COMP60411
Modelling Data on the Web
XPath, XML Schema, and XQuery

Week 3

Bijan Parsia
Uli Sattler
University of Manchester
Week 1 coursework

- All graded!
  - Q1, SE1, M1

- In general,
  - Pay attention to the feedback
    - check the rubrics
    - Try to regenerate
    - Try on other people’s
  - If you don’t understand
    - Come talk to us!
    - We’re happy to explain further
  - Remember, you’ll get essays (and MCQs) on the exam
    - Practice and learn now!
    - It will help!
SE1 General Feedback

• Check the personalised feedback given via BB
• Use a **good spell checker**, and check grammar
• No need to repeat the question or to explain terms introduced or discussed in the lecture, e.g., “conceptual model”
• Structure your essay: either
  – point out ways in which a CM can be useful, make each of these points as clear as possible, e.g., with an example; think whether this is ‘universally true’ or only in certain situations
  – explain why a designing a CM is a waste of time.
• You could have made your statement in 150 words
  – We would appreciate that
• Long conclusions are unnecessary
  – (At most, 1 sentence for summary suffices)
  – (And if you stick to 150 words, that shouldn’t be needed)
M1 & CW1 General Feedback

• Read the specification
  - carefully
  - ask if you’re unsure
  - ask if something is unclear
  - don’t assume
• Work on basic, spec-conform solution first
  - then extend functionality
Last Week

We have encountered many things:

Tree data models:

1. Data Structure formalisms: XML (including name spaces)
2. Schema Language: RelaxNG
3. Data Manipulation: DOM (and Java)

General concepts:

• Semi-structured data
• Self-Describing
• Trees
• Regular Expressions
• Internal & External Representation, Parsing
• Validation, valid, …
• Format

Any Questions?
This Week

• Two new interaction mechanisms:
  – XPath
  – XQuery, extends XPath

• Your second schema language:
  – XML Schema, also known as XSD or WXS

• XSD and XQuery:
  – PSVI and typed queries

• More on Namespaces:
  – Extensibility!
XPath
XML documents...

There are various standards, tools, APIs, data models for XML:

- to **describe** XML documents & **validate** XML document against:
  - we have seen: RelaxNG
  - today: XML Schema

- to parse & **manipulate** XML documents programmatically:
  - we have seen & worked with: DOM (there’s also SAX, etc.)
  - today, we will learn about **XPath** and **XQuery**

- transform an XML document into another XML document or into an instance of another formats, e.g., html, excel, relational tables
  - ....another form of **manipulation**
Manipulation of XML documents

• **XPath** for navigating through and querying of XML documents

• **XQuery**
  – more expressive than XPath, uses XPath
  – for querying and data manipulation
  – Turing complete
  – designed to access large amounts of data, to interface with relational systems

• **XSLT**
  – similar to XQuery in that it uses XPath, ...
  – designed for “styling”, together with XSL-FO or CSS

• contrast this with **DOM** and **SAX**:
  – a collection of APIs for programmatic manipulation
  – includes data model and parser
  – to build your own applications
**XPath**

• designed to navigate to/select parts in a **well-formed** XML document
• no transformational capabilities (as in XQuery and XSLT)
• is a W3C standard:
  – XPath 1.0 is a 1999 W3C standard
  – **XPath 2.0** is a 2007 W3C standard **that extends/is a superset of XPath 1.0**
    • richer set of WXS types & schema sensitive queries
  – XPath 3.0 is a 2014 W3C standard
• allows to select/define *parts* of an XML document: **sequence of nodes**
• uses **path expressions**
  – to navigate in XML documents
  – to select node-lists in an XML document
  – similar to expressions in a traditional computer file system
• provides numerous built-in functions
  – e.g., for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, Boolean values, etc.
• Contrast with SQL!
XPath: Datamodel

- remember how an XML document can be seen as a node-labelled tree
  - with element names as labels: *its DOM tree*

- XPath operates on the abstract, logical tree structure of an XML document, rather than its surface, text syntax
  - *but not on its DOM tree!*

- XPath uses **XQuery/XPath Datamodel**
  - there is a translation at http://www.w3.org/TR/xpath20/#datamodel
    - see XPath process model…
    - it is similar to the DOM tree
      - easier
<table>
<thead>
<tr>
<th>Level</th>
<th>Data unit examples</th>
<th>Information or Property required</th>
</tr>
</thead>
<tbody>
<tr>
<td>cognitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>application</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree adorned with...</td>
<td></td>
<td>nothing</td>
</tr>
<tr>
<td>namespace</td>
<td></td>
<td>well-formedness</td>
</tr>
<tr>
<td>complex</td>
<td>&lt;foo:Name t=&quot;8&quot;&gt;Bob</td>
<td></td>
</tr>
<tr>
<td>simple</td>
<td>&lt;foo:Name t=&quot;8&quot;&gt;Bob</td>
<td></td>
</tr>
<tr>
<td>character</td>
<td>&lt; foo:Name t=&quot;8&quot;&gt;Bob</td>
<td>which encoding (e.g., UTF-8)</td>
</tr>
<tr>
<td>bit</td>
<td>10011010</td>
<td></td>
</tr>
</tbody>
</table>
XPath processing - a simplified view

- **XPath expression** → **XPath parser** → **XPath tree** → **XPath Execution Engine**
- **XML document** → **(Schema-aware) Parser** → **Standard Datamodel** (eg. DOM or XPath)
- **Node Sequence**

- **Input/Output**
- **Generic tools**
XPath processing - a more detailed view
XPath: Datamodel

- the XPath DM uses the following concepts
  - **nodes**: element, attribute, text, namespace, processing-instruction, comment, document (root)
  - **atomic value**: behave like nodes without children or parents, is an atomic value, e.g., xsd:string
  - **item**: atomic values or nodes

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<bookstore>
  <book>
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```
XPath Data Model

```
<?xml version="1.0" encoding="UTF-8"?>
<network>
  <description name="Boston">
    This is the configuration of our network.
  </description>
  <host name="agatha" type="server" os="linux">
    <interface name="eth0" type="Ethernet">
      <arec>agatha.example.edu</arec>
      <cname>mail.example.edu</cname>
      <addr>192.168.0.4</addr>
    </interface>
    <service>SMTP</service>
    <service>POP3</service>
    <service>IMAP4</service>
  </host>
  <host name="gil" type="server" os="linux">
```

From:
Comparison XPath DM and DOM datamodel

- XPath DM and DOM DM are similar, but different
  - most importantly regarding names and values of nodes but also structurally (see ★)
  - in XPath, only attributes, elements, processing instructions, and namespace nodes have names, of form (local part, namespace URI)
  - whereas DOM uses pseudo-names like #document, #comment, #text
  - In XPath, the value of an element or root node is the concatenation of the values of all its text node descendants, not null as it is in DOM:
    - e.g, XPath value of `<a>A<b>B</b></a>` is “AB”

★ XPath does not have separate nodes for CDATA sections (they are merged with their surrounding text)
- XPath has no representation of the DTD
  - or any schema

```
<N>here is some text and <![CDATA[some CDATA < >]]>
</N>
```
XPath: core terms — relation between nodes

• We know trees already:
  – each node has at most one parent
    • each node but the root node has exactly one parent
    • the root node has no parent
  – each node has zero or more children
  – ancestor is the transitive closure of parent, 
    i.e., a node’s parent, its parent, its parent, ...
  – descendant is the transitive closure of child, 
    i.e., a node’s children, their children, their children, ...

• when evaluating an XPath expression $p$, we assume that we know
  – which document and
  – which context we are evaluating $p$ over
  – … we see later how they are chosen/given

• an XPath expression evaluates to a node sequence,
  – a node is a document/element/attribute node or an atomic value
  – document order is preserved among items
<network>
  <description name="Boston">
    This is the configuration of our network in the Boston office.
  </description>
  <host name="agatha" type="server" os="linux">
    <interface name="eth0" type="Ethernet">
      <arec>agatha.example.edu</arec>
      <cname>mail.example.edu</cname>
      <addr>192.168.0.4</addr>
    </interface>
    <service>SMTP</service>
    <service>POP3</service>
    <service>IMAP4</service>
  </host>
  <host name="gil" type="server" os="linux">
    <service>SMTP</service>
  </host>
</network>
XPath - abbreviated syntax by example

XPath expression: `/*[2]`
XPath - abbreviated syntax by example

XPath expression: */*[2]/*[1]/*[3]
XPath - abbreviated syntax know your context node

XPath expression: /*[2]
XPath - abbreviated syntax absolute paths

XPath expression: /*/*[1]
XPath - abbreviated syntax locally globally

XPath expression: //service

<xml version="1.0" encoding="UTF-8"?>
<network>
  <description name="Boston">
    This is the configuration of our network.
  </description>
  <host name="agatha" type="server" os="linux">
    <interface name="eth0" type="Ethernet">
      <arec>agatha.example.edu</arec>
      <cname>mail.example.edu</cname>
      <addr>192.168.0.4</addr>
    </interface>
    <service>SMTP</service>
    <service>POP3</service>
    <service>IMAP4</service>
  </host>
  <host name="gil" type="server" os="linux">
  </host>
</network>
This is the configuration of our network in the Boston office.

XPath - abbreviated syntax

XPath expression: //*/

<?xml version="1.0" encoding="UTF-8"?>
<network>
    <description name="Boston">
        This is the configuration of our network
    </description>
    <host name="agatha" type="server" os="linux">
        <interface name="eth0" type="Ethernet">
            <arec>agatha.example.edu</arec>
            <cname>mail.example.edu</cname>
            <addr>192.168.0.4</addr>
        </interface>
        <service>SMTP</service>
        <service>POP3</service>
        <service>IMAP4</service>
    </host>
    <host name="gil" type="server" os="linux">
        ...
    </host>
</network>
<?xml version="1.0" encoding="UTF-8"?>
<network>
  <description name="Boston">
    This is the configuration of our network.
  </description>
  <host name="agatha" type="server" os="linux">
    <interface name="eth0" type="Ethernet">
      <arec>agatha.example.edu</arec>
      <cname>mail.example.edu</cname>
      <addr>192.168.0.4</addr>
    </interface>
    <service>SMTP</service>
    <service>POP3</service>
    <service>IMAP4</service>
  </host>
  <host name="gil" type="server" os="linu"
Find more about XPATH: read up and play with examples, e.g., in
Contrast with SQL (Just with what you’ve seen!)
XML Schema
another schema language for XML
There is more than 1 schema language

RelaxNG schemas
- S
- S1
- S2

XML Schema schemas
- S3
- S4

all well-formed XML docs
- docs valid wrt S, S1
- docs valid wrt S2, S3
- docs valid wrt S, S1
A more correct picture:

**XML Schema** is an XML schema language with an XML syntax (unlike for RelaxNG, there is no compact syntax)

RelaxNG schemas

- S
- S1
- S2

XML Schema schemas

- S3
- S4

**docs valid wrt S, S1**

**docs valid wrt S2, S3**

- all well-formed XML docs
- all XML docs
Schema languages for XML

provide means to define the legal structure of an XML document

cartoon.rnc, a RelaxNG Schema for cartoon descriptions

```
<?xml version="1.0" encoding="UTF-8"?>
<cartoon copyright="United Feature Syndicate"
  year="2000">
  <prolog>
    <series>Dilbert</series>
    <author>Scott Adams</author>
    <characters>
      <character>The Pointy-Haired Boss</character>
      <character>Dilbert</character>
    </characters>
  </prolog>
  ...
</cartoon>

<?xml version="1.0" encoding="UTF-8"?>
<cartoon copyright="Bill Watterson"
  year="1994">
  <prolog>
    <series>Calvin and Hobbs</series>
    <author>Bill Watterson</author>
    <characters>
      <character>Calvin</character>
      <character>Hobbs</character>
      <character>Snoman</character>
    </characters>
  </prolog>
  ...
</cartoon>
```
Schema languages for XML

A variety of schema languages have been developed for XML; they vary w.r.t.

* their **expressive power**:  
  - “do I have a means to express foo?”  
  - “how hard is it to describe foo?”

* **ease of use/understanding**:  
  - “how easy it is to write a schema?”  
  - “how easy is it to understand a schema written by somebody else?”

* **the complexity of validating** a document w.r.t. a schema:  
  - “how much space/time does it take to verify whether a document is valid w.r.t. a schema (in the size of document and schema)?”  
  - (Mostly for implementors!)
Schema languages for XML

provide means to define the legal structure of an XML document

cartoon.xsd, an XML Schema schema for cartoon descriptions
XML Schema

• XML Schema is also referred to as XML Schema Definition, abbr. **XSD**
  • is a W3C standard, see http://www.w3.org/XML/Schema

• a RNG in compact syntax (or DTD) is **not** a well-formed XML document
  – though you can use the RelaxNG XML format
• an XML Schema **is** a well-formed XML document
  – no human oriented syntax
• XML Schema
  – is *mostly* more expressive than DTDs
  – but *overlaps* with RelaxNG: each has non-shared features
• in contrast to DTDs, XML Schema supports
  – **namespaces**, so we can combine several documents: for schema validation, universal names are used (rather than qualified names)
  – **datatypes**, including simple datatypes for parsed character data and for attribute values, e.g., for *date* (when was 11/10/2006?)
• XML provides more support for describing the (element and mixed) content of elements
XML Schema: a first example

Example with RNG:

```xml
<?xml version="1.0"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <sentOn>2007-01-29</sentOn>
  <body>
    Have a nice weekend!
  </body>
</note>
```

```rnc
default namespace = "http://www.w3schools.com"

element note {
  element to { text },
  element from { text },
  element sentOn { text },
  element body { text }
}
```
XML Schema: a first example

<?xml version="1.0"?>
<note
xmlns="http://www.w3schools.com"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <to>Tove</to>
  <from>Jani</from>
  <sentOn>2007-01-29</sentOn>
  <body>
    Have a nice weekend!
  </body>
</note>

note.xsd:

<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com"
  xmlns="http://www.w3schools.com"
  elementFormDefault="qualified">
  <xsd:element
    name="note">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element
          name="to" type="xs:string"/>
        <xsd:element
          name="from" type="xs:string"/>
        <xsd:element
          name="sentOn" type="xs:date"/>
        <xsd:element
          name="body" type="xs:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
XML Schema: some remarks

- to validate an XML document against an XML schema,
  - we use a **validating XML parser** that supports **XML Schema**
  - e.g., DOM level 2, SAX2

- in XML Schema,
  - each element and type can only be declared once
  - almost all elements can contain an element
    `<xs:annotation>... </xs:annotation>` as their first child: useful, e.g., for
    
    ```xml
    <xs:simpleType name="northwestStates">
      <xs:annotation>
        <xs:documentation>States in the Pacific Northwest of US</xs:documentation>
      </xs:annotation>
      <xs:restriction base="xs:string">
        ...
      </xs:restriction>
    </xs:simpleType>
    ```

- XML Schema provides support for modularity & re-use through
  - `xs:import`
  - `xs:include`
  - `xs:redefine`
XML Schema & Namespaces

• most XML Schema schemas start like this, in note.xsd

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.w3schools.com"
    xmlns="http://www.w3schools.com"
    elementFormDefault="qualified" >

    
</xs:schema>
```

• and a document using such a schema looks like this:

```xml
<?xml version="1.0"?>
<note
    xmlns="http://www.w3schools.com"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  "This document uses a schema"
</note>
```
XML Schema & Namespaces

- XML Schema supports (and uses) **namespaces**
- an XML Schema typically has 2 namespaces:
  - targetNamespace for those **elements defined in schema** and
    - which also might need a separate declaration
  - XMLSchema namespace http://www.w3.org/2001/XMLSchema
  - (and may involve many more!)

```
<?xml version="1.0"?>
<p:note
    xmlns:p="http://www.w3schools.com"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <p:to>Paul</p:to>
</p:note>
```

```
<?xml version="1.0"?>
<note
    xmlns="http://www.w3schools.com"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <to>Paul</to>
</note>
```
XML Schema core concepts: datatypes

Built-In Datatypes and their hierarchy

- in the previous examples, we used 2 Built-in datatypes:
  - `xs:string`
  - `xs:date`

- many more:
  - built-in/atomic/primitive
    e.g., `xs:dateTime`
  - composite/user-defined
    e.g., `xs:lists, xs:union`
  - through restrictions/user-defined
    e.g., `ints < 10`
XML Schema core concepts: datatypes

each XSD datatype comes with a

– **value space**, e.g., for xs:boolean, this is \{true, false\}.
– **lexical space**, e.g., for xs:boolean, this is \{true, false, 1, 0\}, and
– **lexical-to-value** mapping that has to be neither injective nor surjective
  – for xs:boolean, it’s surjective, but not injective
– **constraining facets** that can be used in restrictions of that datatype
  • e.g., maxInclusive, maxExclusive, minInclusive, … for xs:integer
  • e.g., for defining “SmallInteger” or “ShortString”
XML Schema: types

We can define **types** in XSD, in two ways:

- **xs:simpleType** for simple types, to be used for
  - **attribute values** and
  - **elements** without element child nodes and without attributes

- **xs:complexType** for complex types, to be used for
  - **elements** with
    - element content or
    - mixed element content or
    - text content and attributes
XML Schema: type declarations

• can be **anonymous**, e.g., in the definition of age or person below:

```xml
<xs:element name="age">
    <xs:simpleType>
        <xs:restriction base="xs:integer">
            <xs:minInclusive value="3"/>
            <xs:maxInclusive value="7"/>
        </xs:restriction>
    </xs:simpleType>
</xs:element>

<xs:element name="person">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Name" type="Nametype"/>
            <xs:element name="DoB" type="xs:date"/>
        </xs:sequence>
        <xs:attribute name="friend" type="xs:boolean"/>
    </xs:complexType>
</xs:element>
```

• can be **named**, e.g., Agetype or PersonType

```xml
<xs:element name="age" type="AgeType"/>

<xs:simpleType name="AgeType">
    <xs:restriction base="xs:integer">
        <xs:minInclusive value="3"/>
        <xs:maxInclusive value="7"/>
    </xs:restriction>
</xs:simpleType>

<xs:element name="person" type="PersonType"/>

<xs:complexType name="PersonType">
    <xs:sequence>
        <xs:element name="Name" type="Nametype"/>
        <xs:element name="DoB" type="xs:date"/>
    </xs:sequence>
    <xs:attribute name="friend" type="xs:boolean"/>
</xs:complexType>
```
XML Schema: atomic simple types

- are based on the numerous built-in datatypes
- that can be restricted using `xs:restriction` facets, e.g.,

| enumeration | <xs:simpleType name="bikeType">
|             |   <xs:restriction base="xs:string">
|             |     <xs:enumeration value="MTB"/>
|             |     <xs:enumeration value="road"/>
|             |   </xs:restriction>
|             | </xs:simpleType> |
| length      | <xs:simpleType name="eightChar">
|             |   <xs:restriction base="xs:string">
|             |     <xs:length value="8"/>
|             | </xs:restriction>
|             | </xs:simpleType> |
XML Schema: atomic simple types

- are based on the numerous built-in datatypes
- that can be restricted using `xs:restriction` facets, e.g.,

<table>
<thead>
<tr>
<th>Type</th>
<th>XML Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxLength</td>
<td>`&lt;xs:simpleType name=&quot;medStr&quot;&gt;</td>
</tr>
<tr>
<td>minLength</td>
<td>`&lt;xs:restriction base=&quot;xs:string&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>`&lt;xs:minLength value=&quot;5&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>`&lt;xs:maxLength value=&quot;8&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xs:restriction&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xs:simpleType&gt;</td>
</tr>
<tr>
<td>maxExclusive/maxInclusive</td>
<td>`&lt;xs:simpleType name=&quot;age&quot;&gt;</td>
</tr>
<tr>
<td>minExclusive/minInclusive</td>
<td>`&lt;xs:restriction base=&quot;xs:integer&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>`&lt;xs:minInclusive value=&quot;0&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>`&lt;xs:maxInclusive value=&quot;120&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xs:restriction&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xs:simpleType&gt;</td>
</tr>
<tr>
<td>patterns (using regular</td>
<td>`&lt;xs:simpleType name=&quot;simpleStr&quot;&gt;</td>
</tr>
<tr>
<td>expressions close to Perl's)</td>
<td>`&lt;xs:restriction base=&quot;xs:string&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>`&lt;xs:pattern value=&quot;([a-z][A-Z])+&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xs:restriction&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/xs:simpleType&gt;</td>
</tr>
</tbody>
</table>
XML Schema: composite simple types

• we can use built-in datatypes not only in **restrictions**, but also in **compositions** to:
  – `xs:list`
  – `xs:union`

```xml
<xs:simpleType name='myList'>
  <xs:list itemType='xs:integer'/>
</xs:simpleType>

<xs:simpleType name='ShortList'>
  <xs:restriction base='myList'>
    <xs:maxLength value='8'/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name='colourListOrDate'>
  <xs:union memberTypes='colourList xs:date'/>
</xs:simpleType>

<xs:simpleType name='colourList'>
  <xs:list>
    <xs:restriction base='xs:string'>
      <xs:enumeration value='red'/>
      <xs:enumeration value='green'/>
      <xs:enumeration value='blue'/>
    </xs:restriction>
  </xs:list>
</xs:simpleType>
```
XML Schema: simple types

- can be used in
  - element declarations, for elements without element child nodes

```xml
<xs:complexType name="PersonType">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="DoB" type="xs:date"/>
  </xs:sequence>
  <xs:attribute name="friend" type="xs:boolean" default="true"/>
  <xs:attribute name="phone" type="xs:string"/>
</xs:complexType>
```

- we can specify fixed or default values
XML Schema: simple content

• for elements
  – where we cannot use xs:simpleType because of attribute declarations
  – but that have simple (e.g., text) content only,
    ➡️ we can use xs:simpleContent, e.g.

```xml
<xs:element name="size">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country" type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```
XML Schema: complex types

- **element order enforcement constructs:**
  - **sequence:** order preserving
  - **all:** like sequence, but not order preserving
  - **choice:** choose exactly one

- these constructs can be combined with **minOccurs** and **maxOccurs**, 
  - by default, both are set to 1,
  - but they can be set to any non-negative integer or “unbounded”, e.g.

```xml
<xs:complexType name="nametype">
  <xs:sequence>
    <xs:element name="fname" type="xs:string"/>
    <xs:element name="fname" type="xs:string"/>
    <xs:element name="lname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```
XML Schema: mixed content

- to allow for mixed content, set attribute mixed="true", e.g.,

  ```xml
  <xs:complexType name="PersonType" mixed="true">
    <xs:sequence>
      <xs:element name="Name" type="xs:string"/>
      <xs:element name="DoB" type="xs:date"/>
    </xs:sequence>
    <xs:attribute name="friend" type="xs:boolean" default="true"/>
    <xs:attribute name="phone" type="xs:string"/>
  </xs:complexType>
  ```

  but we
  - cannot constrain where the text occurs between elements,
  - can only say that content can be mixed
XML Schema: restriction and extension

• we have already used `xs:extension` and `xs:restriction` both for
  – simple types and
  – complex types
• they are XML Schema’s mechanisms for *inheritance*
• **extension**: specifying a new type X by extending Y
  – this “appends” X’s definition to Y’s, e.g.,

```xml
<xs:simpleType name="AgeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="3"/>
    <xs:maxInclusive value="7"/>
  </xs:restriction>
</xs:simpleType>

<xs:complexType name="NewAgeType">
  <xs:simpleContent>
    <xs:extension base="AgeType">
      <xs:attribute name="range" type="xs:string"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>

<xs:complexType name="PersonType">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="DoB" type="xs:date"/>
  </xs:sequence>
  <xs:attribute name="friend" type="xs:boolean" default="true"/>
  <xs:attribute name="phone" type="xs:string"/>
</xs:complexType>

<xs:complexType name="LongPersonType">
  <xs:complexContent>
    <xs:extension base="PersonType">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```
**XML Schema: restriction and extension**

- **restriction**: easy for simple types
  we have seen it several times

```xml
<xs:simpleType name="AgeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="3"/>
    <xs:maxInclusive value="7"/>
  </xs:restriction>
</xs:simpleType>
```

- **restriction**: “cumbersome” for complex types:
  specifying a new type X by
  restricting a complex type Y
  requires the **reproduction** of
  Y’s definition, e.g.,

```xml
<xs:complexType name="PersonType">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="DoB" type="xs:date"/>
  </xs:sequence>
  <xs:attribute name="friend" type="xs:boolean"/>
  <xs:attribute name="phone" type="xs:string"/>
</xs:complexType>
```

```xml
<xs:complexType name="StrictPersonType">
  <xs:complexContent>
    <xs:restriction base="PersonType">
      <xs:sequence>
        <xs:element name="Name">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:pattern value="[A-Z][a-z]+"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:element>
        <xs:element name="DoB" type="xs:date"/>
      </xs:sequence>
      <xs:attribute name="friend" type="xs:boolean"/>
      <xs:attribute name="phone" type="xs:string"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```
XML Schema: restriction and extension

- **usage:** in a document, an element of a type derived by restriction or extension from Y can be used in place of an element of type Y…
  
  - provided you say so explicitly, e.g., in

  ```xml
  <person phone="2">
    <Name>Peter</Name>
    <DoB>1966-05-04</DoB>
  </person>
  <person xsi:type="LongPersonType" phone="5432" friend="0">
    <Name>Paul</Name>
    <DoB>1967-05-04</DoB>
    <address>Manchester</address>
  </person>
  
  <person xsi:type="LongPersonType" phone="5432" friend="0">
    <Name>Paul</Name>
    <DoB>1967-05-04</DoB>
    <address>Manchester</address>
  </person>
  
  <person xsi:type="LongPersonType" phone="5432" friend="0">
    <Name>Paul</Name>
    <DoB>1967-05-04</DoB>
    <address>Manchester</address>
  </person>
  
  this means that a validating XML parser has to manage a schema’s type hierarchy
  
  - to ensure that `LongPersonType` was really derived by restriction or extension from the type expected for `person`
  
  - but it doesn’t have to “guess” an element’s type from its properties

- In SE3: compare they “pain & gain” of using types to “pain & gain” of using other features like substitution groups!
XML Schema: restriction and extension

- to prevent a type from being instantiated directly, use e.g.,

  `<xs:complexType name="StrictPersonType" abstract="true">`

- to prevent a type from being further extended and/or restricted use e.g.,

  `<xs:complexType name="StrictPersonType" final="#all">`

- closely related to the mechanism of restriction(extension are substitution groups, i.e., a mechanism to allow to replace one element with a group of others
XML Schema: summary of complex types

• we have simple and complex **types**:  
  – simple types for attribute values and text in elements  
  – complex types for elements with child elements or attributes

• we have simple and complex **content** of elements:  
  – simple content:  
    • elements with only text between tags and possibly attributes  
  – complex content  
    • element content (elements only)  
    • mixed content (elements and text)  
    • empty content (at most attributes)

• a complex content type can be specified in 3 ways: using  
  – element order enforcement constructs (all, sequence, choice)  
  – a single child of simpleContent:  
    derive a complex type from a simple or complex type with simple content  
  – a single child of complexContent:  
    derive a complex type from another complex type  
    using restriction or extension
Comparing XML Schema & RelaxNG

- You know one better than the other…one is simpler than the other…
- in RNG, no mechanism for manipulating datatypes, lists, unions,…
  - but you can borrow this from XSD!
- in RNG, no restrictions & extension, no (non-atomic) types
  - in a document, an element of a type derived by restriction from Y can be used in place of an element of type Y
  - this can make writing complex schemas easier!
  - but this means that a validating XML parser has to manage a schema’s type hierarchy
- XML Schema has restrictions on expressing constraints on content models
  - so that matching a node’s childnode sequence against the corresponding content model is “easier”
  - e.g., XML Element Declarations Consistent constraint
- is there a set of XML documents (e.g., your cartoon descriptions)
  - for which we can formulate a RNG
  - but not an XML schema?
  - or the other way round?
Extensibility

is a systemic measure of the ability to extend a system/XML format/XML schema and the level of effort required to implement the extension.

RNGs and extensibility

- **Multiple** RNGs

- **Given a single** RNG, we can easily
  - **loosen** features
    - Choice
    - Repetition (regular expressions!)
    - ANY - for elements of any kind!
    - #IMPLIED and #DEFAULT
  - **tighten** features
    - naturally: every name must have a declaration!
    - No namespace sensitivity
Example

- **Two RNGs**
  - One describing a **superset** of the other
  - Safe for generation
    - Not as safe for consumption
    - But perhaps safe in the right way?

- **Multiple RNGs vs. Well-formedness**
  - Finding a fit
  - Finding a “best” fit
    - Too tight a fit is pointless
    - Too loose can be pointless too!
WXS and Extensibility

- Multiple WXS
  - As with RNGs
  - WXS can relate
    - I.e., A WXS can extend or refine another WXS
    - ...see include and import
    - Just as a type can extend another
      - Inter-schema refinement can do more
  - with namespace support!

- In a single WXS
  - Choice and repetition
  - Wildcards!
    - Strictly more expressive
    - Namespace aware

- (RelaxNG also has modularity and extension features)
Namespaces

• Their fundamental goal:
  – to manage names...
  – provide “Decentralised extensibility”
    • What does this mean?

• Their fundamental limitation:
  – Name extensibility only!
  – Clash prevention only!
    • At least at the technical level...

• Schemas need to be namespace sensitive
  – And to enable more elaborate behavior
XML Schema: Namespaces

• targetNamespace
  – Every WXS has a **targetNamespace**
    • At least implicitly
    • for those elements **defined** in schema
    • It also has a lot of symbol spaces
  – But any `<ws:schema>` has only **one** targetNamespace!
    • We need to relate documents (i.e., DOMs!)
    • a `ws:schema` **component** can have more namespaces!
Some Namespace Patterns

For example

- Contained NS Pattern
- Global Attributes NS Pattern
  - Attributes are weird
- General Extension NS Pattern
- Version
- Abuse

Be sure you understand the difference between
- namespace declarations
- namespaces,
- expanded names,
- namespace scope, etc.
Remember Namespaces?! 

- **Namespace declarations**, e.g., `xmlns:calc="http://bjp.org/calc/"`
  - looks like/can be treated as a **normal** attributes (CW2)

- **Qualified names** (“QNames”), e.g., *calc:+*
  - Prefix, e.g., *calc*
  - Local name, e.g., *plus*

- **Expanded name**, e.g., `{http://bjp.org/calc/}plus`

- **Namespace name**, e.g., *http://bjp.org/calc/*

- **The scope** of a declaration is:
  - The element where the declaration **appears** together with
  - **the descendants** of that element...
    - ...except those descendants which have a **conflicting declaration**
      - (and their descendants, etc.)
    - I.e., a declaration with the same prefix

- **Scopes nest and shadow**
  - Deeper nested declarations redefine/overwrite outer declarations
The Contained Namespace Patterns

• in pattern position where a “context” is shared by subtrees
  – Think SVG in HTML
• where an element with all its attributes and “relevant” descendants share the same namespace and processing
  – a descendant may be the root of a new “context” subtree
  – but then is in a new namespace with its own processing instruction
  – which will also apply to all its descendants, apart from ...

```html
<html xmlns="http://www.w3.org/1999/xhtml">
<head><title>SVG embedded inline in XHTML</title></head>
<body>
<h1>SVG embedded inline in XHTML</h1>
<svg xmlns="http://www.w3.org/2000/svg" width="300" height="200">
  <circle cx="150" cy="100" r="50" />
</svg>
</body>
</html>
```
The Contained Namespace Patterns

another example:

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                 xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 exclude-result-prefixes="xs" version="2.0">
  <xsl:import-schema schema-location="http://ex.org/minischema.xsd"/>
  <xsl:template match="*"/>
    ...
  </xsl:template>
</xsl:stylesheet>
```

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                 xmlns:xs="http://www.w3.org/2001/XMLSchema"
                 exclude-result-prefixes="xs" version="2.0">
  <xsl:import-schema>
    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xs:element name="a" type="Union"/>
      <xs:simpleType name="Union">
        <xs:union memberTypes="xs:integer xs:boolean"/>
      </xs:simpleType>
    </xs:schema>
  </xsl:import-schema>
  <xsl:template match="*"/>
    ...
  </xsl:template>
</xsl:stylesheet>
```
How to Capture in XML Schema?

- **xs:import**
  - Declares a foreign namespace
    - and associated schema (but no prefix for it: the schema does this!)

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
           targetNamespace="http://www.w3.org/1999/XSL/Transform" elementFormDefault="qualified">


<xs:element name="import-schema" substitutionGroup="xsl:declaration">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="xsl:element-only-versioned-element-type">
        <xs:sequence>
          <xs:element ref="xs:schema" minOccurs="0" maxOccurs="1"/>
        </xs:sequence>
        <xs:attribute name="namespace" type="xs:anyURI"/>
        <xs:attribute name="schema-location" type="xs:anyURI"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>

...<xs:attribute name="schema-location" type="xs:anyURI"/>
</xs:extension>
</xs:complexType>
</xs:element>
...
</xs:schema>
```

Brings in the foreign namespace and its declarations

And we can now use elements declared
How to Capture in XML Schema?

- **xs:import**
  - Declares a foreign namespace
  - and associated schema

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
           targetNamespace="http://www.w3.org/1999/XSL/Transform"
           elementFormDefault="qualified">
  <xs:import namespace="http://www.w3.org/2001/XMLSchema"
              schemaLocation="http://www.w3.org/2001/XMLSchema.xsd"/>
...
  <xs:element name="import-schema" substitutionGroup="xsl:declaration">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xsl:element-only-versioned-element-type">
          <xs:sequence>
            <xs:element ref="xs:schema" minOccurs="0" maxOccurs="1"/>
          </xs:sequence>
          <xs:attribute name="namespace" type="xs:anyURI"/>
          <xs:attribute name="schema-location" type="xs:anyURI"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
...</xs:schema>
```

If you delete this

then this breaks

but this is still ok
How to Capture in XML Schema?

- Strange:
  - `xmlns` declares the namespace binding
  - `xs:import` makes that namespace “schema active”

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
           targetNamespace="http://www.w3.org/1999/XSL/Transform"
           elementFormDefault="qualified">


  <xs:element name="import-schema" substitutionGroup="xsl:declaration">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="xsl:element-only-versioned-element-type">
          <xs:sequence>
            <xs:element ref="xs:schema" minOccurs="0" maxOccurs="1"/>
          </xs:sequence>
          <xs:attribute name="namespace" type="xs:anyURI"/>
          <xs:attribute name="schema-location" type="xs:anyURI"/>
        </xs:extension>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>

  ...
</xs:schema>
```

If you delete this

then this breaks

but this is still ok
Attributes & Namespaces

• **Why** do we have attributes?
  – Attributes aren’t **ordered**
  – Attributes don’t **repeat**
  – Attributes don’t **contain markup**
    • They can’t contain **structured data**
  – Require a **special** node type, axes, syntax, etc.
  – **Prefixless** attribute name weirdness:

A default namespace declaration applies to all **unprefixed element** names within its scope.
Default namespace declarations do **not** apply directly to **attribute** names; the interpretation of unprefixed attributes is determined by the **element** on which they appear.
“Local” vs. “Global” Attributes

Another namespace pattern

- Attributes in the null namespace
  - Null namespace attributes are contextually processed
    - Thus “local”

A default namespace declaration applies to all **unprefixed element** names within its scope. Default namespace declarations do **not** apply directly to **attribute** names; the interpretation of unprefixed attributes is determined by the **element** on which they appear.

```xml
<a xmlns:ex1="http://ex.org/1"
   xmlns:ex2="http://ex.org/2">
   <ex1:b name="...">
   <ex2:b name="...">
   <ex1:c ex1:name="..." ex2:name="...">
   <ex1:c ex1:name="..." ex1:name="...">
</a>
```

- Same name, but (perhaps) processed differently
- Different names, no connection
- Same names and illegal
Global Attributes Example

• Language extensions
  – xml:lang
  – xml:base
  – xml:space
  – xml:id

```xml
<xs:attributeGroup name="specialAttrs">
  <xs:attribute ref="xml:base"/>
  <xs:attribute ref="xml:lang"/>
  <xs:attribute ref="xml:space"/>
  <xs:attribute ref="xml:id"/>
</xs:attributeGroup>
```
Consider queries

```xml
<a type="a" xmlns:ex1="bla" xmlns:ex2="bla2" xmlns="bla3">
  <ex1:b name="1"/>
  <ex2:b name="2"/>
  <ex1:c ex1:name="3" ex2:name="4"/>
  <ex1:c ex1:name="5"/>
  <b name="6"/>
</a>
```

- `//@*` (all 6 attribute nodes)
- `//@name` (only 3 unprefixed attribute nodes)
- `//@ex1:name` (3,5)
- `//@*[namespace-uri()="bla2"]` (4)
- `//@*[namespace-uri()=""]` (1,2,6)
What to do with new version of format?

• Make it live in **new namespace**!
  – For what sorts of change?
    • Any change?
    • Extensions?
    • Revisions?
    • Just the “meaning”?
    • “Sufficient” change?

• Changing the namespace breaks stuff
  – So, perhaps do this when a change should break things?
  – http://www.w3.org/2001/tag/doc/namespaceState.html
  – http://www.w3.org/TR/xmlschema-guide2versioning/
Extension within a Namespace?

- **Alternative Schema!**
  - Just make a separate, unrelated document

- **Use xs:include**
  - Like xs:import but for “same namespace”
  - Use xs:redefine to redefine **components**
    - But not elements!
      - (Anonymous/unnamed types bite you)
    - Can only refine not completely redefine

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://MyCompositeSchema">
  <xs:include schemaLocation="http://www.cs.man.ac.uk/~sattler/myFirstSchema.xsd"/>
  <xs:include schemaLocation="http://www.cs.man.ac.uk/~sattler/myOtherSchema.xsd"/>

  .....  
</xs:schema>
```
Wildcards

- **xs:any**
  - Allows *any element* (etc) from *any namespace!*
  - With or *without* a definition
    - That is, can allow for any *well formed XML*
    - Sometimes known as an *open content model*

- **Consider comment**
  - What if we want structured comments?
    - With *any XML whatsoever!*

```xml
<xs:element name="comment">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

```xml
<el:comment xmlns:el="http://owl.cs.manchester.ac.uk/2010/comp/ssd-60372/day2/el">
  <h>What's this element? </h> Oo, mixed content! <a xmlns="http://ex.org" b="?"/>
  <el:foo>Junk! </el:foo>
</el:comment>
```
Tighter Wildcards

- We can control
  - **Which** namespaces
    - Name any specific number of namespaces
    - Explicitly forbid a namespace (e.g., not http://ex.org/)
    - Allow all, only non-targetNS, the targetNS, etc.
  - **Degree of validation**
    - strict: must be valid against a declaration
    - skip: anything well-formed!
    - lax: validate what you can figure out to validate, ignore the rest

```xml
<xs:any namespace="http://MyTrusted" minOccurs="0" maxOccurs="unbounded" processContents="lax" />
```
Rules of Thumb:

• For **multiple** WXS documents over **one** NS
  – Use `xs:include`
  – Can mix content models on existing elements!
  – Modularize development
    • With a bit of version hacking

• For making **one** schema over **multiple** NSs
  – Use `xs:import`
  – “Required” for multi-NS formats
    • since there is only 1 targetNameSpace per WXS
    • encourages NS centered development modularization

• For dealing with NSs not in your control
  – Use wildcards

• For relaxing parts of a document toward well-formed
  – Use wildcards
Empirical Interlude
Schemas?

- In SQL, schema before all
  - CREATE TABLE or nothing happens
  - Can’t INSERT INTO
  - Can’t SELECT FROM
  - So every SQL database has a schema
    - And the data conform

- XML, never *need* a schema
  - Except the minimal schema of well-formed-ness
    - Which is more mere minimal syntax
  - So why?
    - To *communicate*
    - To *error check*
    - To *guide tools*

- Given these advantages
  - How often used?
It was a bit disappointing to notice that a relatively large fraction of the XSDs we retrieved did not pass a conformance test by SQC. As mentioned in Section 2, only 30 out of a total of 93 XSDs were found to be adhering to the current specifications of the W3C [17].

Often, lack of conformance can be attributed to growing pains of an emerging technology: the SQC validates according to the 2001 specification and 19 out of the 93 XSDs have been designed according to a previous specification. Some simple types have been omitted or added from one version from one version of the specs to another causing the SQC to report errors.
Today’s XML

Figure 1: Summary of the Quality of the XML Web.
Today’s XML

• Weird facts:
  – 18% are not well formed
    • 66.4% of non-well formed documents have a DOCTYPE!
      – WHY!?

• “Validity is rare on the web. Just over 10% of the well-formed documents are also valid.”
  – Is there a difference between DTDs and WXS?
Invalid with DOCTYPE

docs that claim to be valid against X (X is ok) but aren’t

Figure 2: Distribution of causes for non-validation: DTD.
Invalid with “schemaLocation”

docs that claim to be valid against X (X is ok) but aren’t

Figure 3: Distribution of causes for non-validation: XSD.
XQuery
XQuery

- is a language for querying XML data
- it is built on/heavily uses/extends XPath expressions
  - smooth syntactic extensions: every XPath is an XQuery
- a W3C standard since 2007, see http://www.w3.org/TR/xquery/
- is supported by major database engines (IBM, Oracle, Microsoft, etc.)
- it can be used to
  - extract information to use in a Web Service
  - generate summary reports
  - transform XML data to HTML
  - search Web documents for relevant information
  - ...and to answer queries
XQuery: some basics

• XQuery provides support for datatypes, i.e., we
  – have variables and can
  – declare their type, yet the **query processor may be strict**: no attempt at a conversion to the correct type *needs* to be made!
  – e.g., if I try to add an integer with a decimal or write an integer into a decimal variable, the query processor *may* stop with an error

• like XPath, XQuery is based on **node sequences**
  – a sequence is a (poss. empty) list of **nodes**
  – as usual, nodes are of one of 7 kinds: element, attribute, text, namespace, processing-instruction, comment, or document (root)
  – if $mySeq is a sequence, $mySeq[3] is its third item

• all variable names start with “$” as in $mySeq
• comments are between “(:” and “:)” as in “(: this is a comment:)”
• a central, SQL-like part are **FLOWR expressions**
FLWOR expressions

- “FLWOR” is pronounced “flower”
- a FLWOR expression has 5 possibly overlapping parts:
  - For e.g., for $x$ in doc("people.xml")/contactList/person
  - Let e.g., let $i := 3$ let $n := $x/name/firstname
  - Where e.g., where $x/@categ = “friend”$
  - Order by e.g., order by $x/name/lastname ascending
  - Return e.g., return concat($x/name/lastname, “, “$x/name/firstname)

F and L can appear any (!) number of times in any order.
W and O are optional, but must appear in the order given.
R has always to be there...depending on who you ask...
FLWOR expressions

- a **for expression** determines what to iterate through
- is basically of the form

  ```
  for variable (as datatype)? (at position)? in expression
  ```

  - where **expression** is
    - any XPath location path or
    - a FLWOR expression (nesting!) or
    - a logic expression (if-then-else, etc.), later more
  - e.g., for $b in doc("people.xml")/contactList/person[@categ = "friend"]
    - query processor goes through the sequence of all (element) nodes
      selected by the XPath location path
  - e.g., for $b at $p in doc("contactlist.xml")/contactList/person
    where $p = 3
    return $b
    - query processor goes through (the singleton sequence containing)
      the third element node of the node set selected by the XPath location

```
FLWOR expressions

- a let expression binds a variable to a value
- is basically of the form

\[
\text{let variable (as datatype)? := expression}
\]

- where expression is
  - any XPath location path or
  - a FLOWER expression or
  - a logic expression (if-then-else, etc.), later more

- e.g.,

```xml
for $b in
doc("people.xml")/contactlist/person
let $name as element() := $b/name/firstname
return $name
```

```xml
for $b in
doc("people.xml")/contactlist/person
let $name as text() :=
  if (xs:integer($b/@age) < xs:integer(16))
    then ($b/name/firstname/text())
  else ($b/name/lastname/text())
return $name
```
FLWOR expressions

- we can repeat and mix for and let expressions
- a FLOWR expression
  - has at least one for or one let expression,
  - but can have any number of them in any order
- careful: the order plays a crucial role for their meaning
- make sure to bind variables to the right values before using them in for expression:

```groovy
let $doc := doc("people.xml")
for $p in $doc/contactlist/person
let $n := $p/name/lastname/text()
let $a := $p/@age
for $double in $doc/contactlist/person[@age = $a][name/lastname/text() = $n]
...
FLWOR expressions

- **return expression** determines output
- is basically of the form

\[
\text{return expression}
\]

- where *expression* is one of the logical expressions to be defined later
- it returns elements as *they are*, i.e., with attributes and descendants
- e.g.,

```xml
<MyFriendList>
  for $b$ in doc("people.xml")/contactlist/person[@categ="friend"]
  return $b/name/firstname/text()
</MyFriendList>
```

returns `<MyFriendList>John Millie…</MyFriendList>`

- careful: we needed "{"","}" to distinguish between text and instructions
FLWOR expressions

• as mentioned before, we can make use of logical expressions including
  – if-then-else
  – some/every
  – Boolean expressions
• e.g.,

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contactlist>
  <person categ="friend" age="25">
    <name>
      <lastname>Doe</lastname>
      <firstname>John</firstname>
    </name>
    <phone>0044 161 1234 5667</phone>
    <address>123 Main Street</address>
    <city>Manchester</city>
  </person>
...
```

```javascript
let $doc := doc("people.xml")
return
<MyFriendList>
{
  for $b in $doc/contactlist/person[@categ="friend"]
    return
      <friend>
        { (if (xs:integer($b/@age) < xs:integer(16))
            then $b/name/firstname/text()
            else $b/name/lastname/text()) }
      </friend>
  }
</MyFriendList>
```
XQuery: constructors

- as we have seen, we can use text in the return part
- to return a more complex XML document, we can make use of constructors
  - e.g., direct element constructors as in the previous example
  - or direct element constructors with attributes
- we use "{" and "}" to delimit expressions that are evaluated, e.g.,

```
let $doc := doc("contactlist-john-doe.xml")
for $p in $doc/contactlist/person
return
  <example>
  <p> Here is a query. </p>
  <$eg> $p/name </eg>
  <p> Here is the result of the query. </p>
  <$eg> { $p/name } </eg>
  </example>
```

- if we want to construct elements with attributes, we can do this easily: e.g.,
  return <friend phone ="{ xs:string($p/phone) }">{ (if (...
FLOWR expressions

- **where** is used to filter the node sets selected through let and for
- like in SQL, we can use **where** for **joins** of several trees or documents
- e.g.,

```xml
for $p$ in doc("contactlist-john-doe.xml")/contactlist/person
  for $c$ in doc("cities.xml")/citylist/city
    where $p/city/text() = $c/name/text()
  return concat("Dear ", $p/name/firstname, ", do you like ", $c/club,"? ")
```

**people.xml**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contactlist>
  <person categ="friend" age="25">
    <name>
      <lastname>Doe</lastname>
      <firstname>John</firstname>
    </name>
    <phone>0044 161 1234 5667</phone>
    <address>123 Main Street</address>
  </person>
  <person categ="family">
    <name>
      <lastname>Doe</lastname>
      <firstname>John</firstname>
    </name>
    <phone>0044 161 1234 5667</phone>
    <address>123 Main Street</address>
  </person>
</contactlist>
```

**cities.xml**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<citylist>
  <city>
    <name>Manchester</name>
    <club>Manchester United</club>
  </city>
  <city>
    <name>Munich</name>
    <club>Die Loewen</club>
  </city>
</citylist>
```
FLOWR expressions

- a more realistic, SQL-like example
  (from <oXygen/>):

```xml
<products>
 <product>
   <productId>1</productId>
   <productName>Wave Runner</productName>
   <productSpec>120 HP blaa</productSpec>
 </product>
...

<sales>
 { 
   for $product in doc("products.xml")/products/product,
   $sale in doc("sales.xml")/sales/sale
   where $product/productId = $sale/@productId
   return <product id="{$product/productId}">
     { $product/productName, $product/productSpec,
       $sale/mrq, $sale/ytd, $sale/margin }
   </product>
 } 
</sales>
```
FLOWR expressions

- like in SQL, we can nest expressions
- e.g., the previous example does not work in case a city has several clubs:

```
for $p in doc("contactlist-john-doe.xml")/contactlist/person
for $c in doc("cities.xml")/citylist/city
where $p/city/text() = $c/name/text()
return concat("Dear ", $p/name/firstname,
" , do you like ", $c/club[$i], ", ?")
```

```
<contactlist>
  <person categ="friend" age="25">
    <name>
      <lastname>Doe</lastname>
      <firstname>John</firstname>
    </name>
    <phone>0044 161 1234 5667</phone>
    <address>123 Main Street</address>
  </person>
</contactlist>

<people.xml>
<?xml version="1.0" encoding="UTF-8"?>
<contactlist>
  <person categ="friend" age="25">
    <name>
      <lastname>Doe</lastname>
      <firstname>John</firstname>
    </name>
    <phone>0044 161 1234 5667</phone>
    <address>123 Main Street</address>
  </person>
</contactlist>

<cities.xml>
<?xml version="1.0" encoding="UTF-8"?>
<citylist>
  <city>
    <name>Manchester</name>
    <club>Manchester United</club>
    <club>Manchester City</club>
  </city>
  <city>
    <name>Munich</name>
    <club>Die Loewen</club>
    <club>Bayern-Muenchen</club>
  </city>
</citylist>
```
XQuery FLOWR expressions

- **order by** allows us to order sequences before we return them
- we can combine several orderings into new ones \textit{lexicographically}
- e.g., for $nr$ in 1 to 5
  for $letter$ in ("a", "b", "c")
  order by $nr$ descending, $letter$ descending
  return concat($nr$, $letter$)
  
  yields 5c 5b 5a 4c 4b ....
- e.g., for $nr$ in 1 to 5
  for $letter$ in ("a", "b", "c")
  order by $letter$ descending, $nr$ descending
  return concat($nr$, $letter$)

  yields 5c 4c 3c 2c 1c 5b...
XQuery: grouping

- like SQL, XQuery provides **aggregation functions**
  - max and min
  - average
  - count, etc
- like in SQL, when we want to use them, we need to *group*:
- but this comes natural, e.g.,

```xml
for $an in fn:distinct-values(doc("orders.xml")/orderlist/order/artNr)
let $arts := doc("orders.xml")/orderlist/order[artNr = $an]
where fn:count($arts) >= 3
return
  <high-demand-item>
    <articleNr> { $an } </articleNr>
    <maxPrice> { fn:max($arts/price) } </maxPrice>
    <avgPrice> { fn:avg($arts/price) } </avgPrice>
  </high-demand-item>
```
Examples

contactlist.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<contactList>
  <person categ="friend" age="25">
    <name>
      <lastname>Doe</lastname>
      <firstname>John</firstname>
    </name>
    <phone>0044 161 1234 5661</phone>
    <address>123 Main Street</address>
    <city>Manchester</city>
  </person>
  <person categ="friend" age="14">
    <name>
      <lastname>Doen</lastname>
      <firstname>Jane</firstname>
    </name>
    <phone>0049 89 1234 5662</phone>
    <address>25 King Street</address>
    <city>Munich</city>
  </person>
  <person categ="foe" age="45">
    <name>
      <lastname>Do</lastname>
      <firstname>Jonathan</firstname>
    </name>
    <phone>0044 161 1234 5663</phone>
    <address>12 Queen Street</address>
    <city>Manchester</city>
  </person>
  <person categ="foe" age="13">
    <name>
      <lastname>Dove</lastname>
      <firstname>Jamie</firstname>
    </name>
    <phone>0049 89 1234 5664</phone>
    <address>23 Main Street</address>
    <city>Munich</city>
  </person>
</contactList>
```
Example queries

- Q1: for $b$ in doc("contactlist.xml")/contactList/person[@categ = "friend"][position() = 1]
  return $b

- Q2: for $b$ at $p$ in doc("contactlist.xml")/contactList/person[@categ = "foe"]
  where $p$ = 2
  return $b

- Q3: for $b$ at $p$ in doc("contactlist.xml")/contactList/person[@categ = "foe"]
  where $p$ = 3
  return $b

- Q4: for $p$ in doc("contactlist.xml")/contactList/person[@age > 16]
  return $p/name
Example queries (cont.)

- Q5: for $p$ in doc("contactlist.xml")/contactList/person
  return $p/phone

- Q6: let $doc := doc("contactlist.xml")
  for $p$ in $doc/contactList/person
  let $a := xs:integer($p/@age)
  let $c := xs:string($p/@categ)
  where $a < xs:integer(16)
  and $c = "foe"
  return $p

- Q7: for $c$ in fn:distinct-values(doc("contactlist.xml")/contactList/person/city)
  let $p := doc("contactlist.xml")/contactList/person[city = $c]
  order by fn:avg($p/@age)
  return 
  <city name = "{$c}">
    <avg_age>{fn:avg($p/@age)}</avg_age>
  </city>
XQuery: functions

- XQuery is more than FLWOR expression
- it provides more than 100 built-in functions, we have already seen some, plus
  - e.g., `<name>{uppercase($p/lastname)}</name>`
  - e.g., let $nickname := (substring($p/firstname,1,4))
- it allows the user to define functions

```
declare function prefix:function_name((parameter as datatype)*)
  as returnDatatype
{
  (; ...your function code here... :)
};
```

- e.g.,
  
  declare function local:minPrice(
    $price as xs:decimal,
    $discount as xs:decimal
  )
  as xs:decimal {
    let $disc := ($price * $discount) div 100
    return ($price - $disc)
  }

  to be used e.g., in

  `<minPrice>
    { local:minPrice($book/price, $book/discount) }
  </minPrice>`
XQuery: functions

- XQuery is more than FLWOR expression
- it provides more than 100 built-in functions, we have already seen some, plus
  - e.g., `<name>{uppercase($p/lastname)}</name>`
  - e.g., let $nickname := (substring($p/firstname,1,4))
- it allows the user to define functions

To summarize the departments from Manchester, use:

```xml
local:summary(doc("acme_corp.xml")//employee[location = "Manchester"])
```

declare function local:summary($emps as element(employee)*) as element(dept) *
{
    for $d in fn:distinct-values($emps/deptno)
    let $e := $emps[deptno = $d]
    return
        <dept>
            <deptno>{$d}</deptno>
            <headcount>{fn:count($e)}</headcount>
            <payroll>{fn:sum($e/salary)}</payroll>
        </dept>
};
```
XQuery Functions: Closure

- XQuery is compositional
  - a query returns a **node sequence**
  - a functions return **node sequence**
    - A single node is a singleton node sequence and vice versa
  - So we can write queries with functions at key steps
    - Not just in predicate tests!

```xml
<this>
  <xmlFragment/>
  <is>actually a bunch of xquery</is>
  <constructor/>
  <which>
    <returns>a sequence of nodes</returns>
  </which>
</this>//returns

<returns>a sequence of nodes</returns>
```
XQuery Functions: Closure

- XQuery is compositional
  - a query returns a **node sequence**
  - a functions return **node sequence**
    - A single node is a singleton node sequence and vice versa
  - So we can write queries with functions at key steps
    - Not just in predicate tests!

```xml
<xmlFragment/>
<is>actually a bunch of xquery</is>
<constructor/>
<which>
  <returns>a sequence of nodes</returns>
</which>
</this/>
```

XQuery query!!

result sequence!

```xml
<returns>a sequence of nodes</returns>
```
XQuery Functions: Closure

**Declare Function** `local:header()` as `node()` {
  `<div class="web-page-header">
    <img src="images/mylogo.jpg" alt="Our Logo"/>
    <h1>Acme Widgets Inc.</h1>
  </div>`;
}

`local:header()`//`h1`

**Declare Function** `local:numbers()` {
  `(1, 2, 3, 4, 5)`
};

`local:numbers()`[.>3]

**Declare Function** `local:gt3($nodes)` {
  $nodes[.>3]
};

`local:gt3(local:numbers())`
XQuery, schemas, and types

- if you query documents that are associated with a schema, you can exploit schema-aware query answering:
  - WXS has **default values**, e.g., answer to this query may vary depending on your schema!

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:element name="person"/>
  <xs:attributeGroup name="attlist.person">
    <xs:attribute name="id" use="required" type="xs:ID"/>
    <xs:attribute name="isFriend" default="true">
      <xs:simpleType>
        <xs:restriction base="xs:token">
          <xs:enumeration value="true"/>
          <xs:enumeration value="false"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:attributeGroup>
</xs:schema>
```

```xquery
for $m in doc('personal.xml')//*[@isFriend = 'true']
  return $m/name/family/text()
```
XQuery, schemas, and types

• if you query documents that are associated with a schema, you can exploit schema-aware query answering, e.g., XML Schema aware like SAXON-EE:
  – careful if you use <oXygen>: it sometimes confuses SAXON-HE/SAXON-EE
  – WXS has default values, e.g., answer to this query may vary depending on your schema

```xml
import schema namespace uli="www.uli.org" at "test4.xsd";
for $m in doc('Untitled7.xml')/uli:nEl
return data($m/@attr)
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<uli:nlist xmlns:uli="www.uli.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="www.uli.org file:test4.xsd">
  <uli:nEl>3</uli:nEl>
  <uli:nEl attr="4">4</uli:nEl>
  <uli:nEl>5</uli:nEl>
</uli:nlist>
```
XQuery, schemas, and types

- If you query documents that are associated with a schema, you can exploit schema-aware query answering, e.g., XML Schema aware like SAXON-SA:
  - WXS has types, e.g., answer to this query may vary depending on your schema.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<ul:list xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="www.uli.org test4.xsd"
    xmlns:uli="www.uli.org">
  <uli:friend>Paul</uli:friend>
  <uli:friend>Peter</uli:friend>
  <uli:friend>Mary</uli:friend>
  <uli:friend>Joanne</uli:friend>
  <uli:friend)Lucy</uli:friend>
</ul:list>
```

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="www.uli.org"
    xmlns:uliS="www.uli.org"
    elementFormDefault="qualified">
  <xs:element name="list" type="uliS:B">
    <xs:complexType name="A">
      <xs:sequence>
        <xs:element name="friend" type="xs:string"
            minOccurs="4" maxOccurs="5"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="B">
    <xs:complexContent>
      <xs:restriction base="uliS:A">
        <xs:sequence>
          <xs:element name="friend" type="xs:string"
              minOccurs="4" maxOccurs="5"/>
        </xs:sequence>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```
XQuery, schemas, and types: the PSVI

Post-schema-validation infoset:
Internal Rep. adorned with schema information e.g., a tree adorned with default values & types
Quick Note on PSVI

- **Post Schema-Validation Infoset**
  - First approximation: DOM + Schema Information
    - What kind of information?
      - Default attribute (and other) values
      - Type information
    - Remember node types in the DOM
      - Atomic values are all text (string)
      - But WXS lets us have loads of atomic types!
        - As well as simple and complex types!
      - XQuery (and XPath >=2.0) can be sensitive to those types
      - Thus, that type information has to get into the queried data

- **PSVIs are known to be valid!**
  - Thus we can make some assumptions about their structure
SQL intuition on PSVI

(SCHEMA) CREATE TABLE

CSV

Program

Populated Database
SQL intuition on PSVI

(Schema) CREATE TABLE

CSV

Program

Populated Database

Schema

Validation

PSVI

Well Formed XML
Namespace, schemas, and queries

- schemas and queries can be used together in a powerful way
  - e.g., to retrieve values and default values
  - e.g., by exploiting type hierarchy in query: this can have various advantage:
    - we can safe big ‘unions’ of queries through querying for instances of super types
    - should we change our schema/want to work with documents with new kind of elements (see XML/OWL coursework), it may suffice to adapt the schema to new types; queries may remain unchanged!

- usage of namespace, schemas, and queries is a bit tricky:
  - when to use/declare which namespace/prefix where
  - tool support required

- more in coursework and later
Coursework: SEs

• As you (should) know by now, we use **rubrics** to mark
  – for SEs and others
  – for coursework and exam

• For you to understand these better, you are going to **apply**
  – SE2 rubric to
  – SE2 essay of a friend
  – …we printed rubrics for you:
    • find a friend (or 2)
    • take a rubric each
    • swap your SE2 essays (e.g. via email)
    • mark each other’s essays
    • discuss the outcome
    • submit a guess for your own mark for SE2 on BB as GuessSE2
  • …keep this in mind when you write
    • SE3, SE4, SE5
    • exam
    • your thesis
Coursework this week

- Get to know tools/oxygen better:
  - use it to test your understanding of XPath
  - collect a fine sample of XML docs, XSDs, RNGs, …

- Q3:
  - use tools to answer questions

- CW3:
  - XQuery for arithmetic learning site

- M3: XPath, XQueries, and XSD
  - do this before SE3

- SE3: robustness, schemas, and different query styles
  - think/read about robustness
  - do M3 before you do this