CHAPTER 4

Navigating Input Documents Using Paths

Path expressions are used to navigate input documents to select elements and attributes of interest. This chapter explains how to use path expressions to select elements and attributes from an input document and apply predicates to filter those results. It also covers the different methods of accessing input documents.

Path Expressions

A path expression is made up of one or more steps that are separated by a slash (/) or double slashes (//). For example, the path:

```
doc("catalog.xml")/catalog/product
```

selects all the `product` children of the `catalog` element in the `catalog.xml` document. Table 4-1 shows some other simple path expressions.

Table 4-1. Simple path expressions

Example	Explanation
`doc("catalog.xml")/catalog`	The `catalog` element that is the outermost element of the document
`doc("catalog.xml")//product`	All `product` elements anywhere in the document
`doc("catalog.xml")//product/@dept`	All `dept` attributes of `product` elements in the document
`doc("catalog.xml")/catalog/*`	All child elements of the `catalog` element
`doc("catalog.xml")/catalog/*/number`	All `number` elements that are grandchildren of the `catalog` element

Path expressions return nodes in document order. This means that the examples in Table 4-1 return the `product` elements in the same order that they appear in the `catalog.xml` document. More information on document order and on sorting results differently can be found in Chapter 7.

Path Expressions and Context

A path expression is always evaluated relative to a particular context item, which serves as the starting point for the relative path. Some path expressions start with a step that sets the context item, as in:

```
doc("catalog.xml")/catalog/product/number
```

The function call `doc("catalog.xml")` returns the document node of the `catalog.xml` document, which becomes the context item. When the context item is a node (as opposed to an atomic value), it is called the *context node*. The rest of the path is evaluated relative to it. Another example is:

```
$catalog/product/number
```

where the value of the variable `$catalog` sets the context. The variable must select zero, one or more nodes, which become the context nodes for the rest of the expression.

A path expression can also be relative. For example, it can also simply start with a name, as in:

```
product/number
```

This means that the path expression will be evaluated relative to the current context node, which must have been previously determined outside the expression. It may have been set by the processor outside the scope of the query, or in an outer expression.

Steps and changing context

The context item changes with each step. A step returns a sequence of zero, one, or more nodes that serve as the context items for evaluating the next step. For example, in:

```
doc("catalog.xml")/catalog/product/number
```

the `doc("catalog.xml")` step returns one document node that serves as the context item when evaluating the `catalog` step. The `catalog` step is evaluated using the document node as the current context node, returning a sequence of one `catalog` element child of the document node. This `catalog` element then serves as the context node for evaluation of the `product` step, which returns the sequence of `product` children of `catalog`.

The final step, `number`, is evaluated in turn for each `product` child in this sequence. During this process, the processor keeps track of three things:

- The context node itself—for example, the `product` element that is currently being processed
- The context sequence, which is the sequence of items currently being processed—for example, all the `product` elements
- The position of the context node within the context sequence, which can be used to retrieve nodes based on their position

Steps

As we have seen in previous examples, steps in a path can simply be primary expressions like function calls (`doc("catalog.xml")`) or variable references (`$catalog`). Any expression that returns nodes can be on the lefthand side of the slash operator.

Another kind of step is the *axis step*, which allows you to navigate around the XML node hierarchy. There are two kinds of axis steps:

Forward step
: This step selects descendents or nodes appearing after the context node (or the context node itself).

Reverse step
: This step selects ancestors or nodes appearing before the context node (or the context node itself).

In the examples so far, `catalog`, `product`, and `@dept` are all axis steps (that happen to be forward steps). The syntax of an axis step is shown in Figure 4-1.

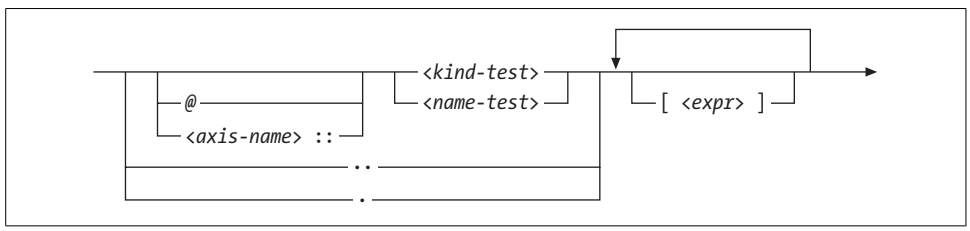


Figure 4-1. Syntax of a step in a path expression

Axes

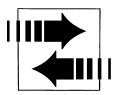
Each forward or reverse step has an *axis*, which defines the direction and relationship of the selected nodes. For example, the `child::` axis (a forward axis) can be used to indicate that only child nodes should be selected, while the `parent::` axis (a reverse axis) can be used to indicate that only the parent node should be selected. The 12 axes are listed in Table 4-2.

Table 4-2. Axes

Axis	Meaning
`self::`	The context node itself.
`child::`	Children of the context node. Attributes are not considered children of an element. This is the default axis if none is specified.
`descendant::`	All descendants of the context node (children, children of children, etc.). Attributes are not considered descendants.
`descendant-or-self::`	The context node and its descendants.
`attribute::`	Attributes of the context node (if any).

Table 4-2. Axes (continued)

Axis	Meaning
`following::`	All nodes that follow the context node in the document, minus the context node's descendants.
`following-sibling::`	All siblings of the context node that follow it. Attributes of the same element are not considered siblings.
`parent::`	The parent of the context node (if any). This is either the element or the document node that contains it. The parent of an attribute is its element, even though it is not considered a child of that element.
`ancestor::`	All ancestors of the context node (parent, parent of the parent, etc.).
`ancestor-or-self::`	The context node and all its ancestors.
`preceding::`	All nodes that precede the context node in the document, minus the context node's ancestors.
`preceding-sibling::`	All the siblings of the context node that precede it. Attributes of the same element are not considered siblings.



An additional forward axis, `namespace`, is supported (but deprecated) by XPath 2.0 but not supported at all by XQuery 1.0. It allows you to access the in-scope namespaces of a node.

Implementations are not required to support the following axes: `following`, `following-sibling`, `ancestor`, `ancestor-or-self`, `preceding`, and `preceding-sibling`.

Node Tests

In addition to having an axis, each axis step has a node test. The *node test* indicates which of the nodes (by name or node kind) to select, along the specified axis. For example, `child::product` only selects `product` element children of the context node. It does not select other kinds of children (for example, text nodes), or other `product` elements that are not children of the context node.

Node name tests

In previous examples, most of the node tests were based on names, such as `product` and `dept`. These are known as name tests. The syntax of a node name test is shown in Figure 4-2.

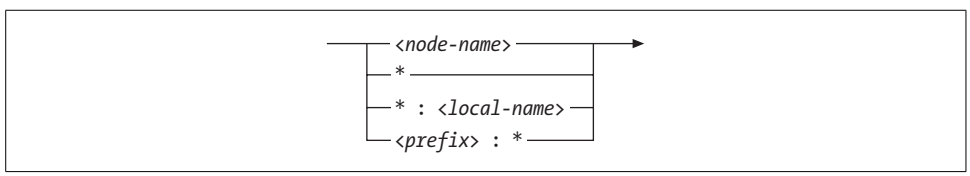


Figure 4-2. Syntax of a node name test

Node name tests and namespaces

Names used in node tests are *qualified names*, meaning that they are affected by namespace declarations. A namespace declaration is in scope if it appears in an outer element, or in the query prolog. The names may be prefixed or unprefixed. If a name is prefixed, its prefix must be mapped to a namespace using a namespace declaration.

If an element name is unprefixed, and there is an in-scope default namespace declared, it is considered to be in that namespace; otherwise, it is in no namespace. Attribute names, on the other hand, are not affected by default namespace declarations.

Use of namespace prefixes in path expressions is depicted in Example 4-1, where the `prod` prefix is first mapped to the namespace, and then used in the steps `prod:product` and `prod:number`. Keep in mind that the prefix is just serving as a proxy for the namespace name. It is not important that the prefixes in the path expressions match the prefixes in the input document; it is only important that the prefixes map to the same namespace. In Example 4-1, you could use the prefix `pr` instead of `prod` in the query, as long as you used it consistently throughout the query.

Example 4-1. Prefixed name tests

```
Input document (prod_ns.xml)
<prod:product xmlns:prod="http://datypic.com/prod">
  <prod:number>563</prod:number>
  <prod:name language="en">Floppy Sun Hat</prod:name>
</prod:product>
Query
declare namespace prod = "http://datypic.com/prod";
<prod:prodList>{
  doc("prod_ns.xml")/prod:product/prod:number
}</prod:prodList>
Results
<prod:prodList xmlns:prod="http://datypic.com/prod">
  <prod:number>563</prod:number>
</prod:prodList>
```

Node name tests and wildcards

You can use wildcards to match names. The step `child::*` (abbreviated simply *) can be used to select all element children, regardless of name. Likewise, `@*` (or `attribute::*`) can be used to select all attributes, regardless of name.

In addition, wildcards can be used for just the namespace and/or local part of a name. The step `prod:*` selects all child elements in the namespace mapped to the prefix `prod`, and the step `*:product` selects all `product` child elements that are in any namespace, or no namespace.

Node kind tests

In addition to the tests based on node name, you can test based on node kind. The syntax of a node kind test is shown in Figure 4-3.

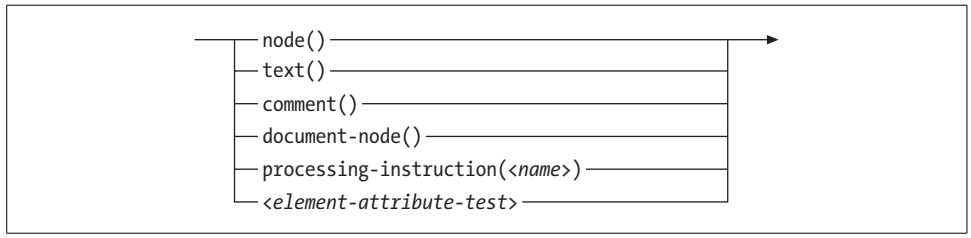


Figure 4-3. Syntax of a node kind test[a]

[a] The detailed syntax of <*element-attribute-test*> is shown in Figure 13-4.

The test `node( )` will retrieve all different kinds of nodes. You can specify `node( )` as the entire step, and it will default to the `child::` axis. In this case, it will bring back child element, text, comment, and processing-instruction nodes (but not attributes, because they are not considered children). This is in contrast to *, which selects child *element* nodes only.

You can also use `node( )` in conjunction with the axes. For example, `ancestor::node( )` returns all ancestor element nodes and the document node (if it exists). This is different from `ancestor::*`, which returns ancestor element nodes only. You can even use `attribute::node( )`, which will return attribute nodes, but this is not often used because it means the same as `@*`.

Four other kind tests, `text( )`, `comment( )`, `processing-instruction( )`, and `document-node( )`, are discussed in Chapter 21.

If you are using schemas, you can also test elements and attributes based on their type using node kind tests. For example, you can specify `element(*, ProductType)` to return all elements whose type is `ProductType`, or `element(product, ProductType)` to return all elements named `product` whose type is `ProductType`. This is discussed further in the section "Sequence Types and Schemas" in Chapter 13.

Abbreviated Syntax

Some axes and steps can be abbreviated, as shown in Table 4-3. The abbreviations "." and ".." are used as the entire step (with no node test). "." represents the current context node itself, regardless of its node kind. Likewise, the step ".." represents the parent node, which could be either an element node or a document node.

Table 4-3. Abbreviations

Abbreviation	Meaning
`.`	`self::node( )`
`..`	`parent::node( )`
`@`	`attribute::`
`//`	`/descendant-or-self::node( )/`

The `@` abbreviation, on the other hand, replaces the axis only, so it is used along with a node test or wildcard. For example, you can use `@dept` to select `dept` attributes, or `@*` to select all attributes.

The // abbreviation is a shorthand to indicate a descendant anywhere in a tree. For example, `catalog//number` will match all `number` elements at any level among the descendants of `catalog`. You can start a path with `.//` if you want to limit the selection to descendants of the current context node.

Table 4-4 shows additional examples of abbreviated and unabbreviated syntax.

Table 4-4. Abbreviated and unabbreviated syntax examples

Unabbreviated syntax	Abbreviated equivalent
`child::product`	`product`
`child::*`	`*`
`self::node( )`	`.`
`attribute::dept`	`@dept`
`attribute::*`	`@*`
`descendant::product`	`.//product`
`child::product/descendant::name`	`product//name`
`parent::node/number`	`../number`

Other Expressions As Steps

In addition to axis steps, other expressions can also be used as steps. You have already seen this in use in:

```
doc("catalog.xml")/catalog/product/number
```

where `doc("catalog.xml")` is a function call that is used as a step. You can include more complex expressions, for example:

```
doc("catalog.xml")/catalog/product/(number | name)
```

which uses the parenthesized expression `(number | name)` to select all `number` and `name` elements. The | operator is a union operator; it selects the union of two sets of nodes.

If the expression in a step contains an operator with lower precedence than /, it needs to be in parentheses. Some other examples of more complex steps are provided in Table 4-5.

Table 4-5. More complex steps (examples start with doc("catalog.xml")/catalog/)

Example	Meaning
`product/(number | name)`	All `number` AND `name` children of `product`.
`product/(* except number)`	All children of `product` except `number`. See "Combining Results" in Chapter 9 for more information on the `|` and `except` operators.
`product/` `(if (desc) then desc else name)`	For each `product` element, the `desc` child if it exists; otherwise, the `name` child.
`product/substring(name,1,30)`	A sequence of `xs:string` values that are substrings of product names.

The last step (and only the last step) in a path may return atomic values rather than nodes. The last example in Table 4-5 will return a sequence of atomic values that are the substrings of the product names. An error is raised if a step that is *not* the last returns atomic values. For example:

```
product/substring(name,1,30)/replace(.,' ','-')
```

will raise an error because the `substring` step returns atomic values, and it is not the last step.

Predicates

Predicates are used in a path expression to filter the results to contain only nodes that meet specific criteria. Using a predicate, you can, for example, select only the elements that have a certain value for an attribute or child element, using a predicate like `[@dept = "ACC"]`. You can also select only elements that *have* a particular attribute child element, using a predicate such as `[color]`, or elements that occur in a particular position within their parent, using a predicate such as `[3]`.

The syntax of a predicate is simply an expression in square brackets ([and]). Table 4-6 shows some examples of predicates.

Table 4-6. Predicates (examples start with doc("catalog.xml")/catalog/)

Example	Meaning
`product[name = "Floppy Sun Hat"]`	All `product` elements that have a `name` child whose value is equal to `Floppy Sun Hat`
`product[number < 500]`	All `product` elements that have a `number` child whose value is less than 500
`product[@dept = "ACC"]`	All `product` elements that have a `dept` attribute whose value is `ACC`
`product[desc]`	All `product` elements that have at least one `desc` child
`product[@dept]`	All `product` elements that have a `dept` attribute
`product[@dept]/number`	All `number` children of `product` elements that have a `dept` attribute

If the expression evaluates to anything other than a number, its effective Boolean value is determined. This means that if it evaluates to the `xs:boolean` value `false`, the number 0 or `NaN`, a zero-length string, or the empty sequence, it is considered false. In most other cases, it is considered true. If the effective Boolean value is true for a particular node, that node is returned. If it is false, the node is not returned.

If the expression evaluates to a number, it is interpreted as the position as described in "Using Positions in Predicates" later in this chapter.

As you can see from the last example, the predicate is not required to appear at the end of the path expression; it can appear at the end of any step.

Note that `product[number]` is different from `product/number`. While both expressions filter out products that have no `number` child, in the former expression, the `product` element is returned. In the latter case, the `number` element is returned.

Comparisons in Predicates

The examples in the previous section use general comparison operators like `=` and `<`. You can also use the corresponding value comparison operators, such as `eq` and `lt`, but you should be aware of the difference. Value comparison operators only allow a single value, while general comparison operators allow sequences of zero, one, or more values. Therefore, the path expression:

```
doc("prices.xml")//priceList[@effDate eq '2006-11-15']
```

is acceptable, because each `priceList` element can have only one `effDate` attribute. However, if you wanted to find all the `priceList` elements that contain the product 557, you might try the expression:

```
doc("prices.xml")//priceList[prod/@num eq 557]
```

This will raise an error because the expression `prod/@num` returns more than one value per `priceList`. By contrast:

```
doc("prices.xml")//priceList[prod/@num = 557]
```

returns a `priceList` if it has *at least one* `prod` child whose `num` attribute is equal to 557. It might have other `prod` children whose numbers are not equal to 557.

In both cases, if a particular `priceList` does not have any `prod` children with `num` attributes, it does not return that `priceList`, but it does not raise an error.

Another difference is that value comparison operators treat all untyped data like strings. If we fixed the previous problem with `eq` by returning `prod` nodes instead, as in:

```
doc("prices.xml")//priceList/prod[@num eq 557]
```

it would still raise an error if no schema were present, because it treats the `num` attribute like a string, which can't be compared to a number. The `=` operator, on the other hand, will cast the value of the `num` attribute to `xs:integer` and then compare it to 557, as you would expect.

For these reasons, general comparison operators are easier to use than value comparison operators in predicates when children are untyped or repeating. The down side of general comparison operators is that they also make it less likely that the processor will catch any mistakes you make. In addition, they may be more expensive to evaluate because it's harder for the processor to make use of indexes.

Using Positions in Predicates

Another use of predicates is to specify the numeric position of an item within the sequence of items currently being processed. These are sometimes called, predictably, *positional predicates*. For example, if you want the fourth product in the catalog, you can specify:

```
doc("catalog.xml")/catalog/product[4]
```

Any predicate expression that evaluates to an integer will be considered a positional predicate. If you specify a number that is greater than the number of items in the context sequence, it does not raise an error; it simply does not return any nodes. For example:

```
doc("catalog.xml")/catalog/product[99]
```

returns the empty sequence.

Understanding positional predicates

With positional predicates, it is important to understand that the position is the position within the current sequence of items being processed, not the position of an element relative to its parent's children. Consider the expression:

```
doc("catalog.xml")/catalog/product/name[1]
```

This expression refers to the first `name` child of each `product`; the step `name[1]` is evaluated once for every `product` element. It does not necessarily mean that the `name` element is the first child of `product`.

It also does not return the first `name` element that appears in the document as a whole. If you wanted just the first `name` element in the document, you could use the expression:

```
(doc("catalog.xml")/catalog/product/name)[1]
```

because the parentheses change the order of evaluation. First, all the `name` elements are returned; then, the first one of those is selected. Alternatively, you could use:

```
doc("catalog.xml")/catalog/descendant::name[1]
```

because the sequence of descendants is evaluated first, then the predicate is applied. However, this is different from the abbreviated expression:

```
doc("catalog.xml")/catalog//name[1]
```

which, like the first example, returns the first name child of each of the products. That's because it's an abbreviation for:

```
doc("catalog.xml")/catalog/descendant-or-self::node( )/name[1]
```

The position and last functions

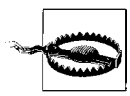
The position and last functions are also useful when writing predicates based on position. The position function returns the position of the context item within the context sequence (the current sequence of items being processed). The function takes no arguments and returns an integer representing the position (starting with 1, not 0) of the context item. For example:

```
doc("catalog.xml")/catalog/product[position( ) < 3]
```

returns the first two product children of catalog. You could also select the first two children of each product, with any name, using:

```
doc("catalog.xml")/catalog/product/*[position( ) < 3]
```

by using the wildcard *. Note that the predicate [position( ) = 3] is equivalent to the predicate [3], so the position function is not very useful in this case.



When using positional predicates, you should be aware that the to keyword does not work as you might expect when used in predicates. If you want the first three products, it may be tempting to use the syntax:

```
doc("catalog.xml")/catalog/product[1 to 3]
```

However, this will raise an error* because the predicate evaluates to multiple numbers instead of a single one. You can, however, use the syntax:

```
doc("catalog.xml")/catalog/product[position( ) = (1 to 3)]
```

You can also use the subsequence function to limit the results based on position, as in:

```
doc("catalog.xml")/catalog/subsequence(product, 1, 3)
```

The last function returns the number of nodes in the current sequence. It takes no arguments and returns an integer representing the number of items. The last function is useful for testing whether an item is the last one in the sequence. For example, catalog/product[last( )] returns the last product child of catalog.

Table 4-7 shows some examples of predicates that use the position of the item. The descriptions assume that there is only one catalog element, which is the case in the catalog.xml example.

* Although several implementations erroneously support this construct.

Table 4-7. Position in predicates (examples start with doc("catalog.xml")/catalog/)

Example	Description
`product[2]`	The second `product` child of `catalog`
`product[position( ) = 2]`	The second `product` child of `catalog`
`product[position( ) > 1]`	All `product` children of `catalog` after the first one
`product[last( )-1]`	The second to last `product` child of `catalog`
`product[last( )]`	The last `product` child of `catalog`
`*[2]`	The second child of `catalog`, regardless of name
`product[3]/*[2]`	The second child of the third `product` child of `catalog`

In XQuery, it's very unusual to use the `position` or `last` functions anywhere except within a predicate. It's not an error, however, as long as the context item is defined. For example, `a/last( )` returns the same number as `count(a)`.

Positional predicates and reverse axes

Oddly, positional predicates have the opposite meaning when using reverse axes such as `ancestor`, `ancestor-or-self`, `preceding`, or `preceding-sibling`. These axes, like all axes, return nodes in document order. For example, the expression:

```
doc("catalog.xml")//i/ancestor::*
```

returns the ancestors of the `i` element in document order, namely the `catalog` element, followed by the fourth `product` element, followed by the `desc` element. However, if you use a positional predicate, as in:

```
doc("catalog.xml")//i/ancestor::*[1]
```

you might expect to get the `catalog` element, but you will actually get the nearest ancestor, the `desc` element. The expression:

```
doc("catalog.xml")//i/ancestor::*[last( )]
```

will give you the `catalog` element.

Using Multiple Predicates

Multiple predicates can be chained together to filter items based on more than one constraint. For example:

```
doc("catalog.xml")/catalog/product[number < 500][@dept = "ACC"]
```

selects only `product` elements with a number child whose value is less than 500 *and* whose `dept` attribute is equal to `ACC`. This can also be equivalently expressed as:

```
doc("catalog.xml")/catalog/product[number < 500 and @dept = "ACC"]
```

It is sometimes useful to combine the positional predicates with other predicates, as in:

```
doc("catalog.xml")/catalog/product[@dept = "ACC"][2]
```

which represents "the second product child that has a dept attribute whose value is ACC," namely the third product element. The order of the predicates is significant. If the previous example is changed to:

```
doc("catalog.xml")/catalog/product[2][@dept = "ACC"]
```

it means something different, namely "the second product child, if it has a dept attribute whose value is ACC." This is because the predicate changes the context, and the context node for the second predicate in this case is the second product element.

More Complex Predicates

So far, the examples of predicates have been simple path expressions, comparison expressions, and numbers. In fact, any expression is allowed in a predicate, making it a very flexible construct. For example, predicates can contain function calls, as in:

```
doc("catalog.xml")/catalog/product[contains(@dept, "A")]
```

which returns all product children whose dept attribute contains the letter A. They can contain conditional expressions, as in:

```
doc("catalog.xml")/catalog/product[if ($descFilter)
                                   then desc else true()]
```

which filters product elements based on their desc child only if the variable $descFilter is true. They can also contain expressions that combine sequences, as in:

```
doc("catalog.xml")/catalog/product[* except number]
```

which returns all product children that have at least one child other than number. General comparisons with multiple values can be used, as in:

```
doc("catalog.xml")/catalog/product[@dept = ("ACC", "WMN", "MEN")]
```

which returns products whose dept attribute value is any of those three values. This is similar to a SQL "in" clause.

To retrieve every third product child of catalog, you could use the expression:

```
doc("catalog.xml")/catalog/product[position() mod 3 = 0]
```

because it selects all the products whose position is divisible by 3.

Predicates can even contain path expressions that themselves have predicates. For example:

```
doc("catalog.xml")/catalog/product[*[3][self::colorChoices]]
```

can be used to find all product elements whose third child element is colorChoices. The *[3][self::colorChoices] is part of a separate path expression that is itself within a predicate. *[3] selects the third child element of product, and [self::colorChoices] is a way of testing the name of the current context element.

Predicates are not limited to use with path expressions. They can be used with any sequence. For example:

```
(1 to 100)[. mod  5 = 0]
```

can be used to return the integers from 1 to 100 that are divisible by 5. Another example is:

```
(@price, 0.0)[1]
```

which selects the `price` attribute if it exists, or the decimal value `0.0` otherwise.

Dynamic Paths

It is a common requirement that the paths in your query will not be static but will instead be calculated based on some input to the query. For example, if you want to provide users with a search capability where they choose the elements in the input document to search, you can't use a static path in your query. XQuery does not provide any built-in support for evaluating dynamic paths, but you do have a couple of alternatives.

For simple paths, it is easy enough to test for an element's name using the `name` function instead of including it directly as a step in the path. For example, if the name of the element to search and its value are bound to the variables `$elementName` and `$searchValue`, you can use a path like:

```
doc("catalog.xml")//*[name( ) = $elementName][. = $searchValue]
```

If the dynamic path is more complex than a simple element or attribute name, you can use an implementation-specific function. Most XQuery implementations provide a function for dynamic evaluation of paths or entire queries. For example, in Saxon, it is the `saxon:evaluate` function, while in Mark Logic it is called `xdmp:eval`. In Saxon, I could use the following expression to get the same results as the previous example:

```
saxon:evaluate(concat('doc("catalog.xml")//',$elementName,
                      '[. = "',$searchValue,'"]'))
```

Input Documents

A single query can access many input documents. The term *input document* is used in this book to mean any XML data that is being queried. Technically, it might not be an entire XML document; it might be a document fragment, such as an element or sequence of elements, possibly with children. Alternatively, it might not be a physical XML file at all; it might be data retrieved from an XML database, or an in-memory XML representation that was generated from non-XML data.

If the input document is physically stored in XML syntax, it must be well-formed XML. This means that it must comply with XML syntax rules, such as that every start tag has an end tag, there is no overlap among elements, and special characters are used appropriately. It must also use namespaces appropriately. This means that if colons are used in element or attribute names, the part before the colon must be a prefix that is mapped to a namespace using a namespace declaration.

Whether it is physically stored as an XML document or not, an input document must conform to other constraints on XML documents. For example, an element may not have two attributes with the same name, and element and attribute names may not contain special characters other than dashes, underscores, and periods.

There are four ways that input documents could be accessed from within a query. They are described in the next four sections.

Accessing a Single Document

The `doc` function can be used to open one input document based on its URI. It takes as an argument a single URI as a string, and returns the document node of the resource associated with the specified URI.

Implementations interpret the URI passed to the `doc` function in different ways. Some, like Saxon, will dereference the URI, that is, go out to the URL and retrieve the resource at that location. For example, using Saxon:

```
doc("http://datypic.com/order.xml")
```

will return the document node of the document that can be found at the URL `http://datypic.com/order.xml`.

Other implementations, such as those embedded in XML databases, consider the URIs to be just names. The processor might take the name and look it up in an internal catalog to find the document associated with that name. The `doc` function is covered in detail in Appendix A.

Accessing a Collection

The `collection` function returns the nodes that make up a collection. A collection may be a sequence of nodes of any kind, identified by a URI. Exactly how the URI is associated with the nodes is defined by the implementation. For example, one implementation might accept a URI that is the name of a directory on a filesystem, and return the document nodes of the XML documents stored in that directory. Another implementation might associate a URI with a particular database. A third might allow you to specify the URI of an XML document that contains URIs for all the XML documents in the collection.

The function takes as an argument a single URI. For example, the function call:

```
collection("http://datypic.com/orders")
```

might return all the document nodes of the XML documents associated with the collection `http://datypic.com/orders`. It is also possible to use the function without any parameters, as in `collection()`, to retrieve a default collection as defined by the implementation.

Some XQuery implementations support a function called `input`, with no arguments. This function appeared in earlier drafts of the XQuery recommendation but is no longer part of the standard. It is equivalent to calling the `collection` function with no arguments.

Setting the Context Node Outside the Query

The context node can be set by the processor outside the query. In this case, it may not be necessary to use the `doc` or `collection` functions, unless you want to open secondary data sources.

For example, a hypothetical XQuery implementation might allow you to set the context node in the Java code that executes the query, as in:

```
Document catalogDocument = new Document(File("catalog.xml"));
String query = "catalog/product[@dept = 'ACC']";
List productElements = catalogDocument.evaluateXQuery(query);
```

In that case, the XQuery expression `catalog/product` might be evaluated in the context of the catalog document node itself. If the processor had not set the context node, a path expression starting with `catalog/product` would not be valid.

Another implementation might allow you to select a document to query in a user interface, in which case it uses that document as the context node.

Using Variables

The processor can bind external variables to input documents or document fragments. These variables can then be used in the query to access the input document. For example, an implementation might allow an external variable named `$input` to be defined, and allow the user to specify a document to be associated with that variable. The hypothetical query processor could be invoked from the command line using:

```
xquery -input catalog.xml
```

and the query could use expressions like `$input/catalog/product` to retrieve the `product` elements. The name `$input` is provided as an example; the implementation could use any name for the variable.

You should consult the documentation for your XQuery implementation to determine which of these four methods are appropriate for accessing input documents.

A Closer Look at Context

The processor can set the context node outside the query. Alternatively, the context node can be set by an outer expression. In XQuery, the *only* operators that change the context node are the slash and the square brackets used in predicates.* For example:

```
doc("catalog.xml")/catalog/product/(if (desc) then desc else name)
```

In this case, the if expression uses the paths `desc` and `name`. Because it is entirely contained in one step of another (outer) path expression, it is evaluated with the context node being the `product` element. Therefore, `desc` and `name` are tested as children of `product`.

In some cases, no context node is defined. This might occur if the processor does not set the context node outside the scope of the query, as described earlier in "Setting the Context Node Outside the Query" and there is no outer expression that sets the context. In addition, the context node is never defined inside the body of a function. In these cases, using a relative path such as `desc` raises an error.

Working with the Context Node

It is sometimes useful to be able to reference the context node, either in a step or in a predicate. A prior example retrieved `product` elements whose `number` child is less than 500 using the expression:

```
doc("catalog.xml")/catalog/product[number < 500]
```

Suppose, instead, you want to retrieve the `number` child itself. You can do this using the expression:

```
doc("catalog.xml")/catalog/product/number[. < 500]
```

The period (.) is used to represent the context node itself in predicates and in paths. You can also use the period as a parameter to functions, as in:

```
doc("catalog.xml")/catalog/product/name[starts-with(., "T")]
```

which passes the context item to the `starts-with` function. Some functions, when they are not passed any arguments, automatically use the context node. For example:

```
doc("catalog.xml")/catalog/product/desc[string-length( ) > 20]
```

uses the `string-length` function to test the length of the `desc` value. It was not necessary to pass the "." to the `string-length` function. This is because the defined behavior of this particular function is such that if no argument is passed to the function, it defaults to the context node.

* This is in contrast to XSLT, where several kinds of expressions change the context node, including the `xsl:for-each` element and template matching.

Accessing the Root

When the context node is part of a complete XML document, the root is a document node (*not* the outermost element). However, XQuery also allows nodes to participate in tree fragments, which can be rooted at any kind of node.

There are several ways of accessing the root of the current context node. When a path expression starts with one forward slash, as in:

```
/catalog/product
```

the path is evaluated relative to the root of the tree containing the current context node. For example, if the current context node is a `number` element in the `catalog.xml` document, the path `/catalog/product` retrieves all `product` children of `catalog` in `catalog.xml`.

When a path expression starts with two forward slashes, as in:

```
//product/number
```

it is referring to any `product` element in the tree containing the current context node. Starting an expression with / or // is allowed only if the current context node is part of a complete XML document (with a document node at its root). / can also be used as an expression in its own right, to refer to the root of the tree containing the context node (provided this is a document node).

The `root` function also returns the root of the tree containing a node. It can be used in conjunction with path expressions to find siblings and other elements that are in the same document. For example, `root($myNode)//product` retrieves all `product` elements that are in the same document (or document fragment) as `$myNode`. When using the `root` function, it's *not* necessary for the tree to be rooted at a document node.