Imagine annotation spam: "Tired of reading this boring technical Web page? Click here for HOT PICS!!!" It turns out that early versions of the pre-Netscape Mosaic browser allowed remote annotations (using a proprietary linking scheme), and you could share your annotations with friends or co-workers, but not with everybody on the Web. There are various other experimental services and products that provide the same ability for the modern-day Web. However, each uses a distinct link description notation so that they cannot share.

We may or may not get to the point where everybody can publish annotations to the whole world, but we already have the technology to create annotations that can be shared by other people we know. Unfortunately, this technology has never been widely deployed. Perhaps XLink will solve the link incompatibility problem and allow Web pages to become readable, writeable, and even more linkable.

So what can you do without a world-wide link database? Well let's say that your organization was considering buying a very expensive software product. You and your co-workers might agree to submit your opinions of the product specifications published on the vendor's Website. You could make a bunch of external links from the vendor's text to your comments on it and submit that to your organizational link database. When your co-workers go to see the page, their browsers can fetch your links and actually display them as if they were part of the original document. When your co-workers click on them, the browser will take them to your annotations.

In fact, with a reasonably big link database, you could annotate any Web page you came upon in this manner. When others from your organization came upon the page, they would see your annotations. In one sense, you are editing the entire Web! Of course, the bigger your organization is, the more points of view you can see on each page. On the other hand, sometimes you might not want to share all of your comments with the entire company, so you might have a smaller departmental database which is separate, and only

shared by your direct co-workers. And of course at the opposite end of the spectrum, there might be a database for everyone on the Web (if we can make link database software that scales appropriately and find someone to run it).

External links can be useful even without a link database. Without such a database, there is no easy way to distribute your links to other people, so you must communicate the links' existence in some other way. For instance, you could include a critique of a Web page as an attachment to an email. You could also build a document full of links that annotated one of your own Web pages with links to glossary and bibliographic information. We might term each collection a link sheet. Depending on which link sheet the reader used, he would get either the glossary links or the bibliographic links or both sets of links overlapping.

If it makes sense to "project" a link from your home computer onto an existing Website, then surely it makes just as much sense to link two existing Websites. For instance, we could make a link that is targeted towards members of the SGML newsgroup that links the World Wide Web consortium's XML Web page to a related page we know about on the Web. This link would still have two ends, but both could be sources and destinations at the same time. If so, we would term that link bidirectional, because you could *traverse* it from either end. Because the link would exist on your Web site, but link two other pages, we would call it *out-of-line*. And if it makes sense to link two pages, then why not three, or four, or five? Extended links allow this.

### 34.1.4 *Link behaviors*

XML authors usually go out of their way to avoid putting information about formatting and other types of document behavior into XML documents. We've already been through all of the benefits of keeping your information "pure". As we have said, if you just mark up your documents according to their abstractions, you can apply formatting and other behavior through stylesheets.

On the other hand, there are a few link behaviors that are so common – almost universal – that the XML working group decided that it would be easier to provide some attributes to specify them directly. This takes a layer of abstraction out and thus makes hyperlinking a little bit easier. The con-

cepts of hyperlinking are already abstract, so anything that makes life a little bit easier will help XLink to become popular.

The most interesting type of link behavior is *traversal.* When you click on a hyperlink, you are *traversing* it. If a link is intended to embed information from one resource in another, then the process of actually accomplishing the embedding is a traversal.

The behavioral descriptions are still "abstract enough" to allow a variety of specific behaviors, depending on the situation. The XLink spec says:

**Spec. Reference 34-1. Behavior**

The mechanism that XLink provides allows link authors to signal certain intentions as to the timing and effects of traversal. Such intentions can be expressed along two axes, labeled *show* and *actuate.* These are used to express policies rather than mechanisms; any link-processing application software is free to devise its own mechanisms, best suited to the user environment and processing mode, to implement the requested policies.

What this means is that different types of software applications are allowed to interpret these suggestions differently. For instance, you might not think of a printer as a machine that would care about hyperlinks, but it might be useful to have a printer that could directly print Web pages and their annotations, or that could resolve graphics embedded through XLink.

### 34.1.4.1 Show

As the name implies, the `show` attribute describes how the results of a link traversal should be shown. When you click on a Web link, that is a link traversal – one initiated by your click. On the other hand, if you have ever been to a site where a Web page comes up and says: "You will be forwarded to another page in just a few seconds", then that is a link traversal that is automatic. Typically on the Web, when a link is traversed (manually or automatically) it replaces the previous document in the Web browser window. XLink allows an author to request this behavior with the `replace` value of the `show` attribute:

For example:

Occasionally you will also come across a link that actually opens a new window, so that after traversal there is a window for the new page in front

**Spec. Reference 34-2. Replace**

"replace" indicates that upon traversal of the link, the designated resource should, for the purposes of display or processing, replace the resource where the traversal started.

**Example 34-5. A `replace` link**

```
<A xml:link="simple" show="replace" href="http://www.gop.org/">
Click here to visit the GOP</A>
```

of the window for the old page. XLink allows this through the `new` value of the `show` attribute.

**Spec. Reference 34-3. New**

"new" indicates that upon traversal of the link, the designated resource should be displayed or processed in a new context, not affecting that of the resource where the traversal started.

**Example 34-6. A `new` link**

```
<A xml:link="simple" show="new" href="http://www.democrats.org/">
Click here to launch a new window and visit the the Dems.</A>
```

As we discussed before, a link can in fact represent *any* relationship. Consider the relationship between a document and an embedded graphic or even text fragment. This can be represented as a link also! Of course there are ways to embed graphics and text using only XML entities, but XLink provides another way of doing the same thing, which can be used in situations where the entity mechanism is not expressive enough by itself.

In this case, you can use the `embed` value:

**Spec. Reference 34-4. Embed**

"embed" indicates that upon traversal of the link, the designated resource should be embedded, for the purposes of display or processing, in the body of the resource and at the location where the traversal started.

**Example 34-7. An `embed` link**

```
<A xml:link="simple" show="embed"
    href="http://www.democrats.org/logo.gif">
```

### 34.1.5 *Actuate*

The `actuate` attribute allows the author to describe when the link traversal should occur. For instance it could be user-triggered, such as by a mouse click or a voice command. Or else it could be automatic, such as the automatic embedding of a graphic, or an automatic forward to another Web page (e.g. "This page has moved. You will be directed to the new page momentarily.")

The `user` value indicates that the traversal should be user-triggered. When it is combined with a a `show` attribute of `replace`, it is a typical, click-here-to-go-there link, at least in a graphical browser. On a text-based browser, it might be a type-this-number-to-go-there link. On a spoken-word browser it might be a say-this-number-to-go-there link.

When it is combined with a value of `new` it opens a new "context" (usually a browser window) at user command and leaves the old one open. When it is combined with a value of `embed`, it would make a link that expands into the embedded object. For instance a footnote or graphic might expand in-place when you click on them.

The `auto` value of the `actuate` attribute is used to specify that traversal should be automatic. For instance, most `show="embed"` links would specify automatic traversal. If you combine `show="new"` with `actuate="auto"`, then you can create a Web page that immediately opens another Web page. Perhaps with a stylesheet or other attribute, you could make them be side by side. The final combination is `show="replace"` with `actuate="auto"`. You would use this to set up a "forwarding" link, such as the one we have described, and thus forward users from one page to another.

### 34.1.6 *Behavior*

XLink also provides a `behavior` attribute for specifying more precise behaviors than the policies described above. You could fill this attribute with commands provided by a browser vendor, or with "hooks" to invoke rules in your stylesheet.

You should be careful with an attribute that is as vaguely defined as this one. Wait until some conventions for its use arise before you fill your documents with markup that could be misinterpreted by confused software.

**Spec. Reference 34-5. Behavior attribute**

A link author can also optionally use an attribute called behavior to communicate detailed instructions for traversal behavior. The contents, format, and meaning of this attribute are unconstrained.

## 34.2 | Extended links

In this section, we will discuss more features of the *extended links*. One that we have already discussed is the ability to specify them out-of-line. Extended links also allow for more link ends, more advanced link roles, and other good stuff. We will also be able to re-describe the simple links that we have already seen in the terminology of the more general extended link system.

### 34.2.1 *Locator elements*

The first extension we will undertake is links with more than two link ends. Consider, for example, that you are redirecting users to several different interpretations of a text. For instance if there were two competitive schools of thought on a topic, each hotspot in the document might allow traversal to a different interpretation of the topic. Now you have three link ends, one for the source and one for each of the interpretations of it. Just as in real life, XLink allows you to make logical links among two or more concepts.

The first big difference between simple links and extended links is that we need to figure out how to specify the address of more than one destination link. We do this by putting *locator* sub-elements into the extended link element. Here is an example:

**Example 34-8. Multi-ended link**

```
<commentary xml:link="extended">
    <locator href="roberts.xml" role="Roberts"/>
    <locator href="beam.xml" role="Beam"/>
    <locator href="goodwin.xml" role="Goodwin"/>
<P>My fellow Americans, this speech will go down in history...
</commentary>
```

In this case, the three locators each address a resource. A sufficiently sophisiticated browser displaying this document might represent each with an icon or supply a popup menu that allows access to each of the resources. It could even open a small window for each interpretation when the hotspot is selected. This could be controlled by a stylesheet or a behavior attribute. As you can see, each locator can have a different role, but they could also share roles. The role just specifies a semantic for processing the resource when processing the link, not some sort of unique identifier.

Locators can also have some other associated attributes. They can have titles, specified through a `title` attribute. These provide information for human consumption. The browser does not act on them. It merely passes them on to the human in some way, such as a popup menu, or text on the status bar. Locators can also have `show`, `actuate` and `behavor` attributes with the same semantics as for a simple link. Locators seem very similar to simple links because a simple link is a combination of a link and a locator. In fact, this is how they are defined in the XML spec:

**Spec. Reference 34-6. Simple links**

Simple links can be used for purposes that approximate the functionality of a basic HTML A link, but they can also support a limited amount of additional functionality. Simple links have only one locator and thus, for convenience, combine the functions of a linking element and a locator into a single element. As a result of this combination, the simple linking element offers both a locator attribute and all the link and resource semantic attributes.

It is both useful and convenient that simple links combine these two things, but it means that we must be careful to keep the ideas separate in our heads. The link describes a relationship. The locators say what resources are being related. A simple link uses its content as one resource and the target of its `href` as the other.

### 34.2.2 *Link groups*

It is often useful to be able to process a group of hyperlinked documents all together. For instance, if one document contains some text and another

contains a rebuttal of the text, the browser might want to show them "side by side". It could also allow link traversals in one window to trigger the correct portion of the rebuttal in the other.

Such processing can only work if the browser knows about both documents at the same time. Extended link groups allow you to tell the browser about all of the nodes that should be processed together.

An *extended link group* element is a special kind of extended link. It describes a list of other documents that should be seen to be in this *link group*. Here is an example of such a link:

**Example 34-9. Extended link group**

```
<related-documents xml:link="group">
<doc xml:link="document" href="annotation.html">
<doc xml:link="document" href="rebuttal.html">
<doc xml:link="document" href="support.html">
</related-documents>
```

In one sense, a link group is a small database of hyperlinks. A browser, editor or other application could look in the link group to see which elements are hyperlink resources and what their behaviors and roles are.

## 34.3 | Addressing

Now that you know how to make all kinds of neat-o links, you might wonder if XML also features neat-o addressing. Good guess! Of course, XML allows the usual kinds of URLs that you use to navigate the Web. But now that those have found their way onto everything from milk cartons to television advertisments, it is time for something new: *XPointers*. XPointers allow sophisticated addressing into the contents of XML documents. That means that you can make a link to an element, or even a span of elements, based on things like position, element type and ID.

Like XLink and XSL, XPointers are still under development. But the concepts are not likely to change much. They are well established in existing projects like the *Text Encoding Initiative* and the *HyTime* International Standard.

## 34.4 | Uniform Resource Identifier (URI)

The basic form of address for XLink is a *URI*, which stands for Uniform Resource Identifier. Today's most important form of URI is an extended form of the URL or *Uniform Resource Locator.*[1]

URLs are uniform, in that they have the same basic syntax no matter what specific type of resource (e.g. Web page, newsgroup) is being addressed or what mechanism is described to fetch it. They describe the locations of Web resources much as a physical address describes a person's location. URLs are hierarchical, just as most physical addresses are. A land mail address is resolved by sending a letter to a particular country, and from there to a local processing station, and from there to an individual. URLs are similar.

The first part of a URL is the *protocol.* It describes the mechanism that the Web browser or other client should use to get the resource. Think of it as the difference between Federal Express, UPS, and the other courier services. The most common such protocol is `http` which is essentially the "official" protocol of the World Wide Web. The `ftp` file transfer protocol is also widely used, chiefly for large downloads such as new browser versions.

After the protocol, there is a *hostname* and then a *datapath.* The datapath is broken into chunks separated by slash ("/") characters, as you have no doubt seen in hundreds of URLs. Technically, a URL ends at that point.

In a URI, the URL can be followed by an optional *query* and then an optional *fragment identifier.* For instance you may have seen links into HTML documents that look like this:

```
http://www.megabank.com/banking#about
```

"#about" is a fragment identifier. It refers to a particular HTML element. XPointers are a similar concept for XML documents, but they are much more flexible. Essentially, XPointers are an extension to URLs to allow you to point not just *to* a document, but into the content of one.

For instance, on today's Web, if you wanted to quote a particular paragraph out of another document, you would go to that document and cut and paste the text into yours. If, in the future, the text on the Web changes,

1. When URIs are finalized by the *Internet Engineering Task Force* (IETF RFC 1738 and IETF RFC 1808), they will also allow *Universal Resource Names,* which aren't location-dependent and perhaps will reduce the number of broken links.

yours does not. If that is what you want, that is fine. But XPointers allow you to construct a "living document" that quotes and refers to the very latest version of the paragraph. You can understand how important this ability is for the types of annotations we have discussed. Without it, you could only annotate complete documents.

## 34.5 | Referring to IDs

The simplest form of XPointer allows you to refer to a particular element named with an ID. This is also the most robust form of XPointer, because it does not at all depend on the location of the referenced text within its document. Consider this XML document:

```
<?xml version="1.0"?>
<!DOCTYPE HEATWAVE SYSTEM "heatwave.dtd">
<HEATWAVE>
<WAVE ID="summer.92">
    <DURATION>July 22 to August 2</DURATION>
    <TEMPERATURE>101 Degrees</TEMPERATURE>
</WAVE>
<WAVE ID="summer.96">
    <DURATION>June 15 to July 18</DURATION>
    <TEMPERATURE>103 Degrees</TEMPERATURE>
</WAVE>
</HEATWAVE>
```

If this document resides at `http://www.hotdays.com/heatwave.xml`, then we could refer to the second `HEATWAVE` with this URI:

```
http://www.hotdays.com/heatwave.xml#id(summer.92)
```

The XPointer is the last little bit of the URI, after the pound-sign ("#"). An important thing to note is that this XPointer does not *do* anything. It refers to something. Whether the object is included, hyperlinked, or downloaded is competely a function of the context of reference.

For instance, you could use the XPointer in an XLink to create a hyperlink to something, or in a browser to download a particular object. It is also up to the software to decide whether the referred to element is returned alone, or in the context of its document. For instance, if you use an XPointer in a browser window, it would probably present the whole document and highlight the referenced element. But if you use it in an XLink to include a paragraph, it would probably take that paragraph out of its context and present it alone in the new context.

## 34.6 | Location terms

In the URI:

```
http://www.hotdays.com/heatwave.xml#id(summer.92)
```

The string `id(summer.92)` is called a *location term.* Another simple XPointer location term is the *root* location term. You use this to refer to the root element of a referenced document. For example:

```
http://www.hotdays.com/heatwave.xml#root()
```

That might seem strange, because you can implicitly refer to the root of a document just by leaving off the XPointer. But with more advanced location terms, we will actually be able to use the root (or an ID) as the starting point for a location ladder. That means that we could, for example, ask for the URI's second sub-element, that element's third sub-element of type `p` and so forth. These types of XPointers are more fragile because document reorganizations can break them. On the other hand, they allow you to refer to things that have not been identified with an ID. This is important if you are referring to text you cannot change, such as a document elsewhere on the Web, or on a read-only medium.

For example, to refer to the root element's third sub-element of type `WAVE`, you would do this:

```
http://.../heatwave.xml#root().child(3, WAVE)
```

Elements of other types would be ignored.

The `child` location term is called a *relative* location term because it depends upon a starting location identified by root() or id() which are *absolute* location terms. The `child` location term is the most common type of relative location term, but there are others. When you string them together, they allow you to "navigate" around a document from absolute or relative points. Here is the complete list:

The word "node", used in the spec, is more general than "element", because it includes constructs like comments, processing instructions and character strings.

So for instance, to refer to the element of type `H1` that follows the parent of the node with the ID `graceland`, you would create the following location ladder:

```
http://.../something.xml#id(graceland).parent().following(1, H1)
```

You can read the ladder like this: "Go to the element named graceland. Go to its parent. Find the H1 following it." Each statement represents a step down the metaphoric ladder. By combining them, you can address any element in an XML document.

**Spec. Reference 34-7. Relative location terms**

**child**
Identifies direct child nodes of the location source.

**descendant**
Identifies nodes appearing anywhere within the content of the location source.

**ancestor**
Identifies element nodes containing the location source.

**preceding**
Identifies nodes that appear before (preceding) the location source.

**following**
Identifies nodes that appear after (following) the location source.

**psibling**
Identifies sibling nodes (sharing their parent with the location source) that appear before (preceding) the location source.

**fsibling**
Identifies sibling nodes (sharing their parent with the location source) that appear after (following) the location source.

XPointers have more advanced and esoteric features, but they are still under development and may change in the future.

## 34.7 | Conclusion

XLink and XPointer have the power to change the Web, and our lives, in unforseeable ways. For more of the vision, see Chapter 13, "Extended linking", on page 176. For the current version of the specs, see the *XML SPECtacular* on the CD-ROM.

# Extensible Style Language (XSL)

- Stylesheets for XML
- Style rules
- Actions
- Flow objects

# Chapter 35

The *Extensible Style Language* (*XSL*) is a specification being developed within the World Wide Web consortium for applying formatting to XML documents in a standard way. Under he covers, XSL is based on *DSSSL*, a more powerful International tandard from the ISO. However, XSL stylesheets do not look much ke DSSSL stylesheets at all. The basic concepts of XSL are similar to SSSL's, but they have been simplified and "prettified" for Web use. Here are some of the design principles that are being used to create SL:

- XSL should be straightforwardly useable over the Internet.
- XSL stylesheets should be human-legible and reasonably clear.
- XSL stylesheets shall be easy to create.

As you can see, usability is an important concern in its design!

As our verb tense implies, XSL is still under development. It is likely to hange quite a bit. The latest set of design goals includes features for anima- on, interactivity and other very advanced features. XSL will be the topic of hany books all by itself!

The important thing is that the central concepts will be the same as those of DSSSL and the current XSL proposal. These concepts are the focus of this chapter.

***Note*** *XSL is changing quickly. This chapter outlines the important ideas that are not likely to change.*

The most important thing to get out of this chapter is a feeling for how XML documents are actually processed. We have worked with them purely as abstractions, but now we are going to put them to work. XSL's mechanism for doing this is very similar to that of most XML processing tools.

## 35.1 | XSL overview

As we discussed earlier, XSL is used to apply style to XML documents. These will usually be marked up entirely according to their abstract structure without (in theory) markup specifically tailored for style application or any other particular kind of processing. Thus XSL is the "missing link" between the data that has been encoded for computer processing and the formatted rendition required for comfortable reading. If you are trying to build a Web page, style languages are very important. Even if you are not, they provide a good example of how processing of XML documents proceeds.

### 35.1.1 *XSL stylesheets*

Most XSL code looks more or less like "ordinary" XML. Simple XSL stylesheets are merely a specialized form of XML markup designed for formatting other XML documents. XSL is not defined in terms of a formal DTD, but you can still think of it as a document type. The XSL language defines element types and attributes, constrains them to occurring in particular places, and gives them semantics.

Let's start simply and consider a stylesheet that would say that "Paragraphs should use a 12pt font" and "Titles should be 20 point and bolded."

These types of simple declarations are sufficient for many easy tasks. We say that XSL is a "declarative" language because declarations are so important. Declarative languages are easier to learn and use than are programming languages. They do not require complicated logical operations for simple tasks. Here is an XSL stylesheet to accomplish these tasks:

**Example 35-1. XSL Example**

```
<xsl>
  <rule>
    <target-element type="p"/>
    <paragraph font-size="12pt">
      <children/>
    </paragraph>
  </rule>

  <rule>
    <target-element type="title"/>
    <sequence font-size="20pt" font-weight="bold">
      <children/>
    </sequence>
  </rule>
</xsl>
```

Each `rule` element contains a `target-element` and an action. They say that whenever the XSL processor encounters a `p` element in an XML document, it should create a paragraph in the browser or word processor and give it a font-size of 12 points. Similarly, it should look for `title` elements and make them bold and 20pt. These are simple declarative rules. You do not have to think about the order in which things will be processed, where they are stored or other housekeeping tasks that programming languages usually require you to look after.

## 35.2 | Referencing XSL stylesheets

There is currently a proposal to allow XML documents to refer to their stylesheets. It has no official standing but it is short, simple and does the job it claims to, so it will probably become a defacto standard. Here is the relevant text of that proposal:

**Spec. Reference 35-1.** `xml:stylesheet` **processing instruction**

The xml:stylesheet processing instruction is allowed anywhere in the prolog of an XML document. The processing instruction can have pseudo-attributes href (required), type (required), title (optional), media (optional), charset (optional).

These are called "pseudo-attributes" instead of attributes because, although they use attribute syntax, they do not describe properties of an element. The only real syntactic difference between pseudo-attributes and attributes is that you must use pseudo-attributes in the order they are described. You can use attributes in any order.

The most important pseudo-attributes are `href`, which supplies a URI for the stylesheet, and `type`, which says that the stylesheet is in XSL and not DSSSL, CSS, or some other stylesheet language. You can also supply a `title` that the browser might use when offering a list of stylesheet choices. The `media` option allows you to specify what medium the stylesheet is for. You could, for example, have different stylesheets for print (with footnotes and page breaks), online (with clickable links), television (large text and easy scroll controls) and telephone (read aloud with inflection representing emphasis). XSL is not powerful enough yet to handle all of these media equally well, but it will be one day.

Here is an example stylesheet processing instruction (PI):

**Example 35-2. Stylesheet PI**

```
<?xml:stylesheet href="http://www.sgmlsource.com/memo.xsl"
                 type="text/xsl"?>
```

You can also provide multiple PIs to allow for choice by media, title or stylesheet language:

## 35.3 | Rules, patterns and actions

Every style sheet language consists of a series of statements that convert structural elements (from the source document) into formatting objects. Even the "Style" menu in Word for Windows consists of such a mapping. It

**Example 35-3. Alternative stylesheets**

```
<?xml:stylesheet rel=alternate
                 href="mystyle1.xsl"
                 title="Fancy"
                 type="text/xsl"?>
<?xml:stylesheet
                 rel=alternate
                 href="mystyle2.css"
                 title="Simple"
                 type="text/css"?>
<?xml:stylesheet
                 rel=alternate
                 href="mystyle2.aur"
                 title="Aural"
                 type="text/aural"?>
```

takes "paragraph" elements and "maps" them to fonts, colors and other typographic effects. LaTeX users and professional publishers are probably quite comfortable with this idea of a stylesheet.

In XSL, this construct is called a *rule*. Rules have *actions* associated with them. The actions translate elements into formatting constructs called *flow objects*. Each element in a document matches a single rule. The entire point of an XSL stylesheet is to look at each element in the document and apply the correct rule.

**Example 35-4. A simple rule**

```
<rule>
  <!-- pattern -->
  <target-element type="emph"/>

  <!-- action -->
  <font font-weight="bold">
    <children/>
  </sequence>
</rule>
```

This rule describes a pattern that matches `emph` elements and makes them bold.

## 35.4 | Flow Objects

Of course we are going to need commands that actually describe the layout of the finished product. XML provides a set of *flow objects* that represent the components of the rendered document.

Imagine a typist taking an XML "manuscript" and typing it into a word processor. He would have to use the constructs provided by the word processor, such as paragraphs, bulleted lists, hypertext links and so on. In XSL terms, those constructs are "flow objects". They are so-called because text flows from one to another, and they are each individual objects representing things like characters, paragraphs, clickable links and pages.

Simple XSL stylesheets will typically contain paragraph flow objects, external graphic flow objects, rule flow objects (for horizontal rules), table flow objects and so forth. A table flow object might in turn contain paragraphs. The paragraphs would contain character flow objects, and also "sequence" flow objects that would apply formatting like italics and bold to sequences of characters.

Conceptually, these objects form a tree. The page (or Web page) is the root. Paragraphs, tables, sequences and other "container" objects are the branches, and characters, graphics and other "atomic" objects are the leaves. The leaf objects are called "atomic" because they are not made up of any other objects, just as atoms are not made up of other atoms. The tree of flow objects is called the "flow object tree."

There is usually a relationship between your document's parse tree and the output flow object tree, but they are not identical. You could suppress elements so that they do not appear in the flow object tree. You could add text, such as boilerplate copyright text. You could combine or re-order elements and so forth. For instance, you would re-order and suppress elements to generate an index. XSL code for creating an index would surpress all elements other than those bound for the index, and reorder them alphabetically.

Every flow object has characteristics. The exact set of characteristics that a flow object exhibits depends on its *class*. For example, Web pages have scroll bars, clickable links have destinations, fonts have font sizes, and pictures have heights and widths.

The current version of XSL also has special flow objects that take advantage of many Web designers' knowledge of HTML. These are called the "HTML flow objects" and they correspond to the element types in the HTML DTD. If you format a document using them, it will look as if it

ıd been created in HTML directly. You can think of this process as a con-
ersion from XML markup to HTML markup.

The goal is usually not to create an actual HTML file, but rather to escribe the formatting of the document in terms that authors are already miliar with. In theory, the HTML document exists only conceptually. In ractice, browsers do not support XSL yet, so XSL processors really do out-ıt HTML documents that you can use on the Web as if you had created ıem by hand.

***Caution*** *At the time of writing, the future of the HTML flow objects is unclear. Many people are confused by them. Some think that XSL is related to HTML, depends on HTML, or is some kind of XML to HTML conversion language. This confusion may be enough to encourage the working group to provide DSSSL-inspired flow objects exclusively.*

Right now, however, the HTML flow objects are much more widely sup-orted than the DSSSL ones. This is because there is so much software round that already knows how to handle HTML. We will use these flow bjects for that reason. We will restrict our usage to only two of them. We ill use the HTML P flow object to create new paragraphs and the FONT ow object to apply a typographic style to a series of characters.

## 35.5 | Using XSL

he easiest way to get started with XSL is to use the simple XSL implemen-ıtion called *Sparse* that is on the CD-ROM and is also available on the Veb. The Web version is very easy to use and more up-to-date than the D-ROM version. Sparse is nice because it is interactive, fast and user iendly.

Sparse has two windows, one for XML text, and one for XSL text. You ype an XML document into one window, and an XSL stylesheet into the ther. To get started, you can type the following document and stylesheet ıto the correct windows:

XML

```
<?XML version="1.0"?>
<para>This is a paragraph.</
para>
<para>This is too, but it has
<emph>emphasis</emph>.</para>
```

XSL

```
<rule>
  <!-- look for paragraphs -->
  <target-element type="para"/>

  <!-- turn them into paragraph
flow objects -->
  <P color="blue">
    <children/>
  </P>
</rule>

<rule>
  <!-- look for emphasis -->
  <target-element type="emph"/>

  <FONT color="red"><children/
></FONT>
</rule>
```

This formats `para` elements as blue paragraphs, and makes `emph` elements red.

## 35.6 | Patterns

Patterns allow the XSL processor to choose which elements to apply which style rules to. Every pattern has a `target-element` that specifies the elements to be matched in the document. You can supply a `type` attribute to force the `target-element` to only match elements of a certain element type. So this rule bolds only `emph` elements:

```
<rule>
  <target-element type="emph"/>

  <FONT font-weight="bold">
    <children/>
  </FONT>
</rule>
```

On the other hand, the following rule bolds elements of *any* type:

```
rule>
 <target-element/>

 <FONT font-weight="bold">
   <children/>
 </FONT>
/rule>
```

If a rule matches elements of any type, we call it the *default rule*, because t is the rule that gets called when no other rule matches. You should almost lways have a default rule. If you do not have a default rule, and you forget o supply a rule for a particular case, then elements of that type are cycled-hrough but do not appear in the output.

We can also choose to match only elements with certain attributes:

```
rule>
 <target-element>
    <attribute name=security value="top-secret"/>
 </target-element>

 <sequence color="red">
   <children/>
 </sequence>
/rule>
```

An `attribute` element has attributes `name`, `value` and `has-value`. You nust always specify the name, but you can either require the attribute to onform to a particular value (`top-secret` in our example), or you can nerely ask whether it conforms or not.

It is possible to match based on an element's context. For instance if we vanted to change the color of all elements inside of a `warning` element, we ould do so with the following rule:

```
rule>
 <warning>
   <target-element type="p"/>
  </warning>

 <sequence color="red">
/rule>
```

Essentially, the elements that surround the `target-element` define a pat-ern that is matched against elements in the document. We can do the same vith the target element's sub-elements.

```
<rule>
  <warning>
    <target-element type="p"/>
      <footnote>
    </target-element>
   </warning>

  <sequence color="red">
</rule>
```

This rule would only match elements of type `p` that contain at least one `footnote` and are inside a `warning`.

XSL's patterns are very powerful and convenient. But it is when you pair them with actions that you can actually make something exciting.

## 35.7 | Actions

After the XSL processor chooses a rule based on a pattern match, it looks at the action part of the rule. The action says what objects to create in the output tree. The actions can directly create flow objects, add literal text (such as boilerplate text), and tell the XSL processor to process the source element's content to get access to its flow objects. We can start with the simplest case first. Let's assume that the input document had an `hr` element that was to be turned into a `horizontal-rule` flow object. You can do that this way:

```
<rule>
    <target-element type="hr"/>

    <horizontal-rule/>
</rule>
```

That is simple because we can assume that `hr` is an empty element and has no content. But most elements do have content. The content is made up of character data and sub-elements. We call each such character string or sub-element a *child*. We can process children by using the `children` element. For instance if we want to put a horizontal rule before and after sections, and then process the children of the section, we would do this:

```
<rule>
    <target-element type="section"/>

    <horizontal-rule/>
    <children>
    <horizontal-rule/>
</rule>
```

We can also mix in literal text:

```
rule>
   <target-element type="warning"/>

   <horizontal-rule/>
   <sequence font-size="20" font-weight="bold">
       WARNING:
   <sequence>
   <children>
   <horizontal-rule/>
/rule>
```

We've almost covered enough to do interesting things now. The last step s actually the easiest. You just have to learn which flow objects and characteristics are available for use in your XSL stylesheets.

## 35.8 | Flow objects and characteristics

As we described earlier, there are two major sets of flow objects. There are those based upon HTML and those based upon DSSSL. They are not meant to be used together. The HTML ones are easy to use if you already know HTML, because they look and behave exactly as HTML elements do on the Web. Here is the list of HTML element types that are available:

These element types can be created with the attributes that they would usually have according to the HTML spec. You can also provide them with attributes that represent style properties from the *Cascading Style Sheet Language*. You can get that full list from the *XML SPECtacular* on the CD-ROM.

The DSSSL flow objects provided are:

Information about these flow objects is available through a link on the CD-ROM. Each of them is described in a chapter of the DSSSL specification.

## 35.9 | XSL and JavaScript

Sometimes formatting tasks can be quite tricky. They can be so tricky that you need the full power of a programming language to solve them. For instance, as a visual aid you might want every second row in a table to be blue, or every title with a numeric attribute in a particular range to be in a

**Spec. Reference 35-2. HTML flow objects in XSL**

```
SCRIPT
    PRE
    HTML
        TITLE
        META
        BASE
    BODY
    DIV
    BR
    SPAN
    TABLE
        CAPTION
        COL
        COLGROUP
        THEAD
        TBODY
        TFOOT
        TR
        TD
    A
    FORM
        INPUT
        SELECT
        TEXTAREA
    HR
    IMG
        MAP
        AREA
    OBJECT
        PARAM
        FRAMESET
```

particular font, and other titles to be in a different font. These more complicated tasks must be solved with the help of a full programming language. Declarations alone are not enough.

XSL embeds such a language in the form of ECMAScript, a standardized version of JavaScript. Although XSL technically uses the standardized form, which is slightly different from JavaScript, we will refer to it as JavaScript for familiarity. The differences are minor. The JavaScript variant in XSL is essentially the same language used to make many Web pages interactive, but with new features designed for applying style to documents. JavaScript itself is beyond the scope of this book, and there are many good books that already cover it. We will just demonstrate how XSL and JavaScript fit together.

**Spec. Reference 35-3. DSSSL flow objects in XSL**

```
scroll--used for online display
paragraph, paragraph-break--used for paragraphs
character--used for text
line-field--used for lists
external-graphic--used for including graphic images
horizontal-rule, vertical-rule--used for rules
score--used for underlining and scoring
embedded-text--used for bi-directional text
box--used for borders
flow objects for tables
    table
    table-part
    table-column
    table-row
    table-cell
    table-border
sequence--used for specifying inherited characteristics
display-group--used for positioning flow objects
simple-page-sequence--used for simple page layout
link--used for hypertext links
```

Just as you can use JavaScript on a Web page when HTML "runs out of steam", you can use it in your stylesheets to solve the harder problems that simple declarations cannot handle.

When JavaScript is used in XSL documents, it has access to many advanced features of the underlying XML system, and this makes style application easier. For instance, there is a query language for selecting and returning document components. This is analogous to the way that database query languages such as SQL select and return particular rows from a database.

For instance, you might want to generate text at the source of a cross-reference based on data at the target. A typical cross reference would say something like: "See Chapter 1.1 'Intro to Foobar'". Of course the chapter number and title must be fetched from that other part of the document. XSL's query features give you this access to information anywhere in the document.

Here is an example of an XSL stylesheet with JavaScript code in it:

Now you can understand the reason for two other XSL design principles:

- XSL should provide a declarative language to do all common formatting tasks.

**Example 35-5. JavaScript stylesheet**

```
<xsl>
  <define-script>
    var defaultFontSize = "12pt";

    function hierarchicalIndent(elementType, element){
    return length(hierarchicalNumberRecursive(
                  elementType, element)) * 12pt;
    }

    </define-script>
    <rule>
     <element type="list">
      <target-element type="item">
      </element>
      <DIV font-size="=defaultFontSize"
       margin-left='=1in+hierarchicalIndent(element, "item")'>
       <children/>
      </DIV>
    </rule>
</xsl>
```

- XSL should provide an "escape" into a scripting language to accommodate more sophisticated formatting tasks and to allow for extensibility and completeness.

***Tip*** *You may find the XML Styler helpful when create XSL stylesheets. The software is free and is described in detail in Chapter 25, "XML Styler: Graphical XSL stylesheet editor", on page 338.*

# Advanced features

- Conditional sections
- Character references
- Processing instructions
- Standalone declaration

Chapter

# 36

The features in this chapter are advanced in the sense that only advanced users will get around to reading them. They do not require advanced degrees in computer science or rocket science to understand. They are just a little esoteric. Most XML users will get by without ever needing to use them.

## 36.1 | Conditional sections

Conditional sections can only occur in the external subset of the document type declaration, and in external entities referenced from the internal subset. The internal subset proper is supposed to be quick and easy to process. In contrast, the external subset is supposed to retain some of the full-SGML mechanisms that make complicated DTDs easier to maintain. One of these mechanisms is the conditional section, which allows you to turn on and off a series of markup declarations.

Like the internal and external subsets, conditional sections may contain one or more complete declarations, comments, processing instructions, or nested conditional sections, with optional white space between them.

A conditional section is turned on and off with a keyword. If the keyword is INCLUDE, then the section is processed just as if the conditional section markers did not exist. If the keyword is IGNORE, then the contents are ignored by the processor as if the declarations themselves did not exist.

**Example 36-1. Conditional sections**

```
<![INCLUDE[
 <!ELEMENT magazine (title, article+, comments* )>
 ]]>
 <![IGNORE[
 <!ELEMENT magazine (title, body)>
]]>
```

This is a useful way of turning on and off parts of a DTD during development.

The real power in the feature derives from parameter entity references. These are described in 33.7, "Internal and external parameter entities", on page 487.

If the keyword of the conditional section is a parameter entity reference, the processor replaces the parameter entity by its content before the processor decides whether to include or ignore the conditional section. That means that by changing the parameter entity in the internal subset, you can turn on and off a conditional marked section. In that way, two different documents could reference the same set of external markup declarations, but get slightly (or largely) different DTDs. For instance, we can modify the example above:

**Example 36-2. Conditional sections and parameter entities**

```
<![%editor[
 <!ELEMENT magazine (title, article+, comments* )>
 ]]>
 <![%author[
 <!ELEMENT magazine (title, body)>
]]>
```

Now editors will have a slightly different DTD from authors. When the parameter entities are set one way, the declaration without comments is chosen:

**Example 36-3.**

```
<!DOCTYPE MAGAZINE SYSTEM "magazine.dtd"[
   <!ENTITY % editor "IGNORE">
   <!ENTITY % author "INCLUDE">
]>
```

Authors do not have to worry about `comments` elements that they are not supposed to use anyway. When the document moves from the author to the editor, the parameter entity values can be swapped, and the expanded version of the DTD becomes available. Parameter entities can also be used to manage DTDs that go through versions chronologically, as an organiztion's needs change.

Conditional sections are also sometimes used to make "strict" and "loose" versions of DTDs. The loose DTD can be used for compatibility with old documents, or documents that are somehow out of your control, and the strict DTD can be used to try to encourage a more precise structure for future documents.

## 36.2 | Character references

It is not usually convenient to type in characters that are not available on the keyboard. With many text editors, it is not even possible to do so. XML allows you to insert such a character with a *character reference*. If, for instance, you wanted to insert a character from the "International Phonetic Alphabet", you could spend a long time looking for a combination of keyboard, operating system and text editor that would make that straightforward. Rather than buying special hardware or software, XML allows you to refer to the character by its Unicode number.

Here is an example:

**Example 36-4. Unicode character**

```
<P>Here is a special character from Unicode: &#161;.
```

That includes the character numbered 161 in Unicode, which happens to be the inverted exclamation mark. If you happen to know it, you could also use that character's hex value, by using a slightly different form of reference:

**Example 36-5. Unicode character**

```
<P>Here is a special character from Unicode: &#xA1;.
```

Hex is a numbering system often used by computer programmers that translates naturally into the binary codes that computers use. The *Unicode Standard book* uses hex, so those that have that book will probably prefer this type of character reference over the other (whether they are programmers or not).

Here are the specifics on character references from the XML spec:

**Spec. Reference 36-1. Character reference**

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

**Spec. Reference 36-2. Interpreting character references**

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

For our purposes, ISO/IEC 10646 is essentially Unicode. Think of Unicode as industry market-speak for the ISO version of the standard.

Note that character references are not entity references, though they look similar to them. Entities have names and values, but character references only have numbers. In a well-formed document, all entities except the predefined ones must be declared and in a valid document even the predefined ones must be declared. But numeric references are never declared.

Because Unicode numbers are hard to remember, it is often useful to declare entities that stand in for them:

```
<!ENTITY inverted-exclamation "&#161;">
```

Most likely this is how most XML users will refer to obscure characters. There will probably be popular character entity sets that can be included in a DTD through a parameter entity. This will free them from learning obscure character numbers and probably even from learning how to use character references.

## 36.3 | Processing instructions

XML comments are for those occasions where you need to say something to another human being without reference to the DTD, and without changing the way the document looks to readers or applications. Processing instructions are for those occasions where you need to say something to *a computer program* without reference to the DTD and without changing the way that the document is processed by other computer programs. This is only supposed to happen rarely.

Many people argued that the occasions would be so rare that XML should not have processing instructions at all. But as one of us (Charles) said in *The SGML Handbook*: "In a perfect world, they would not be needed, but, as you may have noticed, the world is not perfect." It turns out that processing instruction use has changed over the years and is not as frowned upon as it was in the early days of SGML.

Processing instructions are intended to reintroduce software-specific markup. You might wonder why you would want to do that. Imagine that you are creating a complex document, and, like a good user of a generalized markup language, you are concentrating on the structure rather than the formatting. Close to the deadline you print the document using the proprietary formatting system that has been foisted on you by your boss. There are many of these systems, some of which are of fantastic quality and others which are not.

Your document looks reasonable, but you need a way to make the first letter of each paragraph large. However, reading the software's manual, you realize that the formatter does not have a feature that allows you to modify the style for the first letter of a word. The XML Purist in you might want to go out and buy a complete formatting system but the Pragmatist in you knows that that is impossible.

Thinking back to the bad-old days of "What You See is All You Get" word processors, you recall that all you really needed to do is to insert a

code in the beginning of each paragraph to change the font for the first letter. This is not good "XML Style" because XML Purists do not insert formatting codes and they especially do not insert codes specific to a particular piece of software – that is not in the "spirit" of generalized markup. Still, in this case, with a deadline looming and stubborn software balking, a processing instruction may be your best bet. If the formatter has a "change font" command it may be accessible through a processing instruction:

**Example 36-6. Processing instruction**

```
<CHAPTER>The Bald and the Dutiful
<P><?DUMB-FORMATTER.FONT="16PT"?>N<?DUMB-FORMATTER.ENDFONT?>ick
took Judy in his arms</P>
```

If you find yourself using many processing instructions to specify formatting you should try to figure out what is wrong with your system. Is your document's markup not rich enough? Is your formatting language not powerful enough? Are you not taking advantage of the tools and markup you have available to you? The danger in using processing instructions is that you can come to rely on them instead of more reusable structural markup. Then when you want to reuse your information in another context, the markup will not be robust enough to allow it.

Processing instructions start with a fixed string "<?". That is followed by a name and, after that, any characters except for the string that ends the PI, "?>".

Here are the relevant rules from the XML specification:

**Spec. Reference 36-3. Processing Instruction**

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

This name at the beginning of the PI is called the *PI target*. This name should be standardized in the documentation for the tool or specification. After the PI target comes white space and then some totally proprietary command. This command is not processed in the traditional sense at all. Characters that would usually indicate markup are totally ignored. The command is passed directly to the application and it does what it wants to with it. The command ends when the processor hits the string "?>". There

is absolutely no standard for the "stuff" in the middle. Markup is not recognized there. PIs could use attribute syntax for convenience, but they could also choose not to.

It is possible that more than one application could understand the same instructions. They might come from the same vendor or one vendor might agree to accept another vendor's commands. For instance in the early days of the Web, the popular NCSA (National Center for Supercomputing Activities) Web Server introduced special commands into HTML documents in the form of special HTML comments. Because the NCSA server was dominant in those days, many servers now support those commands.

Under XML we would most likely use processing instructions for the same task. The virtue of XML processing instructions in this case is that they are explicitly instructions to a computer program. In our opinion, one of the central tenets of generalized markup is that it is important to be *explicit* about what is going on in a document. Reusing markup constructs for something other than what they were intended for is not explicit.

For instance, since comments are meant to be instructions to users, an ambitious Web Server administrator might decide to write a small script that would strip them out to save download time and protect internal comments from being read by others. But if instructions to software (like the NCSA server commands) were hidden in comments, they would be stripped out as well. It would be better to use the supplied processing instruction facility, which was designed for the purpose.

Better still (from a purist's point of view) would be a robust XML-smart mechanism for accomplishing the task. For instance, one thing that the NCSA servers do is include the text of one HTML file into another. XML's entity mechanism (see Chapter 33, "Entities: Breaking up is easy to do", on page 476) can handle this, so you do not need processing instructions in that case.

If you want to insert the date into a document, then you could connect the external entity to a CGI[1] that returns the date. If you want to insert information from a database then you could have software that generates XML entities with the requested information.

Sometimes, though, the processing instruction solution may be the most expedient. This is especially the case if your application vendor has set it up

1. CGI is the "Common Gateway Interface", a specification for making Web pages that are generated by the server when the user requests them, rather than in advance.

that way. If your document is heavily dependent on a database or other program, then it is not very "application independent" in any case. If a document is inherently dependent on an application then you may decide that strictly adhering to generalized markup philosophy is just too much work. In the end you must choose between expediency and purity. Most people mix both.

Processing instructions are appropriate when you are specifying information about a document that is unrelated to the actual structure of the document. Consider, for instance, the problem of specifying which stylesheets go with which XML documents on a web site. Given enough money and time you could erect a database that kept track of them. If you already had your XML documents in a text database then this would probably be the most efficient mechanism. If you did not have a text database set up, then you could merely keep the information in a flat text file. But you would have to keep that external information up-to-date and write a program to retrieve it in order to do formatting. It would probably be easier to simply stick the information somewhere in the file where it is easy to find (such as at the beginning).

You could add a STYLESHEET element or attribute to each document, but that could cause three problems. First, it would violate the XML Purist principle that elements should represent document components and not formatting or other processing information. Second, if you are using DTDs with your documents then you must add the element or attribute to each DTD that you will be using. This would be a hassle.

The third reason to use processing instructions instead of elements is the most concrete: you may not be able to change those DTDs. After all, DTDs are often industry (or international!) standards. You cannot just go monkeying around with them even if you want to. Instead, you could put a processing instruction at the start of each document. Processing instructions are not associated with particular DTDs and they do not have to be declared. You just use them.

As we described in an earlier chapter (page 519), XML provides a processing instruction for including stylesheets:

**Example 36-7. Stylesheet PI**

```
<?xml:stylesheet
          href="http://www.sgmlsource.com/memo.xsl"
          type="text/xsl"?>
```

Note that the stylesheet processing instruction does not really add anything to the content or structure of the document. It says something about how to *process* the document. It says: "This document has an associated stylesheet and it is available at such and such a location." It is not always obvious what is structural information and what is merely processing information. If your instruction must be embedded in documents of many types, or with DTDs that you cannot change, then processing instructions are typically your best bet.

The XML encodingPI is an example of another processing instruction. It says what character encoding the file uses. Again this information could be stored externally, such as in a database, a text file or somewhere else, but XML's designers decided (after weeks of heated discussion) that it would be most convenient to place it in the XML document itself rather than require it to be stored (and transmitted across the Internet) externally.

If you go back to 31.6.1, "XML declaration", on page 439 you will also notice that the XML declaration has the same prefix ("<?") and suffix ("?>") as processing instructions do. Formally speaking, the XML declaration is a special form of processing instruction. From an SGML processor's point of view, it is a processing instruction that controls the behavior of a particular class of software: XML processing software. Software that treats XML as just another kind of SGML will ignore it, as they do other types of processing instructions.

To summarize: PIs (processing instructions) were invented primarily for formatting hacks but based on our experience with SGML we know that they are more widely useful. There are already predefined processing instructions in the XML specification for some kinds of processing. Processing instructions will probably be used for other things in the future. Everything that can be accomplished with PIs would be accomplished by other means in a perfect world of pure generalized markup, but in the real world they are often convenient.

## 36.4 | Standalone document declaration

We should start by saying that the standalone document declaration is only designed for a small class of problems, and these are not problems that most XML users will run into. We do not advise its use. Nevertheless, it is part of

XML and we feel that you should understand it so that you can understand why it is seldom useful.

A DTD is typically broken into two parts, an external part that contains declarations that are typically shared among many documents, and an internal part that occurs within the document and contains declarations that only that document uses (see Chapter 32, "Creating a document type definition ", on page 448). The external part includes all external parameter entities, including both the external subset of the document type declaration and any external entities referenced from the internal subset.

The DTD describes the structure of the document, but it can also control the interpretation of some of the markup and declare the existence of some other entities (such as graphics or other XML documents) that are required for proper processing. For instance, a graphic might only be used in a particular document, so the declaration that includes it (an *entity declaration*) would usually go in the internal subset rather than the external one.

Processors that validate a document need the entire DTD to do so. A document is not valid unless it conforms to both the internal and external parts of its DTD. But sometimes a system passes a document from program to program and it does not need to be validated at each stage. For instance, two participants in an electronic data interchange system might agree that the sender will validate the document once, instead of having both participants validate it.

Even though the receiving processor may not be interested in full validation, it may need to know if it understands the document in exactly the same way that the sender did. Some features of the DTD may influence this slightly. Documents with defaulted attributes would be interpreted differently if the attribute declarations are read rather than ignored. Entity declarations would allow the expansion of entity references. Attribute values can only be normalized according to their type when the attribute declarations are read. Some white space in content would also be removed if the DTD would not allow it to be interpreted as text.

If a process can reliably skip a part of the DTD dedicated exclusively to validation, then it would have less data to download and process and could let the application do its work (browsing, searching, etc.) more quickly. But it would be important for some "mission critical" applications to know if they are getting a slightly different understanding of the document than they would if they processed the entire DTD.

The *standalone document declaration* allows you to specify whether a processor needs to fetch the external part of the DTD in order to process the

document "exactly right." The Standalone document declaration may take the values (case sensitive) of yes and no.

A value of yes says that the document is *standalone* and thus does not depend on the external part of the DTD for correct interpretation. A value of *no* means that it either depends on the external DTD part or it might, so the application should not trust that it can get the correct information without it. You could always use no as the value for this attribute, but in some cases applications will then download more data than they need to do their jobs. This translates into slower processing, more network usage and so forth.

**Example 36-8. A standalone document declaration that forces processing of the internal subset.**

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE MEMO SYSTEM "http://www.sgmlsource.com/memo.dtd" [
<!ENTITY % pics SYSTEM "http://www.sgmlsource.com/pics.ent">
 %pics;
]>
<MEMO></MEMO>
```

Example 36.4, "Standalone document declaration", on page 541 will tell the application that unless the processor fetched the pictures, the application might get a slightly different understanding of the document than it would if it processed the whole document. For instance, the MEMO element might have defaulted attributes.

But if the value is *yes*, the receiving application may choose not to get the external part of the DTD. This implies that it will never know what was in it. Still, it needs to be able to trust the accuracy of the declaration. What if the security level for a document is set in an attribute and the default level is top-secret? It would be very bad if a careless author could obscure that with a misleading standalone document declaration. In the scenario we outlined, the sender has already validated the document. So the sender has enough information to check that the information is correct. The XML specification requires a validating processor to do this (see Specification reference 36-4).

The last one is very likely to happen. Often people use white space between tags to make the source XML document readable, but that can slightly change the interpretation of the document. Validating processors will tell applications that there are some contexts where character data is not

**Spec. Reference 36-4. Standalone document declaration**

The standalone document declaration must have the value "no" if any external markup declarations contain declarations of:

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

legal, so the white space occuring in those places must be merely formatting white space (see 32.5, "Content models", on page 459). If an application that does not want to validate a document is to get exactly the same information out of the document, it must know whether there are any elements where white space should be interpreted just as source formatting. We say that this sort of white space is *insignificant.*

The standalone document declaration warns the application that this is the case so that mission critical applications may download the DTD just to get the right information out of the document, even when they are not interested in validating it.

The standalone document declaration is fairly obscure and it is doubtful if it will get much use outside of a few mission critical applications. Even there, however, it is safest to just get the external data and do a complete validation before trusting a document. You might find that it had been corrupted in transit.

## 36.5 | Is that all there is?

We've pretty much covered all the details of XML, certainly all that are likely to see extensive use. There are some things we didn't touch on, such as restrictions that must be observed if you are using older SGML tools to

process XML. As the generalized markup industry is retooling rapidly for XML, such restrictions will be short-lived and, we felt, did not warrant complicating our XML tutorial.

In any case, you are now well-prepared – or will be after reading Chapter 37, "Reading the XML specification", on page 546 – to tackle the XML spec yourself. You'll find it in the XML SPECtacular section of the CD-ROM that accompanies this book.

# Reading the XML specification

- Grammars
- Rules
- Symbols

Chapter

# 37

The XML specification is a little tricky to read, but with some work you can get through it by reading and understanding the glossary and applying the concepts described so far in this book. One thing you'll need to know is how to interpret the production rules that make up XML's grammar. This chapter teaches how to read those rules.

When discussing a particular string, like a tag or declaration, we often want to discuss the parts of that string individually. We call each part of the string a *token*. Tokens can always be separated by white space as described above. Sometimes the white space between the tokens is required. For instance we can represent the months of the year as tokens:

**Example 37-1. Tokens**

```
JANUARY FEBRUARY MARCH APRIL MAY JUNE
```

White space between tokens is *normalized* (combined) so that no matter how much white space you type, the processor treats it as if the tokens were

separated by a single space. Thus the example above is equivalent to the following:

**Example 37-2. Tokens after normalization**

```
JANUARY FEBRUARY MARCH APRIL MAY JUNE
```

Whenever we discuss strings made up of tokens, you will know that you can use as much white space between tokens as you need and the XML processor will normalize it for you.

## 37.1 | A look at XML's grammar

There are two basic techniques that we could use to discuss XML's syntax precisely. The first is to describe syntactic constructs in long paragraphs of excruciatingly dull prose. The better approach is to develop a simple system for describing syntax. In computer language circles, such systems are called grammars. Grammars are more precise and compact. Although they are no less boring (as you may recall from primary school), you can skip them easily until you need to know some specific detail of XML's syntax.

As a bonus, once you know how to read a grammar, you can read the one in the XML specification and thus work your way up to the status of "language lawyer".[1] As XML advances, an ability to read the specification will help you to keep on top of its progress.

The danger in this approach is that you might confuse the grammar with XML markup itself. The grammar is just a definitional tool. It is not used in XML applications. You don't type it in when you create an XML document. You use it to figure out what you can type in. Before "the new curriculum", students were taught grammar in primary school. They would be taught parts of speech and how they could combine them. XML's grammar is the same. It will tell you what the parts of an XML document are, and how you can combine them.

Grammars are made up of production *rules* and *symbols*. Rules are simple: they say what is allowed in a particular place in an XML document.

1. You too can nitpick about tiny language details and thus prove your superiority over those who merely use XML rather than obsess over it.

Rules have a symbol on the left side, the string "::=" in the middle and a list of symbols on the right side:

```
people ::= 'Melissa, ' 'Tiffany, ' 'Joshua,' 'Johan'
```

If this rule were part of the grammar for XML (which it is not!) it would say that in a particular place in an XML document you could type the names listed.

The symbols on the right (the names, in the last example) define the set of allowed values for the construct described by the rule ("people"). An allowed value is said to *match*. Rules are like definitions in a dictionary. The left side says what is being defined and the right side says what its definition is. Just as words in a dictionary, are defined in terms of other words, symbols are defined in terms of other symbols. Rules in the XML grammar are preceded by a number. You can look the rule up by number. If an XML document does not follow all of the XML production rules, it is not *well-formed.*

## 37.2 | Constant strings

The most basic type of symbol we will deal with is a *constant string.* These are denoted by a series of characters in between single quote characters. Constant strings are matched case-sensitively (as we discussed earlier). Here are some examples:

```
AlphabetStart ::= 'ABC'
Example1 ::= '<!DOCTYPE'
```

This would match (respectively) the strings

```
ABC
<!DOCTYPE
```

When we are discussing a constant string that is an English word or abbreviation, we will refer to it as a keyword. In computer languages, a keyword is a word that is interpreted specially by the computer. So your mother's maiden name is not (likely) a keyword, but a word like `#REQUIRED` is.

Symbols in XML's grammar are separated by spaces, which means that you must match the first, and then the second, and so on in order.

```
AlphabetStartAndEnd ::= 'ABC' 'XYZ'
NumbersAndLetters ::= '123' 'QPZ'
```

These would match:

```
ABCXYZ
123QPZ
```

Note that a space character in the grammar does not equate to white space in the XML document. Wherever white space can occur we will use the symbol "S". That means that wherever the grammar specifies "S", you may put in as much white space as you need to make your XML source file maintainable.

```
SpacedOutAlphabet ::= 'ABC' S 'XYZ'
```

matches:

```
ABC XYZ
ABC     XYZ
ABC                XYZ
```

This is the first example we have used where a single rule matches multiple strings. This is usually the case. Just as in English grammar there are many possible verbs and nouns, there are many possible strings that match the rule `SpacedOutAlphabet`, depending on how much white space you choose to make your XML source file maintainable.

Obviously XML would not be very useful if you could only insert predefined text and white space. After all, XML users usually like to choose the topic and content of their documents! So they need to have the option of inserting their own content: a *user defined string*. The simplest type of user defined string is character data. This is simply the text that isn't markup. You can put almost any character in character data. The exceptions are characters that would be confused with markup, such as less-than and ampersand symbols.

## 37.3 | Names

The XML specification uses the symbol "Name" to represent names. For example:

```
PersonNamedSmith :: = Name S 'Smith'
```

When we combine the name, the white space and the constant string, the rule matches strings like these:

```
Christina Smith
Allan       Smith
Michael      Smith
Black          Smith
Bla_ck           Smith
_Black            Smith
```

## 37.4 | Occurrence indicators

Sometimes a string is *optional.* We will indicate this by putting a question mark after the symbol that represents it in a rule:

```
Description ::= 'Tall' S? 'dark'? S? 'handsome'? S? 'person'
Tall person
Tallperson
Tall handsomeperson
Tall dark person
Talldarkhandsomeperson
```

Notice that optionality does not affect the order of the tokens. For example, dark can never go before tall. We can also allow a part of a rule to be matched multiple times. If we want to allow a part to be matched one or more times, we can use the plus symbol and make it *repeatable.*

```
VeryTall ::= 'A' S ('very' S)+ 'tall' S 'person.'
A very tall person.
A very very tall person.
A very very very tall person.
```

An asterisk is similar, but it allows a string to be matched zero or more times. In other words it is both repeatable and optional.

```
VerySmall ::= 'A' S ('very' S)* 'small' S 'person.'
A small person.
A very small person.
A very very small person.
A very very very small person.
```

Symbols can be grouped with parentheses so that you could, for instance, make a whole series of symbols optional at once. This is different from making them each optional separately because you must either supply strings for all of them or none:

```
Description2 ::= 'A' S ('tall' S 'dark' S 'handsome' S)? 'man.'
```

This rule matches these two strings (and no others):

```
A tall dark handsome man.
A man.
```

We will sometimes have a choice of symbols to use. This is indicated by separating the alternatives by a vertical bar:

```
Description3 ::= 'A' S ('short'|'tall') S
                 ('fair'|'tan'|'dark') S ('man'|'woman') '.'
A tall dark man.
A short fair woman.
A short tan man.
A tall dark woman.
```

Note that we broke a single long rule over two lines rather than having it run off of the end of the page. This does not in any way affect the meaning

of the rule. Line breaks are just treated like space characters between the symbols.

We can combine all of these types of symbols. This allows us to make more complex rules.

```
Book ::= (('Fascinating'|'Intriguing') S ('XML'|'SGML') S 'Book')
                    | ('Yet another HTML' S 'Book')
Fascinating XML Book
Yet another HTML Book
Intriguing SGML Book
```

So in this case, you should treat the first large parenthesized expression (saying good things about SGML and XML books) as one option, and the second (saying not as good things about HTML books) as another. Inside the first set, you can choose different adjectives and book types, but the ordering is fixed and there must be white space between each part.

## 37.5 | Combining rules

Finally, rules can refer to other rules. Where one rule refers to another, you just make a valid value for each part and then put the parts together like building blocks.

```
FunnyDate ::= Month S Day ',' Year
Month ::= 'Jan'|'Feb'|'Mar'|'Apr'|'May'|'Jun'
                |'Jul'|'Aug'|'Sep'|'Oct'|'Nov'|'Dec'
Day ::= ('1'|'2'|'3')?
                ('1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|'0')
Year ::= '1998'|'1999'|'2000'|'2001'|'2002'
```

This would match strings such as:

```
Jan 21,1998
May 35,2000
Sep 2,2002
```

As you can see, this is not quite a strict specification for dates, but it gets the overall form or *syntax* of them right.

## 37.6 | Conclusion

We've explained the bulk of what is needed to understand XML's production rules. There are a few more details that you can find in section 6 of the XML spec itself. It is included in the XML SPECtacular on the CD-ROM.