XML

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

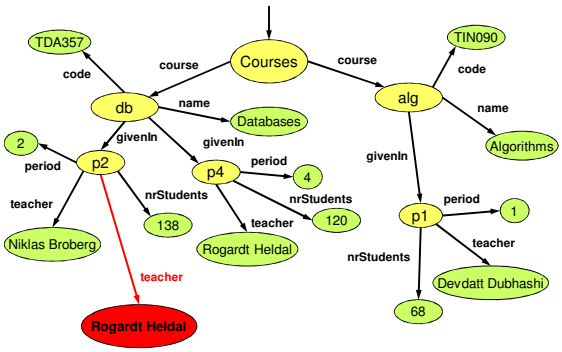
Quiz!

Assume we have a single course (Databases) that is the exception to the rule in that it has two responsible teachers (Niklas Broberg, Rogardt Heldal) when given in the 2nd period. How can we model this?

1. Allow all courses to have two teachers. We extend the GivenCourses table with another attribute teacher2, and put NULL there for all other courses.
2. Allow courses to have any number of teachers. We create a separate table Teaches with attributes course, period and teacher, and make all three be the key.

1 means lots of NULLs, 2 means we must introduce a new table. Seems overkill for such an easy task…

Example: A different way of thinking about data…



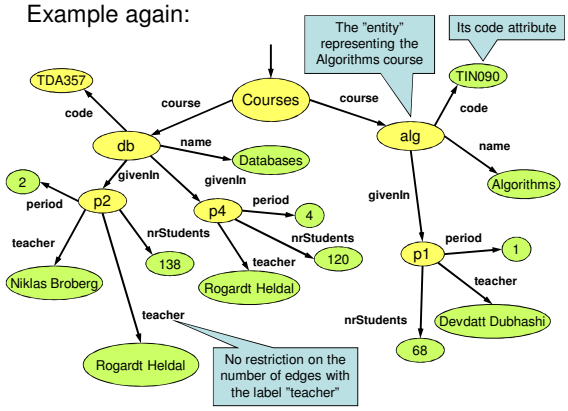
Semi-structured data (SSD)

- 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

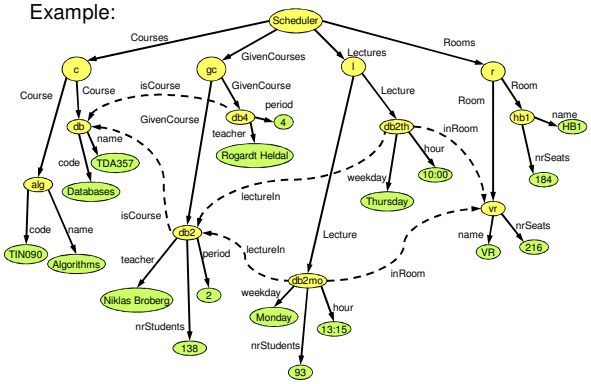
Example again:



Relationships in SSD graphs

- Relationships are marked by edges to some node, that doesn't have to be a child node.
  - This means a SSD graph is not a tree, but a true graph.
  - Cyclic relationships possible.
- Using relationships, it is possible to directly mimic the behavior of the relational model.
  - Graph is three levels deep – one for a relation, the second for its contents, the third for the attributes.
  - References are inserted as relationship edges.
- SSD is a generalization of the relational model!

Example:



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.

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.

XML vs SSD

- XML is a language that describes data and its structure.
  - Cf. relational data: SQL DDL + data in tables.
- The data model behind XML is semi-structured data.
  - Using XML, we can describe an SSD graph as a tagged document.

Example XML document:

Child nodes are represented by child elements inside the parent element.

A node is represented by an element marked by a start and an end tag.

```
<Scheduler>
  <Courses>
    <Course code="TDA357" name="Databases>
      <GivenIn
        nrStudents="138"
        teacher="Niklas Broberg">2</GivenIn>
      <GivenIn
        nrStudents="93"
        teacher="Rogardt Heldal">4</GivenIn>
    </Course>
  </Courses>
</Scheduler>
```

Leaf nodes with values can be represented as either attributes…

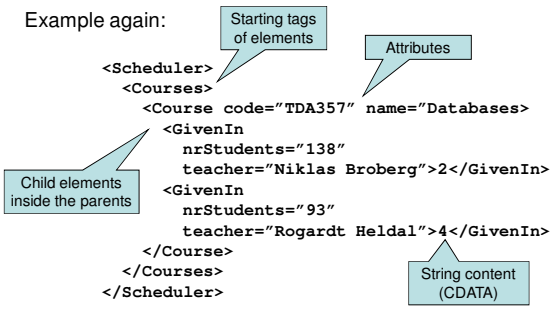
… or as element data

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:




```
<Scheduler>
  <Courses>
    <Course code="TDA357" name="Databases>
      <GivenIn
        nrStudents="138"
        teacher="Niklas Broberg">2</GivenIn>
      <GivenIn
        nrStudents="93"
        teacher="Rogardt Heldal">4</GivenIn>
    </Course>
  </Courses>
</Scheduler>
```

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:

Use xmlns to bind namespaces to variables in this document.

```
<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>
```

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 XML* directly matches semi-structured data:
  - Full flexibility – no restrictions on what tags can be used where, how many, what attributes etc.
  - Well-formed means syntactically correct.
    - E.g. all start tags are matched by an end tag.
- *Valid XML* involves a schema that limits what labels can be used and how.

Well-formed XML

- A document must start with a *declaration*, surrounded by `<? … ?>`
  - Normal declaration is:

```
<?xml version="1.0" standalone="yes" ?>
```

    … where standalone means basically "no schema provided".
- Structure of a document is a *root element* surrounding well-formed sub-documents.

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.

Basic building blocks

- ELEMENT: Define an element and what children it may have.
  - Children use standard regexp syntax: ***** for 0 or more, **+** for 1 or more, **?** for 0 or 1, **|** for choice, commas for sequencing.
  - Example:
```
<!ELEMENT Courses (Course*)>
```
- ATTLIST: Define the attributes of an element.
  - Example:
```
<!ATTLIST Course
  code CDATA #REQUIRED>
```
  - Course elements are required to have an attribute **code** of type **CDATA** (string).

Example: Part of a DTD for the Scheduler domain

```
<!DOCTYPE Scheduler [
  <!ELEMENT Scheduler (Course*)>
  <!ELEMENT Course (GivenIn*)>
  <!ELEMENT GivenIn (#PCDATA)>

  <!ATTLIST Course
    code CDATA #REQUIRED
    name CDATA #REQUIRED
  >
  <!ATTLIST GivenIn
    teacher CDATA #IMPLIED
    nrStudents CDATA "0"
  >
]>
```

A Scheduler element can have 0 or more Course elements as children.

PCDATA means Character Data, i.e. a string. DTDs have (almost) no other base types.

These attributes must be set… (Cf. NOT NULL)

…but not this one.

Default value is 0

Quiz: If we want courses to be able to have more than one teacher, what could we do?

One suggestion is to make a "Teacher" element with PCDATA content, and allow GivenIn elements to have 1 or more of those as children. Period could be an attribute instead.

Non-tree structures

- 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.

```
<!ATTLIST Room
  name ID #REQUIRED>
<!ATTLIST Lecture
  room IDREF #IMPLIED>
```

```
<Scheduler>
 … <Room name="VR" … />
 … <Lecture room="VR" … />
</Scheduler>
```

Beginning of document with DTD

```
<?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
   nrSeats CDATA #IMPLIED >
]>
```

Document body

```
<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="10:00" room="HB1" />
   </GivenIn>
   <GivenIn period="4"
      teacher="Rogardt Heldal">
   </GivenIn>
  </Course>
 </Courses>
 <Rooms>
  <Room name="VR" nrSeats="216"/>
  <Room name="HB1" nrSeats="184"/>
 </Rooms>
</Scheduler>
```

courses.xml (a smaller example)

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE Courses [
 <!ELEMENT Courses (Course*)>
 <!ELEMENT Course (GivenIn*)>
 <!ELEMENT GivenIn EMPTY>
 <!ATTLIST Course
   code ID #REQUIRED
   name CDATA #REQUIRED >
 <!ATTLIST GivenIn
   period CDATA #REQUIRED
   teacher CDATA #IMPLIED >
]>

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

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.
- …

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).

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.
  - A simple path descriptor is a sequence of element names separated by slashes (/).
  - / denotes the root of a document.
  - // means the path can start anywhere in the tree from the current node.

Examples:

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

`/Courses/Course/GivenIn` will return the set of all `GivenIn` elements in the document.

`//GivenIn` will return the same set, but only since we know by our schema that `GivenIn` elements can only appear in that position.

`/Courses` will return the document as it is.

More path descriptors

- There are other path descriptors than / and //:
  - **`*`** denotes any one element:
    - **`/Courses/*/*`** will give all children of all children of a **`Courses`** element, i.e. all **`GivenIn`** elements.
    - **`//*`** will give all elements anywhere.
  - **`.`** denotes the current element:
    - **`/Courses/Course/.`** will return the same elements as **`/Courses/Course`**
  - **`..`** denotes the parent element:
    - **`//GivenIn/..`** will return all elements that have a **`GivenIn`** element as a child.
- Think about how we can traverse the graph – upwards, downwards, along labelled edges etc.

Attributes

- Attributes are denoted in XPath with a **`@`** symbol:
  - **`/Courses/Course/@name`** will give the names of all courses.

Quiz: For the Scheduler example, what will the path expression **`//@name`** result in?

The names of all courses, and the names of all rooms.

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.
  - …

Selection

- We can perform tests in XPath expressions by placing them in square brackets:
  - **`/Courses/Course/GivenIn[@period = 2]`** will give all **`GivenIn`** elements that regard the second period.

Quiz: What will the path expression
`/Courses/Course[GivenIn/@period = 2]`
result in?

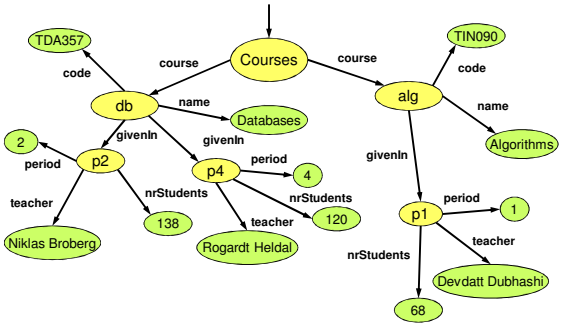
All **`Course`** elements that are given in the second period (but for each of those, all the **`GivenIn`** elements for that course).

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]**



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 Heldal"/>
  </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 Heldal"/>
    </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 Heldal"/>
</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!

```
let $courses := doc("courses.xml")
for $c in $courses/Courses/Course
let $code := $c/@code, $name := $c/@name
let $gc := $c/GivenIn[@period = 2]
where not(empty($gc))
return <Course code="{$code}"
               name="{$name}">{$gc}</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 Heldal"/>
<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 Heldal"/>
  <GivenIn period="1" teacher="Devdatt Dubhashi"/>
</Result>
```

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 Heldal"/>
<Info name="Databases" teacher="Devdatt Dubhashi"/>
<Info name="Algorithms" teacher="Niklas Broberg"/>
<Info name="Algorithms" teacher="Rogardt Heldal"/>
<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 Heldal"/>
</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 ⇔ ??
- There is no standard language for updating XML documents… yet!
  - Plenty of vendor-specific languages though…

XQuery Update

- W3 is working on a language with the working name XQuery Update.
  - Extends XQuery to support insertions, deletions and updates.
  - (as-of-yet-unofficial) Example:

```
update
  for $l in /Scheduler/Courses/Course
            [@code = "TDA357"]/GivenIn
            [@period = 2]/Lectures
  where $l/@hour = "08:00"
    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, …

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.

Exam –XML

"A medical research facility wants a database that uses a semi-structured model to represent different degrees of knowledge regarding the outbreak of epidemic diseases. …"

- Suggest how to model this domain as a DTD (or XML Schema).
- Discuss the benefits of the semi-structured data model for this particular domain.
- Given this DTD, what does this XPath/XQuery expression compute?
- Write an XQuery expression that computes…