Advanced Databases

XML : eXtensible Markup Language

Prof. dr. J. Paredaens
mmv. M. Mampaey
TU/e

Table of Contents

1. Introduction to XML
2. XPath
3. XQuery
4. Typing in XQuery
5. Document Type Definitions
6. XML Schema
7. Light XQuery
8. XSLT

References

I. XML

[1] www.w3.org/TR/xmlschema-0

[2] www.w3.org/TR/xmlschema-1

[3] www.w3.org/TR/xpath

[4] P.M. Lewis, A. Bernstein, M. Kifer, Databases and Transaction Processing, Addison Wesley, Chapter 17, 2002

[5] D. Chamberlin, XQuery, a query language for XML, Web, June 2003

[6] www.w3.org/TR/xquery

[7] www.w3.org/XML/Query

References

[8] www.w3.org/TR/xslt20

[9] Draper et al, XQuery 1.0 and XPath 2.0 Formal Semantics, www.w3.org/TR/xquery-semantics/, 2003

[10] M. Brundage, XQuery, Add. Wesley, 2004

[11] J. McGovern, P. Bothner, K. Cagle, J. Linn, V. Nagarajan, XQuery, Sams Publ., 2004

[12] H. Katz, XQuery from the Experts, Add. Wesley, 2004

1. Introduction to XML [4]

- Web data for human consumption
  - HTML (Cfr. next slide)
  - Self describing: attribute names are included
    - but not explicitly separated from data values
- Web data for machine consumption
  - Characteristics of Semistructured Data:
    - object-like
    - schemaless
    - self-describing
  - XML
    - (optional) structure descr.: DTD, XML Schema

HTML-document

```
<html>
  <head><Title>Student List</Title></head>
  <body>
    <h1>ListName: Students</h1>
    <dl>
      <dt>Name: Jan Vijs
        <dd>Id: 11
        <dd>Address:
          <ul>
          <li>Number: 123
          <li>Street: Turnstreet
          </ul>
      <dt>Name: Jan De Moor
        <dd>Id: 66
        <dd>Address:
          <ul>
          <li>Number: 4
          <li>Street: Hole Rd
          </ul>
    </dl>
  </body>
</html>
```

- Why is XML important?
  – simple open non-proprietary widely accepted data exchange format
- XML is like HTML but
  – no fixed set of tags
    - X = “extensible”
  – no fixed semantics (c.q. representation) of tags
    - representation determined by separate ‘stylesheet’
    - semantics determined by application
  – no fixed structure
    - user-defined schemas

XML-document – Running example 1

```
<?xml version ="1.0"?>
<PersonList Type="Student" Date="2004-12-12">
  <Title Value="Student List"/>
  <Contents>
    <Person>
      <Name>Jan Vijs</Name>
      <Id>11</Id>
      <Address>
        <Number>123</Number>
        <Street>Turnstreet</Street>
      </Address>
    </Person>
    <Person>
      <Id>66</Id>
      <Address>
        <Number>4</Number>
        <Street>Hole Rd</Street>
      </Address>
    </Person>
  </Contents>
</PersonList>
```

- Global structure
  - First line is mandatory;
  - Tags are chosen by author;
  - Opening tag must have a matching closing tag;
    `<a><b></b><c></c></a>`
  - Only one root element `PersonList`;
  - `<a> … </a>`; **`a`** is the name of the element, content, child, descendant, parent, ancestor, sibling;
  - `<PersonList Type="Student">` **`Type`** is name of the attribute of element **`PersonList`**; the value of the attribute is **`"Student"`** ; all attribute values must be quoted;

- empty elements: `<Title Value="Student List"> </Title>`
  `<Title Value="Student List"/>`
- processing instruction: `<? . . . ?>`
- comment: `<!-- here we go -->`
- mixed data-text:

```
<Address>
  Jan lives in <Street> Q Street </Street> number
  <Number>123</Number>
</Address>
```

- elements are ordered:

```
<Address>                          <Address>
  <Number>123</Number>               <Street> Q Street </Street>
  <Street> Q Street </Street>        <Number>123</Number>
</Address>                         </Address>
```

  are different
- weak facilities for constraints

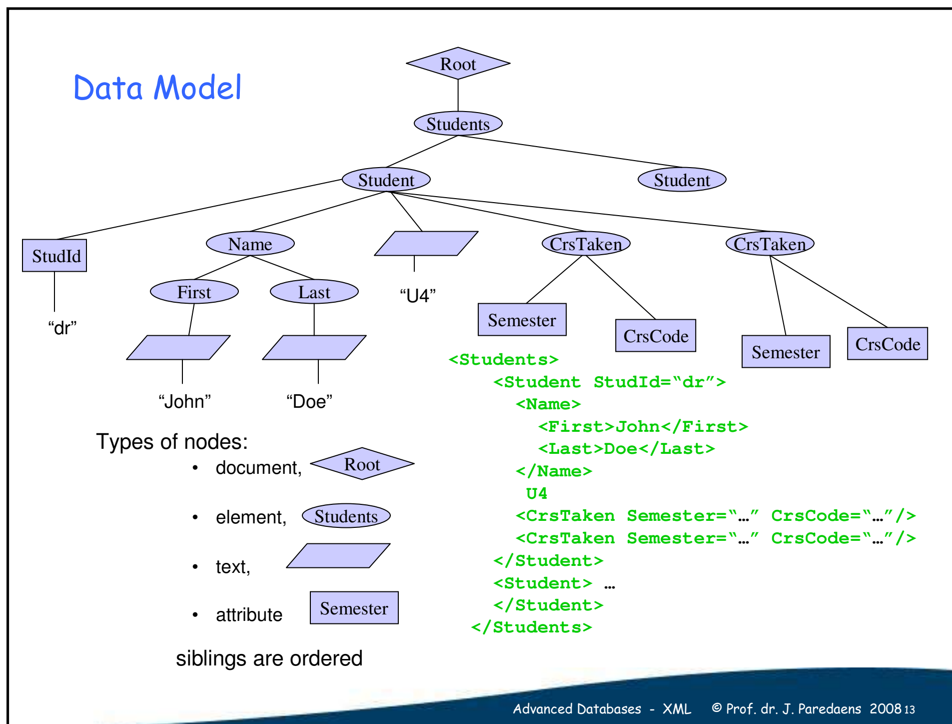
XML Attributes

- An element can have any number of attributes;
- the order of the attributes does not matter;
- an attribute can only occur once within an element;
- attribute values can only be strings;
- the following have the same semantics (except for the ordering of <b> and <c>):

```
<a b="2" c="Jan" />
<a> <b>2</b> <c>Jan</c> </a>
```


Well-formedness

- We call an XML-document well-formed iff
  - it has one root element;
  - elements are properly nested;
  - any attribute can only occur once in a given opening tag and its value must be quoted;



- A value is an ordered sequence of zero or more items;
- An item is a node or an atomic value;
- There are four kinds of nodes:
  - Document Node
  - Element Node
  - Attribute Node
  - Text Node
- Children are element or text nodes (no attribute nodes)
- Examples of values
  - 47
  - <goldfish/>
  - (1, 2, 3)
  - (47, <goldfish/>, "Hello")
  - ()
  - An XML document
  - An attribute standing by itself

Namespaces

- Cfr. C++
- Building vocabularies to prevent naming conflicts;
- uniform use of tag-names;
- general form of a tag:

  `<URI:local-name>` or `<URL:local-name>`

  in principle nothing to do with Internet.
- use different URIs(URLs) for different domains;

  `"http://www.acmeinc.com/jp#students"` for students

  `"http://www.acmeinc.com/jp#toys"` for toys
- synonyms for URIs (URLs) can be declared;

  called namespaces
- default namespace;


```
<item xmlns="http://www.acmeinc.com/jp#supplies"
      xmlns:toy="http://www.acmeinc.com/jp#toys">
  <name>backpack</name>
  <feature>
    <toy:item>
      <toy:name>cyberpet</toy:name>
    </toy:item>
  </feature>
  <item xmlns="http://www.acmeinc.com/jp#supplies2"
        xmlns:toy="http://www.acmeinc.com/jp#toys2">
    <name>notebook</name>
    <toy:name>sticker</toy:name>
  </item>
</item>
```

- the default namespace is declared by the attribute `xmlns`
- the other namespaces are declared by `xmlns`:<synonym>
- the outermost `<item>`, the first `<name>` and `<feature>` belong to default namespace `"http://www.acmeinc.com/jp#supplies"`
- `<toy:item>` and the first `<toy:name>` belong to the namespace `"http://www.acmeinc.com/jp#toys"`
- the innermost `<item>` and the second `<name>` belong to the default namespace `"http://www.acmeinc.com/jp#supplies2"`
- the second `<toy:name>` belongs to the namespace `"http://www.acmeinc.com/jp#toys2"`

Running example 2

```
<?xml version="1.0"?>
<adm:Report adm:Date="2004-12-12">
  <adm:Students>
    <adm:Student adm:StudId="ST11">
      <adm:Name>
        <adm:First>Jan</adm:First>
        <adm:Last>Vijs</adm:Last>
      </adm:Name>
      <adm:Status>U2</adm:Status>
      <adm:CrsTaken adm:CrsCode="CS308" adm:Semester="F2003"/>
      <adm:CrsTaken adm:CrsCode="MAT123" adm:Semester="F2003"/>
    </adm:Student>
    <adm:Student adm:StudId="ST66">
      <adm:Name>
        <adm:First>Jan</adm:First>
        <adm:Last>De Moor</adm:Last>
      </adm:Name>
      <adm:Status>U3</adm:Status>
      <adm:CrsTaken adm:CrsCode="CS308" adm:Semester="S2002"/>
      <adm:CrsTaken adm:CrsCode="MAT123" adm:Semester="F2003"/>
    </adm:Student>
    <adm:Student adm:StudId="ST98">
      <adm:Name>
        <adm:First>Bart</adm:First>
        <adm:Last>Simpson</adm:Last>
      </adm:Name>
      <adm:Status>U4</adm:Status>
      <adm:CrsTaken adm:CrsCode="CS308" adm:Semester="S2002"/>
    </adm:Student>
  </adm:Students>
```



```
  <adm:Classes>
    <adm:Class>
      <adm:CrsCode>CS308</adm:CrsCode>
      <adm:Semester>S2002</adm:Semester>
      <adm:ClassRoster adm:Members="ST66 ST98"/>
    </adm:Class>
    <adm:Class>
      <adm:CrsCode>CS308</adm:CrsCode>
      <adm:Semester>F2003</adm:Semester>
      <adm:ClassRoster adm:Members="ST11"/>
    </adm:Class>
    <adm:Class>
      <adm:CrsCode>MAT123</adm:CrsCode>
      <adm:Semester>F2003</adm:Semester>
      <adm:ClassRoster adm:Members="ST11 ST66"/>
    </adm:Class>
  </adm:Classes>
  <adm:Courses>
    <adm:Course adm:CrsCode="CS308">
      <adm:CrsName>Databases</adm:CrsName>
    </adm:Course>
    <adm:Course adm:CrsCode="MAT123">
      <adm:CrsName>Algebra</adm:CrsName>
    </adm:Course>
  </adm:Courses>
</adm:Report>
```

2 XPath [3,5]

Unabbreviated syntax

A location path transforms a document tree and one of its nodes as a context node to a sequence of distinct nodes in document order.

- `child::a` selects the element children with name `a` of the context node
- `child::*` selects all element children of the context node
- `child::text()` selects all text children of the context node
- `child::node()` selects all the (element or text) children of the context node
- `attribute::a` selects the attribute with name `a` of the context node
- `attribute::*` selects all the attributes of the context node
- `descendant::a` selects the element descendants with name `a` of the context node
- `descendant-or-self::a` selects the element descendants with name `a` of the context node and, if the context node has name `a`, the context node as well
- `ancestor::a` selects all ancestors with name `a` of the context node
- `ancestor-or-self::a` selects the ancestors with name `a` of the context node and, if the context node is an element with name `a`, the context node as well
- `self::a` selects the context node if it is has the name `a`, and otherwise selects nothing
- `child::chapter/descendant::a` selects the element descendants with name `a` of the element children with name `chapter` of the context node
- `child::*/child::a` selects all element grandchildren with name `a` of the context node
- `/` selects the document root

- `/descendant::a` selects all the elements with name `a` in the same document as the context node
- `/descendant::a/child::b` selects all the elements with name `b` that have a parent with name `a` and that are in the same document as the context node
- `child::a[position()=1]` selects the first child with name `a` of the context node
- `child::a[position()=last()]` selects the last child with name `a` of the context node
- `child::a[position()=last()-1]` selects the last but one child with name `a` of the context node
- `child::a[position()>1]` selects all the children with name `a` of the context node other than the first child with name `a` of the context node
- `following-sibling::chapter[position()=1]` selects the next chapter sibling of the context node
- `preceding-sibling::chapter[position()=1]` selects the prev. chapter sibling of the context node
- `/descendant::figure[position()=42]` selects the forty-second element with name `figure` in the document
- `child::a[attribute::type="warning"]` selects all the children with name `a` of the context node that have an attribute with name `type` and value `"warning"`
- `child::a[attribute::type="warning"][position()=5]` selects the fifth child with name a of the context node that has an attribute with name `type` and value `"warning"`
- `child::a[position()=5][attribute::type="warning"]` selects the fifth child with name `a` of the context node if that child has an attribute with name `type` and value `"warning"`
- `child::chapter[child::title]` selects the chapter children of the context node that have one or more children with name `title`
- `child::*[self::chapter or self::appendix]` selects the chapter and appendix children of the context node
- `child::*[self::chapter or self::appendix][position()=last()]` selects the last chapter or appendix

General form of a location path:

• relative lp : $step_1$ / $step_2$ / … / $step_n$ (n > 0)

```
child::*/child::name
```

Each step in turn selects a sequence of nodes relative to a context node. An initial sequence of steps is composed together with a following step as follows: the initial sequence of steps selects a sequence of nodes relative to a context node. Each node in that sequence is used as a context node for the following step.
The sequences of nodes identified by that step are unioned together.
The sequence of nodes identified by the composition of the steps is this union, ordered in document order, without duplicates.

The example selects the union of all name children of children of the context node.

• absolute lp : / $step_1$ / $step_2$ / … / $step_n$ ($n \geq 0$)

```
/child::*/child::name
```

The initial / selects the root node of the document containing the context node.
Then apply $step_1$ / $step_2$ / … / $step_n$

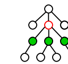
General form of step :

axis :: node-test [$predicate_1$] … [$predicate_n$] ($n \geq 0$)

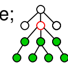
```
child::name[attribute::type="warning"][position()=5]
```

The node-sequence selected by the step is the node-sequence that results from generating an initial node-sequence from the axis, filtering it by the node-test, and then filtering that node-sequence by each of the predicates in turn.

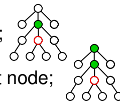
• the `child` axis contains the element and text children of the context node;



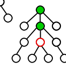
• the `descendant` axis contains the element and text descendants of the context node;



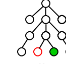
• the `parent` axis contains the parent of the context node;



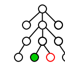
• the `ancestor` axis contains the ancestors of the context node;



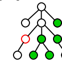
• if the context node is an element or text node the `following-sibling` axis contains all the following element or text siblings of the context node;
if the context node is an attribute node, the `following-sibling` axis is empty;



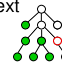
• if the context node is an element or text node the `preceding-sibling` axis contains all the preceding element or text siblings of the context node;
if the context node is an attribute node, the `preceding-sibling` axis is empty;



• the `following` axis contains all element or text nodes in the same document as the context node that are after the context node in document order (preorder), excluding any descendants;

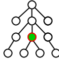


• the `preceding` axis contains all element or text nodes in the same document as the context node that are before the context node in document order, excluding any ancestors;

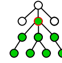


• the `attribute` axis contains the attribute children of the context node;

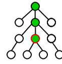
• the `self` axis contains just the context node itself;



• the `descendant-or-self` axis contains the context node and the descendants of the context node;

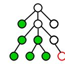


• the `ancestor-or-self` axis contains the context node and the ancestors of the context node;



Remark that the `ancestor`, `descendant`, `following`, `preceding` and `self` axis partition the element nodes of a document.

```
parent::*/parent::*/child::*/following-sibling::*/preceding-sibling::*/
following::*/descendant::*/ancester::*/preceding::*
```

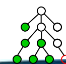


Node-test has the form:

• label : filters the nodes with that label
• `*` : filters all element nodes
• `node()` : filters all nodes
• `text()` : filters the text nodes

Predicate has (until now) the form: `[position()=i]`
The ancestor, ancestor-or-self, preceding, and preceding-sibling axes are reverse axes; all other axes are forward axes. The proximity position of a node of a node-sequence with respect to an axis is defined to be the position of the node in the node-sequence ordered in document order if the axis is a forward axis and ordered in reverse document order if the axis is a reverse axis. The first position is 1.
`[position()=i]` filters those nodes whose proximity position is equal to i.

```
parent::*/parent::*/child::*/following-sibling::*/
preceding-sibling::*[position()=1]/following::*/descendant::*/
            ancestor::*[position()=1]/preceding::*
```



Abbreviated Syntax in red

More used than the unabbreviated syntax.

- `child::` is omitted
  `child::name` `name`
- `attribute::` is abbreviated to `@`
  `attribute::name` `@name`
- `/descendant-or-self::node()/` is abbreviated to `//`
  `child::aa/descendant-or-self::node()/child::bb` `aa//bb`
- `self::node()` is abbreviated to `.`
  `self::node()/descendant-or-self::node()/child::name` `.//name`
- `parent::node()` is abbreviated to `..`
  `parent::node()/child::title` `../title`
- `position()=` is omitted
  `[position()=1]` `[1]`

- `a` selects the element children with name `a` of the context node
- `*` selects all element children of the context node
- `text()` selects all text children of the context node
- `node()` selects all the element and text children of the context node
- `@a` selects the attribute with name `a` of the context node
- `@*` selects all the attributes of the context node
- `*/a` selects all element grandchildren with name `a` of the context node
- `a[1]` selects the first element child with name `a` of the context node
- `a[last()]` selects the last element child with name `a` of the context node
- `/doc/chapter[5]/section[2]` is an abbreviation of
  `/child::doc/child::chapter[position()=5]/child::section[position()=2]`
- `chapter//a` is an abbreviation of `chapter/descendant-or-self::node()/child::a`
- `//a` is an abbreviation of `/descendant-or-self::node()/child::a`
- `./a` is an abbreviation of `self::node()/child::a`, which is equivalent to `child::a`
- `chapter/./a` is an abbreviation of `child::chapter/self::node()/child::a`,
  which is equivalent to `child::chapter/child::a` or `chapter/a`
- `*/..` is an abbreviation of `child::*/parent::node()` selects the context node, if it has children,
  otherwise nothing is selected
- `//a[1]` is an abbreviation of `/descendant-or-self::node()/child::a[position()=1]`
  and selects the name descendant elements of the root that are the first name child of their parent
- `/descendant::a[1]` is an abbreviation of `/descendant::a[position()=1]` and selects
  the first name descendant of the root

Predicates [3]

Predicates can be

- Boolean expression

`[a/b/c/text()=5]` 5 belongs to the result sequence of the location path
`[a/b/c=d/e]` intersection of the 2 result sequences is not empty

- location path

`[a/b/c]` result sequence of location path is not empty

- number

`[5]` means `[position()=5]`

- combinations using and, or, not

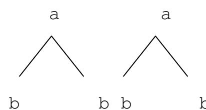
Note the difference between

- `[a/text() = 12]` selects a node if it has an `a` child with a text child equal to 12;
- `[not(a/text() != 12)]` selects a node if all the text children of all its `a` childeren are equal to 12;

(Cfr. Data Model p.12)

```
• //Students/Student/@StudId[../Name/First[string(text())="John"]
                            or ../CrsTaken[@Semester="2"]]
• //Students/Student[Name/First[string(text())="John"]
                            or CrsTaken[@Semester="2"]]
• //Students/Student/Status[../Name/First[string(text())="John"]
                            or ../CrsTaken[@Semester="2"]]
• //Students/Student/Name[First[string(text())="John"]
                            or ../CrsTaken[@Semester="2"]]
• //Students/Student/Name/First[string(text())="John"
                            or ../../CrsTaken[@Semester="2"]]
```

- `//a/b[2]` selects 2nd and 4th b
- `(//a/b)[2]` selects 2nd b



Xpath as a Query Language for XML

Document on file 'po.xml', running example 3

```
<?xml version="1.0"?>
<purchaseOrder orderDate="2004-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>2004-12-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```


Select the date of the purchase order if it is shipped to Belgium:

```
document("po.xml")/purchaseOrder/@orderDate[../shipTo/@country="Belgium"]
document("po.xml")/purchaseOrder[shipTo/@country="Belgium"]/@orderDate
```

Select the items that are shipped to Mechelen and whose price is greater than 40:

```
document("po.xml")//item[USPrice/text()>40 and
                        (../../billTo/city/text()="Mechelen")]
```

Select the purchase order if all its items cost more than 40:

```
document("po.xml")/purchaseOrder[not(.//USPrice/text()<=40)]
```

Select the purchase order if some of its items cost more than 40:

```
document("po.xml")/purchaseOrder[.//USPrice/text>40]
```

Select the purchase order if it contains at least 2 items:

```
document("po.xml")/purchaseOrder[items/item[2]]
```

Given an item, select the preceding item of the same purchase order:

```
preceding-sibling::*[1]
```

Select the dates on which there is an order that is shipped and billed in the same city:

```
document("po.xml")/purchaseOrder/@orderDate
                  [../shipTo/city/text()=../billTo/city/text()]
```

3. XQuery [4,5,6]

Principles of XQuery Design

- A set of operators that are closed under the data model;
- Every expression has a value and no side effects;
- Expressions can be composed with full generality;
- Use the type system of XML Schema;
- XPath compatibility. Adopt XPath as a syntactic subset;
- Roughly equivalent to "relational completeness“;
- No formal standard exists for hierarchic languages;
- XQuery is a case-sensitive language
- Keywords are in lower-case
- Expressions can raise errors

- Recursive Functions;
- Conciseness - Simplicity
- Static Analysis
  - optional static analysis phase before query execution
  - type inference rules based on XML Schema
  - early detection of some kinds of errors
  - optimization
- Comments look like this:

```
(: Houston, we have a problem :)
```

Interesting Implementations

For overview: **http://www.w3.org/XML/Query**

Demos

- *X-Hive's XQuery demo*
- *Software AG's Tamino XML Query Demo*

Free and/or Open Source

- *Fernandez/Simeon's Galax. Open-source.*
- *Saxonica's Saxon. Available in a schema-aware version as a commercial product, and without schema support as open source.*
- *Sourceforge's eXist. Open-source.*
- *Ispras Modis' Sedna. Open-source. … and many more …*

Commerical

- *BEA's Liquid Data*
- *X-Hive; Software AG's Tamino;*
- *Microsoft's SQL Server 2005 Express, with XQuery support*
- *Oracle's Xquery Technology - Preview … and many many more …*

XQuery Expressions

- Literals: `"Hello" 47 4.7 4.7E-2`
- Constructed values:
  `true() false() date("2002-03-15")`
- Variables: `$x`
- Constructed sequences
  `$a, $b` is the same as `($a, $b)`
  `(1, (2, 3), (), (4))` is the same as `1, 2, 3, 4`
  `5 to 8` is the same as `5, 6, 7, 8`
- Functions
  - XQuery functions have expressions for bodies and may be recursive
  - Function calls: `two-argument-function(1, (2,3))`
  - Functions are not overloaded (except certain built-ins)
  - Subtype substitutability in function arguments

- Functions on sequences
  - `union intersect except` (infix) – only on sequences of nodes; result in doc. order without dupl.
  - `empty() count()`
- Location paths of XPath
  - abbreviated and non-abbreviated;
  - examples:

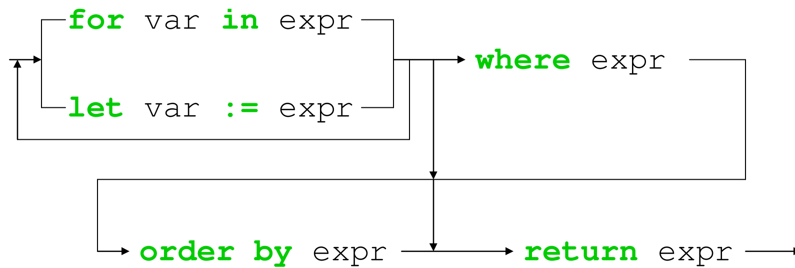
```
book[author/text() = "Mark Twain "]
chapter[2]
book[appendix]
person[@married]
//book[author/text() = "Mark Twain"]/chapter[2]
(1 to 100)[. mod 5=0]
```

- Arithmetic operators: `+ - * div idiv mod`
  - Extract typed value from node
  - Multiple values => error
  - If operand is `()`, return `()`
  - Supported for numeric and date/time types

- Comparison operators
  - `eq ne gt ge lt le` compare single atomic values
  - `= != > >= < <=` implied existential semantics
  - `is`, `is not` compare two nodes based on identity
  - `<< >>` compare two nodes based on document order

- FLWOR Expression

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



Examples in Galax

classes.xml

```
<?xml version="1.0"?>
<Classes>
  <Class CrsCode="CS308" Semester="F1997">
    <CrsName>Market Analysis</CrsName>
    <Instructor>Adrian Jones</Instructor>
  </Class>
  <Class CrsCode="EE101" Semester="F1995">
    <CrsName>Electronic Circuits</CrsName>
    <Instructor>David Jones</Instructor>
  </Class>
  <Class CrsCode="CS305" Semester="F1995">
    <CrsName>Database Systems</CrsName>
    <Instructor>Mary Doe</Instructor>
  </Class>
</Classes>
```

transcripts.xml

```
<?xml version="1.0"?>
<Transcripts>
  <Transcript>
    <Student StudId="111111111" Name="John Doe"/>
    <CrsTaken CrsCode="CS308" Semester="F1990" Grade="B"/>
    <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B"/>
    <CrsTaken CrsCode="EE101" Semester="F1997" Grade="A"/>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
  </Transcript>
  <Transcript>
    <Student StudId="987654321" Name="Bart Simpson"/>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
    <CrsTaken CrsCode="CS308" Semester="F1994" Grade="B"/>
  </Transcript>
</Transcripts>
```

– FOR clause

```
for $c in document("classes.xml")//Class,
    $t in document("transcripts.xml")//Transcript
```

- specify documents used in the query
- declare variables and bind them to a range
- result is a list of bindings

– LET clause

```
let $sn := $t/Student/@Name, $cn := $c/CrsName
```

- bind variables to a value

– WHERE clause

```
where $c/@CrsCode = $t/CrsTaken/@CrsCode and
      $c/@Semester = $t/CrsTaken/@Semester
```

- selects a sublist of the list of bindings

– RETURN clause

```
return
  <CrsStud> $cn <Name> $sn </Name> </CrsStud>
```

- construct result for every selected binding

q01.xq

```
<StudentList>
{
for $t in document("transcripts.xml")//Transcript
let $s := $t/Student
where $t/CrsTaken/@CrsCode = "CS308"
return <Stud id="{$s/@StudId}"> {$s/@Name} </Stud>
}
</StudentList>
```

```
<StudentList>
  <Stud id="111111111" Name="John Doe"/>
  <Stud id="987654321" Name="Bart Simpson"/>
</StudentList>
```

• Element Constructor

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

The result of an element constructor is a new element node, with its own node identity. All the attribute and descendant nodes of the new element node are also new nodes with their own identities, even if they are copies of existing nodes.

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

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

```
<e> <p r="[1,5 to 7,9]"> AA </p> <eg> $i//t </eg>
    <p r="[{1, 5 to 7, 9}]"> BB </p> <eg> {$i//t} </eg> </e>
```

The above query might generate the following result:

```
<e> <p r="[1,5 to 7,9]"> AA </p> <eg> $i//t </eg>
    <p r="[1,5,6,7,9]"> BB </p> <eg> <t>CC</t> </eg> </e>
```

`<a>{1, 2, 3}</a>` The constructed element node has one child, a text node containing the value "1 2 3".

`<fact>I saw <howmany>{5 + 3}</howmany> cats.</fact>` The constructed element node has three children: a text node containing "I saw ", a child element node named howmany, and a text node containing " cats.". The child element node in turn has a single text node child containing the value "8".

doc1.xml

```
<docu>
<el a1="v1"
    a2="v2">
   <sub> v3 </sub>
   <sub> v4 </sub>
</el>
<el a1="v5">
   <sub> v6 </sub>
</el>
<el a1="v7"/>
</docu>
```

The functions `data()` and `string()` give the content of their arguments (Cfr. later)

q14.xq

```
for $e in document("doc1.xml")//el
return $e,
for $e in document("doc1.xml")//el
return <k> $e/@a1 </k>,
for $e in document("doc1.xml")//el
return <k2> {$e/@a1} </k2>,
for $e in document("doc1.xml")//el
return <k3 a1="{$e/@a1}" />,
for $e in document("doc1.xml")//el
return <k4 b="{$e/@a1}" />,
for $e in document("doc1.xml")//el
return <k5> {string($e/@a1)} </k5>,
for $e in document("doc1.xml")//el
for $s in $e/sub
return <k6> {data($s)} </k6>,
for $e in document("doc1.xml")//el
for $s in $e/sub
return <k7> {$s/text()} </k7>
```

Result

```
<el a1="v1" a2="v2"><sub> v3 </sub><sub> v4 </sub></el>,
<el a1="v5"><sub> v6 </sub></el>, <el a1="v7"/>, <k> $e/@a1 </k>,
<k> $e/@a1 </k>, <k> $e/@a1 </k>, <k2 a1="v1"/>, <k2 a1="v5"/>, <k2 a1="v7"/>,
<k3 a1="v1"/>, <k3 a1="v5"/>, <k3 a1="v7"/>, <k4 b="v1"/>, <k4 b="v5"/>,
<k4 b="v7"/>, <k5>v1</k5>, <k5>v5</k5>, <k5>v7</k5>, <k6> v3 </k6>,
<k6> v4 </k6>, <k6> v6 </k6>, <k7> v3 </k7>, <k7> v4 </k7>, <k7> v6 </k7>
```


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

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

The first enclosed expression after the element keyword generates the name of the element, and the second enclosed expression generates the content and attributes:

q15.xq

```
element {string(<f>nnn</f>)} {string(<e> sss</e>)},
let $dict := <dic> <entry word="address">
<variant lang="German">Adresse</variant>
<variant lang="Italian">Indirizzo</variant> </entry> </dic>
let $e:=<address>123 Roosevelt Ave. Flushing, NY 11368</address>
return   element {string($dict/entry[@word=name($e)]/variant[@lang="Italian"])}
{$e/@*, string($e)}
```

results in

```
<nnn> sss</nnn>, <Indirizzo>123 Roosevelt Ave. Flushing, NY 11368</Indirizzo>
```

q02.xq

```
for $c in document("classes.xml")//Class,
    $t in document("transcripts.xml")//Transcript
where $c/@CrsCode = $t/CrsTaken/@CrsCode and
      $c/@Semester = $t/CrsTaken/@Semester
return
  <CrsStud>
    {$c/CrsName}
     <StudName> {$t/Student/@Name} </StudName>
  </CrsStud>
```

```
<CrsStud>
  <CrsName>Market Analysis</CrsName>
  <StudName Name="John Doe"/>
</CrsStud>,
<CrsStud>
  <CrsName>Electronic Circuits</CrsName>
  <StudName Name="John Doe"/>
</CrsStud>,
<CrsStud>
  <CrsName>Database Systems</CrsName>
  <StudName Name="John Doe"/>
</CrsStud>,
<CrsStud>
  <CrsName>Database Systems</CrsName>
  <StudName Name="Bart Simpson"/>
</CrsStud>
```

Flat Join
(wrong result
cfr. next slide)

Why

```
<CrsStud>
  <CrsName>Market Analysis</CrsName>
  <StudName Name="John Doe"/>
</CrsStud>,
```

is in the result

classes.xml

```
<?xml version="1.0"?>
<Classes>
  <Class CrsCode="CS308" Semester="F1997">
    <CrsName>Market Analysis</CrsName>
    <Instructor>Adrian Jones</Instructor>
  </Class>
  <Class CrsCode="EE101" Semester="F1995">
    <CrsName>Electronic Circuits</CrsName>
    <Instructor>David Jones</Instructor>
  </Class>
  <Class CrsCode="CS305" Semester="F1995">
    <CrsName>Database Systems</CrsName>
    <Instructor>Mary Doe</Instructor>
  </Class>
</Classes>
```

transcripts.xml

```
<?xml version="1.0"?>
<Transcripts>
  <Transcript>
    <Student StudId="111111111" Name="John Doe"/>
    <CrsTaken CrsCode="CS308" Semester="F1990" Grade="B"/>
    <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B"/>
    <CrsTaken CrsCode="EE101" Semester="F1997" Grade="A"/>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
  </Transcript>
  <Transcript>
    <Student StudId="987654321" Name="Bart Simpson"/>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
    <CrsTaken CrsCode="CS308" Semester="F1994" Grade="B"/>
  </Transcript>
</Transcripts>
```

q03.xq

```
for $c in document("classes.xml")//Class,
    $t in document("transcripts.xml")//Transcript,
    $ct in $t/CrsTaken
where $c/@CrsCode = $ct/@CrsCode and
      $c/@Semester = $ct/@Semester
return
  <CrsStud>
    {$c/CrsName}
     <StudName> {$t/Student/@Name} </StudName>
  </CrsStud>
```

Flat join

```
<CrsStud>
  <CrsName>Database Systems</CrsName>
  <StudName Name="John Doe"/>
</CrsStud>,
<CrsStud>
  <CrsName>Database Systems</CrsName>
  <StudName Name="Bart Simpson"/>
</CrsStud>
```

- Order of variables in FOR-clause matters.
- Classes with no transcripts are omitted.

q04.xq

```
for $c in document("classes.xml")//Class
order by ($c/CrsName/text())
return
(
  <CrsStud CrsName="{$c/CrsName/text()}">
    {
    for $t in document("transcripts.xml")//Transcript
    for $ct in $t/CrsTaken
    where ($c/@CrsCode = $ct/@CrsCode and
          $c/@Semester = $ct/@Semester)
    return <StudName> {$t/Student/@Name}</StudName>
    }
  </CrsStud>
)
```

Nested Join

```
<CrsStud CrsName="Database Systems">
  <StudName Name="John Doe"/>
  <StudName Name="Bart Simpson"/>
</CrsStud>, <CrsStud CrsName="Electronic Circuits"/>,
<CrsStud CrsName="Market Analysis"/>
```

Group students per course code and semester on the basis of Transcripts alone

```
let $trs := document("transcripts.xml")//Transcript
let $ct := $trs/CrsTaken
for $c in $ct
return
  <CrsStud CrsCode="{$c/@CrsCode}" Semester="{$c/@Semester}">
{     for $tc in $trs/CrsTaken
      where (($c/@CrsCode = $tc/@CrsCode) and
            ($c/@Semester = $tc/@Semester))
      order by ($tc/../Student/@StudId)
      return $tc/../Student
}
  </CrsStud>
```

q05.xq

```
<CrsStud CrsCode="CS308"
  Semester="F1990">
  <Student StudId="111111111" Name="John Doe"/>
</CrsStud>,
<CrsStud CrsCode="MAT123"
  Semester="F1997">
  <Student StudId="111111111" Name="John Doe"/>
</CrsStud>,
<CrsStud CrsCode="EE101"
  Semester="F1997">
  <Student StudId="111111111" Name="John Doe"/>
</CrsStud>,
<CrsStud CrsCode="CS305"
  Semester="F1995">
  <Student StudId="111111111" Name="John Doe"/>
  <Student StudId="987654321" Name="Bart Simpson"/>
</CrsStud>,
<CrsStud CrsCode="CS305"
  Semester="F1995">
  <Student StudId="111111111" Name="John Doe"/>
  <Student StudId="987654321" Name="Bart Simpson"/>
</CrsStud>,
<CrsStud CrsCode="CS308"
  Semester="F1994">
  <Student StudId="987654321" Name="Bart Simpson"/>
</CrsStud>
```


- User-defined functions

`local:` is only necessary in GALAX, not necessary in XQuery;

Count number of descendants

q06.xq

```
declare function local:countElemNodes($e) {
    if (empty($e/*))
    then 0
    else local:countElemNodes($e/*) + count($e/*)
};

local:countElemNodes(document("transcripts.xml")//Transcripts)
```

Result : `10`

- Input and output are typed
- Body can be any XQuery expression, recursion is allowed
- XPath core functions: `sum()`, `count()`
- Automatic generalization of `local:countElemNodes()` to collection

Group students per course code and semester on basis of Transcripts alone q13.xq

```
declare function local:extractClasses($e) {
  for $c in $e//CrsTaken
  return <Class CrsCode="{$c/@CrsCode}" Semester="{$c/@Semester}"/>
};

let $trs := document("transcripts.xml")/Transcripts
for $c in local:extractClasses($trs)
return
  <ClassRoster>
    { $c/@CrsCode, $c/@Semester }
    { for $t1 in $trs//Transcript/CrsTaken[@CrsCode=$c/@CrsCode and
                                    @Semester=$c/@Semester]
      order by ($t1/../Student/@StudentId)
      return $t1/../Student
    }
  </ClassRoster>
```


Result

```
<ClassRoster CrsCode="CS308"
  Semester="F1990">
  <Student StudId="111111111" Name="John Doe"/>
</ClassRoster>,
<ClassRoster CrsCode="MAT123"
  Semester="F1997">
  <Student StudId="111111111" Name="John Doe"/>
</ClassRoster>,
<ClassRoster CrsCode="EE101"
  Semester="F1997">
  <Student StudId="111111111" Name="John Doe"/>
</ClassRoster>,
<ClassRoster CrsCode="CS305"
  Semester="F1995">
  <Student StudId="111111111" Name="John Doe"/>
  <Student StudId="987654321" Name="Bart Simpson"/>
</ClassRoster>,
<ClassRoster CrsCode="CS305"
  Semester="F1995">
  <Student StudId="111111111" Name="John Doe"/>
  <Student StudId="987654321" Name="Bart Simpson"/>
</ClassRoster>,
<ClassRoster CrsCode="CS308"
  Semester="F1994">
  <Student StudId="987654321" Name="Bart Simpson"/>
</ClassRoster>
```

Give all the elements in classes that contain somewhere “ys” and whose elementname ends with “ses”

q17.xq

```
<StudentList>
{
  for $t in document("classes.xml")//*[contains(string(.), "ys")
                                       and ends-with(name(.),"ses")]
  return $t
}
</StudentList>
```

```
<StudentList>
  <Classes>
    <Class CrsCode="CS308"
      Semester="F1997">
      <CrsName>Market Analysis</CrsName>
      <Instructor>Adrian Jones</Instructor>
    </Class>
    <Class CrsCode="EE101"
      Semester="F1995">
      <CrsName>Electronic Circuits</CrsName>
      <Instructor>David Jones</Instructor>
    </Class>
    <Class CrsCode="CS305"
      Semester="F1995">
      <CrsName>Database Systems</CrsName>
      <Instructor>Mary Doe</Instructor>
    </Class>
  </Classes>
</StudentList>
```

Cfr.
www.w3.org/TR/xquery-operators

>>, << document order q18.xq

```
<ua>{
for $c1 in document("transcripts.xml")//CrsTaken[@Semester > "F1994"]
for $c2 in document("transcripts.xml")//CrsTaken[@Semester > "F1994"]
  where (($c1 << $c2) and not($c1/@Grade = $c2/@Grade))
  return <ff> {$c1, $c2} </ff>
}</ua>
```

```
<ua>
  <ff>
    <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B"/>
    <CrsTaken CrsCode="EE101" Semester="F1997" Grade="A"/>
  </ff>
  <ff>
    <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B"/>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
  </ff>
  <ff>
    <CrsTaken CrsCode="MAT123" Semester="F1997" Grade="B"/>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
  </ff>
  <ff>
    <CrsTaken CrsCode="EE101" Semester="F1997" Grade="A"/>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
  </ff>
  <ff>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="A"/>
    <CrsTaken CrsCode="CS305" Semester="F1995" Grade="C"/>
  </ff>
</ua>
```

```
let $doc := <a> <b> aa </b> <c> 1 </c> <c> 2 </c> <b> bb </b> <c> 3 </c>
            <c> 4 </c> <c> 5 </c> </a>
let $i1 := $doc//b[2]
for $i2 in $doc//c[. >> $i1][position()<=2]
return $i2/text()
```

q19.xq

```
text {" 3 "}, text {" 4 "}
```

```
let $doc := <a>  <c> 1 </c> <c> 2 </c> <b> bb </b> <c> 3 </c>
            <c> 4 </c> <c> 5 </c> </a>
for $i2 in $doc//c
where not(some $i1 in $doc//b satisfies  ($i1 << $i2))
return $i2/text()
```

q20.xq

```
text {" 1 "}, text {" 2 "}
```

```
let $doc := <a>  <c> 1 </c> <b> <c> 2 </c> bb </b> <c> 3 </c>
            <c> 4 </c> <c> 5 </c> </a>
for $i2 in $doc//c
where not(some $i1 in $doc//b satisfies  ($i1 << $i2))
return $i2/text()
```

q21.xq

```
text {" 1 "}
```


Find everything between two nodes

```
declare function local:btween($seq, $start, $enda) {
  let $nodes :=
         for $n in $seq except $start//node()
         where $n >> $start and $n << $enda
         return $n
  return ($nodes except $nodes//node())
};

<c_s>
{
let $proc := (<a>  <c> 1 <b> fff </b> </c> <b> <c> 2 </c>  bb </b> <c> 3 </c>
             <c> 4 </c> <c> 5 </c> </a>),
    $first := $proc/c[1],
    $second := $proc/c[last()]
return local:btween($proc//node(), $first, $second)
}
</c_s>
```

q22.xq

```
<c_s><b><c> 2 </c>  bb </b><c> 3 </c><c> 4 </c></c_s>
```

```
<c_s>
{
let $proc := (<abc> cxcxc <cdf> 1 <bef> fffc </bef> </cdf> <bfg> <c> 2 </c>
             cbb </bfg> <cgl> 3 </cgl> <cgt> 4 </cgt> <csd> c5 </csd> </abc>)
return $proc//node()[contains(.,"c")]
}
</c_s>
```

```
<c_s>
   cxcxc
  <cdf> 1 <bef> fffc </bef></cdf>
  <bef> fffc </bef>
   fffc
  <bfg><c> 2 </c>  cbb </bfg>
    cbb
  <csd> c5 </csd>
   c5
</c_s>
```

q23.xq


```
declare function local:one_level($l, $p) {
    <part partid="{ $p/@partid }"
          name="{ $p/@name }" >
        {
            for $s in $l//part
            where $s/@partof = $p/@partid
            return local:one_level($l,$s)
        }
    </part>
};

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

return
(<parttree>
  {
    for $p in $list//part[empty(@partof)]
    return local:one_level($list,$p)
  }
</parttree>)
```

q24.xq

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

Grouping and aggregation
Count courses per student

q08.xq

```
for $t in document("transcripts.xml")//Transcript,
    $s in $t/Student
let $c := $t/CrsTaken
return
  <StudentSummary StudId="{$s/@StudId}" Name="{$s/@Name}"
                  TotalCourses="{count($c)}"/>
```

```
<StudentSummary StudId="111111111" Name="John Doe" TotalCourses="4"/>,
<StudentSummary StudId="987654321" Name="Bart Simpson" TotalCourses="2"/>
```


Compute average grade per class

q09.xq

```
declare function local:numericGrade($a) {
  let $grade := data($a)
  return
    if ($grade = "A") then 10 else if ($grade = "B") then 8
    else if ($grade = "C") then 6 else if ($grade = "D") then 4
    else if ($grade = "E") then 2 else 0
};

for $c in document("classes.xml")//Class
let $g := ( for $ct in document("transcripts.xml")//CrsTaken
            where $ct/@CrsCode = $c/@CrsCode
                  and $ct/@Semester = $c/@Semester
            return local:numericGrade($ct/@Grade)
          )
order by ($c/@CrsCode)
return
  <ClassSummary CrsCode="{string($c/@CrsCode)}" Semester="{string($c/@Semester)}"
                CrsName="{$c/CrsName/text()}" Instructor="{$c/Instructor/text()}"
                AvgGrade="{ if (count($g) > 0) then avg($g) else 0}"/>
```

```
<ClassSummary CrsCode="CS305" Semester="F1995" CrsName="Database Systems"
              Instructor="Mary Doe" AvgGrade="8"/>,
<ClassSummary CrsCode="CS308" Semester="F1997" CrsName="Market Analysis"
              Instructor="Adrian Jones" AvgGrade="0"/>,
<ClassSummary CrsCode="EE101" Semester="F1995" CrsName="Electronic Circuits"
              Instructor="David Jones" AvgGrade="0"/>
```

- Quantification

  - Existential quantification:
    - **some** *$Var* **in** *Expr* **satisfies** *Cond*
  - Universal quantification:
    - **every** *$Var* **in** *Expr* **satisfies** *Cond*

Select courses that were followed by some student

q10.xq

```
for $c in document("classes.xml")//Class
where (
  some $t in document("transcripts.xml")//CrsTaken
  satisfies ($c/@CrsCode = $t/@CrsCode and $c/@Semester = $t/@Semester)
)
return $c/CrsName
```

```
<CrsName>Database Systems</CrsName>
```


Select classes in which every student took MAT123

q11.xq

```
for $c in document("classes.xml")//Class
let $g := ( for $t in document("transcripts.xml")//Transcript
            let $tt := for $b in $t/CrsTaken where
                                       $b/@CrsCode = $c/@CrsCode
                                       and $b/@Semester = $c/@Semester
                       return $b
            where not(empty($tt))
            return $t )
where every $tr in $g
      satisfies not(empty($tr[CrsTaken/@CrsCode = "MAT123"]))
order by ($c/CrsCode)
return $c
```

```
<Class CrsCode="CS308"
  Semester="F1997">
  <CrsName>Market Analysis</CrsName>
  <Instructor>Adrian Jones</Instructor>
</Class>,
<Class CrsCode="EE101"
  Semester="F1995">
  <CrsName>Electronic Circuits</CrsName>
  <Instructor>David Jones</Instructor>
</Class>
```

Order sorts a list <ll> … </ll> of elements on their names.

```
declare function local:Car($x)
{if (empty($x/*)) then <ll/> else for $y at $z in $x/* where $z=1 return $y};

declare function local:Cdr($x)
{if (empty($x/*)) then <ll/> else  <ll> {for $y at $z in $x/* where 1 lt $z return $y} </ll>};

declare function local:Cons($x, $y)
{<ll> {$x, $y/*} </ll> };

declare function local:Decomp($x)
{if (empty($x/*)) then <ll> {$x, $x} </ll>
   else if (count($x/*) eq 1) then (<ll> <ll/> {$x}  </ll>)
          else <ll> {local:Decomp(local:Cdr($x))/*[2]} <ll>{local:Car($x),
                     local:Decomp(local:Cdr($x))/*[1]/*} </ll> </ll>};

declare function local:Merge($x, $y)
{if (empty($x/*)) then $y
   else if (empty($y/*)) then $x
          else if (name($x/*[1]) lt name($y/*[1]))
                 then local:Cons($x/*[1], local:Merge(local:Cdr($x), $y))
                 else local:Cons($y/*[1], local:Merge(local:Cdr($y), $x))};

declare function local:Order($x)
{if (count($x/*) lt 2) then $x
   else let $t := local:Decomp($x) let $t1 := local:Car($t) let $t2 := local:Car(local:Cdr($t))
        return local:Merge(local:Order($t1),local:Order($t2))};

let $15 := <ll> <ss/> <vv/> <df/> <fr/> <ds/> <as/> <gy/> <qn/> <cm/> <an/> <fg/> </ll>
return local:Order($15)

<ll><an/><as/><cm/><df/><ds/><fg/><fr/><gy/><qn/><ss/><vv/></ll>
```

q16.xq

The function `name($e)` gives the name of the element `$e` (Cfr. later)

4 Typing in XQuery [10]

XQuery Data Model

- Sequences are list of 0 or more items;
- an item is a node or an atomic value;
- a sequence of one item is equiv. with that item;
- a sequencetype consists of a typename and an occurrence indicator;

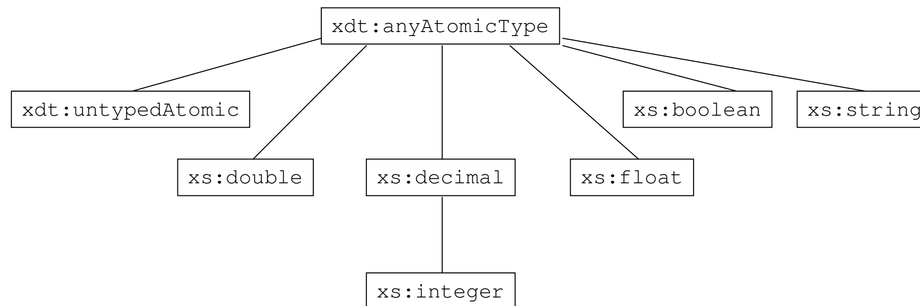
`xs:integer*   xs:integer+   xs:integer?`

Atomic Types

Built-in atomic types are defined in two namespaces:

xs: (http://www.w3.org/2001/XMLSchema)

xdt: (http://www.w3.org/2003/11/xpath-datatypes)



xdt:untypedAtomic

- numbers are double, rest are string
- avoids casting

xs:boolean

- true(), false()

numerical types

- xs:double, xs:decimal, xs:integer, xs:float

Type constructors are used to create values of that type (complex rules for errors)

```
xs:integer("12") => 12
xs:integer(56) => 56
xs:boolean("true") => true()
xs:boolean("1") => true()
xs:boolean("false") => false()
xs:boolean("0") => false()
xs:boolean(other) => error
```

```
12 => 12
xs:integer(<a> 78 </a>) => 78
xs:integer("4.5") => error
xs:float("2.88") => xs:float("2.88")
xs:float("567") => xs:float("567")
xs:float(567) => xs:float("567")
xs:decimal(3.8) => 3.8
xs:decimal("3.8") => 3.8
xs:decimal(3) => 3
xs:decimal("3") => 3
```

Node kinds

Nodes are part of a document or a fragment (whose root is not a document node).

We consider 4 node kinds: element, attribute text, document-node also comment, namespace, processing-instruction.

Every node has type **node()**; other types are **item()**, **element()**, **attribute()**, **document-node()**, **text()**

- **name()** is a function that give the name (type string) of a node;
- **string()** takes a node and gives the concatenation of the string values of all its descendants;
- **data()** takes a sequence of items (with each node having a single atomic value) and gives the sequence of the typed atomic values of the items.
- **boolean()** takes a sequence of items and returns a boolean value;
  empty sequence, empty string, 0, false() => false()
  other sequences => true()
- **instance of** takes a value and a type and verifies whether the value is of that type;

• v **cast as** t has the same meaning as t(v);
• v **castable as** t => **true()** iff v **cast** t gives no error;

```
name(<a/>),
string(<a> cdf <b> 3<c> 4 </c><c>aa</c>1q2</b>;</a>),
data((1, 3.4, 3.4E2, <a>34</a>, <a xsi:type="xs:integer">34</a>)),
boolean(""), boolean(123), boolean((0,0))
=>
"a", " cdf  3 4 aa1q2;", 1, 3.4, 340, xdt:untypedAtomic("34"),
xs:integer("34"), false(), true(), true()
!not in Galax!

<x/> instance of element(),
1 instance of xs:integer,
1 instance of item()*,
(1,2,7) instance of xs:integer*,
(1,2,7) instance of xs:integer+,
(1,2,7) instance of xs:integer?,
(let $s := "Antwerp" return $s instance of xs:string)
=>
true(), true(), true(), true(), true(), false(), true()

"2" cast as xs:integer => 2
"2" castable as xs:integer => true()
```


Function declarations with types

type of the arguments and type of the result are mentioned:

declare function f(p as t1, p2 as t2) as t3 { … };

```
declare function local:seconditin($seq as item()*) as xs:integer
(: gives the second item of a sequence :)
{ $seq[2] };

declare function local:secondinin($seq as xs:integer*) as xs:integer
(: gives the second item of a sequence :)
{ $seq[2] };

declare function local:seconditit($seq as item()*) as item()
(: gives the second item of a sequence :)
{ $seq[2] };

local:seconditin((3, 7, <a>45</a>, 6.7))  => 7
local:seconditin((3, <a>45</a>, 6.7)) => 45
local:seconditin((3, <a>gg</a>, 6.7)) => text: “gg” is not an integer

local:secondinin((3, 7, 45, 6)) => 7
local:secondinin((3, 7, 45, 6.7)) => cannot promote xs:decimal to xs:integer
local:secondinin((3, 7, <a>45</a>, 6.7))  => cannot promote xs:decimal to xs:integer
local:secondinin((3, 7, <a>gg</a>, 6.7))  => text: “gg” is not an integer

local:seconditit((3, 7, <a>45</a>, 6.7)) => 7
local:seconditit((3, <a>45</a>, 6.7)) => <a>45</a>
local:seconditit((3, <a>gg</a>, 6.7)) => <a>gg</a>
local:seconditit((3, 7, 45, 6)) => 7
local:seconditit((3, 7, 45, 6.7)) => 7
local:seconditit((3, 7, <a>45</a>, 6.7)) => 7
local:seconditit((3, 7, <a>gg</a>, 6.7)) => 7
```


Convert all attributes to elements

q07.xq

```
declare function local:convertAttribute($a as attribute()*) as element()* {
  for $attrib in $a
  let $name := name($attrib)
  return
     <element name="{$name}">
     {data($attrib)}
     </element>
};

declare function local:convertElement($e as element()*) as element()* {
  for $el in $e
  let $name := name($el)
  return
    <element name="{$name}">{
      local:convertAttribute($el/@*),
      if (empty($el/*)) then $el/text()
      else local:convertElement($el/*)
    }</element>
};

local:convertElement(document("transcripts.xml")//Transcript)
```

Fails for elements with mixed (elements & text) content.

```
<element name="Transcript">
  <element name="Student">
    <element name="StudId">111111111</element>
    <element name="Name">John Doe</element>
  </element>
  <element name="CrsTaken">
    <element name="CrsCode">CS308</element>
    <element name="Semester">F1990</element>
    <element name="Grade">B</element>
  </element>
  <element name="CrsTaken">
    <element name="CrsCode">MAT123</element>
    <element name="Semester">F1997</element>
    <element name="Grade">B</element>
  </element>
  <element name="CrsTaken">
    <element name="CrsCode">EE101</element>
    <element name="Semester">F1997</element>
    <element name="Grade">A</element>
  </element>
  <element name="CrsTaken">
    <element name="CrsCode">CS305</element>
    <element name="Semester">F1995</element>
    <element name="Grade">A</element>
  </element>
</element>,
```

```
<element name="Transcript">
  <element name="Student">
    <element name="StudId">987654321</element>
    <element name="Name">Bart Simpson</element>
  </element>
  <element name="CrsTaken">
    <element name="CrsCode">CS305</element>
    <element name="Semester">F1995</element>
    <element name="Grade">C</element>
  </element>
  <element name="CrsTaken">
    <element name="CrsCode">CS308</element>
    <element name="Semester">F1994</element>
    <element name="Grade">B</element>
  </element>
</element>
```


Convert all nodes to elements q12.xq

```
declare function local:convertNode($n as node()) as element() {
    typeswitch ($n)
    case attribute() return <attribute name="{name($n)}">{
          data($n)
    }</attribute>
    case element() return <element name="{name($n)}">{
          for $attr  in $n/@*      return local:convertNode($attr),
          for $child in $n/node() return local:convertNode($child)
    }</element>
    case text() return <text>{
          data($n)
    }</text>
    default return <other>{
          data($n)
    }</other>
};

local:convertNode(document("mixed.xml")/mixed)
```

- Can be used with user-defined types in imported schemas
- Supports mixed content model

```
<?xml version="1.0"?>
<mixed>
          <head>
                    <title>Mixed Content Example</title>
          </head>

          <body>
          <p align="center">This paragraph contains
          <strong>mixed content</strong> which
          is no more than <i>a piece of data mixed with some tags</i>.</p>
          </body>
</mixed>
```

becomes next slide


```
<element name="mixed">
  <text>
        </text>
  <element name="head">
    <text>
                </text>
    <element name="title"><text>Mixed Content Example</text></element>
    <text>
        </text>
  </element>
  <text>
        </text>
  <element name="body">
    <text>
        </text>
    <element name="p">
      <attribute name="align">center</attribute>
      <text>This paragraph contains
        </text>
      <element name="strong"><text>mixed content</text></element>
      <text> which
        is no more than </text>
      <element name="i">
        <text>a piece of data mixed with some tags</text>
      </element>
      <text>.</text>
    </element>
    <text>
        </text>
  </element>
  <text>
</text>
</element>
```

5 Document Type Definitions [4]

- DTD is a grammar that specifies **valid** XML-documents;
- XML-documents do not need to have a DTD, nor do they need to be valid;

An attribute can be declared of type **CDATA**, **ID**, **IDREF** or **IDREFS**;

- if `attr1` and `attr2` are declared of type `ID` then `<elem1 attr1="abc" />` and `<elem2 attr2="abc" />` cannot occur in the same document;
- an attribute of type `IDREF` must refer to an ID-value in the same document;
  if there is an `a` with `<a a1="abc"/>` and `a1` of type `IDREF` then there is a `b` with `<b a2="abc"/>` and `a2` of type `ID`
- an attribute of type `IDREFS` represents a space-separated list of references to ID-values in the same document;
  if there is an `a` with `<a a1="abc def"/>` and `a1` of type `IDREFS` then there is a `b` and a `c` with `<b a2="abc"/>` `<c a3="def"/>` and `a2` and `a3` of type `ID`

```
<!DOCTYPE PersonList [
  <!ELEMENT PersonList (Title,Contents)>
  <!ELEMENT Title EMPTY>
  <!ELEMENT Contents (Person*)>
  <!ELEMENT Person ((Name,Id,Address)|(Name))>
  <!ELEMENT Name (#PCDATA)>
  <!ELEMENT Id (#PCDATA)>
  <!ELEMENT Address (Number,Street)>
  <!ELEMENT Number (#PCDATA)>
  <!ELEMENT Street (#PCDATA)>
  <!ATTLIST PersonList Type CDATA #IMPLIED
                       Date CDATA #IMPLIED>
  <!ATTLIST Title Value CDATA #REQUIRED>
]>
```

- the order of the subelements has to be obeyed;
- * [0,∞[, + [1, ∞[, ?[0,1], | alternatives;
- **#PCDATA** : (Parsed Character Data) : character string for elements (unquoted);
- **CDATA** : (Character Data) : character string for attributes (quoted);
- **IMPLIED** : optional;
- **REQUIRED** : mandatory;

DTD for running example 2

```
<!DOCTYPE Report [
  <!ELEMENT Report (Students,Classes,Courses)>
  <!ELEMENT Students (Student*)>
  <!ELEMENT Classes (Class*)>
  <!ELEMENT Courses (Course*)>
  <!ELEMENT Student (Name,Status,CrsTaken*)>
  <!ELEMENT Name (First,Last)>
  <!ELEMENT First (#PCDATA)>
  .
  .
  .
  <!ELEMENT CrsTaken EMPTY>
  <!ELEMENT Class (CrsCode,Semester,ClassRoster)>
  <!ELEMENT Course (CrsName)>
  .
  .
  .
  <!ELEMENT ClassRoster EMPTY>
  <!ATTLIST Report Date #IMPLIED>
  <!ATTLIST Student Studld ID #REQUIRED>
  <!ATTLIST Course CrsCode ID #REQUIRED>
  <!ATTLIST CrsTaken CrsCode IDREF #REQUIRED>
  <!ATTLIST CrsTaken Semester IDREF #REQUIRED>
  <!ATTLIST ClassRoster Members IDREFS #IMPLIED>
]>
```

6 XML Schema [1,2]

- XML Schema is as a DDL for XML-documents;

  it describes the structure of other instance XML-documents;
- Advantages over DTD :
  - uses the same syntax as XML-documents;
  - integrates namespace mechanism;
  - built-in types;
  - complex types can be built from simple types;
  - references can be typed;
  - supports keys and referential integrity constraints;
  - same element name can have different types depending where the element name is nested;
  - XML data do not need to be ordered;

- elements and attributes have types;
- elements that contain subelements have complex types;
- elements with attributes have complex types;
- other elements have simple types;
- attributes have simple types;
- types are given names or are anonymous;
- schema is defined in a schema XML document;
- we presume (for the moment (Cfr. later)) that the instance
- document is not referring to the schema document;

```
<?xml version="1.0"?>
<purchaseOrder orderDate="2004-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <comment>Hurry, my lawn is going wild!</comment>
  <items>
    <item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </item>
    <item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>2004-12-21</shipDate>
    </item>
  </items>
</purchaseOrder>
```

Document on file ‘po.xml’, running example 3

Schema document on file ‘po.xsd’, running example 3

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Purchase order schema for Example.com.
      Copyright 2000 Example.com. All rights reserved.
    </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="purchaseOrder" type="PurchaseOrderType"/>

  <xsd:element name="comment" type="xsd:string"/>

  <xsd:complexType name="PurchaseOrderType">
    <xsd:sequence>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="billTo" type="USAddress"/>
      <xsd:element ref="comment" minOccurs="0"/>
      <xsd:element name="items" type="Items"/>
    </xsd:sequence>
    <xsd:attribute name="orderDate" type="xsd:date"/>
  </xsd:complexType>

  <xsd:complexType name="USAddress">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string"/>
      <xsd:element name="street" type="xsd:string"/>
      <xsd:element name="city" type="xsd:string"/>
      <xsd:element name="state" type="xsd:string"/>
      <xsd:element name="zip" type="xsd:decimal"/>
    </xsd:sequence>
    <xsd:attribute name="country" type="xsd:NMTOKEN"/>
  </xsd:complexType>
```

```
<xsd:complexType name="Items">
    <xsd:sequence>
      <xsd:element name="Item" minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productName" type="xsd:string"/>
            <xsd:element name="quantity">
              <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                  <xsd:maxExclusive value="100"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="USPrice" type="xsd:decimal"/>
            <xsd:element ref="comment" minOccurs="0"/>
            <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="partNum" type="SKU" use="required"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>

  <!-- Stock Keeping Unit, a code for identifying products -->
  <xsd:simpleType name="SKU">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d{3}-[A-Z]{2}"/>
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>
```


- `<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">` is the XML Schema namespace;
- annotation gives info for human readers:

```
<xsd:annotation>
  <xsd:documentation xml:lang="en">
   Purchase order schema for Example.com.
   Copyright 2000 Example.com. All rights reserved.
  </xsd:documentation>
</xsd:annotation>
```

- complex type example ("sequence" Cfr. later) :

```
<xsd:complexType name="USAddress">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="street" type="xsd:string"/>
    <xsd:element name="city" type="xsd:string"/>
    <xsd:element name="state" type="xsd:string"/>
    <xsd:element name="zip" type="xsd:decimal"/>
  </xsd:sequence>
  <xsd:attribute name="country" type="xsd:NMTOKEN"/>
</xsd:complexType>
```

This is an XML element with 2 subelements. It specifies the type 'USAddress'. All elements in the Instance document with type "USAddress" must have a value 'satisfying' this type declaration.

- must have 5 subelements in the specified order;
- may have a 'country' attribute;

```
<xsd:element ref="comment" minOccurs="0"/>
```

References an existing element ‘comment’ that must be declared as a global element (ie. children of the `<schema>` element). There may be comment element in the instance document (`minOccurs="0"`).

- `<xsd:element>` has attributes `name`, `type`, `ref`, `minOccurs`, `maxOccurs`;
  - `minOccurs` is a nonnegative integer, `maxOccurs` is a nonnegative integer or `unbounded`; their default value is 1;
- `<xsd:attribute>` has attributes `name`, `type`, `use`;
  - `use` is `"required"` or `"optional"`; default `optional`

```
<xsd:attribute name="partNum" type="SKU" use="required"/>
```


Simple Types

- Built-in XML simple types: `"string"`, `"byte"`, `"integer"`, `"long"`, `"decimal"`, `"float"`, `"double"`, `"boolean"`, `"dateTime"`, `"ID"`, `"IDREF"`, `"IDREFS"`, `"anyType"`, …

`"anyType"` is the universal type;

- Restriction of built-in simple types

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

The element “simple type”” has a subelement “restriction” with two subelements (called facets)

```
<xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Regular expression Cfr. [2]

```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>
```

Which facets can be combined with which built-in simple types, Cfr. [2].

- List types: lists of built-in simple types or restrictions of built-in simple types

```
<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

`<listOfMyInt>20003 15037 95977 95945</listOfMyInt>` is an element of type "listOfMyIntType".

Several facets can be applied to list types: `length`, `minlength`, `maxlength`.

```
<xsd:simpleType name="USStateList">
  <xsd:list itemType="USState"/>
</xsd:simpleType>

<xsd:simpleType name="SixUSStates">
  <xsd:restriction base="USStateList">
    <xsd:length value="6"/>
  </xsd:restriction>
</xsd:simpleType>
```

`<sixStates>PA NY CA NY LA AK</sixStates>` is an element of type "SixUSStates".

Remark that a space delimites the elements of a list. Hence

```
<a> Paredaens Van Sant </a>
```

is not an element of type twoNames

```
<xsd:simpleType name="stringList">
  <xsd:list itemType="string"/>
</xsd:simpleType>

<xsd:simpleType name="twoNames">
  <xsd:restriction base="stringList">
    <xsd:length value="2"/>
  </xsd:restriction>
</xsd:simpleType>
```

• Union types
the value of elements or attributes with a union type has a type drawn from the union of multiple built-in types, restrictions or list types.

```
<xsd:simpleType name="myInteger">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<xsd:simpleType name="listOfMyIntType">
  <xsd:list itemType="myInteger"/>
</xsd:simpleType>
```

```
<xsd:simpleType name="USState">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="AK"/>
    <xsd:enumeration value="AL"/>
    <xsd:enumeration value="AR"/>
    <!-- and so on ... -->
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="zipUnion">
  <xsd:union memberTypes="USState listOfMyIntType"/>
</xsd:simpleType>
```

`<zips>CA</zips>` is of type “zipUnion”
`<zips>95630 95977 95945</zips>` is of type “zipUnion”
`<zips>AK 78997</zips>` is NOT of type “zipUnion”
`<zips>AK CA</zips>` is NOT of type “zipUnion”

Complex Types

Types can be named (and declared separately) or anonymous.

Anonymous Complex Types

• declaring elements with only attributes:

```
<xsd:element name="internationalPrice">
  <xsd:complexType>
    <xsd:attribute name="currency" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:decimal"/>
  </xsd:complexType>
</xsd:element>
```

`<internationalPrice currency="EUR" value="423.46"/>`
is of the type above.

• Groups of elements
  • “sequence”: an ordered set of elements and choices;
  • “choice”: one element of the given set of elements and sequences;
  • “all”: an unordered set of elements;
  • “sequence” within a “choice” and “choice” within a “sequence” can have “minOccurs” and “maxOccurs” attributes.

“sequence”:
  • can only contain elements or choice-groups;
  • they have to occur (taking into account “minOccurs” and “maxOccurs”) in the given order;
  • “minOccurs” must be nonneg. (default 1), “maxOccurs” must be nonneg. or “unbounded” (default 1)


```
<xsd:complexType name="Items">
  <xsd:sequence>
    <xsd:element name="Item" minOccurs="0" maxOccurs="unbounded">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="productName" type="xsd:string"/>
          <xsd:element name="quantity">
            <xsd:simpleType>
              <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxExclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="USPrice" type="xsd:decimal"/>
          <xsd:element ref="comment" minOccurs="0"/>
          <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

"choice":

- can only contain elements or sequence-groups;
- only at most one can occur (taking into account "minOccurs" and "maxOccurs");
- "minOccurs" must be nonneg. (default 1), "maxOccurs" must be nonneg. or "unbounded" (default 1)

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:sequence minOccurs="0">
        <xsd:element name="shipTo" type="USAddress"/>
        <xsd:element name="billTo" type="USAddress"/>
      </xsd:sequence>
      <xsd:element name="singleUSAddress" type="USAddress"/>
    </xsd:choice>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="Items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>
```


"all":

- can only contain elements;
- they occur in an arbitrary order (taking into account "minOccurs" and "maxOccurs");
- "minOccurs" must be 0 or 1 (default 1), "maxOccurs" must be 1

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:all>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="Items" type="Items"/>
  </xsd:all>
</xsd:complexType>
```

Illegal:

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:all>
      <xsd:element name="shipTo" type="USAddress"/>
      <xsd:element name="Items" type="Items"/>
    </xsd:all>
    <xsd:sequence>
      <xsd:element ref="comment" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:sequence>
</xsd:complexType>
```

• mixed: allows text appearing between elements and their child elements;

```
<xsd:element name="letterBody">
  <xsd:complexType mixed="true">
    <xsd:sequence>
      <xsd:element name="salutation">
        <xsd:complexType mixed="true">
          <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="quantity" type="xsd:positiveInteger"/>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="shipDate" type="xsd:date" minOccurs="0"/>
      <!-- etc. -->
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<letterBody>
<salutation>Dear Mr.<name>Robert Smith</name>.</salutation>
Your order of <quantity>1</quantity> <productName>Baby
Monitor</productName> shipped from our warehouse on
<shipDate>2004-12-21</shipDate>. ....
</letterBody>
```

is declared in the way above.

• General form of an anonymous complex type

```
<complexType name="…" mixed="…">
  (choice|all|sequence)
  (<attribute … >)*
</complexType>
```

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:all>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
  </xsd:all>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<purchaseOrder orderDate="2004-04-29">
  <billTo> … </billTo>
  <shipTo> … </shipTo>
</purchaseOrder>
```

is of the type above.

Named Types and Groups

```
<xsd:complexType name="PurchaseOrderType">
  <xsd:sequence>
    <xsd:choice>
      <xsd:group ref="shipAndBill"/>
      <xsd:element name="singleUSAddress" type="USAddress"/>
    </xsd:choice>
    <xsd:element ref="comment" minOccurs="0"/>
    <xsd:element name="Items" type="Items"/>
  </xsd:sequence>
  <xsd:attribute name="orderDate" type="xsd:date"/>
</xsd:complexType>

<xsd:group name="shipAndBill">
  <xsd:sequence>
    <xsd:element name="shipTo" type="USAddress"/>
    <xsd:element name="billTo" type="USAddress"/>
  </xsd:sequence>
</xsd:group>

<xsd:complexType name="Items">
  <xsd:sequence>
   …
  </xsd:sequence>
</xsd:complexType>
```


XML Schema and Namespaces

- An XML schema-document starts with the declaration of namespaces :
  - **http://www.w3.org/2001/XMLSchema**
    - gives the names of the tags, attributes, types in the schema-document
    - ex.: **schema**, **attribute**, **element**, …
  - target namespace
    - gives the names defined by the schema-document
    - ex.: **CrsTaken**, **Student**, **Status**, …

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:po="http://www.example.com/PO1"
        targetNamespace="http://www.example.com/PO1">

  <element name="purchaseOrder" type="po:PurchaseOrderType"/>

  <element name="comment" type="string"/>

  <complexType name="PurchaseOrderType">
    <sequence>
      <element name="shipTo" type="po:USAddress"/>
      <element name="billTo" type="po:USAddress"/>
      <element ref="po:comment" minOccurs="0"/>
      <element name="Items" type="Items"/>
    </sequence>
    <attribute name="orderDate" type="date"/>
  </complexType>

  <complexType name="USAddress">
    <sequence>
      <element name="name" type="string"/>
      <element name="street" type="string"/>
      <element name="city" type="string"/>
      <element name="state" type="string"/>
      <element name="zip" type="decimal"/>
    </sequence>
    <attribute name="country" type="NMTOKEN"/>
  </complexType>

   …
</schema>
```

running example 3


```
<?xml version="1.0"?>
<apo:purchaseOrder xmlns:apo="http://www.example.com/PO1"
                   orderDate="2004-10-20">
  <shipTo country="US">
    <name>Alice Smith</name>
    <street>123 Maple Street</street>
    <city>Mill Valley</city>
    <state>CA</state>
    <zip>90952</zip>
  </shipTo>
  <billTo country="US">
    <name>Robert Smith</name>
    <street>8 Oak Avenue</street>
    <city>Old Town</city>
    <state>PA</state>
    <zip>95819</zip>
  </billTo>
  <apo:comment>Hurry, my lawn is going wild!</apo:comment>
  <Items>
    <Item partNum="872-AA">
      <productName>Lawnmower</productName>
      <quantity>1</quantity>
      <USPrice>148.95</USPrice>
      <comment>Confirm this is electric</comment>
    </Item>
    <Item partNum="926-AA">
      <productName>Baby Monitor</productName>
      <quantity>1</quantity>
      <USPrice>39.98</USPrice>
      <shipDate>2004-12-21</shipDate>
    <Item>
  </Items>
</apo:purchaseOrder>
```

- In the corresponding XML instance-documents we first declare the target namespace of the schema

running example 3

Key - Refkey

```
<xs:element name="vehicle">
  <xs:complexType> . . .
    <xs:attribute name="plateNumber" type="xs:integer"/>
    <xs:attribute name="state" type="twoLetterCode"/>
  </xs:complexType>
</xs:element>
<xs:element name="state">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="code" type="twoLetterCode"/>
      <xs:element ref="vehicle" maxOccurs="unbounded"/>
      <xs:element ref="person" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:key name="reg">
    <!-- vehicles are keyed by their plate within states -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@plateNumber"/>
  </xs:key>
</xs:element>
<xs:element name="root">
  <xs:complexType>
    <xs:sequence> . . .
      <xs:element ref="state" maxOccurs="unbounded"/> . . .
    </xs:sequence>
  </xs:complexType>
```



```
  <xs:key name="state"> <!-- states are keyed by their code -->
    <xs:selector xpath=".//state"/>
    <xs:field xpath="code"/>
  </xs:key>
  <xs:keyref name="vehicleState" refer="state">
    <!-- every vehicle refers to its state -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@state"/>
  </xs:keyref>
  <xs:key name="regKey">
    <!-- vehicles are keyed by a pair of state and plate -->
    <xs:selector xpath=".//vehicle"/>
    <xs:field xpath="@state"/>
    <xs:field xpath="@plateNumber"/>
  </xs:key>
  <xs:keyref name="carRef" refer="regKey">
    <!-- people's cars are a reference -->
    <xs:selector xpath=".//car"/>
    <xs:field xpath="@regState"/>
    <xs:field xpath="@regPlate"/>
  </xs:keyref>
</xs:element>
<xs:element name="person">
  <xs:complexType>
    <xs:sequence> . . .
      <xs:element name="car">
        <xs:complexType>
          <xs:attribute name="regState" type="twoLetterCode"/>
          <xs:attribute name="regPlate" type="xs:integer"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Each state, within the document, has a different code child value:

```
<xs:key name="state"> <!-- states are keyed by their code -->
  <xs:selector xpath=".//state"/>
  <xs:field xpath="code"/>
</xs:key>
```

element information item: <root>
target node set: <state>–elements within the <root>–element
key-sequence: for each such state, its <code>–child value

The selector, with the element information item as the context node, evaluates to a node-set. Call this the target node set. For each node in the target node set all of the fields, with that node as the context node, evaluate to exactly one member, which must have a simple type. Call the sequence of values of the element and/or attribute information items in those node-sets in order the key-sequence of the node.

Key.
No two members of the target node set have key-sequences whose members are pairwise equal.

Each vehicle within the document has a different state-platenumber pair:

```
<xs:key name="regKey">
  <!-- vehicles are keyed by a pair of state and plate -->
  <xs:selector xpath=".//vehicle"/>
  <xs:field xpath="@state"/>
  <xs:field xpath="@plateNumber"/>
</xs:key>
```

element information item: <root>
target node set: <vehicle>–elements within the <root>–element
key-sequence: for each such vehicle, its "state" and "plateNumber" attribute value

Each vehicle has a different platenumber attrib. value within each state:

```
<xs:key name="reg">
  <!-- vehicles are keyed by their plate within states -->
  <xs:selector xpath=".//vehicle"/>
  <xs:field xpath="@plateNumber"/>
</xs:key>
```

element information item: <state>
target node set: <vehicle>–elements within the <state>–element
key-sequence: for each such vehicle, its "plateNumber" attribute value

Each state attribute value of a vehicle within the document must be the code value of a state within the document:

```
<xs:keyref name="vehicleState" refer="state">
  <!-- every vehicle refers to its state -->
  <xs:selector xpath=".//vehicle"/>
  <xs:field xpath="@state"/>
</xs:keyref>
```

element information item: <root>
referenced key: "state"
target node set: <vehicle>-elements within the <root>-element
key-sequence: for each such vehicle, its "state" attribute value
target node set of "state" : <state>-elements within the <root>-element
key-sequence: for each such state, its <code>-child value

Keyref.
For each member m of the target node set, there must be a member m_r in the target set of the referenced key with key-sequence of m = key-sequence of m_r.

Each (regState attribute, regPlate attribute) value of a car within the document, must be a (state attribute, plateNumber attribute) value of a car within the document:

```
<xs:keyref name="carRef" refer="regKey">
  <!-- people's cars are a reference -->
  <xs:selector xpath=".//car"/>
  <xs:field xpath="@regState"/>
  <xs:field xpath="@regPlate"/>
</xs:keyref>
```

element information item: <root>
referenced key: "regKey"
target node set: <car>-elements within the <root>-element
key-sequence: for each such car, its "regState" and "regPlate" attribute value
target node set of "regKey" : <vehicle>-elements within the <root>-element
key-sequence: for each such vehicle, its "state" and "plateNumber" attribute value

Running Example 2

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:adm="http://xyz.edu/Admin"
        targetNamespace="http://xyz.edu/Admin">

  <complexType name="reportType">
    <sequence>
      <element name="Students" type="adm:studentList"/>
      <element name="Classes">
        <complexType>
          <sequence>
            <element name="Class" type="adm:classType"
                     minOccurs="0" maxOccurs="unbounded"/>
          </sequence>
        </complexType>
      </element>
      <element name="Courses" type="adm:courseCatalog"/>
    </sequence>
    <attribute name="Date" type="date"/>
  </complexType>
```



```
<element name="Report" type="adm:reportType">
  <key name="PrimaryKeyForClass">
    <selector xpath="Classes/Class"/>
    <field xpath="@CrsCode"/>
    <field xpath="@Semester"/>
  </key>

  <keyref name="NoBogusTranscripts" refer="adm:PrimaryKeyForClass">
    <selector xpath="Students/Student/CrsTaken"/>
    <field xpath="@CrsCode"/>
    <field xpath="@Semester"/>
  </keyref>
</element>

  <complexType name="studentList">
    <sequence>
      <element name="Student" type="adm:studentType"
               minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
  <complexType name="studentType">
    <sequence>
      <element name="Name" type="adm:personNameType"/>
      <element name="Status" type="adm:studentStatus"/>
      <element name="CrsTaken" type="adm:courseTakenType"
               minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="StudId" type="adm:studentId"/>
  </complexType>
```

```
<complexType name="personNameType">
  <sequence>
    <element name="First" type="string"/>
    <element name="Last" type="string"/>
  </sequence>
</complexType>

<simpleType name="studentStatus">
  <restriction base="string">
    <enumeration value="U1"/>
    …
    <enumeration value="G5"/>
  </restriction>
</simpleType>

<complexType name="courseTakenType">
  <attribute name="CrsCode" type="adm:courseRef"/>
  <attribute name="Semester" type="string"/>
</complexType>

<simpleType name="courseRef">
  <restriction base="IDREF">
    <pattern value="[A-Z]{3}[0-9]{3}"/>
  </restriction>
</simpleType>
```



```
<simpleType name="studentId">
  <restriction base="ID">
    <pattern value="[0-9]{9}"/>
  </restriction>
</simpleType>

<complexType name="classType">
  <sequence>
    <element name="CrsCode" type="adm:courseCode"/>
    <element name="Semester" type="string"/>
    <element name="ClassRoster" type="adm:classListType"/>
  </sequence>
</complexType>

<complexType name="classListType">
  <attribute name="Members" type="adm:studentIds"/>
</complexType>

<simpleType name="studentIds">
  <list itemType="adm:studentRef"/>
</simpleType>
```

```
  <simpleType name="studentRef">
    <restriction base="IDREF">
      <pattern value="[0-9]{9}"/>
    </restriction>
  </simpleType>

  <complexType name="courseCatalog">
    <sequence>
      <element name="Course" type="adm:courseType"
               minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>

  <complexType name="courseType">
    <sequence>
      <element name="Name" type="string"/>
    </sequence>
    <attribute name="CrsCode" type="adm:courseCode"/>
  </complexType>

  <simpleType name="courseCode">
    <restriction base="ID">
      <pattern value="[A-Z]{3}[0-9]{3}"/>
    </restriction>
  </simpleType>
</schema>
```

Advanced Databases - XML © Prof. dr. J. Paredaens 2008 113

Abstract Example

```
<element name="E1">
  <complexType>
    <all>
      <element name="E2">
        <complexType>
          <sequence>
            <element name="E3">
              <complexType>
                <sequence>
                  <element name="E4">
                    <complexType>
                      <sequence>
                        <element name="E5"
                                 type="string"/>
                      </sequence>
                      <attribute name="A"
                                 type="string"/>
                    </complexType>
                  </element>
                  <element name="E6" type="string"/>
                  <element name="E7">
                    <complexType>
                      <all>
                        <element name="E8"
                                 type="string"
                                 maxOccurs="unbounded"/>
                      </all>
                    </complexType>
                  </element>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
    </all>
  </complexType>
<element>
```

Advanced Databases - XML © Prof. dr. J. Paredaens 2008 114

```
E1 { E2 < E3 < E4 A < E5 String >
               E6 String
               E7 { E8 String }
             >
        >
   }
```

The following are legal key declarations of `E1`

```
<key name="K1">
  <selector xpath="E2/E3"/>
  <field xpath="E4/@A"/>
  <field xpath="E6"/>
</key>

<key name="K2">
  <selector xpath="E2/E3/E4"/>
  <field xpath="E5"/>
</key>

<key name="K3">
  <selector xpath="E2/E3"/>
  <field xpath="E4/E5"/>
</key>
```

The following are illegal key declarations of `E1`

```
<key name="K4">
  <selector xpath="E2/E3"/>
  <field xpath="E7/E8"/>
</key>
```

not one member

```
<key name="K5">
  <selector xpath="E2"/>
  <field xpath="E3"/>
</key>
```

the value of `E3` has no simple type

```
<key name="K6">
  <selector xpath="E3"/>
  <field xpath="E4/E5"/>
</key>
```

the selector-path does not start in `E1`

```
<key name="K7">
  <selector xpath="E2"/>
  <field xpath="E4/E5"/>
</key>
```

the field-path does not start in `E2`


```
E1 { E2 < E3 < E4 A < E5 String >
               E6 String
               E7 { E8 String }
             >
        >
   }
```

The following are legal foreign key declarations of `E1`

```
<keyref name="KR1" refer="K2">
  <selector xpath="E2/E3/E4"/>
  <field xpath="@A"/>
</keyref>

<keyref name="KR2" refer="K2">
  <selector xpath="E2/E3"/>
  <field xpath="E6"/>
</keyref>

<keyref name="KR3" refer="K2">
  <selector xpath="E2/E3/E7"/>
  <field xpath="E8"/>
</keyref>
```

The following are illegal foreign key declarations of `E1`

```
<keyref name="KR4" refer="K2">
  <selector xpath="E2"/>
  <field xpath="E3/E4"/>
</keyref>
```

`E4` has no simple type

```
<keyref name="KR5" refer="K2">
  <selector xpath="E2"/>
  <field xpath="E6"/>
</keyref>
```

the field-path does not start in `E2`

7 Light XQuery

- Concise backwards compatible sublanguage of XQuery
- Complete formal description in a couple of pages

- More info at: http://www.adrem.ua.ac.be/~lixquery

8 XSLT