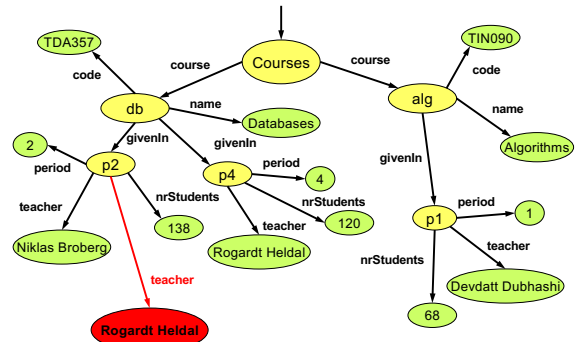


XML

Semistructured data
XML, DTD, (XMLSchema)
XPath, XQuery

Example: A different way of thinking about data...



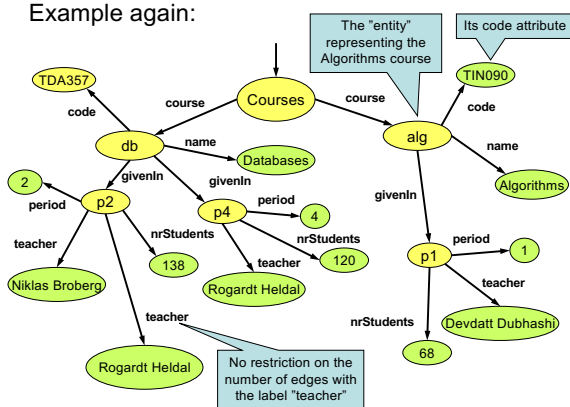
Semi-structured data (SSD)

- More flexible data model than the relational model.
 - Think of an object structure, but with the type of each object its own business.
 - Labels to indicate meanings of substructures.
- Semi-structured: it is structured, but not everything is structured the same way!

SSD Graphs

- Nodes = "objects", "entities"
- Edges with labels represent attributes or relationships.
- Leaf nodes hold atomic values.
- Flexibility: no restriction on
 - Number of edges out from a node.
 - Number of edges with the same label
 - Label names

Example again:



Schemas for SSD

- Inherently, semi-structured data does not have schemas.
 - The type of an object is its own business.
 - The schema is given by the data.
- We can of course restrict graphs in any way we like, to form a kind of "schema".
 - Example: All "course" nodes must have a "code" attribute.

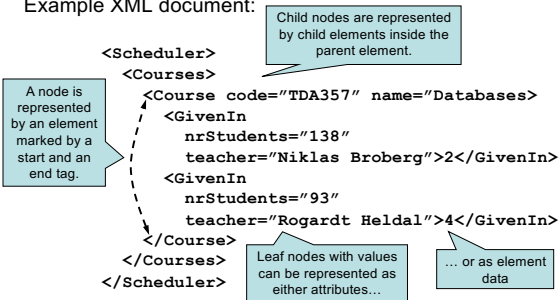
SSD Examples

- XML
 - 90's
 - Case Sensitive
 - `<open_tag>...</close_tag>` or `<tag />` – `<!-- comments -->`
- JSON
 - 2000
 - Collection of key/value pairs (hash table, associative array)
 - Begins with `{` and ends with `}`
 - Each key is followed by `:` (colon) and the key/value pairs are separated by `,` (comma)

XML

- XML = eXtensible Markup Language
- Derives from document markup languages.
 - Compare with HTML: HTML uses "tags" for formatting a document, XML uses "tags" to describe semantics.
- Key idea: create tag sets for a domain, and translate data into properly tagged XML documents.

Example XML document:

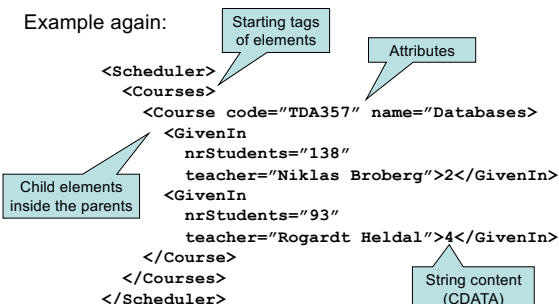


Note that XML is case sensitive!

XML explained

- An XML element is denoted by surrounding tags: `<Course>...</Course>`
- Child elements are written as elements between the tags of its parent, as is simple string content: `<Course><GivenIn>2</GivenIn></Course>`
- Attributes are given as name-value pairs inside the starting tag: `<Course code="TDA357">...</Course>`
- Elements with no children can be written using a short-hand: `<Course code="TDA357" />`

Example again:



Note that XML is case sensitive!

XML namespaces

- XML is used to describe a multitude of different domains. Many of these will work together, but have name clashes.
- XML defines *namespaces* that can disambiguate these circumstances.

– Example:

```

<sc:Scheduler
  xmlns:sc="http://www.cs.chalmers.se/~dbas/xml"
  xmlns:www="http://www.w3.org/xhtml">
  <sc:Course code="TDA357" sc:name="Databases"
    www:name="dbas" />
</sc:Scheduler>
    
```

Use xmlns to bind namespaces to variables in this document.

Quiz!

What's wrong with this XML document?

```
<Course code="TDA357">
  <GivenIn period="2" >
    <GivenIn period="4" >
</Course>
```

No end tags provided for the **GivenIn** elements!
We probably meant e.g. `<GivenIn ... />`

What about the name of the course? Teachers?

Well-formed and valid XML

- Well-Formed:
 - One *root* element
 - Each element must be closed
 - Case sensitive
 - Hierarchy and consistency
 - Attributes between quotes
- Valid:
 - Well-Formed
 - Follows:
 - DTD
 - XML Schema

```
<Employees>
  <Employee>
    <Name>Alberto</name> <NID>34233456-D 35</Age>
    <Salary Moneda="Euro"> 1200 </Employee> </Salary>
  </Employee>
  <Employees>
    ....
  </Employees>
```

DTDs

- DTD = Document Type Definition
- A DTD is a schema that specifies what elements may occur in a document, where they may occur, what attributes they may have, etc.
- Essentially a context-free grammar for describing XML tags and their nesting.

DTD

```
<?xml version="1.0"
standalone="yes"
encoding="utf-8" >
<!-- This is a comment in XML
-->
<Employees>
  <Employee>
    <Name>Alice</Name>
    <NID>34233456-D</NID>
    <Age>35</Age>
    <Salary
Currency="EUR">1200</Salary>
  </Employee>
  <Employee>
    <Name>Bob</Name>
    <NID>31245659-D</NID>
    <Age>29</Age>
    <Salary
Currency="SEK">18000</Salary>
  </Employee>
</Employees>
```

```
<!ELEMENT Employees (Comments?, Employee*)>
<!ELEMENT Comments (#PCDATA)>
<!ELEMENT Employee (Name, NID, Age, Salary)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT NID (#PCDATA)>
<!ELEMENT Age (#PCDATA)>
<!ELEMENT Salary (#PCDATA)>
<!ATTLIST Salary
Currency (EUR | SEK) #Required>
```

Cardinalities:

- ? Optional
- * 0 or more
- + At least 1

Attributes:

- **Optional**
 - **<!ATTLIST Salary Currency>**

- **Required**

- `<!ATTLIST Salary Currency #Required>`
- **Value by default:**
 - `<!ATTLIST Salary Currency (EUR | SEK)`

PCDATA = Parsed Character Data

DTD: ID & IDREF

- DTDs allow references between elements
 - The type of one attribute of an element can be set to **ID**, which makes it unique
 - Another element can have attributes of type **IDREF**, meaning that the value must be an ID in some other element.
 - **IDREFS** is similar to IDREF but it can have more than one value.

```
<?xml version="1.0" encoding="UTF-8"?>
<DOCTYPE Company [
<ELEMENT Company (staff|manager)* >
<ELEMENT staff (name)>
<ELEMENT manager (name)>
<ELEMENT name (#PCDATA)>
<!ATTLIST staff staff_id ID #REQUIRED>
<!ATTLIST staff leader IDREF #IMPLIED>
<!ATTLIST manager manager IDREFS #IMPLIED>
]>
<Company>
  <staff staff_id="s0001">
    <name>Longson</name>
  </staff>
  <staff leader="s0001" staff_id="s0002">
    <name>Carson</name>
  </staff>
  <manager manager="s0001 s0002">
    <name>Samuel</name>
  </manager>
</Company>
```

DTD

Beginning of document with DTD

Document body

```
<?xml version="1.0" encoding="utf-8"
standalone="no" ?>
<!DOCTYPE Scheduler [<ELEMENT
Scheduler(Courses,Rooms)>

<ELEMENT Courses (Course*)>
<ELEMENT Rooms (Room*)>
<ELEMENT Course (GivenIn*)>
<ELEMENT GivenIn (Lecture*)>
<ELEMENT Lecture EMPTY>
<ELEMENT Room EMPTY>

<!ATTLIST Course code ID #REQUIRED
name CDATA #REQUIRED >

<!ATTLIST GivenIn period CDATA
#REQUIRED teacher CDATA #IMPLIED
nrStudents CDATA "0" >

<!ATTLIST Lecture weekday CDATA
#REQUIRED hour CDATA #REQUIRED room
IDREF #IMPLIED >

<!ATTLIST Room name ID #REQUIRED
```

```
<Scheduler>
  <Courses>
    <Course code="TDA357"
      name="Databases">
      <GivenIn period="2"
        teacher="Niklas Broberg"
        nrStudents="138">
        <Lecture weekday="Monday"
          hour="13:15" room="VR" />
        <Lecture weekday="Thursday"
          hour="18:00" room="HB1" />
        </GivenIn>
      <GivenIn period="4"
        teacher="Rogardt Heldal">
        </GivenIn>
      </Course>
    </Courses>
  </Scheduler>
```

Quiz!

What's wrong with DTDs?

- Only one base type – CDATA.
- No way to specify constraints on data other than keys and references.
- No way to specify what elements references may point to – if something is a reference then it may point to any key anywhere.
- DTD is not a XML!!

XML Schema

- Basic idea: why not use XML to define schemas of XML documents?
- XML Schema instances are XML documents specifying schemas of other XML documents.
- XML Schema is much more flexible than DTDs, and solves all the problems listed and more!
- DTDs are still the standard – but XML Schema is the recommendation (by W3)!

Example: fragment of an XML Schema:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="Course">
    <complexType>
      <attribute name="code" use="required" type="string"/>
      <attribute name="name" use="required" type="string"/>
      <sequence>
        <element name="GivenIn" maxOccurs="4">
          <complexType>
            <attribute name="period" use="required">
              <simpleType>
                <restriction base="integer">
                  <minInclusive value="1" />
                  <maxInclusive value="4" />
                </restriction>
              </simpleType>
            </attribute>
            <attribute name="teacher" use="optional" type="string" />
            <attribute name="nrStudents" use="optional" type="integer" />
            <sequence>...</sequence>
          </complexType>
        </element>
      </sequence>
    </complexType>
  </element>
</schema>
```

Multiplicity constraint:
A course can only be given at most four times a year.

Value constraint:
Period must be an integer, restricted to values between 1 and 4 inclusive.

We can have keys and references as well, and any general assertions (though they can be tricky to write correctly).

Quiz!

Write a Document Type Definition (DTD) for this XML:

```
<Question7>
  <Applicants>
    <Applicant name="Andersson" appNum="a1" />
    <Applicant name="Jonsson" appNum="a2" />
    <Applicant name="Larsson" appNum="a3" />
  </Applicants>
  <Choices>
    <Choice applicant="a1" code="MPSOF" choiceNum="1" meritScore="750" />
    <Choice applicant="a1" code="MPALG" choiceNum="2" meritScore="750" />
    <Choice applicant="a1" code="MPCSN" choiceNum="3" meritScore="800" />
    <Choice applicant="a2" code="MPALG" choiceNum="1" meritScore="700" />
    <Choice applicant="a3" code="MPCSN" choiceNum="1" meritScore="850" />
    <Choice applicant="a3" code="MPALG" choiceNum="2" meritScore="850" />
  </Choices>
</Question7>
```

Quiz!

```
<!DOCTYPE Question7 [
  <!ELEMENT Question7 (Applicants, Choices)>

  <!ELEMENT Applicants (Applicant)*>
  <!ELEMENT Applicant EMPTY>
  <!ATTLIST Applicant
    name CDATA #REQUIRED
    appNum ID #REQUIRED >

  <!ELEMENT Choices (Choice)*>
  <!ELEMENT Choice EMPTY>
  <!ATTLIST Choice
    applicant IDREF #REQUIRED
    code CDATA #REQUIRED
    choiceNum CDATA #REQUIRED
    meritScore CDATA #REQUIRED
  >
```

Quiz!

Write a piece of XML that contains the same information as in the example shown above, but which uses nesting, and avoids duplication of applicant identifiers.

```
<Question7>
  <Applicants>
    <Applicant name="Andersson" appNum="a1" />
    <Applicant name="Jonsson" appNum="a2" />
    <Applicant name="Larsson" appNum="a3" />
  </Applicants>
  <Choices>
    <Choice applicant="a1" code="MPSOF" choiceNum="1" meritScore="750" />
    <Choice applicant="a1" code="MPALG" choiceNum="2" meritScore="750" />
    <Choice applicant="a1" code="MPCSN" choiceNum="3" meritScore="800" />
    <Choice applicant="a2" code="MPALG" choiceNum="1" meritScore="700" />
    <Choice applicant="a3" code="MPCSN" choiceNum="1" meritScore="850" />
    <Choice applicant="a3" code="MPALG" choiceNum="2" meritScore="850" />
  </Choices>
</Question7>
```

Quiz!

```
<Question7>
  <Applicant appNum="a1" name="Andersson">
    <Choice meritScore="750" choiceNum="1" code="MPISOF"/>
    <Choice meritScore="750" choiceNum="2" code="MPALG"/>
    <Choice meritScore="800" choiceNum="3" code="MPCSN"/>
  </Applicant>
  <Applicant appNum="a2" name="Jonsson">
    <Choice meritScore="700" choiceNum="1" code="MPALG"/>
  </Applicant>
  <Applicant appNum="a3" name="Larsson">
    <Choice meritScore="850" choiceNum="1" code="MPCSN"/>
    <Choice meritScore="850" choiceNum="2" code="MPALG"/>
  </Applicant>
</Question7>
```

XML query languages

XPath
XQuery

XPath

- XPath is a language for describing paths in XML documents.
 - Think of an SSD graph and *its* paths.
- Path descriptors are similar to path descriptors in a (UNIX) file system.

Symbol	Meaning
/	Root
.	Current Element
..	Parent Element
//*	All elements anywhere
elem1/elem2	Path
[test]	Condition (to filter)
@Att	Attribute

Examples:

```
<?xml version="1.0" standalone="yes" encoding="utf-8" ?>
<!-- This is a comment in XML -->
<Employees>
  <Employee>
    <Name>Alice</Name>
    <NID>34233456-D</NID>
    <Age>35</Age>
    <Salary Currency="EUR">1200</Salary>
  </Employee>
  <Employee>
    <Name>Bob</Name>
    <NID>31245659-D</NID>
    <Age>29</Age>
    <Salary Currency="SEK">18000</Salary>
  </Employee>
</Employees>
```

Employees with salary>1000:

```
/Employees/Employee[Salary>"1000"]
```

Salaries in EUR:

```
//Salary[@Currency="EUR"]/text()
```

NID of employees whose age>35 and their salary>1400 EUR

```
/Employees/Employee[Age>"35"][Salary[@Currency="EUR">"1400"]/NID
```

Axes

- The various directions we can follow in a graph are called *axes* (sing. *axis*).
- General syntax for following an axis is
axis::
 - Example: **/Courses/child::Course**
- Only giving a label is shorthand for **child::label**, while **@** is short for **attribute::**

More axes

- Some other useful axes are:
 - parent:: = parent of the current node.
 - Shorthand is ..
 - descendant-or-self:: = the current node(s) and all descendants (i.e. children, their children, ...) down through the tree.
 - Shorthand is //
 - ancestor::, ancestor-or-self = up through the tree
 - following-sibling:: = any elements on the same level that come *after* this one.
 - ...

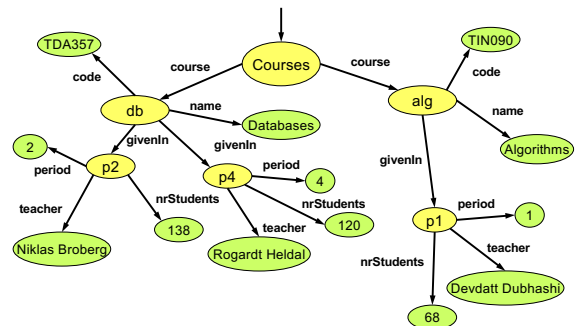
Quiz!

Write an XPath expression that gives the courses that are given in period 2, but with only the GivenIn element for period 2 as a child!

It can't be done!

XPath is not a full query language, it only allows us to specify paths to elements or groups of elements. We can restrict in the path using [] notation, but we cannot restrict further down in the tree than what the path points to.

Example: `/Courses/Course[GivenIn/@period = 2]`



Quiz!

- Write an XPath expression that finds Choice elements where the choice number is 1 and the merit score is greater than 800.

```

<Question7>
  <Applicants>
    <Applicant name="Andersson" appNum="a1" />
    <Applicant name="Jonsson" appNum="a2" />
    <Applicant name="Larsson" appNum="a3" />
  </Applicants>
  <Choices>
    <Choice applicant="a1" code="MPSOF" choiceNum="1" meritScore="750" />
    <Choice applicant="a1" code="MPALG" choiceNum="2" meritScore="750" />
    <Choice applicant="a1" code="MPCSN" choiceNum="3" meritScore="800" />
    <Choice applicant="a2" code="MPALG" choiceNum="1" meritScore="700" />
    <Choice applicant="a3" code="MPCSN" choiceNum="1" meritScore="850" />
    <Choice applicant="a3" code="MPALG" choiceNum="2" meritScore="850" />
  </Choices>
</Question7>

```

Quiz!

`//Choice[@choiceNum="1"and@meritScore>800]`

XQuery

- XQuery is a full-fledged querying language for XML documents.
 - Cf. SQL queries for relational data.
- XQuery is built on top of XPath, and uses XPath to point out element sets.
- XQuery is a W3 recommendation.

XQuery "Hello World"

If our XQuery file contains:

```
<Greeting>Hello World</Greeting>
```

or:

```
let $s := "Hello World"
return <Greeting>{$s}</Greeting>
```

then the XQuery processor will produce the following XML document:

```
<?xml version="1.0" encoding="UTF-8"?>
<Greeting>Hello World</Greeting>
```

Function doc("file.xml")

```
bash$ cat example.xq
doc("courses.xml")
bash$ xquery example.xq
<?xml version="1.0" encoding="UTF-8"?>
<Courses>
  <Course name="Databases" code="TDA357">
    <GivenIn period="2" teacher="Niklas Broberg"/>
    <GivenIn period="4" teacher="Rogardt Haldal"/>
  </Course>
  <Course name="Algorithms" code="TIN090">
    <GivenIn period="1" teacher="Devdatt Dubhashi"/>
  </Course>
</Courses>
```

Quiz!

Write an XQuery expression that puts extra `<Result></Result>` tags around the result, e.g.

```
<Result>
  <Courses>
    <Course name="Databases" code="TDA357">
      <GivenIn period="2" teacher="Niklas Broberg"/>
      <GivenIn period="4" teacher="Rogardt Haldal"/>
    </Course>
    <Course name="Algorithms" code="TIN090">
      <GivenIn period="1" teacher="Devdatt Dubhashi"/>
    </Course>
  </Courses>
</Result>
```

Putting tags around the result

Curly braces are necessary to evaluate the expression between the tags.

```
<Result>{doc("courses.xml")}</Result>
```

Alternatively, we can use a **let** clause to assign a value to a variable. Again, curly braces are needed to get the value of variable `$d`.

```
let $d := doc("courses.xml")
return <Result>{$d}</Result>
```

FLWOR

- Basic structure of an XQuery expression is:
 - FOR-LET-WHERE-ORDER BY-RETURN.
 - Called FLWOR expressions (pronounce as *flower*).
- A FLWOR expression can have any number of FOR (iterate) and LET (assign) clauses, possibly mixed, followed by possibly a WHERE clause and possibly an ORDER BY clause.
- Only required part is RETURN.

Quiz!

What does the following XQuery expression compute?

```
let $courses := doc("courses.xml")
for $gc in $courses//GivenIn
where $gc/@period = 2
return <Result>{$gc}</Result>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Result>
  <GivenIn period="2" teacher="Niklas Broberg"/>
</Result>
```

Quiz!

What does the following XQuery expression compute?

```
let $courses := doc("courses.xml")
let $gc := $courses//GivenIn[@period = 2]
return <Result>{$gc}</Result>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Result>
  <GivenIn period="2" teacher="Niklas Broberg"/>
</Result>
```

Quiz!

What does the following XQuery expression compute?

```
let $courses := doc("courses.xml")
for $c in $courses/Courses/Course
let $code := $c/@code
let $given := $c/GivenIn
where $c/GivenIn/@period = 2
return <Result code="{ $code }">{ $given }</Result>
```

```
<? xml version="1.0" encoding="UTF-8"?>
<Result code="TDA357">
  <GivenIn period="2" teacher="Niklas Broberg"/>
  <GivenIn period="4" teacher="Rogardt Haldal"/>
</Result>
```

Quiz!

Write an XQuery expression that gives the courses that are given in period 2, but with only the **GivenIn** element for period 2 as a child! Courses are in courses.xml

```
let $courses := doc("courses.xml")
for $c in $courses/Courses/Course
let $code := $c/@code, $name := $c/@name
let $given := $c/GivenIn[@period = 2]
where not(empty($given))
return <Course code="{ $code }"
        name="{ $name }">{ $given }</Course>
```

A sequence of elements

The previous examples have all returned a single element. But an XQuery expression can also evaluate to a sequence of elements, e.g.

```
let $courses := doc("courses.xml")
for $gc in $courses/Courses/Course/GivenIn
return $gc
```

```
<GivenIn period="2" teacher="Niklas Broberg"/>
<GivenIn period="4" teacher="Rogardt Haldal"/>
<GivenIn period="1" teacher="Devdatt Dubhashi"/>
```

Putting tags around a sequence

```
let $courses := doc("courses.xml")
let $seq := (
  for $gc in $courses/Courses/Course/GivenIn
  return $gc )
return <Result>{ $seq }</Result>
```

```
<Result>
{
  let $courses := doc("courses.xml")
  for $gc in $courses/Courses/Course/GivenIn
  return $gc
}
</Result>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Result>
  <GivenIn period="2" teacher="Niklas Broberg"/>
  <GivenIn period="4" teacher="Rogardt Haldal"/>
  <GivenIn period="1" teacher="Devdatt Dubhashi"/>
</Result>
```

Quiz!

What will the result of the following XQuery expression be?

```
let $courses := doc("courses.xml")
for $c in $courses/Courses/Course
for $gc in $courses/Courses/Course/GivenIn
return <Info name="{ $c/@name }" teacher="{ $gc/@teacher }" />
```

```
<Courses>
  <Course name="Databases" code="TDA357">
    <GivenIn period="3" teacher="Niklas Broberg" />
    <GivenIn period="2" teacher="Graham Kemp" />
  </Course>
  <Course name="Algorithms" code="TIN090">
    <GivenIn period="1" teacher="Devdatt Dubhashi" />
  </Course>
</Courses>
```

Answer: Cartesian product

Two **for** clauses will iterate over all combinations of values for the loop variables, e.g.

```
let $courses := doc("courses.xml")
for $c in $courses/Courses/Course
for $gc in $courses/Courses/Course/GivenIn
return <Info name="{ $c/@name }" teacher="{ $gc/@teacher }" />
```

```
<Info name="Databases" teacher="Niklas Broberg"/>
<Info name="Databases" teacher="Rogardt Haldal"/>
<Info name="Databases" teacher="Devdatt Dubhashi"/>
<Info name="Algorithms" teacher="Niklas Broberg"/>
<Info name="Algorithms" teacher="Rogardt Haldal"/>
<Info name="Algorithms" teacher="Devdatt Dubhashi"/>
```

Aggregations

XQuery provides the usual aggregation functions: count, sum, avg, min, max.

```
<Result>
{
  count(doc("scheduler.xml")//Room)
}
</Result>
```

```
<Result>
{
  sum(doc("scheduler.xml")//Room/@nrSeats)
}
</Result>
```

Joins in XQuery

We can join two or more documents in XQuery by calling the function doc() two or more times.

```
let $a = doc("a.xml")
let $b = doc("b.xml")
...
(... compare values in $a with values in $b ...)
```

Quiz: what does this XQuery expression compute?

```
<Result>
{
  for $d in ( doc("scheduler.xml"), doc("courses.xml") )
  return $d
}
</Result>
```

Sorting in XQuery

```
<Result>
{
  let $courses := doc("courses.xml")
  for $gc in $courses/Courses/Course/GivenIn
  order by $gc/@period
  return $gc
}
</Result>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<Result>
  <GivenIn period="1" teacher="Devdatt Dubhashi"/>
  <GivenIn period="2" teacher="Niklas Broberg"/>
  <GivenIn period="4" teacher="Rogardt Haldal"/>
</Result>
```

Quantification in XQuery

An XQuery expression might evaluate to a single item or a sequence of items.

```
every variable in expression satisfies condition
some variable in expression satisfies condition
```

Most tests in XQuery, such as the "=" comparison operator, are existentially quantified anyway, so "some" is rarely needed.

Comparing items in XQuery

- The comparison operators eq, ne, lt, gt, le and ge can be used to compare single items.
- If either operand is a sequence of items, the comparison will fail.

Updating XML

- We have corresponding languages for XML and relational databases:
 - SQL DDL ⇔ DTDs or XML Schema.
 - SQL queries ⇔ XQuery
 - SQL modifications ⇔ ??
- XQuery Update is a semi-official extension of XQuery, recommended by W3C.
 - As of June 2009

XQuery Update

- XQuery Update
 - Extends XQuery to support insertions, deletions and updates.
 - Example:

```
for $l in /Scheduler/Courses/Course
  [@code = "TDA357"]/GivenIn
  [@period = 2]/Lectures
where $l/@hour = "08:00"
return
  replace $l/@hour with "10:00"
```

Warning ...

- “Many companies report a strong interest in XML. XML however, is so flexible that this is similar to expressing a strong interest in ASCII characters.”

<http://xml.coverpages.org/BiztalkFrameworkOverviewFinal.html>

Looking to the future

- RDF, RDF Schema, OWL, ...
- Document stores (“NoSQL”)

Summary XML

- XML is used to describe data organized as *documents*.
 - Semi-structured data model.
 - Elements, tags, attributes, children.
 - Namespaces.
- XML can be valid with respect to a schema.
 - DTD: ELEMENT, ATTLIST, CDATA, ID, IDREF
 - XML Schema: Use XML for the schema domain to describe your schema.
- XML can be queried for information:
 - XPath: Paths, axes, selection
 - XQuery: FLWOR.