# INFO-H-509 XML TECHNOLOGIES

## Lecture 6: XQuery

Stijn Vansummeren

March 27, 2019


Introduction

# OUR STORY SO FAR ...

- XML is a standard notation for documents and data
- We can select parts of XML with XPath
- We can describe schemas for XML languages by DTDs and XSDs
- We can transform XML documents using XSLT

**Often we want to query XML documents:**

- What is the equivalent of “SQL” for XML?

# FROM RELATIONS TO TREES (1/2)

## A relational student database:

Students(id,name,age)

| | | |
|---|---|---|
| 100026 | Joe Average | 21 |
| 100078 | Jack Doe | 18 |

Majors(id,major)

| | |
|---|---|
| 100026 | Biology |
| 100078 | Physics |
| 100078 | XML Science |

Grades(id,course,grade)

| | | |
|---|---|---|
| 100026 | Math 101 | C- |
| 100026 | Biology 101 | C+ |
| 100026 | Statistics 101 | D |
| 100078 | Math 101 | A+ |
| 100078 | XML 101 | A- |
| 100078 | Physics 101 | B+ |
| 100078 | XML 102 | A |

# FROM RELATIONS TO TREES (1/2)

## A relational student database:

Students(id,name,age)

| | | |
|---|---|---|
| 100026 | Joe Average | 21 |
| 100078 | Jack Doe | 18 |

Majors(id,major)

| | |
|---|---|
| 100026 | Biology |
| 100078 | Physics |
| 100078 | XML Science |

Grades(id,course,grade)

| | | |
|---|---|---|
| 100026 | Math 101 | C- |
| 100026 | Biology 101 | C+ |
| 100026 | Statistics 101 | D |
| 100078 | Math 101 | A+ |
| 100078 | XML 101 | A- |
| 100078 | Physics 101 | B+ |

SQL Query:

```
SELECT S.name
FROM Students S, Grades G, Majors M
WHERE S.id = M.id AND M.major = "Biology"
AND S.id = G.id AND G.grade = "D"
```

# FROM RELATIONS TO TREES (2/2)

A more natural representation:

```
<students>
  <student id="100026">
    <name">Joe Average</name>
    <age>21</age>
    <major>Biology</major>
    <results>
      <result course="Math 101" grade="C-"/>
      <result course="Biology 101" grade="C+"/>
      <result course="Statistics 101" grade="D"/>
    </results>
  </student>
  <student id="100078">
    <name>Jack Doe</name>
    <age>18</age>
    <major>Physics</major>
    <major>XML Science</major>
    <results>
      <result course="Math 101" grade="A"/>
      <result course="XML 101" grade="A-"/>
      <result course="Physics 101" grade="B+"/>
      <result course="XML 102" grade="A"/>
    </results>
  </student>
</students>
```

# FROM RELATIONS TO TREES (2/2)

A more natural representation:

```
<students>
  <student id="100026">
    <name">Joe Average</name>
    <age>21</age>
    <major>Biology</major>
    <results>
      <result course="Math 101" grade="C-"/>
      <result course="Biology 101" grade="C+"/>
      <result course="Statistics 101" grade="D"/
    </results>
  </student>
  <student id="100078">
    <name>Jack Doe</name>
    <age>18</age>
    <major>Physics</major>
    <major>XML Science</major>
    <results>
      <result course="Math 101" grade="A"/>
      <result course="XML 101" grade="A-"/>
      <result course="Physics 101" grade="B+"/>
      <result course="XML 102" grade="A"/>
    </results>
  </student>
</students>
```

How do we query this?

- With XPath
- With XSLT
- With XQuery

# 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
- The XQuery query language coredoes not provide updates
  - XQuery Update Facility is an extension to XQuery, with a separate recommendation.

# COMMON USE CASES OF XQUERY

- XQuery is sometimes called the “SQL of XML”
- 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

# XQUERY VS XPATH

**XPath ($\geq$ 2.0) is a proper syntactic subset of XQuery**

- So every XPath 2.0 expression is an XQuery expression (but not conversely)!
- XPath expressions can only select sequences of nodes from the input XML tree
- XQuery expressions have additionally the power to:
  - join information from multiple sources,
  - generate new XML fragments,
  - define user-defined functions.

# XQUERY VS XSLT

- XQuery and XSLT are both domain-specific languages for combining and transforming XML data from multiple sources.
- In principle everything that you can do with XSLT ($\geq$ 2.0) you can do with XQuery, and vise verse.
- They are vastly different in design and processing model, however.
- Therefore, some queries are easier to express in XQuery (i.e., the analogs of SQL queries), while others are easier to express in XSLT (transformations based on recursive processing).

# WHAT SOFTWARE USES XQUERY?

Native XML databases:

- Are specifically designed to store collections of XML documents
- Examples: Tamino, eXist, 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

Relational database products

- Examples: Oracle, IBM DB2, Microsoft SQL Server
- Allow you to store XML documents inside a relational database
- And call XQuery from within SQL (or SQL from within XQuery)

Independent XQuery processors

- Example: Saxon
- Input: xquery query + list of documents, computes output
- Might also operate on XML data passed in memory from some other process

# AN EXAMPLE XQUERY

```
declare namespace stud = "http://students.org";

declare function stud:is_failing($s) {
  return (some $g in $s//stud:result satisfies $g/stud:grade eq "D")
}

<failing_students> {
  for $s in fn:doc("students.xml")//stud:student
  where stud:is_failing($s)
  return <student> $s/name </student>
} <failing_students>
```

# AN EXAMPLE XQUERY

```
declare namespace stud = "http://students.org";

declare function stud:is_failing($s) {
  return (some $g in $s//stud:result satisfies $g/stud:grade eq "D")
}

<failing_students> {
  for $s in fn:doc("students.xml")//stud:student
  where stud:is_failing($s)
  return <student> $s/name </student>
} <failing_students>
```

**General structure of an XQuery query**

1. Query Prolog
2. Zero or more function declarations
3. A single XQuery expression computing the output

# AN EXAMPLE XQUERY

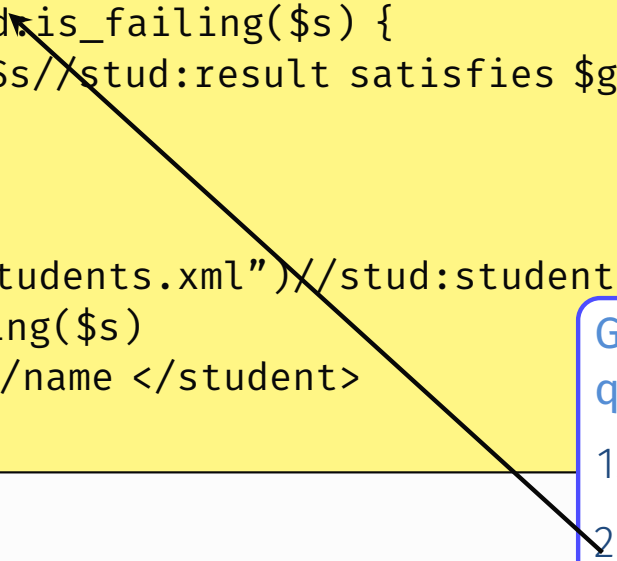
```
declare namespace stud = "http://students.org";

declare function stud:is_failing($s) {
  return (some $g in $s//stud:result satisfies $g/stud:grade eq "D")
}

<failing_students> {
  for $s in fn:doc("students.xml")//stud:student
  where stud:is_failing($s)
  return <student> $s/name </student>
} <failing_students>
```

**General structure of an XQuery query**

1. Query Prolog
2. Zero or more function declarations
3. A single XQuery expression computing the output

# AN EXAMPLE XQUERY

```
declare namespace stud = "http://students.org";

declare function stud:is_failing($s) {
  return (some $g in $s//stud:result satisfies $g/stud:grade eq "D")
}

<failing_students> {
  for $s in fn:doc("students.xml")//stud:student
  where stud:is_failing($s)
  return <student> $s/name </student>
} <failing_students>
```

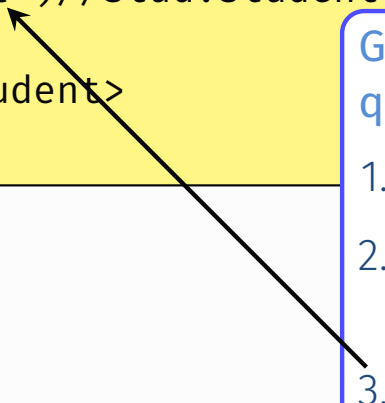


**General structure of an XQuery query**

1. Query Prolog
2. Zero or more function declarations
3. A single XQuery expression computing the output

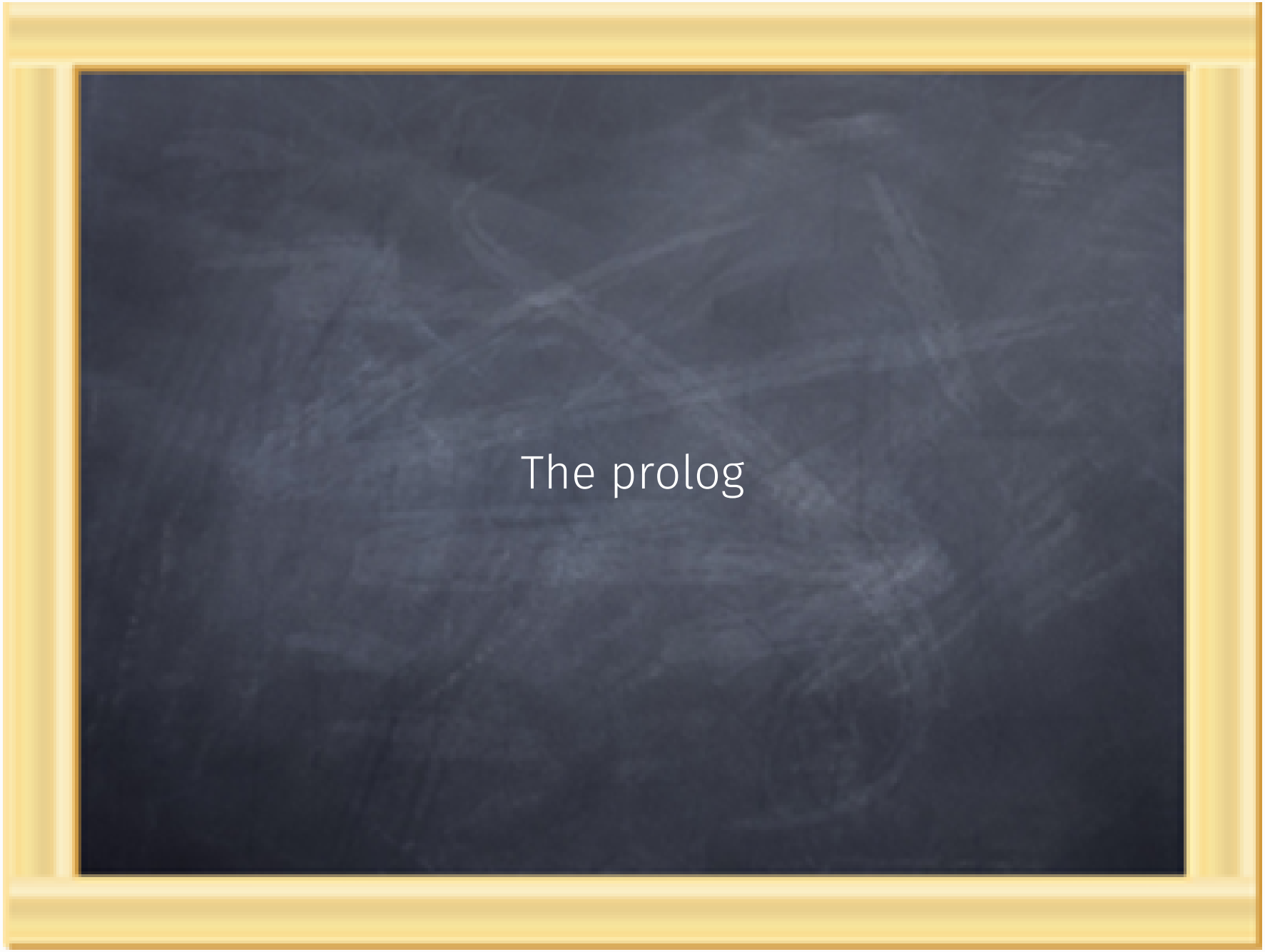
# AN EXAMPLE XQUERY

```
declare namespace stud = "http://students.org";

declare function stud:is_failing($s) {
  return (some $g in $s//stud:result satisfies $g/stud:grade eq "D")
}

<failing_students> {
  for $s in fn:doc("students.xml")//stud:student
  where stud:is_failing($s)
  return <student> $s/name </student>
} <failing_students>
```



**General structure of an XQuery query**

1. Query Prolog
2. Zero or more function declarations
3. A single XQuery expression computing the output


The prolog

# THE XQUERY PROLOG

- The optional XQuery Prolog defines various parameters for the XQuery processor, such as:

```
xquery version "1.0";
declare xmlspace preserve;
declare default element namespace "http://students.org";
declare default function namespace "http://students.org/func";
declare namespace stud = "http://students.org";
import schema at "http://students.org/students.xsd";
```

- The following namespace prefixes are always implicitly declared

```
declare namespace xml = "http://www.w3.org/XML/1998/namespace";
declare namespace xs = "http://www.w3.org/2001/XMLSchema";
declare namespace xsi = "http://www.w3.org/2001/XMLSchema-instance";
declare namespace fn = "http://www.w3.org/2005/11/xpath-functions";
declare namespace xdt = "http://www.w3.org/2005/11/xpath-datatypes";
declare namespace local = "http://www.w3.org/2005/11/xquery-local-functions";
```

# XQuery Expressions

# THE XQUERY DATA MODEL

## XQuery extends the syntax of XPath ($\geq$ 2.0)

- Every XPath expression is an XQuery expression, but not conversely

## In particular, XQuery uses the same data model as XPath

- It operates on and produces sequences of items;
- An item is either a node or atomic data value;
- Atomic data values are simple data values with no markup associated to it (strings, integers, ...);
- And a node corresponds to an XML construct (element nodes, text nodes, attribute nodes, ...)
- From 3.0 onwards: also maps and arrays.

# THE XQUERY DATA MODEL

## XQuery extends the syntax of XPath ($\geq$ 2.0)

- Every XPath expression is an XQuery expression, but not conversely

## In particular, XQuery uses the same data model as XPath

- It operates on and produces sequences of items;
- An item is either a node or atomic data value;
- Atomic data values are simple data values with no markup associated to it (strings, integers, ...);
- And a node corresponds to an XML construct (element nodes, text nodes, attribute nodes, ...)
- From 3.0 onwards: also maps and arrays.

## But there is one essential difference:

- In contrast to XPath, the initial context node, position, and size are undefined
- XML data is accessed through the `fn:doc()` function, e.g.:

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

# XQUERY EXPRESSIONS

## XPath 2.0 Expression categories:

- **Primary**: literals, variables, function calls, and parenthesized expressions
- **Arithmetic** operations using `+`, `-`, `*`, `div`, `idiv`, `mod`
- **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::`, . . .
- **Quantified**: Test whether sequences fulfill conditions `some`, `every`, `in`, `satisfies`
- **Sequence-related**: Create and combine sequences using `union`, `intersect`, `except`
- **For-loops**: Iterate over sequences using `for ... in ... return ...`

## To this, XQuery adds the following new categories:

- **Constructor**: Creating new XML nodes using `<`, `>`, `element`, `attribute`
- **FLWOR**: Select and process nodes using `for`, `let`, `where`, `order by`, `return`
- **Type-related**: Cast and validate values based on type using `instance of`, `typeswitch`, `cast as`, `castable`, `treat`, `validate`

# INTERMEZZO: SAMPLE 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>
```

# INTERMEZZO: SAMPLE 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>
```

# INTERMEZZO: SAMPLE 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>
```

# XQUERY EXPRESSIONS: XML CONSTRUCTORS

- A query typically returns elements and attributes from the input document

Query:

```
for $prod in fn:doc("catalog.xml")/catalog/product[dept = 'ACC']
return $prod
```

Output:

```
<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>
```

- XQuery expressions may construct new XML nodes
- Each node is created with a unique node identity
- Constructors may be either direct or computed

# XQUERY EXPRESSIONS: DIRECT CONSTRUCTORS

- Construct new nodes using standard XML syntax
- Constructors can enclose expressions between braces: `{ expr }`

Query:

```
<html>
  <h1>Product Catalog</h1>
</html>
```

Output:

```
<html>
  <h1>Product Catalog</h1>
</html>
```

# XQUERY EXPRESSIONS: DIRECT CONSTRUCTORS

- Construct new nodes using standard XML syntax
- Constructors can enclose expressions between braces: `{ expr }`

Query:

```
<html>
  <h1>Product Catalog</h1>
  <ul>{
    for $prod in fn:doc("catalog.xml")/catalog/product
    return <li>number</li>
  } </ul>
</html>
```

Output:

```
<html>
  <h1>Product Catalog</h1>
  <ul>
    <li>number</li>
    <li>number</li>
    <li>number</li>
    <li>number</li>
  </ul>
</html>
```

# XQUERY EXPRESSIONS: DIRECT CONSTRUCTORS

- Construct new nodes using standard XML syntax
- Constructors can enclose expressions between braces: `{ expr }`

Query:

```
<html>
  <h1>Product Catalog</h1>
  <ul>{
    for $prod in fn:doc("catalog.xml")/catalog/product
    return <li>number: {data($prod/number)}, name: {data($prod/name)} </li>
  } </ul>
</html>
```

Output:

```
<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>
```

# XQUERY EXPRESSIONS: DIRECT CONSTRUCTORS

- Construct new nodes using standard XML syntax
- Constructors can enclose expressions between braces: `{ expr }`

Query:

```
for $prod in fn:doc("catalog.xml")/catalog/product
return <li>number: {$prod/number}</li>
```

Output:

```
<li>number: <number>557</number></li>
<li>number: <number>563</number></li>
<li>number: <number>443</number></li>
<li>number: <number>784</number></li>
```

# XQUERY EXPRESSIONS: DIRECT ATTRIBUTE CONSTRUCTORS

Query:

```
<html>
  <h1 class="itemHdr">Product Catalog</h1>
  <ul>{
    for $prod in fn:doc("catalog.xml")/catalog/product
    return
      <li dep="{$prod/@dept}">
        number: {data($prod/number)}, name: {data($prod/name)}
      </li>
    }</ul>
</html>
```

Output:

```
<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>
```

- Careful: if the `$prod` element has no `dept` attribute, the `li` item will have an empty `dep` attribute value

# XQUERY EXPRESSIONS: NAMESPACES IN XML CONSTRUCTORS

Query:

```
declare default element namespace "http://businesscard.org";

<card>
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```

Output:

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```

# XQUERY EXPRESSIONS: NAMESPACES IN XML CONSTRUCTORS

Query:

```
declare namespace b = "http://businesscard.org";

<b:card>
  <b:name>John Doe</b:name>
  <b:title>CEO, Widget Inc.</b:title>
  <b:email>john.doe@widget.com</b:email>
  <b:phone>(202) 555-1414</b:phone>
  <b:logo uri="widget.gif"/>
</b:card>
```

Output:

```
<b:card xmlns:b="http://businesscard.org">
  <b:name>John Doe</b:name>
  <b:title>CEO, Widget Inc.</b:title>
  <b:email>john.doe@widget.com</b:email>
  <b:phone>(202) 555-1414</b:phone>
  <b:logo uri="widget.gif"/>
</b:card>
```

# XQUERY EXPRESSIONS: NAMESPACES IN XML CONSTRUCTORS

Query:

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```

Output:

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```

# XQUERY EXPRESSIONS: COMPUTED CONSTRUCTORS (1)

- Construct new nodes using the keywords **element**, **attribute**, **comment**, ...
- The names of the new element and attribute nodes can be computed!

Query:

```
element html {
  element h1 { "Product Catalog" },
  element ul {
    for $prod in fn:doc("catalog.xml")/catalog/product
    return element li {"number:",data($prod/number),", name:",data($prod/name)}
  }
}
```

Output:

```
<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>
```

# XQUERY EXPRESSIONS: COMPUTED CONSTRUCTORS (1)

- Construct new nodes using the keywords **element**, **attribute**, **comment**, ...
- The names of the new element and attribute nodes can be computed!

Create a product catalog that has the names of the departments as element names instead of attribute values

Query:

```
for $dept in fn:distinct-values(fn:doc("catalog.xml")/catalog/product/@dept)
return
  element {$dept}
        {fn:doc("catalog.xml")/catalog/product[@dept = $dept]/name}
```

Output:

```
<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 EXPRESSIONS: COMPUTED CONSTRUCTORS (2)

**Computed element or attribute construction**

- Syntax: `element` $\{e_1\}$ $\{e_2\}$ or `element lbl` $\{e_2\}$
- Syntax: `attribute` $\{e_1\}$ $\{e_2\}$ or `attribute lbl` $\{e_2\}$

- Here, $e_1$ can be any expression that evaluates to a qualified name as in `element {fn:concat("h",$level)} { "Product Catalog" }`
- `element {fn:node-name($myNode)} { "contents" }` will give the new element the same name as the node that is bound to the variable
- If $e_1$ returns a node instead of an atomic data value, then the atomized value of the node will be used (not its name)
- Careful: $e_1$ should always evaluate to a sequence that contains a single item (otherwise runtime error)!

# XQUERY EXPRESSIONS: FLWOR

- FLWOR = for - let - where - order by -return
- FLWOR expressions allow to join data from multiple sources, construct new elements and attributes, evaluate functions on intermediate values, sort results
- A FLWOR must have at least one `for` or `let` clause
- 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

# XQUERY EXPRESSIONS: THE FLWOR FOR CLAUSE

- Syntax: `for $var in` $e_1$ `return` $e_2$
- Works just like in XPath; except that $e_1$ and $e_2$ can now contain XQuery constructs
- You can have multiple for clauses - this works like nested iteration in programming languages

Query:

```
for $i in 1 to 3
return <oneEval>{$i}</oneEval>
```

Output:

```
<oneEval>1</oneEval>
<oneEval>2</oneEval>
<oneEval>3</oneEval>
```

# XQUERY EXPRESSIONS: THE FLWOR FOR CLAUSE

- Syntax: `for $var in` $e_1$ `return` $e_2$
- Works just like in XPath; except that $e_1$ and $e_2$ can now contain XQuery constructs
- You can have multiple for clauses - this works like nested iteration in programming languages

Query:

```
for $i in (1, 2)
for $j in fn:reverse("b", "a")
return <oneEval>i is {$i} and j is {$j}</oneEval>
```

Output:

```
<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>
```

# XQUERY EXPRESSIONS: THE FLWOR FOR CLAUSE

- Syntax: `for $var in` $e_1$ `return` $e_2$
- Works just like in XPath; except that $e_1$ and $e_2$ can now contain XQuery constructs
- You can have multiple for clauses - this works like nested iteration in programming languages

Shorthand:

```
for $i in (1, 2), $j in fn:reverse("b", "a")
return <oneEval>i is {$i} and j is {$j}</oneEval>
```

Output:

```
<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>
```

# XQUERY EXPRESSIONS: THE FLWOR LET CLAUSE

- Syntax: `let $var :=` $e_1$
- Eovaluates $e_1$ and binds the result to `$var`, which can be used in the rest of the FLWOR expression
- `Let` and `for` clauses can be mixed
- All `let` and for clauses must occur before any `where`, `order by`, and `return` clauses.

Query:

```
let $i := (1 to 3)
return <oneEval>$i</oneEval>
```

Output:

```
<oneEval>1 2 3</oneEval>
```

# XQUERY EXPRESSIONS: THE FLWOR LET CLAUSE

- Syntax: `let $var :=` $e_1$
- Eovaluates $e_1$ and binds the result to `$var`, which can be used in the rest of the FLWOR expression
- `Let` and `for` clauses can be mixed
- All `let` and for clauses must occur before any `where`, `order by`, and `return` clauses.

Another example:

```
let $doc := fn:doc("catalog.xml")
for $prod in $doc//product
let $prodDept := $prod/@dept
let $prodName := $prod/name
where $prodDept = "ACC" or $prodDept = "WMN"
return $prodName
```

# XQUERY EXPRESSIONS: THE FLWOR LET CLAUSE

- Syntax: `let $var :=` $e_1$
- Eovaluates $e_1$ and binds the result to `$var`, which can be used in the rest of the FLWOR expression
- `Let` and `for` clauses can be mixed
- All `let` and for clauses must occur before any `where`, `order by`, and `return` clauses.

Or shorter:

```
let $doc := fn:doc("catalog.xml")
for $prod in $doc//product
let $prodDept := $prod/@dept,
    $prodName := $prod/name
where $prodDept = "ACC" or $prodDept = "WMN"
return $prodName
```

# XQUERY EXPRESSIONS: THE FLWOR WHERE CLAUSE

- Specify criteria that filter the results of the FLWOR expression
- Syntax: `where` *e*
- *e* is evaluated for each iteration of the preceding `for` clauses; its results is transformed into a boolean (cf. XPath)
- only those context items for which *e* yields `true` (after transformation) are used to construct the output

Query:

```
for $i in (1 to 3)
where $i > 2
return <oneEval>$i</oneEval>
```

Output:

```
<oneEval>3</oneEval>
```

# XQUERY EXPRESSIONS: THE FLWOR WHERE CLAUSE

- Specify criteria that filter the results of the FLWOR expression
- Syntax: `where` *e*
- *e* is evaluated for each iteration of the preceding `for` clauses; its results is transformed into a boolean (cf. XPath)
- only those context items for which *e* yields `true` (after transformation) are used to construct the output

Another example:

```
let $doc := fn:doc("catalog.xml")
for $prod in $doc//product
let $prodDept := $prod/@dept
let $prodName := $prod/name
where $prodDept = "ACC" or $prodDept = "WMN"
return $prodName
```

# XQUERY EXPRESSIONS: THE FLWOR WHERE CLAUSE

- Specify criteria that filter the results of the FLWOR expression
- Syntax: `where` *e*
- *e* is evaluated for each iteration of the preceding `for` clauses; its results is transformed into a boolean (cf. XPath)
- only those context items for which *e* yields `true` (after transformation) are used to construct the output

Another example: (what does it do?)

```
let $doc := fn:doc("catalog.xml")
for $prod in $doc//product
where $prod/@dept
return $prod/name
```

# XQUERY EXPRESSIONS: JOINS

- FLWORs allow to easily join data from multiple sources.

Join information from products (catalog.xml) and orders (order.xml): list all items in the order, along with their number, name, and quantity

Query:

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

Output:

```
<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"/>
```

# XQUERY EXPRESSIONS: JOINS

- FLWORs allow to easily join data from multiple sources.

Join information from products (catalog.xml) and orders (order.xml): list all items in the order, along with their number, name, and quantity

Alternative Query:

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

Output:

```
<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"/>
```

# XQUERY EXPRESSIONS: JOINS

- FLWORs allow to easily join data from multiple sources.

Join information from products (catalog.xml), orders (order.xml), and prices (prices.xml) list all items in the order, along with their number, name, and price

Query:

```
for $item in fn:doc("order.xml")//item,
    $product in fn:doc("catalog.xml")//product
    $price in fn: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}"/>
```

Output:

```
<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"/>
```

# XQUERY EXPRESSIONS: 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

Query:

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

Output:

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

# XQUERY EXPRESSIONS: SELECTING DISTINCT VALUES

- `fn:distinct-values` function selects distinct atomic values from a sequence
- Function determines whether two values are equal using the `eq` operator

Query:

```
fn:distinct-values(fn:doc("catalog.xml")//product/@dept)
```

Output:

```
"WMN", "ACC", "MEN"
```

# XQUERY EXPRESSIONS: SELECTING DISTINCT VALUES

- `fn:distinct-values` function selects distinct atomic values from a sequence
- Function determines whether two values are equal using the `eq` operator

Query:

```
let $prods := fn:doc("catalog.xml")//product
for $d in fn:distinct-values($prods/@dept),
    $n in distinct-values($prods[@dept = $d]/number)
return <result dept="{$d}" number="{$n}"/>
```

Output:

```
<result dept="WMN" number="557"/>
<result dept="ACC" number="563"/>
<result dept="ACC" number="443"/>
<result dept="MEN" number="784"/>
```

# XQUERY EXPRESSIONS: FLWOR ORDER BY CLAUSE

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

Query with single sort key:

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

Output:

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

# XQUERY EXPRESSIONS: FLWOR ORDER BY CLAUSE

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

Query with multiple sort keys:

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

Output:

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

# XQUERY EXPRESSIONS: FLWOR ORDER BY CLAUSE

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

Query using ascending/descending modifiers (ascending=default):

```
for $item in fn:doc("order.xml")//item
order by $item/dept descending, $item/@num ascending
return $item
```

Output:

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

# XQuery Expressions: FLWOR order by clause (2)

- `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

Query:

```
for $item in fn:doc("order.xml")//item
order by $item/@color empty greatest
return $item
```

Output:

```
<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"/>
```

# XQuery Expressions: FLWOR order by clause (2)

- `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

Query:

```
for $item in fn:doc("order.xml")//item
order by $item/@color empty least
return $item
```

Output:

```
<item dept="ACC" num="563" quantity="1"/>
<item dept="ACC" num="443" quantity="2"/>
<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"/>
```

# XQuery Expressions: FLWOR order by clause (2)

- `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

Query:

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

Output:

```
<item dept="ACC" num="563" quantity="1"/>
<item dept="ACC" num="443" quantity="2"/>
<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"/>
```

# XQuery Expressions: FLWOR order by clause (3)

- Careful: Inadvertent resorting in document order
- Some expressions, including path expressions and operators that combine sequences (`|`, `union`, `intersect`, and `except`), return nodes in document order

Sort products by product number, then returns their names in li elements

Incorrect Query:

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

Output:

```
<li> Fleece Pullover </li>
<li> Floppy Sun Hat </li>
<li> Deluxe Travel Bag </li>
<li> Cotton Dress Shirt </li>
```

# XQuery Expressions: FLWOR order by clause (3)

- Careful: Inadvertent resorting in document order
- Some expressions, including path expressions and operators that combine sequences (`|`, `union`, `intersect`, and `except`), return nodes in document order

Sort products by product number, then returns their names in li elements

Correct Query:

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

Output:

```
<li> Deluxe Travel Bag </li>
<li> Floppy Sun Hat </li>
<li> Fleece Pullover </li>
<li> Cotton Dress Shirt </li>
```

# Functions and Types

# FUNCTIONS

- There are over 100 functions built into XQuery, covering a broad range of functionality (see `http://www.w3.org/TR/xpath-functions/`)
- Functions can be used to manipulate strings and dates, perform mathematical calculations, combine sequences of elements, load documents (`fn:doc`) . . .
- In contrast to XPath, users can also define functions in XQuery, either in the query itself, or in an external library
- Both built-in and user-defined functions can be used (called) anywhere an expression is permitted.

# FUNCTIONS AND NAMESPACES

- Each function ”lives in a namespace”, and should be called by its qualified name (including the namespace)
- Built-in functions live in namespace `http://www.w3.org/2005/xpath-functions`, which is bound to the prefix `fn` by default
- However, for built-in functions this prefix can be omitted
- 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

# INTERMEZZO: SAMPLE DATA

## students.xml: list of student information

```
<students>
  <student id="100026">
    <name">Joe Average</name>
    <age>21</age>
    <major>Biology</major>
    <results>
      <result course="Math 101" grade="C-"/>
      <result course="Biology 101" grade="C+"/>
      <result course="Statistics 101" grade="D"/>
    </results>
  </student>
  <student id="100078">
    <name>Jack Doe</name>
    <age>18</age>
    <major>Physics</major>
    <major>XML Science</major>
    <results>
      <result course="Math 101" grade="A"/>
      <result course="XML 101" grade="A-"/>
      <result course="Physics 101" grade="B+"/>
      <result course="XML 102" grade="A"/>
    </results>
  </student>
</students>
```

# DEFINING AND USING FUNCTIONS

- Functions have a name and zero or more parameters
- The body of a function is an XQuery expression, which computes the result

Query:

```
declare function local:grade($g) {
  if ($g="A") then 4.0 else if ($g="A-") then 3.7
  else if ($g="B+") then 3.3 else if ($g="B") then 3.0
  else if ($g="B-") then 2.7 else if ($g="C+") then 2.3
  else if ($g="C") then 2.0 else if ($g="C-") then 1.7
  else if ($g="D+") then 1.3 else if ($g="D") then 1.0
  else if ($g="D-") then 0.7 else 0
};

declare function local:gpa($s) {
  fn:avg(for $g in $s/results/result/@grade return local:grade($g))
};

for $s in fn:doc("students.xml")//student
return <gpa id="{$s/@id}" gpa="{local:gpa($s)}"/>
```

Output:

```
<gpa id="100026" gpa="1.66666666"/>
<gpa id="100078" gpa="3.75"/>
```

# DEFINING AND USING FUNCTIONS (2)

- Functions can be recursive
- Be careful to avoid infinite recursion!

A recursive function that computes the height of the tree rooted at a node:

```
declare function local:height($x) {
  if (fn:empty($x/*)) then 1
  else fn:max(for $y in $x/* return local:height($y))+1
};
```

# TYPES

- XQuery is actually a strongly typed language
- Every item has at runtime an associated type (possibly `xs:anyType` or `xs:anyAtomicType`)
- The XQuery built-in functions all have a type signature
- Whenever possible and reasonable, values are automatically converted into the right type
- A runtime type error is provoked when
  - an actual argument value does not match the declared type
  - a function result value does not match the declared type
  - a valued assigned to a variable does not match the declared type

Some example signature of built-in functions:

```
fn:upper-case($arg as xs:string?) as xs:string
fn:string-join($arg1 as xs:string*, $arg2 as xs:string) as xs:string
fn:root($arg as node()?) as node()?
```

# SEQUENCE TYPES

- The XQuery type system is relatively complicated due to the fact that it is possible to use XML Schema to define new types
- Most XQuery queries only use atomic types or very generic types
- Types can be combined into sequence types using regular expressions

```
2 instance of xs:integer
2 instance of item()
2 instance of xs:integer?
() instance of empty-sequence() () instance of xs:integer*
(1,2,3,4) instance of xs:integer*
(1,2,3,4) instance of xs:integer+
<foo/> instance of item()
<foo/> instance of node()
<foo/> instance of element()
<foo/> instance of element(foo)
<foo bar="baz"/> instance of element(foo)
<foo bar="baz"/>/@bar instance of attribute()
<foo bar="baz"/>/@bar instance of attribute(bar)
fn:doc("recipes.xml")//rcp:ingredient instance of element()+
fn:doc("recipes.xml")//rcp:ingredient
   instance of element(rcp:ingredient)+
```

# TYPE SIGNATURES FOR USER-DEFINED FUNCTIONS

Function:

```
declare function local:grade($g) {
  if ($g="A") then 4.0 else if ($g="A-") then 3.7
  else if ($g="B+") then 3.3 else if ($g="B") then 3.0
  else if ($g="B-") then 2.7 else if ($g="C+") then 2.3
  else if ($g="C") then 2.0 else if ($g="C-") then 1.7
  else if ($g="D+") then 1.3 else if ($g="D") then 1.0
  else if ($g="D-") then 0.7 else 0
};
```

# TYPE SIGNATURES FOR USER-DEFINED FUNCTIONS

This is actually equivalent to:

```
declare function local:grade($g as item()* ) as item()* {
  if ($g="A") then 4.0 else if ($g="A-") then 3.7
  else if ($g="B+") then 3.3 else if ($g="B") then 3.0
  else if ($g="B-") then 2.7 else if ($g="C+") then 2.3
  else if ($g="C") then 2.0 else if ($g="C-") then 1.7
  else if ($g="D+") then 1.3 else if ($g="D") then 1.0
  else if ($g="D-") then 0.7 else 0
};
```

# TYPE SIGNATURES FOR USER-DEFINED FUNCTIONS

But we can also be more precise (useful for debugging):

```
declare function local:grade($g as xs:string ) as xs:decimal {
  if ($g="A") then 4.0 else if ($g="A-") then 3.7
  else if ($g="B+") then 3.3 else if ($g="B") then 3.0
  else if ($g="B-") then 2.7 else if ($g="C+") then 2.3
  else if ($g="C") then 2.0 else if ($g="C-") then 1.7
  else if ($g="D+") then 1.3 else if ($g="D") then 1.0
  else if ($g="D-") then 0.7 else 0
};
```

# ARGUMENT LISTS

- When calling a function, there must be an argument for every parameter specified in the function signature
- If the function does not take any arguments, the parentheses are still required, as in `fn:current-date( )`
- Passing the empty sequence or a zero-length string for an argument is not the same as omitting an argument!

The signature of built-in function op:union:

```
op:union($parameter1 as node()*, $parameter2 as node()*) as node()*
```

Some example function calls:

Ok: `op:union((1, 2), (3)))`
Ok: `op:union((1, 2), 3)`
Ok: `op:union((1, 2), (3,4))`
Not Ok: `op:union((1, 2), 3, 4)`

# ARGUMENT LISTS

- When calling a function, there must be an argument for every parameter specified in the function signature
- If the function does not take any arguments, the parentheses are still required, as in `fn:current-date( )`
- Passing the empty sequence or a zero-length string for an argument is not the same as omitting an argument!

A user-defined function

```
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)
};
```

Some example function calls:

Ok: `local:discountPrice(1, 2, 3)`
Ok: `local:discountPrice(1, (), 3)`
Not Ok: `local:discountPrice(1, ())`

# REFERENCES


- Anders Moller and Micahel Schwartzbach, *An Introduction to XML and Web Technologies*, Addison Wesley, 2005
- Priscilla Walmsley, *XQuery*, O'Reilly, 2007
- 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