MANCHESTER 1824

The University of Manchester

COMP60411
Modelling Data on the Web

XML Schema, XQuery, and robustness

Week 4

Tim Morris
Uli Sattler
University of Manchester

Week 2 coursework

- All graded!
  - Q2, SE2, M2
  - CW1 and CW2 not yet
- In general,
  - Pay attention to the feedback
    - check the **rubrics**
    - try to **regenerate**
    - try rubric on your friend's essays
  - If you don't understand
    - read: slides, articles (see materials' page), other
    - think/draw
    - check & ask on the forum and/or TAs
    - we're happy to explain further!
  - Remember, you'll get essays (and MCQs) on the exam
    - **Practice** and learn now!
    - It will help!

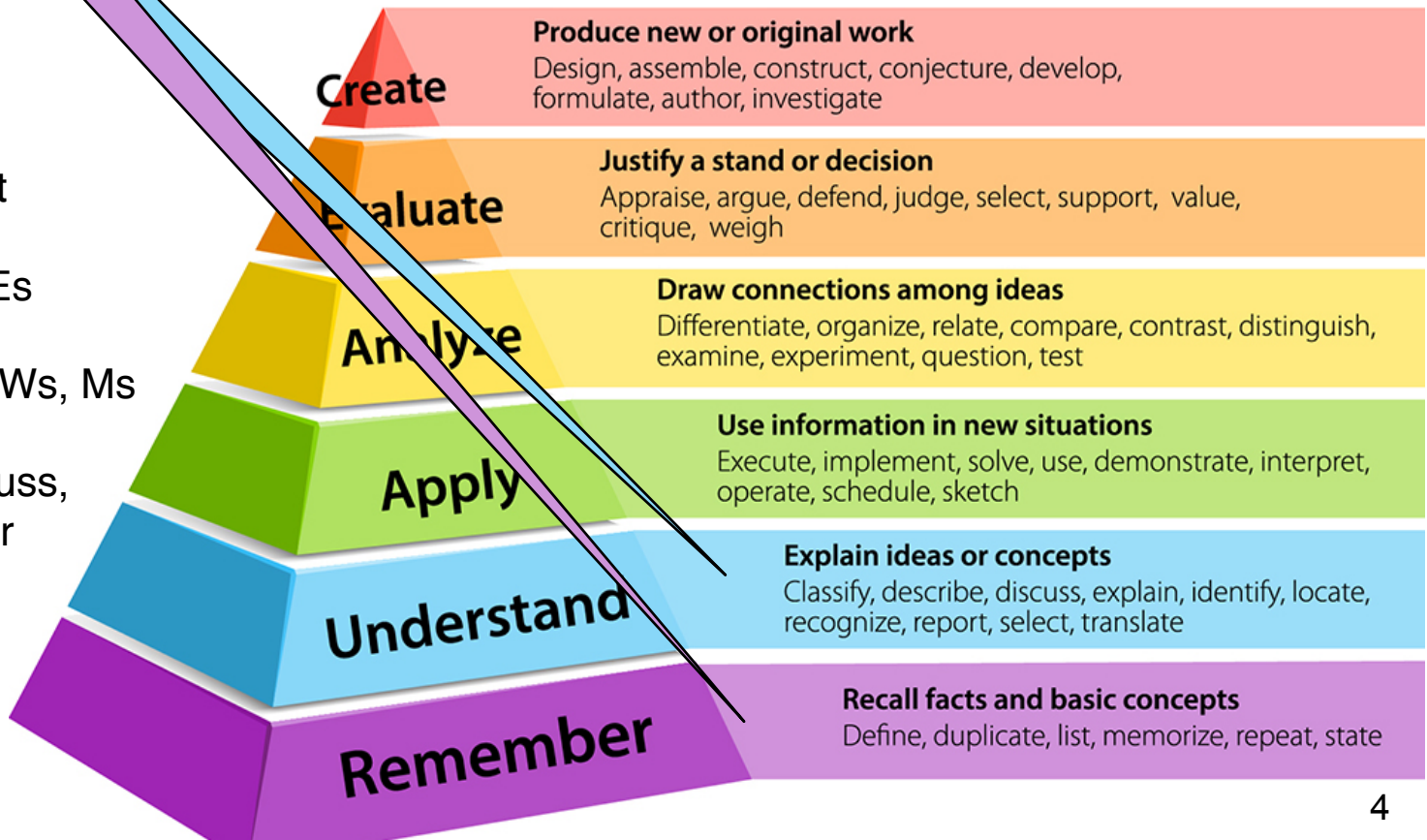
(Technical) Terms & Meaning

- In CS (as a (technical) subject area), people
  - make up & use new **terms**
  - to capture relevant **concepts**
- For people to be able to communicate, we need to
  - agree on the **meaning** of (new) terms…how?
  - ➡ We **define** their meaning and agree to use that one, e.g., for
    - self-describing
    - format
    - (core) data model
    - external/internal representation
    - …

How to Deepen your Understanding

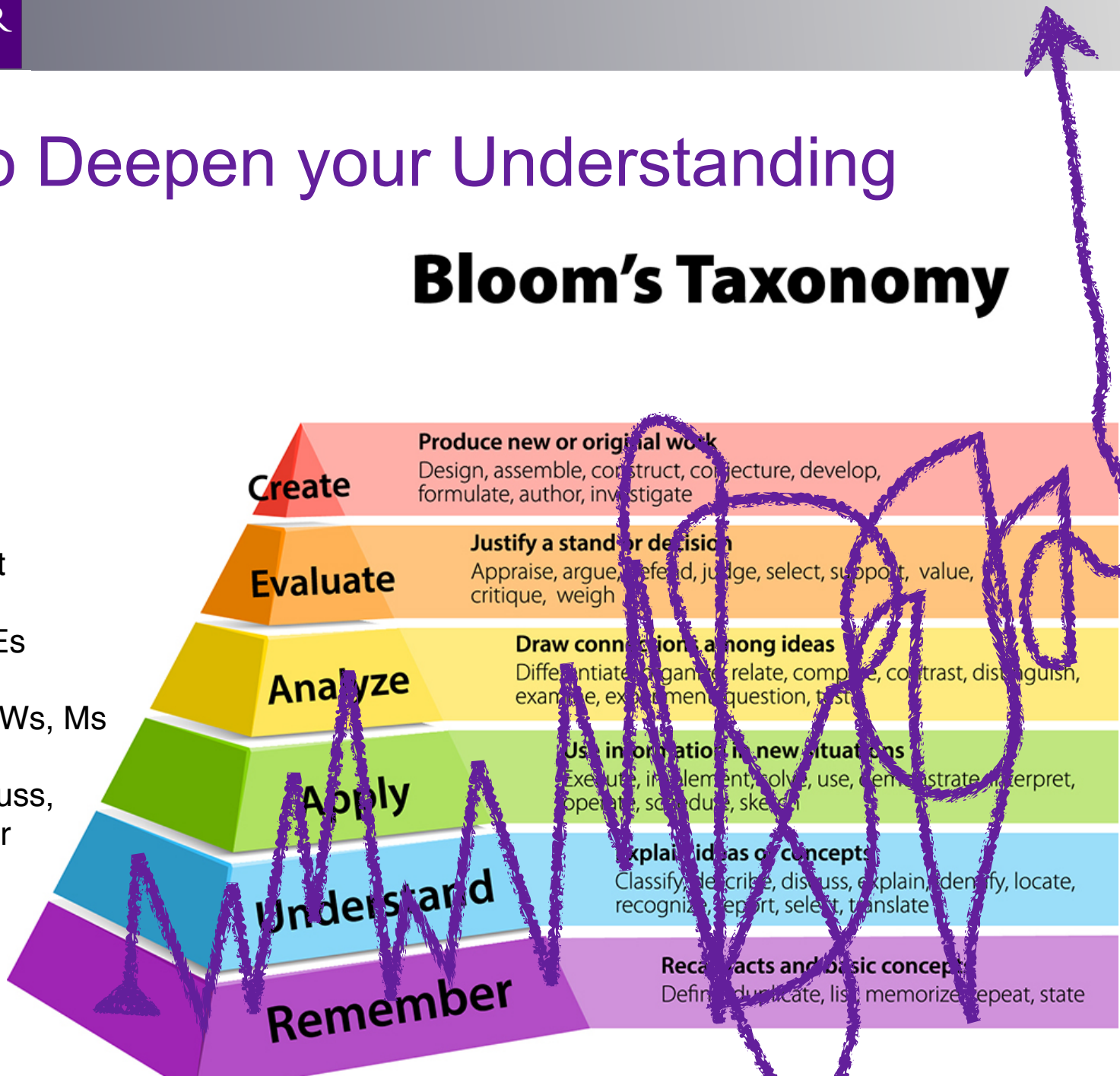
Concepts & terms

- …in your project
- Compare - in SEs
- Apply - use in CWs, Ms
- Describe & discuss, make & consider examples
- Read & repeat



How to Deepen your Understanding

- …in your project
- Compare - in SEs
- Apply - use in CWs, Ms
- Describe & discuss, make & consider examples
- Read & repeat



Let's put this in practise

- With your neighbour
  - consider the following terms
    - format
    - formalism
    - core data model
    - data model
    - database
    - external repr.
    - system
    - internal repr.

Keep thinking about & working on these!

- Then
  1. draw a Conceptual Model (ER Diagram) relating/about these
     - you can add more
  2. give some examples

- We will discuss your findings in 15mins

Plagiarism and Definitions

- You can/are allowed to/should copy **definitions**
  - while referencing its source
  - no need to make up your own
  - we need to agree on definitions to communicate!
  - …but you often don't need to (I will remember those we used in class)

- Other kind of statements are different:
  - examples
  - descriptions
  - discussion
  - proofs
  - opinions
  - … give your own to show your understanding

This Week

Tree data models:

1. Data Structure formalisms: XML (including name spaces)
2. Schema Language: XML Schema (XSD)
3. Data Manipulation: XQuery

New general concepts:

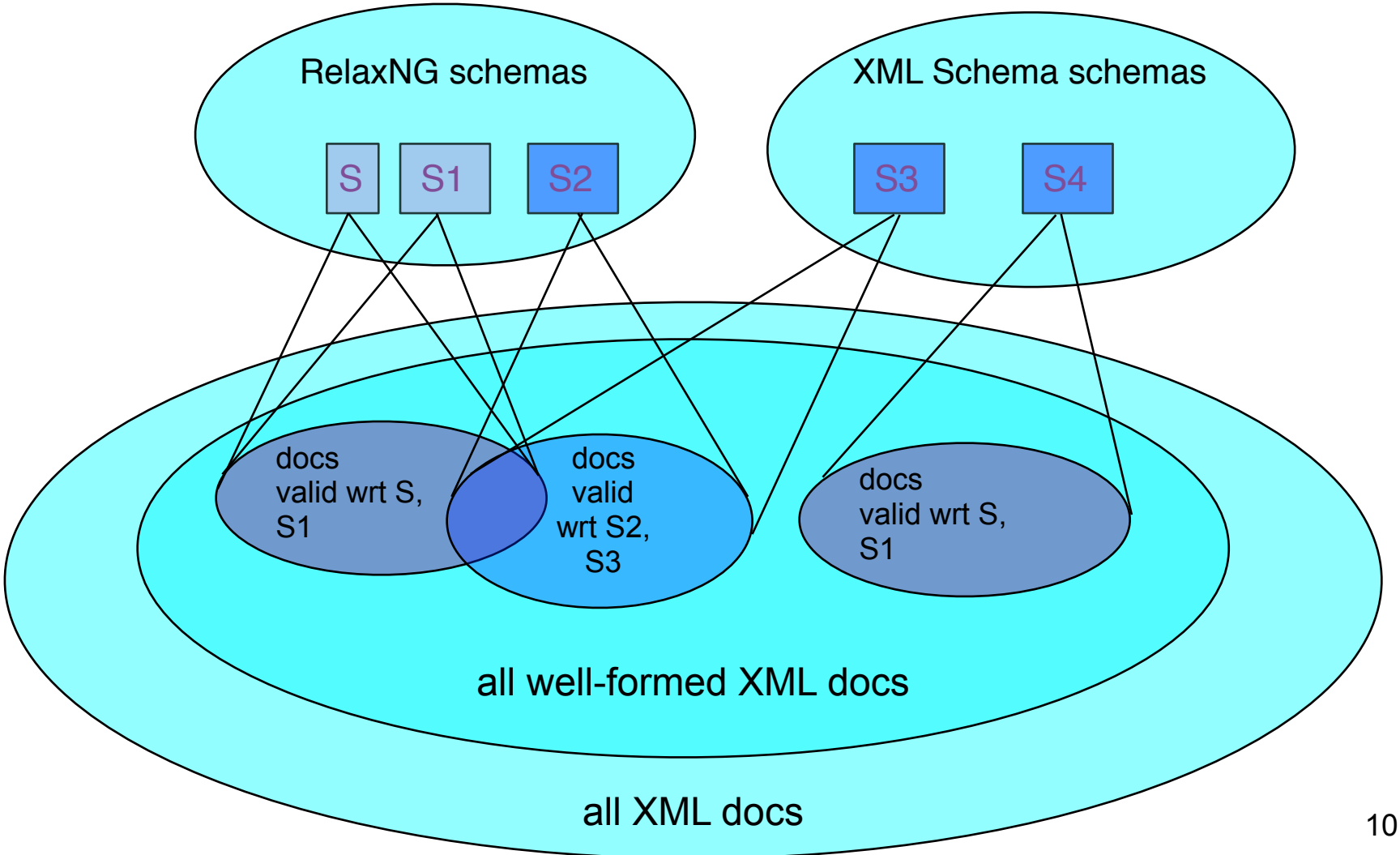
- Error handling

NewPlus deepening of our understanding of other concepts

- Format
- Self-describing

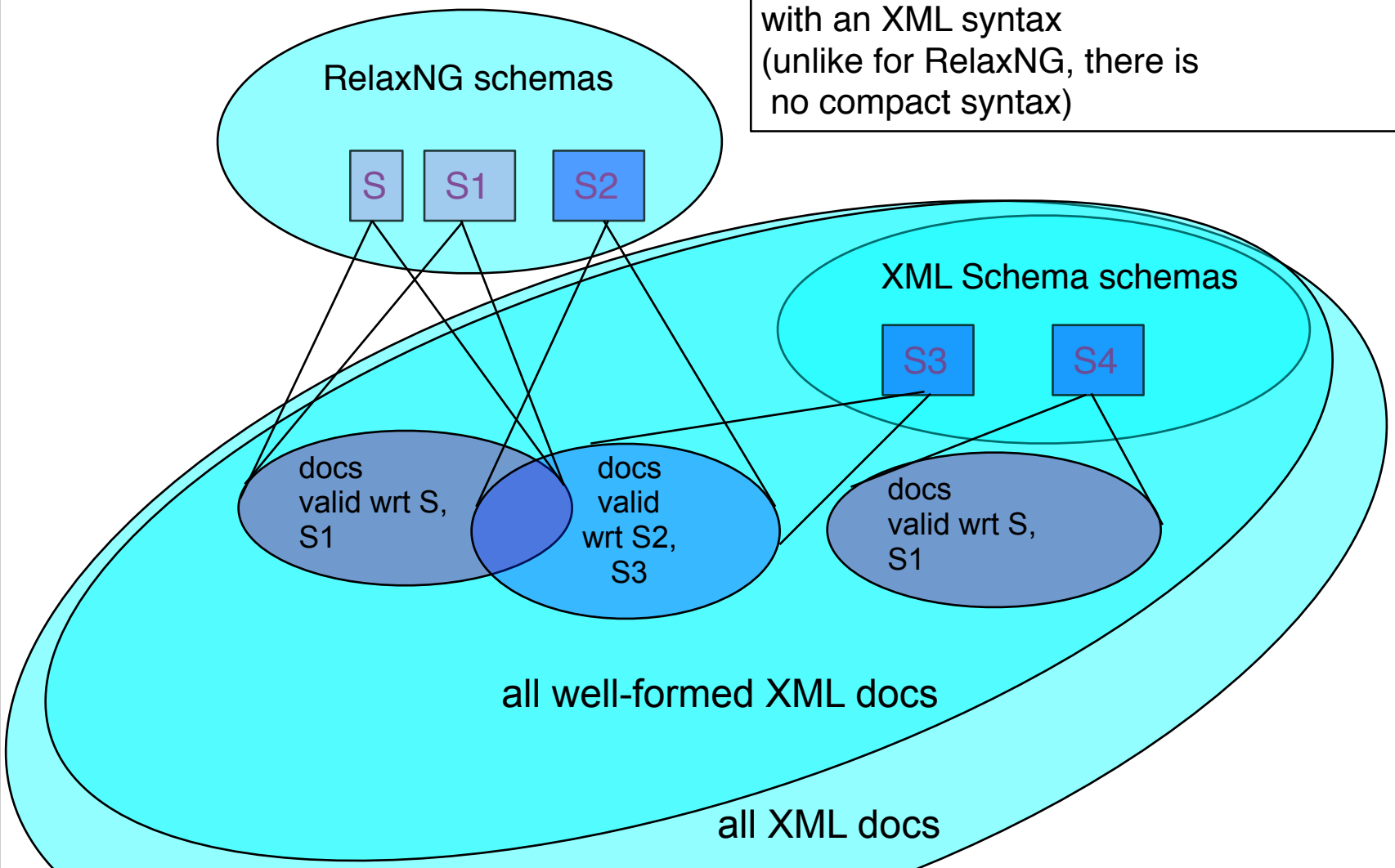
XML Schema
another schema language for XML

There is more than 1 schema language



A more correct picture:

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



Schema languages for XML

provide means to define the legal structure of an XML document

```
grammar {
   start = cartoon
   cartoon = element cartoon { attlist.cartoon, prolog, panels }
      attlist.cartoon &= attribute copyright { text }
      attlist.cartoon &= attribute year { text }
   prolog = element prolog { attlist.prolog, series, author, characters }
      attlist.prolog &= empty
   series = element series { attlist.series, text }
      attlist.series &= empty
...
```

cartoon.rnc, a **RelaxNG Schema** for cartoon descriptions

See Section 9.2 of https://relaxng.org/compact-tutorial-20030326.html for meaning of '&=' in definitions

...

```
<?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>
 ...
```

```
<?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>Snowman</character>
   </characters>
 ...
```

...

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

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
  <xs:element name="cartoon">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="prolog"/>
        <xs:element ref="panels"/>
      </xs:sequence>
      <xs:attributeGroup ref="attlist.cartoon"/>
...
```

cartoon.xsd,
an **XML Schema** 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>
...
```

```
<?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>Snowman</character>
  </characters>
...
```

...

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

- an RNG schema in compact syntax is **not** a well-formed XML document
  - though you can use the RNG XML format
- an XML Schema 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?)
  - more features for describing the (element & mixed) content of elements

XML Schema: a first example

Example with RNG:

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

<note>
   <to>Tove</to>
   <from>Jani</from>
   <sentOn>2007-01-29</sentOn>
   <body>
      Have a nice weekend!
   </body>
</note>
```

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>
   <to>Tove</to>
   <from>Jani</from>
   <sentOn>2007-01-29</sentOn>
   <body>
        Have a nice weekend!
    </body>
</note>
```

note.xsd:

```
<?xml version="1.0"?>
<xs:schema
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns="http://www.w3schools.com"
   elementFormDefault="unqualified">
   <xs:element name="note">
      <xs:complexType>
         <xs:sequence>
            <xs:element
                 name="to" type="xs:string"/>
            <xs:element
                 name="from" type="xs:string"/>
            <xs:element
                 name="sentOn" type="xs:date"/>
            <xs:element
                 name="body" type="xs:string"/>
         </xs:sequence>
      </xs:complexType></xs:element></xs:schema>
```

Datatypes!

XML Schema: the same with namespaces

```
<?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>
```

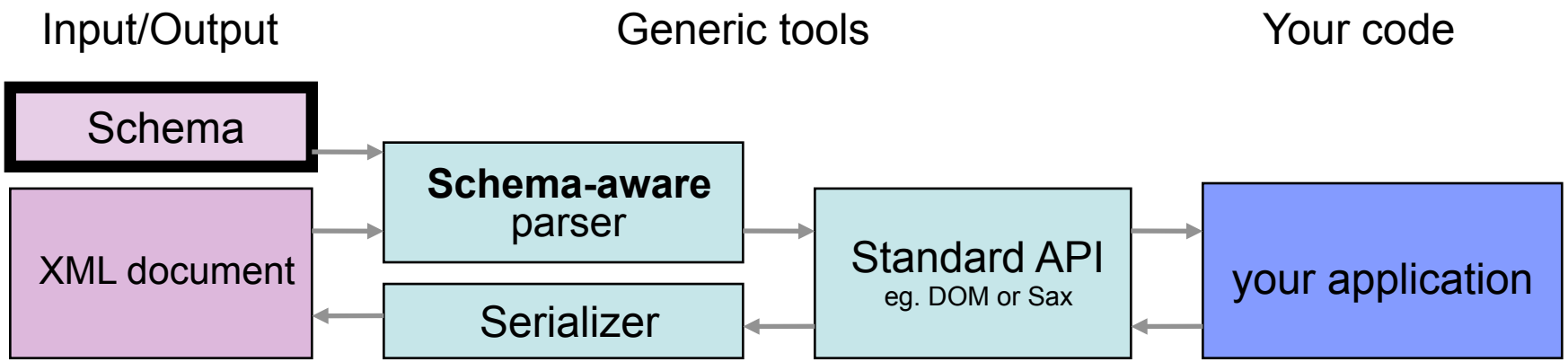
note.xsd:

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

Namspaces!

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



XML Schema: some remarks

- in an XSD 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

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

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

```
<?xml version="1.0"?>
<note xmlns="blaaaa"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

XML Schema namespace e.g. for datatypes like xs:integer

Target namespace of elements defined in this schema, e.g. for sentOn

Local (default) namespace

"This document uses an XSD schema"

XML Schema & Namespaces

- XSD supports (and uses) **namespaces**
- an XSD 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>
```

```
<?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>
```

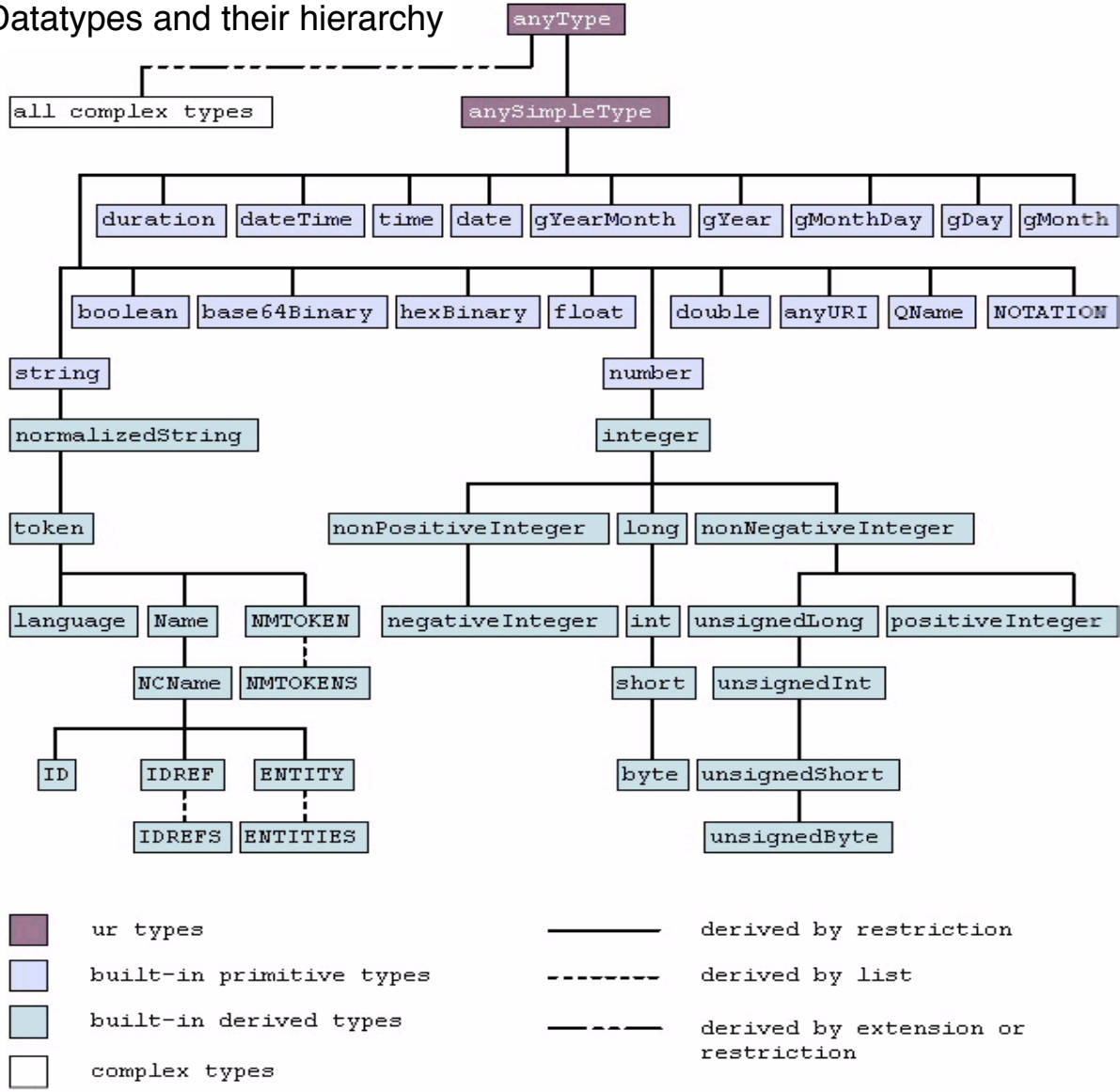
note.xsd:

```
<?xml version="1.0"?>
<xs:schema
   xmlns:xs=
     "http://www.w3.org/2001/XMLSchema"
   targetNamespace=
     "http://www.w3schools.com"
   xmlns="http://www.w3schools.com"
   elementFormDefault="qualified">
   <xs:element name="note">
      <xs:complexType>
         <xs:sequence>
            <xs:element
                 name="to" type="xs:string"/>
         ...
```

XML Schema core concepts: datatypes

- 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

Built-In Datatypes and their hierarchy



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
    e.g., true ↦ true and 1 ↦ true
  - for xs:integer 1 ↦ 1 and 01 ↦ 1 and 001 ↦ 1,…
- **constraining facets** that can be used in restrictions of that datatype
  - e.g., for xs:integer: maxInclusive, maxExclusive, minInclusive, …
  - e.g., for defining "SmallInteger" or "ShortString"

XML Schema: types

We can define **types** in an XSD schema, 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

- …then we can combine the latter with **xs:simpleContent**
  - for elements with text content

XML Schema: Type Declarations

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

```
<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>
```

```
<age>4</age>
```

```
<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>
```

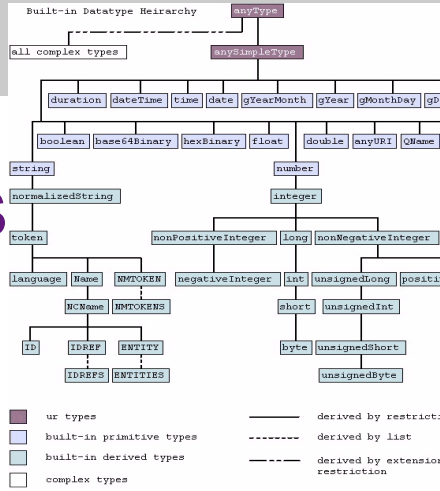
```
<person friend="1">
        <Name>...</Name>
        <DoB>...</DoB>
</person>
```

- can be **named**, e.g., Agetype or Persontype

```
<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

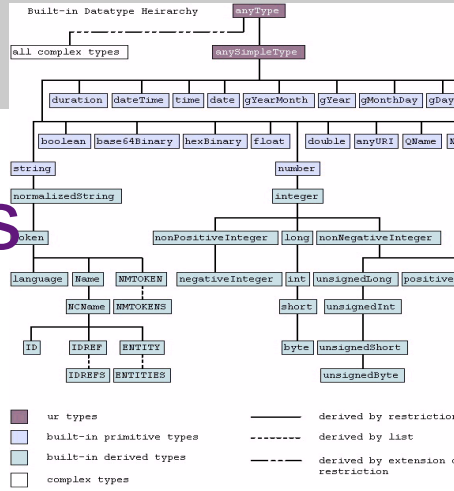


- are based on the numerous built-in datatypes
- that can be restricted using **xs:restriction facets**, e.g.,

enumeration	`<xs:simpleType name="bikeType">`
	`  <xs:restriction base="xs:string">`
	`  <xs:enumeration value="MTB"/>`
	`  <xs:enumeration value="Race"/>`
	`  <xs:enumeration value="Children"/>`
	`  <xs:enumeration value="Hybrid"/>`
	`  <xs:enumeration value="Folding"/>`
	`  </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.,

maxLength minLength	<xs:simpleType name="medStr"> <xs:restriction base="xs:**string**"> <xs:minLength value="5"/> <xs:maxLength value="8"/> </xs:restriction> </xs:simpleType>
maxExclusive/maxInclusive minExclusive/minInclusive	<xs:simpleType name="age"> <xs:restriction base="xs:**integer**"> <xs:minInclusive value="0"/> <xs:maxInclusive value="120"/> </xs:restriction> </xs:simpleType>
patterns using regular expressions	<xs:simpleType name="simpleStr"> <xs:restriction base="xs:**string**"> <xs:pattern value="([a-z][A-Z])+"/> </xs:restriction> </xs:simpleType>

XML Schema: Composite Simple Types

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

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

<xs:simpleType name='ShortList'>
   <xs:restriction base='IntList'>
      <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:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="red"/>
            <xs:enumeration value="green"/>
            <xs:enumeration value="blue"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:list>
</xs:simpleType>
```

XML Schema: Simple Types

- can be used for
  - element content of text elements
  - attribute values

```
<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

- xs:simpleType for
  - **attribute values** and
  - **elements** without element child nodes and without attributes

- 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.

```
<size country="UK">4</size>
```

```
<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>
```

simple content
but
complex
(not simple) type

XML Schema: Complex Types

- xs:complexType for
  - **elements** with
    - element content or
    - mixed element content or
    - text content and attributes

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

```
<xs:complexType name="nametype">
    <xs:sequence>
        <xs:element name="fname" type="xs:string"/>
        <xs:element name="mname" type="xs:string"/>
        <xs:element name="lname" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

- 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.

```
<xs:complexType name="nametype">
    <xs:sequence>
        <xs:element name="fname" type="xs:string"/>
        <xs:element name="mname" type="xs:string" minOccurs="0" maxOccurs="7"/>
        <xs:element name="lname" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

XML Schema: Complex Types

```
<Person>
  <Name>
    <fname>Lucy</fname>
    <mname>Mary</mname>
    <mname>Hilary</mname>
    <mname>Esmeralda</mname>
    <lname>Smith</lname>
  </Name>

  <name>
    <fname>Sally</fname>
    <lname>Miller</lname>
  </name>
```

- **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.

```
<xs:complexType name="nametype">
    <xs:sequence>
      <xs:element name="fname" type="xs:string"/>
      <xs:element name="mname" type="xs:string" minOccurs=“0” maxOccurs="7"/>
      <xs:element name="lname" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
```

XML Schema: mixed content

```
<Person>
   <Name phone="0161-8619897">
     Sally Miller
   </Name>
   <DoB>
      2001-12-24
   </DoB>
</Person>
```

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

```
<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.,

```
<xs:simpleType name="AgeType">
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="“125"/>
   </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

```
<xs:simpleType name="AgeType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"/>
    <xs:maxInclusive value="125"/>
  </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.,

```
<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>

<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

```
<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>
```

- this means that a XSD-aware parser has to **be aware of** a schema's **type hierarchy**
  - to check that LongPersonType was really derived by restriction or extension from the type expected for person
- but XSD-aware parser does NOT have to "guess" an element's type from its properties
- In SE4: 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.,

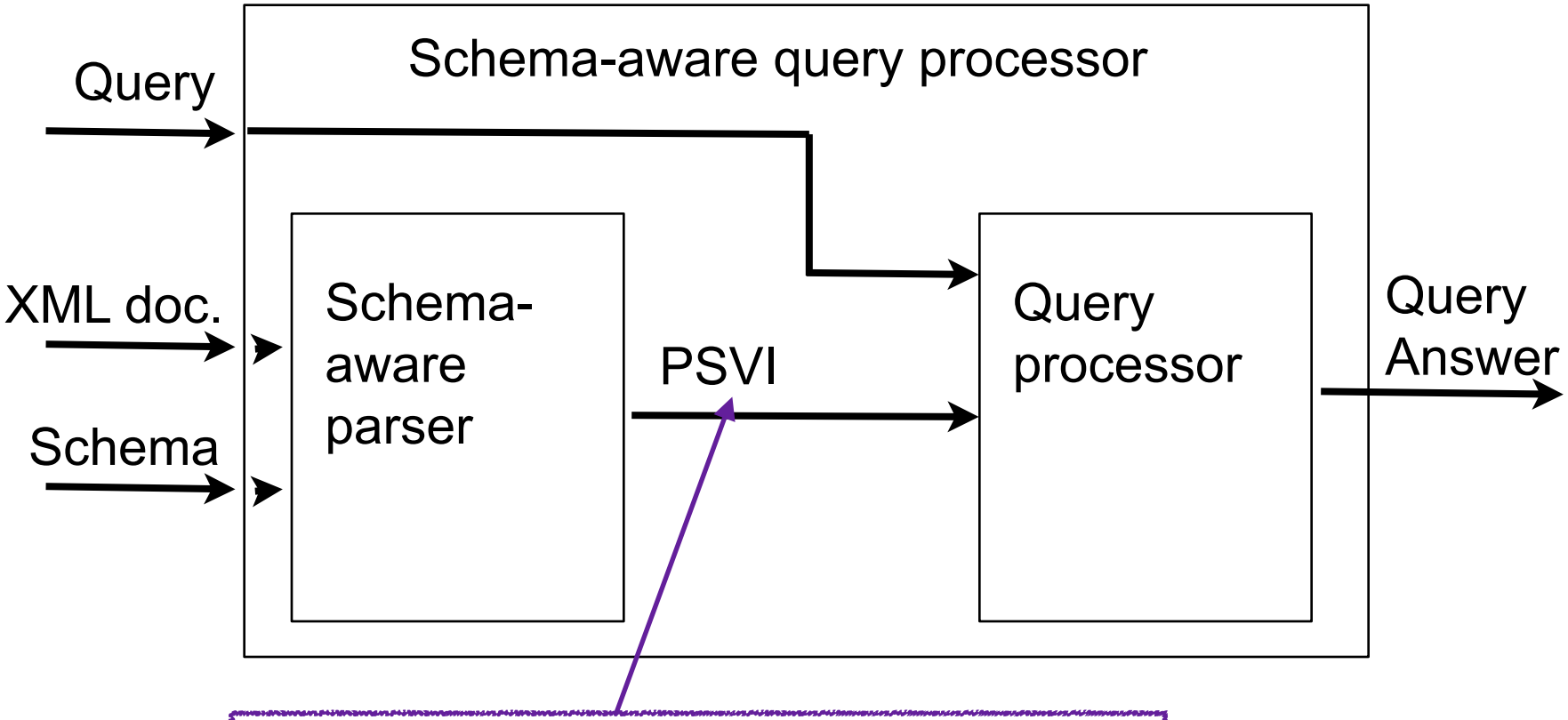
  <xs:complexType name="StrictPersonType" **final="#all"**>

- 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

Schemas and Types: the PSVI

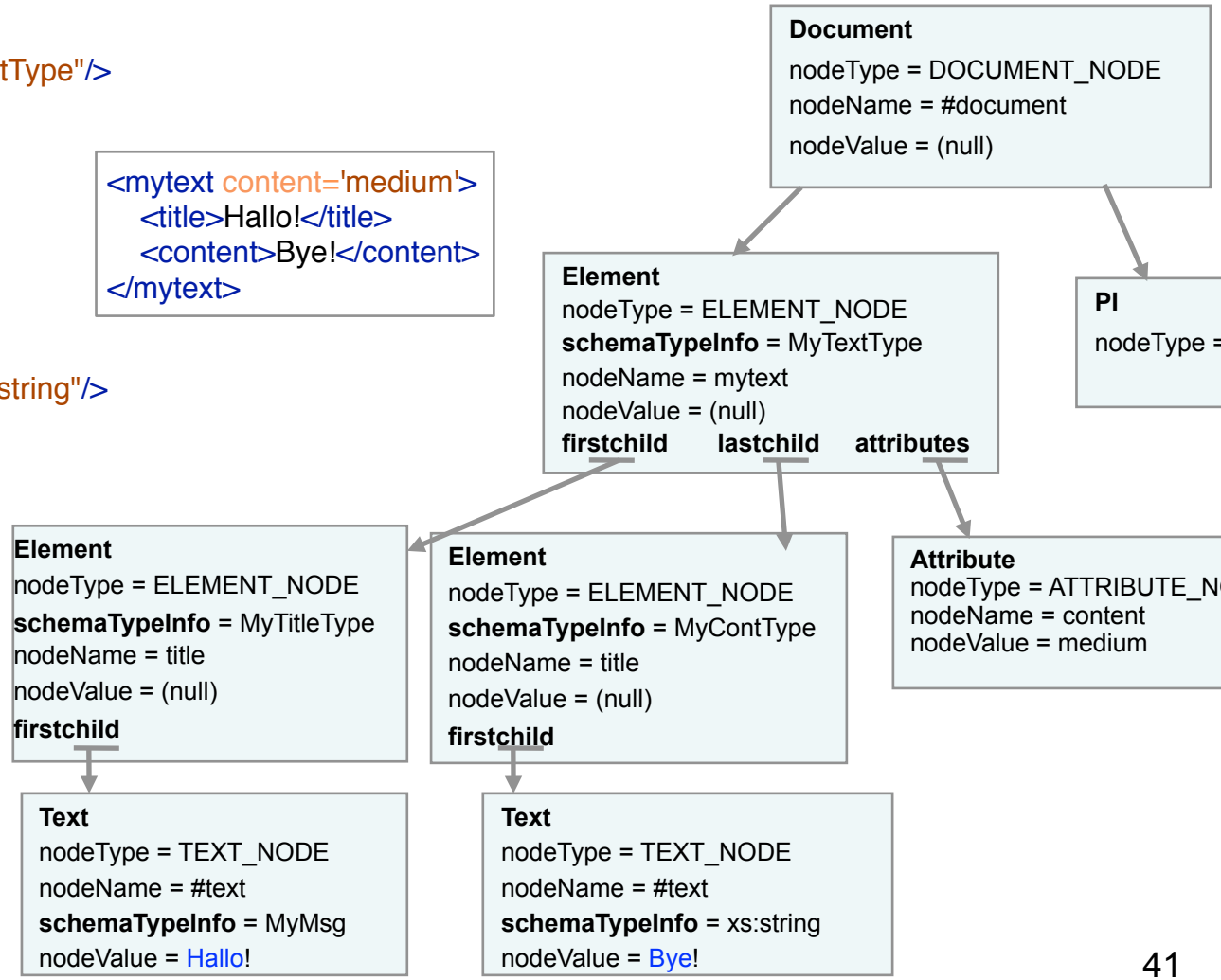


Post-**s**chema-**v**alidation **i**nfoset:
Internal Rep. adorned with schema information
e.g., a tree adorned with default values & types

PSVI = DOM for XML document + XSD schema

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

  <xs:element name="mytext" type="MyTextType"/>

  <xs:complexType name="MyTextType">
    <xs:sequence>
      <xs:element name="title"
        type="MyTitleType"/>
      <xs:element name="content"
        type="MyContType"/>
    </xs:sequence>
    <xs:attribute name="content" type="xs:string"/>
  </xs:complexType>

  <xs:simpleType name="MyTitleType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>
  <xs:simpleType name="MyContType">
    <xs:restriction base="xs:string"/>
  </xs:simpleType>

</xs:schema>
```

```
<mytext content='medium'>
  <title>Hallo!</title>
  <content>Bye!</content>
</mytext>
```



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 & leaves information in IR!
  - 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 child node sequence against the corresponding content model is "easier"
  - e.g., *XML Element Declarations Consistent* constraint
- There are **sets of XML documents** (e.g., your cartoon descriptions)
  - for which we can formulate a RNG
  - but not an XML schema -
  - and vice versa

Back to Self-Describing
&
Different styles of schemas

The Essence of XML

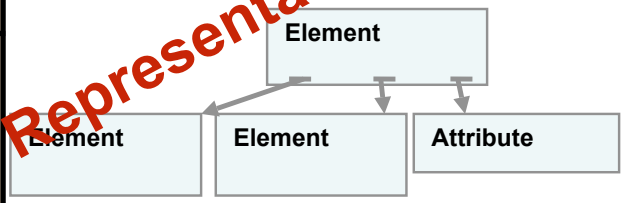
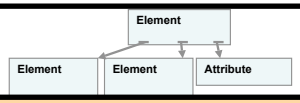
cool paper mentioned in Week 2

- Thesis:
  - "XML is touted as an external format for representing data."
- Two properties
  - Self-describing
    - Destroyed by external validation,
    - i.e., using application-specific schema for validation, one that isn't referenced in the document
  - Round-tripping
    - Destroyed by defaults and union types

http://bit.ly/essenceOfXML2

MANCHESTER 1824

The University of Manchester

Level	Data unit examples	Information or Property required
cognitive		
application		
tree adorned with...		
namespace / schema		nothing / a schema
tree		well-formedness
token: complex	<foo:Name t="8">Bob	
token: simple	<foo:Name t="8">Bob	
character	<foo:Name t="8">Bob	which encoding (e.g., UTF-8)
bit	10011010	

Internal Representation

External Representation

erase

serialise

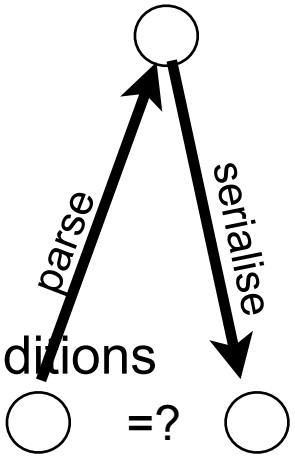
validate

parse

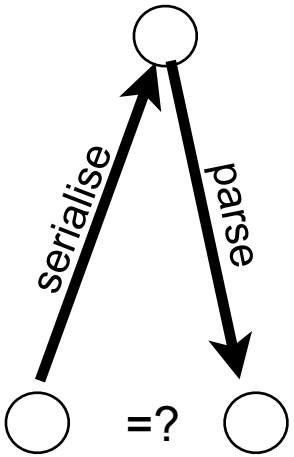
Roundtripping (1)

- is **successful** if "=" holds (to some extent)
- depends on serialisation & parsing mechanisms

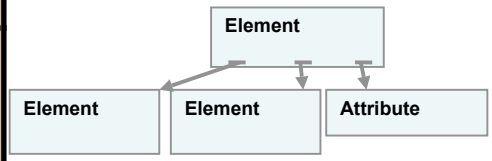
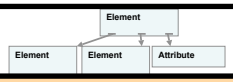
Roundtripping (2)



- Within a single system:
  - roundtripping (both ways) should be *exact*
  - same program should behave the same in similar conditions
- Within various copies of the same systems:
  - roundtripping (both ways) should be *exact*
  - same program should behave the same in similar conditions
  - for interoperability!
- Within different systems
  - e.g., browser/client - server
  - roundtripping should be *reasonable*
  - analogous programs should behave analogously
  - in analogous conditions
  - a weaker notion of interoperability



What again is an XML document?

Level		Data unit examples	Information or Property required
cognitive			
application			
tree adorned with...			
namespace	schema		nothing / a schema
tree			well-formedness
token	complex	<foo:Name t="8">Bob	
token	simple	<foo:Name t="8">Bob	
character		< foo:Name t="8">Bob	which encoding (e.g., UTF-8)
bit		10011010	

PSVI, Types, default values

Errors here ➜ no DOM!

Roundtripping Fail: Defaults in XSD

sparse.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="… >
<xs:element name="a">
  <xs:complexType><xs:sequence>
    <xs:element maxOccurs="unbounded" ref="b"/>
  </xs:sequence></xs:complexType>
</xs:element>
<xs:element name="b">
  <xs:complexType><xs:attribute name="c"/>
  </xs:co
</xs:eleme
```

Test.xml

```
<a>
  <b/>
  <b c="bar"/>
</a>
```

Parse & Validate

count(//@c) = ??

Query

count(//@c) = ??

Serialize

Test-sparse.xml

```
<a>
  <b/>
  <b c="bar"/>
</a>
```

Test-full.xml

```
<a>
  <b c="foo"/>
  <b c="bar"/>
</a>
```

full.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="… >
<xs:element name="a">
  <xs:complexType><xs:sequence>
    <xs:element maxOccurs="unbounded" ref="b"/>
  </xs:sequence></xs:complexType>
</xs:element>
<xs:element name="b">
  <xs:complexType><xs:attribute name="c" default="foo"/>
  </xs:complexType>
</xs:element></xs:schema>
```

only diff!

Can we think of Test-sparse and -full as “the same”?

XML is not (always) self-describing!

- Under external validation
- Not just legality, but content!
  - The PSVIs have different information in them!

Roundtripping "Success": Types

bare.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="a">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="b"  maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="b"/>
</xs:schema>
```

Test.xml

```
<a>
  <b/>
  <b/>
</a>
```

Parse & Validate

count(//b) = ??

Query

Serialize

Test.xml

```
<a>
  <b/>
  <b/>
</a>
```

typed.xsd

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="a"/>
  <xs:complexType name="atype">
    <xs:sequence>
      <xs:element ref="b"  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="b" type="btype"/>
  <xs:complexType name="btype"/>
</xs:schema>
```

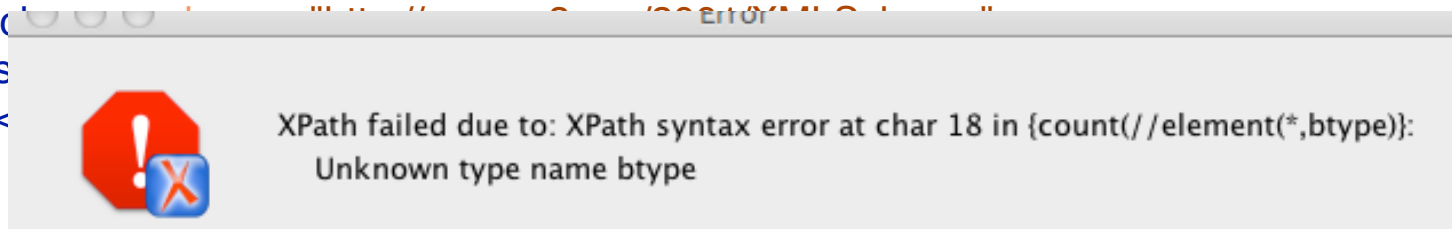
count(//b) = ??

only diff!

Roundtripping “Issue”: Types

bare.xsd

```
<xs:sc
  <xs
    <
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="b"/>
</xs:schema>
```



count(//element(*,btype)) = ??

Test.xml

```
<a>
  <b/>
  <b/>
</a>
```

Parse & Validate → Query → Serialize

Test.xml

```
<a>
  <b/>
  <b/>
</a>
```

typed.xsd

```
<xs:schema xmlns:x="http://www.w3.org/2001/XMLSchema">
  <xs:element name="a" type="atype"/>
  <xs:complexType name="atype">
    <xs:sequence>
      <xs:element ref="b" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="b" type="btype"/>
  <xs:complexType name="btype"/>
</xs:schema>
```

count(//element(*,btype)) = ??

The Essence of XML

- Thesis:
  - "XML is touted as an external format for representing data."
- Two properties
  - Self-describing
    - Destroyed by external validation,
    - i.e., using application-specific schema for validation
  - Round-tripping
    - Destroyed by defaults and union types

http://bit.ly/essenceOfXML2

XQuery

XQuery

- is a language for **querying** XML data
  - one aspect of data **manipulation**
- 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

W3C speak

- 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

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>
  <city>Manchester</city>
   </person>
...
```

- "FLWOR" is pronounced "flower"
- a FLWOR expression has 5 possibly overlapping parts:
  - **F**or e.g., for *$x* in doc("people.xml")/contactList/person
  - **L**et e.g., let *$i := 3* let *$n := $x*/name/firstname
  - **W**here e.g., where *$x*/@categ = "friend"
  - **O**rder by e.g., order by *$x*/name/lastname ascending
  - **R**eturn e.g., return concat(*$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

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>
 <city>Manchester</city>
  </person>
...
```

- 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

let *variable* (as *datatype*)? := *expression*

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

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

```
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
```

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>
 <city>Manchester</city>
   </person>
...
```

FLWOR expressions

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>
 <city>Manchester</city>
   </person>
...
```

- 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:

```
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

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>
 <city>Manchester</city>
   </person>
...
```

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

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.,

```
<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

```
for $b in /contactlist/person
let $name as element() := $b/name/firstname
return  <short> { $name/text() }</short>
```

FLWOR expressions

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

```
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>
```

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>
 <city>Manchester</city>
   </person>
...
```

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.,

```
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 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>
...
```

cities.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>
...
```

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,
```

```
<sales>
 {for $p in doc("contactlist-john-doe.xml")/contactlist/person
  for $c in doc("cities.xml")/citylist/city
  where $p/city = $c/name
  return
    (for $i in 1 to fn:count($c/club)
     return concat("Dear ", $p/name/firstname,
       ", do you like ", $c/club[$i], " ?"))}
 </sales>
```

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>
 <city>Manchester</city>
   </person>
...
```

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>
...
```

FLOWR expressions

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

product.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

sale.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<sales>
   <sale productId="1">
      <mrq>180$</mrq>
      <ytd>18.87% up</ytd>
      <margin>5%</margin>
   </sale
...
```

XQuery FLOWR expressions

- **order by** allows us to order sequences before we return them
- we can combine several orderings into new ones *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.,

```
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 pr

To summarize the departments from Manchester, use:
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> };
```

to be used e.g., in

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

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!

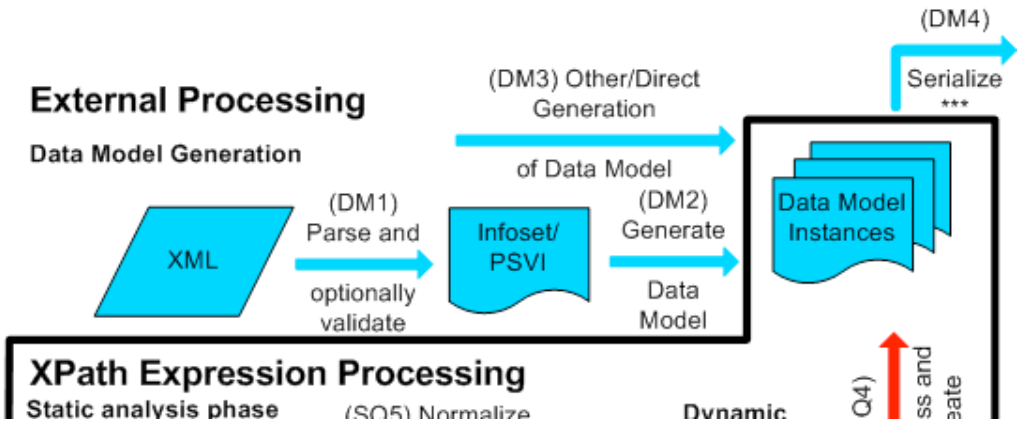
```
<this>
   <xmlFragment/>
   <is>acutally a bunch of xquery</is>
   <constructor/>
   <which>
      <returns>a sequence of nodes</returns>
   </which>
</this>//returns
```

XQuery query!!

result sequence!

```
<returns>a sequence of nodes</returns>
```

XQuery, schemas, and types



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

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

```
<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, schemas, and types

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

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

```
<?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>
```

```
…
<xs:element name="nlist">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="nEl"
               type="uli:number"
               maxOccurs="unbounded"/>
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:complexType name="number">
         <xs:simpleContent>
         <xs:extension base="xs:integer">
         <xs:attribute name="attr"
            default="15"/>
         </xs:extension>
         </xs:simpleContent>
   </xs:complexType>
```

And remember count(//element(*,btype)) = ??

XQuery, schemas, and types

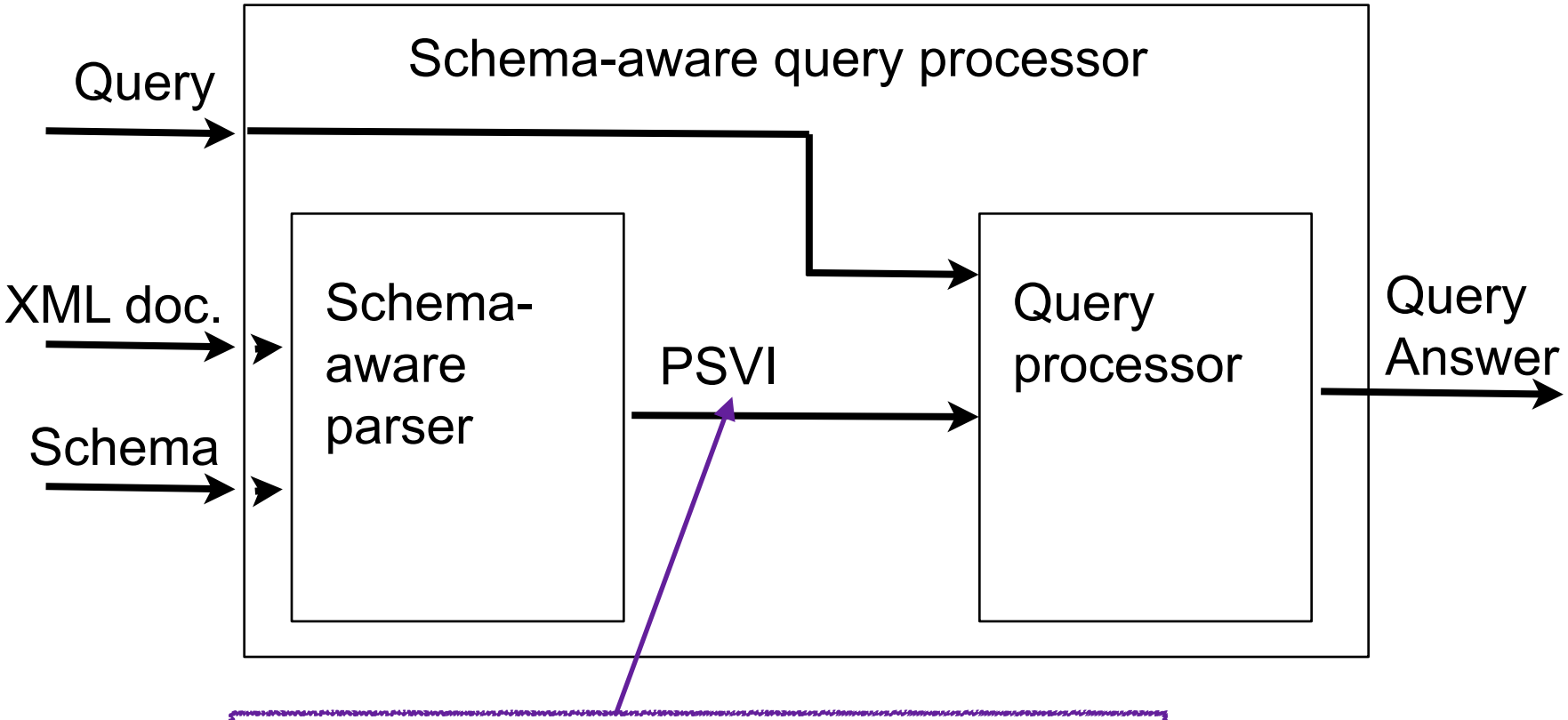
- if you query documents that are associated with a **schema**, you can exploit **schema-aware query answering**, eg XML Schema aware like SAXON-SA:
  - WXS has **types**, e.g., answer to this query may vary depending on your schema

```
module namespace;
import schema namespace uli="www.uli.org" at "test4.xsd";
for $m in doc('Untitled5.xml')//element(*, uli:A)
return $m/uli:friend/text()
```

```
<?xml version="1.0" encoding="UTF-8"?>
<uli: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>
</uli:list>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<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:element>

<xs:complexType name="A">
  <xs:sequence>
    <xs:element name="friend" type='xs:string'
      minOccurs = '3'  maxOccurs ='5'/>
  </xs:sequence></xs:complexType>

  <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>
```

XQuery, schemas, and types: the PSVI

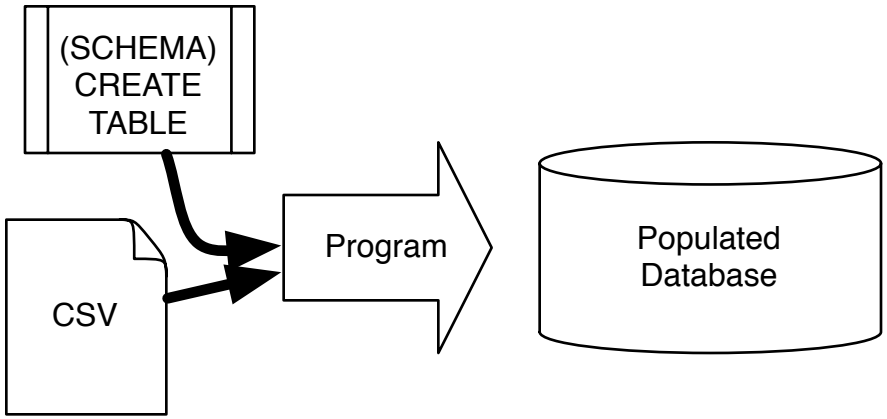


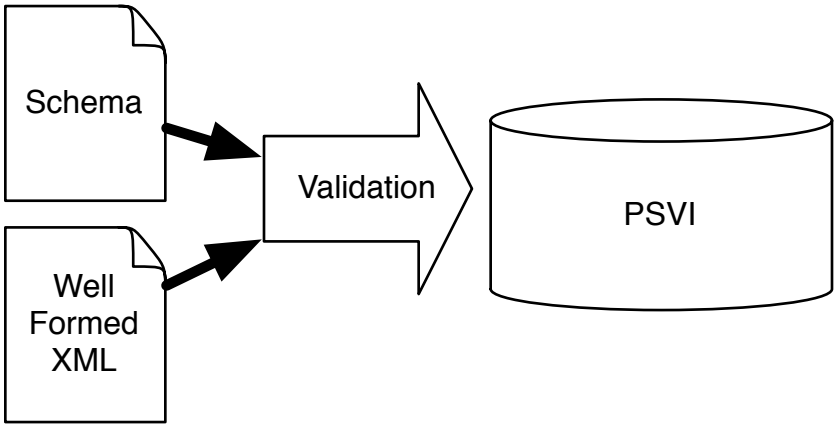
Post-**s**chema-**v**alidation **i**nfoset:
Internal Rep. adorned with schema information
e.g., a tree adorned with default values & types

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





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

Error Handling

Errors - everywhere & unavoidable!

- E.g., CW3 - what to do for (7 + 9)/(3 - (1 + 2))?
- Preventing errors: make
  - errors hard or impossible to make
    - but NOT make doing things hard or impossible
  - doing the right thing easy and inevitable
  - detecting errors easy
  - correcting errors easy
- Correcting errors requires noticing them:
  - ? Fail silently
  - ? Fail randomly
  - ? Fail differently (interop problem)
  - ? Don't fail but do something "ok"

Always think of crying children!

Postel's Law

Be liberal in what you accept,
and
conservative in what you send.

- Liberality
  - Many DOMs, all expressing the same thing
  - Many surface syntaxes (perhaps) for each DOM
- Conservativity
  - What *should* we send?
    - It depends on the receiver!
  - Minimal standards?
    - Well-formed XML?
    - Valid according to a popular schema/format?
    - HTML?

Error Handling - Examples

- XML has **draconian** error handling
  - 1 Well-formedness error…BOOM

- **Unknown properties.** User agents must ignore a declaration with an unknown property. For example, if the style sheet is:

```
h1 { color: red; rotation: 70minutes }
```

the user agent will treat this as if the style sheet had been

```
h1 { color: red }
```

- **Illegal values.** User agents must ignore a declaration with an illegal value. For example:

```
img { float: left }        /* correct CSS 2.1 */
img { float: left here }   /* "here" is not a value of 'float' */
img { background: "red" }  /* keywords cannot be quoted */
img { border-width: 3 }    /* a unit must be specified for length values */
```

- Check out CSS's error handling rules!

XML Error Handling

- De facto XML motto about well-formed-ness:
  - be strict of what you accept,
  - and strict in what you send
  - Draconian error handling
  - Severe consequences on the Web
    - And other places
  - Fail early and fail hard
- What about higher (schema) levels?
  - Validity and other analysis?
  - Most schema languages are poor at error reporting
    - How about XQuery's type error reporting?
    - XSD schema-aware parser report on
      - error location (which element) and
      - what was expected
      - …so we could fix things!?

Typical Schema Languages

- Grammar (and maybe type based)
  - Validation: either succeeds or FAILs
  - Restrictive by default: what is not permitted is forbidden
    - what happens in this case?

```
<a value="3" date="2014"/>
```

```
element a { attribute value { text }, empty }
```

  - Error detection and reporting
    - Is at the *discretion* of the validating parser
    - "Not accepted" **may** be the only answer the validator gives!
    - The point where an error is detected
      - might not be the point where it occurred
      - might not be the most helpful point to look at!
    - Compare to programs!
      - Null pointer deref
        - » Is the right point the deref or the setting to null?

Our favourite Way

Be liberal in what you accept,
and
conservative in what you send.

- Adore Postel's Law
- Explore before prescribe
- Describe rather than define
- Take what you can, when/if you can take it
  - don't be a horrible person/program/app!
- Design your **formats** so that extra or missing stuff is (can be) OK
  - Irregular structure!
- Adhere to the task at hand

How many middle/last/first names does your address **format** have?!

...

Next Week:
a schema language for
good error handling!

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
    - ACIDITY ensures that data conforms
- 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?

Consider....

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

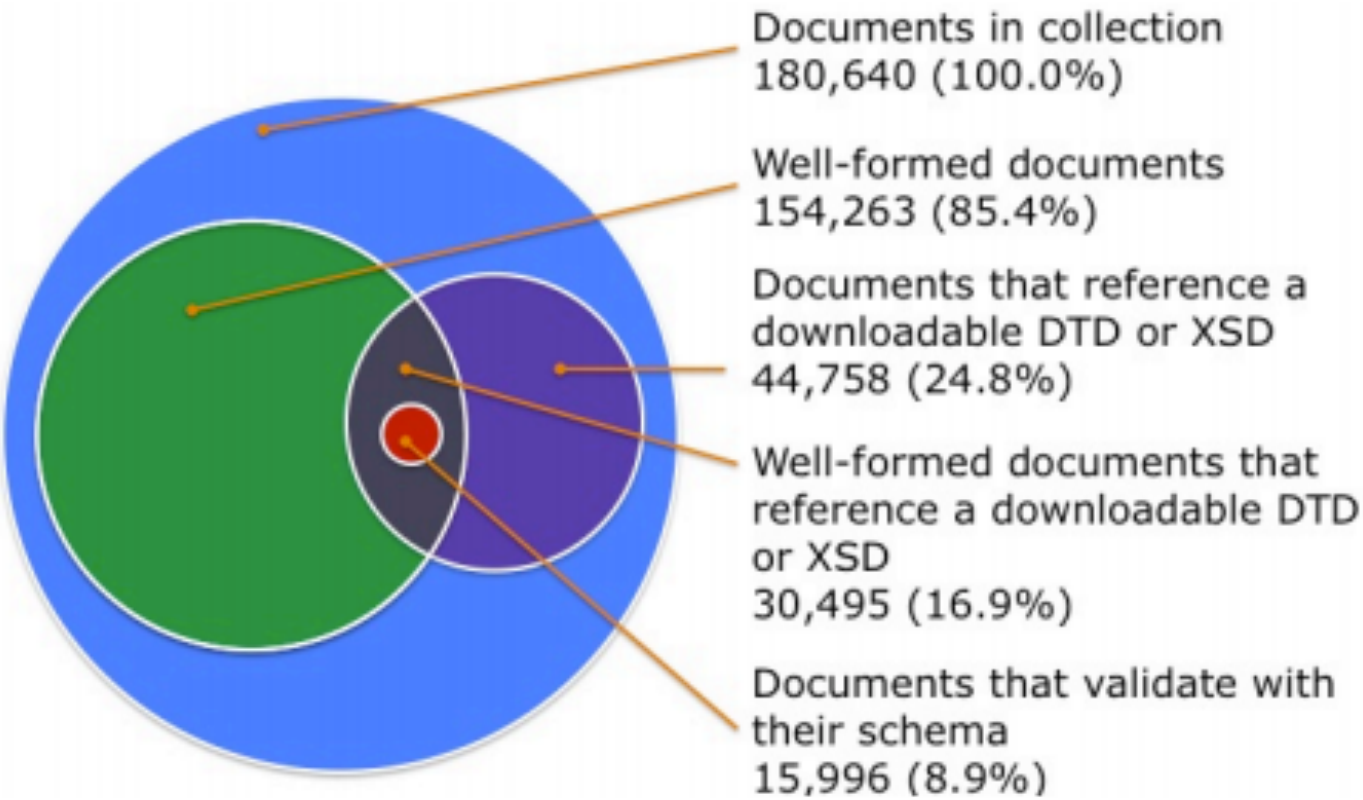


Figure 1: Summary of the Quality of the XML Web.

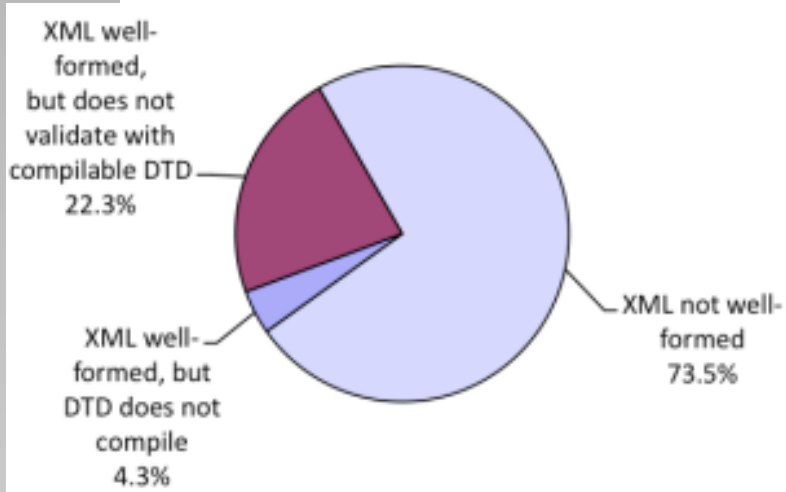
The Quality of the XML Web [2013]

Today's XML

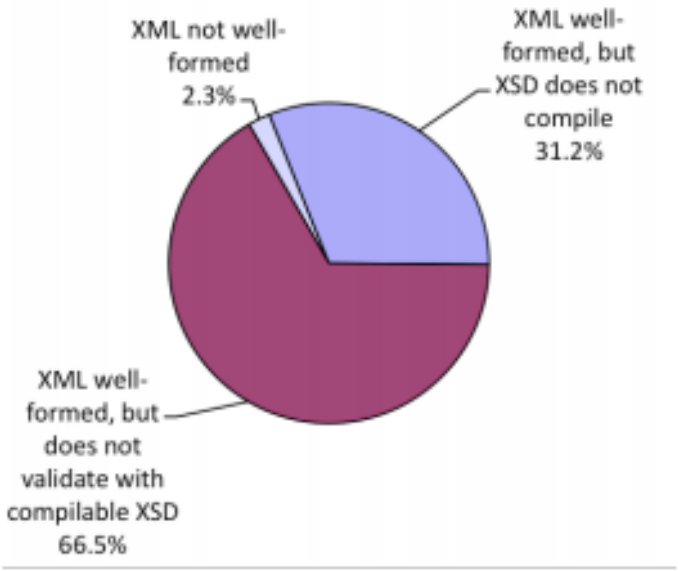
- Weird facts:
  - 15% are not well formed
    - 66.4% of non-well formed documents have a DOCTYPE!
- "Validity is rare on the web. Just over 10% of the well-formed documents are also valid."
  - Is there a difference between DTDs and WXS?

The Quality of the XML Web [2013]

Invalid wrt DTD versus XSD



Distribution of causes for non-validation: DTD.



Distribution of causes for non-validation: XSD

Coursework this week

- Get to know tools/oxygen better:
  - use it to test your understanding of XPath, XQuery, XSD,
  - collect a fine sample of XML docs, XSDs, RNGs, …
- Q4:
  - think first, then
  - use tools to answer questions
- CW4:
  - XQuery for namespace analysis
- M4: Typed queries in XPath & XSD
  - do this before SE4
- SE4: robustness, schemas, and different query styles
  - think/read about robustness
  - do M4 before you do this