8

Sequence Expressions

One of the most notable innovations in XPath 2.0 is the ability to construct and manipulate sequences. This chapter is devoted to an explanation of the constructs in the language that help achieve this.

Sequences can consist either of nodes, or of atomic values, or of a mixture of the two. Sequences containing nodes only are a generalization of the node-sets offered by XPath 1.0. In the previous chapter we looked at the operators for manipulating node-sets, in particular, path expressions, and the operators «`union`», «`intersect`», and «`except`».

In this chapter we look at constructs that can manipulate any sequence, whether it contains nodes, atomic values, or both. Specifically, the chapter covers the following constructs:

- *Sequence concatenation operator:* «`,`»
- *Numeric range operator:* «`to`»
- *Filter expressions:* «`a[b]`»
- *Mapping expressions:* «`for`»
- *Quantified expressions:* «`some`» and «`every`»

First, some general remarks about sequences.

Sequences (unlike nodes) do not have any concept of identity. Given two values that are both sequences, you can ask (in various ways) whether they have the same contents, but you cannot ask whether they are the same sequence.

Sequences are immutable. This is part of what it means for a language to be free of side effects. You can write expressions that take sequences as input and produce new sequences as output, but you can never modify an existing sequence in place.

Sequences cannot be nested. If you want to construct trees, build them as XML trees using nodes rather than atomic values.

A single item is a sequence of length one, so any operation that applies to sequences also applies to single items.

Sequences do not have any kind of type label that is separate from the type labels attached to the items in the sequence. As we will see in Chapter 9, you can ask whether a sequence is an instance of a particular sequence type, but the question can be answered simply by looking at the number of items in the sequence, and at the type labels attached to each item. It follows that there is no such thing as (for example) an "empty sequence of integers" as distinct from an "empty sequence of strings". If the sequence has no items in it, then it also carries no type label. This has some real practical consequences, for example, the `sum()` function, when applied to an expression that can only ever return a sequence of `xs:duration` values, will return the integer 0 (not the zero-length duration) when the sequence is empty, because there is no way at runtime of knowing that if the sequence hadn't been empty, its items would have been durations.

Functions and operators that attach position numbers to the items in a sequence always identify the first item as number 1 (one), not zero. (Although programming with a base of zero tends to be more convenient, Joe Public has not yet been educated into thinking of the first paragraph in a chapter as paragraph zero, and the numbering convention was chosen with this in mind.)

This chapter covers the language constructs that handle general sequences, but there are also a number of useful functions available for manipulating sequences, and these are described in Chapter 10. Relevant functions include: `count()`, `deep-equal()`, `distinct-values()`, `empty()`, `exists()`, `index-of()`, `insert-before()`, `remove()`, `subsequence()`, and `unordered()`.

The Comma Operator

The comma operator can be used to construct a sequence by concatenating items or sequences. We already saw the syntax in Chapter 5, because it appears right at the top level of the XPath grammar:

Expression	Syntax
`Expr`	`ExprSingle («,» ExprSingle)*`

`ExprSingle`	`ForExpr`
	`| QuantifiedExpr`
	`| IfExpr`
	`| OrExpr`

Although the production rule `ExprSingle` lists four specific kinds of expression that can appear as an operand of the «,» operator, these actually cover any XPath expression whatsoever, provided it does not contain a top-level «,».

Because the «,» symbol also has other uses in XPath (for example, it is used to separate the arguments in a function call, and also to separate clauses in «`for`», «`some`», and «`every`» expressions, which we will meet later in this chapter), there are many places in the grammar where use of a general `Expr` is restricted, and only an `ExprSingle` is allowed. In fact, the only places where a general `Expr` (one that contains a top-level comma) is allowed are:

- As the top-level XPath expression
- Within a parenthesized expression

- Within the parentheses of an «if» expression
- Within square brackets as a predicate

Neither of the last two is particularly useful, so in practice the rule is: if you want to use the comma operator to construct a list, then it must either be at the outermost level of the XPath expression, or it must be written in parentheses.

For example, the `max()` function expects a single argument, which is a sequence. If you want to find the maximum of three values `$a`, `$b`, and `$c`, you can write:

```
max(($a, $b, $c))
```

The outer parentheses are part of the function call syntax; the inner parentheses are needed because the expression «`max($a, $b, $c)`» would be a function call with three parameters rather than one, which would be an error.

XPath does not use the JavaScript convention whereby a function call with three separate parameters is the same as a function call whose single parameter is a sequence containing three items.

The operands of the «`,`» operator can be any two sequences. Of course, a single item is itself a sequence, so the operands can also be single items. Either of the sequences can be empty, in which case the result of the expression is the value of the other operand.

The comma operator is often used to construct a list, as in:

```
if ($status = ('current', 'pending', 'deleted', 'closed')) then ...
```

which tests whether the variable `$status` has one of the given four values (recall from Chapter 6 that the «=» operator compares each item in the sequence on the left with each item in the sequence on the right, and returns true if any of these pairs match). In this construct, you probably aren't thinking of «`,`» as being a binary operator that combines two operands to produce a result, but that's technically what it is. The expression «`A,B,C,D`» technically means «`((A,B),C),D)`», but since list concatenation is associative, you don't need to think of it this way.

The order of the items in the two sequences is retained in the result. This is true even if the operands are nodes: there is no sorting into document order. This means that in XSLT, for example, you can use a construct such as:

```
<xsl:apply-templates select="title, author, abstract"/>
```

to process the selected elements in a specified order, regardless of the order in which they appear in the source document. This example is not necessarily processing exactly three elements: there might, for example, be five authors and no abstract. Since the path expression «`author`» selects the five authors in document order, they will be processed in this order, but they will be processed after the `<title>` element whether they precede or follow the title in the source document.

Examples

Here are some examples of expressions that make use of the «,» operator to construct sequences.

Expression	Effect
`max(($net,$gross))`	Selects whichever of `$net` and `$gross` is larger, comparing them according to their actual data type (and using the default collation if they are strings)
`for $i in (1 to 4,8,13) return $seq[$i]`	Selects the items at positions 1, 2, 3, 4, 8, and 13 of the sequence `$seq`. For the meaning of the «to» operator, see the next section
`string-join((@a,@b, @c),"-")`	Creates a string containing the values of the attributes `@a`, `@b`, and `@c` of the context node (in that order), separated by hyphens
`(@code,"N/A")[1]`	Returns the `code` attribute of the context node if it has such an attribute, or the string `"N/A"` otherwise. This expression makes use of the fact that when the `code` attribute is absent, the value of `@code` is an empty sequence, and concatenating an empty sequence with another sequence returns the other sequence (in this case the singleton string `"N/A"`) unchanged. The predicate in square brackets makes this a filter expression: filter expressions are described later in this chapter, on page 244
`book/(author,title, isbn)`	Returns a sequence containing the `<author>`, `<title>`, and `<isbn>` children of a `<book>` element, *in document order*. Although the «,» operator retains the order as specified, the «/» operator causes the nodes to be sorted into document order. So in this case the «,» operator is exactly equivalent to the union operator «|»

Numeric Ranges: The «to» Operator

A range expression has the syntax:

Expression	Syntax
`RangeExpr`	`AdditiveExpr ( «to» AdditiveExpr )?`

The effect is to return a sequence of consecutive integers in ascending order. For example, the expression «`1 to 5`» returns the sequence «`1,2,3,4,5`».

The operands do not have to be constants, of course. A common idiom is to use an expression such as «`1 to count($seq)`» to return the position number of each item in the sequence `$seq`. If the second operand is less than the first (which it will be in this example if `$seq` is an empty sequence), then the

range expression returns an empty sequence. If the second operand is equal to the first, the expression returns a single integer, equal to the value of the first operand.

The two operands must both evaluate to single integers. You can use an untyped value provided it is capable of being converted to an integer: for example you can write «`1 to @width`» if `width` is an attribute in a schema-less document containing the value «`34`». However, you can't use a decimal or a double value without converting it explicitly to an integer. If you write «`1 to @width+1`», you will get a type error, because the value of «`@width+1`» is the double value `35.0e0`. Instead, write «`1 to xs:integer(@width)+1`». or «`1 to 1 + @width idiv 1`».

It's an error if either operand is an empty sequence. For example, this would happen if you ran any of the examples above when the context node did not have a `width` attribute. Supplying a sequence that contains more than one item is also an error.

If you want a sequence of integers in reverse order, you can use the `reverse()` function described in Chapter 10. For example, «`reverse(1 to 5)`» gives you the sequence «`5,4,3,2,1`». In an earlier draft of the specification you could achieve this by writing «`5 to 1`», but the rules were changed because this caused anomalies for the common usage «`1 to count($seq)`» in the case where `$seq` is empty.

Although the semantics of this operator are expressed in terms of constructing a sequence, a respectable implementation will evaluate the sequence lazily, which means that when you write «`1 to 1000000`» it won't actually allocate space in memory to hold a million integers. Depending how you actually use the range expression, in most cases an implementation will be able to iterate over the values one to a million without actually laying them out end-to-end as a list in memory.

Examples

Here are some examples of expressions that make use of the «`to`» operator to construct sequences.

Expression	Effect
`for $n in 1 to 10 return $seq[n]`	Returns the first 10 items of the sequence `$seq`. The «`for`» expression is described later in this chapter, on page 247
`$seq[position() = 1 to 10]`	Returns the first 10 items of the sequence `$seq`. This achieves the same effect as the previous example, but this time using a filter expression alone. It works because the «`=`» operator compares each item in the first operand (there is only one, the value of `position()`), with each item in the second operand (that is, each of the integers 1 to 10), and returns `true` if any of them matches. It's reasonable to expect that XPath processors will optimize this construct so that this doesn't actually involve 10 separate comparisons for each item in the sequence. Note that you can't simply write «`$seq[1 to 10]`». If the predicate isn't a single number, it is evaluated as a boolean, and the effective boolean value of the sequence «`1 to 10`» is `true`, so all the items will be selected

Continues

Expression	Effect
`string-join( for $i in 1 to $N return " ", "")`	Returns a string containing `$N` space characters
`for $i in 1 to count($S) return ($S[$i],$T[$i])`	Returns a sequence that contains pairs of corresponding values from the two input sequences `$S` and `$T`. For example, if `$S` is the sequence `("a","b","c")` and `$T` is the sequence `("x","y","z")`, the result will be the sequence `("a","x","b","y","c","z")`

Filter Expressions

A filter expression is used to apply one or more `Predicates` to a sequence, selecting those items in the sequence that satisfy some condition.

Expression	Syntax
`FilterExpr`	`PrimaryExpr Predicate*`
`Predicate`	«`[`» `Expr` «`]`»

A `FilterExpr` consists of a `PrimaryExpr` whose value is a sequence, followed by zero or more `Predicates` that select a subset of the items in the sequence. Each predicate consists of an expression enclosed in square brackets, for example «`[@name='London']`» or «`[position()=1]`».

The way the syntax is defined, every `PrimaryExpr` is also a trivial `FilterExpr`, including simple expressions such as «`23`», «`'Washington'`», and «`true()`».

Since in XPath 2.0 every value is a sequence, it is possible to apply predicates to any value whatsoever. For example, it is legitimate to write «`1[$param]`». This returns the value «`1`» if `$param` is `true`, or an empty sequence if `$param` is `false`.

Each predicate is applied to the sequence in turn; only those items in the sequence for which the predicate is true pass through to the next stage. The final result consists of those items in the original sequence that satisfy each of the predicates, retaining their original order.

A predicate may be either a numeric predicate (for example «`[1]`» or «`[last()-1]`»), or a boolean predicate (for example «`[count(*) gt 5]`» or «`[@name and @address]`»). If the value of the expression is a single number, it is treated as a numeric predicate; otherwise it is converted, if necessary, to an `xs:boolean`, and is treated as a boolean predicate. The conversion is done using the rules for computing the *effective boolean value*, which are the same rules as are used for the condition in an «`if`» expression (described in Chapter 5 on page 117) or for the operand of the `boolean()` function (described in Chapter 10 on page 304), except that if the value is a single number—which might be an

integer, decimal, float, or double—then the predicate is treated as a numeric predicate rather than a boolean predicate.

If the value of the predicate contains nodes, there is no automatic atomization of the nodes (that is, the values of the nodes are not extracted). In fact, if the value of the predicate contains one or more nodes, then its effective boolean value is always true. This means, for example, that «`person[@isMarried]`» selects any `<person>` element that has an `isMarried` attribute, irrespective of the value of that attribute. If you want to test the value of the attribute, you can atomize it explicity using the `data()` function, or you can use a comparison such as «`person[@isMarried=true()]`».

A numeric predicate whose value is `N` is equivalent to the boolean predicate «`[position() eq N]`». So, for example, the numeric predicate «`[1]`» means «`[position() eq 1]`», and the numeric predicate «`[last()]`» means «`[position() eq last()]`».

It's important to remember that this implicit testing of `position()` happens only when the predicate expression actually evaluates to a single number. For example, «`$paras[1 or last()]`» does not mean «`$paras[position()=1 or position()=last()]`», because the result of evaluating «`1 or last()`» is a boolean, not a number (and as it happens, it will always be true). Similarly, «`book[../@book-nr]`» does not mean «`book[position()=../@book-nr]`», because the result of «`../@book-nr`» is a node, not a number.

> *A neat way to force the node to be atomized in such cases is to use the unary «+» operator: write* «`book[+../@book-nr]`».

A consequence of the rule is that if the predicate is a number that is not equal to an integer, the result will be an empty sequence. For example, «`$S[last() div 2]`» will select nothing when the value of `last()` is an odd number. If you want to select a single item close to the middle of the sequence, use «`$S[last() idiv 2]`», because the `idiv` operator always returns an integer.

In nearly all practical cases, a numeric predicate selects either a single item from the sequence, or no items at all. But this is not part of the definition. To give a counter-example, «`$x[count(*)]`» selects every node whose position is the same as the number of children it has.

As discussed in Chapter 4, every XPath expression is evaluated in some context. For an expression used as a predicate, the context is different from the context of the containing expression. While evaluating each predicate, the context is established as follows:

- The *context item* (the item referenced as «`.`») is the item being tested
- The *context position* (the value of the `position()` function) is the position of that item within the sequence of items surviving from the previous stage
- The *context size* (the value of the `last()` function) is the number of items surviving from the previous stage.

To see how this works, consider the filter expression «`$headings [self::h1] [last()]`». This starts with the sequence of nodes that is the value of the variable «`$headings`» (if this sequence contains items that are not nodes, then evaluating the predicate «`self::h1`» will raise an error). The first predicate is «`[self::h1]`». This is applied to each node in «`$headings`» in turn. While it is

being applied, the context node is that particular node. The expression «self::h1» is a path expression consisting of a single AxisStep: it selects a sequence of nodes. If the context node is an <h1> element this sequence will contain a single node—the context node. Otherwise, the sequence will be empty. When this value is converted to a boolean, it will be true if it contains a node, and false if it is empty. So the first predicate is actually filtering through those nodes in «$headings» that are <h1> elements.

The second predicate is now applied to each node in this sequence of <h1> elements. In each case the predicate «[last()]» returns the same value: a number indicating how many <h1> elements there are in the sequence. As this is a numeric predicate, a node passes the test when «[position()= last()]», that is, when the position of the node in the sequence (taken in its original order) is equal to the number of nodes in the sequence. So the meaning of «$headings [self::h1] [last()]» is "the last <h1> element in the sequence $headings."

Note that this isn't the same as «$headings [last()] [self::h1]», which means "the last item in $headings, provided it is an <h1> element."

The operation of a Predicate in a FilterExpr is very similar to the application of a Predicate in an AxisStep (which we studied in Chapter 7, on page 230), and although they are not directly related in the XPath grammar rules, you can often use Predicates without being fully aware which of these two constructs you are using. For example, «$para[1]» is a FilterExpr, while «para[1]» is an AxisStep. The main differences to watch out for are firstly, that in a path expression the predicates apply only to the most recent Step (for example, in «book/author[1]» the «[1]» means the first author within each book), and secondly, that in a filter expression the items are always considered in the order of the supplied sequence (whereas in an AxisStep they can be in forward or reverse document order depending on the direction of the axis).

Examples

Expression	Description
$paragraphs[23]	This FilterExpr consists of a VariableReference filtered by a Predicate. It selects the 23rd item in the sequence that is the value of variable $paragraphs, taking them in the order of that sequence. If there is no 23rd item, the expression returns an empty sequence
key('empname', 'John Smith')[@loc='Sydney']	This FilterExpr comprises a FunctionCall filtered by a Predicate. The key() function is available only in XSLT. Assuming that the key «empname» has been defined in the containing stylesheet to select employees by name, it selects all employees named John Smith who are located in Sydney
(//section|//subsection) [title='Introduction']	This FilterExpr consists of a parenthesized UnionExpr filtered by a Predicate. It selects all <section> and <subsection> elements that have a child <title> element with the content «Introduction»

Expression	Description
`(//@href/doc(.))[pricelist][1]`	This `FilterExpr` first selects all documents referenced by URLs contained in `href` attributes anywhere in the source document, by applying the `doc()` function to the value of each of these attributes. The «/» operator causes any duplicates to be removed, as described in Chapter 7. From this set of documents it selects those whose outermost element is named `<pricelist>`, and from these it selects the first. The order of nodes that are in different documents is not defined, so if there are several price lists referenced, it is unpredictable which will be selected

Where a predicate is used as part of a `FilterExpr` (as distinct from an `AxisStep`), the items are considered in their original sequence for the purpose of evaluating the `position()` function within the predicate. There are some cases where the order of the sequence is not predictable, but it is still possible to use positional predicates. For example the result of the `distinct-values()` function is in an undefined order, but you can still write «`distinct-values($in)[1]`» to obtain one item in the sequence, chosen arbitrarily.

The «for» Expression

The «`for`» expression is one of the most powerful new features in XPath 2.0, and is closely related to the extension to the data model to handle sequences. Its effect is to apply an expression to every item in an input sequence, and to return the concatenated results of these expressions.

The syntax also allows several sequences to be provided as input, in which case the effect is to apply an expression to every combination of values taken one from each sequence.

The syntax as given in the XPath 2.0 Recommendation is rather clumsy, because the grammar is designed to share as many production rules as possible with XQuery, and the «`for`» expression in XPath can be regarded as a cut-down version of XQuery's much richer FLWOR expressions. For this book, I've rewritten the syntax in the way it would probably have been presented if XQuery didn't exist.

Expression	Syntax
`ForExpr`	`«for $» VarName «in» ExprSingle («,» «$» VarName «in» ExprSingle)* «return» ExprSingle`
`VarName`	`QName`

An `ExprSingle` is any XPath expression that does not contain a top-level «`,`» operator. If you want to use an expression containing a «`,`» operator, write it in parentheses. For example the expression «`for $i in (1,5,10) return $i+1`» returns the sequence «`2,6,11`».

The notation «`for $`» indicates that for the purposes of parsing, the word «`for`» must be followed by a «`$`» sign to be recognized as a keyword. The two parts of this compound symbol can be separated by whitespace and comments.

We'll look first at «`for`» expressions that operate on a single sequence, and then move on to the more general case where there are multiple input sequences.

Mapping a Sequence

When used with a single sequence, the «`for`» expression applies the expression in the «`return`» clause to each item in the input sequence. The relevant item in the input sequence is accessed not as the context item, but as the value of the variable declared in the «`for`» clause.

> *These variables are referred to as range variables, to distinguish them from variables supplied from outside the XPath expression, such as variables declared in an XSLT stylesheet. The term comes originally from the branch of mathematical logic called predicate calculus, and has been adopted in a number of programming languages based on this underlying theory.*

In most cases the expression in the «`return`» clause will depend in some way on the range variable. In other words, the «`return`» value is a function of the range variable, which means we can rewrite the «`for`» expression in the abstract form:

```
for $x in $SEQ return F($x)
```

where «`F($x)`» represents any expression that depends on `$x` (it doesn't have to depend on `$x`, but it usually will).

What this expression does is to evaluate the expression «`F($x)`» once for each item in the input sequence `$SEQ`, and then to concatenate the results, respecting the original order of the items in `$SEQ`.

In the simplest case, the return expression «`F($x)`» returns one item each time it is called. This is illustrated in Figure 8-1, where the function «`F($x)`» in this example is actually the expression «`string-length($x)`».

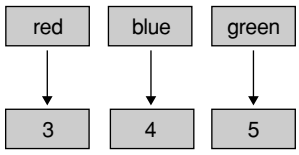

Figure 8-1

We say that the expression «`for $x in $SEQ return string-length($x)`» *maps* the sequence «`"red","blue","green"`» to the sequence «`3,4,5`».

In this case, the number of items in the result will be the same as the number of items in the input sequence.

However, the return expression isn't constrained to return a single item, it can return any sequence of zero or more items. For example, you could write:

```
for $s in ("red", "blue", "green") return string-to-codepoints($s)
```

The function `string-to-codepoints()`, which is part of the standard library defined in Chapter 10, returns for a given string, the Unicode code values of the characters that make up the string. For example, «`string-to-unicode("red")`» returns the sequence «`114,101,100`». The result of the above expression is a sequence of 12 integers, as illustrated in Figure 8-2.

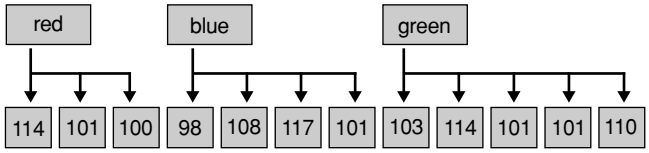

Figure 8-2

The integers are returned in the order shown, because unlike a path expression, there is nothing in the rules for a «`for`» expression that causes the result sequence to be sorted into document order. Indeed, document order is not a meaningful concept when we are dealing with atomic values rather than nodes.

Examples

Expression	Description
`for $i in 1 to 5` `return $i*$i`	Returns the sequence «`1,4,9,16,25`». This example is a one-to-one mapping
`for $i in 0 to 4` `return 1 to $i`	Returns the sequence «`1,1,2,1,2,3,1,2,3,4`». This example is a one-to-many mapping. Note that for the first item in the input sequence (0), the mapping function returns an empty sequence, so this item contributes nothing to the result

For Expressions and Path Expressions

The items in the input sequence of a «`for`» expression can be atomic values or nodes, or any mixture of the two. When applied to a sequence of nodes, «`for`» expressions actually behave in a very similar way to path expressions. The expression:

```
for $c in chapter return $c/section
```

returns exactly the same result as the path expression:

```
chapter/section
```

However, there are some significant differences between «`for`» expressions and path expressions:

- In a path expression, both the input sequence and the step expression are required to return nodes exclusively. A «`for`» expression can work on any sequence, whether it contains nodes or atomic values or both, and it can also return any sequence.

- Path expressions always sort the resulting nodes into document order, and eliminate duplicates. A «`for`» expression returns the result sequence in the order that reflects the order of the input items.
- In a path expression, the context item for evaluating a step is set to each item in the input sequence in turn. In a «`for`» expression, the range variable fulfils this function. The context item is not changed. Nor are the context position and size (`position()` and `last()`) available to test the position of the item in the input sequence.

A common mistake is to forget that «`for`» expressions don't set the context node. The following example is wrong (it's not an error, but it doesn't do what the writer probably intended):

```
(:wrong:) sum(for $i in item return @price * @qty)
```

The correct way of writing this is:

```
(:correct:) sum(for $i in item return $i/@price * $i/@qty)
```

Generally speaking, there is usually something amiss if the range variable is not used in the «`return`» expression. However, there are exceptions to this rule. For example, it's quite reasonable to write:

```
string-join(for $i in 1 to $n return "-", "")
```

which returns a string containing `$n` hyphens.

It's also often (but not invariably) a sign of trouble if the value of the return expression depends on the context item. But it's not actually an error: the context item inside the return expression is exactly the same as the context item for the «`for`» expression as a whole. So it's legal to write an expression such as:

```
chapter/(for $i in 1 to 10 return section[$i])
```

which returns the first 10 sections of each chapter.

Combining Multiple Sequences

The «`for`» expression allows multiple input sequences to be defined, each with its own range variable. For example, you can write:

```
for $c in //customer,
    $o in $c/orders,
    $ol in $o/line
return $ol/cost
```

The simplest way to think about this is as a nested loop. You can regard the «`,`» as a shorthand for writing the keywords «`return for`», so the above expression is equivalent to:

```
for $c in //customer
return
   for $o in $c/orders
```

```
    return
       for $ol in $o/line
       return $ol/cost
```

Note that each of the range variables can be referenced in the expression that defines the input sequence for the next range variable.

In the example above, each iteration is rather like a step in a path expression; it selects nodes starting from the node selected in the containing loop. But it doesn't have to be this way. For example, you could equally write an expression such as:

```
for $c in doc('customers.xml')//customer,
    $p in doc('products.xml')//product
            [$c/orders/product-code = $p/code]
   return $c/line/cost
```

It's still true that this is equivalent to a nested-loop expression:

```
for $c in doc('customers.xml')//customer
return
   for $p in doc('products.xml')//product
            [$c/orders/product-code = $p/code]
   return $c/line/cost
```

The other way to think about this, particularly if you are familiar with SQL, is as a relational join. The system isn't actually obliged to evaluate the «`for`» expression using nested loops (this applies whether you write it in the abbreviated form using multiple range variables separated with commas, or whether you use the expanded form shown above). Instead, the optimizer can use any of the mechanisms developed over the years in relational database technology to evaluate the join more rapidly. There's no guarantee that it will do so (in practice, I think XQuery implementations are likely to put a lot of effort into join optimization, while XPath implementations might be less ambitious), so you need to use potentially expensive constructs like this with some care.

> *Saxon, at the time of writing, will try to move sub-expressions out of a loop if they don't depend on the range variable. So the expression* «`doc('products.xml')//product`» *will probably only be evaluated once, and the expression* «`$c/orders/product-code`» *will only be evaluated once for each customer. But after this, every product code in the customer file will be compared with every product code in the product file. In XSLT, you can avoid this overhead by using keys: see the description of the* `<xsl:key>` *declaration and the* `key()` *function in XSLT 2.0 Programmer's Reference.*

Example

Expression	Description

`count(`
`  for $i in 1 to 8,`
`  $j in 1 to 8`
`  return f:square($i,$j))`
	Assuming that «`f:square(row,column)`» returns an integer identifying the piece that occupies a square on a chessboard, or an empty sequence if the square is unoccupied, this expression returns all the pieces on the board

Chapter 8

Examples in XMLSpy

The XMLSpy 2004 product (see `http://www.altova.com/`) includes a beta release of an XPath 2.0 processor that shows the results of an expression using a graphical user interface. In this section I will provide a couple of examples that illustrate the results of «`for`» expressions using that product.

I'm using the sample document `ipo.xml` that comes with the product: look in the Purchase Order folder. This consists of an outer element `<ipo:purchase-order>` with various namespace declarations, then addresses for shipping and billing:

```
<shipTo export-code="1" xsi:type="ipo:EU-Address">
   <name>Helen Zoe</name>
   <street>47 Eden Street</street>
   <city>Cambridge</city>
   <postcode>126</postcode>
</shipTo>
<billTo xsi:type="ipo:US-Address">
   <name>Robert Smith</name>
   <street>8 Oak Avenue</street>
   <city>Old Town</city>
   <state>AK</state>
   <zip>95819</zip>
</billTo>
```

This is followed by an `<items>` element containing a number of items with the general format:

```
<item partNum="833-AA">
   <productName>Lapis necklace</productName>
   <quantity>2</quantity>
   <price>99.95</price>
   <ipo:comment>Need this for the holidays!</ipo:comment>
   <shipDate>1999-12-05</shipDate>
</item>
```

(Altova took this example from the XML Schema primer published by W3C, but apparently failed to realize the subtlety that UK postcodes are alphanumeric.)

Let's look first at the classic problem of getting the total value of the order. In this expression (Figure 8-3) I'll first list all the individual price and quantity elements, and then their sum.

The total is shown in the bottom line.

XMLSpy takes the namespace context for the XPath expression from the namespaces declared in the source document, so to get this to work, I had to add the namespace declaration «`xmlns:xs="http://www.w3.org/2001/XMLSchema"`» to the `<ipo:purchaseOrder>` element.

The second example from XMLSpy uses a join, and just to show that joins don't arise only from data-oriented XML, I've chosen an example that uses narrative XML as its source. Specifically, it uses the XML source of the XPath 1.0 specification, which happens to be included in XMLSpy as a sample document.

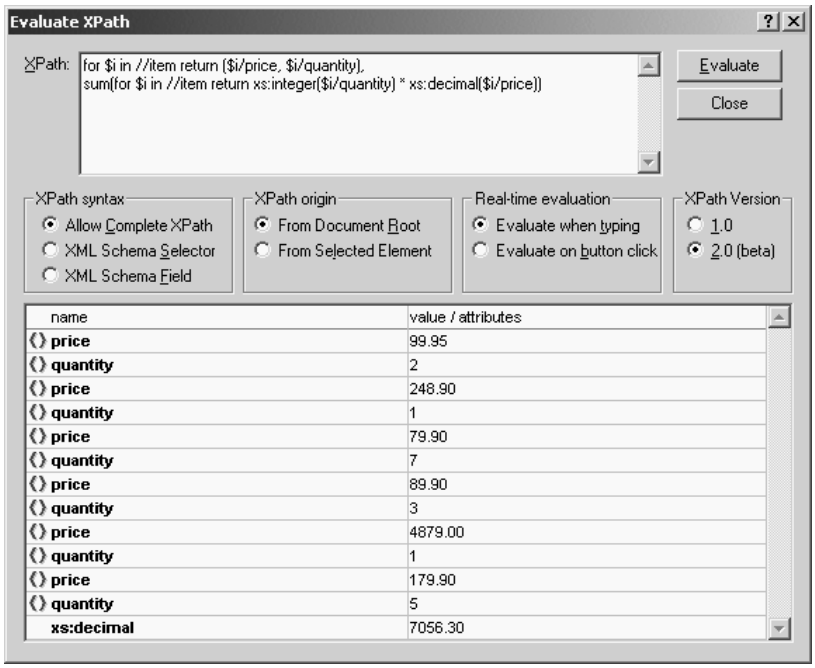

Figure 8-3

The DTD for this document type allows term definitions to be marked up using a `<termdef>` element such as:

```
<termdef id="dt-document-order" term="Document Order">There is an
ordering, <term>document order</term>, defined on all the nodes in the
document corresponding to the order in which the first character of
the XML representation of each node occurs in the XML representation
of the document after expansion of general entities.</termdef>
```

References to a defined term can be marked up using a `<termref>` element. This example shows a `<termref>` that happens to be nested inside another `<termdef>`:

```
<termdef id="dt-reverse-document-order" term="Reverse Document
Order"> <term>Reverse document order</term> is the reverse of
<termref def="dt-document-order">document order</termref>.</termdef>
```

There is a relationship between the `<termref>` element and the `<termdef>` element, by virtue of the fact that the `def` attribute of a `<termref>` must match the `id` attribute of a `<termdef>`. The stylesheet used to construct the published XPath specification turns this relationship into a hyperlink. Where there is a relationship, there is potential for performing a join, as Figure 8-4 shows.

The output here is not particularly visual. It shows a sequence of pairs, each pair containing first, a defined term (the term attribute of a `<termdef>` element), and secondly, the heading (`<head>` element) of the innermost `<div1>`, `<div2>`, or `<div3>` section that contains a reference to that term. The reason that this is shown as a one-dimensional list rather than as a table is of course that it *is* a list: the XPath 2.0

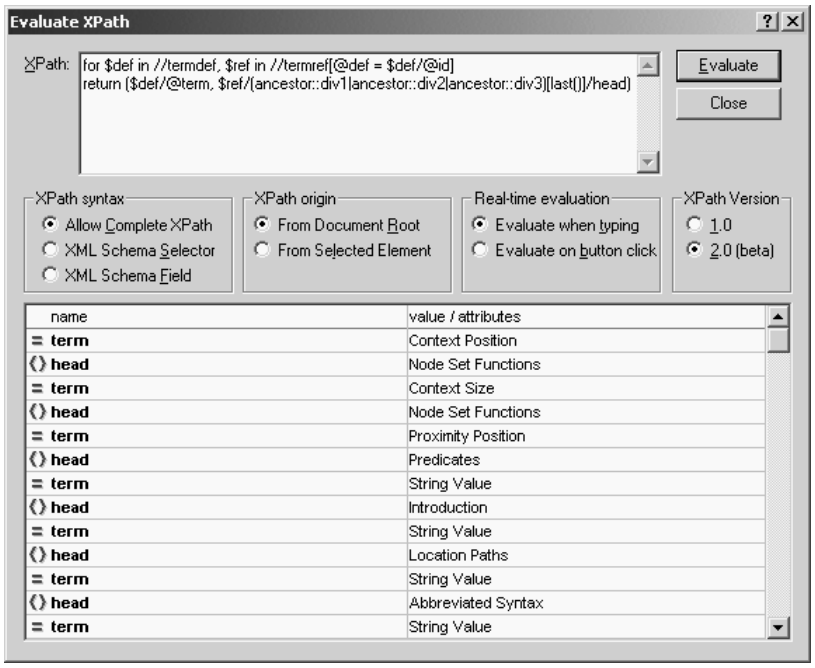

Figure 8-4

data model does not allow construction of trees, or of nested sequences, that would allow a table to be represented more directly. In practice, you would either use a custom application to present the data, or you would embed this XPath expression in an XSLT stylesheet or XQuery query that allows you to construct the output as XML or (say) HTML. The resulting display would probably look something like this:

Term	Section containing Reference
Context Position	Node Set Functions
Context Size	Node Set Functions
Proximity Position	Predicates
String Value	Introduction
String Value	Location Paths
String Value	Abbreviated Syntax

The «some» and «every» Expressions

These expressions are used to test whether some item in a sequence satisfies a condition, or whether all values in a sequence satisfy a condition.

The syntax is like this:

Expression	Syntax
QuantifiedExpr	«some $»|«every $» VarName «in» ExprSingle («,»«$» VarName «in» ExprSingle)* «satisfies» ExprSingle
VarName	QName

The name *quantified expression* comes from the mathematical notations on which these expressions are based: the «some» expression is known in formal logic as an existential quantifier, while the «every» expression is known as a universal quantifier.

As with the «for» expression, these two expressions bind a range variable to every item in a sequence in turn, and evaluate an expression (the «satisfies» expression) for each of these items. Instead of returning the results, however, a quanitified expression evaluates the effective boolean value of the «satisfies» expression. In the case of «some», it returns true if at least one of these values is true, while in the case of «every», it returns true if all of the values are true. The range variables can be referenced anywhere in the expression following the «satisfies» keyword, and the expression following the «in» keyword can use all variables declared in previous clauses of the expression (but not the variable ranging over that expression itself).

For example:

```
some $p in //price satisfies $p > 10000
```

is true if there is a <price> element in the document whose typed value is a number greater than 10,000, while:

```
every $p in //price satisfies $p > 10000
```

is true if every <price> element in the document has a typed value greater than 10,000.

The result of the expression (unless some error occurs) is always a single xs:boolean value.

The «satisfies» expression is evaluated to return a boolean value. This evaluation returns the *effective boolean value* of the expression, using the same rules as for the boolean() function and the condition in an «if» expression. For example, if the result of the expression is a string, the effective boolean value is true if the string is not zero-length. The expression will almost invariably reference each one of the range variables, although the results are still well defined if it doesn't.

As with «for» expressions, «some» and «every» expressions do not change the context item. This means that the following is wrong (it's not an error, but it doesn't produce the intended answer):

```
(:wrong:) some $i in //item satisfies price > 200
```

It should be written instead:

```
(:correct:) some $i in //item satisfies $i/price > 200
```

Note that if the input sequence is empty, the «some» expression will always be `false`, while the «every» expression will always be `true`. This may not be intuitive to everyone, but it is logical—the «every» expression is true if there are no counter-examples, for example, it's true that every unicorn has one horn, because there are no Unicorns that don't have one horn. Equally, and this is where the surprise comes, it is also true that every Unicorn has two horns.

In fact these two expressions are interchangeable: you can always rewrite

```
every $s in $S satisfies not(C)
```

as:

```
not(some $s in $S satisfies C)
```

If there is only a single range variable, you can usually rewrite the expression

```
some $s in $S satisfies $s/C
```

as

```
exists($S[C])
```

which some people prefer, as it is more concise. If the sequence `$S` consists of nodes, you can also leave out the call on the `exists()` function, for example, you can rewrite:

```
if (some $i in //item satisfies $i/price * $i/quantity > 1000) ...
```

as:

```
if (//item[price*quantity > 1000]) ...
```

The difference is a matter of taste. The «some» expression, however, is more powerful than a simple predicate because (like the «for» expression) it can handle joins, using multiple range variables.

The XPath 2.0 specification describes the semantics of the «some» and «every» expressions in a rather complicated way, using a concept of "tuples of variable bindings". This happened because the XPath 2.0 specification is generated by subsetting XQuery 1.0, whose core construct, the FLWOR expression, makes use of this concept already. It would have been possible to specify «some» and «every» in a much simpler way for XPath users. In fact, the expression:

```
some $s in $S, $t in $T, $u in $U satisfies CONDITION
```

has exactly the same effect as the expression:

```
exists(for $s in $S, $t in $T, $u in $U return boolean( CONDITION ))[.]
```

while the expression:

```
every $s in $S, $t in $T, $u in $U satisfies CONDITION
```

has exactly the same effect as the expression:

```
empty(for $s in $S, $t in $T, $u in $U return not( CONDITION ))[.]
```

The rather unusual predicate «[.]» selects all the items in a sequence whose effective boolean value is true. In the first case, the result is `true` if the result of the «for» expression contains at least one value that is `true`, while in the second case, the result is `true` if the result of the «for» expression contains no value that is `false`.

(The functions `exists()` and `empty()` are described in Chapter 10. The `exists()` function returns true if the supplied sequence contains one or more items, while `empty()` returns true if the sequence contains no items.)

Examples

Expression	Description
`some $i in //item satisfies $i/price gt 200`	Returns `true` if the current document contains an `<item>` element with a `<price>` child whose typed value exceeds 200
`some $n in 1 to count($S)-1 satisfies $S[$n] eq S[$n+1]`	Returns `true` if there are two adjacent values in the input sequence `$S` that are equal
`every $p in //person satisfies $p/@dob castable as xs:date`	Returns `true` if every `<person>` element in the current document has a `dob` attribute that represents a valid date, according to the XML Schema format `YYYY-MM-DD`
`some $k in //keyword, $p in //para satisfies contains($p, $k)`	Returns `true` if there is at least one `<keyword>` in the document that is present in at least one `<para>` element of the document
`every $d in //termdef/@id satisfies some $r in //termref satisfies $d eq $r/@def`	Returns `true` if every `<termdef>` element with an `id` attribute is referenced by at least one `<termref>` element with a matching `def` attribute

Quantification and the «=» Operator

An alternative to using the «some» expression (and sometimes also the «every» expression) is to rely on the implicit semantics of the «=» operator, and other operators in the same family, when they are used to compare sequences. As we saw in Chapter 6, these operators can be used to compare two sequences, and return `true` if any pair of items (one from each sequence) satisfies the equality condition.

For example, the expression:

```
//book[author="Kay"]
```

means the same as

```
//book[some $a in author satisfies $a eq "Kay"]
```

Similarly, the expression:

```
//book[author=("Kay", "Tennison", "Carlisle")]
```

means the same as:

```
//book[some $a in author,
            $s in ("Kay", "Tennison", "Carlisle")
      satisfies $a eq $s]
```

It's a matter of personal style which one you choose in these cases. However, if the operator is something more complex than straight equality—for example, if you are comparing the two values using the `compare()` function with a non-default collation—then the only way to achieve the effect within XPath is to use a «`some`» or «`every`» expression.

Errors in «some» and «every» Expressions

Dynamic (runtime) errors can occur in «`some`» and «`every`» expressions just as in any other kind of XPath expression, and the rules are the same. But for these expressions the rules have some interesting consequences that are worth exploring.

Let's summarize the rules here:

- If a dynamic error occurs when evaluating the «`satisfies`» expression, then the «`some`» or «`every`» expression as a whole fails.
- As soon as the system finds an item in the sequence for which the «`satisfies`» expression is true (in the case of «`some`») or false (in the case of «`every`») then it can stop the evaluation. It doesn't need to look any further. This means that it might not notice errors that would be found if it carried on to the bitter end.
- The system can process the input sequence in any order that it likes. This means that if there is one item for which evaluating the «`satisfies`» expression returns `true`, and another for which it raises an error, then you can't tell whether the «`some`» expression will return `true` or raise the error.

Some systems might deliberately choose to exploit these rules by evaluating the error cases last (or pretending to do so) so as to minimize the chance of the expression failing, but you can't rely on this.

What does this mean in practice? Suppose you have an attribute defined in the schema as follows:

```
<xs:attribute name="readings">
  <xs:simpleType>
    <xs:list>
      <xs:simpleType>
        <xs:union>
```

```
            <xs:simpleType base="xs:decimal"/>
            <xs:simpleType base="xs:string"/>
              <xs:enumeration value="n/a"/>
            </xs:simpleType>
          </xs:union>
        </xs:simpleType>
      </xs:list>
    </xs:simpleType>
  </xs:attribute>
```

Or to put it more simply, the attribute's typed value is a list of atomic values, each of which is either a decimal number or the string value «`n/a`». For example, the attribute might be written «`readings="12.2 -8.4 5.6 n/a 13.1"`».

Now suppose you want to test whether the set of readings includes a negative value. You could write:

```
if (some $a in data(@readings) satisfies $a lt 0) then ...
```

The chances are you will get away with this. Most processors will probably evaluate the condition «`$a lt 0`» against each value in turn, find that the condition is true for the second item in the list, and return `true`. However, a processor that decided to evaluate the items in reverse order would encounter the value «`n/a`», compare this with zero, and hit a type error: you can't compare a string with a number. So one processor will give you the answer `true`, while another gives you an error.

You can protect yourself against this error by writing the expression as:

```
if (some $a in data(@readings)[. instance of xs:decimal]
    satisfies $a lt 0)
then ...
```

Or in this case, you can mask the error by writing:

```
if (some $a in data(@readings) satisfies number($a) lt 0) then ...
```

This works because «`number('n/a')`» returns `NaN` (not-a-number), and «`NaN lt 0`» returns `false`.

Summary

This chapter covered all the various kinds of expressions in the XPath language that are designed to manipulate general sequences, specifically:

- The «`,`» operator, which appends two sequences
- The «`to`» operator, which forms a sequence of ascending integers
- Filter expressions, which are used to find those items in a sequence that satisfy some predicate
- The «`for`» expression, which applies an expression to every item in a sequence and returns the results, as a new sequence
- The «`some`» and «`every`» expressions, which test whether a condition is true for some value (or every value) in an input sequence, returning a boolean result.

Don't forget that these are not the only constructs available for manipulating sequences. For sequences of nodes, path expressions can be used, as well as the «`union`», «`intersect`», and «`except`» operators, as discussed in Chapter 7. And in Chapter 10 you will find descriptions of all the functions in the standard XPath library, including many functions that are useful for operating on sequences, for example, `count()`, `deep-equal()`, `distinct-values()`, `empty()`, `exists()`, `index-of()`, `insert-before()`, `remove()`, `subsequence()`, and `unordered()`.

The next chapter deals with operations involving types: operations that convert a value of one type into a value of another type, and operations that test the type of a value.