XQuery: An XML query language

by D. Chamberlin

The World Wide Web Consortium has convened a working group to design a query language for Extensible Markup Language (XML) data sources. This new query language, called XQuery, is still evolving and has been described in a series of drafts published by the working group. XQuery is a functional language comprised of several kinds of expressions that can be nested and composed with full generality. It is based on the type system of XML Schema and is designed to be compatible with other XML-related standards. This paper explains the need for an XML query language, provides a tutorial overview of XQuery, and includes several examples of its use.

Increasingly, Extensible Markup Language (XML)[1] is considered the format of choice for the exchange of information among various applications on the Internet. The popularity of XML is due in large part to its flexibility for representing many kinds of information. The use of tags makes XML data self-describing, and the extensible nature of XML makes it possible to define new kinds of documents for specialized purposes. As the importance of XML has increased, a series of standards has grown up around it, many of which were defined by the World Wide Web Consortium (W3C).[2] For example, XML Schema[3] provides a notation for defining new types of elements and documents; XML Path Language (XPath)[4] provides a notation for selecting elements within an XML document; and Extensible Stylesheet Language Transformations (XSLT)[5] provides a notation for transforming XML documents from one representation to another.

XML makes it possible for applications to exchange data in a standard format that is independent of storage. For example, one application may use a native XML storage format, whereas another may store data in a relational database. Since XML is emerging as a standard for data exchange, it is natural that queries among applications should be expressed as queries against data in XML format. This use gives rise to a requirement for a query language designed expressly for XML data sources. In October 1999, W3C convened the XML Query Working Group[6] for the purpose of designing such a query language, to be called XQuery.

XML data are different from relational data in several important respects that influence the design of a query language. Relational data tend to have a regular structure, which allows the descriptive meta-data for these data to be stored in a separate catalog. XML data, in contrast, are often quite heterogeneous, and distribute their meta-data throughout the document. XML documents often contain many levels of nested elements, whereas relational data are "flat." XML documents have an intrinsic order, whereas relational data are unordered except where an ordering can be derived from data values. Relational data are usually "dense" (nearly every column has a value), and relational systems often represent missing information by a special null value. XML data, in contrast, are often "sparse" and can represent missing information simply by the absence of an element. For these and other reasons, existing relational query languages are not directly suitable for querying XML data.

The design of XQuery is still in progress. The XML Query Working Group has published working drafts of several documents that describe the current state of the design. Of these, perhaps the most important is *XQuery 1.0: An XML Query Language*,[7] which contains a syntax and informal description of the language. The working group has also published a list of requirements,[8] a description of the data model that underlies the language,[9] a formal semantic description,[10] a list of functions and operators,[11] and a collection of use cases that illustrate applications of the language.[12] Each of these documents is updated from time to time as the design of XQuery evolves. This paper is based on the most recent XQuery design at the time of its publication, but since this design is still changing, the documents referenced in this paragraph should be consulted for the latest developments.

The design of XQuery has been subject to a number of influences.

The design of XQuery has been subject to a number of influences. Perhaps the most important of these is compatibility with existing W3C standards, including Schema, XSLT, XPath, and XML itself. XPath, in particular, is so important and so closely related that XQuery is defined as a superset of XPath. The overall design of XQuery is based on a language proposal called Quilt.[13] Quilt, in turn, was influenced by the functional approach of Object Query Language (OQL),[14] by the keyword-based syntax of Structured Query Language (SQL),[15] and by previous XML query language proposals including XQL,[16] XML-QL,[17] and Lorel.[18]

It is an objective of the XML Query Working Group to define two syntaxes for XQuery: one that is expressed in XML, and one that is optimized for human writing and understanding. This paper describes only the human-oriented version of XQuery.

The initial design of XQuery is focused only on information retrieval and does not provide facilities for updating existing XML documents. The XML Query Working Group may consider the addition of an update facility after completing the design of the first version of XQuery.

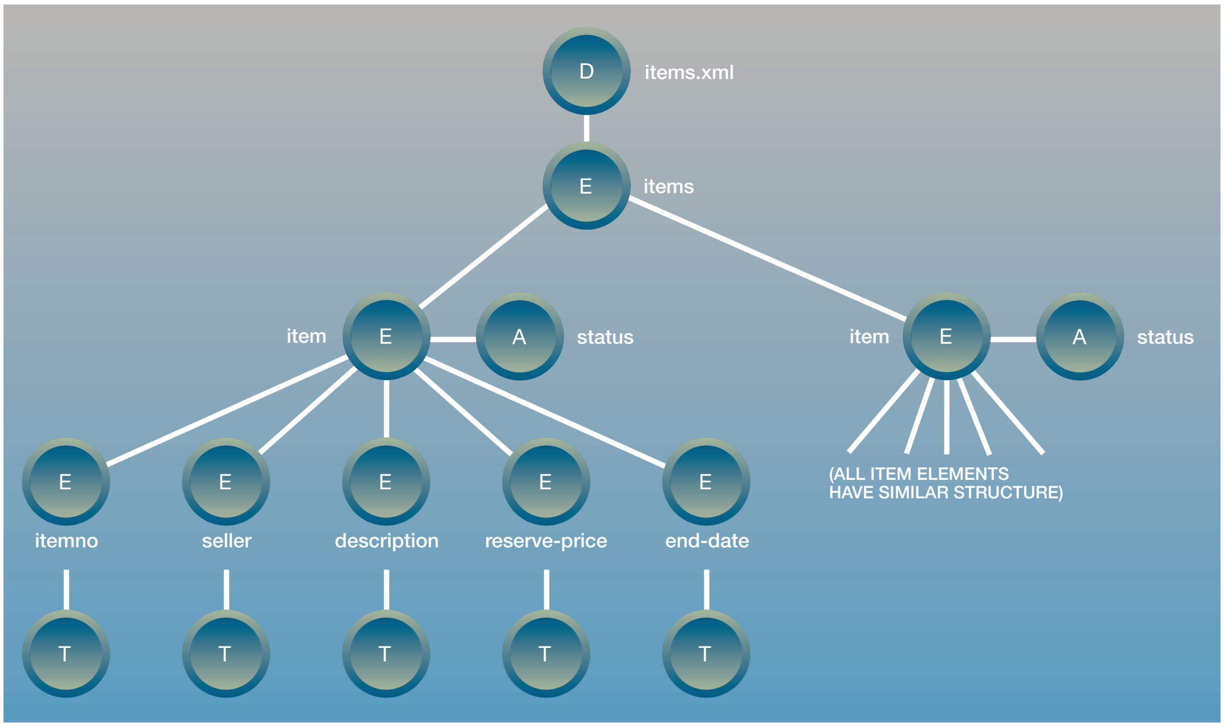
This paper describes the data model on which XQuery is based, and then presents an overview of the XQuery language in the form of a series of examples. This paper is not intended to provide a rigorous or exhaustive definition of the language. The reader is referred to Reference 7 for an XQuery syntax and a more complete language description.

Data model

Formally, the input and output of XQuery are defined in terms of a data model, described in Reference 9. The query data model provides an abstract representation of one or more XML documents or document fragments. The data model is based on the notion of a sequence. A *sequence* is an ordered collection of zero or more items. An *item* may be a node or an atomic value. An *atomic value* is an instance of one of the built-in data types defined by XML Schema, such as strings, integers, decimals, and dates. A *node* conforms to one of seven node kinds, which include element, attribute, text, document, comment, processing instruction, and namespace nodes. A node may have other nodes as children, thus forming one or more node hierarchies. Some kinds of nodes, such as element and attribute nodes, have names or typed values, or both. A *typed value* is a sequence of zero or more atomic values. Nodes have identity (that is, two nodes may be distinguishable even though their names and values are the same), but atomic values do not have identity. Among all the nodes in a hierarchy there is a total ordering called *document order*, in which each node appears before its children. Document order corresponds to the order in which the nodes would appear if the node hierarchy were represented in XML format. Document order between nodes in different hierarchies is implementation-defined but must be consistent; that is, all the nodes in one hierarchy must be ordered either before or after all the nodes in another hierarchy.

Sequences may be heterogeneous; that is, they may contain mixtures of various types of nodes and atomic

Figure 1   Data model representation of items.xml



values. However, a sequence never appears as an item in another sequence. All operations that create sequences are defined to "flatten" their operands so that the result of the operation is a single-level sequence. There is no distinction between an item and a sequence of length one—in other words, a node or atomic value is considered to be identical to a sequence of length one containing that node or atomic value.

Sequences of length zero are valid and are sometimes used to represent missing or unknown information, in much the same way that null values are used in relational systems.

In addition to sequences, the query data model defines a special value called the *error value*, which is the result of evaluating an expression that contains an error. An error value may not be combined in a sequence with any other value.

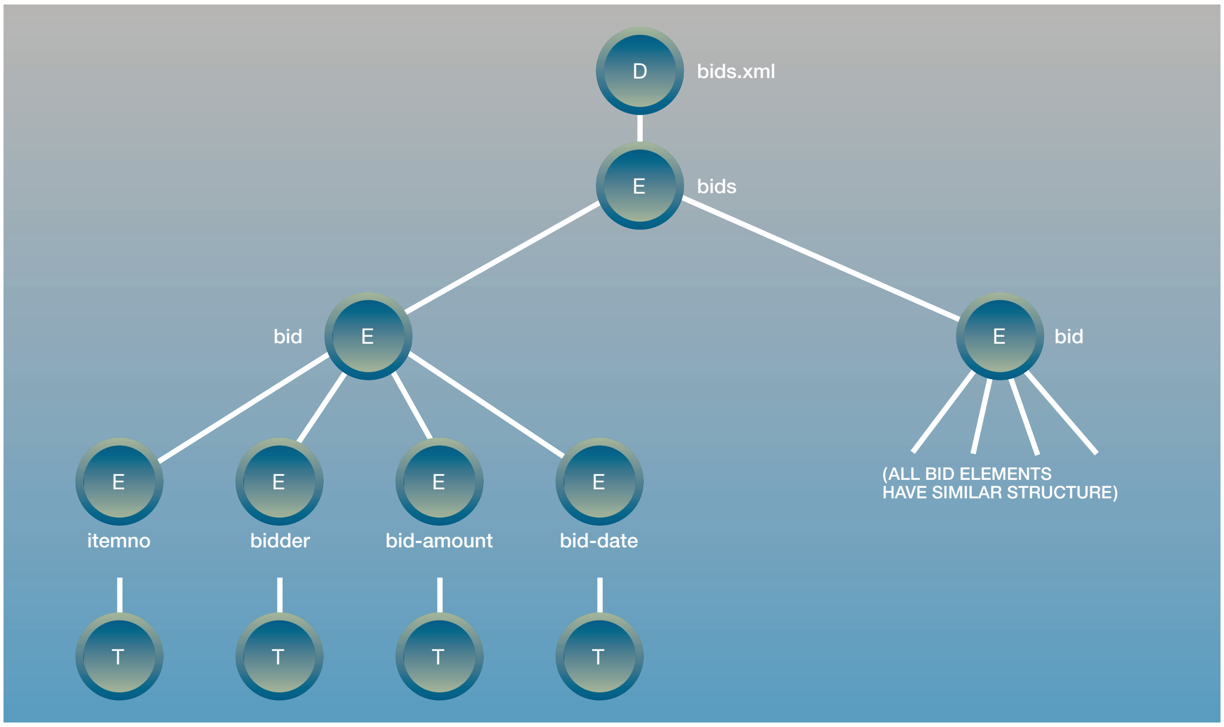
Input XML documents can be transformed into the query data model by a process called *schema validation*, which parses the document, validates it against a particular schema, and represents it as a hierarchy of nodes and atomic values, labeled with type information derived from the schema. If an input document does not have a schema, it is validated against a permissive default schema that assigns generic types—nodes are labeled `anyType` and atomic values are labeled `anySimpleType`. The process of schema validation is described in more detail in Reference 3.

The result of a query may be transformed from the query data model into an XML representation by a process called *serialization*. The details of serialization are beyond the scope of this paper. It is worth noting that the result of a query is not always a well-formed XML document. For example, a query might return an atomic value such as the number 47, or a sequence of elements with no common parent.

Example data

To illustrate the query data model and provide a basis for later examples, we consider a small XML database that contains data from an on-line auction,

Figure 2 Data model representation of bids.xml



based loosely on Use Case R in Reference 12. The database consists of two XML documents named `items.xml` and `bids.xml`.

The `items.xml` document contains a root element named `items`, which in turn contains an `item` element for each item currently for sale at the auction. Each `item` element has a `status` attribute and subelements named `itemno`, `seller`, `description`, `reserve-price`, and `end-date`. The `reserve-price` element names a minimum selling price set by the owner, and the `end-date` element indicates the ending date of the auction.

The `bids.xml` document contains a root element named `bids`, which in turn contains a `bid` element for each bid that has been placed for an item. Each `bid` element has subelements named `itemno`, `bidder`, `bid-amount`, and `bid-date`.

Figures 1 and 2 show the data model representations of the `items.xml` and `bids.xml` documents, respectively (including only a representative item and a representative bid). In the figures, the circles labeled D, E, A, and T represent document, element, attribute, and text nodes, respectively.

Expressions

We now describe expressions in XQuery.

Basics. Like XML and XPath, XQuery is a case-sensitive language, and all its keywords are made up of lowercase characters. Detailed rules for lexing and parsing XQuery are described in Reference 7. Characters enclosed between "`{--`" and "`--}`" are considered to be comments and are ignored during query processing (except, of course, inside a quoted string, where they are considered to be part of the string).

XQuery is a functional language, which means that it is made up of expressions that return values and do not have side effects. XQuery has several kinds of expressions, most of which are composed from lower-level expressions, combined by operators or keywords. XQuery expressions are fully composable, that is, where an expression is expected, any kind of expression may be used. As noted earlier, the value of an expression, in general, is a heterogeneous sequence of nodes and atomic values.

The simplest kind of XQuery expression is a *literal*, which represents an atomic value. The following are several examples of literals:

`47`	is a literal of type `integer`
`4.7`	is a literal of type `decimal` because it contains a decimal point
`4.7E3`	is a literal of type `double` because it contains an exponent
`"47"`	is a literal of type `string` (single quotes are allowed inside double-quoted strings)
`'47'`	is a literal of type `string` (double quotes are allowed inside single-quoted strings)

Atomic values of other types may be created by calling constructors. A *constructor* is a function that creates a value of a particular type from a string containing a lexical representation of the desired type. In general, a constructor has the same name as the type it constructs. The following example uses a constructor to create a value of type `date`:

```
date("2002-5-31")
```

Any XQuery expression may be enclosed in parentheses. Parentheses are useful for making explicit the order in which an expression should be evaluated. The following examples of arithmetic expressions show how parentheses can be used to control the precedence of operators. Arithmetic expressions are discussed in more detail in a later subsection.

`(2 + 4) * 5`	has the value 30 because the subexpression `(2 + 4)` is evaluated first
`2 + 4 * 5`	has the value 22 because * has a higher precedence than +

The comma operator concatenates two values to form a sequence. Sequences are often enclosed in parentheses as explicit delimiters, although this is not required. An empty pair of parentheses denotes an empty sequence. Since sequences cannot be nested, the comma operator constructs a sequence consisting of all the items in its left operand, followed by all the items in its right operand. A sequence can also be constructed by the `to` operator, which returns a sequence consisting of all the integers between its left operand and its right operand, inclusive. The following examples illustrate construction of sequences:

`1, 2, 3`	is a sequence of three values
`(1, 2, 3)`	is identical to `1, 2, 3`
`((1, 2), (), 3)`	is identical to `1, 2, 3`
`1 to 3`	is identical to `1, 2, 3`

A *variable* in XQuery is a name that begins with a dollar sign. A variable may be bound to a value and used in an expression to represent that value. One way to bind a variable is by means of a LET expression, which binds one or more variables and then evaluates an inner expression. The value of the LET expression is the result of evaluating the inner expression with the variables bound. The following example illustrates a LET expression that returns the sequence `1, 2, 3`:

```
let $start := 1, $stop := 3
return $start to $stop
```

A LET expression is a special case of a FLWR (for, let, where, return) expression, which provides additional ways to bind variables. FLWR expressions are described in more detail later.

Another simple form of XQuery expression is a *function call*. XQuery provides a core function library, described in Reference 11, and a mechanism whereby users can define additional functions, described in the next section. Function calls in XQuery employ the usual notation in which the arguments of the function are enclosed in parentheses. The following example calls the core library function `substring` to extract the first six characters from a string:

```
substring("Martha Washington", 1, 6)
```

Path expressions. Path expressions in XQuery are based on the syntax of XPath.[4] A path expression consists of a series of steps, separated by the slash character ("`/`"). The result of each step is a sequence of nodes. The value of the path expression is the node sequence that results from the last step in the path.

Each step is evaluated in the context of a particular node, called the *context node*. In general, a step can be any expression that returns a sequence of nodes. One important kind of step, called an *axis step*, can be thought of as beginning at the context node and moving through the node hierarchy in a particular direction, called an *axis*. As the axis step moves along the designated axis, it selects nodes that satisfy a selection criterion. The selection criterion can select nodes based on their names, their positions with respect to the context node, or a predicate based on the value of a node. XPath defines 13 axes, and some or all of them will be supported by XQuery as well. Current plans are for XQuery to support the six axes named `child`, `descendant`, `parent`, `attribute`, `self`, and `descendant-or-self`.

As a path expression is evaluated, the nodes selected by each step serve in turn as context nodes for the following step. If a step has several context nodes, it is evaluated for each of the context nodes in turn, and the resulting node sequences are combined by the `union` operator to form the result of the step. The result of a step is always a sequence of distinct nodes (without duplicates based on node identity), in document order.

Path expressions may be written in either unabbreviated syntax or abbreviated syntax. The unabbreviated syntax for an axis step consists of an axis and a selection criterion, separated by two colons. Q1 illustrates a four-step path expression using unabbreviated syntax. The first step invokes the built-in `document` function, which returns the document node for the document named `items.xml`. The second step is an axis step that finds all children of the document node ("*" selects all nodes on the given axis, which in this case is only a single element node named `items`). The third step follows the `child` axis again to find all the child elements at the next level that are named `item` and that in turn have a child named `seller` with the value "Smith." The result of the third step is a sequence of `item` element nodes. Each of these `item` nodes is used in turn as the context node for the fourth step, which follows the `child` axis again to find the `description` elements that are children of the given `item`. The final result of the path expression is the result of the fourth step: a sequence of `description` element nodes, in document order.

(Q1) List the descriptions of all items offered for sale by Smith.

```
document("items.xml")/child::*
   /child::item[child::seller = "Smith"]
   /child::description
```

In practice, path expressions are usually written using abbreviated syntax. Several kinds of abbreviations are provided. Perhaps the most important of these is that the axis specifier may be omitted when the `child` axis is used. Since `child` is the most commonly used axis, this abbreviation is helpful in reducing the length of many path expressions. For example, Q1 may be abbreviated as follows:

```
document("items.xml")
   /*/item[seller = "Smith"]/description
```

When two steps are separated by a double slash rather than by a single slash, it means that the second step may traverse multiple levels of the hierarchy, using the `descendants` axis rather than the single-level `child` axis. For example, Q2 searches for `description` elements that are descendants of the root node of a given document. The result of Q2 is a sequence of element nodes that could, in principle, have been found at various levels of the node hierarchy (though, in our sample document, all `description` nodes are found at the same level).

(Q2) List all description elements found in the document items.xml.

```
document("items.xml")//description
```

Within a path expression, a single dot (" . ") refers to the context node, and two consecutive dots (" .. ") refer to the parent of the context node. These notations are abbreviated invocations of the `self` and `parent` axes, respectively. Names found in path expressions are usually interpreted as names of element nodes; however, a name prefixed by the "@" character is interpreted as the name of an attribute node. This is an abbreviation for a step that traverses the `attribute` axis. These abbreviations are illustrated by Q3, which begins at the node that is bound to the variable `$description`, traverses the `parent` axis to the parent `item` node, and then traverses the `attribute` axis to find an attribute named `status`. The result of Q3 is a single attribute node.

(Q3) Find the status attribute of the item that is the parent of a given description.

```
$description/../@status
```

Predicates. In XQuery, a *predicate* is an expression, enclosed in square brackets, that is used to filter a sequence of values. Predicates are often used in the steps of a path expression. For example, in the step `item[seller = "Smith"]`, the phrase `seller = "Smith"` is a predicate that is used to select certain `item` nodes and discard others. We will refer to the items in the sequence being filtered by a predicate as candidate items. The predicate is evaluated for each candidate item, using the candidate item as the context item for evaluating the predicate expression. The term "context item" is a generalization of the term "context node" and may indicate either a node or an atomic value. Within a predicate expression, a single dot (" . ") indicates the context item. Each candidate item is selected or discarded according to the following rules.

If the predicate expression evaluates to a Boolean value, the candidate item is selected if the value of the predicate expression is `true`. This type of predicate is illustrated by the following example, which selects item nodes that have a `reserve-price` child node whose value is greater than 1000:

```
item[reserve-price > 1000]
```

If the predicate expression evaluates to a number, the candidate item is selected if its ordinal position in the list of candidate items is equal to the number. This type of predicate is illustrated by the following example, which selects the fifth item node on the child axis:

```
item[5]
```

If the predicate expression evaluates to an empty sequence, the candidate item is discarded, but if the predicate expression evaluates to a sequence containing at least one node, the candidate item is selected. This form of predicate can be used to test for the existence of a child node that satisfies some condition. This is illustrated by the following example, which selects `item` nodes that have a `reserve-price` child node, regardless of its value:

```
item[reserve-price]
```

Several different kinds of operators and functions are often used inside predicates. In the following six paragraphs, some of the commonest and most useful of these operators and functions are described.

Value comparison operators: `eq`, `ne`, `lt`, `le`, `gt`, `ge`. These operators can compare two scalar values, but they raise an error if either operand is a sequence of length greater than one. If either operand is a node, the value comparison operator extracts its value before performing the comparison. For example, `item[reserve-price gt 1000]` selects an `item` node if it has exactly one `reserve-price` child node whose value is greater than 1000.

General comparison operators: =, !=, >, >=, <, <=. These operators can deal with operands that are sequences, providing implicit "existential" semantics for both operands. Like the value comparison operators, the general comparison operators automatically extract values from nodes. For example, `item[reserve-price > 1000]` selects an item node if it has at least one `reserve-price` child node whose value is greater than 1000.

A predicate is an expression, enclosed in square brackets, that is used to filter a sequence of values.

Node comparison operators: `is` and `isnot`. These operators compare the identities of two nodes. For example, `$node1 is $node2` is true if the variables `$node1` and `$node2` are bound to the same node (that is, the node identity is the same for both variables).

Order comparison operators: These operators compare the positions of two nodes. For example, `$node1 << $node2` is true if the node bound to `$node1` occurs earlier in document order than the node bound to `$node2`.

Logical operators: `and` and `or` operators can be used to combine logical conditions inside a predicate. For example, the following predicate selects `item` nodes that have exactly one `seller` child element with the value "Smith," and also have at least one `reserve-price` child element with any value: `item[seller eq "Smith" and reserve-price]`.

Negation: `not` is a function rather than an operator. It serves to invert a Boolean value, turning `true` into `false` and `false` into `true`. The following step uses the `not` function with an existence test to find `item` nodes that have no `reserve-price` child element: `item[not(reserve-price)]`.

In all of the above examples, element and attribute names have been simple identifiers. However, the XML Namespace recommendation[19] allows elements and attributes to have two-part names in which the first part is a namespace prefix, followed by a colon. A name qualified by a namespace prefix is called a *QName*. Each namespace prefix must be bound to a URI (uniform resource identifier) that uniquely identifies a namespace. This convention allows each application to define names in its own namespace without danger of colliding with names defined by other applications, and it allows a query to unambiguously refer to names defined by various applications. If the prefix `auction` were bound to the namespace URI of our on-line auction application, the step `item[reserve-price > 1000]` might be written using QNames as follows:

```
auction:item[auction:reserve-price > 1000]
```

The process of binding a prefix to a namespace URI is described in the next-to-last section. In most of our examples, we use one-part names rather than QNames. This use is realistic because XQuery provides a way to specify a default namespace for a query. Such use makes it unnecessary for a query to use QNames unless it needs to refer to names from multiple namespaces.

This paper provides only a brief introduction to the path expressions available in XPath and XQuery. Like XQuery, XPath is an evolving language. A working draft of a new version of XPath, called XPath 2.0,[20] has recently been published jointly by the XML Query and XSLT Working Groups. It is expected that XPath 2.0 and XQuery will share not only a common syntax for path expressions and predicates but several other kinds of expressions as well.

Element constructors. Path expressions are powerful, but they have an important limitation: they can only select existing nodes. A full query language needs a facility to construct new elements and attributes and to specify their contents and relationships. This facility is provided in XQuery by a kind of expression called an *element constructor*.

The simplest kind of element constructor looks exactly like the XML syntax for the element to be created. For example, the following expression constructs an element named `highbid` containing one attribute named `status` and two child elements named `itemno` and `bid-amount`:

```
<highbid status = "pending">
   <itemno>4871</itemno>
   <bid-amount>250.00</bid-amount>
</highbid>
```

In the example above, the values of the elements and attributes are constants. However, in many cases it is necessary to create an element or an attribute whose value is computed by some expression. In this case, the expression is enclosed in curly braces to indicate that it is to be evaluated rather than treated as literal text. The expression is evaluated and replaced by its value in the element constructor. In the following example, the values of the elements and attributes are computed by expressions. The variables `$s`, `$i`, and `$bids` used in these expressions must be bound by some enclosing expression.

```
<highbid status = "{$s}">
   <itemno> {$i} </itemno>
   <bid-amount>
      {max($bids[itemno = $i]/bid-amount)}
   </bid-amount>
</highbid>
```

The content of an element constructor may be any expression. In general, the expression used in an element constructor may generate a sequence of items, including atomic values, elements, and attributes. Attributes that are generated inside an element constructor become attached to the constructed element. Elements and atomic values that are generated inside an element constructor become the content of the constructed element. In the following example, an element constructor contains an expression, enclosed in curly braces, that generates one attribute and two subelements. The variable `$b` must be bound by some enclosing expression.

```
<highbid>
   {
    $b/@status,
    $b/itemno,
    $b/bid-amount
   }
</highbid>
```

The element node produced by an element constructor is a new node with its own node identity. If the newly constructed element has child nodes and attributes that are derived from existing nodes, as in the above example, the new child nodes and attributes are copies of the nodes from which they were derived, with new node identities.

In the above examples of element constructors, even though the content of the element may be computed, the name of the constructed element is a known constant. However, it is sometimes necessary to construct an element whose name as well as its content is computed. For this purpose, XQuery provides a special kind of constructor called a *computed element con-*

structor. A computed element constructor consists of the keyword `element`, followed by two expressions in curly braces—the first expression computes the name of the element, and the second expression computes the content of the element.

For an example of the use of a computed constructor, suppose that the variable `$e` is bound to an element with a numeric value. We need to construct a new element that has the same name as `$e` and the same attributes as `$e`, but we want its value to be twice the value of `$e`. This construction can be accomplished by the following expression, which uses the `data` function to extract the numeric value of the original node:

```
element
   {name($e)}
   {$e/@*, data($e)*2}
```

Similar to a computed element constructor, XQuery provides a *computed attribute constructor*, which consists of the keyword `attribute`, followed by two expressions in curly braces—the first expression computes the name of the attribute and the second expression computes its value. An attribute constructor can be used anywhere an attribute is valid, for example, inside an element constructor. The following attribute constructor, based on the bound variable `$p`, might generate an attribute that looks like `father="Frank"` or `mother="Mary"`. This example uses a conditional (if-then-else) expression, described later.

```
attribute
   {if $p/sex="M" then "father" else "mother"}
   {$p/name}
```

Iteration and sorting. Iteration is an important part of a query language. XQuery provides a way to iterate over a sequence of values, binding a variable to each of the values in turn and evaluating an expression for each binding of the variable.

The simplest form of iteration in XQuery consists of a for clause that names a variable and provides a sequence of values over which the variable is to iterate, followed by a return clause that contains the expression to be evaluated for each variable binding. The following example illustrates this simple form of iteration:

```
for $n in (2, 3) return $n + 1
```

The result of this simple iterative expression is the sequence `(3, 4)`.

A for clause may specify more than one variable, with an iteration sequence for each variable. Such a for clause produces tuples of variable bindings that form the Cartesian product of the iteration sequences. Unless otherwise specified, the binding tuples are generated in an order that preserves the order of the iteration sequences, using the leftmost variable as the "outer loop" and the rightmost variable as the "inner loop." The following example illustrates a for clause that contains two variables and two iteration sequences:

Iteration is an important part of a query language.

```
for $m in (2, 3), $n in (5, 10)
return <fact>{$m} times {$n} is
   {$m * $n}</fact>
```

The result of this expression is the following sequence of four elements:

```
<fact>2 times 5 is 10</fact>
<fact>2 times 10 is 20</fact>
<fact>3 times 5 is 15</fact>
<fact>3 times 10 is 30</fact>
```

The for clauses illustrated above and the let clause illustrated earlier are both special cases of a more general expression called a FLWR (pronounced "flower") expression. In its most general form, a FLWR expression may have multiple for clauses, multiple let clauses, an optional where clause, and a return clause.

As we have already seen, the function of the for clause and let clause is to bind variables. Each of these clauses contains one or more variables and an expression associated with each variable. The expressions evaluate to sequences and may contain references to variables bound in previous clauses. The difference between a for clause and a let clause is that a for clause iterates each variable over the associated sequence, binding the variable in turn to each item in the sequence, whereas a let clause binds each variable to the associated sequence as a whole. This difference is illustrated by the following pair of clauses:

```
for $i in (1 to 3)
let $j := (1 to $i)
```

This pair of clauses is not a full FLWR expression because it does not have a return clause. The for clause and let clause simply produce a sequence of binding tuples. The clauses in the above example produce the following sequence of three binding pairs:

```
$i = 1, $j = 1
$i = 2, $j = (1, 2)
$i = 3, $j = (1, 2, 3)
```

In general, the number of binding tuples produced by a series of for clauses and let clauses is equal to the product of the cardinalities of the iteration expressions in the for clauses. A let clause without any for clause, of course, produces only a single binding tuple.

The binding tuples produced by the for clauses and let clauses in a FLWR expression are filtered by the optional where clause. The where clause contains an expression that is evaluated for each binding tuple. If the value of the where expression is the Boolean value `true` or a sequence containing at least one node (an "existence test"), the binding tuple is retained; otherwise the binding tuple is discarded.

The return clause of the FLWR expression is then executed once for each binding tuple retained by the where clause, in order. The results of these executions are concatenated into a sequence that serves as the result of the FLWR expression.

The power of FLWR is illustrated by Q4, a query over our auction database.

(Q4) For each item that has more than ten bids, generate a popular-item element containing the item number, description, and bid count.

```
for $i in document("items.xml")/*/item
let $b := document("bids.xml")
   /*/bid[itemno = $i/itemno]
where count ($b) > 10
return
   <popular-item>
      {
       $i/itemno,
       $i/description,
       <bid-count> {count ($b)} </bid-count>
      }
   </popular-item>
```

The for clause and let clause produce a binding pair for each item in `items.xml`. In each binding pair, `$i` is bound to the item and `$b` is bound to a sequence containing all the bids for that item. The where clause retains only those binding tuples in which `$b` contains more than ten bids. The return clause then generates an output element for each of these bindings, containing the item number, description, and bid count.

By default, the order of the output sequence of a FLWR expression preserves the order of the iteration sequences. The prefix operator `unordered` can be used before any expression to indicate that the order of the result is not significant. This gives the implementation greater flexibility to optimize the execution of the expression (for example, by iterating in a different order).

Any sequence can be reordered by a sortby clause that contains one or more ordering expressions. For each item in the original sequence, the ordering expressions are evaluated using the given item as the context item. The items in the original expression are then reordered into ascending or descending order based on the values of their ordering expressions. Of course, each ordering expression must return a single result, and these results must be comparable by the `gt` operator. For the purpose of a sortby clause, an empty sequence can be treated either as greater than any other value or as less than any other value, under user control.

A sortby clause is often useful in reordering the results of a FLWR expression. For example, if it is desired for the `popular-item` elements generated by Q4 to be sorted into descending order by `bid-count`, the following clause could be added at the end of Q4:

```
sortby bid-count descending
```

It is important to realize that `sortby` is not a part of a FLWR expression but a separate kind of XQuery expression that can be used to reorder any sequence, whether generated by a FLWR expression or not.

However, when a FLWR expression is followed by `sortby`, a smart optimizer will realize that the reordering of the output items relaxes the usual constraints on the ordering of the binding tuples.

Q4 illustrates how a FLWR expression can have some of the same characteristics as a join query in a relational database system and also some of the same characteristics as a grouping query. Q4 is like a join query because it correlates elements found in two different XML files, named `items.xml` and `bids.xml`. It is also like a grouping query because it groups bids together by item number and computes the number of bids in each group (in SQL, this might be expressed as `GROUP BY itemno`).

Arithmetic. We have already seen several examples of the use of arithmetic operators. XQuery provides the usual arithmetic operators: +, −, *, `div`, and `mod`, as well as the aggregating functions `sum`, `avg`, `count`, `max`, and `min`, which operate on a sequence of numbers and return a numeric result. The division operator in XQuery is called `div` to distinguish it from the slash that is used in path expressions. When the subtraction operator follows a name, it must have a preceding blank to distinguish it from a hyphen, since a hyphen is a valid name character in XML.

Arithmetic operators are defined on numeric values (or, in the case of the aggregating functions, sequences of numeric values). Numeric values include values of type `integer`, `decimal`, `float`, `double`, or types derived from these types. When the operands of an arithmetic operator are mixed, they are promoted to the nearest common type using the promotion hierarchy `integer` → `decimal` → `float` → `double`. If an operand of an arithmetic operator is a node, its typed value is automatically extracted.

The behavior of arithmetic operators on empty sequences is an important special case. In XQuery, an empty sequence is sometimes used to represent missing or unknown information, in much the same way that a null value is used in relational systems. For this reason, the +, −, *, `div`, and `mod` operators are defined to return an empty sequence if either of their operands is an empty sequence. To illustrate the application of this rule, suppose that the variable `$emps` is bound to a sequence of `emp` elements, each of which represents an employee and contains a `name` element, a `salary` element, an optional `commission` element, and an optional `bonus` element. The expression in Q5 transforms this sequence into a new sequence of `emp` elements, each of which contains a `name` element and a `pay` element whose value is the employee's total pay. For those employees whose commission or bonus is missing (`$e/commission` or `$e/bonus` evaluates to an empty sequence), the generated `pay` element will be empty.

(Q5) Given a sequence of emp elements, replace their salary, commission, and bonus subelements with a new pay element containing the sum of the values of the original elements, and order the resulting sequence in ascending order by the value of the pay element.

```
for $e in $emps
return
   <emp>
      {
       $e/name,
       <pay> {$e/salary + $e/commission
         + $e/bonus} </pay>
      }
   </emp>
sortby (pay)
```

In some cases, it may be desirable to provide a default value that can be substituted for missing operands in an arithmetic expression. The next section of this paper illustrates how a user-defined function can be written for this purpose.

Operations on sequences. In a sense, all XQuery operations are operations on sequences, since every value in XQuery is either a sequence or an error. However, XQuery provides three operators that are specifically designed for combining sequences of nodes: `union`, `intersect`, and `except`. A `union` of two node sequences is a sequence containing all the nodes that occur in either of the operands. The `intersect` operator produces a sequence containing all the nodes that occur in both of its operands. The `except` operator produces a sequence containing all the nodes that occur in its first operand but not in its second operand.

The `union`, `intersect`, and `except` operators return node sequences in document order and eliminate duplicates from their result sequences, based on node identity. Query Q6 provides an example of the use of the `intersect` operator.

(Q6) Construct a new element named recent-large-bids, containing copies of all the bid elements in the document bids.xml that have a bid-amount of more than 1000 and a bid-date after January 1, 2002.

```
<recent-large-bids>
   document("bids.xml")
      /*/bid[bid-amount > 1000.00]
   intersect
   document("bids.xml")
      /*/bid[bid-date > date("2002-01-01")]
</recent-large-bids>
```

Expressions that apply the `union`, `intersect`, and `except` operators can often be expressed in another way. For example, the following query is equivalent to Q6:

```
<recent-large-bids>
   document("bids.xml")/*/bid
      [bid-amount > 1000.00 and bid-date
         > date("2002-01-01")]
</recent-large-bids>
```

It is important to remember that `intersect` and `except` are not useful in combining sequences of nodes from different documents, since there is no possibility that two nodes in different documents could have the same node identity. For example, consider the following query:

```
document("items.xml")//itemno
except
document("bids.xml")//itemno
```

This query applies the `except` operator to two sequences of `itemno` nodes. Since the node sequences are selected from different documents, there is no possibility that any node in the second sequence could be identical to a node in the first sequence. Therefore, this query returns all the `itemno` nodes in `items.xml`. If the intent of the query had been to make a list of `itemno` elements for items that have no bids, this could have been accomplished as follows, using the library function `empty`, which returns `true` if its operand is an empty sequence:

```
for $i in document("items.xml")//item
where empty(document("bids.xml")
  //bid[itemno eq $i/itemno])
return $i/itemno
```

In the above example, the predicate `itemno eq $i/itemno` compares two `itemno` nodes by extracting and comparing their content rather than by their identity.

The `|` operator, retained for compatibility with XPath 1.0, is equivalent to the `union` operator. These operators are sometimes used in a step of a path expression. For example, the following path expression finds the union of all b children and c children of nodes in the sequence bound to `$a`; the nodes in this union then serve as context nodes for the next step in the path.

```
$a/(b | c)/d
```

Conditional expressions. A *conditional expression* provides a way of executing one of two expressions, depending on the value of a third expression. It is written in the familiar if . . . then . . . else format provided by many languages. In XQuery, all three clauses (`if`, `then`, and `else`) are required, and the expression in the if clause must be enclosed in parentheses.

The result of a conditional expression depends on the value of the expression in the if clause, called the *test expression*. The rules are as follows:

If the value of the test expression is the Boolean value `true`, or a sequence containing at least one node (serving as an "existence test"), the then clause is executed.

If the value of the test expression is the Boolean value `false` or an empty sequence, the else clause is executed.

Otherwise, the conditional expression returns the error value.

The following simple conditional expression might be used to return the price of a part, depending on the existence of an attribute named `discounted` (independently of the value of the attribute):

```
if ($part/@discounted) then $part/wholesale
  else $part/retail
```

Q7 in Figure 3 is an example of a more complex query that contains a conditional expression. The query also illustrates several levels of nesting of FLWR expressions and element constructors.

Quantified expressions. Quantified expressions allow testing of some condition to see whether it is true for *some* value in a sequence (called an *existential quantifier*), or for *every* value in a sequence (called a *universal quantifier*). The result of a quantified expression is always `true` or `false`.

Figure 3 Example of more complex query

(Q7) Generate a report containing the status of the bids for various items. Label each bid with a status "OK," "too small," or "too late." Enclose the report in an element called bid-status-report.

```
<bid-status-report>
   for $i in document ("items.xml")/*/item
   return
      <item>
         {
          $i/itemno,
          for $b in document ("bids.xml")/*/bid[itemno = $i/itemno]
          return
             <bid>
                {
                 $b/bidder,
                 $b/bid-amount,
                 <status>
                    {
                     if ($b/bid-date > $i/end-date) then "too late"
                     else if ($b/bid-amount < $i/reserve-price)
                        then "too small"
                     else "OK"
                    }
                 </status>
                }
             </bid>
         }
      </item>
</bid-status-report>
```

Like a FLWR expression, a quantified expression allows a variable to iterate over the items in a sequence, being bound in turn to each item in the sequence. For each variable binding, a test expression is evaluated. A quantified expression that begins with `some` returns the value `true` if the test expression is true for *some* variable binding, as in the following example:

```
some $n in (5, 7, 9, 11) satisfies $n > 10
```

A quantified expression that begins with `every`, in contrast, returns the value `true` if the test expression is true for *every* variable binding. For example, the following quantified expression returns the value `false` because the test expression is true for some but not all bindings:

```
every $n in (5, 7, 9, 11) satisfies $n > 10
```

The use of a quantified expression in a query is illustrated by Q8.

(Q8) Find the items in items.xml for which all the bids received were more than twice the reserve price. Return copies of all these item elements, enclosed in a new element called underpriced-items.

```
<underpriced-items>
   for $i in document("items.xml")
   where every $b in document("bids.xml")
      /*/bid[itemno = $i/itemno]
      satisfies $b/bid-amount
          > 2 * $i/reserve-price
   return $i
</underpriced-items>
```

Functions

We have already seen several examples of functions, including the `document` function and aggregating functions such as `avg`. XQuery provides a library of predefined functions, listed in Reference 11, and also allows users to define functions of their own. A function may take zero or more parameters. A function definition must specify the name of the function and the names of its parameters. It may optionally specify types for the parameters and the result of the function. It must also provide the body of the function, which is an expression enclosed in curly braces. When the function is called, the arguments of the function call are bound to the parameters of the function and the body is executed, producing the result of the function call. If no type is specified for a function parameter, that parameter accepts values of any type. If no type is specified for the result of the function, the function may return a value of any type.

The following example defines a function named `highbid` that takes an element node as its parameter and returns a decimal value. The function interprets its parameter as an `item` element and extracts its item number; it then finds and returns the largest `bid-amount` that has ever been recorded for that item number. The example also illustrates a function call that invokes the `highbid` function on the item with item number 1234.

```
define function highbid(element $item)
   returns decimal
  {
   max(document("bids.xml")
      //bid[itemno = $item/itemno]/bid-amount)
  }

highbid(document("items.xml")
   //item[itemno = "1234"])
```

The types used as the argument-types and result-type of a function definition may be simple types such as `decimal`, or more complex types such as elements and attributes. The rules for declaring types in function definitions are described in more detail in the next section.

XQuery does not support overloading of user-defined functions; that is, it does not permit two user-defined functions to have the same qualified name. Nevertheless, some of the XQuery built-in functions are overloaded. For example, the `string` function can convert an argument of almost any type into a string.

The arguments of a function call must match the declared types of the function parameters. For this purpose, a function argument of a numeric type may be promoted to the declared parameter type, using the promotion hierarchy `integer` → `decimal` → `float` → `double`. An argument is also considered to be a match if the type of the argument is derived from (i.e., a subtype of) the declared parameter type. If a function that expects an atomic value is called with an argument that is an element, the typed value of the element is extracted and checked for compatibility with the expected parameter type before it is passed to the function. The value produced by the body of a function must also match the return type declared in the function definition, using the same rules that are used for parameter matching.

The following example illustrates how a user might write a function to provide a default value for missing data. The function named `defaulted` takes two parameters: a (possibly missing) element node, and a default value. If the element is present and has a nonempty value, the function returns that value; but if the element is absent or empty, the function returns the default value.

```
define function defaulted
   (element? $e, anySimpleType $d)
   returns anySimpleType
   {
    if (empty($e)) then $d
    else if (empty($e/*)) then $d
    else data($e)
   }
```

Using this function, query Q5 could be rewritten as follows. In this formulation, missing or empty commission or bonus elements are treated as though they have the value zero:

```
for $e in $emps
return
   <emp>
      {
       $e/name,
       <pay> { $e/salary
         + defaulted ($e/commission, 0)
         + defaulted ($e/bonus, 0) }
       </pay>
      }
   </emp>
sortby(pay)
```

A function that invokes itself in its own body is called a *recursive function*, and two functions whose bodies invoke each other are called *mutually recursive functions*. Recursion is a powerful feature in function definitions, particularly in functions that are defined over a hierarchical data model such as XML. As an illustration of a recursive function, the `depth` function in the following example can be invoked on an element and returns the depth of the element hierarchy beginning with its argument. If the argument element has no descendants, the depth of the hierarchy is one. Otherwise, the depth of the hierarchy is one more than the maximum depth of any hierarchy rooted in a child of the argument element; this value is computed by a recursive call to the `depth` function. The example also illustrates a function call that invokes the `depth` function to find the depth of the document named `bids.xml`.

```
define function depth(element $e)
   returns integer
   {
    if (empty($e/*)) then 1
    else 1 + max
      (for $c in $e/* return depth($c))
   }

depth(document("bids.xml"))
```

Types

In writing a query, it is sometimes necessary to refer to a particular type. For example, function definitions need to describe the types of the function parameters and result, as noted in the previous section. Other types of XQuery expressions, described later in this section, also need to refer to specific types.

One way to refer to a type is by its qualified name, or QName. A QName may refer to a built-in type such as `xs:integer` or to a type that is defined in some schema, such as `abc:address`. If the QName has a namespace prefix (the part to the left of the colon), that prefix must be bound to a specific namespace URI. This binding is accomplished by a namespace declaration in the query prolog, described in the next section.

Another way to refer to a type is by a generic keyword such as `element` or `attribute`. This keyword may optionally be followed by a QName that further restricts the name or type of the node. For example, `element` denotes any element; `element shipto` denotes an element whose name is `shipto`; and `element of type abc:address` denotes an element whose type is `address` as declared in the namespace `abc`. The keyword `attribute` denotes any attribute, `node` denotes any node, and `item` denotes any item (node or atomic value).

XQuery also provides additional syntax that makes it possible to refer to other kinds of nodes and to element types that are defined in a local part of a schema. For example, `element city in customer/address` refers to the element named `city`, as defined in the schema context `customer/address`.

A reference to a type may optionally be followed by one of three occurrence indicators: "`*`" means "zero or more"; "`+`" means "one or more," and "`?`" means "zero or one." The absence of an occurrence indicator denotes exactly one occurrence of the indicated type. The use of occurrence indicators is illustrated by the following examples:

`element memo?` denotes an optional occurrence of an element with the name `memo`
`element of type order+` denotes one or more elements with the type `order`
`element*` denotes zero or more unrestricted elements
`attribute?` denotes an optional attribute of any name or type

Type references occur not only in function definitions but also in several other places in XQuery. One of these places is the second operand of `instance of`, a binary operator that returns `true` if its first operand is an instance of the type named in its second operand. The following examples illustrate usage of the `instance of` operator, presuming that the prefix `xs` is bound to the schema namespace, `http://www.w3.org/2001/XMLSchema`:

`49 instance of xs:integer` returns `true`
`"Hello" instance of xs:integer` returns `false`
`<partno>369</partno> instance of element*` returns `true`
`$a instance of element shipto` returns `true` if `$a` is bound to an element whose name is `shipto`

Occasionally it may be necessary for a query to process an expression in a way that depends on the dynamic (run-time) type of the expression. For example, a query might be preparing mailing labels and might need to extract geographical information from various types of addresses. For such applications, XQuery provides an expression called `typeswitch`, which is loosely modeled on the `switch` statement of the C or Java** languages. The first part of a `typeswitch` consists of the expression whose type is being tested, called the *operand expression*, and optionally a variable that is bound to the value of the operand expression. This is followed by one or more case clauses, each of which contains a type and an expression. The operand expression is tested against the type named in each of the case clauses in turn. The first case clause for which the operand expression is an instance of the named type is called the *effective case*; the expression in this case clause is evaluated and serves as the result of the `typeswitch`. If the operand expression does not conform to any of the types named by the case clauses, the result of the `typeswitch` is taken from a final default clause.

The use of a `typeswitch` expression is illustrated by the following example. This expression might occur inside a loop in which the variable `$customer` iterates over a set of `customer` elements, each of which has a subelement named `billing-address`. The `billing-address` subelements are of several different types, each of which needs to be handled in its own way. In the example, `$a` is bound to a `billing-address` and then one of several expressions is evaluated, depending on the dynamic type of `$a`. Within each case clause, `$a` has a specific type, for example, in the first case clause, the type of `$a` is known to be `element of type USAddress`. If a `billing-address` element is encountered that does not conform to one of the expected types, the result of this example expression is "unknown."

```
typeswitch($customer/billing-address) as $a
   case element of type USAddress
     return $a/state
   case element of type CanadaAddress
     return $a/province
   case element of type JapanAddress
     return $a/prefecture
   default return "unknown"
```

Type names are also used in three similar-looking XQuery expressions called `cast`, `treat`, and `assert`. Each of these expressions consists of a keyword, a reference to a type, and an expression enclosed in parentheses.

A `cast` expression is used to convert the result of an expression into one of the built-in types of XML Schema. A predefined set of casts is supported. For example, the result of the expression `$x div 5` could be cast to the `xs:double` type by the expression `cast as xs:double($x div 5)`. A cast may return an error value if it is unsuccessful. For example, `cast as xs:integer($mystring)` will be successful if `$mystring` is a string representation of an integer, but it will return an error if `$mystring` has the value `"Hello"`. The cast expression cannot be used to cast a value into a user-defined type; however, a user can write a function for this purpose.

A `treat` expression is used to ensure that the dynamic (run-time) type of an expression conforms to an expected type. For example, suppose that the static (compile-time) type of the expression `$customer/shipping-address` is `Address`. A certain subexpression may be meaningful only for values that conform to a subtype of `Address`, such as `USAddress`. The writer of the subexpression may use a `treat` expression to declare the expected type of the subexpression, as in the following example:

```
treat as USAddress($customer/billing-address)
```

Unlike a `cast` expression, a `treat` expression does not actually change the type of its operand. Instead, its effect is twofold: (1) it assigns a specific static type to its operand, which can be used for type-checking when the query is compiled; and (2) at execution time, if the actual value of the expression does not conform to the named type, it returns an error value.

To see how a query processor might make use of the information provided by a `treat` expression, consider the following example:

```
$customer/billing-address/zipcode
```

A type-checking XQuery compiler might consider the above example to be a type error, since the static type of `$customer/billing-address` is `Address`, and the `Address` type does not in general have a `zipcode` subelement. However, in the following reformulation of the example, the static type of the expression is changed to `USAddress`, which has a `zipcode` subelement, and the type error is removed:

```
(treat as USAddress
   ($customer/billing-address))/zipcode
```

Like `treat`, `assert` is used to provide a query processor with information that may be useful for type-checking. An `assert` expression serves as an assertion to the query processor that its operand expression has a particular static type. If the processor is checking a query for static type-safety, it may raise an error if it cannot verify that the given expression conforms to the asserted type. An `assert` expression is more strict than a `treat` expression, because it pertains to the static type of the expression, and therefore it is independent of input data and can be checked before execution of the query. A `treat` expression, in contrast, pertains to the dynamic type of the expression, and therefore depends on input data and can only be checked during query processing.

The following example, unlike the similar `treat` expression discussed above, will generate a compile-time type error if the static type of `$customer/billing-address` is `Address`:

```
(assert as USAddress
   ($customer/billing-address))/zipcode
```

XQuery does not require an implementation to provide static type-checking. For a query processor that does not provide static type-checking, an `assert` expression is equivalent to a `treat` expression.

Validation

The process of schema validation is defined in Reference 3. Schema validation may be applied to an XML document or to a part of a document such as an individual element. The material being validated is compared with the definitions in a given schema, which describes a particular kind of document. The validation process may label an element as valid or invalid; it may also assign a specific type to an element and provide additional information such as default values for certain attributes. For example, validation of an element named `shipto` might assign it the specific type `USAddress` and might provide a default value for its `carrier` attribute.

Schema validation is applied to input documents as part of the process of representing them in the query data model. In addition, schema validation can be invoked explicitly on a query result or on some intermediate expression within a query.

The query data model associates a type annotation with each element node. A type annotation indicates that an element has been validated as conforming to a specific named type. Elements that have not been validated or that do not conform to a named type have the generic type annotation `anyType`. For example, an element that is created by an element constructor has the type annotation `anyType` until it is given a more specific type by a `validate` expression. The following example constructs an element and validates it against the schema(s) that are named in the query prolog:

```
validate { <shipto>
              <street>123 Elm St.</street>
              <city>Elko, NV</city>
              <zipcode>85039</zipcode>
           </shipto> }
```

Type annotations are used by expressions such as `instance of` and `typeswitch` that test the type of an element, and by expressions such as function calls that require an element of a particular type. For example, validation of the `shipto` element above might assign it a type annotation of `USAddress`, which might enable it to be used as an argument to a function whose parameter type is `element of type USAddress`.

Structure of a query

In XQuery, a query consists of two parts called the *query prolog* and the *query body*. The query prolog contains a series of declarations that define the environment for processing the query body. The query body is simply an expression whose value defines the result of the query.

The query prolog is needed only if the query body depends on one or more namespaces, schemas, or functions. If such a dependency exists, the object(s) that the query body depends on must be declared in the query prolog. We will discuss declarations for namespaces, schemas, and functions separately.

A namespace declaration defines a namespace prefix and associates it with a namespace URI. The prefix can be any identifier. For example, the following namespace declaration defines the prefix `xyz` and associates it with a specific URI:

```
namespace xyz
   = "http://www.xyz.com/example/names"
```

This declaration enables the prefix `xyz` to be used in QNames in the query body. It associates the prefix with the URI of a specific namespace and serves as a unique qualifier for names of elements, attributes, and types. For example, `xyz:billing-address` might uniquely identify the `billing-address` element defined in the namespace `http://www.xyz.com/example/names`. If desired, multiple namespace prefixes can be associated with the same namespace.

The query prolog can declare a default namespace that applies to all unqualified element and type names and another default namespace that applies to all unqualified function names. The syntax for declaring default namespaces is illustrated in the following example:

```
default element namespace
   = "http://www.xyz.com/example/names"
default function namespace
   = "http://www.xyz.com/example/functions"
```

If no default namespaces are provided, unqualified names of elements, types, and functions are considered to be in no namespace. Unqualified attribute names are always considered to be in no namespace.

In addition to namespace declarations, a query prolog can contain one or more schema imports. A schema import identifies a schema by its URI and optionally provides a second URI that specifies the location where the schema file can be found. The purpose of the schema import is to make available to the query processor the definitions of elements, attributes, and types that are declared in the named schema. The query processor can use these definitions for validating newly constructed elements, for optimization, and for doing static type analysis on a query.

A schema usually defines a set of elements, attributes, and types in a particular namespace, called its *target namespace*, but it does not define a namespace prefix. Therefore, a schema import may specify a namespace prefix to be bound to the target namespace of the given schema. For example, the following schema import binds the namespace prefix `xhtml` to the target namespace of a particular schema and also provides the system with a nonbinding "hint" for where this schema can be found:

```
schema namespace xhtml
   = "http://www.w3.org/1999/xhtml"
     at"http://www.w3.org/1999/xhtml/xhtml.xsd"
```

In addition to namespace declarations and schema imports, a query prolog may contain one or more function definitions. We have seen examples of function definitions in an earlier section. The functions defined in the query prolog may be used in the query body or in the bodies of other functions. It is expected that the query prolog will also provide a means of importing function definitions from an external function library.

Conclusion

XQuery is a functional language consisting of several types of expressions that can be composed with full generality. XQuery expression-types include path expressions, element constructors, function calls, arithmetic and logical expressions, conditional expressions, quantified expressions, expressions on sequences, and expressions on types.

XQuery is defined in terms of a data model based on heterogeneous sequences of nodes and atomic values. An instance of this data model may contain one or more XML documents or fragments of documents. A query provides a mapping from one instance of the data model to another instance of the data model. A query consists of a prolog that establishes the processing environment, and an expression that generates the result of the query.

Currently, XQuery is defined only by a series of working drafts, and design of the language is an ongoing activity of the W3C XML Query Working Group. The working group is actively discussing the XQuery type system and how it is mapped to and from the type system of XML Schema. It is also discussing full-text search functions, serialization of query results, error-handling, and a number of other issues. It is likely that the final XQuery specification will include multiple conformance levels; for example, it may define how static type-checking is done but not require that it be done by every conforming implementation. It is also expected that a subset of XQuery will be designated as XPath Version 2.0 and will be made available for embedding in other languages such as XSLT.[5]

This paper has presented an overview of XQuery, illustrated with some example queries. For a more complete description of XQuery and a BNF (Backus-Naur Form) grammar for the language, the reader is referred to Reference 7. The document at this URI will be updated periodically to contain the latest XQuery specification as the language continues to evolve.

Just as XML is emerging as an application-independent format for exchange of information on the Internet, XQuery is designed to serve as an application-independent format for exchange of queries. If XQuery is successful in providing a standard way to retrieve information from XML data sources, it will help XML to realize its potential as a universal information representation.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references

1. *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation (6 October 2000), see http://www.w3.org/TR/REC-xml.
2. World Wide Web Consortium (W3C), see http://www.w3.org.
3. *XML Schema, Parts 0, 1, and 2*, W3C Recommendation (2 May 2001), see http://www.w3.org/TR/xmlschema-0, http://www.w3.org/TR/xmlschema-1 and http://www.w3.org/TR/xmlschema-2.
4. *XML Path Language (XPath) Version 1.0*, W3C Recommendation (16 November 1999), see http://www.w3.org/TR/xpath.
5. *XSL Transformation (XSLT) Version 1.0*, W3C Recommendation (16 November 1999), see http://www.w3.org/TR/xslt.
6. W3C XML Query Working Group, see http://www.w3.org/XML/Query.
7. *XQuery 1.0: An XML Query Language*, W3C Working Draft (16 August 2002), see http://www.w3.org/TR/xquery.
8. *XML Query Requirements*, W3C Working Draft (15 February 2001), see http://www.w3.org/TR/xmlquery-req.
9. *XQuery 1.0 and XPath 2.0 Data Model*, W3C Working Draft (16 August 2002), see http://www.w3.org/TR/query-datamodel.
10. *XQuery 1.0 Formal Semantics*, W3C Working Draft (16 August 2002), see http://www.w3.org/TR/query-semantics.
11. *XQuery 1.0 and XPath 2.0 Functions and Operators*, W3C Working Draft (16 August 2002), see http://www.w3.org/TR/xquery-operators.
12. *XML Query Use Cases*, W3C Working Draft (16 August 2002), see http://www.w3.org/TR/xmlquery-use-cases.
13. D. Chamberlin, J. Robie, and D. Florescu, "Quilt: An XML Query Language for Heterogeneous Data Sources," *Lecture Notes in Computer Science*, Springer-Verlag (December 2000), see also http://www.almaden.ibm.com/cs/people/chamberlin/quilt.html.
14. T. Atwood, D. Barry, J. Duhl, J. Eastman, G. Ferran, D. Jordan, M. Loomis, and D. Wade, *The Object Database Standard: ODMG-93, Release 1.2*, R. G. C. Catell, Editor, Morgan Kaufmann Publishers, San Francisco, CA (1996).
15. *Information Technology-Database Language SQL*, Standard No. ISO/IEC 9075, International Organization for Standardization (ISO) (1999); available from American National Standards Institute, New York, NY 10036, (212) 642–4900.
16. J. Robie, J. Lapp, and D. Schach, *XML Query Language (XQL)*, see http://www.w3.org/TandS/QL/QL98/pp/xql.html.
17. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, *A Query Language for XML*, see http://www.research.att.com/~mff/files/final.html.
18. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener, "The Lorel Query Language for Semistructured Data," *International Journal on Digital Libraries* **1**, No. 1, 68–88 (April 1997), see http://www-db.stanford.edu/~widom/pubs.html.
19. *Namespaces in XML*, W3C Recommendation (14 January 1999), see http://www.w3.org/TR/REC-xml-names.
20. *XML Path Language (XPath) 2.0*, W3C Working Draft (20 December 2001), see http://www.w3.org/TR/xpath20.

Accepted for publication June 17, 2002.

Don Chamberlin *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: chamberlin@almaden.ibm.com).* Dr. Chamberlin is best known as co-inventor of the SQL database language and as author of two books on IBM's relational database products. He holds a B.S. degree in engineering from Harvey Mudd College and a Ph.D. degree in electrical engineering from Stanford University. He is an ACM Fellow and a member of the National Academy of Engineering and the IBM Academy of Technology. Dr. Chamberlin is currently a research staff member at the Almaden Research Center, where his work is focused on relational database technology, document processing, and XML. He serves as one of IBM's representatives to the W3C XML Query Working Group and as an editor of the XQuery language specification.