COMP60411: Modelling Data on the Web
Tree Data Models
Week 3

Tim Morris & Uli Sattler
University of Manchester
Reminder: Plagiarism & Academic Malpractice

• We assume that you have all by now successfully completed the Plagiarism and Malpractice Test

• ...if you haven’t:
  do so before you submit any coursework (assignment or assessment)

• ...because we work under the assumption that
  – you know what you do
  – you take pride in your own thoughts & your own writing
  – you don’t steal thoughts or words from others

• ...and if you don’t, and submit coursework where you have copied other people’s work without correct attribution
  it costs you at least marks or more, e.g., your MSc
Reminder

We maintain 3 sources of information:

- **syllabus** …/pgt/COMP60411/syllabus/
- **materials** …/pgt/COMP60411/  
  - growing continuously  
  - with slides, reading material, etc  
  - with TA lab times
- **Blackboard via myManchester**  
  - growing continuously  
  - Forums  
    - General  
    - Week 1, Week 2, …  
  - Coursework
Coursework - Week 1 and 2

• Short essays, SE1: looks mostly good
  • use a **good spell & grammar checker**!
  • qualify your statements:
    • “All users can understand a picture” is NOT true!
  • use suitable verbs & technical terms
  • think about
    • caveats,
    • examples,
    • concrete situations
  • avoid
    • **non sequiturs**

**Always:**

• check our **feedback** in the **rubrics**
• if you can’t find them, ask us in **labs**
Today

We will encounter new things:

**Tree data models:**

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

**General concepts:**
more on
- Self-Describing
- Trees
- Regular Expressions
- Internal & External Representation, Parsing
- Validation, valid, …
- Format
XML
a data model with
a tree-shaped internal representation
XML

• is another formalism for the representation of *semi-structured data*
  – e.g., used by UniProt
  – suitable for humans and computers
• is *not* designed to specify the lay-out of documents
  – this what html, css and others are for
• alone will not solve the problem of **efficiently querying (web) data**:
  – we might have to use RDBMSs technology as well
  – see COMP62421
A brief history of XML

- **GML** (Generalised Markup Language), 60ies, IBM
- **SGML** (Standard Generalised Markup Language), 1985:
  - flexible, expressive, with DTDs
  - custom tags
- **HTML** (Hypertext Markup Language), early 1990ies:
  - application of SGML
  - designed for presentation of documents
  - single document type, presentation-oriented tags, e.g., `<h1>...</h1>`
  - led to the web as we know it
  - a **W3C** standard
  - subset/fragment of SGML
  - designed
    • to be “web friendly”
    • for the exchange/sharing of data
    • to allow for the principled decentralized extension of HTML and
      the elimination or radical reduction of **errors** on the web
- **XHTML** is an application of XML
  - almost a fragment of HTML
A rough map of a part of Acronym World

- HTML is an application of SGML
- SGML is basically a restriction of XML
- XML is an application of XSLT
- XSLT describes XQueries
- XQueries describes XPath
- XPath is part of RelaxNG
- RelaxNG describes Schematron
- Schematron describes XML Schema
- XML Schema describes DTD

Legend:
- Markup language formalism
- Schema language
- Query language
Back to our very simple XML example

Running example from last week:

```json
{name: {first: "Uli", last: "Sattler"},
  tel: 56182,
  tel: 56176,
  email: "sattler@cs.man.ac.uk"}
```

In (badly layed-out) XML:

```xml
<person><name><first>Uli</first><last>Sattler</last></name><tel>56182</tel><tel>56176</tel><email>sattler@cs.man.ac.uk</email></person>
```
Back to our very simple XML example

Running example from last week:

```
{name: {first:”Uli”, last: “Sattler”},
tel: 56182,
tel: 56176,
email:”sattler@cs.man.ac.uk”}
```

In (the same but better layed-out) XML:

```
<person>
  <name>
    <first>Uli</first>
    <last>Sattler</last>
  </name>
  <tel>56182</tel>
  <tel>56176</tel>
  <email>sattler@cs.man.ac.uk</email>
</person>
```
Back to our very simple XML example

Running example from last week:

```
{name: {first:"Uli", last: "Sattler"},
  tel: 56182,
  tel: 56176,
  email:"sattler@cs.man.ac.uk"}
```

In (still the same) XML with syntax highlighting:

```
<person>
  <name>
    <first>Uli</first>
    <last>Sattler</last>
  </name>
  <tel>56182</tel>
  <tel>56176</tel>
  <email>sattler@cs.man.ac.uk</email>
</person>
```
Back to our very simple XML example

Running example from last week:

```
{name: {first:”Uli”, last: “Sattler”},
tel: 56182,
tel: 56176,
email:”sattler@cs.man.ac.uk”}
```

In a different XML-based **format**, well layed-out, with syntax highlighting:

```
<person>
  <name first="Uli" last="Sattler"/>
  <phone>
    <number value="56182"/>
    <number value="56176"/>
  </phone>
  <email>
    <address value="sattler@cs.man.ac.uk"/>
  </email>
</person>
```

Design choices for **format** for your data affect query-ability, robustness.
A snippet of XML describing the above Dilbert cartoon

```xml
<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>
  <panels>
    <panel colour="none">
      <scene>Pointy-Haired Boss and Dilbert sitting at table.</scene>
      <bubbles>
        <bubble>
          <speaker>Dilbert</speaker>
          <speech>You haven’t given me enough resources to do my project.</speech>
        </bubble>
      </bubbles>
    </panel>
    ...
  </panels>
</cartoon>
```
What is XML?

- XML is a specialization of SGML
- XML is a W3C standard since 1998, see http://www.w3.org/XML/
- XML was designed to be simple, generic, and extensible
- an XML document is a piece of text that
  - describes
    - structure
    - data
  - has internal representation of a tree: DOM tree or infoset
  - is divided into smaller pieces called elements (associated with nodes in tree), which can
    - contain elements - nesting!
    - contain text/data
    - have attributes
- an XML document consists of (some administrative information followed by)
  - a root element containing all other elements
And here is the full XML document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cartoon SYSTEM "cartoon.dtd">
<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>

<panels>

....

</panels>

</cartoon>
```

Administrative Information

Root element
What is XML? (ctd)

The above mentioned administrative information of an XML document:

1. **XML declaration**, e.g., `<?xml version="1.0" encoding="iso-8859-1"?>` (optional) identifies the
   - XML version (1.0) and
   - character encoding (iso-8859-1)

2. **document type declaration** (optional) references a (specific) schema called **Document Type Definition**
   - e.g. `<!DOCTYPE cartoon SYSTEM "cartoon.dtd">`
     1. a DTD constrains the structure, content & tags of a document
     2. can either be local or remote

3. then we find the **root element**

4. which in turn contains other elements with possibly more elements....
XML Elements

- **elements** are delimited by **tags**
- **tags** are enclosed in angle brackets, e.g., `<panel>`, `</from>`
- tags are case-sensitive, i.e., `<FROM>` is not the same as `<from>`
- we distinguish
  - **start tags**: `<...>`, e.g., `<panel>`
  - **end tags**: `</...>`, e.g., `</from>`
- a pair of matching start- and end tags delimits an **element** (like parentheses)
- **attributes** specify properties of an element
e.g., `<cartoon copyright="United Feature Syndicate">`
Example

And here is the full XML document

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cartoon SYSTEM "cartoon.dtd">
<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>
  <panels>
    ....
  </panels>
</cartoon>
```
Example

And here is the full XML document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cartoon SYSTEM "cartoon.dtd">
<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>
    <panels>
        ....
    </panels>
</cartoon>
```
XML Core Concepts: elements *(the main concept)*

- arbitrary number of attributes is allowed
- each *attr-decl* is of the form *attr-name="attr-value"*
- but each *attr-name* occurs **at most once** in one element
- the *content* can be
  - empty
  - text and/or
  - one or more elements
  - ...those *contained* elements are the element’s **child elements**
- an empty element can be abbreviated as
  `<element-name attr-decl1 ... attr-decln/>`
Example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cartoon SYSTEM "cartoon.dtd">
<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>
  <panels>
    ....
  </panels>
</cartoon>
```
XML Core Concepts: Prologue - XML declaration

More at http://www.w3.org/TR/REC-xml/

<?xml param1 param2 ...?>

Each parami is in the form

parameter-name="parameter-value"

<?xml version="1.0" encoding="US-ASCII" standalone="yes"/>

Parameters for

- the xml version used within document
- the character encoding
- whether document is standalone or uses external declarations (see validity constraint for when standalone="yes" is required)

An XML document should have an XML declaration (but does not need to)
XML Core Concepts:
Prologue - Doctype declaration

```
<!DOCTYPE element-name PUBLIC "pub-id" "f-name.dtd" |
   SYSTEM "f-name.dtd" |
   [dt-declarations]>
```

- at most one such declaration, before root element
  - links document to (a simple) schema describing its structure
- `element-name` is the name of the root element of the document
- the optional `dt-declarations` is
  - called internal subset
  - a list of document type definitions
- the optional `f-name.dtd` refers to the external subset also containing
document type definitions

- e.g., `<!DOCTYPE html PUBLIC "http://www.abc.org/dtgs/html.dtd"
  "http://www.abc.org/dtgs/html.dtd" >`
What is XML? (ctd)

- in XML, the set of tags/element names is not fixed
  - ...you can use whatever you want (within spec)
  - in HTML, the tag set is fixed
  - `<h1>`, `<b>`, `<ul>`...

- elements can be **nested**, to arbitrary depth
  
  ```xml
  <p> <p> <p> ...</p> </p> </p>
  </p>
  ```

- the same **element name** can occur many times in a document,
  - e.g.,
    
    ```xml
    <p>...</p><p> ...</p>...
    ```

- XML itself is **not** a markup language, but we can **specify** markup languages with XML
  - an XML document can **contain** or **refer to** its specification: `!DOCTYPE`
How to view or edit XML?

- **XML is not really for human consumption**
  - far too verbose
  - in contrast to HTML, your **browser** won’t easily help:
    - you can only do a “view source” or
    - first *style it* (using XSLT or CSS, later more) to transform XML into HTML

- **XML is text**, so you can use your favourite editor, e.g., emacs in XML mode
- Or you can use an **XML editor**, e.g., XMLSpy, Stylus Studio, `<oXygen/>`, MyEclipse, and many more
- `<oXygen/>` runs on the lab machines
  - it supports many features
  - query languages
  - schemas, etc.
  - has been given to us for free: license details are in Blackboard
XML versus HTML

- XML is always case sensitive, i.e., "Hello" is different from "hello"
  - HTML isn’t: it uses SGML's default "ignore case"
- in XML, all tags must be present
  - in HTML, some "tag omission" may be permissible (e.g., <br>)
- in XML, we have a special way to write empty elements <myname/>
  - which can’t be used in HTML
- in XML, all attribute values must be quoted, e.g., <name lang= "eng">...
  - in SGML (and therefore in HTML) this is only required if value contains space
- in XML, attribute names cannot be omitted
  - in HTML they may be omitted using shorttags
XML versus JSON

XML
• no arrays/lists/vectors
• IR is tree-shaped
• an element can have many elements with same name
• has many schema languages
• W3C standard
• supports namespaces
• has its own query languages
  – XPath
  – XQuery
  – XSLT
• supported by numerous libraries
• comprehensive datatype support

JSON
• has arrays/lists/vectors
• not tree-shaped
• an object cannot have repeated key
• has 1 schema languages
• not a W3C standard
• supported by numerous libraries
• rather basic datatype support
• less verbose than XML

Read & think critically about these and other points.
When is an XML document well-formed?

An XML document is **well-formed** if
1. there is exactly one root element
2. tags, <, and > are correct (incl. no un-escaped < or & in character data)
3. tags are properly nested
4. attributes are unique for each tag and attribute values are quoted
5. no comments inside tags

Well-formedness is a very weak property: basically, it only ensures that we can parse a document into a tree.
Well-formedness is a very weak property: basically, it only ensures that we can parse a document into a tree...
An Internal Representation for XML documents

- An XML document is a **piece of text**
  - it has tags, etc.
  - it has **no** nodes, structure, successors, etc.
  - it may have whitespace, new lines, etc.

- having an **InR** for XML documents makes many things easier:
  - talking about **structure**: documents, elements, nodes, child-nodes etc.
  - ignoring things like **whitespace** issues, etc.
  - implementing software that handles XML
  - specifying schema languages, other formalisms around it
  ➡ think of relational model as basis for rel. DBMSs

- this has motivated the
  - **XML Information Set** recommendation,
  - Document Object Model, **DOM**, and others

- unsurprisingly, they model an XML document as a **tree**
<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>[Diagram]</td>
<td>nothing</td>
</tr>
<tr>
<td>namespace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>schema</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree</td>
<td></td>
<td>well-formededness</td>
</tr>
<tr>
<td>token</td>
<td>complex</td>
<td>&lt;foo:Name t=&quot;8&quot;&gt;Bob</td>
</tr>
<tr>
<td></td>
<td>simple</td>
<td>&lt;foo:Name t=&quot;8&quot;&gt;Bob</td>
</tr>
<tr>
<td></td>
<td>character</td>
<td>&lt; foo:Name t=&quot;8&quot;&gt;Bob</td>
</tr>
<tr>
<td></td>
<td>bit</td>
<td>10011010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>which encoding (e.g., UTF-8)</td>
</tr>
<tr>
<td>Level</td>
<td>Data unit examples</td>
<td>Information or Property required</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>cognitive</td>
<td></td>
<td></td>
</tr>
<tr>
<td>application</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>tree adorned with...</strong></td>
<td><img src="image" alt="Diagram" /></td>
<td></td>
</tr>
<tr>
<td>namespace</td>
<td></td>
<td>nothing</td>
</tr>
<tr>
<td>schema</td>
<td></td>
<td>a schema</td>
</tr>
<tr>
<td><strong>tree</strong></td>
<td></td>
<td>well-formedness</td>
</tr>
<tr>
<td>complex</td>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
<td></td>
</tr>
<tr>
<td>token</td>
<td></td>
<td></td>
</tr>
<tr>
<td>simple</td>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
<td></td>
</tr>
<tr>
<td>character</td>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
<td>which encoding (e.g., UTF-8)</td>
</tr>
<tr>
<td>bit</td>
<td><code>10011010</code></td>
<td></td>
</tr>
</tbody>
</table>

**Diagram Note:**
- Internal Representation
- External Representation
DOM
an API
and
an internal Representation for XML
DOM trees as an InR for XML documents

A simple example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mytext content="medium">
    <title>Hallo!</title>
    <content>Bye!</content>
</mytext>
```

Diagram:

- **Document**
  - `nodeType = DOCUMENT_NODE`
  - `nodeName = #document`
  - `nodeValue = (null)`

- **Element**
  - `nodeType = ELEMENT_NODE`
  - `nodeName = mytext`
  - `nodeValue = (null)`
  - `firstchild`
  - `lastchild`
  - `attributes`

  - **Element**
    - `nodeType = ELEMENT_NODE`
    - `nodeName = title`
    - `nodeValue = (null)`
    - `firstchild`

  - **Element**
    - `nodeType = ELEMENT_NODE`
    - `nodeName = content`
    - `nodeValue = (null)`
    - `firstchild`

  - **Text**
    - `nodeType = TEXT_NODE`
    - `nodeName = #text`
    - `nodeValue = Hallo!`

  - **Text**
    - `nodeType = TEXT_NODE`
    - `nodeName = #text`
    - `nodeValue = Bye!`

  - **Attribute**
    - `nodeType = ATTRIBUTE_NODE`
    - `nodeName = content`
    - `nodeValue = medium`

  - **PI**
    - `nodeType = Processing Instruction`
DOM: InR for XML documents

- we will use the **DOM tree** as an internal representation: it can be viewed as an implementation of the slightly more abstract **infoset**
- DOM is a **platform & language independent specification of an API** for accessing an XML document in the form of a tree
  - “DOM parser” is a parser that outputs a DOM tree
  - but DOM is much more

XML document, i.e., text → parser e.g., Dom parser → standard API, eg. DOM tree → your application

 serializer
Programmatic Manipulation of XML Documents

As a rule, whenever we manipulate (XML) documents in an application, we should use standard APIs:
Parsing & Serializing XML documents

- **parser:**
  - reads & analyses XML document
  - may generate parse tree that reflect document’s element structure
    - e.g., DOM tree
      - with nodes labelled with
        - tags,
        - text content, and
        - attributes and their values

- **serializer:**
  - takes a data structure, e.g., some trees, linked objects, etc.
  - generates an XML document

- **round tripping:**
  - XML → tree → XML
  - ...doesn’t have to lead to identical XML document...more later
A simple example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mytext content="medium">
    <title>
        Hallo!
    </title>
    <content>
        Bye!
    </content>
</mytext>
```

**DOM tree nodes**

- **Document**
  - `nodeType`: DOCUMENT_NODE
  - `nodeName`: #document
  - `nodeValue`: (null)

- **Element**
  - `mytext`
    - `nodeType`: ELEMENT_NODE
    - `nodeName`: mytext
    - `nodeValue`: (null)
    - `firstchild`
    - `lastchild`
    - `attributes`
      - `content`
        - `nodeName`: content
        - `nodeValue`: medium

- **Text**
  - `Hallo!`
    - `nodeName`: #text
    - `nodeValue`: Hallo!

- **Text**
  - `Bye!`
    - `nodeName`: #text
    - `nodeValue`: Bye!
DOM trees as an InR for XML documents

- In general, we have the following correspondence:
  - XML document $D$ $\rightarrow$ tree $t(D)$
  - element $e$ in $D$ $\rightarrow$ node $t(e)$ in $t(D)$
  - empty element $\rightarrow$ leaf node
  - root element $e$ in $D$ $\rightarrow$ not root node in $t(D)$, but document node

```xml
<?xml version="1.0" encoding="UTF-8"?>
<mytext content="medium">
  <title>Hallo!</title>
  <content>Bye!</content>
</mytext>
```
DOM trees as an InR for XML documents

- In general, we have the following correspondence:
  - XML document $D$ $\rightarrow$ tree $t(D)$
  - element $e$ in $D$ $\rightarrow$ node $t(e)$ in $t(D)$
  - empty element $\rightarrow$ leaf node
  - root element $e$ in $D$ $\rightarrow$ *not* root node in $t(D)$, but document node

- DOM’s **Node interface** provides the following attributes to navigate around a node in the DOM tree:

  - parentNode
  - previousSibling
  - Node
  - nextSibling
  - firstChild
  - ChildNodes
  - lastChild
  - attributes

- and also methods such as appendChild, hasAttributes, insertBefore, etc.
DOM by example

A little Python3 example:

“if 1st child of mytexts is “Hallo” return the content of 2nd child”

1. let a parser build the DOM of mydocument.xml
   
   ```python
   import xml.dom.minidom
   import sys
   filename_xml = sys.argv[1]
   dom = xml.dom.minidom.parse(filename_xml)
   ```

2. Retrieve all “mytext” nodes into a NodeList interface:
   
   ```python
   mytextNodes = dom.getElementsByTagName("mytext")
   ```

3. Navigate and retrieve all contents:
   
   ```python
   for textnode in mytextNodes:
       titleNode = textnode.childNodes[1]
       titlenodetext = titleNode.childNodes[0]
       if titlenodetext.nodeValue == "Hallo!":
           bodynode = textnode.childNodes[3]
           print("The hallo message is %s " % bodynode.childNodes[0].nodeValue)
   ```

mydocument.xml:

```xml
<mytext content="medium">
  <title>Hallo!</title>
  <body>Bye!</body>
</mytext>
```
Parsing XML

- **DOM parsers** parse an XML document into a DOM tree
  - this might be huge/not fit in memory
  - your application may take a few relevant bits from it and build an own datastructure, so (DOM) tree was short-loved/built in vain

- **SAX parsers** work very differently
  - they don’t build a tree but
  - go through document depth first and “shout out” their findings...
Self-Describing
Self-describing?!

- XML is said to be **self-describing**...what does this mean?

```
<a123>
  <b345 b345="$%#987">Hi there!</b345>
</a123>
```

- ...is this well-formed?
- ...can you understand what this is about?
- Let’s again compare to **CSV** (comma separated values):
  - each line is a **record**
  - commas separate **fields** (and no commas in fields!)
  - each record has the same number of fields

```
Bijan, Parsia, 2.32
Uli,     Sattler, 2.24
```

- ...can you understand what this is about?
Self-describing?!

- One way of translating our example into XML
  - ...can you understand what this is about?

```
<csvFile>
  <record>
    <field>Bijan</field>
    <field>Parsia</field>
    <field>2.32</field>
  </record>
  <record>
    <field>Uli</field>
    <field>Sattler</field>
    <field>2.24</field>
  </record>
</csvFile>
```
Self-describing?!

- Let’s consider a **self-describing CSV (ExCSV)**
  - first line is **header** with **field names**
  - ...can you understand what this is about?

- We could even **generically** translate such CSVs in XML:

```xml
<csvFile>
  <record>
    <name>Bijan</name>
    <surname>Parsia</surname>
    <room>2.32</room>
  </record>
  <record>
    <name>Uli</name>
    <surname>Sattler</surname>
    <room>2.24</room>
  </record>
</csvFile>
```

or, manually, even better:

```xml
<addresses>
  <address>
    <name>Bijan</name>
    <surname>Parsia</surname>
    <room>2.32</room>
  </address>
  <address>
    <name>Uli</name>
    <surname>Sattler</surname>
    <room>2.21</room>
  </address>
</addresses>
```
Self-describing versus Guessability

- We can go a long way by **guessing**
  - CSV is *not easily* guessable
    - requires background knowledge
  - ExCSV is *more* guessable
    - still some guessing
    - could read the field tags and guess intent
    - had to guess the record type *address*
  - Guessability is tricky

- Is self-describing just being more or less guessable?
Self-describing

The Essence of XML (Siméon and Walder 2003):
“From the external representation one should be able to derive the corresponding internal representation.”

- **External**: the XML document, i.e., text!
- **Internal**:
  - e.g., the DOM tree, our application’s interpretation of the content
  - seems easy, but: in
    - `<room>2.32</room>` is “2.32” a string or a number?
    - `<height>2.32</height>` is “2.32” a string or a number?
    - …what should a DOM/your parser do?

- Are CSV, ExCSV, XML self-describing?
Self-describing

- Given
  1. a base format, e.g., ExCSV
  2. a/some specific document(s), e.g.,

  - what suitable data structure can we extract?

    - CSV, ExCSV: tables, flat records, arrays, lists, etc.
    - XML: labelled, ordered trees of (unbounded) depth!

- Clearly, you could parse *specific* CSV files into trees, but you need to use *extra*-CSV information for that

<table>
<thead>
<tr>
<th>Name, Surname, Room</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bijan, Parsia, 2.32</td>
</tr>
<tr>
<td>Uli, Sattler, 2.24</td>
</tr>
</tbody>
</table>

Schemas!
Schemas: what are they?

A **schema** is a description

- of **DBs**: describes
  - tables,
  - their names and their attributes
  - keys, keyrefs
  - integrity constraints
Schemas: what are they?

A **schema** is a description

- **of DBs**: describes
  - tables,
  - their names and their attributes
  - keys, keyrefs
  - integrity constraints
- **of XML documents**: describes
  - tag names
  - attribute names
  - structure:
    - how elements are nested
    - which elements have which attributes
  - data: what values (strings? numbers?) go where
Schemas: why?

- **RDBMS**
  - No database without schema
  - DB schema determines tables, attributes, names, etc.
  - Query optimization, integrity, etc.

- **XML (and JSON)**
  - No schema *needed* at all!
  - Well-formed XML can be
    - parsed to yield data that can be
    - manipulated, queried, etc.
  - Non-well formed XML....not so much
  - Well-formededness is a universal minimal schema
Schemas for XML: why?

• Well-formedness is minimal
  – any name can appear as an element or attribute name
  – any shape of content/structure of nesting is permitted
• Few applications want that…
• we’d like to rely on/share a **format**
  – core concepts that result in
  – core (tag & attribute) **names** and
  – **intended structure**
  – **intended data types**
    e.g., string for names, integer for age

  – although you might want to keep it **extensible & flexible**
Schemas for XML: why?

• A schema describes aspects of documents:
  – what’s legal:
    what a document can/may contain
  – what’s expected:
    what a document must contain
  – what’s assumed:
    default values

• Two modes for using a schema
  – descriptive:
    • describing documents
    • for other people
    • so that they know how to serialize their data
  – prescriptive:
    • prevent your application from using wrong documents

<addresses>
  <address>
    <name>Bijan</name>
    <surname>Parsia</surname>
  </address>
  <address>
    <name>Uli</name>
    <minit>M<minit>
    <surname>Sattler</surname>
    <room>2.21</room>
  </address>
</addresses>
Benefits of an (XML) schema

- **Specification**
  - you document/describe/publish your format
  - so that it can be used across multiple implementations
- As **input** for applications
  - applications can do **error-checking** in a **format independent** way
    - checking whether an XML document conforms to a schema can be done by a **generic** tool (see CW2),
    - no need to be changed when schema changes
    - automatically!
Benefits of an (XML) schema

- **Specification**
- As **input** for applications
  - applications can do **error-checking** in a **format independent** way

![Diagram](image_url)

- **Input/Output**
- **Generic tools**
- **Your code**

- **Schema**
- **XML document**
- **Schema-aware parser**
- **Serializer**
- **Standard API** (e.g., DOM or Sax)
- **Your application**

- informs/throws exception if doc doesn’t **validate** against schema!
- may contain additional information from schema: PSVI
RelaxNG, a very powerful schema language for XML

to formulate

Schema

XML document

Schema-aware parser

Serializer

Standard API eg. DOM or Sax

your application

your 3rd schema language
RelaxNG: a schema language

- RelaxNG was designed to be a **simpler** schema language
  - than XML Schema (XSD)
  - described in a readable on-line book by Eric Van der Vlist
- and allows us to describe XML documents in terms of their **tree abstractions**:
  - no default attribute values
  - no entity declarations
  - no key/uniqueness constraints
  - minimal datatypes: only “token” and “string” (like DTDs, similar to JSON Schema)
  - but a mechanism to use XSD datatypes

- since it is so simple/flexible
  - it’s (claimed/designated to be) easy to use
  - it doesn’t have complex constraints on description of element content like determinism/1-unambiguity
  - it’s claimed to be reliable
  - but you need other tools to do other things (like datatypes and attributes)
RelaxNG: “a” side of Validation

General: reasons why one would want to validate an XML document:

• ensure that structure is ok
• ensure that values in elements/attributes are of the correct data type
• generate PSVI to work with
• check constraints on co-occurrence of elements/how they are related
• check other integrity constraints, eg. a person’s age vs. their mother’s age
• check constraints on elements/their value against external data
  – postcode correctness
  – VAT/tax/other numeric constraints
  – spell checking

...only few of these checks can be carried out by validating against schemas...

RelaxNG was designed to

1. describe/validate structure and
2. link to datatype validators to type check values of elements/attributes
RelaxNG: basic principles

• RelaxNG is based on **patterns** (similar to XPath expressions):
  – a pattern is a description of a set of valid node sets
  – we can view our example as different combinations of different parts, and design **patterns** for each

A first RelaxNG schema:

```xml
grammar {  
  start =  
    element name {  
      element first { text },  
      element last { text }  
    }  
}
```

To describe documents like:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<name>
  <first>Harry</first>
  <last>Potter</last>
</name>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<name>
  <first>Magda</first>
  <last>Potter</last>
</name>
```
RelaxNG: good to know

RelaxNG comes in 2 syntaxes

• the compact syntax
  – succinct
  – human readable

• the XML syntax
  – verbose
  – machine readable

✓ Trang converts between the two, phew!
  (and also into/from other schema languages)

✓ Trang can be used from <Oxygen>

```xml
<grammar
  xmlns="http:..."
  xmlns:a="http:..">
  <datatypeLibrary="http:...">
    <start>
      <element name="name">
        <element name="first"><text/></element>
        <element name="last"><text/></element>
      </element>
    </start>
  </datatypeLibrary>
</grammar>
```
RelaxNG - to describe structure:

- 3 kinds of **patterns**, for the 3 “central” nodes:
  - text
    - attribute age \{ text \},
    - attribute type \{ text \},
  - attribute
  - element
    - element name \{ 
      - element first \{ text \},
      - element last \{ text \} 
    \}
  - these can be combined:
    - ordered groups
    - unordered groups
    - choices
- we can constrain cardinalities of patterns
- text nodes
  - can be marked as “data” and linked
- we can specify libraries of patterns
RelaxNG: ordered groups

- we can **name** patterns
- in “chains”
- we can use **regular expressions**:, ?, *, |, and +

```xml
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <person age="41">
    <name>
      <first>Harry</first>
      <last>Potter</last>
    </name>
    <address>4 Main Road</address>
    <project type="epsrc" id="1">
      DeCompO
    </project>
    <project type="eu" id="3">
      TONES
    </project>
  </person>
  ....
</people>
```

```plaintext
grammar { start = people-element

people-element = element people
  { person-element+ }

person-element = element person {
  attribute age { text },
  name-element,
  address-element+,
  project-element*}

name-element = element name {
  element first { text },
  element middle { text }?,
  element last { text } }

address-element = element address { text }

project-element = element project {
  attribute type { text },
  attribute id {text},
  text }
```
RelaxNG: different styles

- so far, we modelled ‘element centric’...we can model ‘content centric’:

```
grammar { start = people-description

people-description = element people
        { person-description+ }

person-description = element person {
                attribute age { text },
                name-description,
                address-description+,}+

name-description = element name {
                element first { text },
                element middle { text }?,
                element last { text } }

address-description = element address { text }

project-description = element project {
                attribute type { text },
                attribute id {text},
                text }}
```

```
grammar { start =
        element people {people-content}

people-content =
        element person { person-content }+

person-content = attribute age { text },
                name-content,
                address-description+,}+

name-content = element first { text },
                element middle { text }?,
                element last { text }

project-content = attribute type { text },
                attribute id {text},
                text }
```

Claim: A document is valid wrt left one iff it is valid wrt right one.
Documents being **valid** wrt schema

A document is valid wrt S1 iff it is valid wrt S2.

What does that mean?
Documents being valid wrt schema

- Validity of XML documents wrt a RelaxNG schema
  - is a complex concept because RelaxNG is a **powerful** schema language:
    - other schema languages, e.g. DTDs, are less powerful, so
      - describing things is harder,
      - describing some things is impossible, but
      - validity is easily defined
    - we concentrate here on **simple** RelaxNG schemata:
      - for each element name X,
        use a “macro” X-description
      - only patterns of the form
        - start = X-description
        - X-description = element X { text }
          or
        - X-description = element X expression
          where expression is a **regular expression over** “…-description”s
          …and exactly 1 such pattern per “…-description"
Simple RelaxNG schemas

- Is this schema simple?

- for each element name X, use a “macro” X-description

- only patterns of the form

  - start = X-description

  - X-description = element X { text }

  or

  - X-description = element X expression

  where expression is a regular expression over “…-description”s

  ...and exactly 1 such pattern per “…-description”

```plaintext
grammar { start =  people-description

people-description = element people { person-description+ } 

person-description = element person {
    attribute age { text },
    name-description,
    address-description+, 
    project-description*}

name-description = element name {
    element first { text },
    element middle { text }?,
    element last { text } }

address-description = element address { text }

project-description = element project {
    attribute type { text },
    attribute id {text},
    text  }
```
Simple RelaxNG schemas

- Is this schema simple?

- for each element name X, use a “macro” X-description
  - only patterns of the form
    - start = X-description
    - X-description = element X { text }
    - or
    - X-description = element X expression
      where expression is a [regular expression over “…-description”’s](#)
      …and exactly 1 such pattern per “…-description”

```plaintext
grammar { start = people-description

people-description = element people { person-description+ } 

person-description = element person { name-description, 
  address-description+, 
  project-description*}

name-description = element name {first-description, 
  middle-description?, 
  last-description }

first-description = element first { text }
middle-description = element middle { text }
last-description = element last { text }

address-description = element address { text }

project-description = element project { text } }
```
Documents described by a RelaxNG schema

- An node \( n \) with name \( X \text{ matches } \) an expression
  - \text{element } X \{\text{text}\} \quad \text{if } X \text{ has a single child node of text content}
  - \text{element } X \text{ expression} \quad \text{if the sequence of } n\text{'s child node names matches expression, after dropping all 
"-description" in expression}
Documents described by a RelaxNG schema

- A node $n$ with name $X$ matches an expression
  - `element X {text}` if $X$ has a single child node of text content
  - `element X expression` if the sequence of $n$’s child node names matches expression, after dropping all “-description” in expression

- Eg., matches `element name {first-description, middle-description?, last-description }`
Documents described by a RelaxNG schema

• A node $n$ with name $X$ matches an expression
  – element $X$ \{text\} if $X$ has a single child node of text content
  – element $X$ expression if the sequence of $n$’s child node names
    matches expression,
    after dropping all “-description” in expression

• Eg., $\text{name}$ matches element name \{first-description, middle-description?, last-description \}

An XML document $D$ is valid wrt a simple RelaxNG schema $S$ if
  – $D$’s root node name is $X$ iff $S$ contains \text{start} = X$-description
  – each node $n$ in $D$ matches its description,
    i.e., if $n$’s name is $X$, then $S$ contains a statement
      $X$-description = $Y$ and $n$ matches $Y.$
RelaxNG: validity by example

Is this document valid wrt this RelaxNG schema?

```
<?xml version="1.0" encoding="UTF-8"?>
<name>
  <first>Harry</first>
  <middle>Bob</middle>
</name>
```

```
grm { start = people-description

people-description = element person { person-description+ }

person-description = element name { name-description, }

name-description = element name {
  first-description, middle-description?, last-description }

first-description = element first { text }
middle-description = element middle { text }
last-description = element last { text }
}
```
RelaxNG: validity by example

Is this document valid wrt this RelaxNG schema?

```xml
<?xml version="1.0" encoding="UTF-8"?>
<name>
    <first>Harry</first>
    <middle>Bob</middle>
    <last>Potter</last>
</name>
```

```markdown
grammar { start = people-description

people-description = element people { person-description+ }

person-description = element person {
    name-description,}

name-description = element name {
    first-description, middle-description?,
    last-description }

first-description = element first { text }
middle-description = element middle { text }
last-description = element last { text }
}
```
RelaxNG: validity by example

Is this document valid wrt this RelaxNG schema?

```xml
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <person>
    <name>
      <first>Magda</first>
      <last>Potter</last>
    </name>
  </person>
</people>
```

```plaintext
grammar { start = people-description

people-description = element people { person-description+ }

person-description = element person {
  name-description,
}

name-description = element name {
  first-description, 
  middle-description?, 
  last-description 
}

first-description = element first { text }
middle-description = element middle { text }
last-description = element last { text }
}
```
Documents valid against RelaxNG schemas

- careful: “is valid” is different from “validates against”
RelaxNG: regular expressions in XML syntax

```xml
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <ref name="people-element"/>
  </start>
  <define name="people-element">
    <element name="people">
      <oneOrMore>
        <ref name="person-element"/>
      </oneOrMore>
    </element>
  </define>
  <define name="person-element">
    <element name="person">
      <attribute name="age">
        <text/>
      </attribute>
      <name-element/>
      <address-element/+>
      <project-element/*>
    </element>
  </define>
  <define name="name-element">
    <element name="name">
      <first>
        <text/>
      </first>
      <middle>
        <text/>
      </middle>?
      <last>
        <text/>
      </last>
    </element>
  </define>
  <define name="address-element">
    <element name="address">
      <text/>
    </element>
  </define>
  <define name="project-element">
    <element name="project">
      <attribute name="type">
        <text/>
      </attribute>
      <attribute name="id">
        <text/>
      </attribute>
    </element>
  </define>
</grammar>
```
RelaxNG: ordered groups

- we can combine patterns in **fancy ways**:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <person age="41">
    <name>
      <first>Harry</first>
      <last>Potter</last>
    </name>
    <address>4 Main Road</address>
    <project type="epsrc" id="1">DeCompO</project>
    <project type="eu" id="3">TONES</project>
  </person>
</people>
```
RelaxNG: structure description summary

- RelaxNG’s specification of structure differs from DTDs and XML Schema (XSD):
  - grammar oriented
  - 2 syntaxes with automatic translation
  - flexible: we can gather different aspects of elements into different patterns
  - unconstrained: no constraints regarding unambiguity/1-ambiguity/deterministic content model/Unique Particle Constraints/Element Declarations Consistent
  - we also have an “ALL” construct for unordered groups, “interleave” &:

  here, the patterns must appear in the specified order, (except for attributes, which are allowed to appear in any order in the start tag):

  ```
  element person {
    attribute age { text},
    attribute phone { text},
    name-element ,
    address-element+,
    project-element*}
  ```

  here, the patterns can appear any order:

  ```
  element person {
    attribute age { text } &
    attribute phone { text} &
    name-element &
    address-element+ &
    project-element*}
  ```
Remember: Benefits of an (XML) schema

- **Specification**
  - you document/describe/publish your format
  - so that it can be used across multiple implementations
- **As input** for applications
  - applications can do **error-checking** in a **format independent** way
    - checking whether an XML document conforms to a schema can be done by a **generic** tool (see CW3),
    - no need to be changed when schema changes
    - automatically!

```
Input/Output       Generic tools       Your code

Schema           Schema-aware parser       PSVI
XML document           Serializer        your application
```
Validity of XML documents w.r.t. RelaxNG

- Try <oXygen/>
  - for your coursework
  - to write XML documents and RelaxNG schemas
  - it can check
    - whether your document is well-formed and
    - whether your document conforms to your schema!
XPath
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
    • handles attributes differently
<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>tree adorned with...</td>
<td></td>
<td>nothing</td>
</tr>
<tr>
<td>namespace</td>
<td></td>
<td>a schema</td>
</tr>
<tr>
<td>schema</td>
<td></td>
<td>well-formedness</td>
</tr>
<tr>
<td>tree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>complex</td>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
<td></td>
</tr>
<tr>
<td>simple</td>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
<td></td>
</tr>
<tr>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
<td></td>
<td>which encoding</td>
</tr>
</tbody>
</table>

**Diagram**: The diagram shows a DOM tree with elements and attributes, illustrating the structure of XML documents. The diagram includes a namespace and a schema, indicating the context of XML data. The tree is parsed and serialized, highlighting the process of converting XML data into a readable format. The diagram also includes a choice between DOM tree, Infoset, and XPath, emphasizing the different methods of XML data representation.
XPath processing - a simplified view

- XPath expression
- XPath parser
- XPath tree
- XPath Execution Engine
- Schema-aware Parser
- Standard Datamodel
  - eg. DOM or XPath
- Node Sequence
- Input/Output
- Generic tools

- XML document
- Schema

Diagram showing the process of XPath processing with input/output and generic tools, leading to a schema-aware parser and standard datamodel.
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">
    ...
  </host>
</network>

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

★ XPath does not have separate nodes for CDATA sections (they are merged with their surrounding text)

```xml
<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
XPath - by example

This is the configuration of our network in the Boston office.

```
<?xml version="1.0" encoding="UTF-8"?>
<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">
    ...
  </host>
</network>
```
XPath - abbreviated syntax by example

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

XPath expression: */*[2]/*[1]/*[3]
XPath - abbreviated syntax

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

XPath expression: /*/*[1]
XPath - abbreviated syntax local 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">
<service>SMTP</service>
</host>
XPath - abbreviated syntax
local globally

XPath expression: `//*`
XPath - abbreviated syntax attributes in filters

XPath expression: //*[@name="agatha"]
Find more about XPath: read up and play with examples, e.g., in <oxygen>
XML Namespaces

or,

making things “simpler”

by

making them much more complex
An observation

• “plus” elements may occur in different situations
• e.g. in arithmetic expression (see CW2/3, M3) and in regular expressions:

\[
\begin{align*}
<\text{plus}> \\
<\text{int value}=4/> \\
<\text{int value}=5/> \\
</\text{plus}>
\end{align*}
\]
for 4+5

\[
\begin{align*}
<\text{plus}> \\
<\text{choice}> \\
<\text{star}>A</\text{star}> \\
<\text{star}>B</\text{star}> \\
</\text{choice}> \\
</\text{plus}>
\end{align*}
\]
for (A*|B*)+

• We have an element name conflict!
• How do we distinguish plus[arithmetic] and plus[reg-exp]?
  • semantically?
  • in a combined document?
Unique-ing the names (1)

• We can add some characters

  <calcplus>
  <int value="4"/>
  <int value="5"/>
  </calcplus>

  <regexplus>
  <choice>
  ...
  </choice>
  </regexplus>

• No name clash now
  • But the “meaningful” part of name (plus) is hard to see
  • “calcplus” isn’t a real word!
Unique-ing the names (2)

- We can use a separator or other convention

```
<calc:plus>
  <int value="4"/>
  <int value="5"/>
</calc:plus>
```

- No name clash now
  - The “meaningful” part of the name is clear
  - The disambiguator is clear
    - But we still can get clashes
      - e.g., 2 calc:plus with different rounding!?
    - Need a registry to coordinate?
Unique-\ing the names (3)

- Use URIs for disambiguation

\[
\begin{align*}
<\text{http://bjp.org/calc/:plus}> \\
<\text{int value}="4"/> \\
<\text{int value}="5"/> \\
</\text{http://bjp.org/calc/:plus}>
\end{align*}
\[
<\text{http://uli.org/calc/:plus}> \\
<\text{rat value}="4"/> \\
<\text{rat value}="5.5"/> \\
</\text{http://uli.org/calc/:plus}>
\]

- No name clash now
  - The “meaningful” part of the name clear
  - The disambiguator is clear
    - Clashes are hard to get
    - Existing URI allocation mechanism
  - But not well formed!
Uniquing the names (4)

- Combine the (2) and (3)!

```xml
<calc:plus
    xmlns:calc="http://bjp.org/calc/">
    <int value="4"/>
    <int value="5"/>
</calc:plus>
```

```xml
<regex:plus xmlns:regex="http://bjp.org/regex/">
    <choice>
        ...
    </choice>
</regex:plus>
```

- No name clash now
  - The “meaningful” part of the name clear
  - The disambiguator is clear
    - Clashes are hard to get
    - Existing URI allocation mechanism
  - But well formed!
- But the model doesn’t know
## Layered!

<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></td>
</tr>
<tr>
<td>namespace</td>
<td>schema</td>
<td>nothing</td>
</tr>
<tr>
<td>tree</td>
<td></td>
<td>well-formedness</td>
</tr>
<tr>
<td>token</td>
<td>complex</td>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
</tr>
<tr>
<td></td>
<td>simple</td>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
</tr>
<tr>
<td>character</td>
<td></td>
<td><code>&lt;foo:Name t=&quot;8&quot;&gt;Bob</code></td>
</tr>
<tr>
<td>bit</td>
<td></td>
<td>10011010</td>
</tr>
</tbody>
</table>

The encoding (e.g., UTF-8) is required for character data.
Anatomy & Terminology of Namespaces

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

- **Qualified names** ("QNames"), e.g., `calc:plus` consist of
  - **Prefix**, e.g., `calc`
  - **Local name**, e.g., `plus`

- **Expanded name**, e.g., `{http://bjp.org/calc/}plus`
  - they don’t occur in doc
  - but we can talk about them!

- **Namespace name**, e.g., `http://bjp.org/calc/`
We don’t need a prefix

```xml
<plus xmlns="http://bjp.org/calc/">
  <int value="4"/>
  <int value="5"/>
</plus>
```

```xml
<calc:plus xmlns:calc="http://bjp.org/calc/">
  <int value="4"/>
  <int value="5"/>
</calc:plus>
```

- We can have **default** namespaces
  - Terser/Less cluttered
  - Retro-fit legacy documents
  - Safer for non-namespace aware processors
- But trickiness!
  - What’s the expanded name of “int” in each document?

```xml
{http://bjp.org/calc/}int
```

```xml
{}int
```

- Default namespaces and attributes interact weirdly...
Multiple namespaces

- We can have **multiple declarations**
- Each declaration has a **scope**
- 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

```xml
<plus xmlns="http://bjp.org/calc/
     xmlns:n="http://bjp.org/numbers/ ">
  <n:int value="4"/>
  <n:int value="5"/>
</plus>

<plus xmlns="http://bjp.org/calc/"
      xmlns:n="http://bjp.org/numbers/">
  <int xmlns="http://bjp.org/numbers/
       value="4"/>  
  <int value="5"/>
</plus>
```
Let’s test our understanding...

 `<a:expression xmlns="foo1" xmlns:a="foo2" xmlns:b="bah" >
    <b:plus xmlns:a="foobah"> 
        <int value="3"/>
        <a:int value="3"/>
    </b:plus>
</a:expression>`

Let’s test our understanding via some Kahoot quiz: go to kahoot.it
Some more about NS in our future

- Issues: Namespaces are increasingly controversial
- Modelling principles
- Schema language support
Phew - Summary of today

We have seen many things - you’ll deepen your understanding in coursework:

Tree data models:

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

General concepts:

• Semi-structured data
• Self-Describing (again)
• Trees (again)
• Regular Expressions (again)
• Internal & External Representation, Parsing (again)
• Validation, valid, …
• Format
Next: Coursework Old & New

- Quiz
- Short essay
- M3: build a JSON schema (next week: RelaxNG!)
  - as an extension to a given one
  - for arithmetic expressions
  - test your schema, share tests
- CW3:
  - use python and DOM to parse XML document with arithmetic expression, compute value of arithmetic expression after validating it against RelaxNG schema
  - test your program, share tests

- do Quiz first, then CW3 and M3, then SE3
- …see you in the labs!