XQuery: Contents

What is XQuery

- A **query language** for XML data and documents
- Can be used to **extract sections** of XML documents, but also to **manipulate and transform the results**
  - ◇ Selecting information based on specific criteria
  - ◇ Filtering out unwanted information
  - ◇ Searching for information within a document or a set of documents
  - ◇ Joining data from multiple documents
  - ◇ Sorting, grouping, and aggregating data
  - ◇ Transforming and restructuring XML data into another XML vocabulary or structure
  - ◇ Performing arithmetic calculations on numbers and dates
  - ◇ Manipulating strings to reformat text
- XQuery 1.0 **does not provide updates**
  - ◇ XQuery Update Facility is an extension to XQuery that became a W3C Candidate Recommendation on 14 March 2008

Common Uses for XQuery

- Extracting information from a relational database for use in a web service
- Generating reports on data stored in a database for presentation on the Web as XHTML
- Searching textual documents in a native XML database and presenting the results
- Pulling data from databases or packaged software and transforming it for application integration
- Combining content from non-XML sources to implement content management and delivery
- Ad hoc querying of standalone XML documents for the purposes of testing or research

Processing Scenarios (1)

- XQuery is sometimes called the “SQL of XML”
- **Native XML database**: database designed for XML content from the ground up, as opposed to a traditional relational database
- Its data model is based on hierarchical documents and collections of documents (rather than tables and columns)
- Used for narrative content and other data that is less predictable than what would typically be stored in a relational database
- Examples: Tamino, eXist (open source), Berkeley DB XML, MarkLogic Server, TigerLogic XDMS, X-Hive/DB
- Provide traditional capabilities of databases: data storage, indexing, querying, loading, extracting, concurrency control (ACID properties), backup, recovery
- Also provide additional functionality, e.g., advanced full-text searching, document conversion services, end-user interfaces

Processing Scenarios (2)

- Major **relational database products** (Oracle, IBM DB2, Microsoft SQL Server) also have support for XML and XQuery
- Early implementations of XML in relational databases stored XML in table columns as blobs or character strings and provided query access to those columns
- They are increasingly blurring the line between native XML databases and relational databases with new features that allow XML to be stored natively
- **Independent XQuery processors**: not embedded in a database product
- Used on physical XML documents stored as files on a file system or on the Web
- Might also operate on XML data passed in memory from some other process
- Most notable product in this category is Saxon, has both open source and commercial versions

Introduction to XQuery: Example Data (1)

- `catalog.xml`: product catalog containing general information about products

```
<catalog>
  <product dept="WMN">
    <number>557</number>
    <name language="en">Fleece Pullover</name>
    <colorChoices>navy black</colorChoices>
  </product>
  <product dept="ACC">
    <number>563</number>
    <name language="en">Floppy Sun Hat</name>
  </product>
  <product dept="ACC">
    <number>443</number>
    <name language="en">Deluxe Travel Bag</name>
  </product>
  <product dept="MEN">
    <number>784</number>
    <name language="en">Cotton Dress Shirt</name>
    <colorChoices>white gray</colorChoices>
    <desc>Our <i>favorite</i> shirt!</desc>
  </product>
</catalog>
```

Introduction to XQuery: Example Data (2)

- `prices.xml`: contains prices for the products, based on effective dates

```
<prices>
  <priceList effDate="2006-11-15">
    <prod num="557">
      <price currency="USD">29.99</price>
      <discount type="CLR">10.00</discount>
    </prod>
    <prod num="563">
      <price currency="USD">69.99</price>
    </prod>
    <prod num="443">
      <price currency="USD">39.99</price>
      <discount type="CLR">3.99</discount>
    </prod>
  </priceList>
</prices>
```

Introduction to XQuery: Example Data (3)

- `order.xml`: list of products ordered along with quantities and colors

```
<order num="00299432" date="2006-09-15" cust="0221A">
  <item dept="WMN" num="557" quantity="1" color="navy"/>
  <item dept="ACC" num="563" quantity="1"/>
  <item dept="ACC" num="443" quantity="2"/>
  <item dept="MEN" num="784" quantity="1" color="white"/>
  <item dept="MEN" num="784" quantity="1" color="gray"/>
  <item dept="WMN" num="557" quantity="1" color="black"/>
</order>
```

Path Expressions: XPath (1)

- Used to traverse an XML tree to select elements and attributes of interest
- Similar to paths used for filenames in many operating systems
- Consist of a series of **steps**, separated by slashes, that traverse the elements and attributes in the XML documents
- Select all the `product` elements from the `catalog.xml` document

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

  Result: the four `product` elements in the input document, with the same attributes and contents
- Path expressions can also return attributes, using the `@` symbol
- `*`: wildcard that indicates any element name
- Select the `dept` attributes of the `product` element in the input document

  ```
  doc("catalog.xml")/*/product/@dept
  ```
- `//`: return elements that appear anywhere in the document starting from a certain node

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

Path Expressions: XPath (2)

- **Predicates** allow to filter out elements/attributes that do not meet a criterion
- Select only those `product` elements whose `dept` attribute value is `ACC`

  ```
  doc("catalog.xml")/catalog/product[@dept = "ACC"]
  ```
- When a predicate contains a number, it serves as an index
- Return the second `product` element in the catalog

  ```
  doc("catalog.xml")/catalog/product[2]
  ```
- Order of elements in an XML document is important ($\neq$ relational databases)
- Path expressions: convenient because of their compact, easy-to-remember syntax
- **Limitation**: return elements and attributes as they appear in input documents
  - ◇ Appear in the result with the same names, the same attributes and contents, and in the same order as in the input document

FLWORs

- FLWOR expression: basic structure of many (but not all) queries
- Pronounced "flower", it stands for `for`, `let`, `where`, `order by`, `return`, the keywords used in the expression
- FLWORs, unlike path expressions, allows one to **manipulate**, **transform**, and **sort** results
- Return the names of all products in the `ACC` department

  ```
  for $product in doc("catalog.xml")/catalog/product
  let $name := $product/name
  where $product/@dept = "ACC"
  order by $name
  return $name
  ```
- Equivalent to the following path expression

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

  ```
  <name language="en">Deluxe Travel Bag</name>
  <name language="en">Floppy Sun Hat</name>
  ```

Clauses of a FLWOR Expression

- `for`
  - ◇ Iterates through the `product` nodes: the rest of the FLWOR expression is evaluated once for each of the four products
  - ◇ Each time, a variable named `$prod` is bound to a different `product` element
- `let`
  - ◇ Used to set the value of a variable using `:=`
  - ◇ Unlike a `for` clause, it does not set up an iteration
- `where`
  - ◇ Selects only products in the `ACC` department
  - ◇ Has the same effect as a predicate `[@dept = "ACC"]` in a path expression
- `order by`
  - ◇ Sorts the results by product name, which is not possible with path expressions
- `return`
  - ◇ Indicates that the `product` element's `name` children should be returned

Adding XML Elements (1)

- XML constructors can be used to create elements and attributes that appear in the query results
- Wrap the name elements in a `ul` element in XHTML
```
<ul>{
  for $product in doc("catalog.xml")/catalog/product
  where $product/@dept='ACC'
  order by $product/name
  return $product/name
}</ul>
```
Results
```
<ul>
  <name language="en">Deluxe Travel Bag</name>
  <name language="en">Floppy Sun Hat</name>
</ul>
```
- Everything between the `ul` start tag and end tag is an **element constructor**
- **Enclosed expression**: expression between `{ }` that is to be evaluated
- Enclosed expression returns two elements, which become children of `ul`

Adding XML Elements (2)

- Content in an element constructor outside `{ }` appears in the results as is
- Query
```
<h1>There are {count(doc("catalog.xml")//product)} products.</h1>
```
Results
```
<h1>There are 4 products.</h1>
```
- Wrap each resulting `name` element in its own `li` element
```
<ul>{
  for $product in doc("catalog.xml")/catalog/product
  where $product/@dept='ACC'
  order by $product/name
  return <li>{$product/name}</li>
}</ul>
```
Results
```
<ul>
  <li><name language="en">Deluxe Travel Bag</name></li>
  <li><name language="en">Floppy Sun Hat</name></li>
</ul>
```

Adding XML Elements (3)

- Built-in function `data()` can be used to extract the contents of an element
- Query
```
<ul>{
  for $product in doc("catalog.xml")/catalog/product
  where $product/@dept='ACC'
  order by $product/name
  return <li>{data($product/name)}</li>
}</ul>
```
Results
```
<ul>
  <li>Deluxe Travel Bag</li>
  <li>Floppy Sun Hat</li>
</ul>
```

Adding XML Attributes

- Attributes can also be added to results using an XML-like syntax
```
<ul type="square">{
  for $product in doc("catalog.xml")/catalog/product
  where $product/@dept='ACC'
  order by $product/name
  return <li class="{$product/@dept}">{data($product/name)}</li>
}</ul>
```
Result
```
<ul type="square">
  <li class="ACC">Deluxe Travel Bag</li>
  <li class="ACC">Floppy Sun Hat</li>
</ul>
```
- Attribute values can either be literal text or enclosed expressions
- For attributes it is not necessary to use the `data` function to extract the value
- Constructors above are **direct constructors**, they use an XML-like syntax
- **Computed constructors** are used for elements and attributes with dynamically determined names (see later)

Functions

- There are over 100 functions built into XQuery, covering a broad range of functionality
- Functions can be used to manipulate strings and dates, perform mathematical calculations, combine sequences of elements, . . .
- Users can also define their functions, either in the query itself, or in an external library
- Both built-in and user-defined functions can be called from almost any place in a query
- For instance, examples above calls the `doc` function in a `for` clause, and the `data` function in an enclosed expression

Joins

- FLWORs can be used to join data from multiple sources
- Join information from product catalog and orders: obtain a list of all the items in the order, along with their number, name, and quantity

```
for $item in doc("order.xml")//item
let $name := doc("catalog.xml")//product[number = $item/@num]/name
return <item num="{$item/@num}"
  name="{$name}" quan="{$item/@quantity}"/>
```

Results

```
<item num="557" name="Fleece Pullover" quan="1"/>
<item num="563" name="Floppy Sun Hat" quan="1"/>
<item num="443" name="Deluxe Travel Bag" quan="2"/>
<item num="784" name="Cotton Dress Shirt" quan="1"/>
<item num="557" name="Fleece Pullover" quan="1"/>
```

- The `for` clause sets up an iteration through each item from the order
- For each item, the `let` clause goes to the product catalog and gets the name of the product

Aggregating and Grouping Values

- One common use for XQuery is to summarize and group XML data
- Give the number of items contained in an order, grouped by department

```
for $d in distinct-values(doc("order.xml")//item/@dept)
let $items := doc("order.xml")//item[@dept = $d]
order by $d
return <department name="{$d}"
        totQuantity="{sum($items/@quantity)}"/>
```

Results

```
<department name="ACC" totQuantity="3"/>
<department name="MEN" totQuantity="2"/>
<department name="WMN" totQuantity="2"/>
```

- The `for` clause iterates over the list of distinct departments
- The `let` clause binds `$items` to the `item` elements for a particular department
- The `sum` function calculates the totals of the `quantity` attribute values for the items in `$items`

XQuery: Contents

XQuery and XPath

- XPath started out as a language for selecting elements and attributes from an XML document while traversing its hierarchy and filtering out unwanted content
- XPath 1.0 specifies path expressions and a limited set of functions
- XPath 2.0 now encompasses a wide variety of expressions and functions
- XQuery 1.0 and XPath 2.0 overlap to a very large degree
- Have the same data model and the same set of built-in functions and operators
- XPath 2.0 is essentially a subset of XQuery 1.0
- XQuery has a number of features that are not included in XPath, such as FLWORs and XML constructors
- These features are not relevant to selecting, but instead have to do with structuring or sorting query results
- The two languages are consistent: any expression that is valid in both languages evaluates to the same value using both languages

XQuery Versus XSLT

- XSLT is a language for transforming XML documents into other documents
- There is a lot of overlap in the capabilities of XQuery and XSLT
- XSLT 2.0 is based upon XPath 2.0: same data model, built-in functions, and operators as XQuery, many of the same expressions
- Some differences between XQuery and XSLT
  - ◇ XSLT optimized for transforming entire documents; loads the entire input document into memory ⇔
    XQuery optimized for selecting fragments of data, e.g., from a database; designed to be scalable and to take advantage of database features such as indexes for optimization
  - ◇ XQuery has a more compact non-XML syntax, which is sometimes easier to read and write (and embed in program code) than the XML syntax of XSLT
  - ◇ XQuery designed to select from a collection of documents ⇔
    XSLT 2.0 stylesheets can operate on multiple documents, but XSLT processors are not optimized for this less common use case

XQuery Versus SQL

- XQuery borrows ideas from SQL, many of the designers of XQuery were also designers of SQL
- The line between XQuery and SQL may seem clear: XQuery is for XML, and SQL is for relational data
- This line is increasingly blurred: relational database vendors are putting XML frontends on their products and allowing XML to be stored in relational databases
- XQuery is unlikely to replace SQL for highly structured data traditionally stored in relational databases
- Most likely, the two will coexist: XQuery for querying less-structured data, SQL for highly structured relational data

XQuery and XML Schema

- XML Schema is a standard for defining schemas, used to validate XML documents and to assign types to XML elements and attributes
- XQuery uses the type system of XML Schema, which includes built-in datatypes such as decimal, date, and string
- XML Schema allows users to define their types based on the built-in types
- If an input document to a query has a schema, the types can be used when evaluating expressions on the input data
- This also has the advantages of allowing the processor to better optimize the query and to catch errors earlier
- XQuery users are not required to use schemas: it is possible to write a complete query with no mention of schemas or any of the schema types
- However, functions and operators operate on typed data, so it is useful to understand the type system and use the built-in types, even if no schema is present

The XQuery Data Model

- Used to define formally all the values used within queries
- Officially known as the XQuery 1.0 and XPath 2.0 Data Model, or XDM
- It is not the same as the Infoset (the W3C model for XML documents) because it has to support values that are not complete XML documents, such as sequences of elements (without a single outermost element) and atomic values
  - ◇ **Node**: An XML construct such as an element or attribute
  - ◇ **Atomic value**: A simple data value with no markup associated with it
  - ◇ **Item**: A generic term that refers to either a node or an atomic value
  - ◇ **Sequence**: An ordered list of zero, one, or more items

Nodes

- Used to represent XML constructs such as elements and attributes
- XQuery uses six kinds of nodes
  - ◇ **Element nodes**: Represent an XML element
  - ◇ **Attribute nodes**: Represent an XML attribute
  - ◇ **Document nodes**: Represent an entire XML document (not its outermost element)
  - ◇ **Text nodes**: Represent some character data content of an element
  - ◇ **Processing instruction nodes**: Represent an XML processing instruction
  - ◇ **Comment nodes**: Represent an XML comment

The Node Family

- Each node can have a number of relatives
  - ◇ **Children**: An element may have zero, one, or several other elements as its children. It can also have text, comment, and processing instruction children. Attributes are not considered children of an element. A document node can have an element child (the outermost element), as well as comment and processing instruction children.
  - ◇ **Parent**: The parent of an element is either another element or a document node. The parent of an attribute is the element that carries it
  - ◇ **Ancestors**: A node's parent, parent's parent, etc.
  - ◇ **Descendants**: A node's children, children's children, etc.
  - ◇ **Siblings**: The other children of its parent. Attributes are not considered to be siblings.

Node Identity and Name

- Every node has a unique identity
- Two XML elements in the input document may contain the exact **same data**, but that does not mean they have the **same identity**
- Identity is unique to each node and is assigned by the query processor
- Identity values cannot be retrieved, but identities can be compared with the `is` operator
- In addition to their identity, element and attribute nodes have **names**
- These names can be accessed using the built-in functions `node-name`, `name`, and `local-name`

String and Typed Values of Nodes

- Two kinds of values for a node: string and typed
- All nodes have a string value
  - ◇ Element node: its character data content and that of all its descendant elements concatenated together
  - ◇ Attribute node: the attribute value
- The string value of a node can be accessed using the `string` function
  - ◇ `string(doc("catalog.xml")/catalog/product[4]/number)` returns `"784"`
  - ◇ `string(<desc>Our <i>favorite</i> shirt!</desc>)` returns `"Our favorite shirt!"`
- Element and attribute nodes may also have a typed value, if it has been validated with a schema
- The typed value of a node can be accessed using the `data` function
  - ◇ `data(doc("catalog.xml")/catalog/product[4]/number)` returns the integer `784`, if the number element is declared in a schema to be an integer. Otherwise, its typed value is `784`, but it is considered to be **untyped**

Atomic Values

- A simple data value such as `784` or `ACC`, with no markup, and no association with any particular element or attribute
- An atomic value can have a specific type, such as `xs:integer` or `xs:string`, or it can be untyped (i.e. `xs:untypedAtomic`)
- Can be extracted from element or attribute nodes using the `string` and `data` functions
- Can also be created from literals in queries as in `@dept = 'ACC'`
- Functions and operators that expect atomic values as their operands also accept nodes, as in `doc("catalog.xml")//product[4]/substring(name, 1, 15)`
- Function `substring` expects a string atomic value as the first argument, but an element node (`name`) can be passed
- In this case, the atomic value is automatically extracted from the node in a process known as **atomization**
- Atomic values do not have identity

Sequences (1)

- Sequences are ordered collections of zero, one, or many items
- Each item in a sequence can be either an atomic value or a node
  - ◇ `doc("catalog.xml")/catalog/product` returns a sequence of four element nodes
- A sequence can also be created explicitly using a **sequence constructor**: a series of values, delimited by commas, surrounded by parentheses
  - ◇ `(1, 2, 3)` creates a sequence consisting of three atomic values
  - ◇ `(doc("catalog.xml")/catalog/product, 1, 2, 3)` uses an expression in the sequence constructor
- Sequences do not have names, they may be bound to a named variable
  - ◇ `let $prodList := doc("catalog.xml")/catalog/product` binds sequence of four `product` elements to the variable `$prodList`

Sequences (2)

- **Empty sequence**: A sequence with zero items
- The empty sequence is different from a zero-length string (`""`) or a zero value
- Many built-in functions and operations accept empty sequence as an argument
- Some expressions return the empty sequence, such as `doc("catalog.xml")//foo`, if there are no `foo` elements in the document
- Sequences **cannot be nested**: there is only one level of items
  - ◇ `(10, (20, 30), 40)` is equivalent to `(10, 20, 30, 40)`
- Many functions and operators in XQuery operate on sequences
  - ◇ Aggregation functions: `min`, `max`, `avg`, `sum`
  - ◇ `union`, `except`, and `intersect` expressions allow sequences to be combined
  - ◇ A number of functions operate generically on any sequence, such as `index-of` and `insert-before`
- Like atomic values, sequences have no identity

Types

- XQuery is a **strongly typed language**: functions and operators expect arguments or operands to be of a particular type
- XQuery type system is based on that of XML Schema
- XML Schema has built-in datatypes such as `xs:integer`, `xs:string`, and `xs:date`
- Types are assigned to items in the input document during schema validation
- If no schema is used, the items are untyped
- Untyped items are casted to the type required by a particular operation
- Casting converts a value from one type to another following specified rules
  - ◇ `doc("order.xml")/order/substring(@num, 1, 4)` does not require that the `num` attribute be declared to be of type `xs:string`
  - ◇ If it is untyped, it is cast to `xs:string`
  - ◇ If attribute is of type `xs:integer` it is necessary to convert the value of the attribute: `doc("order.xml")/order/substring(xs:string(@num), 1, 4)`

Namespaces (1)

- Namespaces are used to identify the vocabulary to which XML elements and attributes belong, and to disambiguate names from different vocabularies
- Input document (`prod_ns.xml`) with a namespace declaration

```
<prod:product xmlns:prod="http://datypic.com/prod">
  <prod:number>563</prod:number>
  <prod:name language="en">Floppy Sun Hat</prod:name>
</prod:product>
```

- The `prod` prefix is mapped to the namespace `http://datypic.com/prod`
- This means that any element or attribute name in the document that is prefixed with `prod` is in that namespace

Namespaces (2)

- Query

```
declare namespace prod = "http://datypic.com/prod";
for $product in doc("prod_ns.xml")/prod:product
return $product/prod:name
```

  Results

```
<prod:name xmlns:prod="http://datypic.com/prod"
  language="en">Floppy Sun Hat</prod:name>
```

- The namespace declaration in the first line of the query maps the namespace `http://datypic.com/prod` to the prefix `prod`
- The `prod` prefix is used in the body of the query to refer to elements in the input document
- The **namespace** (**not the prefix**) is a significant part of the name of an element or attribute
- The namespace URIs in the query and input document must match exactly
- Prefixes **do not have to be the same** in the input document and the query

XQuery: Contents

Categories of Expressions

Primary: literals, variables, function calls, and parenthesized expressions
Comparison based on value, node identity, or document order using `=`, `!=`, `<`, `<=`, `>`, `>=`, `eq`, `ne`, `lt`, `le`, `gt`, `ge`, `is`, `<<`, `>>`
Conditional: `if-then-else` expressions
Logical: Boolean operators using `or`, `and`
Path: Selecting nodes from XML documents using `/`, `//`, `..`, `.`, `child::`, ...
Constructor: Adding XML to the results using `<`, `>`, `element`, `attribute`
FLWOR: Select and process nodes using `for`, `let`, `where`, `order by`, `return`
Quantified: Test whether sequences fulfill conditions `some`, `every`, `in`, `satisfies`
Sequence-related: Create and combine sequences using `union`, `intersect`, `except`
Type-related: Cast and validate values based on type using `instance of`, `typeswitch`, `cast as`, `castable`, `treat`, `validate`
Arithmetic operations using `+`, `-`, `*`, `div`, `idiv`, `mod`

XQuery Elements (1)

- **Keywords** and **names** are case-sensitive
- **Names** must conform to the rules for XML qualified names: they start with a letter or underscore and contain letters, digits, underscores, dashes, and periods
- **No reserved words** in XQuery: a name (for example, a variable or function name) used in a query may be the same as any of the keywords
- Names are **namespace-qualified**: can be prefixed to associate them with a namespace name, may be affected by default namespace declarations
- **Whitespace** (spaces, tabs, and line breaks) is allowed almost anywhere in a query to break up expressions and make queries more readable

XQuery Elements (2)

- **Literals**: constant values represented in a query, such as `"ACC"` and `29.99`
- **String literals**: must be enclosed in single or double quotes
- **Numeric literals**: simple integers, such as `1`, decimal numbers, such as `1.5`, or floating-point numbers, such as `1.5E2`
- Type constructors can be used to convert literal values to the desired type: `xs:date("2006-05-03")` or Boolean values `true()` and `false()`
- **Variables**: identified by names that are preceded by `$`
- The names (not including the `$`) must be a XML qualified name
- When a query is evaluated, a variable is bound to a particular value
- Value may be any sequence, including a single node, a single atomic value, the empty sequence, or multiple nodes and/or atomic values
- Once the variable is bound to a value, **its value does not change**

XQuery Elements (2)

- Variables can be bound in global variable declarations, `for` or `let` clauses of a FLWOR, quantified expressions, or typeswitch expressions, as in
```
for $prod in doc("catalog.xml")/catalog/product
return $prod/number
```
- **Function declarations** also bind variables to values, as in
```
declare function local:addTwo ($value as xs:integer) as xs:integer
{ $value + 2 };
```
- **Function calls** as in `substring($prodName, 1, 5)`
- **XQuery comments** are delimited by `(:` and `:)`
- They are ignored during processing, can contain any text, as in
```
(: This query returns the <number> children :)
```
- XQuery comments can be nested within other XQuery comments
- **XML comments**, delimited by `<!--` and `-->`, can also be included in queries
- They appear in the result document, can include expressions that are evaluated $\Rightarrow$ useful debugging tool

XQuery Elements (3): General Comparisons

- Use the operators `=`, `!=`, `<`, `<=`, `>`, `>=`
- Can operate on sequences of more than one item, as well as empty sequences
- If either operand is the empty sequence, the expression evaluates to false
- Expression evaluates to true if the value comparison is true for any combination of two items from the two sequences
- `(2, 5) < (1, 3)` returns true since `2` is less than `3`; similarly `(2, 5) > (1, 3)`
- `doc("catalog.xml")/catalog/product/@dept = 'ACC'` is true if at least one of the four `dept` attributes is equal to `ACC`
- A type error is raised if the two operands contain any incomparable values, as in `(2, "a") = (1, "b")`

XQuery Elements (4): Value Comparisons

- Operate on **single atomic values** using operators `eq`, `ne`, `lt`, `le`, `gt`, and `ge`
- Operands: a single atomic value, a single node containing a single atomic value, or the empty sequence
  - ◇ `3 gt 4` is false, `"abc" lt "def"` is true, `<a>3</a> gt <z>2</z>` is true
- If either operand is the empty sequence, the empty sequence is returned
  ⇒ empty sequence behaves like null in SQL
- If either operand is a sequence of more than one item, a type error is raised, as in `doc("catalog.xml")/catalog/product/@dept eq 'ACC'`
- Two operands must have comparable types, otherwise explicitly casting is needed, as in `xs:integer("4") gt 3`
- Untyped values are always treated like strings by value comparisons
- Casting needed for comparing value of an untyped element to a numeric literal
- `doc("catalog.xml")/catalog/product[1]/number gt 1` raise a type error if the number element is untyped ⇒ comparing a string to a number

XQuery Elements (5): Node Comparisons

- `is` operator: determine whether two operands are actually the same node
- Each of the operands must be a single node, or the empty sequence
- If one of the operands is the empty sequence, the result is the empty sequence
- Compares the nodes based on their **identity** rather than their value
- `deep-equal` function: compare the contents and attributes of two nodes

XQuery Elements (6): Conditional Expressions

- **Syntax**: `if ( <expr> ) then <expr> else <expr>`
- Query
```
for $prod in (doc("catalog.xml")/catalog/product)
return if ($prod/@dept = 'ACC')
       then <accessoryNum>{data($prod/number)}</accessoryNum>
       else <otherNum>{data($prod/number)}</otherNum>
```
  Results
```
<otherNum>557</otherNum>
<accessoryNum>563</accessoryNum>
<accessoryNum>443</accessoryNum>
<otherNum>784</otherNum>
```
- The `else` part **is required**; it can be the empty sequence `()`
- If the `then` or the `else` return the results of multiple expressions, they need to be concatenated together using a sequence constructor

XQuery Elements (6): Conditional Expressions

- **Effective Boolean value** of the test expression is calculated
- False if evaluates to `false`, `0`, `""`, or `()`
- `if (doc("order.xml")//item) then "Item List: " else ""` returns string `Item List:` if there are any `item` elements in the order document
- Conditional expressions can be nested

```
for $prod in (doc("catalog.xml")/catalog/product)
return if ($prod/@dept = 'ACC')
  then <accessory>{data($prod/number)}</accessory>
  else if ($prod/@dept = 'WMN')
    then <womens>{data($prod/number)}</womens>
    else if ($prod/@dept = 'MEN')
      then <mens>{data($prod/number)}</mens>
      else <other>{data($prod/number)}</other>
```

Results

```
<womens>557</womens>
<accessory>563</accessory>
<accessory>443</accessory>
<mens>784</mens>
```

45

Logical Expressions (1)

- Combine Boolean values using the operators `and` and `or`
- Used in conditional expressions, `where` clauses of FLWORs, and path expression predicates
- Logical operators have lower precedence than comparison operators, `and` operator takes precedence over the `or` operator, ...
- Precedence can be changed with parentheses
  - ◇ `true() and true() or false() and false()` is true
  - ◇ `true() and (true() or false()) and false()` is false
- `not` function used for negating any Boolean value
- It is required to use parentheses around the value to be negated
- The function accepts a sequence of items, from which it calculates the effective Boolean value before negating it

46

Logical Expressions (2)

- Difference between the `!=` operator and calling the `not` function with an expression that uses the `=` operator
- `$prod/@dept != 'ACC'` returns
  - ◇ `true` if the `$prod` element has a `dept` attribute that is not equal to `ACC`
  - ◇ `false` if it has a `dept` attribute that is equal to `ACC`
  - ◇ `false` if it does not have a `dept` attribute
- `not($prod/@dept = 'ACC')` will return `true` in the third case

47

XQuery: Contents

48

Path Expressions

- Made up of one or more steps separated by `/` or `//`
- Return nodes in document order
- Evaluated relative to a particular **context item**
- Path expressions may start with a step that sets the context item, as in `doc("catalog.xml")/catalog/product/number`
- In `$catalog/product/number` the value of variable `$catalog` sets the context
- Context item changes with each step: a step returns a sequence of zero, one, or more nodes that serve as the context items for evaluating the next step
- A path expression can also be relative
- `product/number` evaluated relative to the current context node, which must have been previously determined outside the expression

Path Expressions: Axis (1)

- `self::` The context node itself
- `child::` Children of the context node
  - ◇ Attributes are not considered children of an element
  - ◇ This is the default axis if none is specified
- `descendant::` All descendants of the context node
- `descendant-or-self::` The context node and its descendants
- `attribute::` Attributes of the context node (if any)
- `following::` All nodes that follow the context node in the document, minus the context node's descendants

Path Expressions: Axis (2)

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

Node Tests

- Each axis step has a **node test** that indicates which of the nodes to select, along the specified axis
- Two kinds of node tests: by name or by node kind
- **Node name tests**: tests based on names, such as `product` and `dept`
- **Node kind tests**
  - ◇ `node()`: retrieve all different kinds of nodes (child element, text, comment, and processing-instruction nodes) but not attributes
    Different from `*`, which selects child element nodes only
  - ◇ `ancestor::node()` returns all ancestor element nodes and the document node (if it exists)
    Different from `ancestor::*`, which returns ancestor element nodes only
  - ◇ `attribute::node()` return attribute nodes (means the same as `@*`)
  - ◇ Four other kind tests: `text()`, `comment()`, `processing-instruction()`, and `documentnode()`

Node Name Tests and Namespaces

- Names used in node tests are **qualified names** (**QName**) ⇒ they are affected by namespace declarations
- A namespace declaration is in scope if it appears in an outer element, or in the query prolog
- If a name is prefixed, its prefix must be mapped to a namespace using a namespace declaration
- If an element name is unprefixed, and there is an in-scope default namespace declared, it is considered to be in that namespace; otherwise, it is in no namespace
- Attribute names are not affected by default namespace declarations

Node Name Tests and Namespaces: Example

- Input document (`prod_ns.xml`)

```
<prod:product xmlns:prod="http://datypic.com/prod">
  <prod:number>563</prod:number>
  <prod:name language="en">Floppy Sun Hat</prod:name>
</prod:product>
```

- Query

```
declare namespace prod = "http://datypic.com/prod";
<prod:prodList>{
  doc("prod_ns.xml")/prod:product/prod:number
}</prod:prodList>
```

Results

```
<prod:prodList xmlns:prod="http://datypic.com/prod">
  <prod:number>563</prod:number>
</prod:prodList>
```

Wildcards, Abbreviations

- **Wildcards** can be used to match names
- `child::*` (abbreviated `*`) selects all element children, regardless of name
- `attribute::*`, (or `@*`) selects all attributes, regardless of name
- Wildcards can be used for just the namespace and/or local part of a name
- `prod:*` selects all child elements in the namespace mapped to the prefix `prod`
- `*:product` selects all `product` child elements that are in any namespace, or no namespace
- Some axes and steps can be **abbreviated**

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

Expressions as Steps

- Expressions can be used as tests
  - ◇ `product/(number | name)`: all `number` and `name` elements
  - ◇ `product/(* except number)`: all children of `product` except `number`
  - ◇ `product/ (if (desc) then desc else name)`: for each `product` element, the `desc` child if it exists; otherwise, the `name` child
  - ◇ `product/substring(name,1,30)`: a sequence of `xs:string` values that are substrings of product names
- Only the last step in a path may return atomic values rather than nodes
- An error is raised if a step that is not the last returns atomic values, as in `product/substring(name,1,30)/replace(.,' ','-')`

Predicates in Path Expressions

- Used to filter the results to contain only nodes that meet specific criteria
- Select elements that have a certain value for an attribute or child element: `product[@dept = "ACC"]`, `product[number < 500]`
- Select elements that have a particular attribute or child element: `product[@dept]`, `product[color]`,
- Select elements that occur in a particular position within their parent: `product[3]`
- `product[number]` ≠ `product/number`
  - ◇ both expressions filter out products that have no `number` child
  - ◇ return the `product` or the `number` element, respectively
- Effective Boolean value of the expression is determined
- If it is true for a particular node, that node is returned, otherwise the node is not returned

57

Comparisons in Predicates

- Value comparison operators only allow a single value
- General comparison operators allow sequences of zero, one, or more values
- `//priceList[@effDate eq '2006-11-15']` is acceptable
  ⇒ each `priceList` element can have only one `effDate`
- `//priceList[prod/@num eq 557]` will raise an error
  ⇒ `prod/@num` returns more than one value per `priceList`
- `//priceList[prod/@num = 557]` returns a `priceList` if it has at least one `prod` child whose `num` attribute is equal to `557`
  - ◇ If a `priceList` does not have any `prod` children with `num` attributes, it does not return that `priceList`, but it does not raise an error
- Value comparison operators treat untyped data like strings
- `//priceList/prod[@num eq 557]` raises an error if no schema is present
- Using `=`, the value of the attribute will be casted to `xs:integer` and then compared to `557`

58

Positional Predicates

- Specify the position of an item within the sequence of items being processed
- Numbering starts at 1
- `doc("catalog.xml")/catalog/product[4]`: The fourth product in the catalog
- May be a predicate expression that evaluates to an integer
- If the number is greater than the number of items in the context sequence, it does not raise an error; it does not return any nodes
- `doc("catalog.xml")/catalog/product[99]` returns the empty sequence
- Specifies the position within the current sequence of items being processed, not the position of an element relative to its parent's children
- `doc("catalog.xml")/catalog/product/name[1]`: first `name` child of **each** product
- `(doc("catalog.xml")/catalog/product/name)[1]`: first `name` element in document

59

The `position` and `last` functions

- `position` function: Takes no argument, returns an integer representing the position (starting with 1, not 0) of the context item
- `doc("catalog.xml")/catalog/product[position() < 3]`: first two `product` children of `catalog`
- `doc("catalog.xml")/catalog/product/*[position() < 3]`: first two children of each `product`, with any name
- The first three products: `doc("catalog.xml")/catalog/product[1 to 3]` raises an error because predicate evaluates to multiple numbers instead of a single one
- ⇒ `doc("catalog.xml")/catalog/product[position() = (1 to 3)]`
- `subsequence` function can be used to limit the results based on position, as in `doc("catalog.xml")/catalog/subsequence(product, 1, 3)`
- `last` function: Takes no arguments, returns an integer representing the number of nodes in the current sequence
- `doc("catalog.xml")/catalog/product[last()]`: last `product` child of `catalog`

60

Complex Predicates (1)

- Multiple predicates can be chained together
- Second `product` child that has a `dept` attribute whose value is `ACC`
  `doc("catalog.xml")/catalog/product[@dept = "ACC"][2]`
- Second `product` child, if it has a `dept` attribute whose value is `ACC`
  `doc("catalog.xml")/catalog/product[2][@dept = "ACC"]`
- Predicates can contain function calls or conditional expressions
- All `product` children whose `dept` attribute contains the letter `A`
  `doc("catalog.xml")/catalog/product[contains(@dept, "A")]`
- Select `product` elements that have `desc` child if variable `$filter` is true
  `doc("catalog.xml")/catalog/product[if ($filter) then desc else true()]`
  If variable is false, all `product` elements are selected
- Can contain expressions that combine sequences
- All `product` children that have at least one child other than `number`
  `doc("catalog.xml")/catalog/product[* except number]`

Complex Predicates (2)

- General comparisons with multiple values can be used
- Products whose `dept` attribute value is any of the values `"ACC"`, `"WMN"`, `"MEN"`
  `doc("catalog.xml")/catalog/product[@dept = ("ACC", "WMN", "MEN")]`
- Every third `product` child of `catalog`
  `doc("catalog.xml")/catalog/product[position() mod 3 = 0]`
- Can contain path expressions that themselves have predicates
- `product` elements whose third child element is `colorChoices`
  `doc("catalog.xml")/catalog/product[*[3][self::colorChoices]]`
  `*[3]`: the third child element of `product`
  `[self::colorChoices]`: test the name of the current context element
- Predicates can be used with any sequence
  - ◇ `(1 to 100)[. mod 5 = 0]`: the integers from `1` to `100` that are divisible by `5`
  - ◇ `(@price, 0.0)[1]`: `price` attribute if it exists, value `0.0` otherwise

Dynamic Paths

- When the paths in a query are calculated based on some input to the query
- Example: to provide users with a search capability where they choose the elements in the input document to search
- XQuery **does not provide built-in support** for evaluating dynamic paths
- Simple dynamic paths: test for an element's name using the `name` function
- `doc("catalog.xml")//*[name() = $name][. = $value]`: name of element to search and its value bound to variables `$name` and `$value`
- More complex dynamic paths: implementation-specific functions
  - ◇ `saxon:evaluate` in Saxon, `util:eval` in eXist, `xdmp:eval` in Mark Logic
- In Saxon, the following expression get the same results as above

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

XQuery: Contents

Including Elements and Attributes in Results

- A query may return elements and attributes from the input document
- Query
```
for $prod in doc("catalog.xml")/catalog/product[@dept = 'ACC']
return $prod
```
Results
```
<product dept="ACC">
  <number>563</number>
  <name language="en">Floppy Sun Hat</name>
</product>
<product dept="ACC">
  <number>443</number>
  <name language="en">Deluxe Travel Bag</name>
</product>
```
- There are two ways to create **new elements and attributes**
- **Direct constructors**: use an XML-like syntax, create elements and attributes with fixed names
- **Computed constructors**: create names that are generated dynamically

Direct Element Constructors

- Specify XML elements and attributes using XML-like syntax
- Use **enclosed expresions**: expression between { and }
- Query
```
<html>
<h1>Product Catalog</h1>
<ul>{
  for $prod in doc("catalog.xml")/catalog/product
  return <li>number: {data($prod/number)}, name: {data($prod/name)}</li>
}</ul>
</html>
```
Results
```
<html>
  <h1>Product Catalog</h1>
  <ul>
    <li>number: 557, name: Fleece Pullover</li>
    <li>number: 563, name: Floppy Sun Hat</li>
    <li>number: 443, name: Deluxe Travel Bag</li>
    <li>number: 784, name: Cotton Dress Shirt</li>
  </ul>
</html>
```

Enclosed Expressions (1)

- May evaluate to elements
- Query
```
for $prod in doc("catalog.xml")/catalog/product
return <li>number: {$prod/number}</li>
```
Results
```
<li>number: <number>557</number></li>
<li>number: <number>563</number></li>
<li>number: <number>443</number></li>
<li>number: <number>784</number></li>
```
- May evaluate to attributes
- Query
```
for $prod in doc("catalog.xml")/catalog/product
return <li>{$prod/@dept}number: {$prod/number}</li>
```
Results
```
<li dept="WMN">number: <number>557</number></li>
<li dept="ACC">number: <number>563</number></li>
<li dept="ACC">number: <number>443</number></li>
<li dept="MEN">number: <number>784</number></li>
```

Enclosed Expressions (2)

- If expression evaluates to one or more atomic values, these values are cast to `xs:string` and included as character data content of the element
- Adjacent atomic values are separated by a space
- `<li>{"x", "y", "z"}</li>` returns `<li>x y z</li>` with spaces
- To avoid this, three separate expressions can be used: `<li>{"x"}{"y"}{"z"}</li>`
- May include more than one subexpression
- Query
```
for $prod in doc("catalog.xml")/catalog/product
return <li>{$prod/@dept,"string",5+3,$prod/number}</li>
```
Results
```
<li dept="WMN">string 8<number>557</number></li>
<li dept="ACC">string 8<number>563</number></li>
<li dept="ACC">string 8<number>443</number></li>
<li dept="MEN">string 8<number>784</number></li>
```

Specifying Attributes Directly

- Query
  ```
  <html>
  <h1 class="itemHdr">Product Catalog</h1>
  <ul>{
    for $prod in doc("catalog.xml")/catalog/product
    return <li dep="{$prod/@dept}">number: {data($prod/number)},
      name: {data($prod/name)}</li>
  }</ul>
  </html>
  ```
  Results
  ```
  <html>
    <h1 class="itemHdr">Product Catalog</h1>
    <ul>
      <li dep="WMN">number: 557, name: Fleece Pullover</li>
      <li dep="ACC">number: 563, name: Floppy Sun Hat</li>
      <li dep="ACC">number: 443, name: Deluxe Travel Bag</li>
      <li dep="MEN">number: 784, name: Cotton Dress Shirt</li>
    </ul>
  </html>
  ```
- If the `$prod` element has no `dept` attribute, its value will be a zero-length string
- In previous example `li` will not have a `dept` attribute in this case

69

Declaring Namespaces in Direct Constructors

- Query
  ```
  <xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
  <xhtml:h1 class="itemHdr">Product Catalog</xhtml:h1>
  <xhtml:ul>{
    for $prod in doc("catalog.xml")/catalog/product
    return <xhtml:li class="{$prod/@dept}">number: {
      data($prod/number)}</xhtml:li>
  }</xhtml:ul>
  </xhtml:html>
  ```
  Results
  ```
  <xhtml:html xmlns:xhtml="http://www.w3.org/1999/xhtml">
    <xhtml:h1 class="itemHdr">Product Catalog</xhtml:h1>
    <xhtml:ul>
      <xhtml:li class="WMN">number: 557</xhtml:li>
      <xhtml:li class="ACC">number: 563</xhtml:li>
      <xhtml:li class="ACC">number: 443</xhtml:li>
      <xhtml:li class="MEN">number: 784</xhtml:li>
    </xhtml:ul>
  </xhtml:html>
  ```

70

Modifying an Element from the Input Document

- Add to `product` elements an attribute `id` equal to `P` concatened with the product `number`
- Query
  ```
  for $prod in doc("catalog.xml")/catalog/product[@dept = 'ACC']
  return <product id="P{$prod/number}">
      {$prod/(@*, *)}
    </product>
  ```
  Results
  ```
  <product dept="ACC" id="P563">
    <number>563</number>
    <name language="en">Floppy Sun Hat</name>
  </product>
  <product dept="ACC" id="P443">
    <number>443</number>
    <name language="en">Deluxe Travel Bag</name>
  </product>
  ```
- `{$prod/(@*, *)}` copies all attributes and child elements from `product` element
- `{$prod/(@*, * except number)}` would copy all attributes and all child elements but `number`

71

Computed Element Constructors (1)

- Uses the keyword `element`, followed by a name and some content in curly braces
- Query
  ```
  element html {
    element h1 { "Product Catalog" },
      element ul {
        for $prod in doc("catalog.xml")/catalog/product
        return element li {"number:",data($prod/number),",
          name:",data($prod/name)}
      }
    }
  ```
  Results
  ```
  <html>
    <h1>Product Catalog</h1>
    <ul>
      <li>number: 557, name: Fleece Pullover</li>
      <li>number: 563, name: Floppy Sun Hat</li>
      <li>number: 443, name: Deluxe Travel Bag</li>
      <li>number: 784, name: Cotton Dress Shirt</li>
    </ul>
  </html>
  ```

72

Computed Element Constructors (2)

- **Name** can be an enclosed expression that evaluates to a qualified name as in `element {concat("h",$level)} { "Product Catalog" }`
- `element {node-name($myNode)} { "contents" }` will give the new element the same name as the node that is bound to the variable
- If expression is a node, it is atomized to extract its typed value (not its name)
- **Content**: enclosed expression for attributes, child elements, and character data
- `<li>number: {data($prod/number)}, name: {data($prod/name)}</li>` ≡ `element li {"number:",data($prod/number),", name:",data($prod/name)}`
- Values of four expressions above will be separated by spaces in the results
- `element li {concat("number:", data($prod/number), ", name:", data($prod/name))}` remove these spaces
- If the constructed element is to be empty, then put `{ }`

Computed Attribute Constructors

- Same syntax as a computed element constructor, but uses `attribute` keyword
- `attribute myattr { $prod/@dept }` is equivalent to `attribute {concat("my", "attr")} { $prod/@dept }`
- Computed attribute constructors can be used in direct element constructors
- `<result>{attribute {concat("my", "attr")} { "xyz" } }</result>` returns `<result myattr="xyz"/>`

Turning Content to Markup

- Create a product catalog that has the names of the departments as element names instead of attribute values
- Query
```
for $dept in distinct-values(doc("catalog.xml")/catalog/product/@dept)
return element {$dept}
  {doc("catalog.xml")/catalog/product[@dept = $dept]/name}
```
  Results
```
<WMN>
  <name language="en">Fleece Pullover</name>
</WMN>
<ACC>
  <name language="en">Floppy Sun Hat</name>
  <name language="en">Deluxe Travel Bag</name>
</ACC>
<MEN>
  <name language="en">Cotton Dress Shirt</name>
</MEN>
```

XQuery: Contents

FLWOR Expressions

- Path expressions used to select elements from input documents
- FLWOR expressions allow to join data from multiple sources, construct new elements and attributes, evaluate functions on intermediate values, sort results
- The following query
```
for $prod in doc("catalog.xml")//product
let $prodDept := $prod/@dept
where $prodDept = "ACC" or $prodDept = "WMN"
return $prod/name
```
  is equivalent to the path expression
```
doc("catalog.xml")//product[@dept = "ACC" or @dept = "WMN"]/name
```
- There can be multiple `for` and `let` clauses, in any order, followed by an optional `where` clause, an optional `order by` clause, and the required `return` clause
- A FLWOR must have at least one `for` or `let` clause
- FLWORs can appear in other expressions
```
max(for $prod in doc("catalog.xml")//product
  return xs:integer($prod/number))
```

The for Clause

- Sets up an iteration that evaluates the rest of the FLWOR multiple times, once for each item in the sequence returned by the expression after the `in` keyword
- This **binding sequence** can evaluate to any sequence of zero, one or more items
- Can be a sequence of elements, atomic values, nodes of any kind, or a mixture of items
- If the binding sequence is the empty sequence, the rest of the FLWOR is not evaluated (it iterates zero times)

Range Expressions

- Create a sequence of integers specifying the number of times to iterate
- Query
```
for $i in 1 to 3
return <oneEval>{$i}</oneEval>
```
  Results
```
<oneEval>1</oneEval>
<oneEval>2</oneEval>
<oneEval>3</oneEval>
```
- Can be included within parenthesized expressions, as in `(1 to 3, 6, 8 to 10)`
- Can use variables, as in `1 to $prodCount`
- If the first integer is greater than the second, as in `3 to 1`, or if either operand is the empty sequence, the expression evaluates to the empty sequence
- `for $i in 1 to count($seq)` works even if `$seq` is an empty sequence
- `for $i in reverse(1 to 3)` allows to descend in value
- `for $i in (1 to 100)[. mod 2 = 0]` gives 2, 4, 6, ... up to 100

Multiple for Clauses

- Similar to nested loops in a programming language
- The rest of the FLWOR is evaluated for every combination of the values of the variables
- Query
```
for $i in (1, 2)
for $j in ("a", "b")
return <oneEval>i is {$i} and j is {$j}</oneEval>
```
  Results
```
<oneEval>i is 1 and j is a</oneEval>
<oneEval>i is 1 and j is b</oneEval>
<oneEval>i is 2 and j is a</oneEval>
<oneEval>i is 2 and j is b</oneEval>
```
- Multiple variables can be bound in a single for clause, separated by commas
- Query above is equivalent to
```
for $i in (1, 2), $j in ("a", "b")
return <oneEval>i is {$i} and j is {$j}</oneEval>
```

The let Clause

- Binds a variable to a value
- **Does not result in iteration**: It binds the whole sequence to the variable
- Query
  ```
  let $i := (1 to 3)
  return <oneEval>{$i}</oneEval>
  ```
  Result
  ```
  <oneEval>1 2 3<oneEval>
  ```
- `let` and `for` clauses can be mixed
- All must appear before any `where`, `order by`, or `return` clauses of that FLWOR
  ```
  let $doc := doc("catalog.xml")
  for $prod in $doc//product
  let $prodDept := $prod/@dept
  let $prodName := $prod/name
  where $prodDept = "ACC" or $prodDept = "WMN"
  return $prodName
  ```
- Short syntax: `let $prodDept := $prod/@dept, $prodName := $prod/name`

The where Clause

- Specify criteria that filter the results of the FLWOR expression
  ```
  where $prodDept = "ACC" or $prodDept = "WMN"
  ```
- In addition to expressing complex filters, is also useful for joins
- A where clause with multiple expressions
  ```
  for $prod in doc("catalog.xml")//product
  let $prodDept := $prod/@dept
  where $prod/number > 100 and starts-with($prod/name, "F")
  and exists($prod/colorChoices)
  and ($prodDept = "ACC" or $prodDept = "WMN")
  return $prod
  ```
- The effective Boolean value of the `where` expression is calculated
- `where $prod/name` returns true if `$prod` has a `name` child, false otherwise
- `where $numProds` returns true if `$numProds` is a numeric value that is not zero and not `NaN` (not a number)
- However, preferable to use `where exists($prod/name) and $numProds > 0`

The return Clause

- Evaluated once for each iteration, provided `where` expression evaluated to true
- Result value of the entire FLWOR: sequence of items returned by each evaluation of the `return` clause
- The value of
  ```
  for $i in (1 to 3)
  return <oneEval>{$i}</oneEval>
  ```
  is a sequence of three `oneEval` elements
- Expression to be included in the return clause can be combined in a sequence
- Query
  ```
  for $i in (1 to 3)
  return (<one>{$i}</one>, <two>{$i}</two>)
  ```
  returns a sequence of six elements, two for each evaluation of the `return` clause
- If no parentheses or comma were used, the `two` element constructors would not be considered part of the FLWOR

Quantified Expressions (1)

- Determines whether some or all of the items in a sequence meet a condition
- To know whether **any** of the items in an order are from the accessory department
  ```
  some $dept in doc("catalog.xml")//product/@dept
  satisfies ($dept = "ACC")
  ```
- To know if **every** item in an order is from the accessory department
  ```
  every $dept in doc("catalog.xml")//product/@dept
  satisfies ($dept = "ACC")
  ```
- A quantified expression evaluates to a Boolean value (true or false)
- The processor tests the `satisfies` expression (using its effective Boolean value) for every item in the sequence
- `some`: returns true if the `satisfies` expression is true for any of the items
- `every`: returns true if the `satisfies` expression is true for all items
- If there are no items in the sequence, an expression with `some` returns false, an expression with `every` returns true

Quantified Expressions (2)

- Combined with `not` function allows to express “not any” (none) and “not every”
- Expression
  ```
  not(some $dept in doc("catalog.xml")//product/@dept
  satisfies ($dept = "ACC"))
  ```
  returns true if none of the `product` elements have a dept attribute equal to `ACC`
- Multiple variables can be bound in a quantified expression by separating the clauses with commas
- Every combination of the items in the sequences is taken
- Expression
  ```
  some $i in (1 to 3), $j in (10, 11)
  satisfies $j - $i = 7
  ```
  returns true since `$i=3` and `$j=10` makes the `satisfies` expression true

Selecting Distinct Values

- `distinct-values` function selects distinct atomic values from a sequence
- `distinct-values(doc("catalog.xml")//product/@dept)` returns all the distinct values of the `dept` attribute, namely `("WMN", "ACC", "MEN")`
- Function determines whether two values are equal using the `eq` operator
- It is also common to select a distinct set of combinations of values
- Select all the distinct department/product number combinations
  ```
  let $prods := doc("catalog.xml")//product
  for $d in distinct-values($prods/@dept),
    $n in distinct-values($prods[@dept = $d]/number)
  return <result dept="{$d}" number="{$n}"/>
  ```
  Results
  ```
  <result dept="WMN" number="557"/>
  <result dept="ACC" number="563"/>
  <result dept="ACC" number="443"/>
  <result dept="MEN" number="784"/>
  ```

Joins (1)

- FLWORs allow to easily join data from multiple sources
- Join information from product catalog (`catalog.xml`) and order (`order.xml`): list all items in the order, along with their number, name, and quantity
  ```
  for $item in doc("order.xml")//item,
      $product in doc("catalog.xml")//product[number = $item/@num]
  return <item num="{$item/@num}"
      name="{$product/name}" quan="{$item/@quantity}"/>
  ```
  Results
  ```
  <item num="557" name="Fleece Pullover" quan="1"/>
  <item num="563" name="Floppy Sun Hat" quan="1"/>
  <item num="443" name="Deluxe Travel Bag" quan="2"/>
  <item num="784" name="Cotton Dress Shirt" quan="1"/>
  <item num="784" name="Cotton Dress Shirt" quan="1"/>
  <item num="557" name="Fleece Pullover" quan="1"/>
  ```
- Same two-way join in a `where` clause
  ```
  for $item in doc("order.xml")//item,
      $product in doc("catalog.xml")//product
  where $item/@num = $product/number
  return <item num="{$item/@num}"
      name="{$product/name}" quan="{$item/@quantity}"/>
  ```

Joins (2)

- More than two sources can be joined together
- Three-way join in a `where` clause
  ```
  for $item in doc("order.xml")//item,
      $product in doc("catalog.xml")//product,
      $price in doc("prices.xml")//prices/priceList/prod
  where $item/@num = $product/number and $product/number = $price/@num
  return <item num="{$item/@num}"
      name="{$product/name}" price="{$price/price}"/>
  ```
  Results
  ```
  <item num="557" name="Fleece Pullover" price="29.99"/>
  <item num="563" name="Floppy Sun Hat" price="69.99"/>
  <item num="443" name="Deluxe Travel Bag" price="39.99"/>
  <item num="557" name="Fleece Pullover" price="29.99"/>
  ```
- The `where` clauses use the `=` operator to determine whether two values are equal
- If schemas are not used, both values are untyped $\Rightarrow$ compared as strings
- Unless they are cast to numeric types, join does not consider different representations of the same number equal, e.g., `0557` and `557`

Outer Joins

- Previous join examples are **inner joins**: results do not include items without matching products or products without matching items
- Create a list of products and join it with the price information: Even if there is no price, include the product in the list

```
for $product in doc("catalog.xml")//product
return <product number="{$product/number}">{
  attribute price
    {for $price in doc("prices.xml")//prices/priceList/prod
    where $product/number = $price/@num
    return $price/price}
}</product>
```

  Results

```
<product number="557" price="29.99"/>
<product number="563" price="69.99"/>
<product number="443" price="39.99"/>
<product number="784" price=""/>
```

- This is known in relational databases as an **outer join**
- Product `784` doesn't have a price in `prices.xml` ⇒ `price` attribute has an empty value

XQuery: Contents

- Introduction to XQuery
- XQuery Foundations
- XQuery Expressions
- Navigating Input Documents Using Paths
- Adding Elements and Attributes to Results
- Selecting and Joining Using FLWORs
- ➥ Sorting and Grouping
- Functions
- Updates

Sorting in XQuery

- Path expressions return items in document order
- By default, results of FLWORs are based on the order of sequence of the `for` clause
- `order by` clause is used to sort data in an order other than document order

```
for $item in doc("order.xml")//item
order by $item/@num
return $item
```

- Several ordering specifications can be used: `order by $item/@dept, $item/@num`
- Expressions in ordering specifications can only return one value for each item being sorted
- Unlike SQL, allowed to order by a value that is not returned by the expression: can order by `$item/@dept` and only return `$item/@num` in the results
- Untyped values are treated like strings
- `number` function can be used if untyped values must be treated as numeric for sorting purposes: `order by number($item/@num)`

Order Modifiers

- Several order modifiers can optionally be specified for each ordering specification
- `ascending` and `descending` specify sort direction; default is `ascending`
- Order modifiers apply to only one order specification, as in `order by $item/@dept, $item/@num descending`
- `empty greatest` and `empty least` indicate whether the empty sequence and `NaN` should be considered a low value or a high value
- `empty greatest`: empty sequence is greater than `NaN`, `NaN` is greater than all other values
- `empty least`: empty sequence is less than `NaN`, `NaN` is less than all other values
- This applies to the empty sequence and `NaN` only, not to zero-length strings
- `collation`, followed by a collation URI in quotes, specifies a collation used to determine the sort order of strings

Empty Order: Example

- Default behavior for all order by clauses can be specified in the query prolog, as in declare default order empty greatest;
- A query that uses an empty order declaration and sorts the results by the color attributes

```
declare default order empty greatest;
for $item in doc("order.xml")//item
order by $item/@color
return $item
```

Results

```
<item dept="WMN" num="557" quantity="1" color="black"/>
<item dept="MEN" num="784" quantity="1" color="gray"/>
<item dept="WMN" num="557" quantity="1" color="navy"/>
<item dept="MEN" num="784" quantity="1" color="white"/>
<item dept="ACC" num="563" quantity="1"/>
<item dept="ACC" num="443" quantity="2"/>
```

- Because the greatest option is chosen, the items with no color attribute appear last in the results

93

Stable Ordering, Complex Ordering

- When sorting, several values may be returned that have the same sort value
- Implementations may return values that have equal sort values in any order
- **Stable ordering**: equal sort values sorted in the order of the input sequence
- stable order by $item/@num: items with the same num value are always returned in the order returned by the for expression, within the sorted results
- Sort can be based on any expression, provided it returns a single item
- order by substring($item/@dept, 2, 2): sorts on a substring of department
- order by (if ($item/@color) then $item/@color else "unknown"): sorts on the color if it exists or the string "unknown" if it does not
- A path expression that refers to a different XML document can be used
- order by doc("catalog.xml")//product[number = $item/@num]/name: orders the results based on a name it looks up in the catalog.xml document
- order by $item/@*[name( )=$param]: allows to decide at runtime what sort key to use

94

Inadvertent Resorting in Document Order

- Some expressions, including path expressions and operators that combine sequences (|, union, intersect, and except), return nodes in document order
- Results may be resorted to document order when using a FLWOR in another expression
- Sorts products by product number, then returns their names in li elements

```
let $sortedProds := for $prod in doc("catalog.xml")//product
    order by $prod/number
    return $prod
for $prodName in $sortedProds/name
return <li>{string($prodName)}</li>
```

- Query returns the products in document order, not product number, due to the expression $sortedProds/name
- The expression can be rewritten as follows

```
for $prod in doc("catalog.xml")//product
order by $prod/number
return <li>{string($prod/name)}</li>
```

95

Order Comparisons, Reverse Ordering

- << and >>: node comparison based on their relative position in document order
- For example, $n1 << $n2 returns true if $n1 precedes $n2 in document order
- According to the definition of document order, a parent precedes its children
- For each product, checks whether there are any other products later in the document that are in the same department. If so, returns the product element

```
let $prods := doc("catalog.xml")//product
for $prod in $prods
where $prod << $prods[@dept = $prod/@dept][last( )]
return $prod
```

- reverse function reverses the order of items in a sequence
- reverse(doc("catalog.xml")//product) returns the product elements in reverse document order
- reverse((6, 2, 3)) returns the sequence (3, 2, 6)

96

Indicating That Order Is Not Significant

- Processor may be much more efficient if it does not have to keep track of order
- Especially true for FLWORs that perform joins
- `unordered` function: returns items passed as argument in undetermined order
```
unordered(
  for $item in doc("order.xml")//item,
    $product in doc("catalog.xml")//product
  where $item/@num = $product/number
  return <item number="{$item/@num}"
    name="{$product/name}" quantity="{$item/@quantity}"/>
)
```
- `unordered` expression: affects also every embedded expression
```
unordered { ... }
```
- **Ordering mode declaration**: specify whether order is significant for a query
```
declare ordering unordered;
```
- If no ordering mode declaration is present, the default is ordered
- `ordered` expression: specifies that order matters in a certain section of query
- Unnecessary except to override an ordering mode declaration

Achieving Grouping

- XQuery 1.0 **does not provide** a `group by` clause
- List of items to be grouped by department
```
for $d in distinct-values(doc("order.xml")//item/@dept)
let $items := doc("order.xml")//item[@dept = $d]
order by $d
return <department code="{$d}">{
    for $i in $items
    order by $i/@num
    return $i
}</department>
```

Results
```
<department code="ACC">
    <item dept="ACC" num="443" quantity="2"/>
    <item dept="ACC" num="563" quantity="1"/>
</department>
<department code="MEN">
    <item dept="MEN" num="784" quantity="1" color="white"/>
    <item dept="MEN" num="784" quantity="1" color="gray"/>
</department>
```

Grouping with the group by Clause

- A `group by` clause was introduced in XQuery 1.1
- List of items to be grouped by department
```
for $item in doc("order.xml")//item
group by $item/@dept
order by $item/@dept
return <department code="{$item/@dept}">{
    for $i in $items
    order by $i/@num
    return $i
}</department>
```
- **Pre-grouping** tuples replaced by new **post-grouping** tuples that represent groups
- Post-grouping tuples have same variable-names as pre-grouping ones
- Each pre-grouping tuple is assigned to a group, one post-grouping tuple is generated for each group
- Pre-grouping tuples are no longer accessible after group by clause
- Subsequent clauses in the FLWOR expression operate on post-grouping tuples

Aggregating Values

- Determine the number of item elements in a department, and the sum of the quantities for a department
```
for $d in distinct-values(doc("order.xml")//item/@dept)
let $items := doc("order.xml")//item[@dept = $d]
order by $d
return <department code="{$d}" numItems="{count($items)}"
  distinctItemNums="{count(distinct-values($items/@num))}"
  totQuant="{sum($items/@quantity)}"/>
```

Results
```
<department code="ACC" numItems="2" distinctItemNums="2" totQuant="3"/>
<department code="MEN" numItems="2" distinctItemNums="1" totQuant="2"/>
<department code="WMN" numItems="2" distinctItemNums="1" totQuant="2"/>
```

Aggregating Values with group by Clause

- Determine the number of item elements in a department, and the sum of the quantities for a department

```
for $items := doc("order.xml")//item
group by $items/@dept
order by $items/@dept
return <department code="{$items/@dept}" numItems="{count($items)}"
  distinctItemNums="{count(distinct-values($items/@num))}"
  totQuant="{sum($items/@quantity)}"/>
```

Results

```
<department code="ACC" numItems="2" distinctItemNums="2" totQuant="3"/>
<department code="MEN" numItems="2" distinctItemNums="1" totQuant="2"/>
<department code="WMN" numItems="2" distinctItemNums="1" totQuant="2"/>
```

101

Aggregation Functions

- `count`: number of items in a sequence
- `sum`: total value of the items in a sequence
- `min` and `max`: minimum and maximum values of the items in a sequence
- `avg`: average value of the items in a sequence
- `sum` and `avg` accept values that are all numeric, all `xs:yearMonthDuration` values, or all `xs:dayTimeDuration` values
- `max` and `min` accept values of any type that is ordered (i.e., values can be compared using `<` and `>`): strings, dates, . . .
- `sum`, `min`, `max`, and `avg` treat untyped data as numeric
- If a schema is not used, to find a maximum string value

```
max(doc("order.xml")//item/string(@dept))
```

102

"Missing" Values

- Sequence passed to an aggregation function may contain zero-length strings
- If an item has `color=""`, `min(doc("order.xml")//item/string(@color))` will return `""`
- `avg` function ignores absent nodes
- `avg(doc("prices.xml")//discount)` gets the average of the two discount values
- To count the discount as zero for the product with no discount child

```
sum(doc("prices.xml")//prod/discount) div count(doc("prices.xml")//prod)
```

- An empty discount child (i.e., `<discount></discount>`), would be considered a zero-length string ⇒ `avg` function would raise an error
- `avg(doc("prices.xml")//prod/discount[. != ""])` tests for missing values

103

Aggregating on Multiple Values

- Determine the number of items and total quantity for each department/product number combination

```
let $allItems := doc("order.xml")//item
for $d in distinct-values($allItems/@dept)
for $n in distinct-values($allItems[@dept = $d]/@num)
let $items := $allItems[@dept = $d and @num = $n]
order by $d, $n
return <group dept="{$d}" num="{$n}"
    numItems="{count($items)}" totQuant="{sum($items/@quantity)}"/>
```

Results

```
<group dept="ACC" num="443" numItems="1" totQuant="2"/>
<group dept="ACC" num="563" numItems="1" totQuant="1"/>
<group dept="MEN" num="784" numItems="2" totQuant="2"/>
<group dept="WMN" num="557" numItems="2" totQuant="2"/>
```

104

Aggregating on Multiple Values with group by Clause

- Determine the number of items and total quantity for each department/product number combination

```
for $items := doc("order.xml")//item
group by $items/@dept, $items/@num
order by $items/@dept, $items/@num
return <group dept="{$items/@dept}" num="{$items/@num}"
    numItems="{count($items)}" totQuant="{sum($items/@quantity)}"/>
```

Results

```
<group dept="ACC" num="443" numItems="1" totQuant="2"/>
<group dept="ACC" num="563" numItems="1" totQuant="1"/>
<group dept="MEN" num="784" numItems="2" totQuant="2"/>
<group dept="WMN" num="557" numItems="2" totQuant="2"/>
```

Constraining and Sorting on Aggregated Values

- Similar to previous query, but display only groups whose total quantity (`totQuant`) is greater than 1, and sort results by the number of items (`numItems`)

```
let $allItems := doc("order.xml")//item
for $d in distinct-values($allItems/@dept)
for $n in distinct-values($allItems/@num)
let $items := $allItems[@dept = $d and @num = $n]
where sum($items/@quantity) > 1
order by count($items)
return if (exists($items))
      then <group dept="{$d}" num="{$n}" numItems="{count($items)}"
      totQuant="{sum($items/@quantity)}"/>
else ( )
```

Results

```
<group dept="ACC" num="443" numItems="1" totQuant="2"/>
<group dept="WMN" num="557" numItems="2" totQuant="2"/>
```

Constraining and Sorting on Aggregated Values with group by Clause

- Similar to previous query, but display only groups whose total quantity (`totQuant`) is greater than 1, and sort results by the number of items (`numItems`)

```
for $items := doc("order.xml")//item
group by $items/@dept, $items/@num
where sum($items/@quantity) > 1
order by count($items)
return if (exists($items))
      then <group dept="{$d}" num="{$n}" numItems="{count($items)}"
      totQuant="{sum($items/@quantity)}"/>
else ( )
```

Results

```
<group dept="ACC" num="443" numItems="1" totQuant="2"/>
<group dept="WMN" num="557" numItems="2" totQuant="2"/>
```

XQuery: Contents

Functions

- XQuery has a standard set of built-in functions
- Some string functions
  - ◇ `string(arg)`: Returns the string value of the argument
  - ◇ `concat(string,string,...)`: Returns the concatenation of the strings
  - ◇ `string-join((string,string,...),sep)`: Returns a string created by concatenating the string arguments and using the sep argument as the separator
  - ◇ `substring(string,start,len)`, `substring(string,start)`: Returns the substring from the start position to the specified length. Index of the first character is 1. If length is omitted it returns the substring from the start position to the end
  - ◇ ...
- Function calls can be included anywhere an expression is permitted, as in
  - ◇ `let $name := substring($prodName,1,5)`
  - ◇ `<name>{substring($prodName,1,5)}</name>`
  - ◇ `doc("catalog.xml")/catalog/product[substring(name,1,5) = 'Short']`

109

Function Names

- Functions have namespace-qualified names
- **Built-in functions**: in namespace `http://www.w3.org/2005/xpath-functions`
  ⇒ `fn` prefix for these functions is unnecessary
- **User-defined functions** must be called by its prefixed name
- Functions declared in the same query module can be called using the same prefixed name found in the declaration
- `local`: built-in prefix for locally declared functions

```
declare function local:return2 ( ) as xs:integer {2};
<size>{local:return2( )}</size>
```

- Functions in a separate query module have a different namespace that needs to be declared
- Call a function `discountPrice` in the namespace `http://datypic.com/prod`:

```
import module namespace prod = "http://datypic.com/prod"
    at "http://datypic.com/prod/module.xq";
<price>{prod:discountPrice( )}</price>
```

110

Function Signatures

- Describe the inputs and outputs of a function
- Signature of the built-in `upper-case` function

```
upper-case($arg as xs:string?) as xs:string
```

- Signature indicates
  - ◇ Name of the function: `upper-case`
  - ◇ List of parameters: only one, whose name is `$arg` and whose type is `xs:string?`
    `?`: function accepts a single `xs:string` value or the empty sequence
  - ◇ Return type of the function: `xs:string`
- There may be several signatures associated with the same function name, with a different number of parameters
- Signatures for the `substring` function

```
substring($sourceString as xs:string?, $startingLoc as xs:double)
    as xs:string
substring($sourceString as xs:string?, $startingLoc as xs:double,
    $length as xs:double) as xs:string
```

111

Argument Lists

- When calling a function, there must be an argument for every parameter specified in the function signature
- If there is more than one signature, the argument list may match either function signature
- If the function does not take any arguments, the parentheses are still required, as in `current-date( )`
- A function call may have complex, nested expressions, as in

```
concat(substring($name,1,$sublen), (if ($addT) then "T" else ""))
```

- Calling a function never changes the value of the variables that are passed to it
- Passing the empty sequence or a zero-length string for an argument is not the same as omitting an argument
- `substring($myString, 2)` is not the same as `substring($myString, 2, ( ))`
- `upper-case` function expects one argument, which can be the empty sequence
  ⇒ `upper-case( )` is not acceptable, while `upper-case(( ))` is

112

Arguments: Lists vs Sequences

- Each expression in argument list is considered a single argument
- A sequence passed to a function is considered a single argument
- Some functions expect sequences as arguments
- function `max($arg as xs:anyAtomicType*) as xs:anyAtomicType?` expects one argument that is a sequence
  ⇒ `max ((1, 2, 3))` is appropriate but not `max (1, 2, 3)`
- `substring( ($myString, 2) )` has only one argument (a sequence of two items)
  ⇒ raises an error because the function expects two (or three) arguments
- Apply a function to each element of a sequence: the substring of each of the product names
  - ◇ `substring( doc("catalog.xml")//name, 1, 3)` won't work ⇒ first argument of substring is not allowed to contain more than one item
  - ◇ `doc("catalog.xml")//name/substring( ., 1, 3 )` returns a sequence of four strings

Sequence Types (1)

- Types of parameters are expressed as sequence types, which specify the number and type (and/or node kind) of items that make up the parameter
- Most commonly used sequence types: atomic types such as `xs:integer`, `xs:double`, `xs:date`, or `xs:string`
- Sequence type `xs:anyAtomicType` matches any atomic value
- Some built-in functions also use `numeric` to allow values of any numeric type
- Occurrence indicators indicate how many items can be in a sequence
  - ◇ `?` For zero or one items
  - ◇ `*` For zero, one, or many items
  - ◇ `+` For one or many items
- If no occurrence indicator is specified, this means one and only one
- There is no difference between an item and a sequence containing only that item
- If a function expects `xs:string*` it is acceptable to pass it a string such as `"xyz"`

Sequence Types (2)

- In a function call, the type of an argument may differ from the type specified in the function signature
- An `xs:integer` can be passed to a function that expects an `xs:decimal`
- An element containing a string can be passed to a function that expects a string
- XQuery defines **function conversion rules** for converting arguments to the expected type
- Not all arguments can be converted using the function conversion rules, because function conversion does not involve straight casting from one type to another
- A string cannot be passed to a function that expects an integer
- If an argument does not match the sequence type specified in the function signature, a type error is raised

User-Defined Functions

- Functions are defined using function declarations, which can appear either in the query prolog or in an external library

```
declare function local:discountPrice(
     $price as xs:decimal?,
     $discount as xs:decimal?,
     $maxDiscountPct as xs:integer?) as xs:decimal?
{
     let $maxDiscount := ($price * $maxDiscountPct) div 100
     let $actualDiscount := min(($maxDiscount, $discount))
     return ($price - $actualDiscount)
};
local:discountPrice($prod/price, $prod/discount, 15)
```

- A function declaration consists of several parts
  - ◇ The keywords `declare function` followed by the qualified function name
  - ◇ A list of parameters enclosed in parentheses and separated by commas
  - ◇ The return type of the function
  - ◇ A function body enclosed in curly braces and followed by a semicolon

The Function Body

- An expression enclosed in curly braces, which may contain any valid XQuery expressions, including FLWORs, path expressions, etc.
- A return clause is not mandatory: return value is the value of the expression

  ```
  declare function local:get-pi( ) {3.141592653589};
  ```
- Can call other functions declared in the module or in an imported module
- Once the function body has been evaluated, its value is converted to the return type using function conversion rules
- If the return type is not specified, it is assumed to be `item*`, that is, a possibly empty sequence of atomic values and nodes

The Function Name

- Functions are uniquely identified by their qualified name and their number of parameters
- There can be more than one function declaration that has the same qualified name, as long as the number of parameters is different
- Function name must be a valid XML name: it can start with a letter or underscore and contain letters, digits, underscores, dashes, and periods
- Like other XML names, function names are case-sensitive
- All user-defined function names must be in a namespace
- In the main query module, any prefix that is declared in the prolog can be used
- Predefined prefix `local` puts the function in the namespace `http://www.w3.org/2005/xquery-local-functions`
- It can then be called from within that main module using the prefix `local`
- If a function is declared in a library module, its name must be in the target namespace of the module

The Parameter List

- Each parameter has a unique name, and optionally a type
- Name is expressed as a variable name, preceded by `$`
- When a function is called, the variable specified is bound to the value that is passed to it
- If no type is specified for a particular parameter, it allows any argument
- However, it is best to specify a type, for error checking and clarity
- When the function is called, each argument value is converted to the appropriate type according to the function conversion rules

Arguments: Nodes versus Atomic Values

- Accept a node that contains an atomic value, or accept the atomic value itself?
- In the declaration of `local:discountPrice`, the `price` and `discountPct` element could be accepted instead of accepting their `xs:decimal` and `xs:integer` values
- In some cases it is advantageous to pass the entire element as an argument
  - ◇ Need to access its attributes, e.g., `currency` attribute of price
  - ◇ Need to access its parent or siblings
- It is generally better to accept the atomic value
  - ◇ It is more flexible: a node can be passed to a function that expects an atomic value, but an atomic value cannot be passed to a function that expects a node
  - ◇ Can be more specific about the desired type of the value, e.g., to ensure that it is an `xs:integer`
  - ◇ Don't have to cast untyped values to the desired type; this will happen automatically

Accepting Arguments that are the Empty Sequence

- Given previous `local:discountPrice` function, suppose `$prod` has no `discount` child ⇒ `$discount` is bound to the empty sequence
- The function returns the empty sequence ⇒ all arithmetic operations on the empty sequence return the empty sequence
- The function should return the original price if no discount amount is provided

```
declare function local:discountPrice(
    $price as xs:decimal?, $discount as xs:decimal?,
    $maxDiscountPct as xs:integer?) as xs:double?
{
    let $newDiscount := if ($discount) then $discount else 0
    let $maxDiscount := if ($maxDiscountPct)
        then ($price * $maxDiscountPct) div 100
        else  0
    let $actualDiscount := min(($maxDiscount, $discount))
    return ($price - $actualDiscount)
};
local:discountPrice($prod/price, $prod/discount, 15)
```

121

Recursive Functions

- Functions can recursively call themselves
- Count the number of descendant elements of an element, not just the immediate children, but all the descendants

```
declare namespace functx = "http://www.functx.com";
declare function functx:num-descendant-elements
    ($el as element( )) as xs:integer {
        sum(for $child in $el/*
            return functx:num-descendant-elements($child) + 1)
};
```

- There must be a level at which the function stops calling itself
- In this case, it will eventually reach an element that has no children: the for clause will not be evaluated returning an empty sequence, and `sum(())` yields 0
- Following function

```
declare function local:addItUp () { 1 + local:addItUp( ) };
```

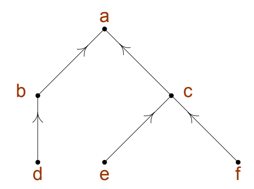
results in an infinite loop, which will possibly end with an “out of memory” or “stack full” error

122

Transitive Closure (1)

- `employees.xml`: Supervision relationship between employees

```
<employees>
  <employee name="a"/>
  <employee name="b">
    <super>a</super>
  </employee>
  <employee name="c">
    <super>a</super>
  </employee>
  <employee name="d">
    <super>b</super>
  </employee>
  <employee name="e">
    <super>c</super>
  </employee>
  <employee name="f">
    <super>c</super>
  </employee>
</employees>
```



123

Transitive Closure (2)

- Give the direct or indirect subordinates of all employees
- Recursive function

```
declare function local:subordinates($emp as element())
    as element()* {
  let $direct :=
    doc('employees.xml')/employees/employee[super=$emp/@name]
  return $direct | $direct/local:subordinates(.)
};
```

- Query

```
for $emp in doc('employees.xml')/employees/employee
return
  <employee>{
    $emp/@name,
    for $sub in local:subordinates($emp)
    return
      <subordinate>{data($sub/@name)}</subordinate>
    }
  </employee>
```

124

Transitive Closure (3)

- Results

```
<employee name="a">
  <subordinate>b</subordinate>
  <subordinate>c</subordinate>
  <subordinate>d</subordinate>
  <subordinate>e</subordinate>
  <subordinate>f</subordinate>
</employee>
<employee name="b">
  <subordinate>d</subordinate>
</employee>
<employee name="c">
  <subordinate>e</subordinate>
  <subordinate>f</subordinate>
</employee>
<employee name="d"/>
<employee name="e"/>
<employee name="f"/>
```

Transitive Closure (4)

- If there is a cycle in the data, the above function will cause an infinite recursion
- To avoid this, an extra parameter is needed containing the list of subordinates found so far

```
declare function local:subordinates($emp as element(),
    $sofar as element()*) as element()* {
  let $direct :=
    doc('employees1.xml')/employees/employee[super=$emp/@name]
  return
  if ($direct intersect $sofar) then $direct intersect $sofar
    else $direct | $direct/local:subordinates(.,$sofar|.)
};
```

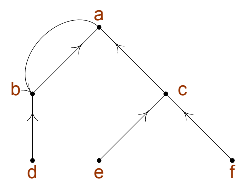
- Similar query as before

```
for $emp in doc('employees.xml')/employees/employee
return
  <employee>{
    $emp/@name,
    for $sub in local:subordinates($emp,$emp)
    return
      <subordinate>{data($sub/@name)}</subordinate>
    }
```

Transitive Closure (5)

- Results for supervision relationship with a cycle

```
<employee name="a">
  <subordinate>a</subordinate>
  <subordinate>b</subordinate>
  <subordinate>c</subordinate>
  <subordinate>e</subordinate>
  <subordinate>f</subordinate>
</employee>
<employee name="b">
  <subordinate>a</subordinate>
  <subordinate>b</subordinate>
  <subordinate>d</subordinate>
</employee>
<employee name="c">
  <subordinate>e</subordinate>
  <subordinate>f</subordinate>
</employee>
<employee name="d"/>
<employee name="e"/>
<employee name="f"/>
```



XQuery: Contents

Updates

- XQuery Update Facility 1.0: W3C Candidate Recommendation on 14 March 2008
- Provides facilities to perform the following operations
  - ◇ Insertion of a node
  - ◇ Deletion of a node
  - ◇ Modification of a node by changing some of its properties while preserving its node identity
  - ◇ Creation of a modified copy of a node with a new node identity
- Five new kinds of expressions
  - ◇ `insert`, `delete`, `replace`, `rename`, and `transform` expressions

Insert Expression (1)

- Inserts copies of nodes into a designated position with respect to a target node
- Unless specified otherwise (see later)
  - ◇ Inserted nodes become **children** of the target node
  - ◇ Their position of among the children is implementation-dependent
- Add a new product `Silk Skirt` to the catalog

```
insert nodes
  <product dept="WMN">
    <number>666</number>
    <name language="en">Silk Skirt</name>
  </product>
into doc("catalog.xml")/catalog
```

Result

```
<catalog>
  ...
  <product dept="WMN">
    <number>666</number>
    <name language="en">Silk Skirt</name>
  </product>
</catalog>
```

Insert Expression (2)

- In the price list, enter a price of 45 dollars for the product `Silk Skirt`

```
let $num := doc("catalog.xml")/catalog/product[name="Silk Skirt"]/number
return
  insert nodes
    <prod num="{$num}">
      <price currency="USD">45.00</price>
    </prod>
  into doc("prices.xml")/prices/priceList
```

Result

```
<prices>
  <priceList effDate="2006-11-15">
    ...
    <prod num="666">
      <price currency="USD">45.00</price>
    </prod>
  </priceList>
</prices>
```

Insert Expression (3)

- In the price list, insert for the product `Silk Skirt` a discount for clearance sales that is equal to the highest discount of all products

```
let $num := doc("catalog.xml")/catalog/product[name="Silk Skirt"]/number
let $maxdisc := max(doc("prices.xml")/prices/priceList/prod/discount)
return
  insert nodes
    <discount type="CLR">{$maxdisc}</discount>
  into doc("prices.xml")/prices/priceList/prod[num=$num]
```

Result

```
<prices>
  <priceList effDate="2006-11-15">
    ...
    <prod num="666">
      <price currency="USD">45.00</price>
      <discount type="CLR">10</discount>
    </prod>
  </priceList>
</prices>
```

Insert: As First and As Last Clauses

- If `as first` (or `as last`) is specified
  - ◇ inserted nodes become the first (or last) **children** of the target node
  - ◇ If multiple nodes are inserted by a single insert expression, the nodes remain adjacent and their order is preserved
- In the price list, add the element `<comment>This is a bargain !</comment>` as the last child of the product number `666`

```
insert nodes
  <comment>This is a bargain !</comment>
as last into doc("prices.xml")/prices/priceList/prod[numb=666]
```

Result

```
<prices>
  <priceList effDate="2006-11-15">
    ...
    <prod num="666">
      <price currency="USD">45.00</price>
      <discount type="CLR">10</discount>
      <comment>This is a bargain !</comment>
    </prod>
  </priceList>
</prices>
```

Insert: Before and After Clauses

- If `before` (or `after`) is specified
  - ◇ inserted nodes become the preceding (or following) **siblings** of the target node
  - ◇ If multiple nodes are inserted by a single insert expression, the nodes remain adjacent and their order is preserved
- Insert a price of 19.99 euros after the price in dollars of the first product

```
insert nodes <price currency="EUR">19.99</price>
after doc("prices.xml")/prices/priceList/prod[1]/price[@currency="USD"]
```

Result

```
<prices>
  <priceList effDate="2006-11-15">
    <prod num="557">
      <price currency="USD">29.99</price>
      <price currency="EUR">19.99</price>
      <discount type="CLR">10.00</discount>
    </prod>
    ...
  </priceList>
</prices>
```

Replace the Value of a Node (1)

- Used to modify the value of a **single** node while preserving its node identity
- Increase the price of the product number `666` by 10%

```
replace value of node
     doc("prices.xml")/prices/priceList/prod[num="666"]/price
with doc("prices.xml")/prices/priceList/prod[num="666"]/price * 1.1
```

- Applying a similar update for the first product raises an error

  ⇒ now it has both a price in dollars and in euros
- Increase the **price in dollars** of the first product by 10%

```
let $price :=
  doc("prices.xml")/prices/priceList/prod[1]/price[@currency="USD"]
return replace value of node $price with $price * 1.1
```

- Increase **all prices** of the first product by 10%

```
for $price in doc("prices.xml")/prices/priceList/prod[1]/price
return replace value of node $price with $price * 1.1
```

Replace the Value of a Node (2)

- Increase the **price in dollars** of **all products** by 10%

```
for $prod in doc("prices.xml")/prices/priceList/prod
return
    replace value of node $prod/price[@currency="USD"]
    with $prod/price[@currency="USD"] * 1.1
```

- Increase the price of the product `Silk Skirt` by 10% if its discount for clearance sales is less than 5%

```
let $num := doc("catalog.xml")/catalog/product[name="Silk Skirt"]
let $prod := doc("prices.xml")/prices/priceList/prod[@num=$num]
where $prod/discount[type="CLR"] < 5
return
  replace value of node $prod/price with $prod/price * 1.1
```

  `prices.xml` is not modified since the discount of the product is 10%

Setting/Replacing the Value of a Node

- To take into account the case where the node may not exist
- Set the colorChoices element of the product Silk Skirt to purple blue

```
let $prod := doc("catalog.xml")/catalog/product[name="Silk Skirt"]
return
  if ($prod/colorChoices)
    then replace value of node $prod/colorChoices with "purple blue"
    else insert node <colorChoices>purple blue</colorChoices> into $prod
```

Result

```
<catalog>
  ...
  <product dept="WMN">
    <number>666</number>
    <name language="en">Silk Skirt</name>
    <colorChoices>purple blue</colorChoices>
  </product>
</catalog>
```

Replace a Node

- Replaces one node with a new sequence of zero or more nodes
- Replacement nodes occupy the position in the node hierarchy of the node that was replaced
  - ◇ An attribute node can be replaced only by zero or more attribute nodes
  - ◇ An element, text, comment, or processing instruction node can be replaced only by zero or more element, text, comment, or processing instruction nodes
- Replace the color choices of the first product with those of the third product

```
replace node doc("catalog.xml")/catalog/product[1]/colorChoices
with doc("catalog.xml")/catalog/product[3]/colorChoices
```

- If the expression following the with keyword is empty, this implies a deletion
- Replace the color choices of the first product with those of the second product

```
replace node doc("catalog.xml")/catalog/product[1]/colorChoices
with doc("catalog.xml")/catalog/product[2]/colorChoices
```

will remove the element colorChoices from the first product, since the second product does not have such element

Delete Expression

- Deletes zero or more nodes
- Delete the last item from the order number 00299432

```
delete node doc("orders.xml")/order[@number="00299432"]/item[last()]
```

- Erase the product Silk Skirt and the associated prices and orders

```
let $prod := doc("catalog.xml")/catalog/product[name="Silk Skirt"]
let $prices := doc("prices.xml")/prices/priceList/prod[@num=$prod/number]
let $orders := doc("orders.xml")/order/item[@num=$prod/number]
return (
  delete nodes $prod,
  delete nodes $prices,
  delete nodes $orders
)
```

Alternative solution

```
let $prod := doc("catalog.xml")/catalog/product[name="Silk Skirt"]
let $prices := doc("prices.xml")/prices/priceList/prod[@num=$prod/number]
let $orders := doc("orders.xml")/order/item[@num=$prod/number]
return (
  delete nodes $prod, $prices, $orders
)
```

Rename Expression

- Replaces the name property of a node with a new QName
- Rename the desc element of the last product to description

```
rename node doc("catalog.xml")/catalog/product[last()]/desc
as "description"
```

- Attributes and descendants of the target node are not affected
- A global change of names or namespaces needs some form of explicit iteration
- Change attributes and descendants of a node bound to $root with the prefix abc to have a new prefix xyz and a new namespace http://xyz/ns

```
for $node in $root//abc:*
let $localName := local-name($node),
    $newQName := concat("xyz:", $localName)
return
   rename node $node as QName("http://xyz/ns", $newQName),
   for $attr in $node/@abc:*
   let $attrLocalName := local-name($attr),
       $attrNewQName := concat("xyz:", $attrLocalName)
   return
      rename node $attr as QName("http://xyz/ns", $attrNewQName)
```

Transform Expression

- Creates modified copies of existing nodes with **new node identity**
- Consists of three clauses, denoted by the keywords `copy`, `modify`, and `return`
- The `modify` may contain an empty expression `( )`
- Return a sequence consisting of all products that have a `colorChoice` element, excluding their `desc` child-elements

```
for $e in doc("catalog.xml")/catalog/product[colorChoice]
return
   copy $ce := $e
   modify delete node $ce/desc
   return $ce
```

- Copy the product 557 in `prices.xml`, add 10% to the price in the copy, and return both the original node and the modified copy

```
let $oldprod := doc("prices.xml")/prices/priceList/prod[@num=557]
return
   copy $newprod := $oldprod
   modify (rename node $newprod as "newprod",
           replace value of node $newprod/price by $newprod/price * 1.1)
   return ($oldprod, $newprod)
```

141

Use Case: Synchronizing Address Book Entries

- An address book is synchronized among a central archive and two local copies
- Entries with the same `name` element are considered to be the same entry
- It is assumed that entries may be modified, but not inserted or deleted
- Synchronization rules
  - ◇ If an entry in one of the two copies is different from the archived one, but the other copy matches the archive, the modified copy is propagated to the archive and to the other copy
  - ◇ If both copies differ from the archive, but they do not both modify the same element in the entry, or they modify the same element but the modified elements have the same value, then the changes from each copy are propagated to both the archive and the other copy
  - ◇ If the copies have each modified the same entry, but modified it in different ways, a problem is reported in the log, and neither the archive nor the copies are changed (do not attempt to merge updates)

142

Synchronizing Address Book Entries: Input Data (1)

- `archive.xml`: The central archive

```
<archived-agenda>
  <last-synch-time>2005-10-05T10:00</last-synch-time>
  <entry>
    <name>Benjamin</name>
    <contact>benjamin@inria.fr</contact>
  </entry>
  <entry>
    <name>Dario</name>
    <contact>dario@uni-pisa.it</contact>
  </entry>
  <entry>
    <name>Anthony</name>
    <contact>tony@uni-toulon.fr</contact>
  </entry>
</archived-agenda>
```

- `log.xml`: The central log, before synchronization

```
<log>
</log>
```

143

Synchronizing Address Book Entries: Input Data (2)

- `copy1.xml`: The first modified copy of the address book

```
<agenda-version>
  <entry>
    <name>Benjamin</name>
    <contact>benjamin@uni-versailles.fr</contact>
  </entry>
  <entry>
    <name>Dario</name>
    <contact>dario@uni-parissud.fr</contact>
  </entry>
  <entry>
    <name>Anthony</name>
    <contact>tony@ena.fr</contact>
  </entry>
</agenda-version>
```

144

Synchronizing Address Book Entries: Input Data (3)

- `copy2.xml`: The second modified copy of the address book

```
<agenda-version>
  <entry>
    <name>Benjamin</name>
    <contact>benjamin@uni-versailles.fr</contact>
  </entry>
  <entry>
    <name>Dario</name>
    <contact>dario@uni-pisa.it</contact>
  </entry>
  <entry>
    <name>Anthony</name>
    <contact>tony@ehess.fr</contact>
  </entry>
 </agenda-version>
```

Synchronizing Address Book Entries: Query

```
for $a in doc("archive.xml")/archived-agenda/entry,
    $v1 in doc("copy1.xml")/agenda-version/entry,
    $v2 in doc("copy2.xml")/agenda-version/entry
where $a/name = $v1/name and $v1/name = $v2/name
return
  if ($a/contact = $v1/contact and $v1/contact=$v2/contact) then ()
  else
    if ($v1/contact = $v2/contact)
    then replace value of node $a/contact with $v1/contact
    else
      if ($a/contact = $v1/contact)
      then ( replace value of node $a/contact with $v2/contact,
             replace value of node $v1/contact with $v2/contact )
      else
        if ($a/contact = $v2/contact)
        then ( replace value of node $a/contact with $v1/contact,
               replace value of node $v2/contact with $v1/contact )
        else ( insert node
            <fail>
               <arch>{ $a }</arch> <v1>{ $v1 }</v1> <v2>{ $v2 }</v2>
            </fail>
          into doc("log.xml")/log ),
replace value of node doc("archive.xml")/*/last-synch-time
        with current-dateTime()
```

Synchronizing Address Book Entries: Result (1)

- `archive.xml` after synchronization

```
<archived-agenda>
  <last-synch-time>2006-04-23T12:00</last-synch-time>
  <entry>
    <name>Benjamin</name>
    (: copied from the modified entries :)
    <contact>benjamin@uni-versailles.fr</contact>
  </entry>
  <entry>
    <name>Dario</name>
    (: copied from first modified version :)
    <contact>dario@uni-parissud.fr</contact>
  </entry>
  <entry>
    <name>Anthony</name>
    (: unchanged due to conflict :)
    <contact>tony@uni-toulon.fr</contact>
  </entry>
</archived-agenda>
```

Synchronizing Address Book Entries: Result (2)

- `log.xml` after synchronization

```
<log>
  <fail>
    <arch>
      <entry>
        <name>Anthony</name>
        <contact>tony@uni-toulon.fr</contact>
      </entry>
    </arch>
    <v1>
      <entry>
        <name>Anthony</name>
        <contact>tony@ena.fr</contact>
      </entry>
    </v1>
    <v2>
      <entry>
        <name>Anthony</name>
        <contact>tony@ehess.fr</contact>
      </entry>
    </v2>
  </fail>
</log>
```

Use Case: Modifying Recursive Aggregations (1)

- part-tree.xml: Hierarchical representation

```
<parttree>
  <part partid="0" name="car">
    <part partid="1" name="engine">
      <part partid="3" name="piston"/>
    </part>
    <part partid="2" name="door">
      <part partid="4" name="window"/>
      <part partid="5" name="lock"/>
    </part>
  </part>
  <part partid="10" name="skateboard">
    <part partid="11" name="board"/>
    <part partid="12" name="wheel"/>
  </part>
  <part partid="20" name="canoe"/>
</parttree>
```

Use Case: Modifying Recursive Aggregations (2)

- partlist.xml: Flat representation

```
<partlist>
  <part partid="0" name="car"/>
  <part partid="1" partof="0" name="engine"/>
  <part partid="2" partof="0" name="door"/>
  <part partid="3" partof="1" name="piston"/>
  <part partid="4" partof="2" name="window"/>
  <part partid="5" partof="2" name="lock"/>
  <part partid="10" name="skateboard"/>
  <part partid="11" partof="10" name="board"/>
  <part partid="12" partof="10" name="wheel"/>
  <part partid="20" name="canoe"/>
</partlist>
```

Modifying Recursive Aggregations: Queries (1)

- Delete all parts in part-tree.xml

```
delete nodes doc("part-tree.xml")//part
```

- Delete all parts belonging to a car in part-tree.xml, leaving the car itself

```
delete nodes doc("part-tree.xml")//part[@name="car"]//part
```

Result

```
<parttree>
  <part partid="0" name="car"/>
  <part partid="10" name="skateboard">
    <part partid="11" name="board"/>
    <part partid="12" name="wheel"/>
  </part>
  <part partid="20" name="canoe"/>
</parttree>
```

Modifying Recursive Aggregations: Queries (2)

- Delete all parts belonging to a car in part-list.xml, leaving the car itself

```
for $pt in doc("part-tree.xml")//part[@name="car"]//part,
    $pl in doc("part-list.xml")//part
where $pt/@partid eq $pl/@partid
return
  delete nodes $pl
```

Alternative solution 2 using a recursive updating function

```
declare updating function local:delete-subtree($p as element(part))
  {
      for $child in doc("part-list.xml")//part
      where $p/@partid eq $child/@partof
      return (
        delete nodes $child,
        local:delete-subtree($child)
      )
  };
for $p in doc("part-list.xml")//part[@name="car"]
return
  local:delete-subtree($p)
```

Modifying Recursive Aggregations: Queries (3)

- Add a radio to the car in part-tree.xml, using a part number that hasn't been taken.

```
let $next := max(doc("part-tree.xml")//@partid) + 1
  return
    insert nodes <part partid="{$next}" name="radio"/>
      into doc("part-tree.xml")//part[@partid=0 and @name="car"]
```

  Position of new element with respect to its siblings is implementation-dependent

  If position is significant, the next query ensures that the element appears last

```
let $next := max(doc("part-tree.xml")//@partid) + 1
  return
    insert nodes <part partid="{$next}" name="radio"/>
      as last into doc("part-tree.xml")//part[@partid=0 and @name="car"]
```

153

Modifying Recursive Aggregations: Queries (4)

- The head office has adopted a new numbering scheme. In part-tree.xml, add 1000 to all part numbers for cars, 2000 to all part numbers for skateboards, and 3000 to all part numbers for canoes

```
for $keyword at $i in ("car", "skateboard", "canoe"),
    $parent in doc("part-tree.xml")//part[@name=$keyword]
let $descendants := $parent//part
for $p in ($parent, $descendants)
return
  replace value of node $p/@partid with $i*1000+$p/@partid
```

154

References

- Priscilla Walmsley, *XQuery*, O'Reilly, 2007
- Don Chamberlin, Daniela Florescu, Jim Melton, Jonathan Robie, Jrme Simon, *XQuery Update Facility 1.0*, http://www.w3.org/TR/xquery-update-10/
- Ioana Manolescu, Jonathan Robie, *XQuery Update Facility 1.0 Use Cases* http://www.w3.org/TR/xquery-update-10-use-cases/
- Jim Melton, Stephen Buxton, *Querying XML: XQuery, XPath, and SQL/XML in context*, Morgan Kaufmann, 2006
- Akmal B. Chaudhri, Awais Rashid, Roberto Zicari, *XML Data Management: Native XML and XML-Enabled Database Systems*, 2003

155