

XQuery: A Query Language for XML

Don Chamberlin

IBM Almaden Research Center

June 11, 2003

History

- 1998: W3C sponsors workshop on XML Query
- 1999: W3C charters XML Query working group
 - Chair: Paul Cotton
 - Currently 39 members, representing 25 companies
- 2000: WG publishes req'ts, use cases, data model
- 2001: WG publishes draft language spec's
- 2002: Working drafts updated periodically
- 2003: WG publishes full-text req'ts and use cases; XQuery Version 1 working drafts enter "last call"

Resources

- Public website: www.w3.org/XML/Query
 - Working drafts of language spec's
 - Links to reference implementations
 - Link to XQuery grammar test applet
- Member's website: www.w3.org/XML/Group/Query
 - Minutes, membership, internal documents
- Public comments:
 - Post to: public-qt-comments@w3.org
 - Archive: lists.w3.org/Archives/Public/public-qt-comments

Working Drafts

- Linked from www.w3.org/XML/Query:
 - XQuery 1.0: An XML Query Language
 - XML Path Language (XPath) 2.0
 - XQuery and XPath Data Model (*"LAST CALL"*)
 - XQuery and XPath Functions and Operators (*"LAST CALL"*)
 - XQuery Formal Semantics
 - XML Query Requirements
 - XML Query Use Cases
 - XSLT and XQuery Serialization
 - XML Syntax for XQuery (XQueryX)
 - XQuery and XPath Full-Text Requirements
 - XQuery and XPath Full-Text Use Cases

Does the world need a new query language?

The Web

Structured
Databases

- Most of the world's business data is stored in relational databases.
- The relational language SQL is mature and well-established.
- Can SQL be adapted to query XML data?
 - Leverage existing software
 - Leverage existing user skills
- How is XML data different from relational data?

Nesting

- Relational data is "flat"—rows and columns
- XML data is nested—and its depth may be irregular and unpredictable
- Relations can represent hierarchic data by foreign keys or by structured datatypes
- In XML it is natural to search for objects at unknown levels of the hierarchy: "Find all the red things."
- XPath is a compact and convenient notation for this type of query:

```
//*[@color = "Red"]
```

Metadata

- Relational data is uniform and repetitive
 - All bank accounts are similar in structure
 - Metadata can be factored out to a system catalog
- XML data is highly variable
 - Every web page is different
 - Each XML object needs to be self-describing
 - Metadata is distributed throughout the document
 - Queries may access metadata as well as data:
"Find elements whose name is the same as their content"

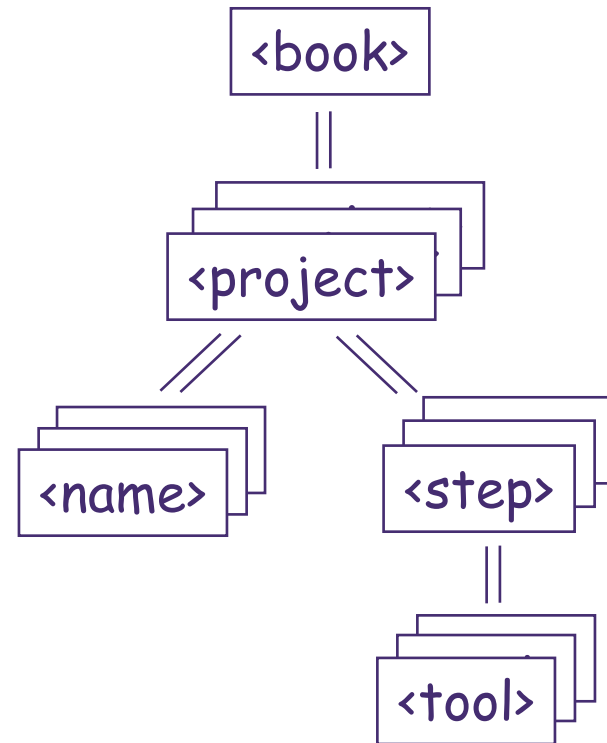
```
//*[name(.) = string(.)]
```

Heterogeneous Sequences

- Relational queries return uniform sets of rows
- The results of an XML query may have mixed types and complex structures.
 - "Red things": a flag, a cherry, a stopsign, . . .
 - Elements can be mixed with atomic values ("mixed data")
- XML queries need to be able to perform structural transformations
 - Example: invert a hierarchy

Ordering

- The rows of a relation are unordered
 - Any desired output ordering must be derived from values
- The elements in an XML document are ordered
- Implications for query:
 - Preserve input order in query results
 - Specify an output ordering at multiple levels
 - "Find the fifth step"
 - "Find all the tools used before the hammer"



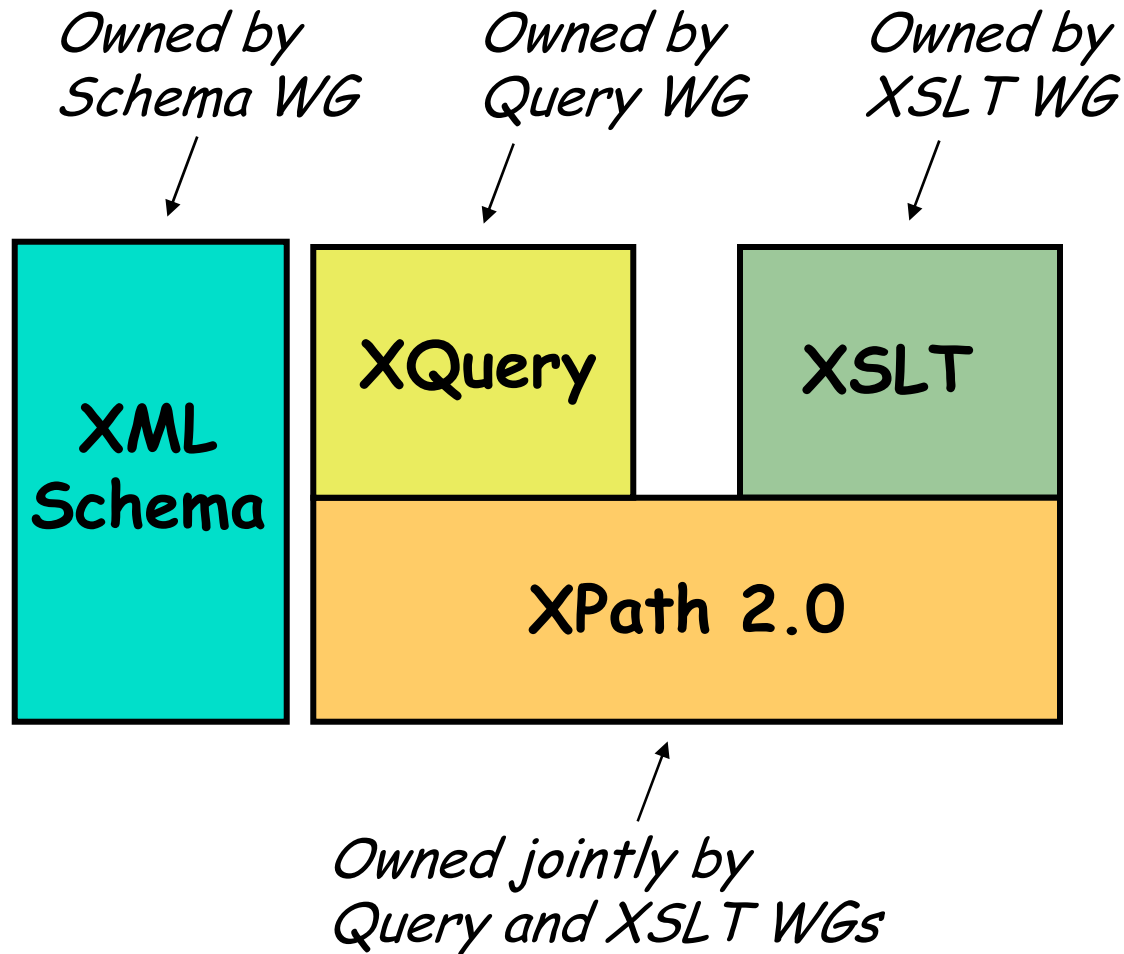
Missing Information

- Relational data is "dense"
 - Every row has a value in every column
 - A "null" value is needed for missing or inapplicable data
- XML data can be "sparse"
 - Missing or inapplicable elements can be "empty" or "not there"
 - This gives XML a degree of freedom not present in relational databases

The conclusion . . .

- XML is sufficiently different from relational data to justify its own query language.
- Designing an XML query language is going to be a complex task.

XQuery does not exist in a vacuum



Principles of XQuery Design

- Closure
 - Define a data model and a set of operators that are closed under the data model
- Compositionality
 - XQuery consists of several kinds of expressions
 - Every expression can be evaluated without side effects
 - Expressions can be composed with full generality
- Schema conformance
 - Use the type system of XML Schema
 - 44 built-in types, two kinds of inheritance, 12 "constraining facets", "substitution groups", etc.
 - "Schema validation" assigns types to elements

Principles of XQuery Design, cont'd.

- XPath compatibility
 - Adopt XPath as a syntactic subset
 - XPath and XQuery are mutually recursive
 - Evolve XPath 2.0, backward compatible with XPath 1.0
 - But XPath 1.0 has only four datatypes!
- Completeness
 - Roughly equivalent to "relational completeness"
 - No formal standard exists for hierarchic languages
 - Recursive Functions

Principles of XQuery Design, cont'd.

- Conciseness

`bonus > salary` vs.

`some b in bonus, s in salary
satisfies data(b) > data(s)`

- Simplicity

- (if possible for a language designed by committee)

- Static Analysis

- optional static analysis phase before query execution
- type inference rules based on XML Schema
- early detection of some kinds of errors
- optimization

Why does XQuery need a data model?

What does this mean?

```
/emp[salary > 10000]
```


The Query Data Model

- A value is an ordered sequence of zero or more items.
- An item is a node or an atomic value.
- There are seven kinds of nodes:
 - Document Node
 - Element Node
 - Attribute Node
 - Text Node
 - Comment Node
 - Processing Instruction Node
 - Namespace Node

Examples of values

- 47
- <goldfish/>
- (1, 2, 3)
- (47, <goldfish/>, "Hello")
- ()
- An XML document
- An attribute standing by itself

Facts about values

- There is no distinction between an item and a sequence of length one
- There are no nested sequences
- There is no null value
- A sequence can be empty
- Sequences can contain heterogeneous values
- All sequences are ordered

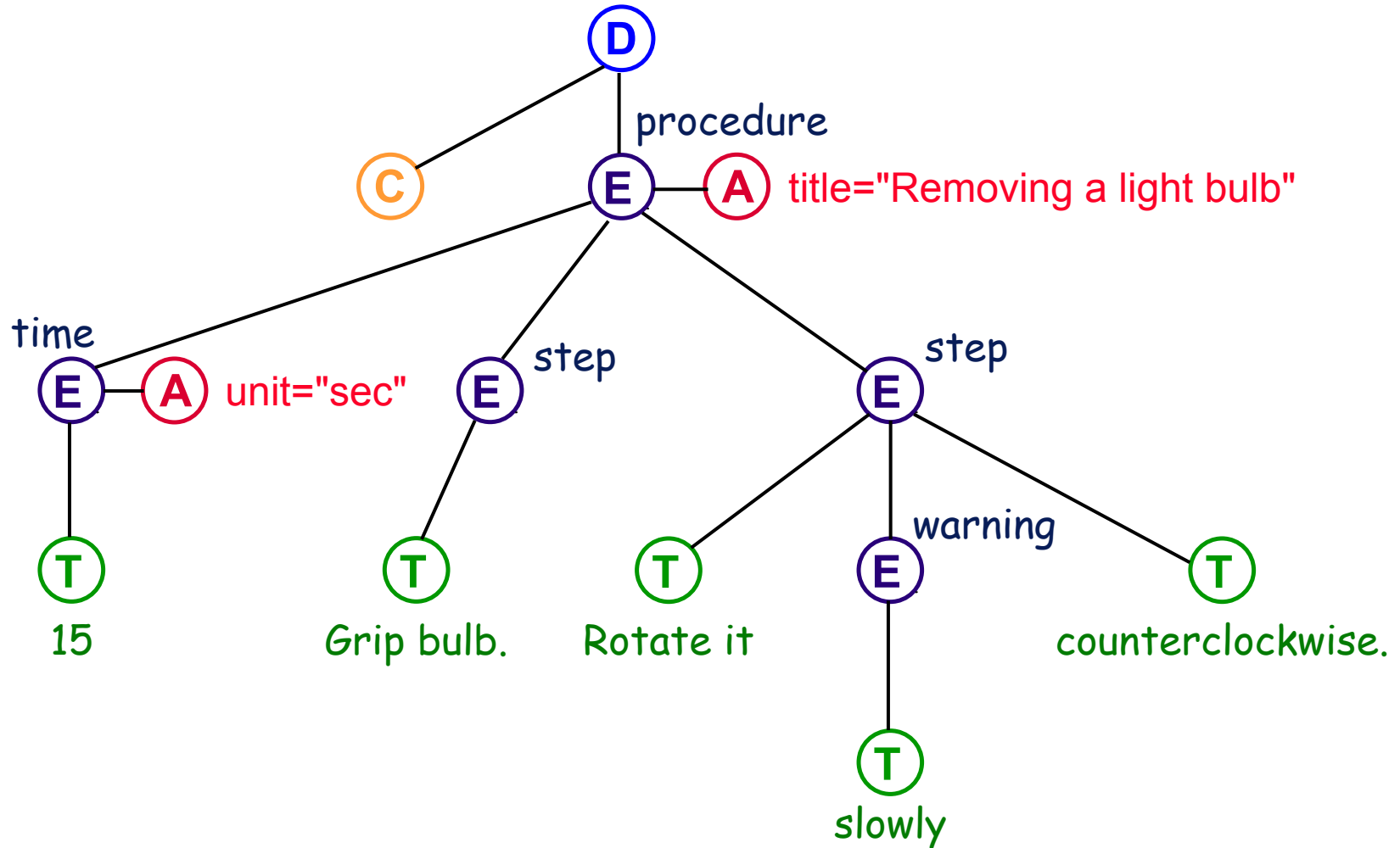
Facts about nodes

- Nodes have identity (atomic values don't)
- Element and attribute nodes have a type annotation
 - Generated by validating the node
 - May be a complex type such as PurchaseOrder
 - Type may be unknown ("anyType")
- Each node has a typed value:
 - a sequence of atomic values
 - Type may be unknown ("anySimpleType")
- There is a document order among nodes
 - Ordering among documents and constructed nodes is implementation-defined but stable

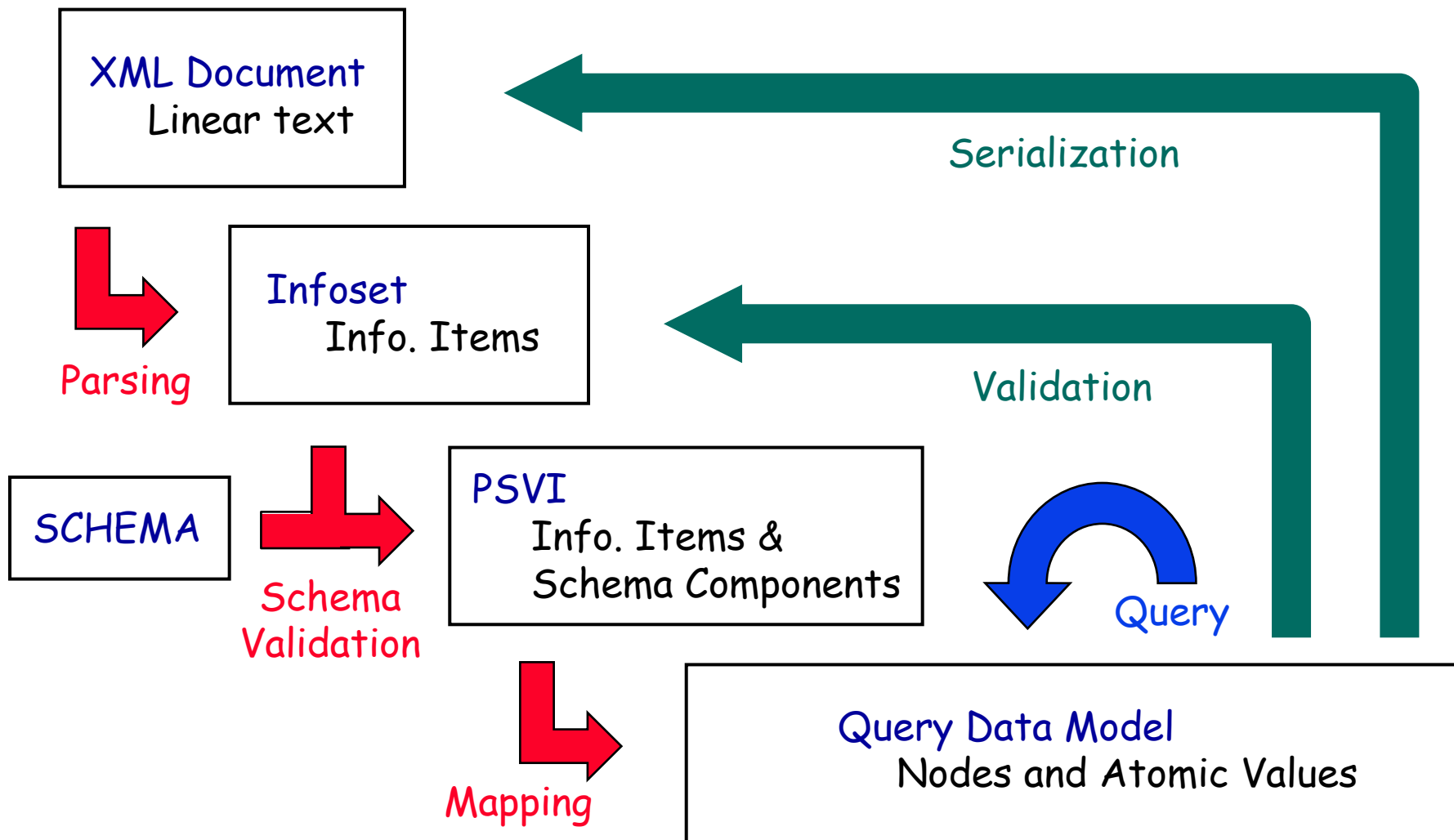
An XML Document ...

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

... and its Data Model Representation



XML and the Query Data Model



General XQuery Rules

- XQuery is a case-sensitive language
- Keywords are in lower-case
- Every expression has a value and no side effects
- Expressions are fully composable
- Expressions can raise errors
- Expressions (usually) propagate lower-level errors
 - Exception: if-then-else
- Comments look like this:
(: Houston, we have a problem :)

XQuery Expressions

- Literals: "Hello" 47 4.7 4.7E-2
- Constructed values:
true() false() date("2002-03-15")
- Variables: \$x
- Constructed sequences
 - \$a, \$b is the same as (\$a, \$b)
 - (1, (2, 3), (), (4)) is the same as 1, 2, 3, 4
 - 5 to 8 is the same as 5, 6, 7, 8

Functions

- XQuery functions have expressions for bodies and may be recursive
- Function calls
 - `three-argument-function(1, 2, 3)`
 - `two-argument-function(1, (2, 3))`
 - `one-argument-function()`
 - `zero-argument-function()`
- Functions are not overloaded (except certain built-ins)
- Subtype substitutability in function arguments

Path Expressions

- Path expressions are inherited from XPath 1.0
- A path always returns a sequence of distinct nodes in document order
- A path consists of a series of steps:
 - `/book/project[name = "Doghouse"]/step[5]`
- Each step can be any expression that returns nodes
- Here's what E1/E2 means:
 - Evaluate E1—it must be a set of nodes
 - For each node N in E1, evaluate E2 with N as context node
 - Union together all the E2-values
 - Eliminate duplicate node-ids and sort in document order

Path Expressions, cont'd.

- A step may contain an axis, a node test, and predicates
- The default axis is "child"
- XQuery does not support all the axes of XPath

Supported:

child
descendant
attribute
self
descendant-or-self
parent

Not supported:

ancestor
ancestor-or-self
preceding
preceding-sibling
following
following-sibling
namespace

Predicates

- Boolean expressions:

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

- Numeric expressions:

```
chapter[2]
```

- Existence tests:

```
book[appendix]
```

```
person[@married]    (Tests existence, not value!)
```

- Predicates can be used in path expressions:

```
//book[author = "Mark Twain"]/chapter[2]
```

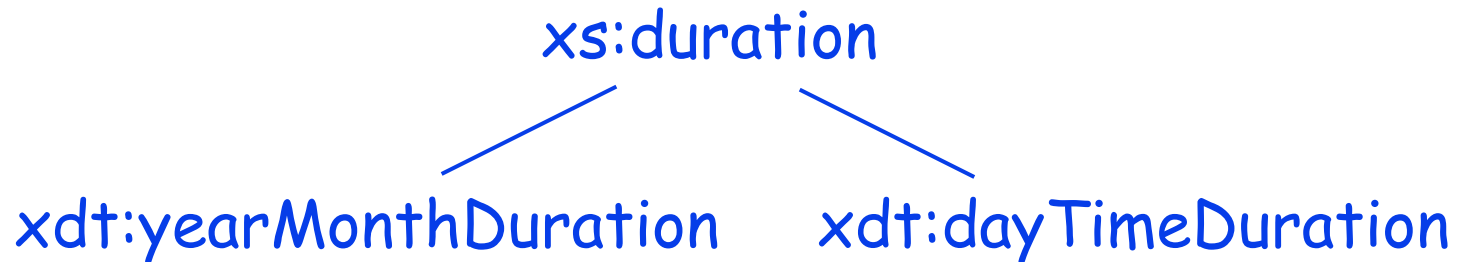
... and in other kinds of expressions:

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

Expressions, continued

- Combining sequences: `union intersect except`
 - return sequences of distinct nodes in document order
- Arithmetic operators: `+ - * div idiv mod`
 - Extract typed value from node
 - Multiple values => error
 - If operand is `()`, return `()`
 - Supported for numeric and date/time types
- Comparison operators
 - `eq ne gt ge lt le` compare single atomic values
 - `= != > >= < <=` implied existential semantics
 - `is is not` compare two nodes based on identity
 - `<< >>` compare two nodes based on document order

Two new built-in datatypes



- In XML Schema a "duration" may contain years, months, days, hours, minutes, and seconds
- XQuery defines two subtypes derived from duration
 - yearMonthDuration contains only years, months
 - dayTimeDuration contains only days, hours, minutes, sec's
- Arithmetic and comparison are supported within each subtype but not across subtypes

Logical Expressions

- Operators: `and` `or`
- Function: `not()`
- Return TRUE or FALSE (2-valued logic)
- "Early-out" semantics (need not evaluate both operands)
- Result depends on Effective Boolean Value of operands
 - If operand is of type boolean, it serves as its own EBV
 - If operand is `()`, zero, or empty string, EBV is FALSE
 - In any other case, EBV is TRUE
- Note that EBV of a node is TRUE, regardless of its content (even if the content is FALSE)!

Constructors

- To construct an element with a known name and content, use XML-like syntax:

```
<book isbn = "12345">  
  <title>Huckleberry Finn</title>  
</book>
```

- If the content of an element or attribute must be computed, use a nested expression enclosed in { }

```
<book isbn = "{$x}">  
  { $b/title }  
</book>
```

- If both the name and the content must be computed, use a computed constructor:

```
element { name-expr } { content-expr }  
attribute { name-expr } { content-expr }
```

Constructors, continued

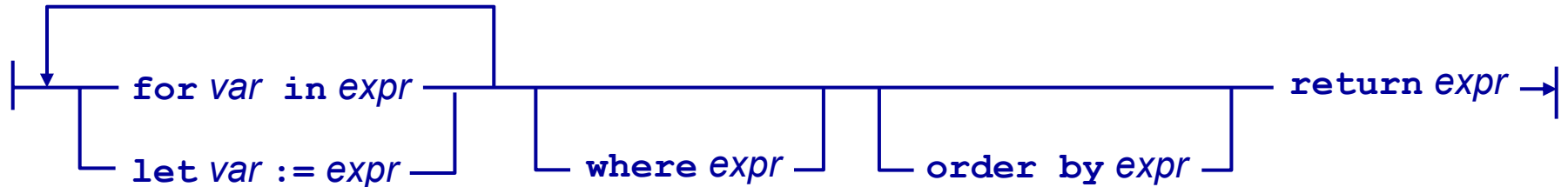
- An element constructor automatically validates the new element against "in-scope schema definitions"
 - Results in a type annotation
 - Can be generic: `xs:anyType`
- Validation mode (default = lax)
 - **Strict**: element must be defined in schema
 - **Lax**: element must match schema def'n if it exists
 - **Skip**: ignore this element
 - Mode is set in Prolog or by explicit Validate expression
- Validation context:
 - Schema path inside which current node is validated
 - Each constructed element adds its name to the context
 - Can be overridden by an explicit Validate expression

Constructors, continued

- Why should a constructed node be validated?
(Why not rely on the type of its content expression?)
- Example: `<a>{8}`
 - Type of content expression is integer
 - After validation, type of `<a>` may be `hatsize`
- Example: `<a>{1, "2"}`
 - Type of content expression is `(integer, string)`
 - After validation, type of `<a>` will certainly be different.
- Automatic validation allows static typing to rely on the type of a constructed element.

FLWOR Expressions

- A FLWOR expression binds some variables, applies a predicate, and constructs a new result.



FOR and LET clauses generate a list of tuples of bound variables, preserving input order.

WHERE clause applies a predicate, eliminating some of the tuples

ORDER BY clause imposes an order on the surviving tuples

RETURN clause is executed for each surviving tuple, generating an ordered list of outputs

An Example Query

- "Find the description and average price of each red part that has at least 10 orders"

```
for $p in doc("parts.xml")
    //part[color = "Red"]
let $o := doc("orders.xml")
    //order[partno = $p/partno]
where count($o) >= 10
order by count($o) descending
return
    <important_red_part>
        { $p/description }
        <avg_price> {avg($o/price)} </avg_price>
    </important_red_part>
```

Expressions, continued

- **unordered** (*expr*)
 - Indicates that the order of *expr* is not significant
 - Provides opportunity for an optimizer
- **if** (*expr1*) **then** *expr2* **else** *expr3*
 - Uses effective boolean value, like **and** and **or**
- $\left\{ \begin{array}{l} \text{some} \\ \text{every} \end{array} \right\}$ *var* **in** *expr1* **satisfies** *expr2*
 - Also based on effective boolean value
 - Allow early-out for errors

SequenceType

- "SequenceType" is the syntax used to specify a type in an XQuery expression
 - Function parameters and results
 - Path expressions **(NEW!)**
 - CAST, INSTANCE OF, etc.
- A SequenceType can be:
 - A named atomic type: `xs:decimal`
 - A kind of node: `element()`, `attribute()`, `text()`, `node()`
 - An element, attribute, or document node qualified by its name and/or type annotation: `element(shipto, address)`
 - Any of the above followed by an occurrence indicator:
`*`, `+`, `?`

Ways of qualifying a node ("Kind tests")

- "Kind tests" for an element node:
 - `element(N, T)`: name is N and type is T
 - `element(N, *)`: name is N, any type
 - `element(*, T)`: any name, type is T
 - `element(N)`: name is N, type is schema type of N
 - `element(P)`: conforms to name and type of schema path P
(Example: `order/item/cost`)
 - `element()`: any name or type
- Name N is matched by any name in the "substitution group" of N
- Type T is matched by any type derived by restriction or extension from T

Using "Kind Tests"

- In a function signature:

```
define function
```

```
  zip($x as element(*, USAddress)?) as string?
```

- In a path expression:

```
//order/element(*, USAddress)/zipcode
```

- This is an extension of XPath-1.0 "kind tests" such as `text()` and `node()`.

Testing Types

- Instance Of expression returns TRUE or FALSE:

```
$order/shipto instance of element(*, Address)
```

- Typeswitch expression executes one branch, based on the type of its operand:

```
typeswitch($order/shipto)
  case $us as element(*, USAddress)
    return $us/zipcode
  case $uk as element(*, UKAddress)
    return $uk/postcode
  default
    return error("unknown address type")
```

Tinkering with Types

- `expr cast as ST`
 - Converts value to target type (may return error)
 - Example: `($product/(price * discount)) cast as decimal`
- The following casts are (statically) valid:
 - Certain predefined pairs of atomic types
Example: `integer -> string`
 - Derived atomic type \leftrightarrow its supertype (checks "facets")
Example: `hatsize \leftrightarrow integer`
 - String or untyped atomic \rightarrow derived atomic type (checks facets)
Example: `string -> hatsize`
 - Any transitive combination of the above
Example: `hatsize -> IQ`

Tinkering with Types (cont'd.)

- Constructor functions
 - Every atomic type has a constructor function
 - Includes both built-in and user-defined atomic types
 - Example: `hatsize(expr)`
 - Semantics are identical to `expr cast as hatsize`

Tinkering with Types (cont'd.)

- `expr castable as ST`

- Predicate, returns Boolean
- True if value of `expr` can be cast to type `ST`
- Can be used to choose a valid target type for `expr`


```
if ($x castable as hatsize) then hatsize($x)
else if ($x castable as IQ) then IQ($x)
else string($x)
```

- `expr treat as ST`

- Serves as a compile-time "promise"
- Static type of `treat` expression is `ST`
- At run-time, returns an error if dynamic type of `expr` is not `ST`
- `$myaddress treat as element(*, USAddress)`

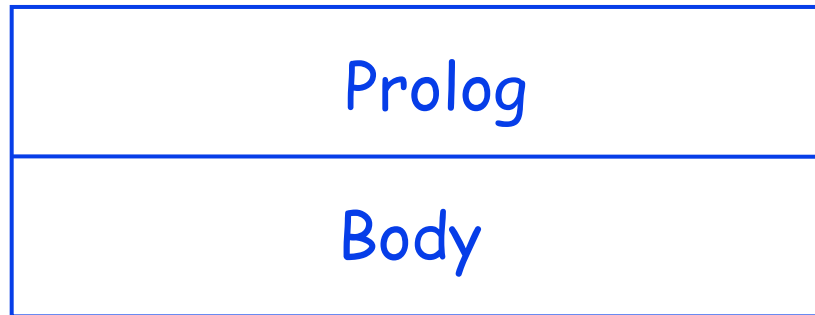
Explicit validation

- Syntax:

`validate`  { *expr* }

- Semantics: evaluate *expr*, then convert its value to an XML Infoset and invoke schema validation on it
 - Validated nodes get new identity
 - Attributes may get default values
- Rarely necessary since element constructors automatically validate
- Used mainly to control validation mode and context

Structure of an XQuery



- The Prolog contains:
 - Namespace declarations (bind namespace prefixes to URI's)
 - Schema imports (import namespaces and their schemas)
 - Module imports (import function definitions and variables)
 - Function definitions (may be recursive)
 - Declarations of global and external variables
 - Controls for whitespace handling, default collation, etc.
- The Body contains:
 - an expression that defines the result of the query

Namespace Declarations

- In XQuery, all names are two-part "QNames"
- A QName consists of a namespace-prefix and a local name, separated by a colon
- A namespace-prefix is shorthand for a namespace, which is a URI
- Namespace prefixes are mapped to namespaces by namespace declarations in the Prolog:

`acme:product`

`http://www.acme.com/names`

```
declare namespace
```

```
  acme = "http://www.acme.com/names"
```


Namespace Declarations, continued

- Two default namespaces can be declared:
 - `default element namespace = "http://whatever-1"`
(applies to unqualified names of elements and types)
 - `default function namespace = "http://whatever-2"`
(applies to unqualified names of functions)
- An element constructor can also declare namespace prefixes for use within the scope of the element
 - `<foo xmlns:bar = "http://www.bar.com/names">`
- Names are always compared in expanded form
 - `abc:product` and `xyz:product` are the same name if `abc` and `xyz` are bound to the same namespace

Schema Imports

- A namespace is defined by a schema
- This statement binds a prefix to a namespace *and* imports the schema that defines the namespace:

```
import schema
    namespace acme = "http://www.acme.com/names"
```

- All definitions in the schema become visible (used for validation)
- The system is responsible for finding the schema
- You can provide a "hint" about the schema location:

```
import schema
    namespace acme = "http://www.acme.com/names"
        at "http://www.acme.com/schemas/names.xsd"
```

Function Definitions

- Example of a function definition:

```
define function depth($n as node()) as xs:integer
{
  (: A node with no children has depth 1 :)
  (: Else, add 1 to max depth of children :)
  if (empty($n/*)) then 1
  else max(for $c in $n/* return depth($c)) + 1
}
```

- Example of an external function definition:

```
define function longitude($c as element(city))
  as double external
```

- Linkage conventions for external functions are implementation-defined.

Function Definitions (continued)

- Function definitions may not be overloaded in Version 1
 - Much XML data is untyped
 - XQuery attempts to cast arguments to the expected type
 - Example: `abs($x)` expects a numeric argument
 - If `$x` is a number, return its absolute value
 - If `$x` is untyped, cast it to a number
 - If `$x` is a node, extract its value and treat as above
 - This "argument conditioning" conflicts with function overloading
 - XML Schema substitution rules are already very complex (two kinds of inheritance; substitution groups; etc.)
 - A function can simulate overloading by branching on the type of its argument, using a `typeswitch` expression

Modules

- A query can include several "modules"
- Main Module:
 - contains the query body
 - is executable
- Library Module:
 - defines functions and variables
 - declares its "target namespace"
 - can be "imported" by other library or main modules
 - exports functions and variables in the "target namespace"
- Importing a module
 - "import module" clause in Prolog

Modules (continued)

- Example of a library module:

```
module "http://www.ibm.com/xquery-functions"
import schema namespace abc = "http://abc.com"
import module namespace xyz = "http://xyz.com"
define variable $pi as double {3.14159}
define function
    triple($x as xs:integer) as xs:integer
    { 3 * $x }
```

- A main module can import this library as follows:

```
import module namespace ibmfns =
    "http://www.ibm.com/xquery-functions"
```

- The main module must *also* import schema **abc** and module **xyz**

Two Phases in Query Processing

- Static analysis (compile-time; optional)
 - Depends only on the query itself
 - Infers result type of each expression, based on types of operands
 - Raises error if operand types don't match operators
 - Purpose: catch errors early, guarantee result type
 - May be helpful in query optimization
- Dynamic evaluation (run-time)
 - Depends on input data
 - Computes the result value based on the operand values

Formal Semantics

- The formal semantics of XQuery are specified by a set of "inference rules"

$$\text{statEnv} \mid\text{- } Expr_1 : \underline{\text{xs:boolean}} \quad \text{statEnv} \mid\text{- } Expr_2 : Type_2 \quad \text{statEnv} \mid\text{- } Expr_3 : Type_3$$

$$\text{statEnv} \mid\text{- if } (Expr_1) \text{ then } Expr_2 \text{ else } Expr_3 : (Type_2 \mid Type_3)$$
$$\text{dynEnv} \mid\text{- } Expr_1 \Rightarrow \text{true} \quad \text{dynEnv} \mid\text{- } Expr_2 \Rightarrow \text{value}_2$$

$$\text{dynEnv} \mid\text{- if } Expr_1 \text{ then } Expr_2 \text{ else } Expr_3 \Rightarrow \text{value}_2$$
$$\text{dynEnv} \mid\text{- } Expr_1 \Rightarrow \text{false} \quad \text{dynEnv} \mid\text{- } Expr_3 \Rightarrow \text{value}_3$$

$$\text{dynEnv} \mid\text{- if } Expr_1 \text{ then } Expr_2 \text{ else } Expr_3 \Rightarrow \text{value}_3$$

Processing Phases, continued

- If a query passes static analysis, it may still raise an error at evaluation time
 - It may divide by zero
 - Casts may fail. Example:
`cast as integer($x)` where value of `$x` is "garbage"
- If a query fails static type checking, it may still evaluate successfully and return a useful result. Example (with no schema):
`$emp/salary + 1000`
 - Static semantics says this is a type error
 - Dynamic semantics executes it successfully if `$emp` has exactly one salary subelement with a numeric value

Optional Features

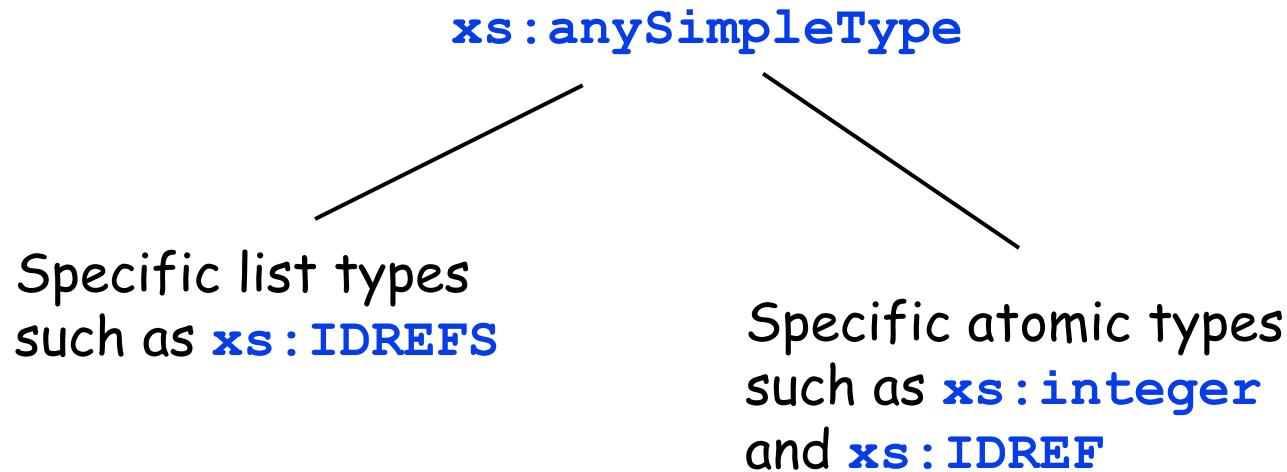
- The language consists of Basic XQuery and two optional features:
- Schema Import
 - If not implemented, system recognizes only built-in types
 - All types appearing in a query must be built-in types
 - Derived types in input documents are promoted to the nearest built-in base type as needed for processing
- Static Typing
 - If not implemented, a system need not detect static type errors
 - Static type analysis for optimization is permitted
 - Dynamic type errors must still be detected

Summary: XQuery on one slide

- Query prolog: namespaces, schema and module imports, function definitions, certain controls
- Composable expressions:
 - Literals & variables
 - Sequences
 - Function calls
 - Path expressions
 - Predicates
 - Constructors
 - Union, intersect, except
 - Comparisons
 - and, or
 - Arithmetic
 - FLWOR expressions
 - unordered
 - if ... then ... else
 - some, every
 - instance of
 - typeswitch
 - cast as
 - castable
 - treat as
 - validate

In Depth (1): Untyped Data

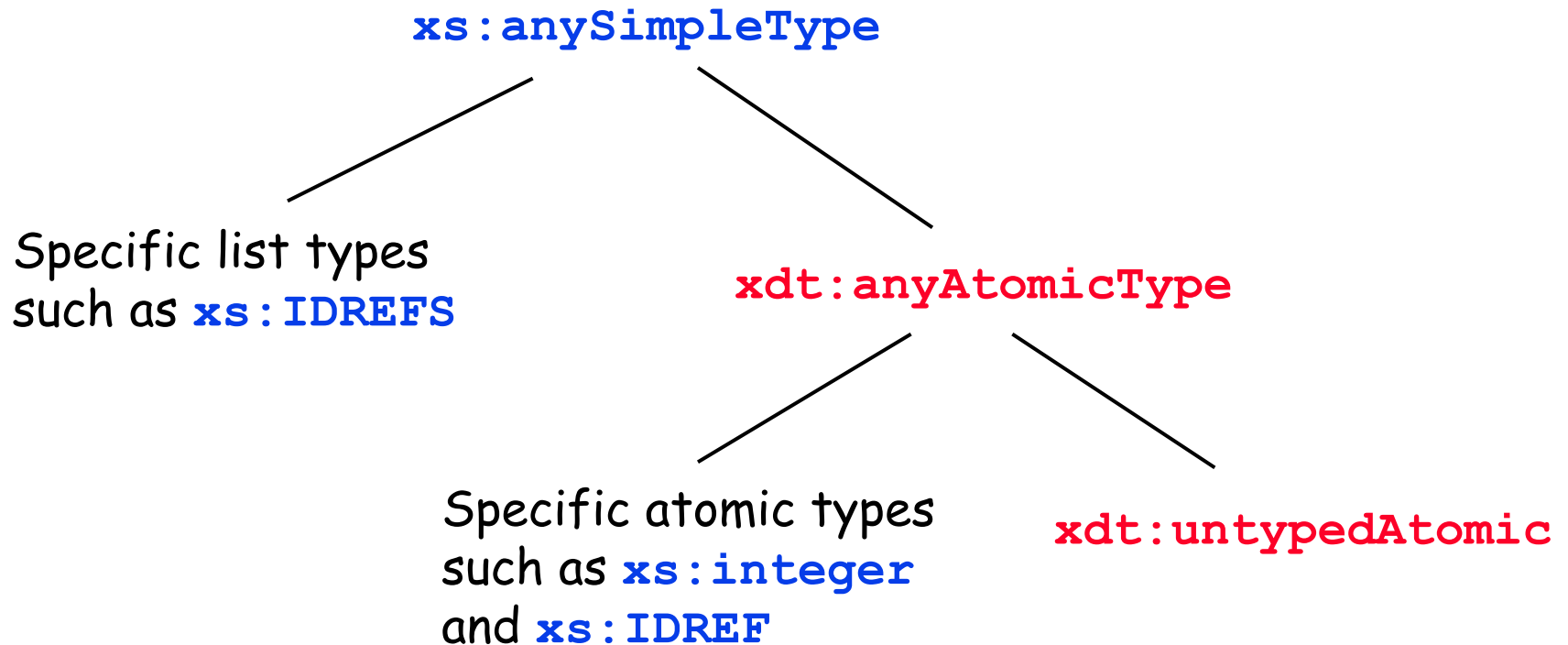
- The hierarchy of simple types in XML Schema:



- At run-time, an element may have a "type" attribute (**xsi:type="decimal"**, **xsi:type="float"** etc.)
- Dynamic type must be a subtype of static type.

Untyped Data (continued)

- Type hierarchy as enhanced by XQuery:



Untyped Data (continued)

- How should run-time untyped data be treated?
- Arithmetic: always treat `xdt:untypedAtomic` as `xs:double`
 - `length * width`
 - `salary + bonus`
- General Comparisons: treat `xdt:untypedAtomic` as the type of the other operand; `xs:string` if both are untyped
 - `age > 21`
 - `city = "San Francisco"`
 - `city = county`

Transitivity of Comparisons

- XPath general comparison operators (`=`, `!=`, `<`, `<=`, `>`, `>=`) have existential semantics:
`//book[author = "Chris Date"]`
- Therefore these operators are not transitive:
`(1, 2) = (2, 3)` and `(2, 3) = (3, 4)` but `(1, 2) != (3, 4)`
- Value comparison operators (`eq`, `ne`, `gt`, `ge`, `lt`, `le`) were defined to require single values as operands.
- Transitivity problem: rank these values by `lt`:
`untyped(1)`, `integer(2)`, `untyped(03)`
- Solution: value comparisons always treat untyped values as strings, to preserve transitivity.

Dynamic Dispatch

- Arithmetic and comparison operators depend on dynamic type of both operands

```
for $e in //emp return $e/salary + $e/bonus
```

- Individual run-time values may have a type attribute that is more specific than their static type

```
(xsi:type="integer" etc.)
```

- Processing these values correctly may require run-time dispatch of operators

- Rationale:

- If a schema is available, static type analysis can usually select the proper operator at compile time
- If data is untyped, the user can insert an explicit cast
- Otherwise, you must pay the cost of run-time dispatch

In depth (2): Missing data

- The Query Data Model does not have a "null value"
- XML notation provides the following "states":
 - Present, with a value:
`<car> <mileage>25</mileage> </car>`
 - Present but empty:
`<car> <mileage/> </car>`
 - Absent:
`<car> </car>`
- XQuery leaves it up to the user to map these "notation states" onto "knowledge states"
 - Known, unknown, not applicable, etc.

Missing Data (continued)

- How do XQuery operators behave on missing data?
 - Operators extract "typed values" from nodes
 - The typed value of a node is a sequence of atomic values
 - If the node is absent or empty, typed value = ()
- Arithmetic operators (+, -, *, div, idiv, mod)
 - If either operand is (), result is ()
 - Similar to "null propagation" in SQL
- General comparison operators (=, !=, <, <=, >, >=)
 - Based on existential semantics
 - If either operand is (), result is False
 - If A is (), **A = B** and **A != B** are both False
 - If A is (1, 2) and B is (2, 3), **A = B** and **A != B** are both True

Missing Data (continued)

- Value comparison operators (`eq`, `ne`, `lt`, `le`, `gt`, `ge`)
 - By definition, each operand is exactly one atomic value
 - If either operand is `()`, raise an error
- Functions
 - Function signature declares whether `()` is acceptable arg.
`function f1($x as xs:integer?) as xs:integer?`
 - Function body determines how `()` is handled.
- Logical operators (`and`, `or`, `not()`)
 - `()` is consistently treated as False
 - `//person[hat or coat]` is an existence test in XPath 1.0
 - XQuery is based on 2-valued logic

Missing Data (continued)

- 3-valued logic can be simulated by user functions:
`and3`, `or3`, `not3`, `eq3`, `ne3`, `gt3`, etc.
- Various truth tables are possible:

<code>and3</code>	T	F	()
T	T	F	()
F	F	F	F
()	()	F	()

<code>not3</code>	
T	F
F	T
()	()

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

In depth (3): Errors

- Any expression can raise an error
- Errors are identified by unique codes
- An error may carry a value
- Explicit function: `error(expr)`
- In general, expressions propagate errors
- An expression with multiple operands can choose which error to propagate
- Mechanism for reporting errors is implementation-defined

Errors and Indeterminacy

- `true or error` can return true or raise an error
- `false and error` can return false or raise an error
- `some $x in (1, 2, error) satisfies $x < 10` can return true or raise an error
- `every $x in (8, 9, error) satisfies $x < 5` can return false or raise an error
- General comparison operators behave like existentials:
`(47, error) = 47` can return true or raise an error
- If one operand raises an error, an expression need not evaluate its other operands

In depth (4): Ordering

- Approach #1: a separate **sort by** operator
`//part[color = "Red"] sort by price`
- Approach #2: an **order by** clause in FLWOR expr
`for $p in //part[color = "Red"]
order by $p/price
return $p`
- Advantages of Approach #1:
 - More orthogonal (sorting does not require FLWOR)
 - More concise in many cases
- Disadvantages of Approach #1:
 - More difficult to sort by a key that is not returned
 - Separates sorting from iteration, complicates optimization
- Decision: XQuery uses Approach #2 only.

Summary and Prognosis

- The basic structure of XQuery is reasonably stable
- Several parts of the XQuery spec will enter "last call" in 2003.
- Some important features will be deferred until after Version 1:
 - Data modification (insert, delete, update)
 - Full-text search (but requirements and use cases are available)
 - View definitions and other DDL
 - Language bindings (for applications and external functions)
 - Better error handling: try/catch?
 - More query features: explicit grouping? etc.