The emergence of the World Wide Web in the 1990s was a seminal event in human culture. Suddenly, as if overnight, a significant fraction of the world's computers were connected, not only by a physical network but also by a common protocol for exchanging information. The Web offered an unprecedented opportunity to make information truly ubiquitous. It seemed to promise that people would no longer need to move physically to places and times where information was available, since all information would be everywhere, all the time.

Realizing this promise required some organizing principle for the exchange of information. This principle had to be independent of any particular language or application and easily extensible to new and unanticipated kinds of information. At present, the leading candidate for this organizing principle is the Extensible Markup Language, XML [XML]. XML provides a neutral notation for labeling the parts of a body of information and representing the relationships among these parts. Since XML does not attach any semantic meaning to its labels, applications are free to interpret them as they see fit. Applications that agree on a common vocabulary can use XML for data interchange. Since XML does not mandate any particular storage technique, it can be used as a common interchange format among systems that store data in file systems, relational databases, object repositories, and many other storage formats.

Since XML is emerging as a universal format for data interchange among disparate applications, it is natural for queries that cross application boundaries to be framed in terms of the XML representation of data. In

other words, if an application is viewed as a source of information in XML format, it is logical to pose queries against that XML format. This is the basic reason why a query language for XML data is extremely important in a connected world.

Recognizing the importance of an XML query language, the World Wide Web Consortium (W3C) [W3C] organized a query language workshop called QL'98 [QL98], which was held in Boston in December 1998. The workshop attracted nearly a hundred participants and fostered sixty-six papers investigating various aspects of querying XML data. One of the long-term outcomes of the workshop was the creation of a W3C working group for XML Query [XQ-WG]. This working group, chaired by Paul Cotton, met for the first time in September 1999. Its initial charter called for the specification of a formal data model and query language for XML, to be coordinated with existing W3C standards such as XML Schema [SCHEMA], XML Namespaces [NAMESP], XML Information Set [INFOSET], and **XSLT** [XSLT]. The purpose of the new query language was to provide a flexible facility to extract information from real and virtual XML documents. Approximately forty participants became members of the working group, representing about twenty-five different companies, along with a W3C staff member to provide logistical support.

One of the earliest activities of the Query working group was to draw up a formal statement of requirements for an XML query language [XQ-REQ]. This document was quickly followed by a set of **use cases** [XQ-UC] that described diverse usage scenarios for the new language, including specific queries and expected results. The XML Query Working Group undertook to define a language with two alternative syntaxes: a **keyword**-based syntax called XQuery [XQ-LANG], optimized for human reading and writing, and an XML-based syntax called XQueryX [XQ-X], optimized for machine generation. This chapter describes only the keyword-based XQuery syntax, which has been the major focus of the working group.

Creating a new query language is a serious business. Many person-years have been spent in defining XQuery, and many more will be spent on its implementation. If the language is successful, developers of web-based applications will use it for many years to come. A successful query language

can enhance productivity and serve as a unifying influence in the growth of an industry. On the other hand, a poorly designed language can inhibit the acceptance of an otherwise promising technology. The designers of XQuery took their responsibilities very seriously, not only in the interest of their individual companies but also in order to make a contribution to the industry as a whole.

The purpose of this chapter is to discuss the major influences on the design of the XQuery language. A tutorial introduction to XQuery appears in Chapter 1. Some of the influences on XQuery were principles of computer language design. Others were related languages, interfaces, and standards. Still others were "watershed issues" that were debated by the working group and resolved in ways that guided the evolution of the language. We discuss several of these watershed issues in detail, including the alternatives that were considered and the reasons for the final resolution.

This chapter is based on the most recent XQuery specification at the time of publication. At this time, the broad outline of the language can be considered to be reasonably stable. However, readers should be cautioned that XQuery is still a work in progress, and the design choices discussed here are subject to change until the language has been approved and published as a W3C recommendation.

The Need for an XML Query Language

Early in its history, the XML Query Working Group confronted the question of whether XML is sufficiently different from other data formats to require a query language of its own. The SQL language [SQL99] is a very well established standard for retrieving information from relational databases and has recently been enhanced with new facilities called "structured types" that support nested structures similar to the **nesting** of elements in XML. If SQL could be further extended to meet XML query requirements, developers could leverage their considerable investment in SQL implementations, and users could apply the features of these robust and mature systems to their XML databases without learning a completely new language.

Given these incentives, the working group conducted a study of the differences between XML data and relational data from the point of view of a query language. Some of the significant differences between the two data models are summarized below.

- Relational data is "flat"—that is, organized in the form of a two-dimensional array of rows and columns. In contrast, XML data is "nested," and its depth of nesting can be irregular and unpredictable. Relational databases can represent nested data structures by using structured types or tables with **foreign keys**, but it is difficult to search these structures for objects at an unknown depth of nesting. In XML, on the other hand, it is very natural to search for objects whose position in a document hierarchy is unknown. An example of such a query might be "Find all the red things," represented in the XPath language [XPATH1] by the expression `//*[@color = "Red"]`. This query would be much more difficult to represent in a relational query language.
- Relational data is regular and homogeneous. Every row of a table has the same columns, with the same names and types. This allows **metadata**—information that describes the structure of the data—to be removed from the data itself and stored in a separate catalog. XML data, on the other hand, is irregular and heterogeneous. Each instance of a web page or a book chapter can have a different structure and must therefore describe its own structure. As a result, the ratio of metadata to data is much higher in XML than in a relational database, and in XML the metadata is distributed throughout the data in the form of tags rather than being separated from the data. In XML, it is natural to ask queries that span both data and metadata, such as "What kinds of things in the 2002 inventory have color attributes," represented in XPath by the expression `/inventory[@year = "2002"]/*[@color]`. In a relational language, such a query would require a join that might span several data tables and system catalog tables.
- Like a stored table, the result of a relational query is flat, regular, and homogeneous. The result of an XML query, on the other hand, has none of these properties. For example, the result of the query "Find all the red things" may contain a cherry, a flag, and a

stop sign, each with a different internal structure. In general, the result of an expression in an XML query may consist of a heterogeneous sequence of elements, attributes, and primitive values, all of mixed type. This set of objects might then serve as an intermediate result used in the processing of a higher-level expression. The heterogeneous nature of XML data conflicts with the SQL assumption that every expression inside a query returns an array of rows and columns. It also requires a query language to provide constructors that are capable of creating complex nested structures on the fly—a facility that is not needed in a relational language.

- Because of its regular structure, relational data is "dense"—that is, every row has a value in every column. This gave rise to the need for a "null value" to represent unknown or inapplicable values in relational databases. XML data, on the other hand, may be "sparse." Since all the elements of a given type need not have the same structure, information that is unknown or inapplicable can simply not appear. This gives an XML query language additional degrees of freedom for dealing with missing data. The XQuery approach to representing unknown or inapplicable data is discussed in Issue 2 under "Watershed Issues" below.
- In a relational database, the rows of a table are not considered to have an ordering other than the orderings that can be derived from their values. XML documents, on the other hand, have an intrinsic order that can be important to their meaning and cannot be derived from data values. This has several implications for the design of a query language. It means that queries must at least provide an option in which the original order of elements is preserved in the query result. It means that facilities are needed to search for objects on the basis of their order, as in "Find the fifth red object" or "Find objects that occur after this one and before that one." It also means that we need facilities to impose an order on sequences of objects, possibly at several levels of a hierarchy. The importance of order in XML contrasts sharply with the absence of intrinsic order in the relational data model.

The significant data model differences summarized above led the working group to decide that the objectives of XML queries could best be served by

designing a new query language rather than by extending a relational language. Designing a query language for XML, however, is not a small task, precisely because of the complexity of XML data. An XML "value," computed by a query expression, may consist of zero, one, or many items, each of which may be an element, an attribute, or a primitive value. Therefore, each operator in an XML query language must be well defined for all these possible inputs. The result is likely to be a language with a more complex semantic definition than that of a relational language such as SQL.

Basic Principles

The XML Query Working Group did not draw up a formal list of the principles that guided the design of XQuery. Nevertheless, throughout the design process, a reasonably stable consensus existed in the working group about at least some of the principles that should underlie the design of an XML query language. Some of these principles were mandated by the charter of the working group, and others arose from strongly held convictions of its members. The following list is my own attempt to enumerate the basic ideas and principles that were most influential in shaping the XQuery language. Tension exists among some of these principles, and several design decisions were the result of an attempt to find a reasonable compromise among conflicting principles.

- *Compositionality:* Perhaps the longest-standing principle in the design of XQuery is that XQuery should be a functional language incorporating the principle of **compositionality**. This means that XQuery consists of several kinds of expressions, such as path expressions, conditional expressions, and element constructors, that can be composed with full generality. The result of any expression can be used as the operand of another expression. No syntactic constraints are imposed on the ways in which expressions can be composed (though the language does have some **semantic constraints**). Each expression returns a value that depends only on the operands of the expression, and no expression has any side effects. The value returned by the outermost expression in a query is the result of the query.

- *Closure*: XQuery is defined as a transformation on a data model called the **Query data model**. The input and output of every query or subexpression within a query each form an instance of the Query data model. This is what is meant by the statement that XQuery is *closed* under the Query data model. The working group spent considerable time on the definition of the Query data model and on how instances of this model can be constructed from input XML documents and/or serialized in the form of output XML documents.
- *Schema conformance*: Since XML Schema has recently been adopted as a W3C Recommendation, the working group considered it highly desirable for XQuery to be based on the type system of XML Schema. This constraint strongly influenced the design of XQuery by providing a set of **primitive types,** a type-definition facility, and an **inheritance** mechanism. The validation process defined by XML Schema also strongly influenced the XQuery facilities for constructing new elements and assigning their types. Nevertheless, members of the working group attempted to modularize the parts of the language that are related to type definition and validation, so that XQuery could potentially be used with an alternative schema language at some future time.
- *XPath compatibility*: Because of the widespread usage of XPath in the XML community, a strong effort was made to maintain compatibility between XQuery and XPath Version 1.0. Despite the importance of this goal, it was necessary in a few areas to compromise compatibility in order to conform to the type system of XML Schema, because the design of XPath Version 1.0 was based on a much simpler type system.
- *Simplicity*: Many members of the working group considered simplicity of expression and ease of understanding to be primary goals of our language design. These goals were often in conflict with other goals, resulting in some painful compromises.
- *Completeness*: The working group attempted to design a language that would be complete enough to express a broad range of queries. The existence of a well-motivated use case was considered a strong argument for inclusion of a language feature. The expressive power

of XQuery is comparable to the criterion of "relational completeness" defined for database query languages [CODD], though no such formal standard has been defined for an XML data model. Informally, XQuery is designed to be able to construct any XML document that can be computed from input XML documents using the power of the first-order predicate calculus. In addition, recursive functions add significant expressive power to the language.

- *Generality*: XQuery is intended for use in many different environments and with many kinds of input documents. The language should be applicable to documents that are described by a schema, or by a Document Type Definition [XML], or by neither. It should be usable in strongly typed environments where input and output types are well known and rigorously enforced, as well as in more dynamic environments where input and output types may be discovered at execution time and some data may be untyped. It should accommodate input documents from a variety of sources, including XML files discovered on the Web, repositories of prevalidated XML documents, streaming data sources such as stock tickers, and XML data synthesized from databases.
- *Conciseness*: In the interest of conciseness, the semantics of the XQuery operators were defined to include certain implicit operations. For example, arithmetic operators such as +, when applied to an element, automatically extract the numeric value of the element. Similarly, comparison operators such as =, when applied to sequences of values, automatically iterate over the sequences, looking for a pair of values that satisfies the comparison (this process is called **existential quantification**). These implicit operations are consistent with XPath Version 1.0 and were preferred over a design that would require each operation to be explicitly specified by the user.
- *Static analysis*: From the beginning, the processing of a query was assumed to consist of two phases, called *query analysis* and *query evaluation* (roughly corresponding to compilation and execution of a program.) The analysis phase was viewed as an opportunity to perform **optimization** and to detect certain kinds of errors. A great deal of effort went into defining the kinds of checks that could be performed during the analysis phase and in deciding which of these checks should be required and which should be permitted.

The Query Data Model

The first step in designing XQuery was to specify the data model on which the language operates. The Query data model [XQ-DM] represents XML data in the form of nodes and values, which serve as the operands and results of the XQuery operators. XQuery is closed under the Query data model, which means that the result of any valid XQuery expression can be represented in this model. Since all the operators and expressions of XQuery are defined in terms of the Query data model, understanding this model is the key to understanding the language.

In defining the Query data model, the working group did not intend to deviate from existing standards, but to conform to them wherever possible. Therefore, the Query data model draws from several previously existing specifications. The information that results from parsing an XML document is specified by the XML **Information Set** [INFOSET], in the form of a collection of **information items.** The XML Information Set (or Infoset) contains no type information, and it represents data at a very primitive level—for example, every character has its own information item. XML Schema specifies an augmented form of the XML Infoset called the **Post-Schema Validation Infoset,** or PSVI. In the PSVI, information items that represent elements and attributes have type information and normalized values that are derived by a process called *schema validation*. The PSVI contains all the information about an XML document that is needed for processing a query, and the Query data model is based on the information contained in the PSVI.

For reasons described in the next section, the working group decided early to include the existing language XPath [XPATH1] as a subset of XQuery. XPath provides a notation for selecting information within an existing XML document, but it does not provide a way to construct new XML elements. Section 5 of the XPath specification shows how to represent the information in the XML Infoset in terms of a tree structure containing seven kinds of nodes. The operators of XPath are defined in terms of these seven kinds of nodes. In order to retain the original XPath operators and still take advantage of the richer type system of XML Schema, the XQuery designers decided to augment the XPath data model with the additional type information contained in the PSVI. The result of this process is the Query data model. The Query data model can be thought of as representing the PSVI in the form of a **node hierarchy,**

much as the XPath data model represents the XML Infoset in the form of a node hierarchy.

In the Query data model, every value is an ordered sequence of zero or more **items.** An item can be either an *atomic value* or a *node*. An atomic value has a *type*, which is one of the atomic types defined by XML Schema or is derived from one of these types by restriction. A node is one of the seven kinds of node defined by XPath, called document, element, attribute, text, comment, processing instruction, and namespace nodes. Nodes have identity, and an ordering called *document order* is defined among all the nodes that are in scope.

An instance of the Query data model may contain one or more XML documents or fragments of documents, each represented by its own tree of nodes. The **root node** of the tree that represents an XML document is a document node. Each element in the document is represented by an element node, which may be connected to attributes (represented by attribute nodes) and content (represented by text nodes and nested element nodes). The primitive data in the document is represented by text nodes, which form the leaves of the node tree.

Figure 2.1 illustrates the Query data model representation of a simple XML document. Nodes are represented by circles labeled *D* for document nodes, *E* for element nodes, *A* for attribute nodes, and *T* for text nodes. The XML document represented by Figure 2.1 is shown in Listing 2.1:

Listing 2.1 XML Document Represented by Figure 2.1

```
<?xml version="1.0" ?>
<procedure title="Removing a light bulb">
   <time unit="sec">15</time>
   <step>Grip bulb.</step>
   <step>
      Rotate it
      <warning>slowly</warning>
      counterclockwise.
   </step>
</procedure>
```

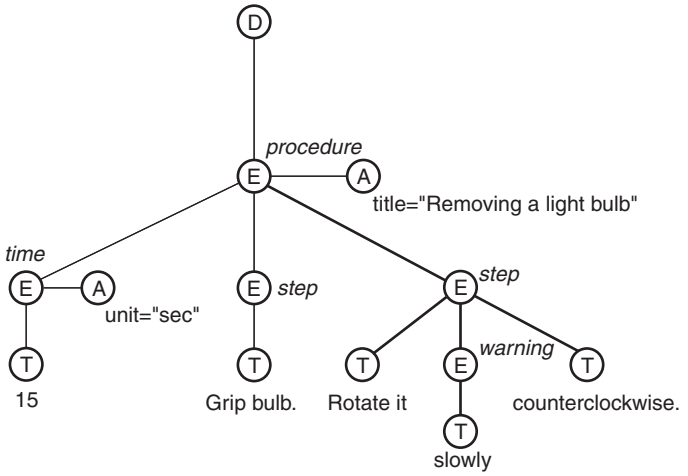

Figure 2.1 Example of the Query Data Model

In the Query data model, each element or attribute node has a name, a string value, a **type annotation,** and a typed value. These properties are not independent. The type annotation of an element represents its type as determined by the schema validation process. An element that has not been validated, or for which no more specific type is known, has the type annotation `xs:anyType`, where `xs:` is a prefix representing the namespace of XML Schema. If an element has no descendant elements, then its typed value can be derived from its string value and its type annotation.

Bear in mind that the type of an element describes the potential content of the element and does not depend on the name of the element. For example, two elements named `cost` and `price` could both have the type annotation `decimal` because they both require decimal content. Similarly, two elements named `shipto` and `billto` could both have the type annotation `address`, which might be a complex type defined in a schema that describes the potential content of the elements.

XQuery is defined as a transformation from one instance of the Query data model to another instance of the Query data model. This simplifies the definition of XQuery but leaves open the issues of where input data

comes from and how output data is delivered to applications. A query gains access to input data by calling an XQuery input function such as `doc` or `collection`, or by referencing some part of the external context (such as a prebound variable or "current node"). Each of these input methods is defined to return a Query data model instance in the form of one or more node hierarchies. One way in which a node hierarchy could be created is by parsing an XML document, validating it against a known or default schema, and converting the resulting PSVI into the Query data model as described in [XQ-DM]. Another way is for a system to store XML documents in a pre-validated form so that their Query data model representation can be materialized quickly on demand. A third way is for the Query data model to be synthesized directly from some data source such as a relational database, deriving its type information from "metadata" in the database catalog.

The process of serializing a Query data model instance as a linear XML document remains unspecified at present. All XML documents can be represented using the Query data model, but not all instances of the Query data model are valid XML documents. For example, the result of a query might be a sequence of atomic values, or an attribute that is not attached to any element. Mechanisms for serializing these values and for binding them to variables in a host programming language remain to be specified.

Related Languages and Standards

The designers of XQuery did not begin with a completely blank slate. The design of XQuery was strongly constrained by the requirement for compatibility with established standards and was also influenced by the design of other query languages with which the members of the working group were familiar. This section describes some of the ways in which XQuery was influenced by related languages and standards.

XML and Namespaces

Since XQuery is a query language for XML data sources, it is obvious that the language must be strongly influenced by the structure of XML itself

[XML]. From XML comes the notion that information is represented as a hierarchy of elements that have names, optional attributes, and optional content. The content of an element may consist of text, nested elements, or some mixture of these. The content of an XML document has an intrinsic order, and it is often important to preserve this order.

XQuery was also influenced by some of the lexical conventions of XML. Since XML is a **case-sensitive** language, it was decided that XQuery should be case-sensitive also. Since XML allows a hyphen to be used as part of a name, XQuery adopted the same convention. A consequence of this convention is that whitespace is sometimes significant in XQuery expressions. For example, spaces are used to distinguish the arithmetic expression `a - b` from the name `a-b`.

One important feature of an XML query language is the ability to construct an element with a given name and content. One of the ways in which XQuery supports element construction is by using XML notation. An XQuery element constructor can consist of a start tag and an **end tag,** enclosing character data that is interpreted as the content of the element, as illustrated in the following example:

```
<price>15.99</price>
```

Within an element constructor, curly braces are used to enclose expressions that are to be evaluated rather than treated as text, as in the following example, which computes a price from two variables named `$regprice` and `$discount`:

```
<price>{$regprice - $discount}</price>
```

Namespaces are very important to XML because they define the structure of an XML name. Namespaces provide a way for XML applications to be developed independently while avoiding the risk of name collisions. **Qualified names** *(QNames)*, as defined in the XML Namespaces specification [NAMESP], are used as the names of XML elements and attributes. XQuery also uses *QNames* as the names of functions, types, and variables. A *QName* consists of two identifiers, called the *namespace prefix* and the *local part*, separated by a colon. The namespace prefix and colon are optional. If present, the namespace prefix must be bound to a Uniform Resource Identifier (URI) that uniquely identifies a namespace.

As an example, suppose that the namespace prefix `student` is bound to the URI `http://example.org/student`, and the namespace prefix `investment` is bound to the URI `http://example.org/investment`. Then the *QNames* `student:interest` and `investment:interest` are recognized as distinct names even though their local parts are the same.

XQuery provides two ways of binding a namespace prefix to a URI. The first of these is by a declaration in the **prolog,** a part of a query that sets up the environment for query execution. Namespace prefixes declared in the prolog remain in scope throughout the query. This method of declaring a namespace prefix is illustrated by the following example:

```
declare namespace student = "http://example.org/student"
```

The second way to bind a namespace prefix to a URI in XQuery can be used when an element is constructed and defines a namespace prefix for use within the scope of the element. This method relies on an attribute with the prefix `xmlns`, which indicates that the attribute is binding a namespace prefix. For example, in the following start tag, the attribute named `xmlns:student` binds the namespace prefix `student` to a given URI within the scope of a constructed element named `school`:

```
<school xmlns:student = "http://example.org/student">
```

XQuery allows a user to specify, in the prolog, default namespace URIs to be associated with *QNames* that have no namespace prefix. Separate default namespaces can be specified for names of functions and for names of elements and types.

XQuery also provides a set of predefined namespace prefixes that can be used in any query without an explicit declaration. For example, the prefix `xs` is automatically bound to the namespace of XML Schema, so it is easy to refer to the names of built-in schema types such as `xs:integer`. Similarly, the prefix `fn` is automatically bound to the namespace of the XQuery core function library [XQ-FO], so it is easy to refer to the names of built-in XQuery functions such as `fn:max` and `fn:string`. If a query does not declare otherwise, the default namespace for function names is the namespace of the XQuery core function library (also bound to the prefix `fn`.)

XML Schema

As noted earlier, one of the major goals of the XML Query Working Group has been to define a query language based on the type system of XML Schema. This goal was made more difficult by the fact that XML Schema was designed to support validation of documents rather than to serve as the type system for a query language.

XML Schema has had a strong impact on XQuery because its type system is quite complex and includes some unusual features. The influences of XML Schema on the design of XQuery include the following:

- In XQuery, there is no distinction between a single value and a sequence of length one. To state this rule in another way, all XQuery values are sequences of length zero, one, or more. This rule arises from the XML Schema "facets" named `minOccurs` and `maxOccurs`, which can be attached to a part of a schema in order to constrain its number of occurrences. For example, in a schema, an **element declaration** without any occurrence constraints is considered identical to an element declaration with the facets `minOccurs="1"` and `maxOccurs="1"`; both declarations specify a sequence of elements of length one. Since all XQuery values are sequences, each parameter of a function can potentially accept a sequence of multiple items. For example, the function call `foo(1, (2, 3), ( ))` invokes a function with three arguments: the first argument is a sequence of length one, the second argument is a sequence of length two, and the third argument is a sequence of length zero.
- In XQuery, there is no notion of nested sequences—that is, a sequence directly containing another sequence as one of its members. The members of an XQuery sequence are always nodes or atomic values. A node, however, may in turn have another sequence as its content. These rules are derived from XML Schema, in which the content of an element is always a "flat" sequence of atomic values and other elements (which may, in turn, have content of their own).
- In XQuery, sequences may be heterogeneous—that is, a sequence may contain mixtures of nodes and atomic values, and may contain

atomic values of different types. Again, these rules are derived from XML Schema, in which the content of an element can be declared to be "mixed" (that is, consisting of a mixture of text and nested elements), and a sequence can contain values that conform to a "choice" of types.

- The working group chose to rely on the features of XML Schema for defining and naming complex types. As a result, XQuery depends on the conventions of XML Schema for associating names with types. Unfortunately, in XML Schema, some types have no name, and many types may have the same name. XML Schema does not, in general, provide a unique way to refer to a user-defined type. The impact on XQuery of the XML Schema naming system is discussed under Issue 3 ("What Is a Type?") in the "Watershed Issues" part of this chapter.
- XML Schema defines two different forms of type inheritance, called **derivation by extension** and **derivation by restriction,** and also introduces the concept of a *substitution group*, which can allow one kind of element to substitute for another kind of element based on its name. This combination of features has added a considerable amount of complexity to the syntax of XQuery. For example, as a step in a path expression, the name `frog` refers to an element whose name is exactly `frog`, but the notation `element(frog)` refers to an element that is either named `frog` or is in the same substitution group as the element named `frog`.

XML Schema defines a large set of built-in primitive types and an additional set of built-in **derived types.** In general, XQuery operators are defined on the primitive types of XML Schema, and operators on derived types are defined to promote their operands to the nearest primitive type. However, an exception to this rule was made for the type `integer`. Although `integer` is considered by most languages to be a primitive type, XML Schema considers it to be derived from `decimal`. If the general rule of promoting derived types to their primitive base types were applied to integers, arithmetic operations on integers such as `2 + 2` would return decimal results. As a consequence, an expression such as `2 + 2` would raise a type error when used in a function call where an integer is expected. In order to avoid these type errors, operations on integers in XQuery are

defined to return integers even though XML Schema considers `integer` to be a derived type.

XML Schema defines a `duration` type, which consists of six components named `year`, `month`, `day`, `hour`, `minute`, and `second`. This definition ignores the experience of the relational database community, which has discovered that neither comparison nor arithmetic operators can be supported by a duration type defined in this way. The following questions illustrate the problems encountered by operations on the `duration` type of XML Schema: Which is greater, one month or thirty days? What is the result of dividing one month by two? To deal with these problems, the SQL Standard in 1992 [SQL92] introduced two datatypes called a "year-month interval" and a "day-time interval." Each of these supports a well-defined set of arithmetic and comparison operators, but they cannot be mixed in a single expression. In order to facilitate arithmetic and comparison operations on dates, times, and durations, XQuery followed the practice of SQL in defining subtypes of `duration` called `xdt:yearMonthDuration` and `xdt:dayTimeDuration` (`xdt` is a predefined namespace prefix that represents the namespace containing all new datatypes defined by the XQuery specification).

Following the mandate of its charter, the working group designed XQuery to be fully compatible with XML Schema. At the same time, the group attempted to design XQuery in a way that would not preclude its adaptation to alternative schema definition languages. XQuery might be viewed as relying on an external schema facility for defining types and type hierarchies and for determining the type of a given element (in XML Schema, this process is called *validation*). To the extent that a schema facility meets these requirements, it can be considered compatible with XQuery.

XPath

Comparable to XML Schema in its influence on the design of XQuery is XPath [XPATH1], which has been a W3C Recommendation since November 1999. XPath is widely used in the XML community as a compact notation for navigating inside XML documents, and it is an integral part of other standards, including XSLT [XSLT] and XPointer [XPTR].

The functionality of XPath is clearly needed as part of an XML query language, and there is a clear precedent that this functionality should be expressed using the syntax of XPath Version 1. Therefore, from the beginning, compatibility with XPath Version 1 was a major objective and constraint on the design of XQuery.

Initially, the Query working group considered using the path expression of XPath as a "leaf expression" in the XQuery syntax—that is, as a primitive form of expression that could be used as an operand in higher-level XQuery expressions but could not in turn contain other XQuery operators. However, at the same time that XQuery was being designed, the XSLT working group had collected a set of requirements for new functionality in XPath [XPATH2REQ], and these requirements overlapped substantially with the functionality proposed for XQuery. As a result, it was decided that a new version of XPath would be developed jointly by the XSLT and Query working groups. The new version, to be called XPath Version 2 [XPATH2], would be a syntactic subset of XQuery, would be backward-compatible with XPath Version 1, and would be available for use in XSLT and other standards. XPath Version 2 would include many of the features of XQuery and would be fully integrated with the rest of the XQuery syntax rather than serving as a non-decomposable "leaf expression."

Types in XPath

For XPath to be used in a query language based on the type system of XML Schema, its own type system had to be revised. XPath Version 1 recognized only four types: Boolean, string, "number" (a double-precision, floating numeric type), and "node-set" (an unordered collection of nodes). XPath Version 1 was designed with a very permissive view of types, in which conversions of one type to another could be done with very few limitations. For example, if a node-set is encountered where a number is expected, the string value of the first node in the node-set (in document order) is extracted and cast into a number. These permissive rules were deliberately designed to minimize the likelihood of non-recoverable errors during the processing of path expressions, which are often used in rendering web pages by a browser or in other contexts where run-time errors are unwelcome.

From a type system based on only four types, XPath had to be adapted to the type system of XML Schema, which included forty-four built-in types and a complex set of rules for defining additional types, encompassing atomic, simple, complex, primitive, derived, list, union, and anonymous types, as well as two forms of inheritance, twelve "constraining facets," substitution groups, and various other features. Also, from a very permissive set of type-conversion rules, XPath had to be adapted to a philosophy of strict typing, including both static and dynamic type-checking. Users of XPath were assured that these changes constituted an improvement. Adaptation of XPath to the XML Schema type system also provided an opportunity to make a very small number of incompatible changes to the semantics of the language.

Syntax and Semantics

The adoption of XPath as a subset had significant effects on both the syntax and semantics of XQuery, including the following:

- Since XPath uses the symbol `/` in path expressions, it is not available for use as a division operator. XQuery adopted the XPath operator `div` for division, supplemented by a new `idiv` operator for division of integers, returning an integer.
- XPath has a few keywords, such as `and`, `or`, `div`, and `mod`, but none of these keywords are **reserved words.** This means that the XPath grammar is defined in such a way that an XPath expression can search for an element named (for example) `mod`, without confusing the element name with the keyword. It is obviously desirable to avoid any limitation on names that can be searched for in documents. One way to accomplish this would be to require a special "escape" syntax to be used with names that are the same as keywords (one alternative that was considered was to prefix these names with a colon). But for compatibility with XPath Version 1, it was decided that XQuery should have no reserved keywords and no special syntax for names. This was accomplished by careful grammar design and by defining some rules for "lookahead" during the process of converting a query from a stream of characters into grammatical tokens.

- The concept of *document order* is very important to the definition of several XPath operators, and as a result it also plays an important role in XQuery semantics, as described below.

Document order is an ordering that is defined among all the nodes in the Query data model representation of a document. As defined in XPath, document order corresponds to the order in which the XML representations of the various nodes would be encountered if the document were to be serialized in XML format. In other words, each element node is followed by its namespace nodes, its attribute nodes, and its children (text, element, comment, and processing instruction nodes) in the order in which they naturally appear in the document. Reverse document order is defined as the reverse of document order.

One of the defining features of XPath is the *path expression*, which consists of a series of *steps*, each of which selects a set of nodes. In the set of nodes selected by a step, each node has a *position* based on its relationship to the other nodes in (forward or reverse) document order. In effect, the set of nodes resulting from each step must be sorted on the basis of document order, a potentially expensive process. The idea of sorting intermediate results was particularly unfamiliar to people with a background in relational databases, in which sets of data values have no intrinsic order. The working group briefly considered relaxing this requirement and allowing the nodes selected by each step to remain in the order in which they were generated (based on iterating over the nodes selected by the previous step). This idea was put to rest by an example suggested by Michael Kay. The example is based on the following input document:

```
<warning>
<p>
Do <emph>not </emph> touch the switch.
The computer will <emph>explode!</emph>
</p>
</warning>
```

The representation of this input document in the Query data model is shown in Figure 2.2, in which element nodes are represented as circles labeled *E*, and text nodes are represented as circles labeled *T*.

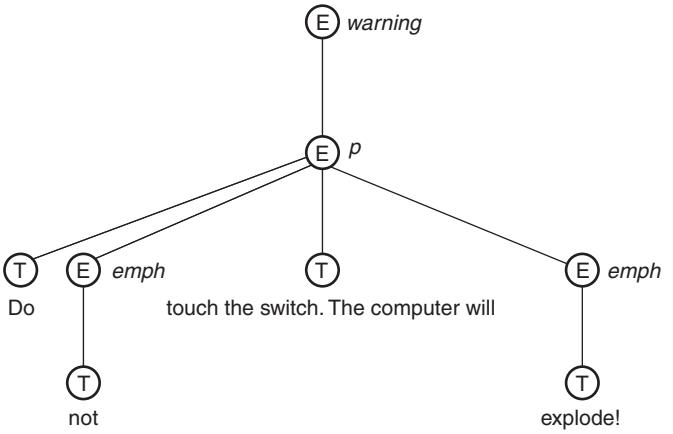

Figure 2.2 Query Data Model Representation of "Warning" Document

Against this input document we wish to execute the following path expression:

```
/warning//text()
```

In XPath Version 1, this path expression would return all the text nodes that are descendants of the `warning` element, in document order. The concatenated content of these nodes is as follows:

```
Do not touch the switch. The computer will explode!
```

It is interesting to consider how the result of this path expression would change if each step in the path preserved the order of nodes generated by the previous step rather than sorting its results in document order. Under these rules, the path expression would be executed as follows:

1. The first step, `/warning`, returns the top-level element node, which has the name `warning`.
2. The notation `//` is an abbreviation for a second step, which in fully expanded form might be written as `/descendant-or-self::node()/`. This step returns the `warning` element node

returned by the previous step, and all its descendants—in other words, all eight of the nodes shown in Figure 2.2, in document order.

3. The third step, `text()`, uses the default child axis of XPath to return text nodes that are children of the nodes returned by the previous step. Only element nodes have children that are text nodes. Processing the element nodes returned by the previous step, in order, leads to the following result (ignoring some subtle issues relating to the handling of whitespace):

   The first element node to be processed is named `p` and has two text node children, containing the strings `"Do"` and `"touch the switch. The computer will"`.

   The second element node to be processed is named `emph` and has one text node child, containing the string `"not"`.

   The third element node to be processed is named `emph` and has one text node child, containing the string `"explode!"`.

The concatenated contents of the text nodes returned by the final step is as follows:

```
Do touch the switch. The computer will not explode!
```

This example is a good illustration of how processing documents places some requirements on a query language that are beyond the scope of a traditional database query language. The Query working group included representatives from both the database and document processing communities, and these individuals had much to learn from each other during the process of designing XQuery.

Predicates

XPath has several kinds of *predicates*, which are tests that are used to filter sequences of nodes. All of these predicates have the general form `E1[E2]`, in which the expression `E2` is used to filter the items in the sequence generated by expression `E1`. The different kinds of XPath predicates are illustrated by the following examples:

- `/employee[salary > 1000]` selects `employee` elements that have a `salary` **subelement** with value greater than 1000.
- `/employee[5]` selects the fifth `employee` element in a sequence.
- `/employee[secretary]` selects `employee` elements that have a `secretary` subelement.

For XQuery, it was necessary to preserve all these kinds of predicates, but to generalize their definitions so that the value of an expression (either `E1` or `E2` in the above format) could be a heterogeneous sequence of nodes and atomic values. In XQuery, `E1[E2]` is defined as follows: For each item `e1` in the sequence returned by `E1`, the expression `E2` is evaluated with `e1` as the **context item** (the context item serves as the "starting point" for a path expression). For a given `e1`, if `E2` returns a number `n`, the value `e1` is retained only if its ordinal position in the `E1`-sequence is equal to `n`. Otherwise, `e1` is retained only if the **Effective Boolean Value** of `E2` is `true`. Effective Boolean Value is defined to be `false` for an empty sequence and for the following single atomic values: the Boolean value `false`, a numeric or binary zero, a zero-length string, or the special float value `NaN`. Any other sequence has an Effective Boolean Value of `true`. Note especially that the Effective Boolean Value of any node is `true`, regardless of its content, and the Effective Boolean Value of any sequence of length greater than one is `true`, regardless of its content.

This definition of Effective Boolean Value is used not only in predicates but also in other parts of XQuery where it is necessary to reduce a general sequence to a Boolean value (for example, in conditional expressions and quantified expressions). The definition was arrived at by considerations of XPath Version 1 compatibility, logical consistency, and performance. This definition has the desirable property that the Effective Boolean Value of a sequence of arbitrary length depends only on the value of the first item in the sequence and the existence (but not the values) of additional items. It also has the surprising property that an element with the Boolean content `false` has an Effective Boolean Value of `true` (as required for compatibility with XPath Version 1). Another surprising property of this definition is that a sequence of atomic values, all of which are `false`, has the Effective Boolean Value of `true` (because it contains more than one item).

Implicit Operations and Transitivity

XPath Version 1 is defined to perform many implicit conversions during the processing of an expression. Some of these conversions are illustrated by the following example:

```
//book[author = "Mark Twain"]
```

On the left side of the = operator, we find `author`, which denotes a sequence of zero or more element nodes. On the right side of the = operator, we find `"Mark Twain"`, which is a string. Since a sequence of zero or more nodes is not the same thing as a string, these expressions are made comparable by the following implicit actions:

- The values of the `author` nodes are extracted and treated as atomic values.
- Since these atomic values are being compared with a string, they are treated as strings.
- If there is more than one `author` value, an implied existential quantifier is inserted, so the predicate is considered to be true if any `author` value is equal to the string `"Mark Twain"`.

These implicit actions make the above expression equivalent to the following expression in which the same actions are represented explicitly:

```
//book[some $a in ./author satisfies string(data($a)) = "Mark Twain"]
```

In keeping with the basic principle of conciseness as well as the principle of backward compatibility, XQuery preserved these implicit XPath conversions and in fact extended them in a uniform way to apply to other parts of the XQuery language. The extraction of atomic values from nodes is called *atomization* in XQuery, and is applied to sequences as well as to individual nodes. For example, the expression `avg(/employee/salary)` extracts numeric values from a sequence of `salary` nodes before applying the `avg` function.

The implicit conversions described above led to a serious concern for the designers of XQuery: They caused the comparison operators such as = and > to lack the **transitivity** property. Thus, if `$book1/author = $book2/author` is `true`, and `$book2/author = $book3/author` is `true`, it is not possible to conclude that `$book1/author = $book3/author` is

`true`. For example, this inference would fail if `$book1/author` has the value ("Billy", "Bonnie"), `$book2/author` has the value ("Bonnie", "Barry"), and `$book3/author` has the value ("Barry", "Benny").

Transitivity is a useful property for comparison operators. For example, transitivity of equality comparisons is required for certain kinds of query transforms that are useful in optimization. Transitivity of a comparison operator is also required if the operator is to serve as the basis for imposing a global ordering on a sequence of values. Since the six *general comparison* operators (`=`, `!=`, `>`, `>=`, `<`, and `<=`) lack transitivity, the designers of XQuery decided to supplement them with six more primitive *value comparison* operators (`eq`, `ne`, `gt`, `ge`, `lt`, and `le`) that have the transitivity property. These primitive comparison operators can be used to compare single atomic values, but they raise an error if either of the operands to be compared is not a single value. The value comparison operators always treat an untyped operand as a string.

Incompatible Changes

Despite the general objective of backward compatibility, a small number of XPath Version 1 features remained unacceptable to the designers of XQuery. Some of these features were considered important for other usages of XPath, such as XSLT style sheets that had been written to exploit these features. To deal with this problem, an "XPath Version 1 compatibility mode" was defined for XPath Version 2. When embedded in XQuery, XPath Version 2 will not run in compatibility mode, and the semantics of certain operators will be different from those of XPath Version 1. Other host environments of XPath Version 2, such as XSLT, are free to interpret XPath Version 2 in compatibility mode to preserve the semantics expected by existing applications. The cases in which compatibility mode influences the semantics of XPath Version 2 include the following:

- In XPath Version 1, if an operand of an arithmetic operator such as `+` is a sequence containing more than one node, the numeric value of the first node in the sequence is extracted and used as the operand. In XQuery, this case is treated as an error.
- In XPath Version 1, inequality operators on strings cast their operands to `double`, but the equality operator on two strings performs a string comparison. This leads to the surprising result that

`"4" <= "4.0"` and `"4" >= "4.0"` are both `true`, but `"4" = "4.0"` is `false`. In XQuery, all comparison operators on strings perform string comparisons without attempting to convert their operands to numbers.

- In XPath Version 1, arithmetic operators can be applied to strings and implicitly convert their operands to `double`. For example, the expression `"1" + "2"` returns `3.0E0`. In XQuery, arithmetic on strings is treated as an error.
- In XPath Version 1, the `=` operator on two elements compares their string values, ignoring nested markup. For example, a book with title `"Tom Sawyer"` and author `"Mark Twain"` is considered to be equal (by the `=` operator) to a book with author `"Tom Sawyer"` and title `"Mark Twain"`. In XQuery, applying the `=` operator to two elements whose content consists entirely of subelements is treated as an error. However, XQuery provides several functions, such as `fn:deep-equal`, that can be used to perform various kinds of comparisons between element nodes. Comparison of the string values of two element nodes can be done by extracting their string values, using the `fn:string` function, as in `fn:string($node1) = fn:string($node2)`.

Other Query Languages

The influence of other languages on XQuery is not limited to directly related standards such as XPath and XML Schema. XQuery has also been strongly influenced by other query languages used in both the database and information retrieval communities. In several cases, designers of these precursor languages have also contributed to the design of XQuery.

The immediate ancestor of XQuery is **Quilt** [QUILT], a language proposal submitted to the XML Query Working Group by three of its participants in June 2000. Quilt provided the basic framework of XQuery as a functional language based on several types of composable expressions, including an iterative expression and an element constructor. The FLWOR expression, one of the most important of the XQuery expression types, was adopted from Quilt (though the original Quilt version did not have an `order-by` clause). From its origin in the Quilt proposal, XQuery has evolved by changing the syntax for element constructors, adding a more complex syntax for declaring the types of function parameters, and

adding some new kinds of expressions, such as `validate`, `instance-of`, and `typeswitch`. The XQuery specification is also much more complete and rigorous than the original Quilt proposal in formally specifying the semantics of various kinds of expressions, as described in Chapter 4.

The Quilt proposal, in turn, reflects the influence of several other query languages. In fact, the name "Quilt" was intended to suggest that features had been patched together from a variety of sources to form a new language.

From SQL [SQL99] and from the relational database community in general, XQuery adopted an English-keyword notation and a rich collection of use cases. Many SQL facilities such as **grouping** and outer join have their counterparts in XQuery, though they are often expressed in different ways. The `select-from-where` query block of SQL has a rough analogy in the `for-let-where-order-return` (FLWOR) expression of XQuery, in the sense that both kinds of expression are used for both *selection* (retaining certain items while discarding others) and *projection* (retaining certain properties of the selected items while discarding others.) Some features of SQL, however, such as a special null value and three-valued logic for the `and` and `or` operators, were considered unnecessary in the XML context and were not duplicated in XQuery.

From XML-QL [XML-QL], Quilt (and hence XQuery) adopted the approach of binding variables to sequences of values and then constructing new output elements based on the bound variables. An XML-QL query consists of a `WHERE` clause followed by a `CONSTRUCT` clause. The result of the `WHERE` clause is a stream of tuples of bound variables. The `CONSTRUCT` clause is executed once for each of these tuples, generating a stream of output elements. As in XQuery, the combination of `WHERE` and `CONSTRUCT` is a nestable unit from which queries can be constructed. The creation of an ordered stream of tuples, and iteration over these tuples to generate output, have direct counterparts in the FLWOR expression of XQuery. XQuery also adopted from XML-QL the convention of prefixing variable names with a `$` sign.

From OQL [OQL], Quilt (and hence XQuery) adopted the approach of designing the language around a fully composable set of expressions, all of which use a common data model for their operands and results. The `select-from-where` expression of OQL, rather than providing a frame-

work for the whole language, as in SQL, is simply one of several independent expressions that include arithmetic and set operators, function calls, and universal and existential quantifiers. Similarly, in XQuery, the FLWOR expression is simply one of many types of expressions that can be combined in various ways. The atoms, structures, and collections in the OQL data model are suggestive of the atomic values, elements, and sequences in the Query data model (though the Query data model supports only one kind of collection). Interestingly, OQL (like Quilt) has a fully independent `sort-by` operator that can be applied to any sequence of values, whereas in XQuery the sorting facility is supported only as a clause inside the FLWOR expression, for reasons described under Issue 8 ("Ordering Operators") in the following section.

Watershed Issues

A large fraction of the total effort invested in the design of XQuery by the working group has been focused on a relatively small number of "watershed issues." These are issues whose resolution had a significant global impact on the language by influencing a number of related decisions. All of these issues are complex, and some of them were quite contentious. Eight of these watershed issues are discussed here; they are listed below.

1. *Handling of untyped data.* This issue deals with the kinds of operations that can be applied to data whose type is not known.
2. *Unknown and inapplicable data.* This issue deals with how unknown values can be encoded in XML and how various operators should be defined on these values.
3. *What is a type?* This issue deals with how a datatype is specified in XQuery.
4. *Element constructors.* This issue deals with the details of how to construct a new XML element by using a query expression.
5. *Static typing.* This issue deals with the kinds of type-checking that can be performed on a query independently of any input data.
6. *Function resolution.* This issue deals with how functions are defined and how function calls are processed.
7. *Error handling.* This issue deals with the kinds of errors that can be encountered in queries, and whether query expressions are

deterministic (that is, whether they always return the same result for a given input).

8. *Ordering operators.* This issue deals with the kinds of operators that are provided in the language for controlling the order of results.

Issue 1: Handling of Untyped Data

Despite publication of XML Schema as a W3C Recommendation in May 2001, a great many XML documents exist that are not described by a schema. Some of these documents have Document Type Definitions (DTDs), and some do not have a formal description of any kind. In the absence of a schema, the character data in an XML document does not have a well-defined type. A DTD might describe this data as CDATA ("character data") or PCDATA ("parsed character data"), but in fact it might represent either text or numeric data.

Many of the operators of XQuery require data of a particular type—for example, the arithmetic operators are defined on numeric data. It is highly desirable that these operators be usable for querying schema-less documents, even though the data in these documents is not explicitly declared to be of any particular type. The approach of requiring all untyped data to be explicitly cast into a specific type whenever it is referred to in a query was rejected as putting an unreasonable burden on the query writer. Instead, the designers of XQuery attempted to define a set of rules that allow types to be inferred for input data based on the context in which the data is used.

Consider the following fragment of an input document:

```
<employee>
   <name>Fred</name>
   <salary>5000</salary>
   <bonus>2000</bonus>
</employee>
```

The process of representing this input data in the Query data model causes each element node to receive a type annotation. If the input document is validated against a schema, the `employee` node might receive a type annotation of `employee-type`, and the `salary` and `bonus` nodes might receive a type annotation of `xs:decimal`.

Suppose that a query binds a variable named `$fred` to the `employee` element, and evaluates the expression `$fred/salary + $fred/bonus`. Each of the operands of the `+` operator evaluates to an element node whose type annotation is `xs:decimal`. The `+` operator extracts the decimal values from the nodes and adds them together. The result of the expression is an atomic value of type `xs:decimal`.

Now consider the following similar fragment from an input document that has no schema:

```
<a>
   <b>5</b>
   <c>17</c>
</a>
```

Syntactically, this fragment consists of an element containing two nested elements. In the absence of a schema, the elements are given the generic type annotation `xs:anyType`, which indicates that no type information is available. An atomic value occurring inside an untyped element, such as `5` or `17` in the above example, is given the type annotation `xdt:untypedAtomic`,[1] which denotes an atomic value for which no type is available.

In the above example, suppose that a query binds a variable named `$a` to the outer element, and executes the expression `$a/b + $a/c`. Looking at the input document, we see that the nested elements `b` and `c` contain numbers, and it seems reasonable that these numbers should be added together. But it is necessary to define the rules under which this operation is performed. As in the earlier example, the `+` operator extracts the typed values from the `b` and `c` nodes. The value of the `b` node is `5`, and the value of the `c` node is `17`, and the type of both values is `xdt:untypedAtomic`. Each operator in XQuery has a rule for how it handles values of type `xdt:untypedAtomic`. In the case of arithmetic operators such as `+`, the rule states that any operand of type `xdt:untypedAtomic` is cast to the

[1] The type `xdt:untypedAtomic` is closely related to the type `xs:anySimpleType`, which is the root of the simple type hierarchy in XML Schema [SCHEMA]. The type `xdt:untypedAtomic` is different from `xs:anySimpleType` in two ways: (1) `xs:anySimpleType` includes non-atomic values such as lists that are considered to be "simple" by XML Schema, whereas `xdt:untypedAtomic` includes only atomic values; and (2) `xs:anySimpleType` includes primitive types such as `xs:string` and `xs:decimal` as subtypes, whereas `xdt:untypedAtomic` has no subtypes. In general, an atomic value in a PSVI whose type property is `xs:anySimpleType` is represented in the Query data model with the type annotation `xdt:untypedAtomic`.

type `xs:double` before the addition is performed. If a value of type `xdt:untypedAtomic` cannot be successfully cast to the type `xs:double`, the arithmetic operator raises an error. In our example expression, the operands are successfully cast to the double-precision values `5.0E0` and `1.7E1`, and the result of the expression is `2.2E1`.

Various operators in XQuery have different rules for handling of values of type `xdt:untypedAtomic`. As we have seen, arithmetic operators cast `xdt:untypedAtomic` values to `xs:double`. General comparison operators such as `=` and `>`, on the other hand, cast operands of type `xdt:untypedAtomic` to the type of the other operand, and if both operands are of type `xdt:untypedAtomic`, they are both cast to the type `xs:string`. These rules reflect the facts that addition is defined only for numbers, but comparison can be defined for any string value. Casting `xdt:untypedAtomic` values into numbers before comparing them would preclude comparison of non-numeric strings.

Suppose that, instead of `$a/b + $a/c`, our example expression had been `$a/b > $a/c`. Since the type of both operands is `xdt:untypedAtomic`, `$a/b` is cast to the string `"5"` and `$a/c` is cast to the string `"17"`, and the strings are compared using a default collation order. The result depends on the default collation, but the expression is likely to return the value `true` since in most collations the string `"5"` is greater than the string `"17"`.

Each XQuery operator has its own rule for processing operands based on their types. For some operators, such as the general comparison operators, the rule depends on the **dynamic types** of both operands. These rules make XQuery by definition a dynamically dispatched language, meaning that, in general, the semantics of an operation are determined at run-time. However, the rules are such that, when operating on well-typed data that is described by a schema, static analysis can often determine the semantics of an operation at compile type and can use this information to execute the query efficiently (avoiding run-time branching).

The working group considered the alternative of treating all untyped data as strings. In the example `$a/b + $a/c` above, this design would have resulted in the `+` operator having strings as its operands. In order for this useful example to work, it would have been necessary for arithmetic operators to cast their string operands into a numeric type (say, `double`). Then `5 + "7"` would no longer be a type error, even though it adds a number to a string.

Similarly, if untyped data were to be treated as strings, comparison operators between numbers and strings would need to be defined. For example, consider the expression `age > 40`. In the absence of a schema, `age` is untyped. If untyped data is treated as a string, `age > 40` will give the expected result only if its string operand is cast to a number. But then `5 = "5"` and `5 = "05"` would be true rather than type errors (but `"5" = "05"` would be false). Also, then `5 > 17` and `"5" > 17` would be false but `"5" > "17"` would be true. It was felt that these rules would generate confusion and would compromise the strong typing of the language. Therefore, the working group chose to treat untyped data according to its context rather than always treating it as a string.

Issue 2: Unknown and Inapplicable Data

In the real world, data is sometimes unknown or inapplicable. Any data model needs to have a way to represent these states of knowledge.

In the relational data model, data is represented in a uniform way using rows and columns. Every row in a given table has the same columns, and there is exactly one value in every column. For example, if a table describing cars has a `mileage` column containing information about the expected miles per gallon of each car, then every row in that table must have some value in the `mileage` column, including rows that describe cars whose mileage is unknown and rows that describe electric cars for which the concept of mileage is inapplicable. This requirement gave rise to the "null" value in relational databases, an "out-of-range" value that is used to represent unknown or inapplicable data. In the relational query language SQL, arithmetic and comparison operators always return the null value if any of their operands is null. Since logical operators such as `and` and `or` must deal with the null value as well as the Boolean values `true` and `false`, SQL is said to use **three-valued logic.**

XML is considerably more flexible than the relational model when it comes to representing unknown or inapplicable data. In an XML Schema, a `car` element might be defined to contain `make` and `year` elements and an optional `mileage` element. The `mileage` element in turn might be defined to contain an optional decimal value. Then, any of the following is a valid XML representation of a `car` element:

The `mileage` element could be present and have a value:

```
<car><make>Toyota</make><year>2002</year><mileage>26</mileage></car>
```

The `mileage` element could be present but empty:

```
<car><make>Ford</make><year>2002</year><mileage/></car>
```

The `mileage` element could be absent:

```
<car><make>Porsche</make><year>2002</year></car>
```

XQuery leaves it up to the application to define the mapping of states of knowledge about cars onto various XML representations. For example, the absence of a `mileage` element might be used to represent the absence of any knowledge about mileage, whereas the presence of an empty `mileage` element might be used to represent the knowledge that mileage is inapplicable for the given car.

Because XML already provides a way to represent various states of knowledge about data, including unknown and inapplicable data, the designers of XQuery saw no need to introduce a special "out-of-range" value (which, in any case, would not have been consistent with XML Schema). Instead, XQuery focuses on defining the operators of the language in such a way that they return a reasonable and useful result for all of the possible syntactic states of their operands. This approach provides application developers with the tools they need to represent various knowledge states as they see fit.

The main operators of XQuery that operate on scalar values are arithmetic, comparison, and logical operators. These operators are also defined on nodes, but always begin by reducing their operands to scalar values by a process called *atomization*. The result of atomization on an absent or empty element is an empty sequence.

If arithmetic is performed on unknown or inapplicable data (for example, `mileage div cost` where `mileage` is unknown), the result should be unknown or inapplicable. Therefore, the arithmetic operators of XQuery are defined in such a way that if either of their operands (after atomization) is an empty sequence, the operator returns an empty sequence.

XQuery provides two sets of comparison operators. The general comparison operators, `=`, `!=`, `>`, `>=`, `<`, and `<=`, which are inherited from XPath Version 1, have an implied existential quantifier on both operands. For example, the expression `A = B`, if `A` and `B` are sequences of multiple values, returns `true` if some item in sequence `A` is equal to some item in sequence `B`. If either of their operands is an empty sequence, the general comparison operators return `false`. These operators (as in XPath Version 1) have the interesting property that the predicates `length > width`, `length < width`, and `length = width` can all be simultaneously `true` (if `length` or `width` contains multiple values), and can also all be simultaneously `false` (if `length` or `width` contains no values).

As noted earlier, XQuery also provides a set of value comparison operators, `eq`, `ne`, `gt`, `ge`, `lt`, and `le`, which require both of their operands to consist of exactly one atomic value. These operators raise an error if either of their operands is an empty sequence. These operators are designed to be "building blocks" with well-defined mathematical properties that can serve as the basis for defining more complex logical operations.

Some consideration was given to defining three-valued comparison operators that return an empty sequence (representing the unknown truth value) if either of their operands is an empty sequence. This approach was rejected because such operators would not be transitive. For example, consider the expression `salary eq 5 and bonus eq salary`. A query optimizer might wish to transform this expression by adding an additional predicate `bonus eq 5`, deduced by the transitivity property. This transformation is valid under **two-valued logic** but not under three-valued logic. For example, under three-valued logic, if `salary` is empty (unknown) and `bonus` has the value `7`, then the truth value of `salary eq 5 and bonus eq salary` would be unknown, but the truth value of `bonus eq 5` would be `false`. Since transitive comparisons were considered necessary for defining other operations, such as sorting and grouping, the value comparison operators were defined using two-valued logic.

The semantics of the logical operations `and`, `or`, and `not` in the presence of empty sequences are constrained by compatibility with XPath Version 1. In XPath, a predicate expression that returns an empty sequence is interpreted as `false` (as a failed test for the existence of something). For example, the

query `//employee[secretary]` searches for `employee` elements that have a `secretary` subelement; if the predicate expression `secretary` returns an empty sequence, the predicate is considered to be `false`. For compatibility with XPath Version 1, the `and` and `or` operators of XQuery (and the `not` function) treat the empty sequence as having a Boolean value of `false`.

In languages such as SQL that are based on three-valued logic, queries do not generally return data for which the search-condition evaluates to the unknown truth value. Therefore, the distinction between two-valued and three-valued logic is somewhat subtle. The difference can be illustrated by a query that performs negation on an intermediate result. Consider how the following two queries would operate on a `car` element that has no `mileage` subelement:

Query: Find cars whose mileage is less than or equal to 25.

```
//car[mileage <= 25]
```

Query: Find cars for which it is not true that mileage is greater than 25.

```
//car[not(mileage > 25)]
```

In SQL, if the value of `mileage` is null, the predicates `mileage <= 25` and `mileage > 25` both return the unknown truth value. The inverted predicate `not(mileage > 25)` also returns the unknown truth value. According to this logic, the `car` with no `mileage` subelement would not be returned by either query.

In XQuery, if `mileage` is an empty sequence, the predicates `mileage <= 25` and `mileage > 25` both return `false`, and `not(mileage > 25)` is `true`. Therefore the car with no mileage subelement would be returned by the second query above but not by the first query.

A language that wishes to define three-valued `and`, `or`, and `not` operators needs to specify the truth tables for these operators. One approach is to adopt the rule that `and`, `or`, and `not` return the unknown truth value if any of their operands is the unknown truth value. SQL, however, adopts a different approach, represented by the truth tables in Figure 2.3.

AND	T	F	?
T	T	F	?
F	F	F	F
?	?	F	?

OR	T	F	?
T	T	T	T
F	T	F	?
?	T	?	?

NOT	
T	F
F	T
?	?

Figure 2.3 Truth Tables in SQL

If an XQuery application needs three-valued logic, it can easily be simulated by writing functions that use the built-in XQuery operators. The value comparison operator `eq` can be supplemented by a function, possibly named `eq3`, that returns the empty sequence if either of its operands is empty. This function, and functions that implement the SQL truth tables of Figure 2.3, are illustrated in Listing 2.2.

Listing 2.2 Defined Function eq3 and Others Implementing the Truth Tables of Figure 2.3

```
define function eq3($a as xdt:anyAtomicType?, $b as
xdt:anyAtomicType?)
   as xs:boolean?
   {
    if (empty($a) or empty($b)) then ()
    else ($a eq $b)
   }

define function not3($a as xs:boolean?)
   as xs:boolean?
   {
    if (empty($a)) then ()
    else not($a)
   }

define function and3($a as xs:boolean?, $b as xs:boolean?)
   as xs:boolean?
   {
    if ($a and $b) then true()
    else if (not3($a) or not3($b)) then false()
    else ()
   }

define function or3($a as xs:boolean?, $b as xs:boolean?)
   as xs:boolean?
   {
    if ($a or $b) then true()
    else if (not3($a) and not3($b)) then false()
    else ()
   }
```

It is worth mentioning that XML Schema defines a special attribute named `xsi:nil`. This attribute affects only the process of Schema validation. It permits an element to be accepted as valid with no content, even though the element has a declared type that requires content. The `xsi:nil` attribute is treated by XQuery as an ordinary attribute. Its presence or absence does not affect the semantics of XQuery operators.

Issue 3: What Is a Type?

Query languages use types to check the correctness of queries and to ensure that operations are being applied to data in appropriate ways. The specification and handling of types has been one of the most complex and contentious parts of the XQuery design.

XML Schema provides a set of built-in types and a facility for defining and naming new types. In XML Schema, the name of an element or attribute is distinguished from the name of its type. For example, a schema might define an element named `shipto` with the type `address`. This means that the content of the `shipto` element is determined by the declaration of the `address` type—possibly consisting of subelements such as `street`, `city`, `state`, and `zipcode`. However, not all elements have a named type. For example, a schema might define an element named `offering` to contain subelements named `product` and `price` without giving a name to the type of the `offering` element. In this case, the `offering` element is said to have an *anonymous* type. Furthermore, an `offering` element might be defined to have different types in different contexts—for example, an `offering` in a `university` might be different from an `offering` in a `catalog`. XML Schema also allows an element, an attribute, and a type all to have the same name, so it is necessary to distinguish somehow among different usages of a name (for example, to distinguish an element named `price` from an attribute named `price`). All the names defined in a schema belong to an XML namespace called the *target namespace* of the schema.

XQuery allows the names of elements, attributes, and types that are defined in a schema to be used in queries. The prolog of a query explicitly lists the schemas to be imported by the query, identifying each schema by its target namespace. Importing a schema makes all the names defined in that schema available for use in the query. An error results if two or more imported schemas define the same name with conflicting meanings. The

prolog of a query can also bind a namespace prefix to the target namespace of an imported schema. For example, the following statement imports a schema and assigns the namespace prefix `po` to its target namespace:

```
import schema namespace po = "http://example.org/purchaseOrder"
```

As noted earlier, each element or attribute node in the Query data model has a type annotation that identifies the type of its content. For example, a `salary` element might have the type annotation `xs:decimal`, indicating that it contains a decimal number. A type annotation identifies either a built-in Schema type, such as `xs:decimal`, or a type that is defined in some imported schema. For example, a schema might define the type `hatsize` to be derived from `xs:integer` by restricting its values to a certain range. Element and attribute nodes acquire their type annotation through a process called *schema validation*, which involves comparing the content of the node to the schema declaration for nodes with that name. For example, the element named `hat` might be declared in a schema to have the type `hatsize`. If the content of a given element node with the name `hat` conforms to the definition of the type `hatsize`, the element node receives the type annotation `hatsize` during schema validation; otherwise, schema validation fails, and an error is reported. Schema validation may be performed on input documents before query processing, and on newly constructed elements and attributes during query processing. Validation of nodes against user-provided schemas is an optional feature of an XQuery implementation. In an implementation that does not support this feature, all element nodes have the generic type annotation `xs:anyType`.

Several places in the XQuery syntax call for a reference to a type. One of these is a function definition, which specifies the types of the function parameters and result. Other XQuery expressions that require type references include `instance-of`, `typeswitch`, and `cast` expressions. In these expressions, it is not always sufficient to refer to a type simply by its name. For example, when declaring the type of a function parameter, the user may wish to specify an element with a particular name, or with a particular type annotation, or both. Alternatively, the type of a function parameter might be an unrestricted element node, or some other kind of node such as an attribute or text node, or an unrestricted atomic value. Furthermore, since all XQuery values are sequences, we may want a type

reference to indicate a permissible number of occurrences, such as exactly one, zero or one, zero or more, or one or more.

Specifying the syntax of a type reference in XQuery was a difficult and important part of the language design. Although XML Schema itself has all the necessary features, embedding fragments of XML Schema language in a query was not considered appropriate, because Schema syntax is quite verbose and is inconsistent in style with XQuery. The working group considered inventing a new type-definition syntax with power comparable to that of XML Schema, but rejected this approach because it would have added a great deal of complexity to XQuery and would have duplicated the features of an existing W3C recommendation.

In the end, the working group defined a type reference syntax containing a few keywords and symbols to give users a limited degree of flexibility in defining types. The main features of this type reference syntax are listed below:

- An atomic type is referred to simply by its *QName*, such as `xs:integer` or `po:price`. The generic name `xdt:anyAtomicType` refers to a value of any atomic type.
- Element nodes can be referred to in any of the following ways:
  - `element(`*N*`, `*T*`)` denotes an element with name *N* and type annotation *T*. For example, `element(shipto, address)` denotes an element named `shipto` whose type annotation is `address`.
  - `element(`*N*`, *)` denotes an element with name *N*, regardless of its type annotation.
  - `element(*, `*T*`)` denotes an element with type annotation *T*, regardless of its name.
  - `element(`*N*`)` denotes an element whose name is *N* and whose type annotation is the (possibly anonymous) type declared in the imported schema for elements named *N*.
  - `element(`*P*`)`, where *P* is a valid schema "path" beginning with the name of a global element or type and ending with one of its component elements, denotes an element whose name and type annotation conform to that schema path. For example, `element (order/shipto/zipcode)` denotes an element named `zipcode`

that has been validated against the type definition in the schema context `order/shipto`.

- `element()` denotes any element, regardless of its name or type annotation.

- Attribute nodes can be referred to in a similar way to element nodes, except that attribute names are prefixed by an `@` sign. For example, `attribute(@`*N*`,` *T*`)` denotes an attribute node with name *N* and type annotation *T*, and `attribute()` denotes any attribute node, regardless of its name or type annotation.
- Other kinds of nodes can be referred to by keywords that indicate the kind of node, followed by empty parentheses. For example, `text()`, `document()`, `comment()`, and `processing-instruction()` refer to any node of the named kind; `node()` refers to a node of any kind; and `item()` refers to either a node or an atomic value.
- Any type reference may be followed by one of the following *occurrence indicators:* `*` denotes a sequence of zero or more occurrences of the given type; `+` represents one or more occurrences; and `?` represents zero or one occurrence. The absence of an occurrence indicator denotes exactly one occurrence of the given type. For example, `xs:integer?` denotes an optional integer value, and `element(hatsize)*` denotes a sequence of zero or more elements named `hatsize`.

Occurrence indicators are often useful in defining the parameter types of functions. For example, consider a function that classifies jobs into categories such as "Professional," "Clerical," and "Sales," based on criteria such as job title and salary. The definition of this function might be written as follows, indicating that the function accepts an element named `job` and returns a string:

```
define function job-category($j as element(job)) as xs:string
```

The `job-category` function defined above will raise an error if its argument is anything other than exactly one `job` element. This error might be encountered by the following query:

```
for $p in //person return $p/name, job-category($p/job)
```

For any `person` in the input stream that has no `job` or more than one `job`, this query will invoke the `job-category` function with an invalid argument. This error could be avoided by making the `job-category` function more flexible. The following definition uses the occurrence indicator `*` on the function parameter, allowing the function to accept a sequence of zero or more `job` elements:

```
define function job-category($j as element(job)*) as xs:string
```

Of course, the body of the `job-category` function must be written in such a way that it returns an appropriate result for any argument that matches the declared parameter type in the function definition.

If a function parameter is specified as an element node with a particular name, such as `element(shipto)`, the actual argument passed to the function can be an element that is in the substitution group of the named element. Similarly, if a function parameter is specified as an element node with a particular type, such as `element(*, address)`, the actual argument passed to the function can be an element whose type is derived (by restriction or by extension) from the named type.

Type references in XQuery are designed to be compatible extensions of the node tests that are defined in XPath Version 1. For example, in XPath Version 1, `text()` and `comment()` are valid node tests that can be used in path expressions to select text nodes or comment nodes. The following path expression, valid in XPath Version 1, finds all text nodes that are descendants of a particular `contract` element node:

```
/contract[id="123"]//text()
```

In XQuery, the following path expression uses an extended node test to find all element nodes of type `address` that are descendants of a particular `contract` element node. Note that the type reference syntax in XQuery is an extension of the node test syntax of XPath Version 1, and that XQuery allows a type reference to serve as a node test in a path expression.

```
/contract[id="123"]//element(*, address)
```

Issue 4: Element Constructors

One of the strengths of XQuery is its ability to construct new XML objects such as elements and attributes. The syntax used to do this has evolved considerably during the design of the language.

In the original Quilt proposal, a constructed element was denoted simply by an XML start tag and end tag, enclosing an expression that represented the content of the element. The expression enclosed between the tags was evaluated in the same way as any other Quilt expression. This design was criticized because it resembled XML notation but was not interpreted in the same way as XML notation, as illustrated by the following examples:

- `<greeting>Hello</greeting>`. In XML this is an element containing character content. But in Quilt, `Hello` would be interpreted as a path expression and evaluated, unless it were denoted as a string by enclosing it in quotes as follows:
  `<greeting>"Hello"</greeting>`
- `<animal>A <color>black</color> cat</animal>`. In XML this is an element with mixed content. But in Quilt it would be a syntax error because the sequence of values inside the `animal` element is not separated by commas. The Quilt equivalent of the given XML expression would be as follows:
  `<animal>"A ", <color>"black"</color>, " cat"</animal>.`

The working group decided that XQuery should provide a form of element constructor that conforms more closely to XML notation. However, in the content of an element, it is necessary to distinguish literal text from an expression to be evaluated. Therefore, in an XQuery element constructor, an expression to be evaluated must be enclosed in curly braces. For example:

- `<price>123.45</price>` is a constructed element named `price`, containing a literal value.
- `<price>{$retail * 0.85}</price>` is a constructed element whose content is computed by evaluating an expression.

The syntax described above is not identical to XML notation because it uses the left-curly-brace character `{` to denote the beginning of an evaluated expression, whereas in XML `{` is an ordinary character. If a left-curly-brace is intended to be interpreted as an ordinary character in the content of a constructed element, XQuery requires it to be doubled (`{{`).

In addition to an XML-like syntax for constructing elements, XQuery provides an alternative syntax called a *computed constructor* that can be used to construct elements with computed names as well as other kinds of nodes. The use of computed constructors is illustrated by the following examples:

- `element {$name} {$content}` constructs an element node with a given name and content (of course, `$name` and `$content` can be replaced by any valid XQuery expression).
- `attribute {$name} {$content}` constructs an attribute node with a given name and content.
- `document {$content}` constructs a document node with a given content.
- `text {$content}` constructs a text node with a given content.

The most complex and difficult issue in the design of XQuery constructors turned out to be determining the type of a constructed element. Since every node in the Query data model has a type annotation, it is necessary to attach a type annotation to the node produced by an element constructor. Since the type annotation of an element node indicates the type of the element's content, the first approach investigated by the working group was to derive the type annotation of the constructed node from the type of its content. For example, the following element constructor contains an integer-valued expression, so it seems natural to annotate the resulting element node with the type `xs:integer`:

```
<a>{8}</a>
```

The problem with this approach is that, by definition, elements are assigned their type annotations by the process of schema validation. The type assigned by schema validation may not be the same as the type of the

data from which the element was constructed. For example, if the element in the above example were validated against a schema, it might receive an annotation that is more specific than `xs:integer`, such as `hatsize`.

In some cases, the type assigned to an element by schema validation is guaranteed to be different from the original type of its content. As an example, consider the following element constructor:

```
<a>{1, "2"}</a>
```

This element constructor contains an expression whose value is a sequence of an integer followed by a string. However, XML provides no way to represent this element in such a way that, after validation, its content is typed as an integer followed by a string. The XML representation of this element is as follows:

```
<a>1 2</a>
```

Depending on its definition in the relevant schema, this XML element might be validated as containing one string, or two integers, or possibly a more specific user-defined type such as two hatsizes. But there is no possible schema definition that will cause this element to be validated as containing an integer followed by a string.

The designers of XQuery were confronted with the fact that a type annotation based on the content of a constructed element is never guaranteed to be correct after validation of the element, and in some cases is guaranteed to be incorrect. The working group tried hard to find a reasonable rule by which the type of a constructed element could be derived from its content in "safe and obvious" cases. After much study and debate, the group found it impossible to define such a rule that satisfied everyone. As a result, they decided that each constructed element in XQuery should be automatically schema-validated as part of the construction process.

In order to perform schema validation on a constructed element, a *validation mode* must be specified. Validation mode can have one of the following three values:

- `strict` requires that the given element name must be declared in an imported schema and that the content of the element must comply with its schema declaration.

- `skip` indicates that no validation is attempted for the given element. In this mode, constructed elements are given the type annotation `xs:anyType`, and constructed attributes are given the type annotation `xs:anySimpleType`.
- `lax` behaves like `strict` if the element name is declared in an imported schema, and like `skip` otherwise.

XQuery allows a query writer to specify the validation mode for a whole query or for any expression nested within a query. By specifying `lax` validation, a user can allow the construction of *ad hoc* elements that are not declared in any schema, and by specifying `skip` validation, a user can avoid the performance overhead of validating constructed elements entirely.

In addition to a validation mode, the process of schema validation for an element requires a *validation context*. Validation context controls how the name of the given element is matched against declarations in the imported schemas. If the context is `global`, the element name is matched against global (topmost) schema declarations. Alternatively, the context may specify a path, beginning with a global element or type declaration, within which the given element name is to be interpreted. For example, the element `zipcode` might not match a global schema declaration, but might match an element declaration within the context `PurchaseOrder/customer/shipto`.

The outermost element constructor in a query is validated in the global context unless the user specifies a different context. Each element constructor then adds the name of its element to the validation context for nested elements. For example, if the outermost constructed element in a query is named `PurchaseOrder`, then the immediate children of this element have a validation context of `PurchaseOrder`.

Issue 5: Static Typing

As noted earlier, one of the basic principles of XQuery design is that the processing of a query can include a static analysis phase in which some error checking and optimization can be performed. By definition, static analysis is performed on the query only and is independent of input data. Some kinds of static analysis are clearly essential, such as parsing and

finding syntax errors. In addition, the working group saw some value in taking advantage of schema information to infer the types of query expressions at query analysis time. This kind of type inference is called *static typing*. Chapter 4, "Static Typing in XQuery," looks at this subject in some depth.

Static typing offers the following advantages: (1) It can guarantee in some cases that a query, if given valid input data, will generate a result that conforms to a given output schema; (2) It can be helpful in early detection of certain kinds of errors such as calling a function with the wrong type of parameter; (3) It can produce information that may be helpful in optimizing the execution of a query—for example, it may be possible to prove by static analysis that the result of some expression is an empty sequence.

The *static type* of an expression is defined as the most specific type that can be deduced for that expression by examining the query only, in the absence of input data. The *dynamic type* of a value is the most specific type assigned to that value as the query is executed. Note that static type is a compile-time property of an expression, whereas dynamic type is a run-time property of a value.

Static typing in XQuery is based on a set of *inference rules* that are used to infer the static type of each expression, based on the static types of its operands. Some kinds of expressions have requirements for the types of their operands (for example, arithmetic operators are defined for numeric operands but not for strings). Static analysis starts with the static types of the "leaves" of the expression tree (simple constants and input data whose type can be inferred from the schema of the input document). It then uses inference rules to infer the static types of more complex expressions in the query, including the query itself. If, during this process, it is discovered that the static type of some expression is not appropriate for the context where it is used, a type error is raised.

Type-inference rules are written in such a way that any value that can be returned by an expression is guaranteed to conform to the static type inferred for the expression. This property of a type system is called **type soundness.** A consequence of this property is that a query that raises no type errors during static analysis will also raise no type errors during execution on valid input data. The importance of type soundness depends somewhat on which errors are classified as "type errors," as we will see below.

$$\frac{\begin{array}{c} statEnv \vdash Expr_0 : Type_0 \\ statEnv + varType(Var : Type_0) \vdash Expr_1 : Type_1 \end{array}}{statEnv \vdash \texttt{let \$}Var := Expr_0 \texttt{ return } Expr_1 : Type_1} \quad \text{(LET-STATIC)}$$

Figure 2.4 A Typical Type-Inference Rule

Figure 2.4 illustrates a type-inference rule used in XQuery.

Some inference rules use a notation (`Type1 | Type2`) to denote a type that includes all the instances of type `Type1` as well as all the instances of type `Type2`. This inferred type has no name. It is an example of the fact that, in many cases, the inferred static type of an expression can be given only a structural description. *XQuery 1.0 Formal Semantics* [XQ-FS] uses its own notation for the structural description of types, using operators such as `|` (or), `&` (and), "`,`" (sequence), and `?` (optional).

XML Schema also includes a notation for structural description of types. The working group decided not to use the notation of XML Schema in the type-inference rules in the XQuery formal semantic specification because XML Schema notation is very verbose, and because not all types inferred by the XQuery type inference rules are expressible in XML Schema notation. For example, the rule named "Schema Component Constraint: Element Declarations Consistent" [SCHEMA] prevents an XML Schema type from containing two subelements with the same name but different types. Similarly, the rule named "Schema Component Constraint: Unique Particle Attribution" prevents the definition of an XML Schema type that cannot be validated without lookahead. The XQuery type system does not have these constraints. As a result, unlike XML Schema, the XQuery type system has the **closure** property (any type inferred by the XQuery type inference rules is a valid type).

The structural type notation used in [XQ-FS] is used only in the formal semantic specification and is not part of the syntax of XQuery as seen by users. This decision was made to avoid adding complexity to the query language, and because the working group did not wish to introduce a user-level syntax that duplicated the facilities of XML Schema.

The working group spent considerable time on the issue of whether static typing should be based on validated type annotations or on structural

comparison. As an illustration of this issue, consider a function named `surgery` that expects as one of its parameters an element of type `surgeon`. Suppose the type `surgeon` is defined to consist of a `name` and a `schedule`. Is it an error if the `surgery` function is invoked on an element that contains a `name` and a `schedule` but has not been schema-validated as a `surgeon`? The working group decided that this should be a type error. In addition to assigning a specific type annotation to an element, schema validation may perform processing steps that are not otherwise supported in the XQuery type system, such as checking pattern facets and providing default attribute values.

Another question closely related to the preceding one is this: Is it a type error if the `surgery` function is invoked on an element that has been schema-validated as having the type `brainSurgeon`? In XQuery, this function call is valid if the type `brainSurgeon` is declared in an imported schema to be a subtype of (derived from) the type `surgeon`. In other words, **static type-checking** in XQuery is based on validated type names and on the type hierarchy declared in imported schemas.

Although the benefits of static typing are well understood in the world of programming languages, extending this technology to the very complex world of XML Schema proved to be a challenging task. Many of the features of XML Schema, such as ordered sequences, **union types,** substitution groups, minimum and maximum cardinalities, pattern facets, and two different kinds of inheritance, were difficult to assimilate into a system of type-inference rules. Even after some simplifying approximations (such as limiting cardinalities to zero, one, or multiple), the resulting set of rules is quite complex.

XQuery is designed to be used in a variety of environments. Static typing of a query is of greatest benefit when the expected type of the query result is known in advance, the input documents and expected output are both described by schemas, and the query is to be executed repeatedly. This represents one usage scenario for XQuery, but not the only one. Some queries may be exploratory in nature, searching loosely structured data sources such as web pages for information of unknown types. Many XML documents are described by a DTD rather than by a schema, and many have neither a DTD nor a schema. Some XML documents may be described by structural notations other than XML Schema, such as

RELAX-NG [RELAXNG]. In some cases, query inputs may be synthesized directly from information sources such as relational databases rather than from serialized XML documents.

To make XQuery adaptable for many usage environments, the language was organized as a required subset called Basic XQuery and two optional features called Schema Import and Static Typing. This approach keeps Basic XQuery relatively simple and lowers the implementation cost of the language in environments where the optional features are of limited benefit.

Basic XQuery includes all the kinds of expressions in the full language. However, a Basic XQuery implementation is not required to deal with user-defined types and is not required to raise static type errors. In Basic XQuery, the only types that are recognized are the built-in atomic types of XML Schema, such as `xs:string`, `xs:integer`, and `xs:date`; the derived duration types `xdt:yearMonthDuration` and `xdt:dayTimeDuration`; and predefined generic types such as `xdt:untypedAtomic` and `xs:anyType`. Basic XQuery might be used, for example, to query data that is exported in XML format from a relational database whose datatypes can be mapped into built-in XML types. When a Basic XQuery implementation is used to query a document that has been validated against an XML schema, the type annotations in the PSVI are mapped into their nearest built-in supertypes (complex types are mapped into `xs:anyType`, and derived atomic types such as `hatsize` are mapped into their built-in base types such as `xs:integer`).

The first optional feature, Schema Import, provides a way for a query to import a schema and to take advantage of type definitions in the imported schema. For example, a query might import a schema containing the definition of the user-defined type `hatsize`, derived from `xs:integer`. This query could then define a function that accepts a parameter of type `hatsize`. Under the Schema Import feature, an attempt to call this function with an argument that has not been schema-validated as a `hatsize` is an error.

The second optional feature, Static Typing, defines a specific set of type errors and requires conforming implementations to detect and report these errors during query analysis. An XQuery implementation that does

not claim conformance to this feature can still do static query analysis in its own way, for optimization or for error detection. Also, an implementation that does not claim conformance to Static Typing is still responsible for detecting and reporting dynamic type errors. Any condition that will necessarily lead to a run-time error can be reported as a static error at the discretion of the implementation.

It is important to understand that an expression that raises a static type error may nevertheless execute successfully on a given input document. This is because the rules for static type inference are conservative and require a static type error to be raised whenever it cannot be proven that no possibility of a type error exists. For example, consider the following function definition:

```
define function pay($e as element(employee)) as xs:decimal?
   { $e/salary + $e/bonus }
```

A user may want to execute this query against an XML document containing records from a department in which every `employee` has at most one `bonus`. However, the only schema available may be a company-wide schema in which the declaration of the `employee` element allows multiple (zero or more) `bonus` subelements. In this case, a system that conforms to the Static Typing feature would be required to raise a static error, although execution of the query on the given input data would have been successful.

Conversely, a query that passes static analysis without error may still raise an error at run-time. For example, consider the following expression, where the variable `$c` is bound to a `customer` element:

```
$c/balance + $c/address
```

If the `customer` element has no schema declaration, the typed values of the `balance` and `address` subelements of a given `customer` will be of type `xdt:untypedAtomic`. Addition of two values of type `xdt:untypedAtomic` does not raise a static type error—instead, at execution time the values are converted to the type `xs:double`. If, at run-time, the value of `$c/balance` or `$c/address` cannot be converted to `xs:double`, a dynamic error is raised. In order to preserve the claim of "type soundness," this kind of error is not classified as a type error.

Before leaving the subject of static typing, it is worth discussing the influence that static typing has had on the syntax of the XQuery language. Two kinds of XQuery expressions are present in the language solely in order to support better static type inference. The first kind is the `typeswitch` expression, which is illustrated by the example below. The expression in this example branches on the type of a variable named `$employee`, computing total pay in one of three ways depending on whether the employee is a salesperson, a manager, or a generic employee.

```
typeswitch ($employee)
  case $s as element(*, salesperson)
     return $s/salary + $s/commission
  case $m as element(*, manager)
     return $m/salary + $m/bonus
  default $d
     return $d/salary
```

The dynamic behavior of a `typeswitch` expression can always be obtained by using two other kinds of expression called an `if-then-else` expression and an `instance-of` expression. For example, the `typeswitch` expression in the preceding example might have been expressed as follows:

```
if ($employee instance of element(*, salesperson))
then $employee/salary + $employee/commission
else if ($employee instance of element(*, manager))
then $employee/salary + $employee/bonus
else $employee/salary
```

The dynamic (run-time) behavior of the two expressions above is identical. However, the first example will pass static type-checking because it can be proven statically that the variable `$s` can only be bound to an element of type `salesperson`, and therefore `$s/commission` will always return a valid result. The second example, on the other hand, is considered to be a static type error because static analysis cannot predict which branch of the conditional expression will be executed, nor can it prove that the subexpression `$employee/commission` will not be evaluated for some `employee` element that is not of type `salesperson`.

The second type of expression that was added to XQuery solely in support of static typing is the `treat` expression. To understand the use of `treat`, consider the following expression, in which the variable `$a` is bound to an element whose static type is `Address`:

```
$a/zipcode
```

Suppose that only an element of type `USAddress`, a subtype of `Address`, is guaranteed to have a `zipcode`. In this case, the above expression is a static error, since static analysis cannot guarantee that the element bound to `$a` has the type `USAddress`. To avoid this static error, the Static Typing feature requires this expression to be written as follows:

```
($a treat as element(*, USAddress))/zipcode
```

At query analysis time, the static type of the `treat` expression is `element(*, USAddress)`, which enables the `zipcode` to be extracted without a static type error. The `treat` expression serves as a "promise" that the variable `$a` will be bound to an element of type `USAddress` at execution time. At execution time, if `$a` is not in fact bound to an element of type `USAddress`, a dynamic error is raised.

As in the case of `typeswitch`, the dynamic behavior of `treat` can be simulated by using a combination of an `if-then-else` expression and an `instance-of` expression. For example, the dynamic behavior of the preceding expression can be simulated as follows:

```
if ($a instance of element(*, USAddress))
then $a/zipcode
else error("Not a USAddress")
```

Although this expression simulates the dynamic behavior of the preceding `treat` expression, it is nevertheless considered to be a static error because the compiler cannot predict which branch of the `if-then-else` expression will be executed. (Chapter 4 provides more details about static type-checking in XQuery.)

Issue 6: Function Resolution

The *signature* of a function is defined by its expected number of parameters and by the respective types of those parameters. Many modern query and programming languages support **function overloading,** which permits multiple functions to be defined with the same name but with different signatures. The process of choosing the best function for a given function call based on static analysis of the argument types is called **function resolution.** Some languages also support *polymorphism*, a form of function overloading in which static analysis of a function call does not

necessarily choose a single function—instead, it can result in a list of applicable functions from which the most specific function is selected at execution time on the basis of the dynamic types of the actual arguments.

The designers of XQuery made an early decision not to support function overloading. The reasons for this decision were as follows:

- Despite the advent of schemas, much XML data is untyped. For the convenience of the user, XQuery automatically coerces untyped data to a specific type based on the context where it is used. In order for this feature to work, each function-name must correspond to a single, well-defined signature. For example, consider the untyped element `<a>47</a>`. If this element is passed to a function that expects an element, it retains its character as an element; if it is passed to a function that expects a decimal, the value `47` is extracted and treated as a decimal value; if it is passed to a function that expects a string, the value `47` is extracted and treated as a string. This flexibility in treatment of untyped data would not be possible if three functions could have the same name while one takes an element parameter, one takes a decimal parameter, and one takes a string parameter.
- The type system of XQuery is based on XML Schema and is quite complex. It has two different kinds of type inheritance: derivation by restriction and derivation by extension. In addition, it has the concept of substitution groups, which allow one kind of element to be substituted for another on the basis of its name. The rules for casting one type into another and for promotion of numeric arguments are also complex. They are made even more complex by XPath Version 1 compatibility rules, which come into play when XPath Version 2 (a subset of XQuery) is used in "XPath Version 1 Compatibility Mode" (as it is when embedded in XSLT). The designers of XQuery were not eager to add another increment of complexity on top of the rules they inherited from XML Schema and XPath. In the end, it was decided that the added complexity of function overloading outweighed its benefit, at least for the first version of XQuery.

Despite the decision not to support function overloading, it was necessary for XQuery to support the XPath Version 1 core function library,

which already contained some overloaded functions. Support for these functions was reconciled with the XQuery design in the following ways:

- Some XPath Version 1 functions take an optional parameter. For example, the `name` function returns the name of a node, but if called with no argument, it returns the name of the context node. In XQuery, this kind of function is treated as two functions with the same name but with different *arity* (number of parameters). The built-in function library of XQuery contains several sets of functions that differ only in their arity. This form of overloading is permitted in the built-in function library but not among user-defined functions. It does not complicate the rules for function resolution, since a function is always uniquely identified by its name and its arity.
- Some XPath Version 1 functions can accept several different parameter types. For example, the `string` function of XPath Version 1 can turn almost any kind of argument into a string. In XQuery, functions like `string` have a generic signature that encompasses all their possible parameter types. For example, the signature of the XQuery `string` function is `string($s as item?)`. The type `item?` denotes an optional occurrence of a node or atomic value. The description of the `string` function specifies how it operates on arguments of various types that satisfy its signature. The technique of creating a function with a generic signature is also available to users for creating user-defined functions. The body of such a function can branch on the dynamic type of the actual argument.

One painful consequence of the decision not to support function overloading was the loss of polymorphism. Polymorphism is convenient because it facilitates introducing new subtypes of existing types that inherit the behavior of their supertypes where appropriate and specialize it where necessary. A polymorphic language allows a query to iterate over a heterogeneous sequence of items, dispatching the appropriate function for each item based on its dynamic type. For example, in a polymorphic language, a generic function named `postage` might be defined for elements of type `Address`, with specialized `postage` functions for elements of type `USAddress` and `UKAddress` that are derived from `Address`.

Although XQuery does not support true polymorphism, the designers wished to make it possible and reasonably convenient for users to simulate a form of polymorphism. This was accomplished by the following rule: Any function can be invoked with an argument whose dynamic type is a subtype of the expected parameter type. When this occurs, the argument retains its original dynamic type within the body of the function.

This rule makes it possible for a user to write functions such as the `postage` function shown in Listing 2.3, which invokes one of several more specialized functions depending on the dynamic type of its argument:

Listing 2.3 Function Simulating a Form of Polymorphism

```
define function postage($a as element(*, Address)) as xs:decimal
{
typeswitch($a)
   case $usa as element(*, USAddress)
      return us-postage($usa)
   case $uka as element(*, UKAddress)
      return uk-postage($uka)
   default
      return error("Unknown address type")
}
```

The `postage` function can be used in a query that iterates over a set of addresses of mixed type. However, if a new type named `GermanAddress` is introduced that is also derived from the `Address` type, it is necessary to add a new case to the body of the `postage` function, since XQuery will not automatically dispatch a specialized function for the new type.

It is worth considering whether function overloading could be introduced in a later version of XQuery. The main barrier to the introduction of this feature would be the implicit coercions of function parameters that are defined in XQuery. As noted above, untyped data can be coerced into either a string or a number, depending on context. If it were possible to define two functions with the same name that respectively take a string parameter and a numeric parameter, it would be necessary to define a priority order among the possible coercions, in order to resolve an ambiguous function call. This could certainly be done. If function overloading is introduced in a future version of XQuery, users will need to exercise care in the evolution of their function libraries to ensure that existing queries continue to run successfully.

Issue 7: Error Handling

In general, XQuery expressions return values and have no side effects. However, some special consideration is needed for error cases. For example, what should be the value of an expression that contains a division by zero?

One approach that the working group considered was to define a special *error value* in the Query data model, to be returned by all expressions that encounter an error. The error value would have its own *error type* that is different from all other types. In this approach, for example, the static type of a double-precision arithmetic expression would be `(double | error)` rather than simply `double`. After some study, the group decided that including a special error value (and error type) in the Query data model complicated the semantics of the language and made it difficult to distinguish one kind of error from another.

The approach finally adopted for handling dynamic errors in XQuery is to allow an expression either to return a value or to raise a dynamic error. An expression that raises an error is not considered to return a value in the normal sense. Therefore dynamic errors are handled outside the scope of the type system. An error carries with it an ordinary value (of any type) that describes the error. A function named `error` is provided that raises an error and attaches the argument of the function to the error as a descriptive value.

Propagation of dynamic errors through XQuery expressions is handled by a general rule and some special rules that apply to specific kinds of expressions. The general rule is that if the operands of an expression raise one or more dynamic errors, the expression itself must also raise a dynamic error. Which of several operand errors is propagated by an expression is not specified. If one operand of an expression raises an error, the expression is permitted, but is not required, to evaluate its other operands. For example, in evaluating the expression `"Hello" + (length * width * height)`, an XQuery implementation can raise an error as soon as it discovers that `"Hello"` is not a number.

The special error-handling rules for certain kinds of expressions are intended to make these expressions easier to optimize and more efficient to implement. These rules have the effect of making the result of an

XQuery expression non-deterministic in a certain limited sense. In some cases, an expression may either return a result or raise an error. However, if an expression returns a result, the result is always well defined by the semantics of the language. The circumstances in which non-deterministic handling of errors is permitted are as follows:

- If one operand of `and` is `false` and the other operand raises an error, the `and` operator may either return `false` or raise an error.
- If one operand of `or` is `true` and the other operand raises an error, the `or` operator may either return `true` or raise an error.
- The result of a quantified expression may depend on the order of evaluation of its operands, which may vary from one implementation to another. As an example, consider the expression `some` *variable* `in` *range-expr* `satisfies` *test-expr*. If *range-expr* contains a value for which *test-expr* is `true` and another value for which *test-expr* raises an error, the quantified expression may either return `true` or raise an error. Similarly, in the universally quantified expression `every` *variable* `in` *range-expr* `satisfies` *test-expr*, if *range-expr* contains a value for which *test-expr* is `false` and another value for which *test-expr* raises an error, the quantified expression may either return `false` or raise an error.
- Since some comparison operators, such as `=`, are defined to include an implicit existential quantifier, these operators are made non-deterministic by the above rules. For example, the expression `(47, 3 div 0) = 47`, which compares a sequence of two values to a single value, might either return `true` or raise an error.
- The Effective Boolean Value of an expression is defined to be `true` if the expression returns a sequence containing more than one item. But if an error is raised during evaluation of the items in such a sequence, the Effective Boolean Value computation may either return `true` or raise an error.
- If a function can evaluate its body expression without evaluating one of its arguments, the function is allowed, but not required, to evaluate that argument. If such an argument contains an error, the function may either raise an error or return the value of its body expression.

An ideal language would be both deterministic and efficient in execution. As illustrated by some of the preceding examples, the presence of

sequences in the Query data model leads to some tension between the goals of determinism and efficiency in XQuery. In general, this tension has been resolved in favor of efficiency for queries that contain errors; however, a query that contains no errors must always return a deterministic result.

Users can protect themselves against some kinds of errors by using an `if-then-else` expression, which evaluates one of two subexpressions (called "branches" here) depending on the Boolean value of a test expression. The `if-then-else` expression is defined in such a way that it evaluates only the branch that is selected by the test expression; therefore errors in the other branch are ignored. By using an `if-then-else` expression, a query can control the value that is returned when an error condition is encountered. For example, a user could guard against division by zero by the following expression, which returns an empty sequence rather than raising an error if the divisor is zero:

```
if ($divisor ne 0) then $dividend div $divisor else ( )
```

The working group discussed a proposal for a "try/catch" mechanism similar to that used in the Java language, which would have allowed expressions at various levels of the query hierarchy to "catch" errors raised by nested expressions. An expression might specify "handlers" for various kinds of errors encountered in its operands, and in some cases might be able to recover from these errors rather than propagating them. In the end, this approach was considered to be too complex for the first version of XQuery. The working group may revisit this proposal in a future version of the language.

Issue 8: Ordering Operators

Since all values in XQuery are ordered sequences, operators for controlling the order of a sequence are of considerable importance to the language. All of the operators of XQuery return their results in a well-defined order. For example, path expressions and the `union` and `intersect` operators always return sequences of nodes in document order. Of course, the ordering of the sequences returned by various expressions in a query is not always essential to the meaning of the query. The cost of evaluating a sequence may depend strongly on the order in which it is evaluated, due to

the organization of physical storage or the availability of access aids such as indexes. If the order of some result is not important, the query writer should have some way to say so, thus giving the implementation the flexibility to generate the result in the fastest or least expensive way.

For a time, the working group experimented with "unordered" versions of various operators. In the end, it was decided that the XQuery function library should include a general-purpose function named `unordered`. The `unordered` function can be applied to any sequence, and it returns the same sequence in a non-deterministic order. In effect, the `unordered` function signals an optimizing compiler that its argument expression can be materialized in any order that the optimizer finds convenient. Obviously, applying the `unordered` function to an expression *E* can have implications for the ordering properties of expressions that are nested inside *E* or that use *E* as an operand, and XQuery implementations are free to take advantage of these implications.

XQuery also needs to provide a way for users to specify a particular order for the items in a sequence. In fact, the result of a query may be a hierarchy of elements in which an ordering needs to be specified at several levels of the hierarchy. The working group explored two alternative approaches to this problem.

The first approach was to provide a general-purpose operator called `sort by` that could be applied to any expression to specify an order for the result of the expression. This might be called the "independent sorting" approach because `sort by` is an operator that can be applied to any sequence, and is independent of any other operator. In this approach, the meaning of `E1 sort by E2` is defined as follows: For each item in the result of expression `E1`, the expression `E2` is evaluated with that item as the context item. The resulting values are called **sort keys.** The items returned by `E1` are then arranged in an order that is controlled by their respective sort keys. Options are provided that allow the user to specify primary and secondary sort keys, ascending or descending order, placement of items whose sort key is empty, and other details.

The independent sorting approach is illustrated by the following example, in which a query searches its input for employees in a particular department, and returns them in alphabetical order by their

names. In this example, the `sort by` operator is applied to a path expression.

```
//employee[dept = "K55"] sort by name
```

The independent sorting approach has two important disadvantages. First, it can only specify an ordering for a sequence based on sort keys that are actually present in the sequence. Suppose, for example, that a query is intended to return the names of all employees in a particular department, in the order of their salaries, but not to return the actual salaries. This query, while not impossible under the independent sorting approach, is not as straightforward as the previous example. It might be expressed as follows:

```
for $e in //employee[dept = "K55"] sort by salary
return $e/name
```

The difficulty of expressing an ordering by the independent sorting approach becomes greater if the sort key is computed from two or more bound variables but is not included in the query result. As an example of this problem, a query might need to join `employee` elements with their respective `department` elements, and sort the result according to the ratio between the employee's salary and the department's budget, but not include this ratio in the final output. Such a query is difficult (but not impossible) to express with an independent `sort by` operator.

The second disadvantage of the independent sorting approach is that it separates iteration and ordering into two separate syntactic constructs. The result of an iteration might be processed in various ways, combined with other results, given a new name, and finally sorted. The most efficient way to process such a query might be to perform the initial iteration in the order required for the final result. However, the specification of this ordering may be syntactically distant from the iteration, and it may not be expressed in a way that is easy for an implementation to use in optimizing the iteration.

Another ordering approach that the working group investigated and ultimately adopted consists of combining the ordering expression with the expression that controls iteration. The iteration expression in XQuery was originally called a "FLWR" expression because of its keywords `for`, `let`, `where`, and `return`. An `order by` clause was added to this expression

between the `where` and `return` clauses, changing the name of the expression to "FLWOR." Positioning the `order by` clause inside the FLWOR expression gives it access to the bound variables that control the iteration, even though the values bound to these variables may not ultimately be returned. Associating the ordering directly with the iteration also makes it easier for a system to implement the iteration efficiently. The disadvantage of this approach is that an ordering must always be expressed in the form of a FLWOR expression, even though it would not otherwise require an explicit iteration.

The advantages and disadvantages of the "ordered FLWOR" approach can be illustrated by repeating the two preceding example queries. The query that returns all the employees in a given department, sorted by name, can be expressed as follows (note that it now requires an explicit iteration, which was not needed in the "independent sorting" approach):

```
for $e in //employee[dept = "K55"]
order by $e/name
return $e
```

Similarly, the query that returns names of employees in a given department, ordered by their salaries but without returning the salaries, can be expressed as follows:

```
for $e in //employee[dept = "K55"]
order by $e/salary
return $e/name
```

The choice between the "independent sorting" approach and the "ordered FLWOR" approach is not obvious. The working group considered including both approaches, but decided not to do this on the grounds that the two alternatives are redundant and potentially confusing. In the end, the working group decided to include only the "ordered FLWOR" approach to control ordering of results in XQuery.

Conclusion

The design of XQuery has been a process of resolving the tensions between conflicting goals. It has been a slow process, and compromise has frequently been necessary.

The designers of XQuery did not begin with a blank slate. The Query data model was largely dictated by XML itself and by related W3C Recommendations such as XML Namespaces. The design of the language was also constrained by compatibility with existing standards such as XPath and XML Schema. Reconciling these constraints was not straightforward—for example, XPath Version 1 has only four types and a rather loose set of rules for type conversions, whereas XML Schema has forty-four built-in atomic types, a complex syntax for defining new types, and a strict set of rules for type validation.

As a general-purpose query language, XQuery needs to support a variety of usage modes. It needs to operate on untyped documents as well as on documents that are described by schemas and by DTDs. It needs to operate on individual XML files, on large repositories of pre-validated documents, on XML data synthesized from sources such as relational databases, and on streaming XML data. It needs to support applications in which all types are known and static typing guarantees are important, as well as exploratory queries in which the expected type of the result is not known in advance. It needs to support precise and reproducible queries as well as approximate searches, such as searches for synonyms of a given word. In order to span these modes of use while keeping the entry cost of an implementation as low as possible, the working group organized XQuery as a basic language and a set of optional features.

The Query working group did not conduct its work in isolation. All changes to XPath were discussed and approved jointly by the Query working group and the XSLT working group, since XPath is embedded in both the XQuery and XSLT languages. The Query working group consulted frequently with the Schema working group on issues such as date/time arithmetic, which required two new subtypes to be derived from the `duration` type. Internationalization aspects of XQuery were designed in cooperation with the Internationalization (I18N) working group. Suggestions on XQuery requirements and features were received from other working groups and individuals from time to time.

During the design of XQuery, the working group typically had a membership of between thirty and forty members, representing around twenty-five companies. These representatives formed a diverse group, with backgrounds ranging from relational database management to library science, and included the designers of several query languages.

Some were theoretical in their orientation, and others were more pragmatic. Some represented software vendors and some represented the user community. The world's largest software companies were represented, as were several new startup companies.

The working group conducted weekly conference calls and held a face-to-face meeting about every six weeks. Subgroups were formed to deal with specialized topics such as full-text search and the core function library. As is customary in the W3C, the working group operated by a process of consensus building, in which design alternatives were explored and decisions were made only after a consensus was reached (as determined by the chair). Design progress was documented in a series of working drafts that described the language in general, the data model, the function library, the formal semantics of the language, and a set of use cases. Each of these working drafts was republished on the working group web page about every three months. Throughout this process, public feedback was invited, and the members of the working group responded to public comments in online forums. At all times, the working group maintained a prototype parser based on the latest XQuery grammar.

At the time of this writing, the broad outline of XQuery Version 1 is reasonably stable, and the working group is engaged in editing the details of the language specification. Because of a strong desire to publish a stable recommendation as soon as possible, it is likely that several important features will not be included in XQuery Version 1. Examples include an update facility, text search features such as stem-matching and relevance ranking, a view definition facility, and bindings to specific host languages such as Java. These features are strong candidates for inclusion in a future version of XQuery.

The design process used by the Query working group is probably not the fastest way to design a query language, and languages designed by large committees are not often noted for elegance and simplicity. Nevertheless, the members of the working group hope that their diverse backgrounds and interests have contributed to making XQuery robust for a broad spectrum of usage environments. Time and experience will provide a measure of their success.