Databases 2

7

XML Data Management

XML

- eXtensible Markup Language
- Data representation format proposed by W3C (WWW Consortium) for Web documents, such as:
    - product catalogs,
    - order forms,
    - shared configurations,
    - messages between applications,
    - shared data specifications…

History of XML

- 1986: Standard Generalized Markup Language (SGML) ISO 8879-1986
- ~1991 : “unofficial” HTML 1.0
- Nov. 1995: HTML 2.0 (IETF)
- Gen. 1997: HTML 3.2 (W3C)
- Aug. 1997: XML W3C Working Draft
- Feb 10, 1998: XML 1.0 Recommendation
- Dec. 13, 2001: XML 1.1 W3C Working Draft
- Oct. 15, 2002 : XML 1.1 W3C Candidate Recommendation
- Aug 16, 2006 Extensible Markup Language (XML) 1.0 (Fourth Edition) W3C Recommendation

A «meta-grammar» for defining the grammars of tag-based languages

The Origin of XML

- **Original idea:** a meta-language used to specify markup languages
- **As in HTML** (or in other markup languages ...)
  - XML data are contained in documents
  - data properties are expressed with mark-ups

- **XML was designed to describe data and to focus on what data are**

- **HTML was designed to display data and to focus on how data look like**

HTML vs XML

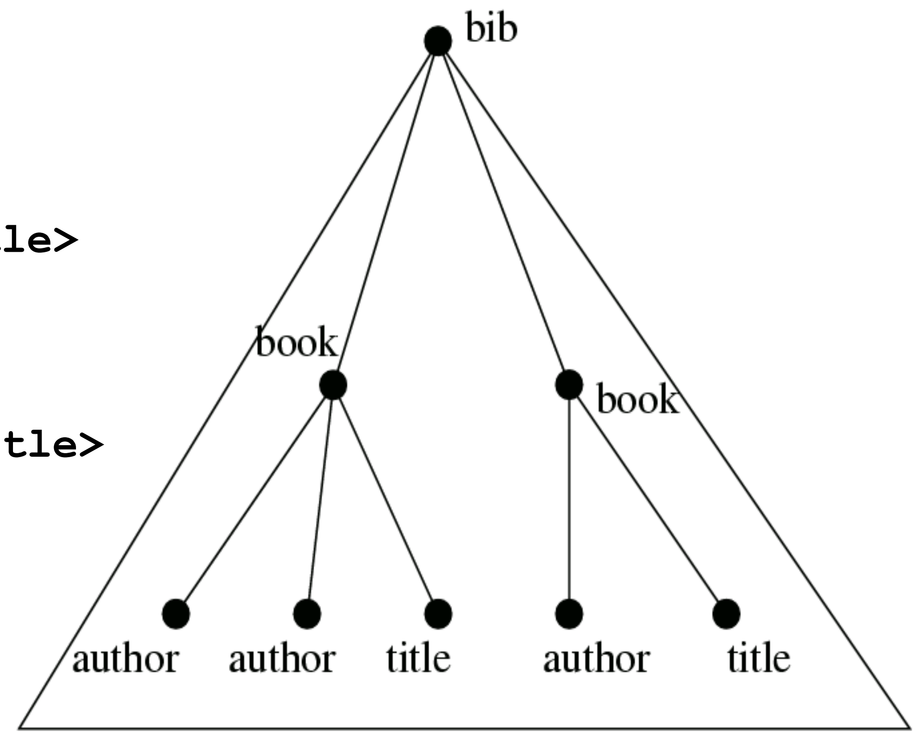
```
<h1>The Idea
    Methodology</h1><br>
<ul>
 <li> by Stefano Ceri,
   Piero Fraternali </li>
 <li> Addison-Wesley </li>
 <li> US$ 49 </li>
</ul>
```

You can **invent** your own tag language !

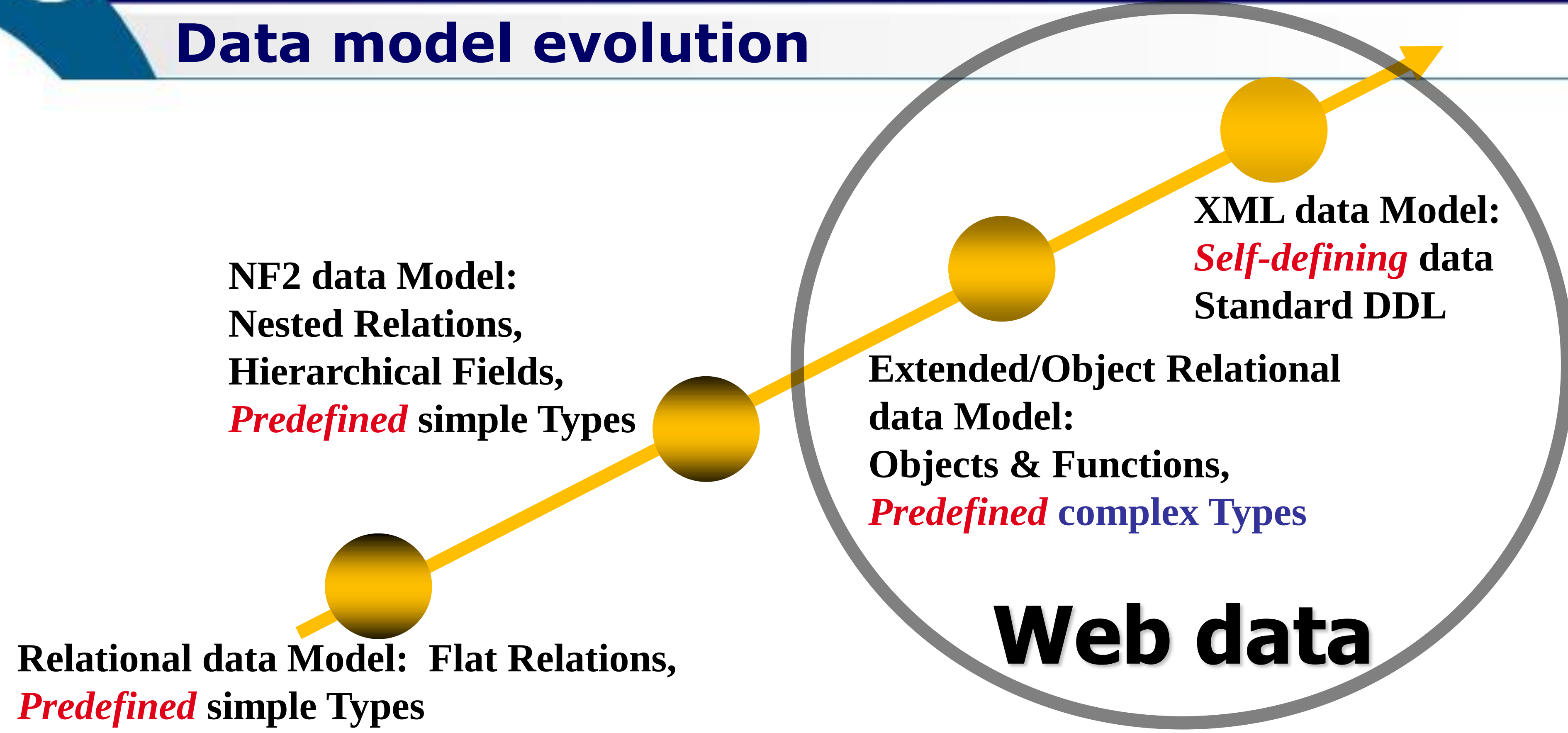
```
<bibliography>
  <book>
    <title> The Idea Methodology </title>
    <author>
      <first> Stefano </first>
      <last> Ceri </last>
    </author>
    <author>
      <first> Piero </first>
      <last> Fraternali </last>
    </author>
    <pub> Addison-Wesley </pub>
    <price> US$ 49 </price>
   </book>
</bibliography>
```

XML documents and their data model

```
<bib>
 <book>
  <author> S. Ceri </author>
  <author> P. Fraternali </author>
  <title> The Idea Methodology </title>
 </book>
 <book>
  <author> R. J. Rolling </author>
  <title> Philosophers' Pottery </title>
 </book>
</bib>
```



Data model evolution



XML has many virtues...

- **Separation of data structure from data representation**
  - Data can be stored in an XML file and the display be governed by a CSS coupled with an HTML page
- **Easier data exchange between heterogeneous systems**
  - XML works as a "transport layer" for Web applications
- **Plain text files can be used to share and store data**
  - XML is a simple standard way to agree on data formats and store them in a hardware- and software-independent way
- **XML documents are self-describing**
  - The "document schema" can be embedded within the doc itself

When XML is used to store data

- **With XML, plain text files can be used to store data**
  - XML data can be stored in simple **text files** or in **databases**
  - Ad-hoc applications can add and retrieve data from the store, generic applications can be used to display the data
- Data management extensions include data models (DTD,XSD), query languages (XPath, XQuery, XSLT, …)
- Data management occurs
  - Within **relational** systems (Oracle, DB2, SQLServer, …)
  - Within **native** systems (eXists,Galax,IPSI-XQ,**BaseX**,...)

First Example

- The syntax rules of XML are very simple (and very strict)
- An example XML document:

```
<?xml version=″1.0″ encoding=″ISO-8859-1″?>
<note>
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
</note>
```

The **root** element

Nested (**child**) elements

XML declaration defines the XML version and the character set (*not part of the document*)

closing tags

Elements

- Whatever is between a pair of corresponding **opening** and **closing** tags is an **XML Element**
- Elements can have different **content types**
  - **element** content
  - simple (**text**) content
  - **mixed** content
  - **empty** content
- An element can also have **attributes**

Second Example

```
<book>
    <title>My First XML</title>
    <prod id="33-657" media="paper"></prod>
    <chapter>Introduction to XML
        <para>What is HTML</para>
        <para>What is XML</para>
    </chapter>
    <chapter>XML Syntax
        <para>Elements must have a closing tag</para>
        <para>Elements must be properly nested</para>
    </chapter>
</book>
```

Element content

Text content (**PCData**)

Empty content

Attributes (*name* = "*value*")

Siblings (same **parent** element)

Mixed content (text and elements)

XML Syntax (basic rules)

- **All XML elements must have opening and closing tags**
  - Not the declaration (not part of the XML document)
- **XML tags are case sensitive**

  <Message>wrong</message> <message>This is correct</message>
- The syntax for **comments** as in HTML: <!-- This is a comment -->
- **Any XML document must have exactly one root element**
- **Elements must be properly nested**
  - All elements can have sub elements (child elements). Sub elements must be **correctly nested** within their parent element:

  <root> <child> <subchild>.....</subchild> </child> </root>
- **Attribute values must always be quoted**

  <incorrectNote date=12/11/2002> <note date="12/11/2002">

Element naming rules

- Names can contain letters, numbers, and other characters
  - must not *start* with numbers or punctuation characters
    - must not start with "xml" (or XML or Xml ...)
  - must not contain spaces
- No words are reserved, but the idea is to have descriptive names
  - Underscore as separator is recommended
    - Examples: <first_name>, <last_name>
- XML documents often have corresponding databases
  - a good practice is to use their naming rules to avoid problems
  - non-English letters (like éòá) are perfectly legal, but risky
- The ":" is reserved to namespaces (more later)

Attributes

- **Attributes are less flexible than elements**
  - cannot contain multiple values
  - not easily expandable (for future changes)
  - cannot contain structured content (child elements can)
  - harder to manipulate by parsers
  - attribute values are harder to test against a DTD (later)
- Attributes are typically devoted to provide information that is **not part of the data** themselves (**meta-data**)

```
<file type="gif">computer.gif</file>
```

Name conflicts

- Since **element names in XML are not predefined**, name conflicts occur whenever different documents use the same names
  - XML data with information **within** a table (an **html table**):
    - <table>
      <tr> <td>Apples</td> <td>Bananas</td> </tr>
      </table>
  - XML data with information **about** a table (a piece of **furniture**):
    - <table>
      <name>African Coffee Table</name>
      <width>80</width> <length>120</length>
      </table>
- If these two XML documents were added together, there would be an element name conflict because both documents contain a <table> element with different content and definition.

Namespaces

- **Name conflicts are solved by prefixes** (*not a revolutionary idea…* ☺)
- The document with information in a table:

```
<h:table>
   <h:tr> <h:td>Apples</h:td> <h:td>Bananas</h:td> </h:tr>
</h:table>
```

- The document with information about a table:

```
<f:table>
   <f:name>African Coffee Table</f:name>
   <f:width>80</f:width> <f:length>120</f:length>
</f:table>
```

- Now there is no name conflict
  - the two documents use different names, <h:table> and <f:table>

The XML Namespace Attribute (xmlns)

- The XML namespace is specified by an attribute:

  xmlns:*namespace-prefix* = "*namespace****URI***"
- The URI is not checked by the parser
  - **URI**: Uniform Resource Identifier (URLs, URNs, …)
  - The only purpose is to give the namespace a unique name
    - <h:**table** xmlns:h="http://www.w3.org/TR/html4/">
<h:tr> <h:td>Apples</h:td> <h:td>Bananas</h:td> </h:tr>
</h:**table**>
    - <f:**table** xmlns:f="http://www.w3schools.com/furniture">
<f:name>African Coffee Table</f:name>
<f:width>80</f:width> <f:length>120</f:length>
</f:**table**>

XML well-formedness and validity

- A document is **well formed** if it has **correct XML syntax**
  - w.r.t. the syntax rules described earlier
- A document is **valid** if it also **conforms to a schema specification**
  - A valid XML document is a well formed document also conforming to the rules of
    - a **Document Type Definition** (**DTD**) or
    - an **XML Schema Definition** (**XSD**)

DTD

- Defines the document structure listing the legal **components** and the legal **containment** relationships
  - Very similar to a regular grammar
  - Included in the XML source file in a DOCTYPE definition:

    <!DOCTYPE root-element [element-declarations]>
- XML documents may <u>contain their own schema specification</u>
  - independent subjects can agree on a common DTD
  - application can use the DTDs to check the validity of the received data (is it corrupted? incomplete?)
  - DTDs also useful to verify *your own* data!

A DTD embedded in the first document

```
<?xml version="1.0"?>
 <!DOCTYPE note
      [ <!ELEMENT note (to,from,heading,body)>
        <!ELEMENT to (#PCdata)>
        <!ELEMENT from (#PCdata)>
        <!ELEMENT heading (#PCdata)>
        <!ELEMENT body (#PCdata)> ]>
 <note>
      <to>Tove</to>
      <from>Jani</from>
      <heading>Reminder</heading>
      <body>Don't forget me this weekend</body>
 </note>
```

Order and **Cardinality** of the child elements

Only **PCdata** content

Declaring elements

- Element declarations have the following syntax:

  <!ELEMENT *element-name* *category* **>**

  or

  <!ELEMENT *element-name* **(** *element-content* **)>**

category : **(#PCdata)** *or* **EMPTY** *or* **ANY**

element-content : an ordered list of element names, with the specification of their cardinality

Element content (list of child elements)

- **Exactly one** occurrence of the element

  <!ELEMENT note (message)>

  - A "message" must occur once within the "note" element
- **At least one** occurrence of the same element

  <!ELEMENT note (message**+**)>

  - The **+** sign declares that "message" must occur one or more times
- **Any number of** occurrences of the same element (possibly zero)

  <!ELEMENT note (message*****)>

  - The ***** sign declares that "message" can occur zero or more times
- **Optional** elements

  <!ELEMENT note (message**?**)>

  - The **?** sign declares that "message" can occur zero or one times

Alternatives and mixed content

- Declaring **either/or content**

  example:<!ELEMENT note (to,from,header,(message | body))>

- The example above declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.

- Declaring **mixed content**

  example:<!ELEMENT note (#PCdata | to | from | header | message)* >

- The example above declares that the "note" element can contain zero or more occurrences of parsed character, "to", "from", "header", or "message" elements.

Declaring attributes

- An attribute declaration has the following syntax:

```
<!ATTLIST element-name attribute-name attribute-type default-value >
```

- Example:

```
<!ELEMENT  payment  EMPTY >
<!ATTLIST  payment  mode  CDATA  "check" >
```

Corresponds to the XML fragment:

```
<payment  mode="cash" />
```

Attribute types

attribute-type can be:

CDATA	The value is character data
(en1|en2|..)	The value must be one from an enumerated list
ID	The value is a unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other ids
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities
NOTATION	The value is a name of a notation
xml:	The value is a predefined xml value

Defaults

default-value can be:

• *a value*	That is the default value if the attribute is not specified
• **#REQUIRED**	The attribute value must be specified
• **#IMPLIED**	The attribute does not have to be included (is optional)
• **#FIXED** *value*	The attribute value is fixed

Declaration of more than one attribute

ATTLIST can specify a ***list*** of attributes for each element

```
<!ELEMENT PRODUCT ( ………… )

<!ATTLIST PRODUCT
   code            ID                       #REQUIRED
   label          CDATA                     #IMPLIED
   status  (available|unavailable)  ‘available’  >
```

An example of DTD

```
<!DOCTYPE NEWSPAPER [
   <!ELEMENT NEWSPAPER (ARTICLE+)>
   <!ELEMENT ARTICLE (HEADLINE,BYLINE,LEAD,BODY,NOTES)>
   <!ELEMENT HEADLINE (#PCDATA)>
   <!ELEMENT BYLINE (#PCDATA)>
   <!ELEMENT LEAD (#PCDATA)>
   <!ELEMENT BODY (#PCDATA)>
   <!ELEMENT NOTES (#PCDATA)>
   <!ATTLIST ARTICLE AUTHOR CDATA #REQUIRED
                     EDITOR CDATA #IMPLIED
                     DATE CDATA #IMPLIED
                     EDITION CDATA #IMPLIED >
 ]>
```

PLACELINE

BYLINE- Name of author (or) journalist's speciality

HEADLINE

How dolphins hear sounds

Special Correspondent (NIE Hindu)

They use two areas of their brains to process noise and create mental images

NEW YORK: Unlike most mammals that primarily process sound in a single area, two areas of the dolphin brain are associated with the **auditory** system, a new research says.

"Dolphins are incredibly intelligent, social animals and yet very little is known about how their brains function," said lead author of the study Gregory Berns, neuroscientist at Emory University in the US. For the study, the researcher mapped for the first time the sensory and motor systems in the brains of dolphins.

"We now have the first picture of the entire dolphin brain and all of the white matter connections inside of it," Berns said. The researchers applied a technique of diffusion tensor imaging (DTI) on the preserved brains of two dolphins, which died after being stranded on a beach in North Carolina more than a decade ago. The study focused on the dolphin auditory system, since dolphins — along with several other animals, such as bats — use echolocation to sense their environments.

"We found that there are probably multiple areas in the dolphin brain associated with auditory information, and the neural pathways look similar to those of a bat," Berns said.

"This is surprising because dolphins and bats are far apart on evolutionary tree. They diverged millions of years ago but their brains may have evolved similar mechanisms for using sound to create mental images," Berns said. IANS

LEAD
- catches reader interest,
- most important info
- introduces topic
- answers most of the 5W's

BODY:
- supporting details

QUOTATION

XML Schema Definition (XSD)

- XSDs are a **richer alternative** to DTDs, **written in XML**
- XSDs define:
  - elements and attributes that can appear in a document
  - which elements are child elements, and in which order
  - the **exact cardinality** of child elements (more powerful)
  - whether an element is empty or can include text
  - default and fixed values for elements and attributes
  - **data types** for elements and attributes
  - … and much more

XSD for the first example

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" …>
<xs:element name="note">
    <xs:complexType>
      <xs:sequence>
            <xs:element name="to" type="xs:string"/>
            <xs:element name="from" type="xs:string"/>
            <xs:element name="heading" type="xs:string"/>
            <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Simple elements

- The syntax for defining a simple element is:

  <xs:element **name**="**xxx**" **type**="**yyy**"/>

- where xxx is the name of the element and yyy is the data type of the element. Here are some XML elements:

  <lastname>Refsnes</lastname>

  <age>34</age>

  <birthdate>1968-03-27</birthdate>

  and here are the corresponding simple element definitions:

  <xs:element name="lastname" type="**xs:string**"/>

  <xs:element name="age" type="**xs:integer**"/>

  <xs:element name="birthdate" type="**xs:date**"/>

many built-in data types

Value restrictions

- This example defines an element called "age" with a restriction. The value of age cannot be lower than 0 or greater than 100:

```
<xs:element name="age">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="100"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>
```

user-defined data types !

Pattern-based constraints

- To limit the content of an XML element to define a series of numbers or letters, we would use the pattern constraint.
- This example defines an element called "letter":
-
```
<xs:element name="letter">
      <xs:simpleType>
            <xs:restriction base="xs:string">
                  <xs:pattern value="[a-z]"/>
            </xs:restriction>
      </xs:simpleType>
</xs:element>
```

Fine-grained domain restictions

- The "letter" element is a simple type with a restriction. The acceptable value is one of the lowercase letters from a to z.

Example of a complex element

- The "employee" element can be declared directly by naming the element, like this:
-
```
<xs:element name="employee">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="firstname" type="xs:string"/>
            <xs:element name="lastname" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
```

Use of externally defined complex types

- The "employee" element can have a type attribute that **refers to the name of a complex type** to use:

```
<xs:element name="employee" type="personinfo"/>
<xs:element name="student" type="personinfo"/>
<xs:complexType name="personinfo">
    <xs:sequence>
        <xs:element name="firstname" type="xs:string"/>
        <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

Reuse of data types !

MaxOccurs and minOccurs

- The <**maxOccurs**> indicator specifies the maximum number of times an element can occur. The <**minOccurs**> indicator specifies the minimum number of times an element can occur:
-
```
<xs:element name="person">  <xs:complexType> <xs:sequence>
        <xs:element name="full_name" type="xs:string"/>
        <xs:element name="child" type="xs:string"
                    maxOccurs="10" minOccurs="0"/>
</xs:sequence>  </xs:complexType> </xs:element>
```

fine-grained cardinalities !

- The example indicates that the "child" element can miss or can occur a maximum of ten times in a "person" element.
- The default value for minOccurs and maxOccurs is 1
- To allow an element to appear an unlimited number of times, use the maxOccurs="unbounded" statement.

Choice

- The <choice> indicator specifies that either one child element or another can occur:
-
```
<xs:element name="person">
  <xs:complexType>
    <xs:choice>
      <xs:element name="employee" type="employee"/>
      <xs:element name="member" type="member"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Alternatives !

In a DTD:

<!Element person **(** employee **|** member **)**>

On schema specification for XML documents

- The schema of a document class can be…
  - Extremely **rigid** and detailed, with few alternatives (or none)
    - The documents in the class are all strictly similar
  - Extremely **flexible**, with many alternatives
    - ANY, mixed content, … documents can exhibit a lot of “diversity”

- XML data are typically **semi-structured data**
  - With a **full spectrum of shades** between the maximum degree of “structuredness” of relational databases to the “total anarchy” of unstructured data (free text, images, …)

DTDs vs. XSDs

- DTDs are quite simple (and limited in their granularity)
  - Also, they are not written in XML
    - Require ad-hoc parsers
  - However, they are **terse and readable**

- XSDs, instead, are very powerful, and parsable by XML parsers
  - But they are extremely **verbose**
  - Many applications do not require that “precision”
  - Also, they are hardly human-readable…

For exams, we prefer DTDs

C. XML (9 p.)

<!ELEMENT **WCC** (Edition+)>
<!ELEMENT **Edition** (Date, Location, Competitor+)>
<!ELEMENT **Competitor** (Name, Cocktail+)>
<!ELEMENT **Cocktail** (Name, Category, Ingredient+, AppearancePoints, TastePoints, OverallScore)>

The DTD above (in which unspecified elements only contain PCData) describes the results of the World Cocktail Championships, organized by the International Bartender Association, where competitors present originally conceived cocktails, and a professional jury grades their creations w.r.t their originality, appearance, and taste. Extract in XQuery:

(3 p.) The name of the creator of the "ugliest" cocktail ever presented in the "Daiquiri" category (*w.r.t. the appearance score*).

(3 p.) For each competitor, their best creation (w.r.t. the overall score) for each category in which they have competed.

Query Languages for XML

(XPath, XQuery)

Query languages for XML

- A set of XML documents can be considered a data collection
- Query languages
  - for extracting relevant information from XML documents
  - and transforming it into new documents
- The languages are:
  - **XPath**: a simple document selection language
  - **XQuery**: a rich query language
  - **XSLT**: especially used for document transformations (e.g. Producing HTML from XML) – not discussed in this course

XPath, XQuery: many implementations exist

Just to mention one:

BaseX (8.6.7, as of October 2017)

http://basex.org/

Playing with the language is **fundamental** to become sufficiently quick and confident

Data model

- Instances of the data model:
  - **Ordered Sequences of** (zero or more) **XML *items***
    - The empty sequence is often considered as the “null value”
- **XML items**: atomic **values** or **nodes**
- **Atomic values**:
  - instances of all XML Schema atomic types
    - string, boolean, ID, IDREF, decimal, QName, URI, ...
  - untyped atomic values
- **Nodes**:
  - document | element | attribute | text | namespaces | PI | comment
- Nodes and values can be **typed** (i.e. schema validated) or **untyped** (i.e. non schema validated)

This must be clear

- At the core of the **data model**:
  - **Ordered sequences of XML items**
- **Query languages**:
  - XPath
    - **Extracting sequences** of items from existing documents
  - XQuery
    - Combining *existing* and *new* items into **new sequences**
- They both work as **ALGEBRAS of SEQUENCES**
  - Input: **Item Sequences** Output: **Item Sequences**

XML Items: Atomic values and Nodes

- **Atomic values** are values of
  - the 19 atomic types available in XML Schema
    - E.g. xs:integer, xs:boolean, xs:date
  - All the user defined derived atomic types
    - e.g.: myNS:ShoeSize, xdt:untypedAtomic
  - Atomic values carry their type together with the value
    - (8, myNS:ShoeSize) is not the same as (8, xs:integer)
- **Nodes**: 7 types:
  document | element | attribute | text | namespaces | PI | comment
  - Every node has a unique **node identifier**
  - Nodes can be nested into one another (→ conceptual "tree")
  - Nodes are **topologically ordered** in the tree (**document order**)

Sequences

- Can be heterogeneous (nodes and atomic values)
  - (<a/>, 3)
- Can contain duplicates (by value and by identity)
  - (1,1,1)
- Are not necessarily ordered in document order
- Nested sequences are automatically flattened
  - (1, 2, (3, 4)) = (1, 2, 3, 4)
- Single items and singleton sequences are the same
  - 1 = (1)

Comparisons

- Comparison predicates may behave differently based on
  - The cardinality of operands (semantics for **set comparison**)
  - The type of operands

Value	For comparing single values	`eq, ne, le, lt, gt, ge`
General	Existential quantification + automatic type coercion	`=, !=, <=, <, >, >=`
Node	Testing identity of single nodes	`is, isnot`
Order	Testing relative position of one node w.r.t. another (in document order)	`<<, >>`

Values and comparisons

<a>text eq "text"	true	*two strings*
<a>42 eq 42	type error	*string and number*
<a>42 eq "42"	true	*two strings*
<a>42 eq 42.0	type error	*string and float*
<a>42 eq 42	true	*two strings*
<a>42 eq 42	false	*there is an extra " "*
<a>baz eq 42	type error	*string and number*
ns:shoesize(5) eq ns:hatsize(5)	true	*same base type*
<a>42 = 42	true	*string → integer*
<a>42 = 42.0	true	*string → int → float*
() = 42	false	*existential*
(<a>42,43) = 42	true	*existential*
(1,2) = (2,3)	true	*existential*
(1,2) != (1,2)	true	*existential*

Functions

- *XPath and XQuery Functions and Operators* (**F&O**)
  - a W3C specification of many useful functions & operators
    - doc(xs:anyURI) => document?
    - empty(item*) => boolean
    - index-of(item*, item) => xs:unsignedInt* (*all occurrences*)
    - distinct-values(item*) => atomic-value*
    - distinct-nodes(node*) => node*
    - union(node*, node*) => node*
    - add-date(xs:date, xs:duration) => xs:date
    - string-length(xs:string?) => xs:integer?
    - contains(xs:string, xs:string) => xs:boolean
    - ...

Combining sequences

- **union**, **intersect**, **except**
  - Work <u>only for</u> sequences of <u>nodes</u>, not atomic values
  - Eliminate duplicates and reorder to document order

$x := <a/>, $y := <b/>, $z := <c/>

($x, $y) union ($y, $z) => (<a/>, <b/>, <c/>)

- **F&O** provides many useful functions & operators for the manipulation of sequences
  - particularly useful: xf:distinct-values(), xf:distinct-nodes()

```
<bookstore>
    <book available='Y'>
        <title>The Jungle Book</title>
        <author>R. Kipling</author>
        <date>1894</date>
        <ISBN>88-452-9005-0</ISBN>
        <publisher>Macmillan</publisher>
    </book>
    <book available='Y'>
        <title>Il nome della rosa</title>
        <author>U. Eco</author>
        <date>1980</date>
        <ISBN>55-344-2345-1</ISBN>
        <publisher>Bompiani</publisher>
    </book>
    <book available='N'>
        <title>Alice in Wonderland</title>
        <author>L. Carroll</author>
        <date>1865</date>
        <ISBN>88-07-81133-2</ISBN>
        <publisher>Macmillan</publisher>
    </book>
</bookstore>
```

Path expressions

```
doc("bib.xml")/bookstore/book
```

```
<book available='Y'>
    <title>The Jungle Book</title>
    <author>R. Kipling</author>
    <date>1894</date>
    <ISBN>88-452-9005-0</ISBN>
    <publisher>Macmillan</publisher>
</book>
<book available='Y'>
    <title>Il nome della rosa</title>
    <author>U. Eco</author>
    <date>1980</date>
    <ISBN>55-344-2345-1</ISBN>
    <publisher>Bompiani</publisher>
</book>
<book available='N'>
    <title>Alice in Wonderland</title>
    <author>L. Carroll</author>
    <date>1865</date>
    <ISBN>88-07-81133-2</ISBN>
    <publisher>Macmillan</publisher>
</book>
```

Path expressions

- Path expressions typically start from the root of documents
  - e.g. doc("books.xml") returns the document node that in turn contains the root element (and all its content)
- Starting from the root, it is possible to express path expressions to "reach" and extract the desired content

  doc("books.xml")/bookstore/book

  returns **the sequence** of all <book>s in the document

- Path expressions define a "**path**" through the document nodes
  - the **current node** changes as the expressions are evaluated "**one step at a time**"

XPath

Path expressions are made of **steps**. The most common steps are:

Expression	**Description**
Nodename	Selects all child nodes of the current node
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects **the current node**
..	Selects the parent of the current node
@	Qualifies attribute names

Conditions

- doc(“books.xml”)/bookstore/book[./publisher=‘Macmillan’]/title

    Returns the sequence of all titles of books published by Macmillan

  Equivalent to:
    doc(“books.xml”)/bookstore/book[publisher=‘Macmillan’]/title

```
<title>The Jungle Book</title>
<title>Alice in Wonderland</title>
```

Descendants

- doc(“books.xml”)//author

Returns the sequence of all authors in the document, independently of their nesting level

```
<author>R. Kipling</author>
<author>U. Eco</author>
<author>L. Carroll</author>
```

Ordered access

- doc (“books.xml”)/bookstore/book[2]

    Returns the second book in the document

```
<book available='Y'>
  <title>Il nome della rosa</title>
  <author>U. Eco</author>
  <date>1980</date>
  <ISBN>55-344-2345-1</ISBN>
  <publisher>Bompiani</publisher>
</book>
```

Wildcards

- doc (“books.xml”)/bookstore/book[2]/*

  Returns all the elements (* = with any tagname) contained into the second book

```
<title>Il nome della rosa</title>
<author>U. Eco</author>
<date>1980</date>
<ISBN>55-344-2345-1</ISBN>
<publisher>Bompiani</publisher>
```

More XPath examples

/bookstore	Selects the root element bookstore (if the path starts with a slash (/) it always represents an absolute path to an element)
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

XPath filter predicates

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets

/bookstore/book[1]	Selects the first book element that is the child of the bookstore element
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element

XPath filter predicates

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets

//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='eng']	Selects all the title elements that have an attribute named lang with a value of 'eng'
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element having a price element with a value greater than 35.00

XPath filter predicates

- [ ] is an overloaded operator
- Filtering by position (if numeric value) :
  - /book[3]
  - /book[3]/author[1]
  - /book[3]/author[2 to 4]
- Filtering by predicate :
  - //book[author/firstname = “ronald”]
  - //book[@price <25]
  - //book[count(author[@gender=“female”])>0]
- Existential filtering:
  - //book[author]   (equivalently //book[./author])

Wildcards

XPath wildcards can be used to select unknown XML elements

Wildcard	Description
*	Matches any element node
@*	Matches any attribute node
node()	Matches any node of any kind

Path	Expression Result
/bookstore/*	Selects all the children of the bookstore element
//*	Selects all elements in the document
//title[@*]	Selects all title elements which have any attribute

Alternative paths

The | operator indicates alternative paths to reach the results

- //book/title | //book/price
  Selects all the **title** and the **price** elements of all books
- //title | //price
  Selects all the **title** and the **price** elements in the document
- /bookstore/book/title | //price
  Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

Composition semantics of XPath expressions

- Expression: ***expr1 / expr2***
- Semantics:
  1. Evaluate *expr1* => get a sequence S1 of nodes
  2. Bind the current node (**.**) to each node in S1
  3. Evaluate *expr2* in the context of each different binding for **.**
     => get as many sequences $S2_i$ of nodes in result
  4. Concatenate the partial sequences $S2_i$ into sequence S2
  5. (Eliminate duplicates)
  6. (Sort by document order)
- A standalone step is an expression
  1. step = (axis, nodeTest) where
     nodeTest = (node kind, node name, node type)

XPath Axes

- **An axis defines a node-set relative to the current node (.)**
  - ancestor: selects all ancestors (parent, grandparent, etc.) of the current node
  - ancestor-or-self: selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself
  - attribute: selects all attributes of the current node
  - child: selects all children of the current node
  - descendant: selects all descendants (children, grandchildren, etc.) of the current node

XPath Axes

- descendant-or-self: selects all descendants of the current node **and** the current node itself
- following: selects everything in the document after the closing tag of the current node
- following-sibling: selects all siblings after the current node
- parent: selects the parent of the current node
- preceding: selects everything in the document that is before the start tag of the current node
- preceding-sibling: selects all siblings before the current node
- **self**: selects the current node **(.)**

Abbreviated syntax

- Axis can be missing: by default the *child* axis is implied
    - ./person → ./child::person
- Common short-hands for frequently used axes
    - Descendent-or-self
        - .//name → ./descendant-or-self::*/child::name
    - Parent
        - ./.. → ./parent::*
    - Attribute
        - ./@year → ./attribute::year
    - Self
        - ./. → ./self::*

XQuery

- **XQuery is to XML what SQL is to relational databases**
- **XQuery is designed to query XML data - not just XML files, but anything that can appear as XML, including databases**
- XQuery is defined by the W3C and builds on XPath
- XQuery is supported by all the major database engines (IBM, Oracle, Microsoft, etc.)
- XQuery is a W3C standard - and developers who correctly implement its specs are guaranteed that the code will work among different products

XQuery type system

- XQuery has a powerful (and complex!) type system
- Types are imported from XML Schemas
- Every XQuery expression has a **static** type
- Every XML data model instance has a **dynamic** type
- Goals of the type system:

1. statically detect errors in the queries
2. infer the type of the result of valid queries
3. ensure statically that if the input dataset is guaranteed to be of a given type then the result will be of a given (expected) type

XQuery Structure

- An XQuery basic structure:
  - **Prologue + XQuery expression**
- Role of the prologue:
  - Populates the context where expressions are compiled and evaluated
  - Contains:
    - namespace definitions
    - schema imports
    - default element and function namespace
    - **function definitions**
    - function library imports
    - global and external variables definitions
    - etc.

User-defined functions

- In-place defined XQuery functions (in the prologue) :

declare function *name(formal_params)* ***as*** *returnedType* ***{***
    XQuery expression
}

  - Functions can be recursive and mutually recursive

XQuery expressions

XQuery **expr** := Constant | Variable | Path-Expr |
Comparison-Expr | Arithmetic-Expr |
Logical-Expr | Constructor-Expr |
Conditional-Expr | FunctionCall |
FLWOR-Expr |
Quantified-Expr | TypeSwitch-Expr |
Instanceof-Expr | Cast-Expr | Union-Expr |
IntersectExcept-Expr | Validate-Expr

Expressions can be nested with full generality!

Constants

- The XQuery grammar has built-in support for:
  - Strings: “125.0” or ‘125.0’
  - Integers: 150
  - Decimal: 125.0
  - Double: 125.e2
  - ...all atomic types available in XML Schema
  - Values can be constructed by
    - constructors in F&O doc: xf:true(), xf:date(“2002-5-20”)
    - casting
    - schema validation

Variables

- $ + VariableName
- **Bound** to the result of the evaluation of an expression
- The **binding** is created by *let*, *for*, *some, every* clauses, or by *type-switch* expressions, or by coupling of function parameters
- There is **NO ASSIGNMENT** (variables aren't memory locations)
- example of binding by a let clause:

```
let $x := ( 1, 2, 3 )
return count($x)
```

  - Here, the scope of $x ends at the end of the return expression

Arithmetic expressions

- Apply the following rules:
  - atomize all operands
    - if every item in the input sequence is either an atomic value or a node whose typed value is a sequence of atomic values, then return it
  - if an operand is untyped, cast to xs:double (if unable, => error)
  - if the operand types differ but can be promoted to common type, do so (e.g.: xs:integer can be promoted to xs:decimal)
  - if operator is consistent w.r.t. types, apply it; result is either atomic value or error
  - otherwise, throw type exception
- Examples:
  - -1 - (4 * 8.5)
  - $b mod 10
  - <a>42 + 1

Logical expressions

- They are:
  - expr1 **and** expr2 (**not()** *is implemented as a function*)
  - expr1 **or** expr2
- Returns *true* / *false* (in 2-value logic, not 3-value logic like SQL!)
- Rules:
  - first compute the **Boolean Effective Value** (BEV) of operands:
    - if (), “”, 0, zero length string then return false
    - if the operand is of type boolean, its BEV is its value;
    - else return true
  - then use standard 2-value Boolean logic on the operands’ BEVs
- (false) and (error) => either false or error!
  - non-deterministic: it is impossible to foresee the result!!

Construction expressions

- <newtag> text content shown as is </newtag>
- <newtag>
      { doc(...)//book[2]/author **}**
  </newtag>
- Braces "**{}**" are used to delineate **evaluated** contents

  <constructionExample>
      count((1,2,2,3,34)) = **{** count((1,2,2,3,34)) **}** !!
  </constructionExample>

Conditional expressions

- Syntax: **if (** *expr1* **)**
  then *expr2*
  else *expr3* ← not optional (**else ()**)

- if ($book/@year < 1980)
  then (<old-book> { $book/title } </old-book>)
  else (<new-book> { $book/title } </new-book>)

Often used in function definitions

Function Calls

- In addition to the functions listed in F&O, any function defined in the prologue can be invoked in an XQuery expression

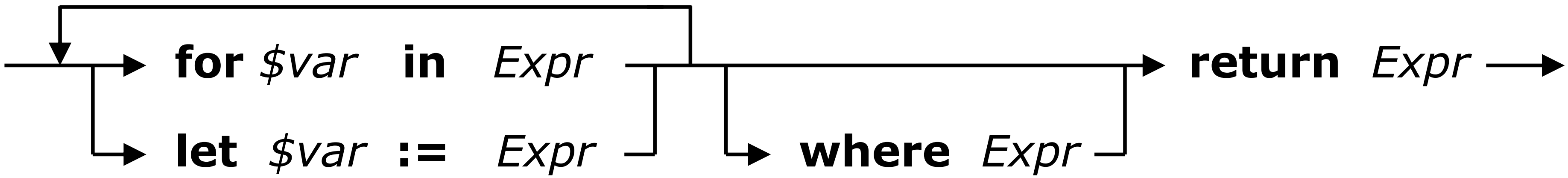
```
declare function ns:foo($x as xs:integer) as element() {
        <a>{ $x+1 } </a>
}

declare function ns:fibo($n as xs:integer) as xs:integer {
        if ( $n = 0  or  $n = 1 )
        then ( 1 )
        else ( fibo( $n-1 ) + fibo( $n – 2 ) )
}
```

FLWOR expressions

- 5 clauses:
  - **F**OR
  - **L**ET
  - **W**HERE
  - (ORDER BY)
  - **R**ETURN

FLWR expressions

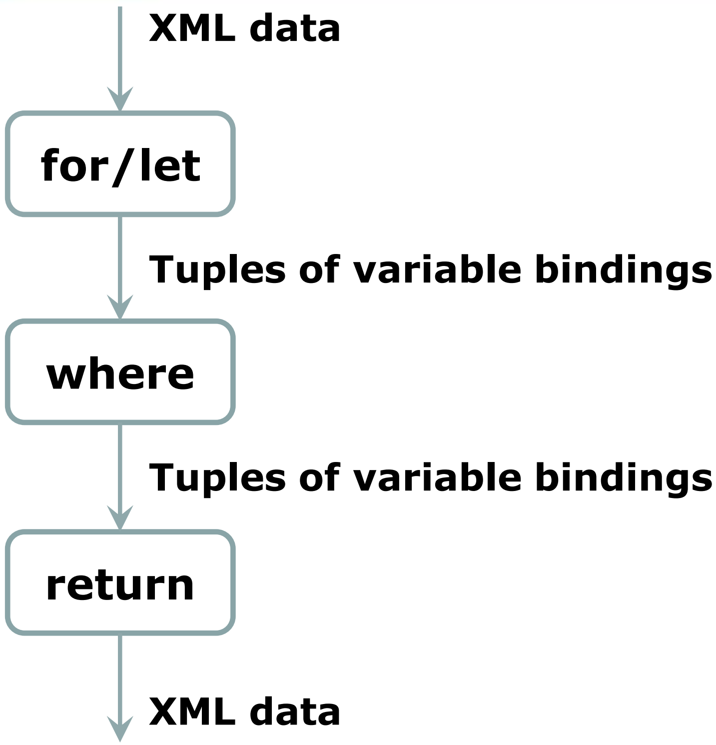
- Syntax for the interleaving of F, L, W and R clauses



- Example

```
for $x in /bib/book              (: similar to the SQL from :)
let $y := $x/author              (: no analogy in SQL :)
where $x/title = "Data on the Web"
                                 (: similar to the WHERE from :)
return count( $y )               (: similar to the SELECT from :)
```

FLWR expressions: evaluation



- **for** : **iteration** + "**individual**" bindings
  - Every item in a sequence generates a different binding (whose value is that item)
- **let** : "**collective**" bindings
  - A sequence of items is collectively bound to one variable (whose value is the whole sequence)
- **where** : filtering expressions
  - Independently evaluated on each tuple of bindings to keep or discard it
- **return** : construct results
  - Executed once for each tuple of bindings

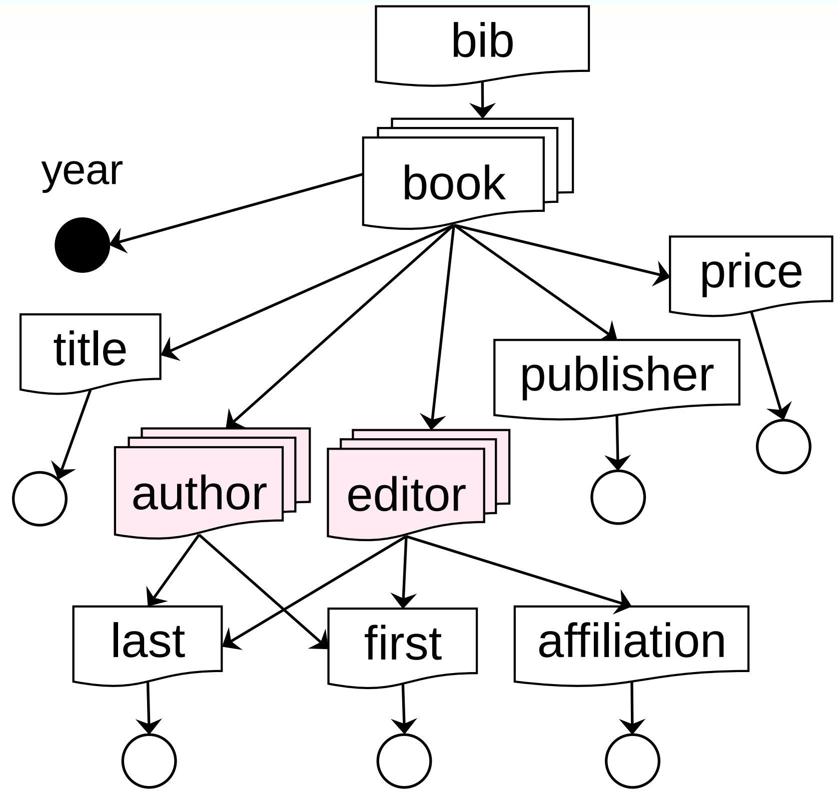
Running example

```xml
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>The Economics of Technology and Content for Digital TV</title>
    <editor><last>Gerbarg</last><first>Darcy</first><affiliation>CITI</affiliation></editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
   </book>
</bib>
```

Running example

```
<!ELEMENT bib       ( book* ) >
<!ELEMENT book      (  title,
                       ( author+ | editor+ ),
                       publisher,
                       price          ) >
<!ATTLIST book      year  CDATA  #REQUIRED >
<!ELEMENT author ( last, first ) >
<!ELEMENT editor   ( last, first, affiliation ) >

<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```



Simple iteration expressions

- **for** *variable* **in** *expression1*
  return *expression2*
- Example: **for** $x **in** doc("bib.xml")/bib/book
  return $x/title

- Semantics :
  - iteratively bind the *variable* to each root node of the forest returned by *expression1*
    - for each such binding, evaluate *expression2*
  - concatenate the resulting sequences
  - As usual, nested sequences are automatically flattened

Nested FOR clauses (cartesian product)

- FOR expressions can be nested (**independent** expressions):

```
for $book in doc("books.xml")//book
   for $author in doc("books.xml")//author
 return $author
```

  - iteratively binds $book to each <book> element returned by doc("books.xml")//book
  - <u>for each such binding</u>, evaluates doc("books.xml")//author
  - For each couple of resulting bindings {$book,$author} evaluate the expression in the return clause
  - All authors are returned 3 times → 12 authors in the result

Nested FOR clauses (structural join)

- FOR expressions can be nested (**dependent** expressions):

```
for $book in doc("books.xml")//book
    for $author in $book/author
 return $author
```

- ...<u>for each such binding</u>, evaluates **$book**/author
  - This time, the "inner" expression is dependent of the variable bound in the outer clause
  - Each author is only returned in the iteration relative to its containing book

    → the 4 authors are returned only once, in document order

LET clauses

- **let** *variable* **:=** *expression1*
  return *expression2*
- Example : **let** $x **:=** doc("bib.xml")/bib/book
  return count($x)
- Semantics:
  - binds the *variable* to the result of the *expression1* (the entire sequence is bound to the variable)
  - add this binding to the current environment
  - evaluate and return *expression2*
    - The return clause is evaluated <u>only once</u>

Equivalence of expressions

- The example with nested for can be also expressed as:

```
for $book in doc("books.xml") //book
  let $a :=  $book/author           (: $a bound to the entire sequence
                                       of each book 's authors :)
return $a
```

- and also as

```
doc("books.xml")/bib/book/author
```

- The example with let can be also expressed as:

```
count(doc("books.xml")/bib/book)
```

- All this is due to the automatic flattening of nested sequences

FOR vs LET

A simple example may further clarify the difference

- let $x :=doc("bib.xml")/bib/book
  return count($x)

  → **3**

- for $x in doc("bib.xml")/bib/book
  return count($x)

  → (**1**, **1**, **1**)

WHERE clause

- WHERE clauses filter out the tuples of variable bindings generated by the previous let/for clauses
- Syntax: **where** *expression*
- Semantics: calculate the **BEV** of *expression* in the context of each tuple, and only tuples satisfying the clause are further considered
- Example:

```
for $book in doc ("books.xml")//book
where $book/publisher="Macmillan"
    and $book/@available="Y"
return $book
```

- Equivalent to:
  doc("books.xml")//book[publisher="Macmillan" and @available="Y"]

RETURN clause

- Syntax: **return** *expression*
- Semantics: evaluate the expression in the context of each “survived” tuple of variable bindings, and return it
  - concatenating all the results into one flattened sequence, as usual
- Each evaluation can return
  - A node (or tree)
  - An ordered “forest” (of nodes or trees)
  - A textual value (PCdata)
- RETURN clauses can contain **node constructors**

Node constructors within a RETURN clause

- ```
  return <result>
         </result>
  ```
  literal text content
- ```
  return  <result>
              some content here {$x/name} and some more here
          </result>
  ```
- ```
  return <result>
              { $x/title } <firstnames>
                                { $x/author/first }
                           </firstnames>
         </result>
  ```
  can be nested
  ```

A FWR query

```
for $book in doc("books.xml")//book
where $book/price>60
return  <expensiveBook>
            { $book/title }
        </expensiveBook>
```

```
<expensiveBook>
   <title>TCP/IP Illustrated</title>
</expensiveBook>
<expensiveBook>
   <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBook>
```

Similarly, with use of *text()*

```
for $book in doc("books.xml")//book
where $book/price>60
return  <expensiveBook>
               {  $book/title/text()  }
          </expensiveBook>
```

<expensiveBook>TCP/IP Illustrated</expensiveBook>

<expensiveBook>The Economics of Technology and Content for Digital TV</expensiveBook>

Similarly, with a LET clause

```
let $books := doc("books.xml")//book[price>60]
return  <expensiveBooks>
            {  $books/title }
        </expensiveBooks>
```

```
<expensiveBooks>
   <title>TCP/IP Illustrated</title>
   <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBooks>
```

Not-so-similarly, with LET (quite strange)

```
let $books := doc("books.xml")//book
where $books/price > 60        (: existential... :)
return  <expensiveBooks>
            {  $books/title  }
        </expensiveBooks>
```

price is 40$

```
<expensiveBooks>
   <title>TCP/IP Illustrated</title>
   <title>Data on the Web</title>
   <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBooks>
```

And… in the most compact form

```
<expensiveBooks>
    { doc("books.xml")//book[price>60]/title }
</expensiveBooks>
```

```
<expensiveBooks>
    <title>TCP/IP Illustrated</title>
    <title>The Economics of Technology and Content for Digital TV</title>
</expensiveBooks>
```

“grouping”: count() and LET clauses

Extract the publishers of more than 50 books

```
for $p in doc(“books.xml”)//publisher
    let $books :=  doc(“books.xml”)//book[publisher = $p]
where count($books) > 50
return <prolificPublisher> { $p/text() } </prolificPublisher>
```

- **Each publisher is returned 50 times or more!!**

distinct-values()

```
for $p in  distinct-values( doc("books.xml")//publisher )
   let $book :=  doc("books.xml")//book[publisher = $p]
where count($book) > 50
return <prolificPublisher> { $p } </prolificPublisher>
```

- Each publisher is now returned only once (and the number of "iterations" due to the for clause is dramatically reduced)
- N.B: text() cannot be applied to $e in this case: $e is NOT a reference to an XML element, but a (string) value
- distinct-values() returns the <u>distinct textual values</u> of the items in the sequence given as argument, also for structured elements

distinct-values()

For each Publisher, list of the titles of their published books

```
for $p in  distinct-values( doc(“books.xml”)//publisher )
   let $books :=  doc(“books.xml”)//book[publisher = $p]
where count($book) > 50
return <Publisher>
               <Name>{ $p } <Name>
               <PublishedBooks>
                    { $books/title }
               </PublishedBooks>
         </Publisher>
```

Once more: FOR vs LET

- **FLWR expression**:

```
for $b in //book
  for $a in $b/author
return count($a)
```

- **Returns**:

(1, 1, 1, 1)

Once more: FOR vs LET

- **FLWR expression**:

```
for $b in //book
  let $a := $b/author
return count($a)
```

- **Returns**:

(1, 3, 0)

FLWR vs conditional expressions

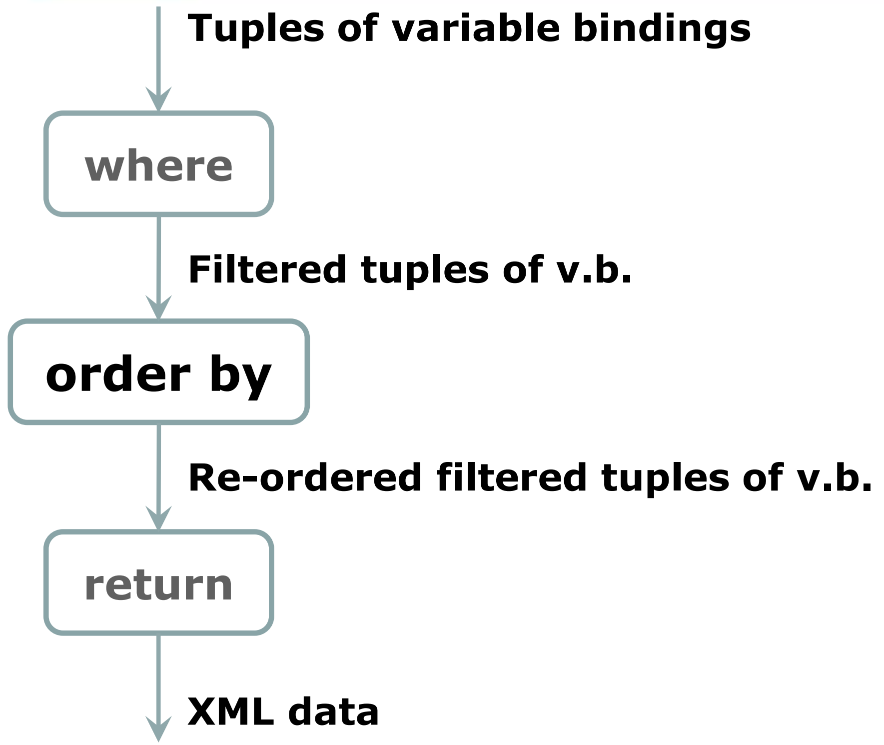
- **FLWR expression**:

```
for $b in //book
  let $a := $x/author
where $b/title = “Ulysses”
return count($a)
```

- **Equivalent to**:

```
for $b in //bib/book
return (  let $a := $x/author
              return if ( $b/title=“Ulysses” )
                         then count($y)
                          else ()                 )
```

The ORDER BY clause



- Intercepts the bindings between filtering and the execution of the return clause
- Alters the order in which tuples are processed, and therefore the final result

Order by

```
for $book in doc(“books.xml”)//book
order by $book/title
return <book>
          { $book/title, $book/publisher }
       </book>
```

```
for $b in doc("bib.xml")//book
where $b/publisher = “Springer Verlag"
order by $b/@year
return $b/title
```

The ordering criteria are independent of the data in output (different from SQL). Any expression is allowed.

Quantified expressions

- **Existential**

  some $*var* **in** *expr1* **satisfies** *expr2*

  - True iff the BEV of *expr2* is true for **at least one** binding of $*var* iterated ("for-like") over the nodes returned by *expr1*

- **Universal**

  every $*var* **in** *expr1* **satisfies** *expr2*

  - True iff the BEV of *expression2* is true for **all** the bindings of *$var* iterated ("for-like") over the nodes returned by *expr1*

- In both cases, the **scope** of $*var* ends at the end of *expr2*

More recently added

- Updates
  - W3C Recommendation 17 March 2011

```
do insert
        <user_tuple>
                <userid>U07</userid>
                <name>Annabel Lee</name>
        </user_tuple>
into doc("users.xml")/users
```

- Xquery 3.0 – W3C Recommendation 8 April 2014
  - Group by clause, count clause, window clause, allowing empty (outer joins functionalities), try-catch expressions, switch expressions, ...

XML Data Management Technology

- Two families of XML storage systems:
- **Native XML databases**
  - Use XML-specific storage management
  - Support only XML query languages
- **Relational databases with XML support**
  - Based on relational storage systems which are suitably extended
  - Integrate SQL with XQuery

Native XML databases

- Use data models which are not relational and can be standard or proprietary
  - ES: DOM, XPath Data Model, XML Information Set
- Use proprietary physical data stores, which are document-centric
- Organize databases as document collections
- Examples: Tamino, Xyleme, eXist, Galax, ...

Relational DBMSs with XML support

- Use relational data stores heavily, but support as well native XML data
- When they map XML data to relational structures, may have rather different mapping to relational schemas:
  - Fixed and DTD independent (canonic)
  - Variable and DTD specific (custom)
  - Decided by the physical DB optimizer (e.g. with mapping rules)