1

XPath 2.0 in Context

This chapter explains what kind of language XPath is, and some of the design thinking behind it. It explains how XPath relates to the other specifications in the growing XML family, and to describe what's new in XPath 2.0 compared with XPath 1.0.

The chapter starts with an introduction to the basic concepts behind the language, its data model and the different kinds of expression it supports. This is followed by a survey of new features, since I think it's likely that many readers of this book will already have some familiarity with XPath 1.0. I also introduce a few software products that you can use to try out these new features.

The central part of the chapter is concerned with the relationships between XPath and other languages and specifications: with XSLT, with XML itself and XML namespaces, with XPointer, with XQuery, and with XML Schema. It also takes a look at the way XPath interacts with Java and with the various document object models (DOM and its variations).

The final section of the chapter tries to draw out the distinctive features of the language, the things that make XPath different. The aim is to understand what lies behind the peculiarities of the language, to get an appreciation for the reasons (sometimes good reasons and sometimes bad) why the language is the way it is. Hopefully, with this insight, you will be able to draw on the strengths of the language and learn to skirt round its weaker points.

What Is XPath?

This is how the XPath 2.0 specification describes the language:

> *XPath 2.0 is an expression language that allows the processing of values conforming to the [XPath] data model…The data model provides a tree representation of XML documents, as well as atomic values such as integers, strings, and booleans, and sequences that may contain both references to nodes in an XML document and atomic values. The result of an XPath expression may be a selection of nodes from the input documents, or an atomic value, or more generally, any sequence allowed by the data model. The name of the language derives from its most distinctive feature, the path expression, which provides a means of hierarchic addressing of the nodes in an XML tree.*

So what is this trying to say?

Chapter 1

Firstly, XPath is an expression language. It isn't described as a programming language or a query language, and by implication, it has less functionality than you would expect to find in a programming language or a query language. The most common kind of expression in XPath is one that takes an XML document as input, and produces a list of selected nodes as output: that is, an expression that selects some of the nodes in a document. XPath specializes in making it easy to access data in XML documents. However, XPath expressions are rather more general than this. «`2+2`» is a valid XPath expression, as is «`matches($input,"[a-z]*[0-9]")`».

The language is described in this summary primarily by reference to the kinds of data that it manipulates, that is, its data model. The question of what you are actually allowed to do with these data values is secondary. The data model for XPath 2.0 (which is shared also by the closely-related languages XSLT 2.0 and XQuery 1.0) provides essentially three building blocks, all mentioned in this summary:

- Atomic values of various types including strings, integers, booleans, dates, times, and other more specialized types such as QNames and URIs
- Trees consisting of nodes, which are used to represent the content of an XML document
- Sequences (or lists), whose items are either atomic values, or references to nodes in a tree. Sequences can't be nested.

We will be discussing this data model in considerable detail in Chapter 2. It's worth spending time on this, because understanding the data model is the key to understanding the language.

The expressions that you can write in XPath perform operations on these data values. XPath is a read-only language: it cannot create new nodes or modify existing nodes (except by calling functions written in a different language). It can however create new atomic values and new sequences. There's an important difference, which is that nodes (like objects in Java) have identity, whereas atomic values and sequences don't. There is only one number «`2`», and there is only one sequence «`1,2,3`» (or if you prefer, there is no way of distinguishing two sequences that both contain the values «`1,2,3`»), but there can be any number of distinct `<a/>` nodes.

Given that there are three broad categories of data values, there are similarly three broad categories of operations that can be performed:

- *Operations on atomic values*: These include a range of operators and functions for adding numbers, comparing strings, and the like. Example expressions in this category are «`price * 1.1`» and «`discount > 3.0`». Many of these operations are likely to be familiar from other more conventional languages, though there are a few surprises in store. Chapter 6 of this book describes these operations in detail.
- *Operations that select nodes in a tree*: The most powerful expression here, which forms the heart of XPath, is the path expression. An example of a path expression is «`book[author="Kay"]/@isbn`» which selects the «`isbn`» attributes of all the `<book>` elements that have a child `<author>` element with the value «`Kay`». Path expressions are analyzed in detail in Chapter 7, but I will have a few introductory words to say about them in this chapter.
- *Operations on sequences*: Here, the most important construct is the «`for`» expression. This applies an expression to every item in a sequence, forming a new sequence from the results. As an example, the result of the expression «`for $i in 1 to 5 return $i*$i`» is the sequence «`1, 4, 9, 16, 25`». (XPath variables are prefixed with a «`$`» sign to distinguish them from XML element names.) In practice, the items in the input sequence are more likely to be nodes: the

expression «`for $n in child::* return name($n)`» returns a list containing the names of the child elements of the current node in the document. The «`for`» expression is referred to as a *mapping expression*, because it performs an item-by-item mapping from an input sequence to an output sequence. The «`for`» expression, and other operations on sequences, are described in Chapter 8 of this book.

We will now take a closer look at path expressions, which give XPath its distinctive flavor.

Path Expressions

A typical path expression consists of a sequence of steps, separated by the «`/`» operator. Each step works by following a relationship between nodes in the document. This is in general a one-to-many relationship. The different relationships are called axes. The most commonly used axes are:

- The `child` axis selects the children of the starting node, that is, the elements and text nodes (and perhaps comments and processing instructions) that are found between the begin and end tags of an element, assuming the starting node is an element.
- The `attribute` axis selects the attributes of an element.
- The `ancestor` axis selects all the elements that enclose the starting node in the XML document. The last ancestor is generally a node that represents the document itself: this is called the document node, and it is distinct from the node that represents the outermost element.
- The `descendant` axis selects the children of a node, the children of the children, and so on recursively, down to the leaves of the tree.

In the full form of a step, the axis name is followed by a construct called a `NodeTest` that indicates which nodes are to be selected. Often this consists simply of a node name, or it might be «`*`» to select all elements or attributes. So «`child::title`» selects the `<title>` elements that are children of the current node (there may be more than one), «`ancestor::*`» selects all the ancestor elements, and so on.

The «`/`» operator strings together a sequence of steps into a path. So, for example, «`child::book/child::chapter/attribute::status`» selects the «`status`» attributes of all the chapters of all the books that are children of the starting node.

In practice, steps are usually written using a shorthand syntax. Because the «`child`» axis is the one that's used most often, the prefix «`child::`» can be omitted. Attributes are also used frequently, so the prefix «`attribute::`» can be shortened to the mnemonic «`@`» sign. This means the path given in full above can be abbreviated to «`book/chapter/@status`».

The other common abbreviation is «`//`», which you can think of as searching the entire subtree below the starting node. For example, «`//figure`» selects all the `<figure>` elements in the current document. A more precise definition of this construct (and the others) is given in Chapter 7.

Any step in a path expression can also be qualified by a predicate, which filters the selected nodes. For example, «`book/chapter[@ed="John"]/@status`» returns the «`status`» attribute of all the chapters of all the books provided that the chapter has an attribute named «`ed`» whose value is «`John`».

Path expressions thus provide a very powerful mechanism for selecting nodes within an XML document, and this power lies at the heart of the XPath language.

Composability

As we've seen, the introduction to the XPath specification describes the language as an expression language, and this has some implications that are worth drawing out.

Expressions can be nested. In principle, any expression can be used in a position where a value is allowed. This theoretical freedom is slightly restricted by two factors: at a trivial level, you might have to enclose the expression in parentheses; more seriously, you can only use an expression that returns the right type of value. For example, you can't use an expression that returns an integer in a place where a sequence of nodes is expected. XPath values have a type, and the language has rules about the types of expressions. The type system of the language is unusual, because it is closely integrated with XML Schema. I shall have more to say about the type system in Chapter 3, and I will describe the language constructs that relate specifically to types in Chapter 9.

A language in which you can nest expressions in arbitrary ways is often referred to as being *composable*. Composability is regarded as a good principle of modern language design. Most languages have some restrictions on composability, for example in Java you can use an array initializer (a construct of the form «`{1,2,3}`») on the right-hand side of the «=» in a variable declaration, but you can't use it as an arbitrary expression. XPath has tried hard to avoid including any such restrictions—even to the extent of allowing you to do things that no one would ever want to do, like writing «`--1`», whose value is +1.

Closely allied with the idea of composability is the principle of *closure*. This term comes from mathematics. A closed group consists of a set of possible values (for example, the positive integers) and a set of possible operations, in such a way that every operation when applied to these values produces a new value that is also within the same space of possible values. If you take the positive integers together with the operators of addition and multiplication you have a (not very useful) closed group, but if you also allow subtraction and division the group is no longer closed, because the results are not always positive integers. In XPath the set of possible values is defined by the data model: as we have already seen, this allows nodes, atomic values, and sequences. The set of possible operations is defined by the expressions in the language, and the whole system is closed, because the result of every expression is also a value within the scope of the data model. Closure is a necessary property to achieve composability, because you can't use the result of one expression as the input to another unless the result is in the same value space.

What's New in XPath 2.0?

XPath 2.0 represents a major advance on version 1.0: the number of operators has doubled, and the number of functions in the standard function library has grown by a factor of four or five depending on how you do the counting. The changes to the core syntax are not so dramatic, but the introduction of a new type system based on XML Schema represents a pretty radical overhaul of the language semantics. The W3C working groups have also tried to define the language much more rigorously than XPath 1.0 was defined, with the result that the number of trees used when you print the spec has grown astronomically.

It's easy to list the new features in version 2.0 as a simple catalog of goodies. What's harder to do is to stand back and make sense of the total picture: Where is the language going? I'll try to answer the

question posed in the section heading in both ways, first by listing the features, and then by trying to see if we can understand what it all means.

New Features in Version 2.0

Firstly, the XPath 2.0 data model offers new data types:

- XPath 1.0 had a single numeric data type (double precision floating point), XPath 2.0 offers in addition integers, decimals, and single precision
- There are new data types for dates, times, durations, and more
- It is also possible to exploit user-defined data types that are defined using XML Schema
- XPath 2.0 supports sequences as a data type. Sequences can contain nodes and/or atomic values. An important peculiarity of the XPath data model is that a singleton item such as an integer is indistinguishable from a sequence of length one containing that item.

The data model for representing XML documents has not actually changed very much, despite the fact that the description has grown from five pages to about 60. It still has the same seven kinds of node, namely document nodes (which were called root nodes in XPath 1.0), elements, attributes, text nodes, comments, processing instructions, and namespace nodes, and the relationship between them has not changed significantly. The main change is that element and attribute nodes can now have a type annotation. This is a label identifying the data type of the content of the element or attribute, which is determined by the definition of the element or attribute in the XML Schema that was used to validate the document. If the document has not been validated (which is still considered a perfectly respectable state of affairs) then the type annotation is set to one of the special values «`xdt:untyped`» for elements, or «`xdt:untypedAtomic`» for attributes.

Going hand-in-hand with the type annotation is the idea that an element or attribute node has a typed value: for example if the type annotation is «`xs:integer`», then the typed value will be an integer, while if the type annotation is «`xs:NMTOKENS`», then the typed value will be a sequence of «`xs:NMTOKEN`» values. Because the typed value is always a sequence of atomic values, the process of extracting the typed value of a node (which is performed implicitly by many XPath operations, for example equality comparison) is referred to as *atomization*.

Path expressions too have not changed very much since XPath 1.0. The biggest change is that the `NodeTest` (the part that follows the «*`axis`*`::`» if you write a step in full) can now test the type of the node as well as its name. For example, you can select all elements of type `Person`, regardless of the name of the element. This is very useful if you are using a schema with a rich type hierarchy in which many elements can be derived from the same type definition: many of the bigger and more complex XML vocabularies have this characteristic. It corresponds to the ability to use a generic supertype in an object programming language such as Java or C#, rather than having to list all the possible subtypes you are interested in.

Another significant change in path expressions is that you can use a function call in place of a step. This means that you can follow logical relationships in the XML document structure, not just physical relationships based on the element hierarchy. For example, if someone writes a function that finds all the orders for a customer, you can invoke this function in the middle of a path expression by writing «`customer[@id="123"]/find-orders(.)/order-value`». This means that the person writing this path expression doesn't necessarily need to know how the orders for a customer are found, and it means that the way that they are found can change without invalidating the expression. XPath itself does not

allow you to write the `find-orders()` function—you can do this in either XQuery or XSLT, or perhaps in other languages in the future. Functions written in XQuery or XSLT can be invoked from anywhere within an XPath expression.

Outside the realm of path expressions, there's a raft of new operators in the language. These include:

- Operators «`is`», «`<<`», «`>>`» to test whether two expressions return the same node, or to test which of the two nodes is first in document order
- Operators «`intersect`» and «`except`» to find the intersection or difference between two sets of nodes
- Operators «`eq`», «`ne`», «`lt`», «`le`», «`gt`», «`ge`» to compare atomic values. These are provided alongside the XPath 1.0 operators «`=`», «`!=`», «`<`», «`<=`», «`>`», «`>=`» which allow sequences of values to be compared
- An integer division operator «`idiv`»
- An operator «`to`» which allows you to construct a range of integers, for example, «`1 to 10`».

The most important new syntactic constructs are:

- The «`for`» expression, which as we have already seen on page 8 is used to apply the same expression to every item in a sequence.
- The «`if`» conditional expression. For example, the expression «`if (@price > 10) then "high" else "low"`» returns one of the two strings `"high"` or `"low"` depending on the value of the «`price`» attribute.
- The «`some`» and «`every`» expressions. The expression «`some $p in $products satisfies (every $o in $p/orders satisfies $o/value > 100)`» returns true if there is at least one product all of whose orders are worth more than $100.

The function library has grown so much that it's hard to know where to begin. A full specification of all the functions is included in Chapter 10. The main highlights are:

- There are many new functions for handling strings, for example, to perform case conversion, to join a sequence of strings, and an `ends-with()` function to complement the XPath 1.0 `starts-with()`.
- In particular, there are three functions `matches()`, `replace()`, and `tokenize()` that bring the power of regular expressions into the XPath language, greatly increasing its string-manipulation capabilities.
- All functions that perform comparison of strings can now use a user-specified collation to do the string comparison. This allows more intelligent localization of string matching to the conventions of different languages.
- There are new functions for aggregating sequences; specifically, `max()`, `min()`, and `avg()` are now available, alongside `sum()` and `count()` from XPath 1.0.
- There's a large collection of functions for manipulating dates and times.
- There are new functions for manipulating QNames and URIs.

Now let's try to stand back from the trees and examine the wood.

A Strategic View of the Changes

Is there any kind of unifying theme to these new features?

To find out, it helps to look back at the original requirements specification for XPath 2.0, which can be found at `http://www.w3.org/TR/2001/WD-xpath20req-20010214` (there is also a later version, which describes how the requirements were met in the actual language design). It starts, very briefly, with a summary of the goals of the new version:

- Simplify manipulation of XML Schema-typed content
- Simplify manipulation of string content
- Support related XML standards
- Improve ease of use
- Improve interoperability
- Improve i18n support
- Maintain backward compatibility
- Enable improved processor efficiency

After this disappointingly brief introduction, it then launches into what is, frankly, a catalog of desired features rather than a true requirements statement (it never attempts to answer the question *Why is this needed?*). But the goals, and the way the detailed requirements are written, do give some clues as to what the working groups were collectively thinking about.

We'll talk more about the process by which XPath 2.0 was defined later in the chapter. For the moment, it's enough to note that it was produced jointly by two working groups: the XSL Working Group, who were responsible for XSLT and had produced the XPath 1.0 specification, and the XQuery Working Group who were interested in extending XPath to make it suitable as a query language for XML databases. This requirements statement, produced on St. Valentine's Day 2001, was the first fruit of the collaboration between the two groups. The thinking of the two groups at this stage had not converged, and if you read the document carefully, you can detect some of the tensions.

Let's try and read between the lines of the eight goals listed above. The ordering of the goals, incidentally, was probably not debated at length, but I think it is important psychologically as an indication of the relative priorities which some members at least attached to the various goals.

- *Simplify manipulation of XML Schema-typed content.* We've already seen that the introduction of a type system based on XML Schema is probably the most radical change in XPath 2.0. At this time, early in 2001, XML Schema was seen as absolutely central to W3C's future architectural direction. It was also central to the plans of many of W3C's member companies, such as Microsoft, Oracle, and IBM. Although James Clark (the designer of XSLT and XPath 1.0) was starting to make discontented noises about the technical qualities of the XML Schema specification, no one in the establishment really wanted to know. Everyone wanted XML Schema to be a success and was confident that it would indeed be a success, and it was self-evident that languages such as XPath for manipulating XML documents should take advantage of it.

  A great deal of the requirements document is given over to outline ideas of how the language might integrate with XML Schema. Looking at it now, it reads much more like a design sketch than a true requirements list.

- *Simplify manipulation of string content.* It was generally agreed that the facilities in XPath 1.0 for manipulating strings were too weak. Facilities were needed for matching strings using regular expressions, for changing strings to upper and lower case, and so on.
- *Support related XML standards.* This appears as a catch-all in the list of goals, but it reflects the fact that W3C specifications are not produced in isolation from each other. The different working groups spend a lot of time trying to ensure that their efforts are coordinated and that all the specifications work well together.

  The actual requirements listed in the body of the document under the heading *Must support the XML Family of Standards* actually form a very motley collection, and some of them bear no relationship to this heading at all. The requirements that do make some sense in this category relate to the need to support common underlying semantics for XSLT 2.0 and XQuery 1.0, the need for a data model based on the InfoSet published by the XML Core Working Group (more on this on page 16), and the need for backward compatibility with XPath 1.0. Interestingly, this last requirement is classified as a *should* rather than as a *must*, which meant that backward compatibility could be sacrificed to meet other objectives.
- *Improve ease of use.* This heading was clearly seen as an open invitation for everyone to add their favorite features. So in this category we see things such as the need to add a conditional expression, the need to generalize path expressions, and the need for new string functions and aggregation functions. More fundamentally, there is also a subsection calling for consistent implicit semantics for operations that handle collections, and criticizing some of the design choices made in XPath 1.0 such as the way the «=» operator was defined over sets of nodes. Although these were described as *must* requirements, there was clearly no way of satisfying them without radical change to the language semantics, which would have had a devastating effect on backward compatibility. In the end, much of the debate of the next two years was spent finding an acceptable compromise to this problem.

  One might imagine that a gathering of some of the brightest minds in the computer industry would not write "improve ease of use" as a goal without defining some way of measuring the ease of use of the language before and after the addition of these features. Sadly, one would be disappointed. A committee can do mindless things, regardless how bright the minds are that make it up.
- *Improve interoperability.* The word *interoperability* in W3C circles means the ability for different implementations of a specification to produce the same result.

  I can't actually find any detailed requirements that support this goal, so it should be no surprise that XPath 2.0 actually allows a lot more freedom to implementers to introduce differences than XPath 1.0 did.
- *Improve i18n support.* Here i18n is shorthand for internationalization, the ability of the specification to support the needs of different languages and cultures worldwide. This is something the W3C takes fairly seriously (despite requiring editors of its specifications to write in English, with American spelling).

  Again there is actually nothing concrete in the requirements to support this goal. The main new feature in XPath 2.0 that affects internationalization is the introduction of user-selected collations to support string comparison and sorting, but this feature does not actually appear explicitly in the requirements (instead, it found its way into XPath via the XSLT and XQuery requirements).

  Two features that are notably lacking from XPath 2.0 are support for localized formatting of numbers and dates. For this, you need to turn to the additional function library provided by XSLT 2.0, which is available only when you use XPath expressions within an XSLT stylesheet.

- *Maintain backward compatibility.* As I've already mentioned, this appears as a *should* rather than a *must*. This has been a tension throughout the development of the language, as some XQuery people felt they wanted to be unconstrained by the past, whereas XSLT representatives felt a strong responsibility to their existing user base.

  In the end, each decision was made on its merits. Incompatible changes were introduced only when the group as a whole felt that the gain was worth the pain. Some incompatibilities were inevitable, given the change in the data model and type system, but by and large gratuitous incompatibilities were avoided. Some of the worst conflicts were resolved by the introduction of the ability to run in backward compatibility mode (the infamous "mode bit", for those who have read Tracy Kidder's *Soul of a New Machine*). In many cases, the XPath 1.0 way of doing things was eventually retained because people came to see that it wasn't such a bad design after all.

- *Enable improved processor efficiency.* Once again, there is nothing in any of the requirements that explains how it is intended to contribute to this goal (and of course, there is again no measure of success). I think one could go through the requirements and explain how some of them might improve the performance of applications, but whether anyone actually was thinking this through at the time, I don't know.

  Efficiency has frequently come up during the design discussions on language features, sometimes for good reasons and sometimes for bad. For example, designs were often rejected if they inhibited pipelining, that is, the ability to process a sequence of values without retaining all the values in memory. An example of such a rule that was present at one time was that the `max()` of a sequence should be the numeric maximum if all items in the sequence were numbers, or the string maximum otherwise. This means you need to read the whole sequence before you can compare the first two values, so this rule was rightly rejected. Quite often, however, vendors would come to the working group and ask for a feature to be changed simply because it looked difficult to implement (I've been guilty of this myself). Usually other implementors would squash such arguments, pointing out that alternative techniques were available. In general, a language design that is clean and simple from a user perspective turns out to be a better choice than one that developers find easy to implement.

One design decision that doesn't emerge clearly from this study of the requirements is the question of how big the language should be. There were (and are) differing views on this, and there is no obvious right answer that suits everyone. Some people wanted the language to be much smaller than it is, others to accommodate some of the XQuery features that have been left out, such as full FLWOR expressions. The final outcome is a compromise, but it is a compromise that has some rationale: in particular, the language includes sufficient power to make it *relationally complete*, as defined by E. F. Codd in the theory of the relational model. There is a mathematical definition of this term, and achieving this property gives reasonable confidence that the language will be powerful enough for most data retrieval tasks. However, this doesn't provide an absolute criterion for what should be included: for example, the decision not to include any sorting capability in XPath 2.0 could have gone either way.

I hope this summary gives a little bit of a feel of what the working groups were trying to achieve with the design of XPath 2.0. If it seems like something half-way between a carefully-thought out strategy and an ad hoc ragbag, then that's probably because it is. That's the way committees work.

XPath 2.0 Processors

At the time of writing there are three ways you can actually use XPath 2.0.

Firstly, you can use an XQuery 1.0 processor. There are quite a few XQuery processors available, and you can find them listed on the W3C home page for XQuery at `http://www.w3.org/XML/Query`. Since XPath 2.0 is a subset of XQuery 1.0, you can use any XQuery processor to execute XPath expressions. They vary considerably in the extent to which they implement the full specifications, and in how up-to-date they are with the latest drafts of the specifications. I haven't tried them all, and the situation changes from month to month, so I won't recommend any implementation in particular. One of the processors listed on the XQuery home page is my own Saxon implementation—Saxon is both an XSLT and an XQuery processor, with a common runtime engine supporting both languages, and this runtime also, of course, supports XPath 2.0.

Broadly speaking, the suppliers of these XQuery processors have either concentrated on building an industrial-strength XML database product, or they have concentrated on tracking the latest language standards. The more advanced a product is in terms of the language specification that it supports, the less advanced it is likely to be in terms of other database features such as updates, recovery, transactions, fast database loading capabilities, and so on.

One product that I have found very easy to install and use is IPSI-XQ, developed by the Fraunhofer Institut in Germany. An advantage that it has over Saxon is that it has a graphical user interface (Saxon can only be driven from the command line or from a Java API, which isn't very appealing when you're showing it off in a conference). You can get IPSI-XQ from `http://www.ipsi.fraunhofer.de/oasys/projects/ipsi-xq/index_e.html`. Figure 1-1 shows a simple query.

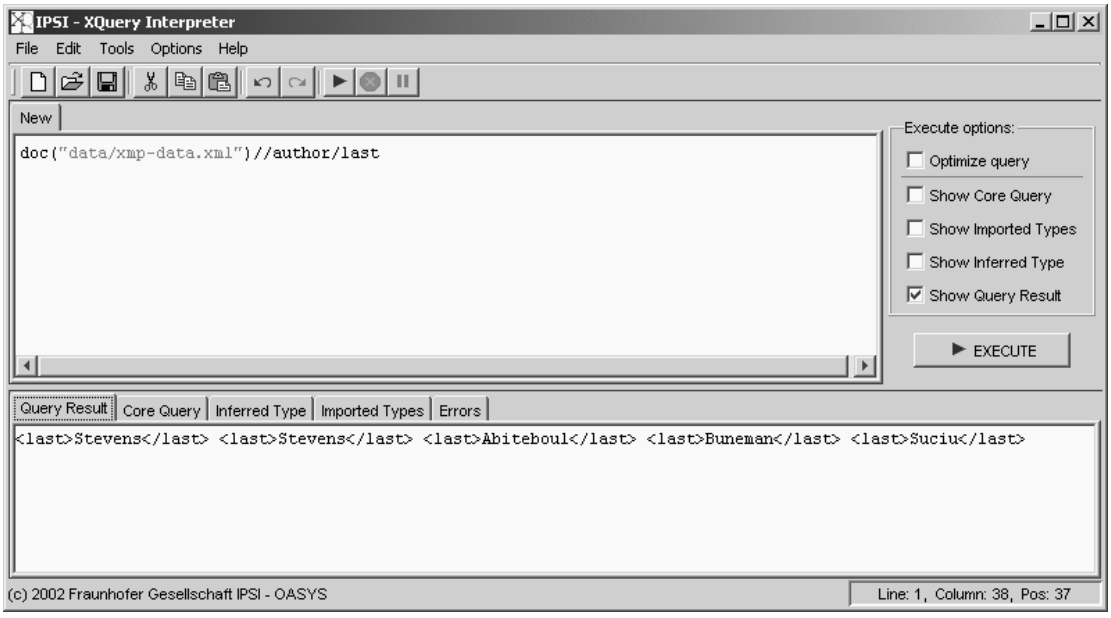

Figure 1-1

Usually when you use XPath 2.0, the program will be launched to process some particular source document. XQuery processors, however, are designed primarily to process multiple documents, so there is often no initial context for your expression. In this example the `doc()` function (which is described in Chapter 10 of this book) is used to select the document that you want to process, and everything else is

selected within that document. The document selected in this query is found in the `examples` subfolder of the directory where IPSI-XQ is installed.

By comparison, there are relatively few pure XPath 2.0 products. The reason for this is that XPath isn't usually used on its own: it has always been designed as a specialized sublanguage that's intended for use in some kind of host environment.

One product that does include an XPath 2.0 processor is XML Spy. This is a commercial product, but you can get a free evaluation license for a limited period. You can download the code from `http://www.altova.com/`. Altova offers many different product configurations, so check that the one you are using includes the XPath 2.0 support.

To use the XPath 2.0 Analyzer within XML Spy, first load the document that you want to analyze. Then open up the XPath window, which you can do by selecting XML, then Evaluate XPath, from the menu. The default is to execute XPath 1.0: select the radio button labeled XPath 2.0 beta (by the time you read this, it may no longer be a beta release, of course).

You can then enter an XPath expression to run against the loaded document. The document I loaded was the text of Shakespeare's *Macbeth* in `macbeth.xml`, and my first attempt was the query «`distinct-values(//SPEAKER)`», which returned a somewhat alarming error message: "unexpected argument type, found First Witch". It turned out that this was because the `distinct-values()` function in the version of XML Spy I was using wasn't quite up-to-date with the latest version of the specs: this is a recurrent problem with all these products and will remain so until the final versions of the specifications are released. It appears that in XML Spy this query has to be written as shown in Figure 1-2.

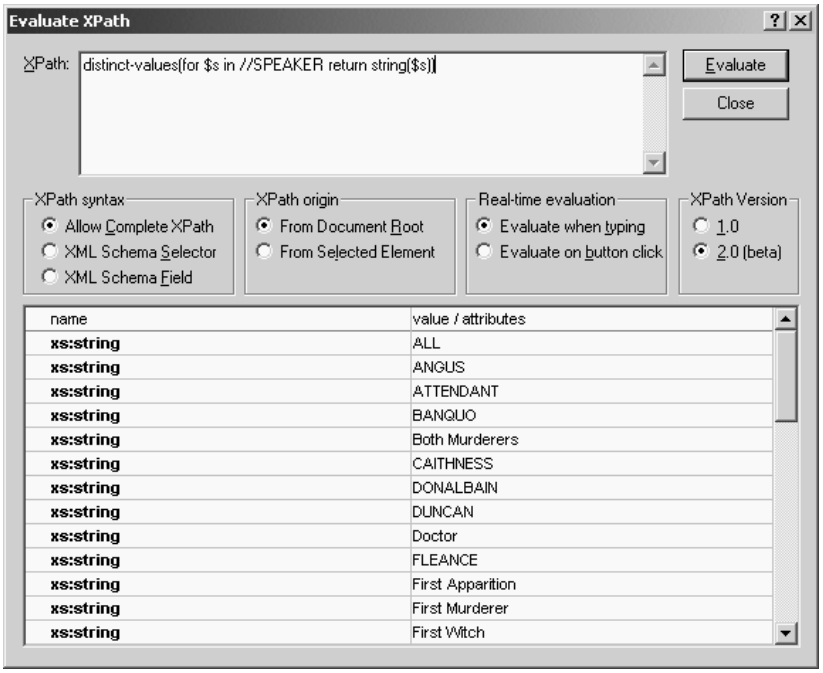

Figure 1-2

This expression uses the `distinct-values()` function, one of the many new functions available in XPath 2.0, to return a sequence that contains all the values appearing in a `<SPEAKER>` element anywhere in the document, with duplicate values removed. XML Spy appears to sort the results in alphabetical order, but the specification says that the order of the results is up to the implementation to decide.

Figure 1-3 shows another example using XML Spy. This one finds all the lines in the play containing the word "spot".

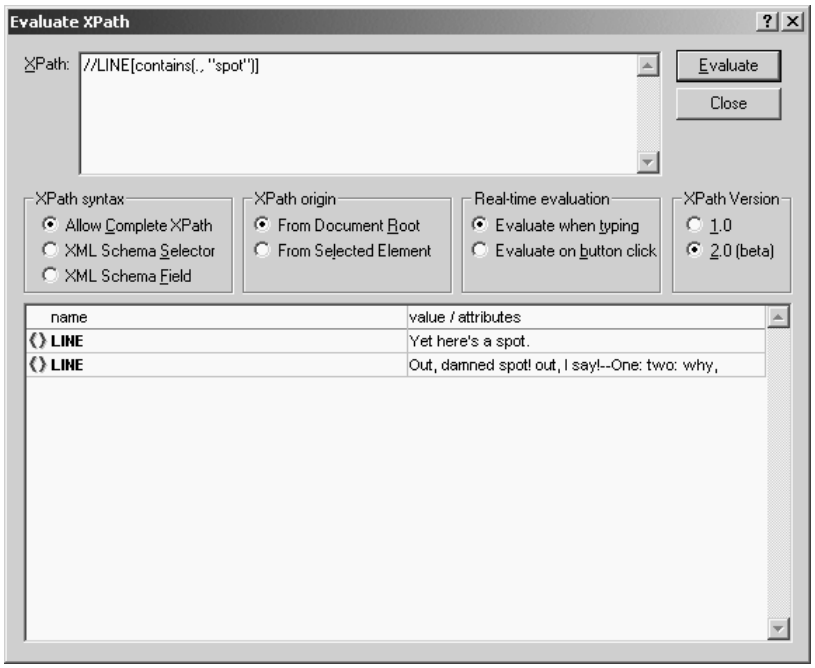

Figure 1-3

Since the result of every XPath expression is a sequence, it's not easy to get any formatted results out of this tool: you can't produce tabular output, for example, and you can't generate XML. If you select an element, XML Spy shows you the text that's directly contained in that element, even if there isn't any. No doubt the tool will improve a lot in the months to come: this is a very early beta.

Another product that includes an XPath 2.0 processor in its latest release is Stylus Studio (`http://www.StylusStudio.com`). I ran the same expression «`distinct-values(//SPEAKER)`» using Stylus Studio 5.1, with the results shown in Figure 1-4. Note that to enable XPath 2.0 support you need to click the button labeled v.2: by default, the product uses XPath 1.0. This version of Stylus Studio claims to support the XPath 2.0 working draft of November 2003.

This product shows the results of the `distinct-values()` function in order of first appearance: which works well for the speakers in a play, but might not always be the most appropriate choice.

In this particular example the results of the queries are strings. But if you enter an expression whose results are nodes in the source document, for example, the expression «`//SPEAKER[.="HECATE"]`»,

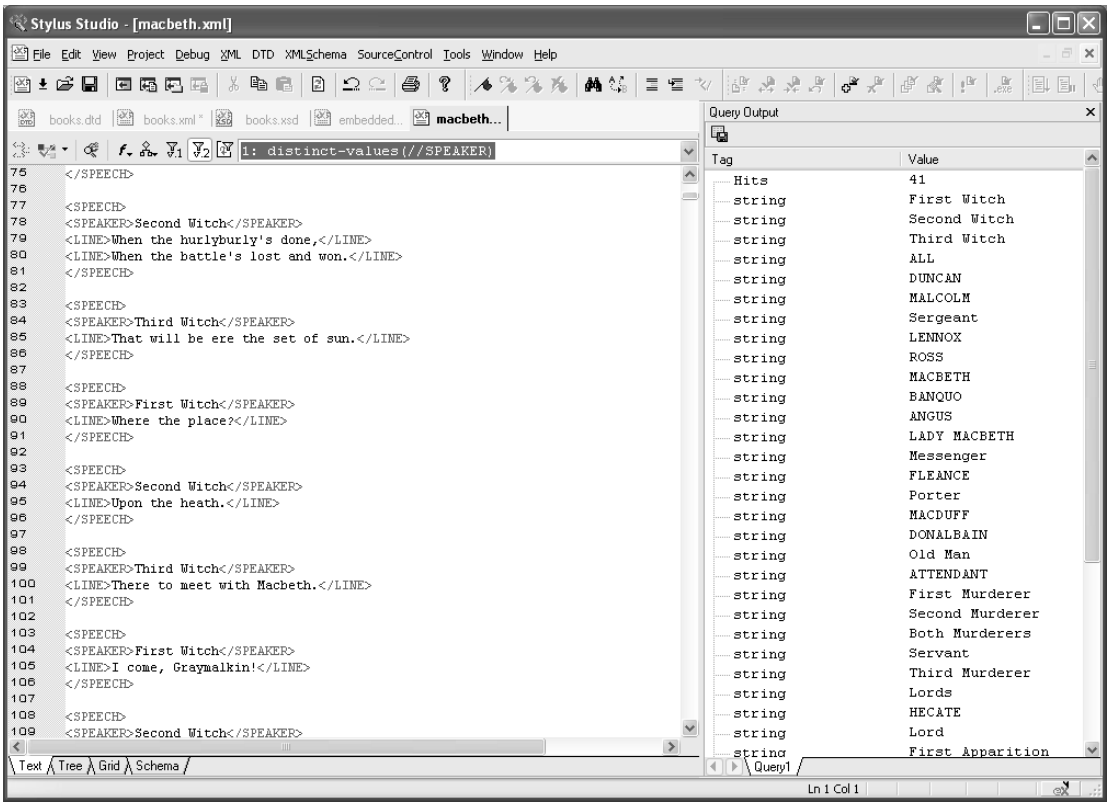

Figure 1-4

then the results will be shown in the right-hand pane as links, allowing you to click any node in the query results to locate the relevant element in the main editing window.

A tool that gives a much more graphic impression of how XPath works is the XPath Visualizer obtainable from `http://www.vbxml.com/xpathvisualizer/`. Unfortunately, however, this only supports XPath 1.0 at the time of writing. The tool works directly with Internet Explorer and its inbuilt XPath engine. To use it, unzip the download file into a suitable directory, and open the file `XPathMain.htm`. Then, browse to the source document you want to analyze, and click Process File. You can now enter XPath expressions, and see the nodes you have selected highlighted on the screen (use the arrow buttons to scroll to the next highlighted node). Figure 1-5 shows the same XPath expression as shown by XPath Visualizer.

One further XPath 2.0 implementation I have come across is Pathan 2: see `http://software.decisionsoft.com/pathanIntro.html`. This currently describes itself as an alpha release. It is a no-frills open-source implementation, that offers XPath only, and is designed as a component for integration into applications and tools.

The other way to use XPath 2.0 is from within XSLT 2.0. Currently, there are few XSLT 2.0 processors available: there is my own Saxon product (look for the latest release at `http://saxon.sf.net/`), and

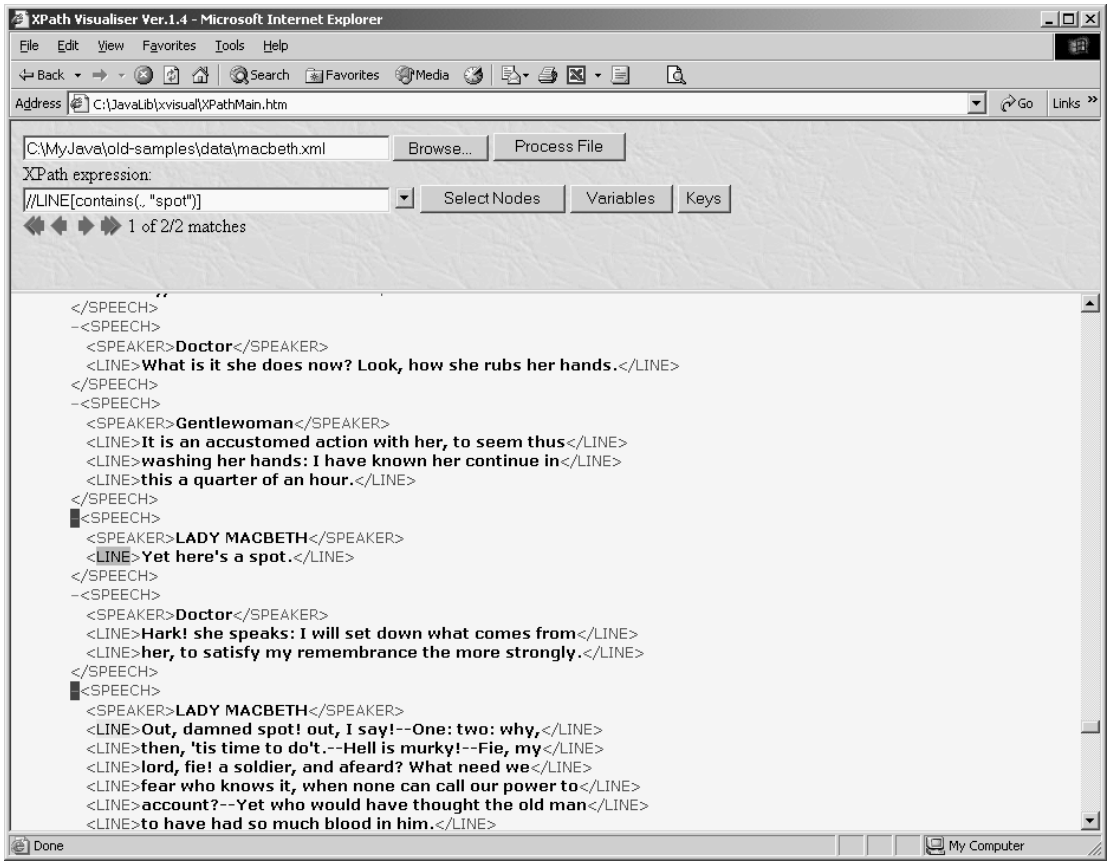

Figure 1-5

there is a beta release from Oracle, which has not been updated for some time (no doubt Oracle are waiting for the specifications to stabilize before they make their next shipment). Other processors are known to be under development, so keep an eye open for news. My companion book *XSLT 2.0 Programmer's Reference* explains the use of XSLT 2.0 in great detail, and XPath 2.0 plays a significant role in this.

Here's an example of a complete XSLT 2.0 stylesheet that uses XPath 2.0 features to get a count of all the words appearing in a document (perhaps the text of *Macbeth*), together with the frequency of each word:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<xsl:stylesheet
   version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="xml" indent="yes"/>

<xsl:template match="/">
  <wordcount>
```

```
      <xsl:for-each-group group-by="." select="
            for $w in tokenize(string(.), '\W+') return lower-case($w)">
        <xsl:sort select="count(current-group())" order="descending"/>
        <word word="{current-grouping-key()}"
              frequency="{count(current-group())}"/>
      </xsl:for-each-group>
    </wordcount>
  </xsl:template>

  </xsl:stylesheet>
```

You can run this using the Saxon XSLT processor with a command such as:

```
java -jar c:\saxon\saxon8.jar macbeth.xml wordcount.xsl
```

to produce output which starts like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<wordcount>
   <word word="the" frequency="735"/>
   <word word="and" frequency="567"/>
   <word word="to" frequency="405"/>
   <word word="i" frequency="373"/>
   <word word="of" frequency="348"/>
   <word word="macbeth" frequency="291"/>
   <word word="a" frequency="255"/>
   <word word="that" frequency="237"/>
   <word word="in" frequency="208"/>
   <word word="you" frequency="207"/>
   <word word="s" frequency="199"/>
   <word word="is" frequency="192"/>
   <word word="my" frequency="192"/>
```

This stylesheet uses a number of new XSLT 2.0 features, notably the `<xsl:for-each-group>` instruction which provides a long-awaited grouping facility for XSLT. It also makes heavy use of new constructs in XPath 2.0: the `tokenize()` function, which splits a string using a regular expression, the `lower-case()` function which converts text to lower case, and a «`for`» expression which applies the `lower-case()` function to every string in a sequence of strings.

Where XPath fits in the XML Family

XPath 2.0 is published by the World Wide Web Consortium (W3C) and fits into the XML family of standards, most of which are also developed by W3C. In this section, I will try to explain the relationship of XPath 2.0 to other standards and specifications in the XML family.

XPath and XSLT

XSLT started life as part of a bigger language called *XSL* **(***Extensible Stylesheet Language***)**. As the name implies, XSL was (and is) intended to define the formatting and presentation of XML documents for display on screen, on paper, or in the spoken word. As the development of XSL proceeded, it became clear

that this was usually a two-stage process; first a structural transformation, in which elements are selected, grouped and reordered, and then a formatting process in which the resulting elements are rendered as ink on paper, or pixels on the screen. It was recognized that these two stages were quite independent, so XSL was split into two parts, XSLT for defining transformations, and "the rest"—which is still officially called XSL, though most people prefer to call it XSL-FO *(XSL Formatting Objects)*—for the formatting stage.

Halfway through the development of XSLT 1.0, it was recognized that there was a significant overlap between the expression syntax in XSLT for selecting parts of a document, and the XPointer language being developed for linking from one document to another. To avoid having two separate but overlapping expression languages, the two committees decided to join forces and define a single language, XPath, which would serve both purposes. XPath 1.0 was published on the same day as XSLT 1.0, November 16, 1999.

XPath acts as a sublanguage within an XSLT stylesheet. An XPath expression may be used for numerical calculations or string manipulations, or for testing boolean conditions, but its most characteristic use is to identify parts of the input document to be processed. For example, the following instruction outputs the average price of all the books in the input document:

```
<xsl:value-of select="avg(//book/@price)"/>
```

Here the `<xsl:value-of>` element is an instruction defined in the XSLT standard, which causes a value to be written to the output document. The `select` attribute contains an XPath expression, which calculates the value to be written: specifically, the average value of the `price` attributes on all the `<book>` elements. (The `avg()` function is new in XPath 2.0.)

As this example shows, the XSLT and XPath languages are very intimately related.

In previous editions of this book I covered both languages together, but this time I have given each language its own volume, mainly because the amount of material had become too large for one book, but also because there are an increasing number of people who use XPath without also using XSLT. For the XSLT user, though, I'm afraid that at times you may have to keep both books open on your desk at once.

XPath and the InfoSet

XPath is primarily a language for extracting information from XML documents. However, the effect of an XPath expression isn't defined directly in terms of the lexical XML structure. XPath uses an abstraction of an XML document that consists, as we have seen, of a tree containing seven different kinds of nodes. This model itself is defined in terms of the XML Information Set, usually called the InfoSet, for short. In this section we'll take a quick look at the relationship of XPath to the InfoSet: we will return to this in more detail in Chapter 2.

XPath is designed to work on the information carried by an XML document, not on the raw document itself. The tree model is an abstraction of the original lexical XML, in which information that's deemed significant is retained, and other information is discarded. For example, you can see the attribute names and values, but you can't see whether the attribute was written in single or double quotes, you can't see what order the attributes were in, and you can't tell whether or not they were written on the same line.

One messy detail is that there have been many attempts to define exactly what constitutes the essential information content of a well-formed XML document, as distinct from its accidental punctuation. All

attempts so far have come up with slightly different answers. The *XML Information Set* definition (or InfoSet), which may be found at `http://www.w3.org/TR/xml-infoset`, is the most definitive attempt to provide a common vocabulary for the content of XML documents.

Unfortunately, the InfoSet came too late to make all the standards consistent. For example, some treat comments as significant, others not; some treat the choice of namespace prefixes as significant, others take them as irrelevant. I shall describe in Chapter 2 exactly how XPath defines the tree model of XML, and how it differs in finer points of detail from some of the other definitions such as the Document Object Model or DOM.

Another more elaborate model of the information content of an XML document is the *post schema validation infoset* or PSVI. This contains the significant information from the source document, augmented with information taken from its XML Schema. It identifies which types in the schema were used to validate each element and attribute, and as such it underpins the notion of nodes having a type annotation and a typed value, which as we saw earlier are the two most important changes in the data model for XML documents introduced in version 2.0.

XML Namespaces

As far as XPath is concerned *XML namespaces* are an essential part of the XML standard. If a document doesn't conform with the XML Namespaces Recommendation, then you can't use it with XPath. This doesn't mean that the document actually has to use namespaces, but it does mean that it can't misuse them.

Here's a quick reminder of how namespaces work:

- Namespaces are identified by a Uniform Resource Identifier (URI). This can take a number of forms. The most common form is the familiar URL, for example `http://www.wrox.com/namespace`. The detailed form of the URI doesn't matter, but it is a good idea to choose one that will be unique. One good way of achieving this is to use the URL of your own Web site. But don't let this confuse you into thinking that there must be something on the Web site for the URL to point to. The namespace URI is simply a string that you have chosen to be different from other people's namespace URIs; it doesn't need to point to anything.
- The latest version of the spec, XML Namespaces 1.1, allows you to use an International Resource Identifier (IRI) rather than a URI. The main difference is that an IRI permits characters from any alphabet, whereas a URI is confined to ASCII. In practice, most XML parsers have always allowed you to use any characters you like in a namespace URI.
- Since namespace URIs are often rather long and use special characters such as «/», they are not used in full as part of the element and attribute names. Instead, each namespace used in a document can be given a short nickname, and this nickname is used as a prefix of the element and attribute names. It doesn't matter what prefix you choose, because the real name of the element or attribute is determined only by its namespace URI and its local name (the part of the name after the prefix). For example, all my examples use the prefix `xsl` to refer to the namespace URI `http://www.w3.org/1999/XSL/Transform`, but you could equally well use the prefix `xslt`, so long as you use it consistently.
- For element names, you can also declare a default namespace URI, which is to be associated with unprefixed element names. The default namespace URI, however, does not apply to unprefixed attribute names.

A namespace prefix is declared using a special pseudo.attribute within any element tag, with the form:

```
xmlns:prefix = "namespace-URI"
```

This namespace prefix can be used for the name of that element, for its attributes, and for any element or attribute name contained in that element. The default namespace, which is used for elements having no prefix (but not for attributes), is similarly declared using a pseudo-attribute:

```
xmlns = "namespace-URI"
```

It's important to remember when using XPath that the true name of an element is the combination of its local name (the part after any prefix) and the namespace URI. For example the name of the element `<a xmlns="http://ns.example.com/"/>` is the combination of the local name «`a`» and the namespace «`http://ns.example.com/`». A step in a path expression will only match this element if both the local name and the namespace URI match. The XPath expression doesn't have to use the same prefix as the element that's being matched, but the prefixes do have to refer to the same URI.

XML Namespaces 1.1 became a Recommendation on February 4, 2004, and XPath 2.0 can in principle work with either 1.0 or 1.1. Apart from the largely cosmetic change from URIs to IRIs mentioned earlier, the main innovation is the ability to undeclare a namespace, using a namespace undeclaration of the form «`xmlns:prefix=""`». This new feature doesn't have a great deal of effect on XPath itself, though it does create complications for XSLT and XQuery, which unlike XPath have instructions to create new nodes.

XPath and XPointer

One of the original reasons that XPath was defined in its own W3C specification, rather than as part of XSLT, was so that it could be used independently by the XPointer specification. Subsequently, XPointer has had a checkered history.

The intended role of XPointer is to define the syntax of fragment identifiers in URIs (that is, the part of the URI after the «#» sign) when referencing into the detail of an XML document. The theory is that the syntax of a URI fragment identifier depends on the media type of the resource identified by the URI. With the familiar HTML URIs, the fragment identifier is the value of an «`id`» or «`name`» attribute of an HTML element within the document. For XML, people wanted something more powerful. In particular, the hyperlinking community wanted to be able to reference into an XML document without requiring the document author to do anything special (like creating uniquely named anchors) to make this possible. XLink, the hyperlinking standard, and XPointer, which it used for defining cross-references, were closely coupled in everyone's minds. Hence the use of XPath as a general-purpose, powerful addressing mechanism.

XPointer remained work-in-progress for a long time. I don't know all the reasons for this, but I suspect that the main underlying cause was that it was too ambitious. In addition, I think that the success of the XSLT model for publishing content on the Web rather took the wind out of the sails of XLink. XSLT allows you to model your information in XML in any way that you like, and then convert it to HTML for presentation on the Web. If you are going to end up generating HTML as your presentation format, then it doesn't make much difference whether your XML represents inter-document relationships using XLink or in some other way. In fact, there's a strong argument for modeling relationships using tags that represent the meaning of the relationship, just as you do when modeling business objects and their properties. If an employee is represented by an `<employee>` element, and a department by a `<department>` element,

then most people will choose to represent the relationship from an employee to a department using an element or attribute called «`department`», not by one called «`xlink:href`».

XLink managed to decouple itself from XPointer, and became a Recommendation on June 27, 2001. At that time, XPointer was a Last Call Working Draft, published six months earlier. The spec had retreated from Candidate Recommendation status because of technical problems with namespaces. An XPath expression contains element and attribute names such as «`mf:product`» that use namespace prefixes. The question is, where are these prefixes defined? Does it use the prefixes defined in the source document, or those defined in the target document? What happens if the XPointer is used in a free-standing URI reference, that isn't itself part of an XML document? This problem was discovered late in the day, and the language designers responded to this problem by going back to the drawing board, and devising an extension to XPointer that allowed the namespace prefixes to be defined within the XPointer itself.

Perhaps more importantly, XPointer hit a serious political problem in that it appeared to use a technique that Sun had patented. Sun proposed terms and conditions under which they agreed to license this patent, but the terms and conditions were not acceptable to everyone and this led to a fierce debate on patent policy in the W3C which overshadowed the original technical issues.

In the end, the XPointer specification was refactored into a number of separate modules. There is a framework specification which allows the definition of an extensible number of referencing schemes. The most basic scheme is to identify an element by its ID value, just as with an HTML fragment identifier. The next refinement is the `element()` scheme, which adds the ability to use hierarchic references: for example «`element(/1/3)`» refers to the third child of the first child of the root node. The full `xpointer()` scheme, which contains the original XPath-based syntax, has been languishing at Working Draft status since 2002 (see `http://www.w3.org/TR/xptr-xpointer/`) and since there is no longer an active Linking Working Group, it seems unlikely to progress further.

XPath and XQuery

XPath 2.0 is defined as a subset of XQuery 1.0; or to put it another way, XQuery has been designed as an extension of XPath 2.0. Unlike the embedding of XPath in XSLT, where there are two distinct languages with one invoking the other, XQuery is designed as a single integrated language that incorporates XPath as a subset. What this means in practice is that in XQuery (unlike XSLT), you can freely nest XQuery expressions and XPath expressions. For example, you can use an element constructor inside a path expression, like this:

```
<lookup><data/><data/></lookup>//data[code=$value]
```

XQuery is therefore more composable than XSLT (composability is the ability to construct any expression by combining simpler expressions), but the price it pays for this is that the syntax is not pure XML.

Although XQuery uses XPath as a subset, the XQuery specification doesn't actually refer to the XPath specification; rather it bodily copies the text of the XPath specification as part of the XQuery specification (this is all done, of course, by maintaining a single XML master document, with XSLT stylesheets used to generate the XPath and XQuery versions of the specification). The reason XQuery copies the XPath specification rather than referencing it is because XQuery doesn't simply allow you to use legal XPath expressions as subexpressions in your query, it also allows you to use XQuery expressions as operands to XPath constructs, thus changing the scope of the XPath language.

Chapter 1

XPath and XML Schemas

As we have already seen, integration with XML Schema was listed as the first of the goals in the XPath 2.0 requirements document. Achieving this integration has created an enormous upheaval in the language semantics, and although this is something that will be a hidden change below the surface for many users, it does actually have a profound impact.

XPath 1.0 was a weakly typed language. It had types, but it had very few of them, and very few rules about what operations were applicable to what types of data. The general model was that if you supplied an integer where (say) a string was expected, the integer would be quietly converted to a string. Another example of a weakly typed language is JavaScript.

The conventional wisdom for database query languages is that they should be strongly typed. A strongly typed language has lots of rules about how you can use values, depending on their type. Many programming languages, such as Java, for example, are also strongly typed.

Although weak typing appears to be more user-friendly, it has many disadvantages. One of the main disadvantages is that the processor isn't able to do so much work at compile time, instead it has to make most of the decisions at runtime. One of the key roles of a database query language is to identify, while compiling the query, which indexes can be used to execute the query efficiently. Unlike optimization in a conventional language such as C or Java which might give you code that runs three or four times faster, optimization in a database query language can produce a thousand-fold improvement in speed, or more. Therefore, anything that can be done in the language design to give a query optimizer a better chance is considered a Good Thing.

Another difference is that in the weak typing world, the philosophy is generally to avoid runtime errors. XPath 1.0 works on the principle that if you ask a silly question, you get a silly answer. Ask it whether the string "apple" equals the boolean true, and the answer is yes. (If you ask whether the string "false" equals the boolean false, the answer, more surprisingly, is no). By contrast the philosophy in the strong typing world is that if you ask a silly question you get an error message. There are a number of reasons why this might be considered preferable, but one of the reasons is that if you ask a silly question against a multi-gigabyte database, it can take many hours to come up with the silly answer.

So the influence of database query language thinking led to pressure for XPath to become more strongly typed. This was one of the factors driving the adoption of XML Schema. The XSLT group were also pushing in this direction, however, for rather different reasons. It was recognized that when document types are managed for a large community of users, managing the schema (or DTDs) for the documents and managing the stylesheets are two activities that need to be closely coordinated. When the schema changes (as it does, frequently) then the stylesheet needs to change too. There is a clear correspondence between declarations in the schema and rules in the stylesheet. Therefore, it was argued, there ought to be some linkage between the schema and the stylesheet to make it easier to keep the two in sync, and to report errors if they were out of sync.

A move towards stronger typing didn't have to mean support for XML Schema, but given the way working groups in W3C review each others' work and meet to reconcile their differences, it was almost inevitable.

One of the big challenges was to introduce schema-derived types in such a way that they were an optional feature, so that users who had no schema (and perhaps no wish for one) could carry on as they were. The result is a language that is in some ways a strange hybrid between strong typing and weak

typing. If this seems odd, it is worth reflecting that XML handles a vast spectrum from very highly structured data to very loosely structured documents, and that this ability to span the full range of information management requirements is one of its greatest strengths. So it shouldn't be surprising that XPath too is designed to handle a wide spectrum.

XPath, the DOM, and Java

The Document Object Model or DOM has origins that predate XML: it was originally the programming interface used to navigate your way around the objects on an HTML page, and was only later adapted so that it could handle XML as well. Later still, support for namespaces was bolted on. The DOM is a W3C specification that has grown considerably over the years. It is defined in a language-neutral way, but there are specific language bindings for a number of languages. Most users will be familiar either with the Microsoft implementation of the DOM, or with the Java language bindings.

Microsoft's MSXML product was probably the first to integrate an XPath processor into a DOM implementation. In fact, they did this before XPath was fully standardized, and in MSXML3 the default processor is still a non-standard variant of XPath (if you want real XPath, you have to ask for it specially, by setting the `selectionLanguage` property of the `Document` object to «`XPath`»).

The idea behind this interface is that instead of navigating your way laboriously to the required nodes in a DOM document using low-level methods such as `getFirstChild()`, `getNextSibling()`, `getAttribute()`, and so on, you should be able to select the set of nodes you want using a single call that supplies an XPath expression as an argument. In the Microsoft version of the interface, there are two methods that do this: `selectNode()` is used when you know that the XPath expression will select a single node in the tree, and `selectNodes()` is used when you want to select multiple nodes. The result is a `Node` or a `NodeList` respectively, which you can then manipulate using the normal DOM methods.

One of the drawbacks of this is that it doesn't allow you to use the full capability of XPath. If you want to count how many nodes satisfy a certain condition, for example, the only way to find out is to retrieve them all and then count them in the application. There is no way in this interface of invoking an XPath expression that returns a number, a string, or a boolean, as distinct from a node or a set of nodes.

Another limitation of the original Microsoft interface is that it doesn't allow you much control over the context of the XPath expression. For example, there is no way of supplying values of variables used in the expression. But set against these limitations, the API is delightfully simple to use.

Various Java implementations of the DOM also tried to provide XPath capabilities, many of them modeled directly on the Microsoft API, but some much more sophisticated. The Xalan XSLT processor, for example, provides an XPath API that works with the Xerces implementation of the DOM. This actually has two layers. The XPathAPI class provides a number of simple static methods such as `selectSingleNode()` and `selectNodeList()` which can be seen as parallels to the methods in Microsoft's API, though they are actually rather more complex (and therefore powerful). Underneath this is a much richer API that provides anything you could possibly want to control the execution of XPath statements, including, for example, the ability to compile expressions that can be evaluated later multiple times, with different settings for variables in the expression. This underlying API looks as if it was designed to provide the interface between the XSLT and XPath components of the Xalan product, so as you might expect it is a very rich and complex interface.

In the Java world there have been two serious attempts to provide alternatives to the DOM that are simpler, better integrated with Java, and more up-to-date in terms of XML specifications. One of these is JDOM (`www.jdom.org`), the other is called DOM4J (`www.dom4j.org`). Both have their merits and their band of enthusiastic followers. JDOM is pleasantly easy to use but has the major drawback that it uses concrete Java classes rather than interfaces, which means there is no scope for multiple implementations to coexist. DOM4J is a much richer API, which also means it is more complex. Both share the objective of being well integrated with the Java way of doing things, and both include XPath support as a standard part of the API.

A standard binding for XPath in the DOM came only with DOM level 3 (`http://www.w3.org/TR/DOM-Level-3-XPath/`), which became a W3C Candidate Recommendation in March 2003. (A Candidate Recommendation is rather like a beta release of software, which means that the specification is considered finished enough to ship, but might still have bugs that need fixing.) This is, of course, an interface to XPath 1.0 rather than XPath 2.0. Like the other DOM specifications, it includes an abstract interface defined in the CORBA IDL language, together with concrete interfaces for Java and JavaScript (or ECMAScript, to use its `its` official name). The Java interface treads a reasonable middle ground between simplicity and functionality: perhaps its most complex area is the way it delivers expression results of different types.

At the time of writing, there is an activity underway in the Java Community Process to define a standard Java API for XPath that will form part of JAXP 1.3 (JAXP is the Java API for XML Processing). Although this is not yet finalized, public previews have been made available (`http://jcp.org/aboutJava/communityprocess/review/jsr206/index.html`). This API does not simply endorse the DOM level 3 API. The main reasons for this decision appear to be that the designers wanted an API that was more Java-like (not just a Java binding of a CORBA IDL interface), that could be used with object models other than DOM, and that was easily extensible to handle XPath 2.0 in the future.

The net result of this is that there are quite a few different XPath APIs to choose from in Java. Hopefully, the JAXP 1.3 initiative will unify this, and will also succeed in its goal of being extensible to XPath 2.0, so that we end up with a single way of invoking XPath expressions from Java, that is independent of the choice of object model.

So much for the background and positioning of XPath 2.0. Let's look now at the essential characteristics of XPath 2.0 as a language.

XPath 2.0 as a Language

This section attempts to draw out some of the key features of the design of the XPath language.

The Syntax of XPath

The XPath syntax has some unusual features, which reflect the fact that it amalgamates ideas from a number of different sources.

One can identify three different syntactic styles within XPath expressions:

- *Conventional programming expressions*: This allows the same kind of expressions, infix operators, and function calls as many other programming languages; an example is an expression such as

«`$x + 1 = round($y) mod 3`». Such expressions trace their roots via programming languages such as Algol and Fortran back to the notations of elementary mathematics.

- *Path expressions*: These perform hierarchic selection of a node within a tree, an example is «`/a/b//c`». These expressions can be seen as a generalization of the syntax used by operating systems to identify files within a hierarchic filestore.
- *Predicate logic*: This includes the «`for`», «`some`» and «`every`» expressions, for example «`for $i in //item[@price > 30] return $i/@code`». These expressions, which are new in XPath 2.0, derive from the tradition of database query languages (SQL, the object database language OQL, and precursors to XQuery) which can be seen as adaptations of the notation of mathematical symbolic logic.

Some other factors that have influenced the design of the XPath syntax include:

- A decision that XPath should have no reserved words. This means that any name that is legal as an XML element name (which includes names such as «`and`» and «`for`») should be legal in a path expression, without any need for escaping. As a result, all names used with some other role, for example function names, variable names, operator names, and keywords such as «`for`» have to be recognizable by their context.
- In both the original applications for XPath (that is, XSLT and XPointer) the language was designed to be embedded within the attributes of an XML document. It therefore has no mechanisms of its own for character escaping, relying instead on the mechanisms available at the XML level (such as numeric character references and entity references). This also made the designers reluctant to use symbols such as «`&&`» which would require heavy escaping. This principle has been abandoned in XPath 2.0 with the introduction of the operators «`<<`» and «`>>`»; however, these operators are not likely to be used very often.
- There was originally an expectation that XPath expressions (especially in an XPointer environment) would often be used as fragment identifiers in a URI. As we have seen, this usage of XPointer never really took off—though there are XML database engines such as Software AG's Tamino that allow queries in the form of XPath expressions to be submitted in this way. This factor meant there was a reluctance to use special characters such as «`#`», «`%`», and «`?`» that have special significance in URIs.

Despite its disparate syntactic roots and its lexical quirks, XPath has managed to integrate these different kinds of expression surprisingly well. In particular, it has retained full composability, so any kind of expression can be nested inside any other.

An Embedded Language

XPath is designed as an embedded language, not as a stand-alone language in its own right. It is designed to provide a language module that can be incorporated into other languages.

This design assumption has two specific consequences:

- Firstly, the language does not need to have every conceivable piece of functionality. In the language of computer science, it does not need to be computationally complete. In more practical terms, it can be restricted to being able to access variables but not to declare them, to call functions but not to define them, to navigate around nodes in a tree but not to create new nodes.

- Secondly, the language can depend on a context established by the host language in which it is embedded. If an embedded language is to be well integrated with a host language, then they should share information so that the user does not need to declare things twice, once for each language. The information that XPath shares with its host language is called the context. This can be divided into information that's available at compile time (the static context), and information that's not available until runtime (the dynamic context). Both aspects of the XPath context are described in Chapter 4 of this book.

A Language for Processing Sequences

The striking feature of the XML data model is that the information is hierarchic. The relationship from an element to its children is intrinsically a one-to-many relationship. Moreover, the relationship is inherently ordered. Sometimes you don't care about the order, but it's part of the nature of XML that the order of elements is deemed to be significant.

This means that when you write an expression such as «`author`» (which is short for «`child::author`», and selects all the `<author>` elements that are children of the context node) then, in principle, the result is a sequence of elements. Very often you know that there will be exactly one author, or that there will be at most one, but in general, the result is a sequence of zero or more elements. The XPath language therefore has to make it convenient to manipulate sequences.

One of the notable consequences of this is the decision that the «=» operator should work on sequences. Suppose a book can have multiple authors. When you write an expression such as «`book[author="Kay"]`», you are selecting all the `<book>` elements that have «`Kay`» as one of their authors: the expression is a shorthand for «`book[some $a in child::author satisfies $a="Kay"]`». It would be very tedious if users had to write this extended expression every time, even in cases where they know there will only be one author, so the language builds this functionality into the semantics of the «=» operator. This feature is known by the rather grand name of *implicit existential quantification*. It's very convenient in many simple cases, though it can trip you up with more complex expressions, especially those involving negation.

When you apply this construct to an element that can only have zero or one occurrences, or to an attribute (which can never have more than one occurrence), the same definition comes into play. A test such as «`book[discount>10]`» will always be false when applied to a book that has no discount. This works in a very similar way to null values in SQL, except that it does not use three-valued logic. In SQL, the corresponding query is «`select book where discount > 10`». In this query, a book that has no discount (that is, where the discount is null) will not be returned. However, because of the way SQL defines three-valued logic, the query «`select book where not(discount > 10)`» will also fail to select any book whose discount is null. By contrast, XPath uses conventional two-valued logic, so the expression «`book[not(discount>10)]`» will return such books.

Some of the people on the XQuery working group whose background was in the design of database query languages were never very happy with the implicit semantics of the «=» operator in XPath, nor with the absence of three-valued logic. However, after much debate, these features of the XPath 1.0 semantics survived intact. The reason, I think, is that for SQL a cell in a table always contains either zero or one values, and "null" represents the zero case. For XML, a child element can have zero, one, or more occurrences within its parent, and (despite the invention of `xsi:nil` by the XML Schema people) the normal way of representing absent data in XML is by the absence of a child element or attribute, which means that selecting that element will return an empty sequence. The empty sequence in XPath therefore

fulfils the same kind of role as the null value in the relational model, but in the context of a model that allows zero to many values, where SQL only allows zero or one.

XPath 1.0 only supported one kind of sequence, namely a set of nodes. Some people liked to think of this as an unordered collection, others as a sequence of nodes in document order, but this doesn't really make any real difference: it wasn't possible to represent a collection in an arbitrary user-defined order, such as employees in order of date of birth. XPath 2.0 has generalized this in two directions: firstly, sequences can now be in any order you like (and can contain duplicates), and secondly, you can have sequences of values (such as strings, numbers, or dates) as well as sequences of nodes.

Many of the operators in XPath 1.0 generalize quite nicely to support arbitrary sequences. For example, the «=» operator still matches if any item in the sequence matches, so you can write for example «`@color = ("red", "green")`» which will be true if the value of the color attribute is either red or green. Other operators, notably the «/» operator used in path expressions and the «|» operator used to combine two sets of nodes, only really make sense in the context of sets containing no duplicates, and these have not been generalized to work on arbitrary sequences.

Types Based on XML Schema

I've already discussed the relationship of XPath 2.0 to XML Schema, so I won't labor it again. But the type system of XPath is something that is highly distinctive, so it deserves a place as one of the key characteristics that gives the language its flavor.

I'll be exploring the type system in depth in Chapter 3. Here, I'll just give a few highlights, as a taste of things to come.

- We need to distinguish the types of the values that XPath can manipulate from the types that can appear as annotations on nodes.
- Atomic types can appear in both roles: You can declare an XPath variable of type integer, and you can also validate the content of an element or attribute as an integer, following which the element or attribute will be annotated with this type.
- Node kinds, such as element, attribute, and comment, appear as types of XPath values (I call these *item types*) but never as type annotations. You can't annotate an attribute as a comment, or even as an attribute, you can only annotate it with a simple type defined in XML Schema.
- Schema types divide into two groups: complex types, and simple types. Simple types divide further into atomic types, list types, and union types. All of these are either built-in types defined in the XML Schema specification, or user-defined types defined in a user-written schema. All of these can be used as type annotations on nodes, if the node has been validated against the appropriate type definition in the schema. But the only schema types that can be used for freestanding values, that is, for values that don't exist as the content of a node, are the atomic types.
- It's possible to have nodes that haven't been validated against any schema. These nodes are labeled as `untyped` in the case of elements, or `untypedAtomic` in the case of attributes.
- Whenever you use a node in an XPath expression in a context where an atomic value (or a sequence of atomic values) is expected, the typed value of the node is extracted. For example, if you write «`@a + 1`», the typed value of the attribute «`a`» is used. If this is a number, all is well.

If it's some other type such as string or date, the expression fails with a type error. But if the value is untyped, that is, if there is no schema, then weak typing comes into play: the value is automatically converted to the required type, in this case, to a number.

This is just a foretaste: the full explanations will appear in Chapter 3.

Summary

This introductory chapter offers an overview of XPath in general, and XPath 2.0 in particular. It tried to answer questions such as:

- What kind of language is it?
- Where does it fit into the XML family?
- Where does it come from and why was it designed the way it is?

We established that XPath is an expression language, and we looked at some of the implications of this in terms of the properties of the language and its relationship to other languages such as XSLT and XQuery. We tried to find some rationale for the large collection of new features that have been added to the language, and for the more fundamental changes to its underlying semantics.

Now it's time to start taking an in-depth look inside XPath 2.0 to see how it works. The next three chapters are about important concepts: the data model, the type system, and the evaluation context. Once you understand these concepts, you should have little difficulty using the language constructs that are introduced later in the book.