MANCHESTER 1824

The University of Manchester

COMP60411: Modelling Data on the Web

Tree Data Models

Week 2

Bijan Parsia & 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

Subscribe
Read
Contribue

MANCHESTER 1824

The University of Manchester

Coursework - Week 1

- Q1: looks good, will look better next week
- SE1: looks mostly good
  - use a **good spell checker!**
  - **answer the question!**
- M1:
  - …
- CW1:
  - …

Today

We will encounter many things:

Tree data models:

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

General concepts:

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

Extending Last Week’s Running Example

MANCHESTER 1824

The University of Manchester

Extended Running Example

- consider last week's example:
  - per person 1 data record
- now combine this with **management information**:
  - who supervises/line manages whom?

Employees

Employee ID	Postcode	City	…
1234123	M16 0P2	Manchester	…
1234124	M2 3OZ	Manchester	…
1234567	SW1 A	London	…
...	...	...	...

Management

Manager ID	Managee ID
1234124	1234123
1234567	1234124
1234124	1234567
...	...

- …what could go wrong?
- …what did go wrong?

Running Example (2)

- Take a few minutes and sketch 2 SQL queries:
  Q1: all Postcodes of *4th-level* managers
  Q2: “error” if we have a cyclic management structure

Employees

Employee ID	Postcode	City	…
1234123	M16 0P2	Manchester	…
1234124	M2 3OZ	Manchester	…
1234567	SW1 A	London	…
...	...	...	...

Management

Manager ID	ManageeID
1234124	1234123
1234567	1234124
1234123	1234567
...	...

Q1: Tricky..

Q1’: Postcodes of all managers:

```
SELECT Postcode
FROM Employees E, Management M
WHERE E.EmployeeID =
          M.ManagerID
```

Q1”: Postcode of 2nd level managers:

```
SELECT Postcode
FROM Employees E
INNER JOIN
   (SELECT ManagerID
    FROM Management M1, Management M2
    WHERE M1.ManageeID =  M2.ManagerID) M
ON E.EmployeeID = M.ManagerID
```

…more and more joins!

Q2: Tricky…

- Detecting management cycles of length 1:

```
SELECT EmployeeID
FROM Management M
WHERE M.ManageeID =
          M.ManagerID
```

- Detecting management cycles of length 2:

```
SELECT EmployeeID
FROM Employees E1
INNER JOIN
  (SELECT EmployeeID
   FROM Management M1, Management M2
   WHERE M1.ManageeID =  M2.ManagerID) M
ON E1.EmployeeID = M.ManagerID
```

- …where do we stop?

A new example: UniProt, a Protein Database

- a research community based & curated knowledge base of
  - 550K protein sequences,
  - comprising 192M amino acids
  - abstracted from 220K references.
- Proteins largely determine how (parts of) living things work and interact
  - how/where diseases work
- used for a variety of research into
  - diseases
  - genetics
  - (personalized) drugs

MANCHESTER 1824

The University of Manchester

UniProt › UniProtKB   Downloads

Search | Blast * | Align * | Retrieve | ID Mapping *

Search in: Protein Knowledgebase (UniProtKB)   Query: [ ] Search Clear Fields »

Q9BX63 (FANCJ_HUMAN) Reviewed, UniProtKB/Swiss-Prot

Last modified August 10, 2010. Version 80. History...

Contribute
Send fe
Read co

Clusters with 100%, 90%, 50% identity | Documents (6) | Third-party data

Customize display · Names · Attributes · General annotation · Ontologies · Alt products · Sequence annotation · Sequences · References · Web links · Cross-refs · Entr

Names and origin

Protein names	*Recommended name:* **Fanconi anemia group J protein** Short name=Protein FACJ EC=3.6.4.13 *Alternative name(s):* ATP-dependent RNA helicase BRIP1 BRCA1-interacting protein C-terminal helicase 1 Short name=BRCA1-interacting protein 1 BRCA1-associated C-terminal helicase 1
Gene names	Name: **BRIP1** Synonyms:BACH1, FANCJ
Organism	**Homo sapiens (Human)** [Complete proteome]
Taxonomic identifier	9606 [NCBI]
Taxonomic lineage	Eukaryota › Metazoa › Chordata › Craniata › Vertebrata › Euteleostomi › Mammalia › Eutheria › Euarchontoglires › Primates › Haplorrhini › Catarrhini ›

Protein attributes

Sequence length	1249 AA.
Sequence status	Complete.
Protein existence	Evidence at protein level.

General annotation (Comments)

Function	DNA-dependent ATPase and 5' to 3' DNA helicase required for the maintenance of chromosomal stability. Acts late in the Fanconi anemia pathway, a involved in the repair of DNA double-strand breaks by homologous recombination in a manner that depends on its association with BRCA1. Ref.1
Catalytic activity	ATP + H_2O = ADP + phosphate.

Protein data from UniProt

UniProt

- provides a **web query interface** to Uniprot **database**
- e.g., query http://www.uniprot.org/uniprot/ for 'BRCA'

- ...biologists need to integrate, share, query, analyse, and search this data
- ...so what format is/should it be in?
- ...or what format should it be made available in to be integrated with other data?

MANCHESTER 1824

The University of Manchester

Protein data from UniProt in a table (1)

Protein Full Name	Short Name	Alternative Name 1	Alternative Name 2	Alternative Name 3	Gene 1	Gene 2	Gene 3	...	Organism	Taxon 1	Taxon 2	…
Fanconi anemia group J	FACJ	ATP-dependent RNA helicase BRIP1	BRCA1-interacting protein C-termin	BRCA1-interacting protein 1	BRIP1	BACH1	FANCJ		Halorubrum phage HF2	Viruses	dsDNA viruses, no RNA stage	…
ATP-dependent helicase	N/A	N/A	N/A	N/A	helicase	N/A	N/A		Gallus gallus / Chicken	Eukaryota	Metazoa	…
…	…	…	…	…	…	…	…	…	…	…	…	…

Protein data from UniProt in many tables (2)

Proteins

Protein ID	Full Name	Short Name	Organism	...
1234123	Fanconi anemi	FACJ	Halorubrum phage HF2	...
1234567	ATP-dependent	N/A	Gallus gallus / Chicken	...
	...	...	...	...

Protein-names

Protein ID	Alternative Name
1234123	ATP-dependent RNA helicase BRIP1
1234123	BRCA1-interacting protein C-terminal helicase 1
1234123	BRCA1-interacting protein 1
	...

Protein-genes

Protein	Genes
1234123	BRIP1
1234123	BACH1
1234567	helicas
	...

...

too many joins!

MANCHESTER 1824

The University of Manchester

Protein data from UniProt in an XML doc (1)


```xml
<?xml version="1.0" encoding="UTF-8"?>
<uniprot xmlns="http://uniprot.org/uniprot" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://uniprot.org/uniprot http://www.uniprot.org/support/docs/uniprot.xsd">
  <entry dataset="Swiss-Prot" created="2005-01-04" modified="2010-08-10" version="80">
    <accession>Q9BX63</accession>
    <accession>Q3MJE2</accession>
    <accession>Q8NCI5</accession>
    <name>FANCJ_HUMAN</name>
    <protein>
      <recommendedName ref="1">
        <fullName>Fanconi anemia group J protein</fullName>
        <shortName>Protein FACJ</shortName>
      </recommendedName>
      <alternativeName>
        <fullName>ATP-dependent RNA helicase BRIP1</fullName>
      </alternativeName>
      <alternativeName>
        <fullName>BRCA1-interacting protein C-terminal helicase 1</fullName>
        <shortName>BRCA1-interacting protein 1</shortName>
      </alternativeName>
      <alternativeName>
        <fullName>BRCA1-associated C-terminal helicase 1</fullName>
      </alternativeName>
    </protein>
    <gene>
      <name type="primary">BRIP1</name>
      <name type="synonym">BACH1</name>
      <name type="synonym">FANCJ</name>
    </gene>
```

…….Protein data from UniProt in an XML doc (2)

```xml
  <organism>
        <name type="scientific">Homo sapiens</name>
        <name type="common">Human</name>
        <dbReference type="NCBI Taxonomy" id="9606" key="2"/>
        <lineage>
           <taxon>Eukaryota</taxon>
           <taxon>Metazoa</taxon>
           <taxon>Chordata</taxon>
           <taxon>Craniata</taxon>
           <taxon>Vertebrata</taxon>
           <taxon>Euteleostomi</taxon>
           <taxon>Mammalia</taxon>
           <taxon>Eutheria</taxon>
           <taxon>Euarchontoglires</taxon>
           <taxon>Primates</taxon>
           <taxon>Haplorrhini</taxon>
           <taxon>Catarrhini</taxon>
           <taxon>Hominidae</taxon>
           <taxon>Homo</taxon>
        </lineage>
     </organism>
     <reference key="3">
        <citation type="journal article" date="2001" name="Cell" volume="105" first="149"  last="160">
           <title>BACH1, a novel helicase-like protein, interacts directly with BRCA1 and contributes to its DNA repair
function.</title>
           <authorList>
              <person name="Cantor S.B."/>
              <person name="Bell D.W."/>
              <person name="Ganesan S."/>
              <person name="Kass E.M."/>
              <person name="Drapkin R."/>
```

Two **pain points** common to both examples

Storing data in RDBMs/tables may require

- **Many** joins
  - due to irregular structure
    - varying number of ‘values’ for certain attributes
    - e.g., phone number, email, …
    - e.g., author, alternative name, Protein Names
  - making queries tricky/complicated, thus easy-to-get-wrong
- **Recursive** joins
  - due to unbounded depth,
  - e.g., “cyclic management”

MANCHESTER
1824

The University of Manchester

Alternative to Tables:
Semi-Structured Data Models

Database Alternatives to Tables

- **Trees,** underlying various **semi-structured data models**:
  - OEM
  - Lore
  - **JSON**
  - **XML**

- **Graphs**

- what are they?
- what are they good at?
- Schema Languages: how do we describe ‘legal structures’?
- Data Manipulation: how do we interact with them?

The Basics First: Semi-structured data

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

there is structure!

but not too much structure!

- predates XML
- is an attempt to reconcile
  - (Web) document view and
  - (DB) strict structures
- is data organised in semantic entities, where
  - similar entities are grouped together
  - entities in same group may not have same fields
- often defined as a **possibly nested set** of **field-value pairs**
- order of fields is not necessarily important
  - e.g.: do we have sets or lists of telephone numbers?
  - ….. fixing an order allows to give meaning to rank
- not all fields may be required
- carries its own description

aka **attribute-value** pairs

The Basics First: Semi-structured data

field

value

Example (ctd):

Values can in turn be **structured**:

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

And we can have **several values** for the same field:

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

The Basics First: Semi-structured data

Important: are field-value pairs **lists** or **sets**?

I.e., is

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

the same as

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

(yes if f-v-ps are **sets**, no if they are **lists**)

MANCHESTER 1824

The University of Manchester

The Basics First: Semi-structured data

Important: does white space matter?

I.e., is

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

the same as

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

We need an **Internal Representation**

to know when
two pieces of semi-structured data are the same, and
to determine what matters

The Basics First: trees as InternRepr for SSD

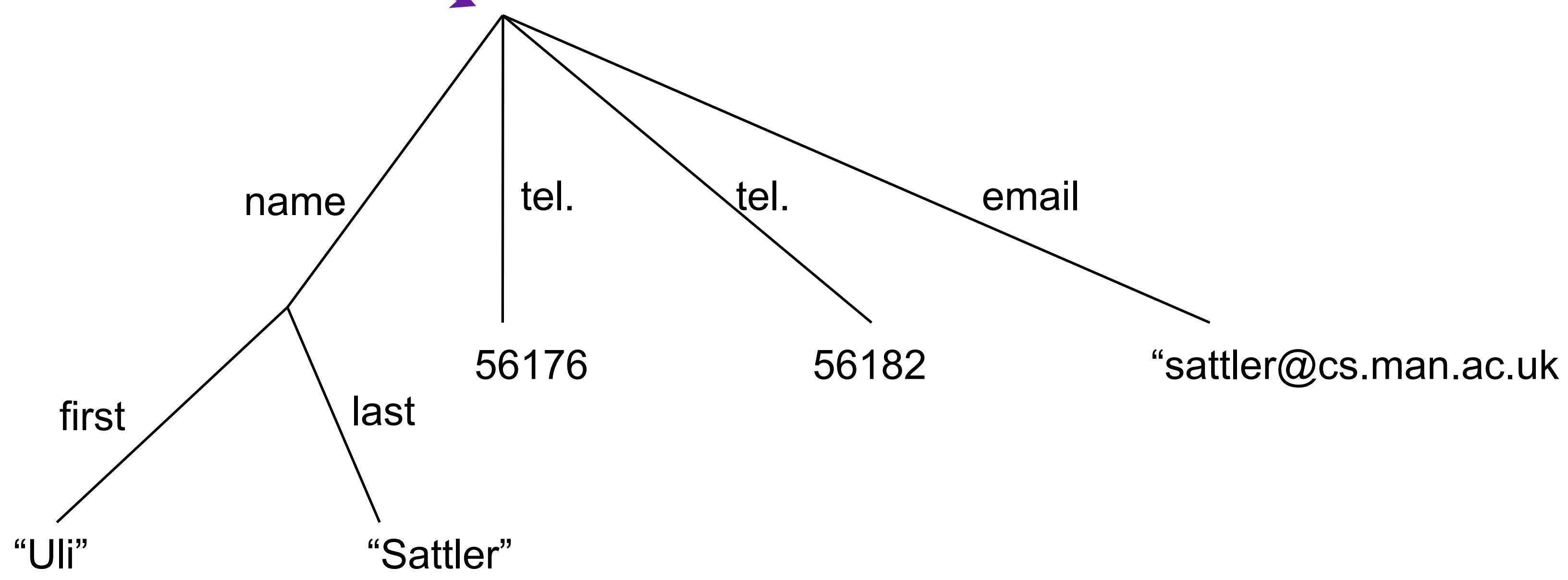
Let's view/treat

nested field-value pairs

as

trees

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



The Basics First: trees as InternRepr for SSD

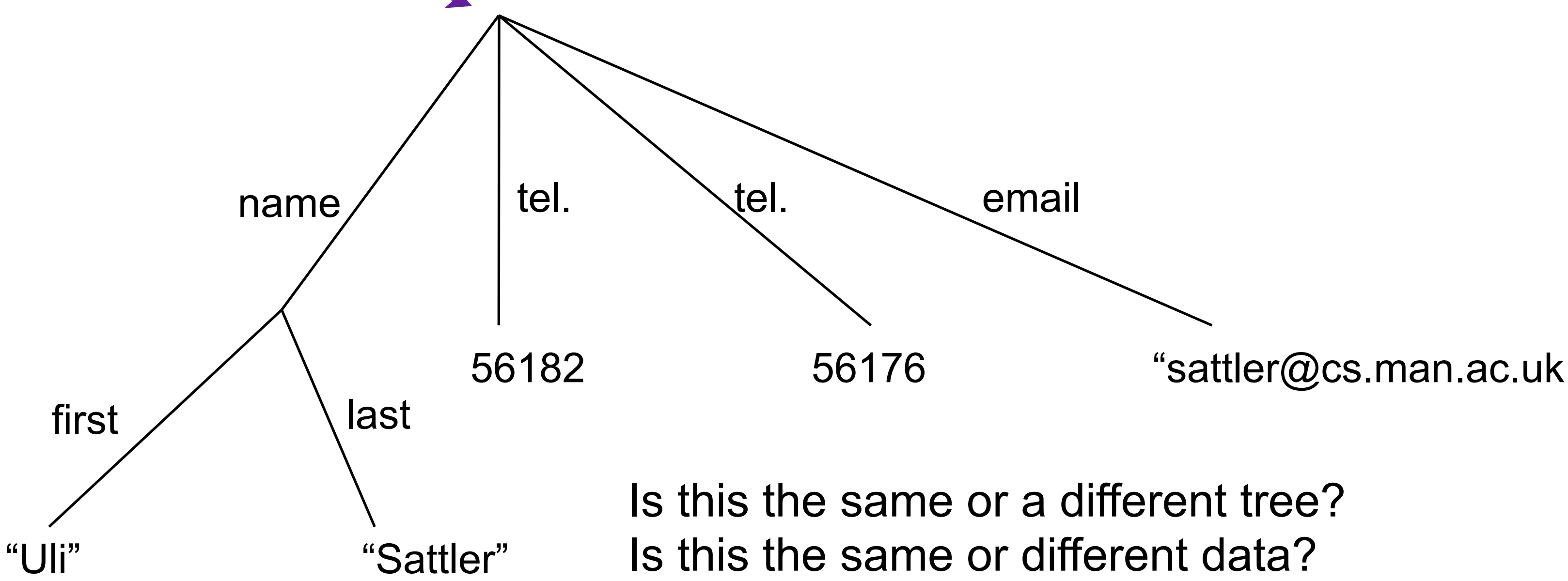
Let's view/treat

nested field-value pairs

as

trees

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

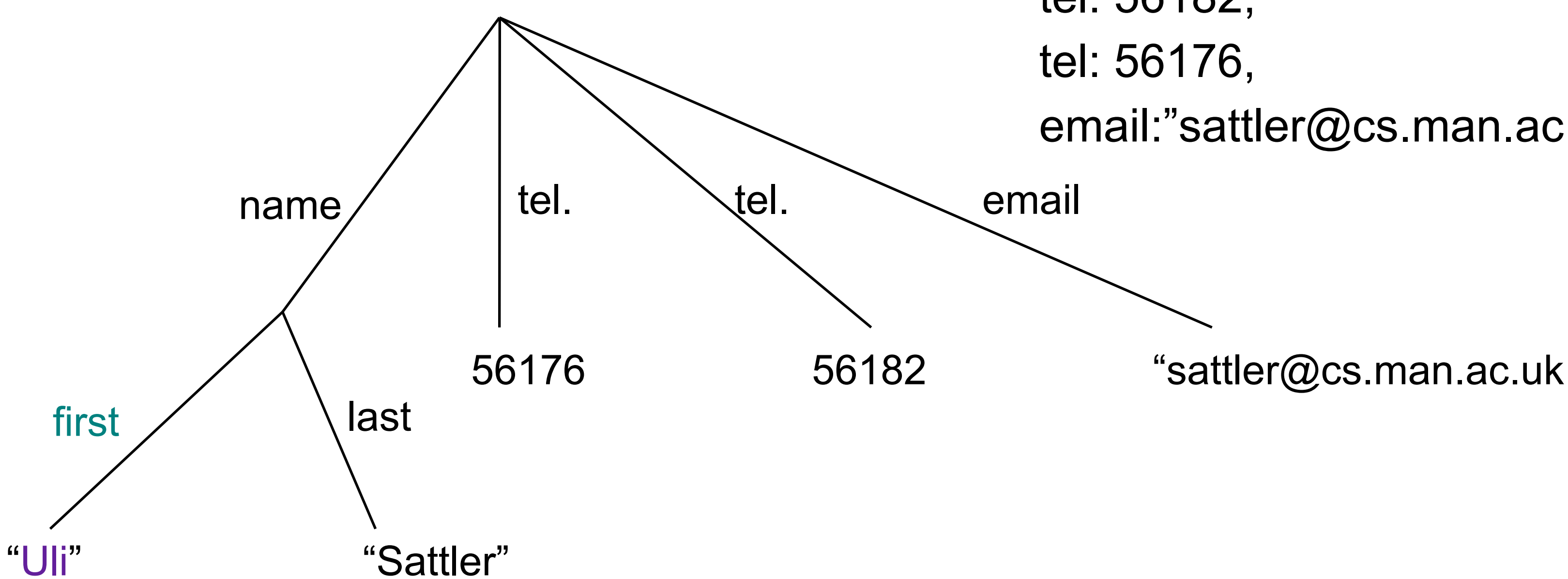


Is this the same or a different tree?
Is this the same or different data?

The Basics First: trees as InternRepr for SSD

- In general, a piece of SSD/**nested set** of **field-value pairs**,
  - can be represented as a **tree**
    - **leaf** nodes standing for single **data items**
    - **inner** nodes carry no label
    - **edges** labelled with **field names**

{name: {first:"Uli", last: "Sattl
tel: 56182,
tel: 56176,
email:"sattler@cs.man.ac.ul



Semi-structured data: tuples with variations

We can easily represent **nested tuples**

[[[Uli, Sattler], 56176, sattler@cs.man.ac.uk],
 [Bijan, 56183, 783 4672, bparsia@cs.man.ac.uk],
 [Leo, 8488342, leo@gmx.com]]

as sets of field-value pairs
even if they have *missing* or *duplicated* pairs
...best if we know which element belongs to what
e.g., is " 783 4672" Bijan's telephone number? his email address? age?

{person:
{name: {first: "Uli", last: "sattler}, tel: 56176, email: "sattler@cs.man.ac.uk"}
person:
{name: "Bijan", tel: 56183, tel: 783 4672,
email: "bparsia@cs.man.ac.uk"}
person:
{name: "Leo", tel: 8488342, email: "leo@gmx.com"}}

Semi-structured data: tuples with variations

We can easily represent **nested tuples**

[[[Uli, Sattler], 56176, sattler@cs.man.ac.uk],
[Bijan, 56183, 783 4672, bparsia@cs.man.ac.uk],
[Leo, 8488342, leo@gmx.com]]

as sets of field-value pairs
even if they have *missing* or *duplicated* pairs
...but also without knowing role of elements:

{*1*:
{*1*: {1: “Uli”, *2*: “sattler}, *2*: 56176, *3*: “sattler@cs.man.ac.uk”}
2:
{*1*: “Bijan”, *2*: 56183, *3*: 783 4672, *4*: “bparsia@cs.man.ac.uk”}
3:
{*1*: “Leo”, *2*: 8488342, *3*: “leo@gmx.com”}}

SSD: representing relational data

Consider two relations :

R	a	b	c
	a1	b1	c1
	a2	b2	c2

S	c	d
	c2	d2
	c3	d3
	c4	d4

and their tree representation:

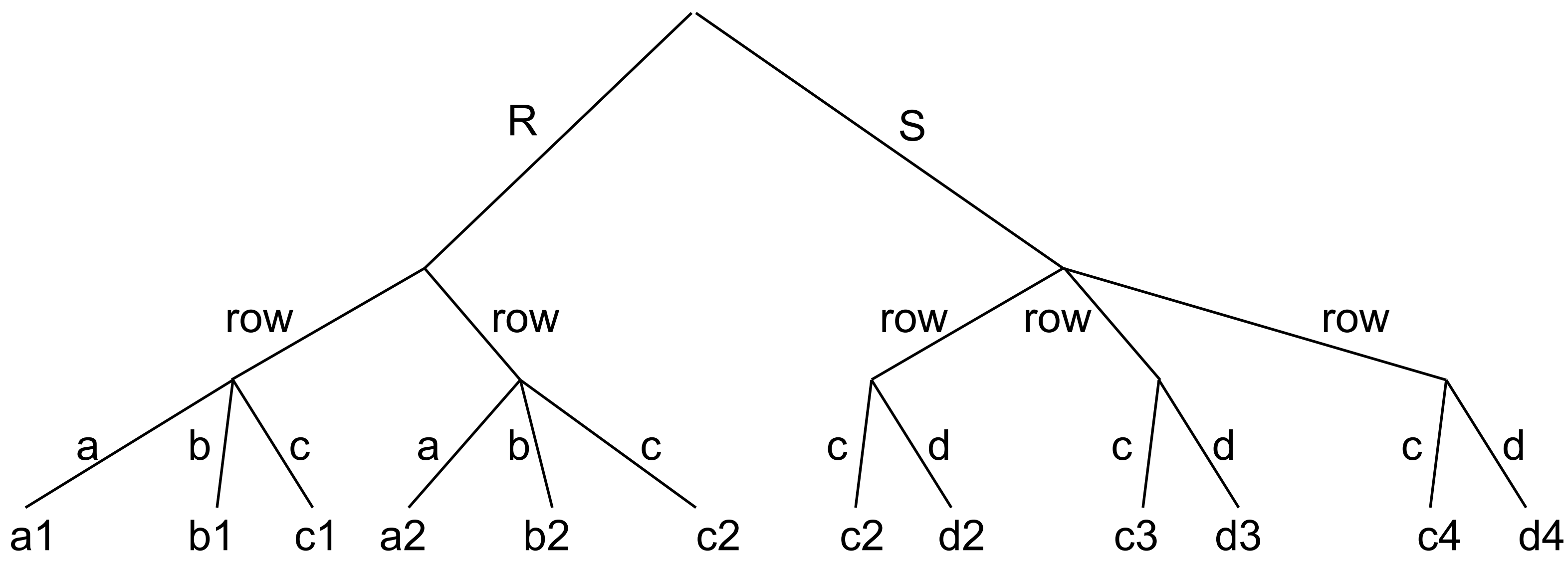
SSD: representing relational data

Consider two relations :

R	a	b	c
	a1	b1	c1
	a2	b2	c2

S	c	d
	c2	d2
	c3	d3
	c4	d4

and their tree representation:



SSD: representing relational data

Consider two relations :

R	a	b	c
	a1	b1	c1
	a2	b2	c2

S	c	d
	c2	d2
	c3	d3
	c4	d4

and their tree representation:

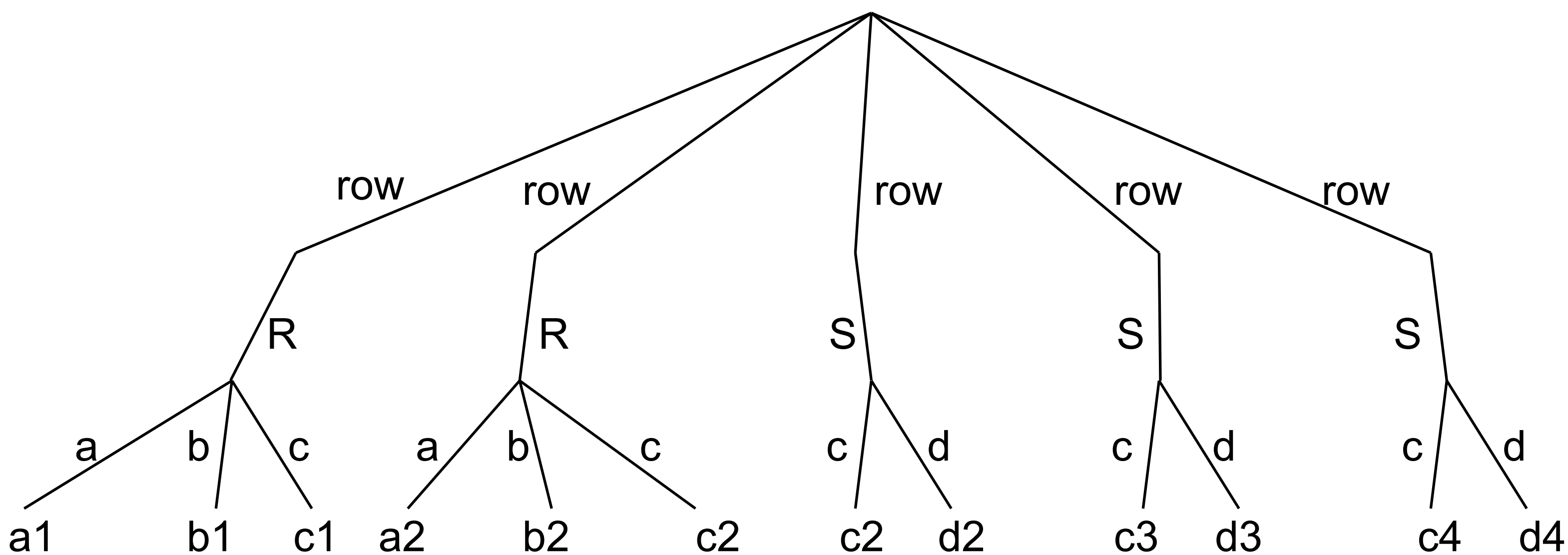
SSD: representing relational data

Consider two relations :

R	a	b	c
	a1	b1	c1
	a2	b2	c2

S	c	d
	c2	d2
	c3	d3
	c4	d4

and their tree representation:



SSD: representing relational data

Consider two relations :

R	a	b	c
	a1	b1	c1
	a2	b2	c2

S	c	d
	c2	d2
	c3	d3
	c4	d4

and their tree representation:

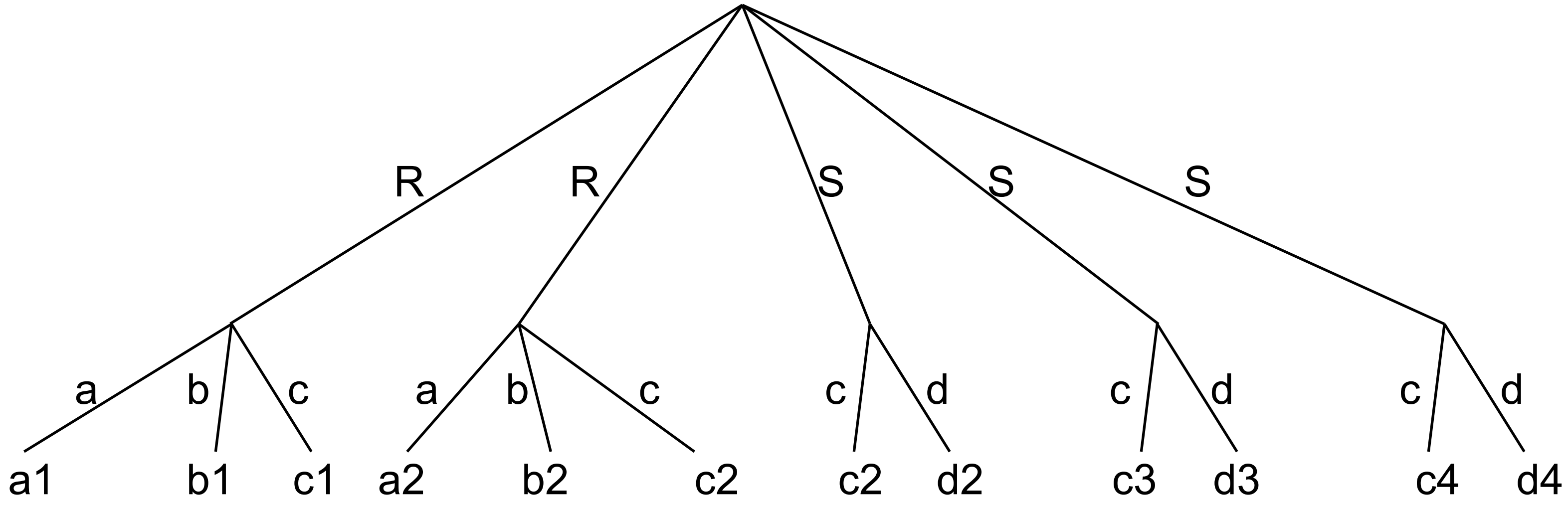
SSD: representing relational data

Consider two relations :

R	a	b	c
	a1	b1	c1
	a2	b2	c2

S	c	d
	c2	d2
	c3	d3
	c4	d4

and their tree representation:



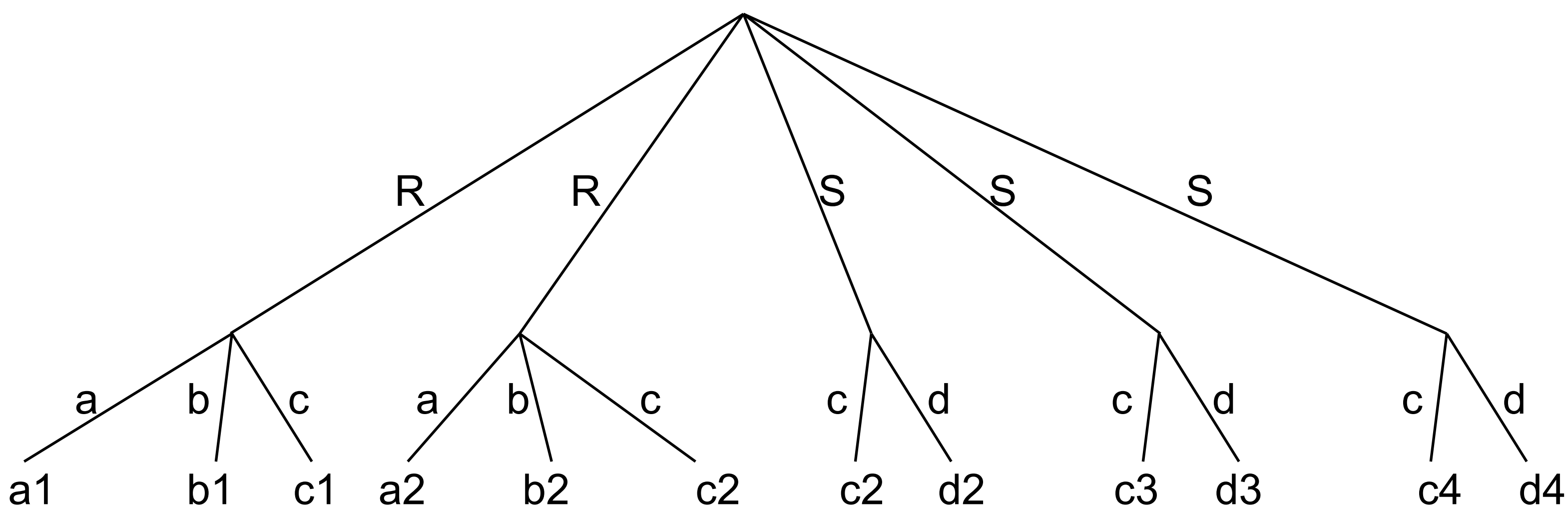
SSD: representing relational data

Consider two relations :

R	a	b	c
	a1	b1	c1
	a2	b2	c2

S	c	d
	c2	d2
	c3	d3
	c4	d4

and their tree representation:



➔ we can represent relational data, though with an overhead

SSD: representing object databases

- we can represent data from object-oriented DBMSs or SE as SSD
  - provided we have *object identifiers*, e.g., &o1
  - so that objects can refer to each other

Example: { persons: {person: &o1 { name: “John”,
age: 47,
relatives: {child: &o2,
child: &o3}}
person: &o2 { name: “Mary”,
age: 21,
relatives: {father: &o1,
sister: &o3}}
person: &o3 { name: “Paula”,
age: 23,
relatives: {father: &o1,
sister: &o2}}}}

➡ **Draw a graph representation of this piece of semi-structured data!**

MANCHESTER 1824

The University of Manchester

SSD: how to represent/store

- there are various formalisms to store semi-structured data
  - for example
    - Object Exchange Model (OEM, close to previous examples)
    - Lore
    - **XML**
    - **JSON**
- different formalisms with different
  - internal representations
  - mechanisms for self-describing
  - datatypes (e.g., integer, Boolean, string,…) supported
  - description mechanisms for (semi) structure:
    schema languages to describe
    - which fields are allowed/required where
    - which values allowed/required where
  - query languages & **manipulation** mechanisms

XML
a data model with
a tree-shaped internal representation

XML

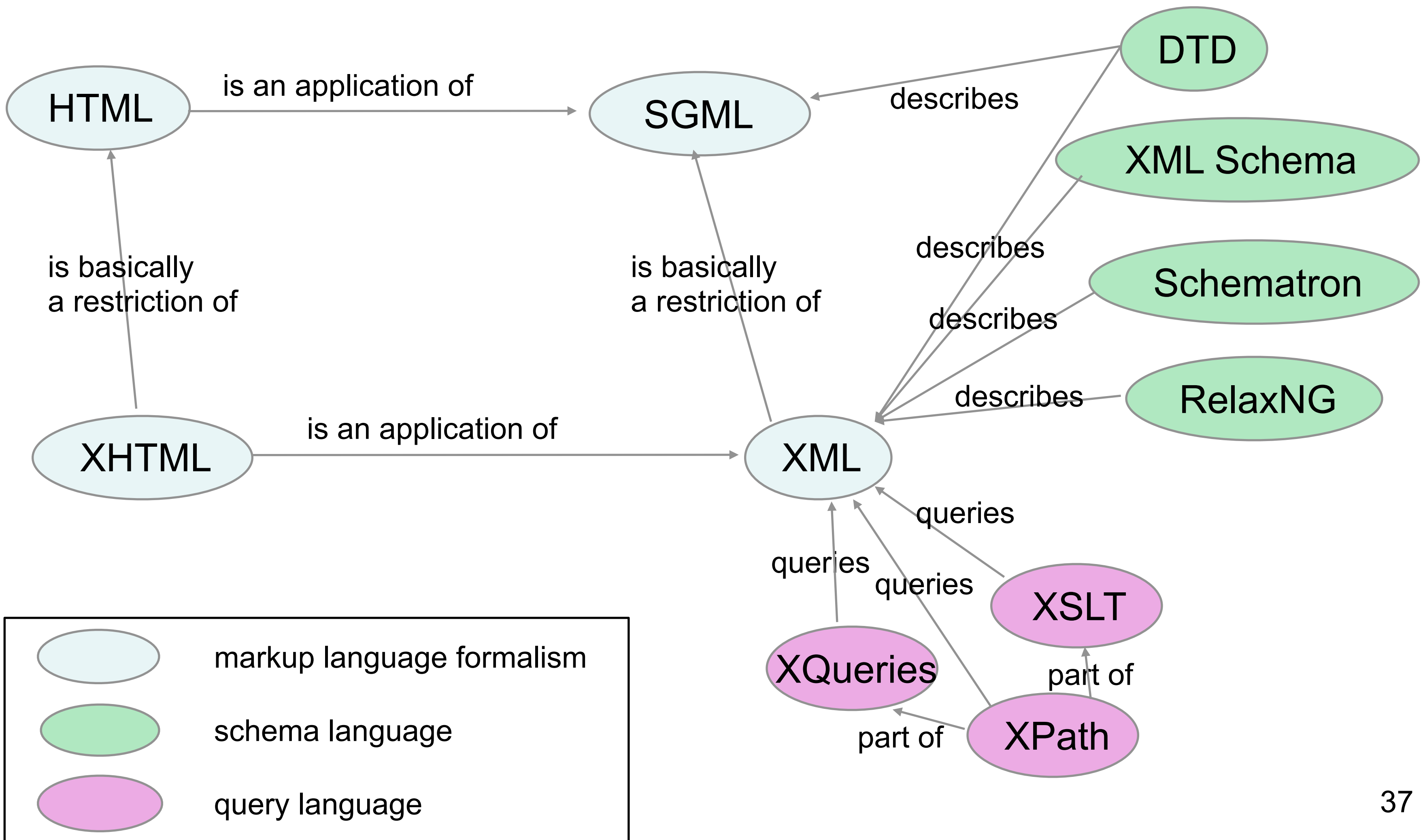
- is a formalism for the representation of *semi-structured data*
  - e.g., used by UniProt
  - suitable for humans and computers
- is *not* designed to specify the lay-out of documents
  - this what html, css and others are for
- alone will not solve the problem of **efficiently querying (web) data:** 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
- **XML**, 1998 first edition of XML 1.0 (now 4th edition)
  - 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

W3C?!

A rough map of a part of Acronym World



Back to our very simple XML example

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

XML documents are **text** documents

In badly 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

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

In 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>
```

XML documents
are
text documents

Back to our very simple XML example

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

Use an **XML editor** to work with XML documents

In better layed-out XML with syntax highlighting:

```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

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

still based on XML

In a different **format**, with better layed out XML 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

MANCHESTER 1824

The University of Manchester

An XML Example

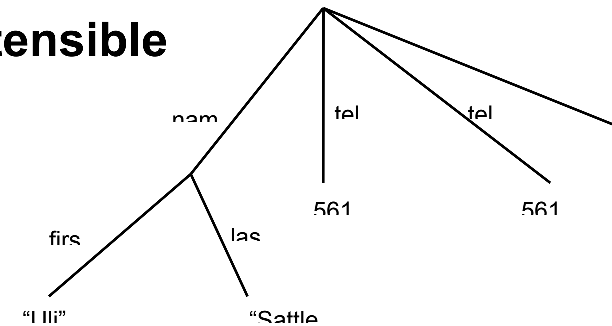
A **snippet of XML** describing the above Dilbert cartoon

```
<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?

Technical terms, when used for the first time, are marked **red**

- 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
  - can be associated with a **tree**, its **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



Example

And here is the full XML document

```
<?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 *grammar describing document/*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** -- also called **document 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 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>
```

element

MANCHESTER 1824

The University of Manchester

Example

And here is the full XML document

```
<?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>
```

Attributes

Start Tag

End Tag

XML Core Concepts: elements *(the main concept)*

<element-name attr-decl1 … attr-decln>

content

</element-name>

<cartoon copyright="United Feature”>

content

</cartoon>

- arbitrary number of attributes is allowed
- each *attr-decli* 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

simple content
mixed content
element content

  - …those *contained* elements are the element’s **child elements**
- an empty element can be abbreviated as *<element-name attr-decl1 … attr-decln/>*

Example


```
<?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>
```

Simple content

Element content

The University of Manchester

MANCHESTER 1824

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/dtds/html.dtd”
  “http://www.abc.org/dtds/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

  <p> <p> <p> ...</p> </p> </p>

- the same **element name** can occur many times in a document,
  - e.g.,

    <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.,
)
- 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

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

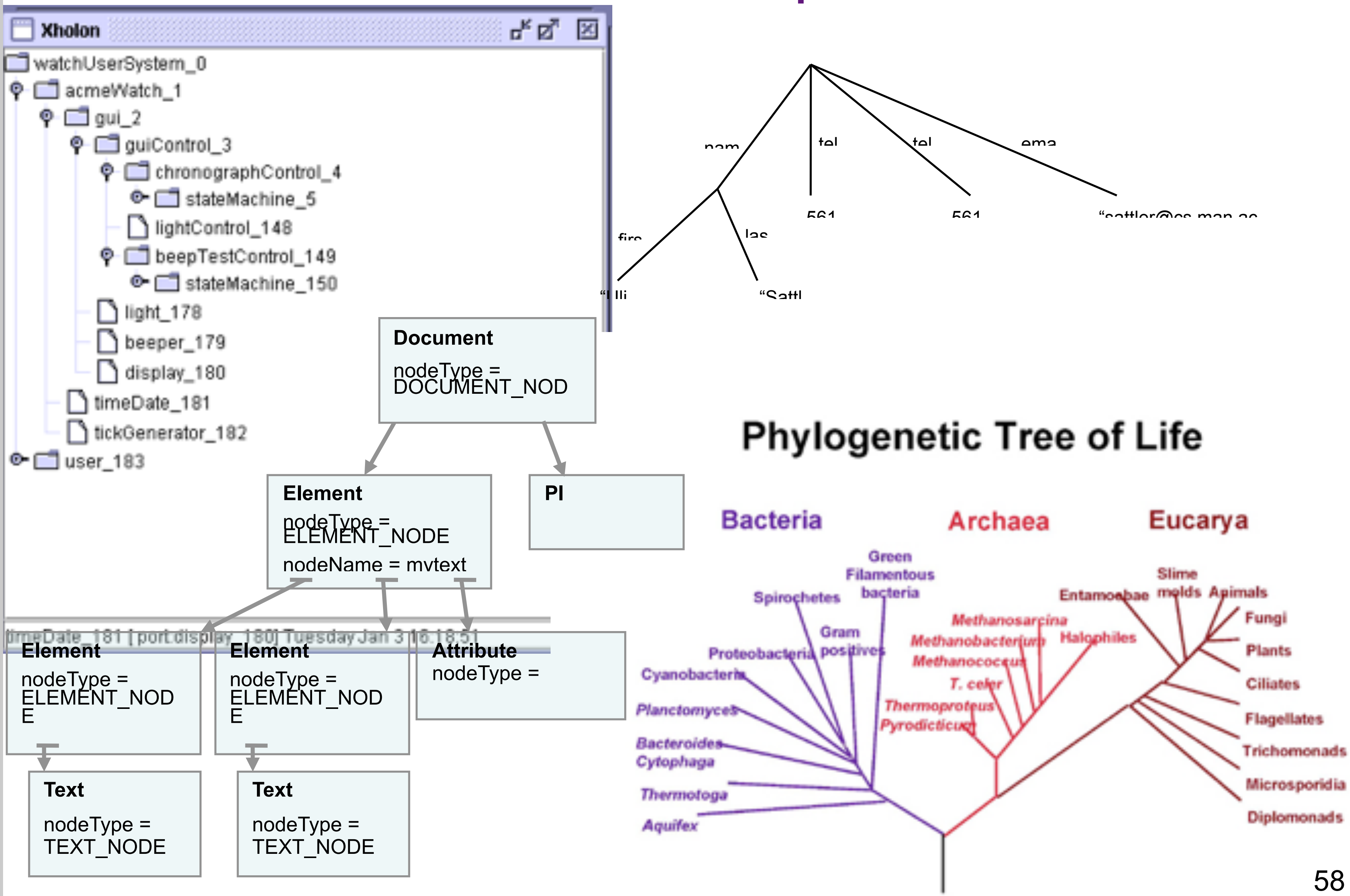
Let’s test our understanding via some Kahoot quiz: go to kahoot.it

Well-formedness is a very weak property:
basically, it only ensures that we can **parse a document into a tree**

Interlude: Trees!

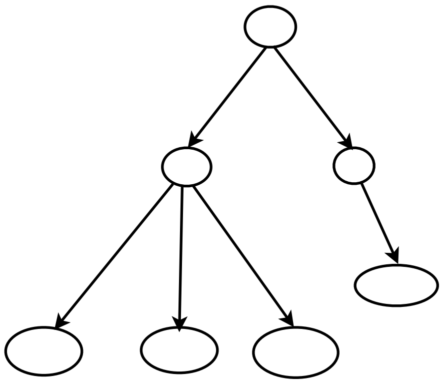
play a central role for SSD, XML,…. everything!

Trees come in different shapes!



Interlude: Abstract trees - nodes as strings!

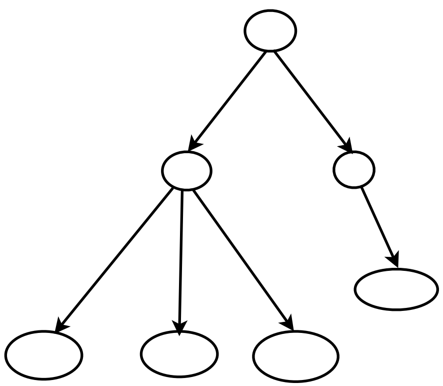
A tree

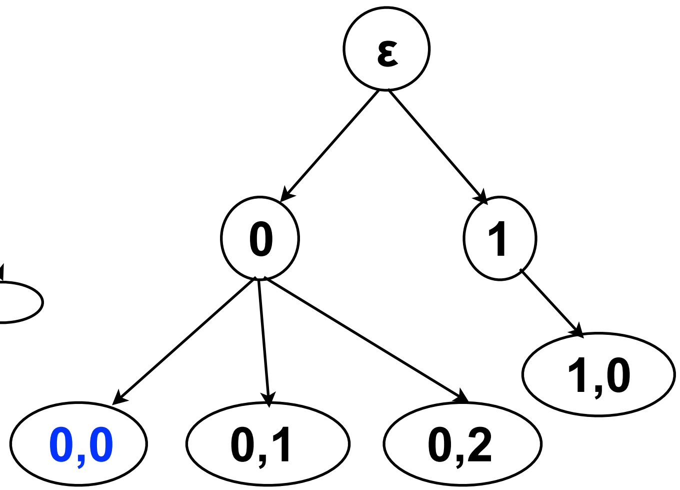


Interlude: Abstract trees - nodes as strings!

A tree

A tree
with strings
as node **names**

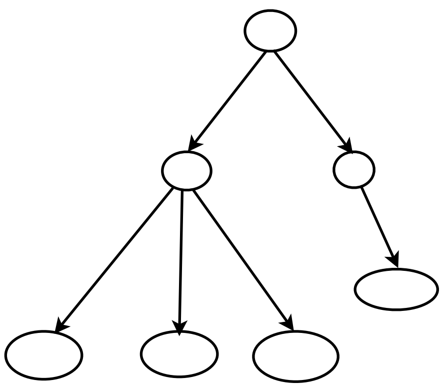




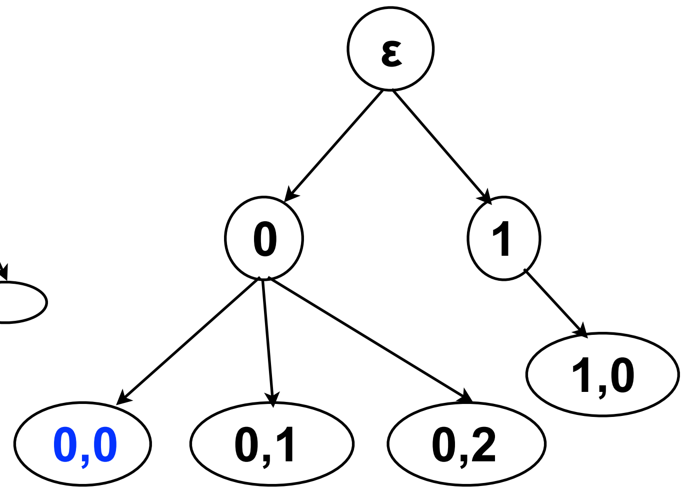
- so we can refer to nodes by names
- order matters!
    - the node 0,0 is different from 0,1

Interlude: Abstract trees - nodes as strings!

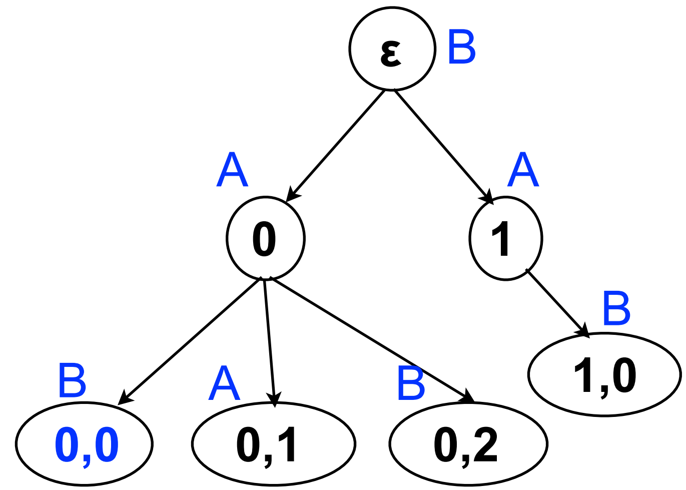
A tree



A tree
with strings
as node **names**



A labelled tree over
{A,B,C} (as node **labels**)



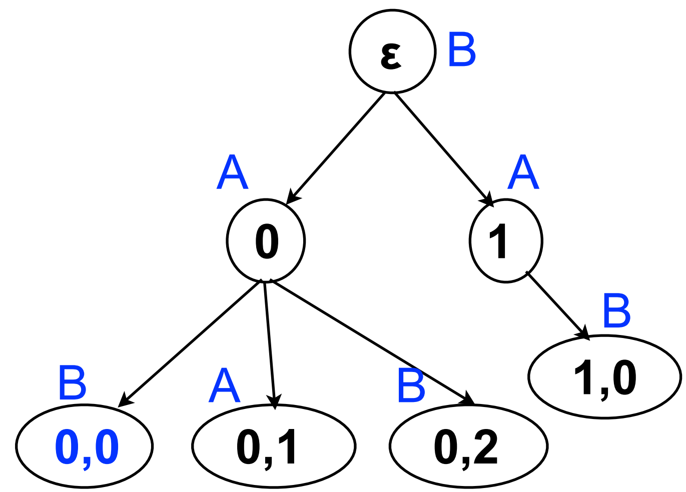
- so we can refer to nodes by names
- order matters!
  - the node 0,0 is different from 0,1

- so we can distinguish
  - a node from
  - a node's label

Interlude: Abstract trees - nodes as strings!

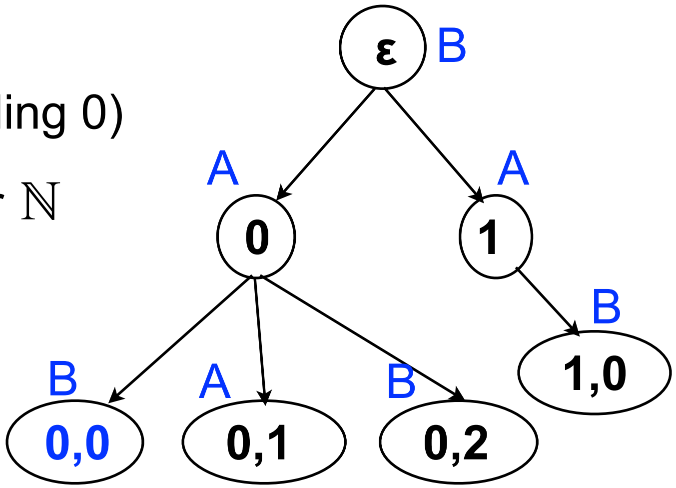
The tree T as a function:
T(ε) = B
T(0) = A
T(1) = A
T(0,0) = B
T(0,1) = A
….

A labelled T tree over {A,B,C} (as node **labels**)



- so we can distinguish
  - a node from
  - a node’s label

Interlude: Abstract trees - nodes as strings!



- We use $\mathbb{N}$ for the non-negative integers (including 0)
- we use $\mathbb{N}^*$ for the set of all (finite) strings over $\mathbb{N}$
  - ε is used for the empty string
  - 0,1,0 is a string of length 3
  - each string stands for a node
- An **alphabet** is a finite set of symbols
- A **tree T over an alphabet Σ** is a mapping T: $\mathbb{N}^* \rightarrow \Sigma$ whose **domain** is
  - **finite**, i.e., T(n) is defined for **only finitely many** strings over $\mathbb{N}$

    ⇒ each tree has only finitely many nodes
  - **contains ε** , i.e., T(**ε**) is defined

    ⇒ each tree has a root **ε**
  - is **prefixed-closed**, i.e., if T(w,n) is defined, then T(w) is as well

    ⇒ the predecessor w of a node (w,n) is in T

Interlude: Abstract trees - nodes as strings!

- Explanation:
  - the **strings** in the domain of T represent T's nodes
  - (w,n) is the successor of w,
  - T(w) is the label of w (as shown in picture)
  - we use nodes(T) for the (finite) domain of/nodes in T

- **Is the following mapping T a tree? If yes, draw the tree T!**

  $\Sigma = \{W, X, Y, Z\}$
  $T(\boldsymbol{\varepsilon}) = X$
  $T(0) = X$
  $T(1) = X$
  $T(2) = X$
  $T(3) = Z$
  $T(0,0) = Y$
  $T(0,0,0) = Y$
  $T(3,1) = Z$

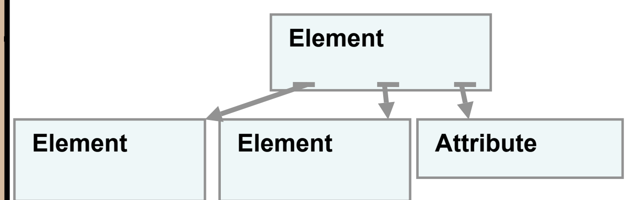
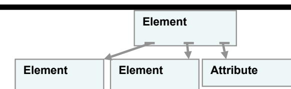
Well-formedness is a very weak property: basically, it only ensures that we can parse a document into a tree…

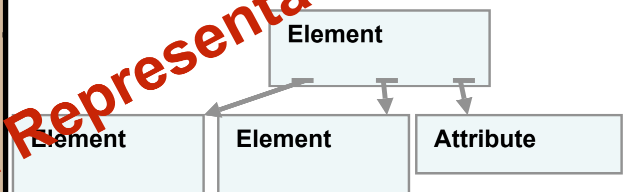
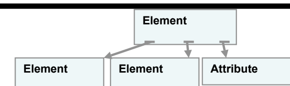
the document's
internal representation

An **Internal Representation** for XML documents

```
<?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<charact
er> </characters></prolog><panels>
      ....</panels></cartoon>
```

- 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 a **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**

Level		Data unit examples	Information or Property required	
cognitive				
application				
tree adorned with... namespace	tree adorned with... 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		

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

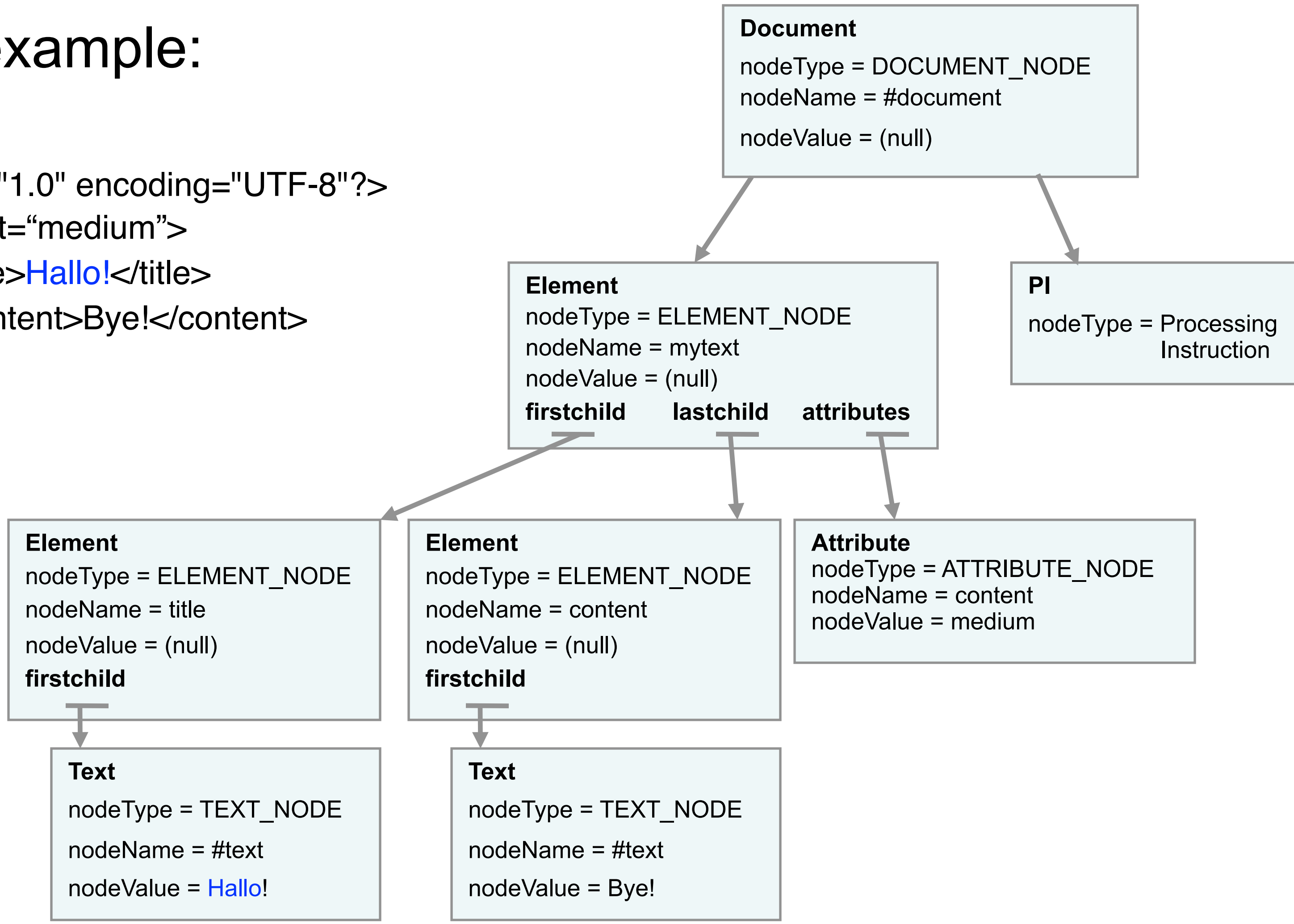
External Representation

DOM!

DOM trees as an InR for XML documents

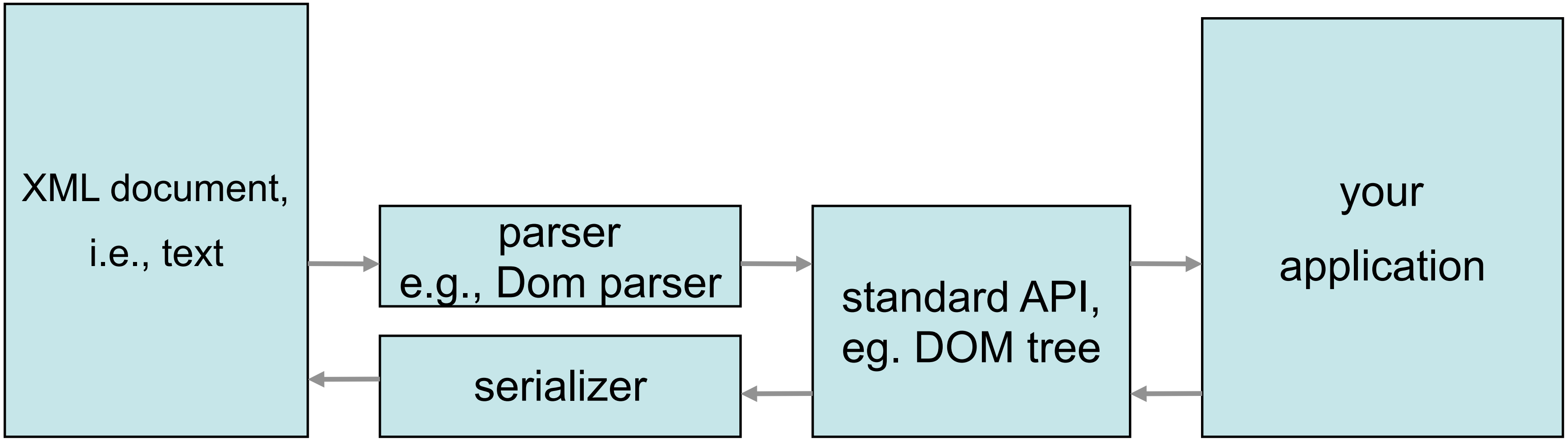
A simple example:

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



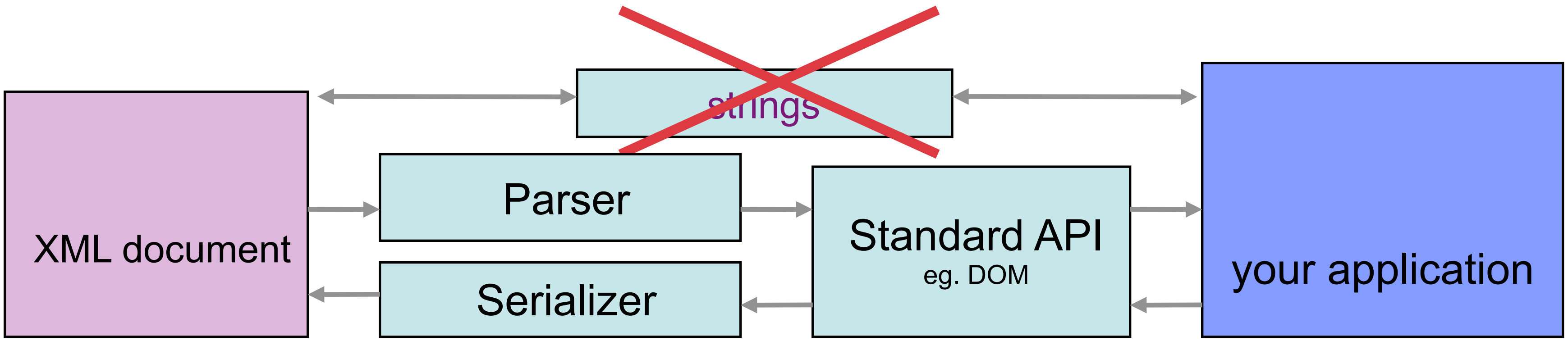
DOM: InR for XML documents

- 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

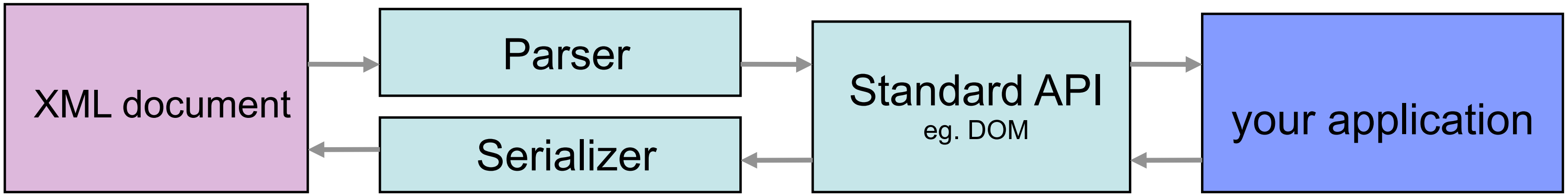


Programmatic Manipulation of XML Documents

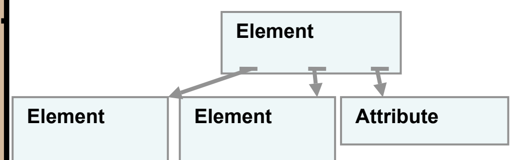
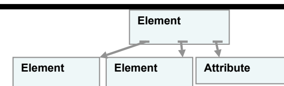
As a rule, whenever we manipulate XML documents in an application, we should use standard APIs:

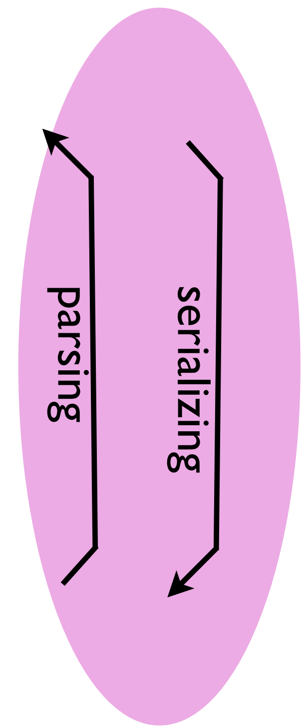


Parsing & Serializing XML documents



- **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

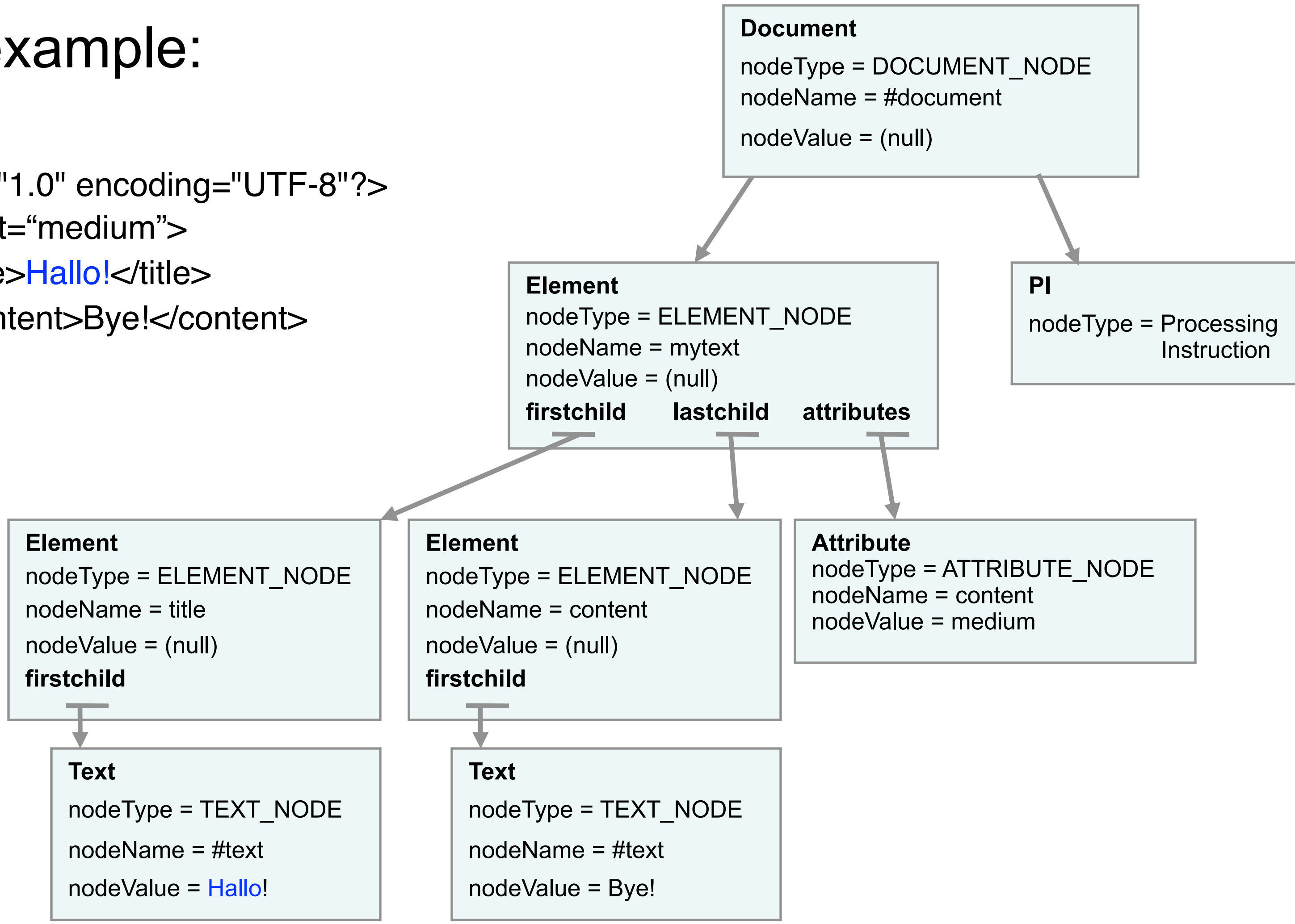
Level		Data unit examples	Information or Property required
cognitive			
application			
tree adorned with... namespace	schema		nothing | a schema
tree			well-formedness
token	complex	<foo:Name t=”8”>Bob	
token	simple	<foo:Name t=”8”>Bob	
character		< foo:Name t=”8”>Bob	which encoding (e.g., UTF-8)
bit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



DOM trees as an InR for XML documents

A simple example:

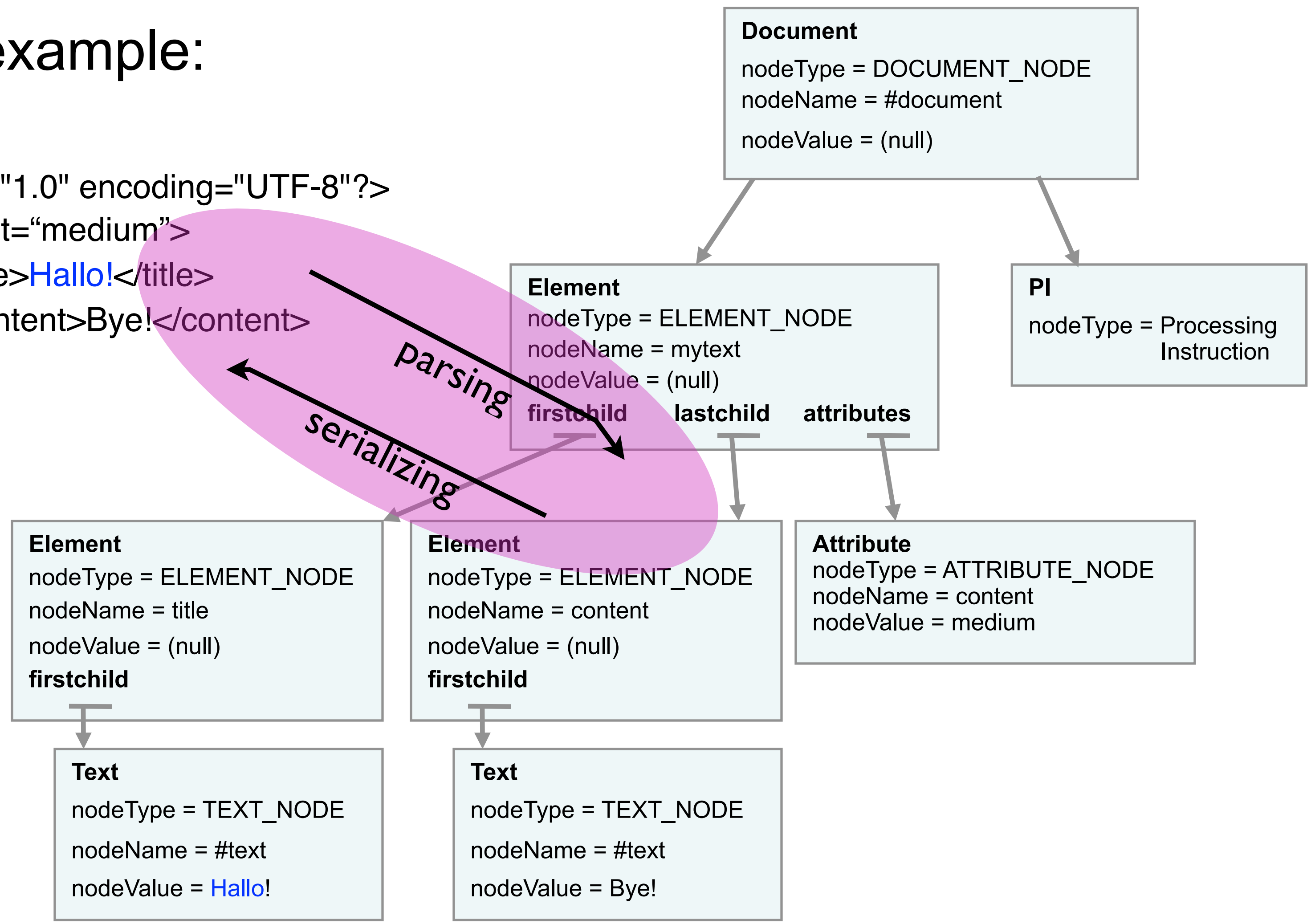
```
<?xml version="1.0" encoding="UTF-8"?>
<mytext content="medium">
            <title>Hallo!</title>
            <content>Bye!</content>
</mytext>
```



DOM trees as an InR for XML documents

A simple example:

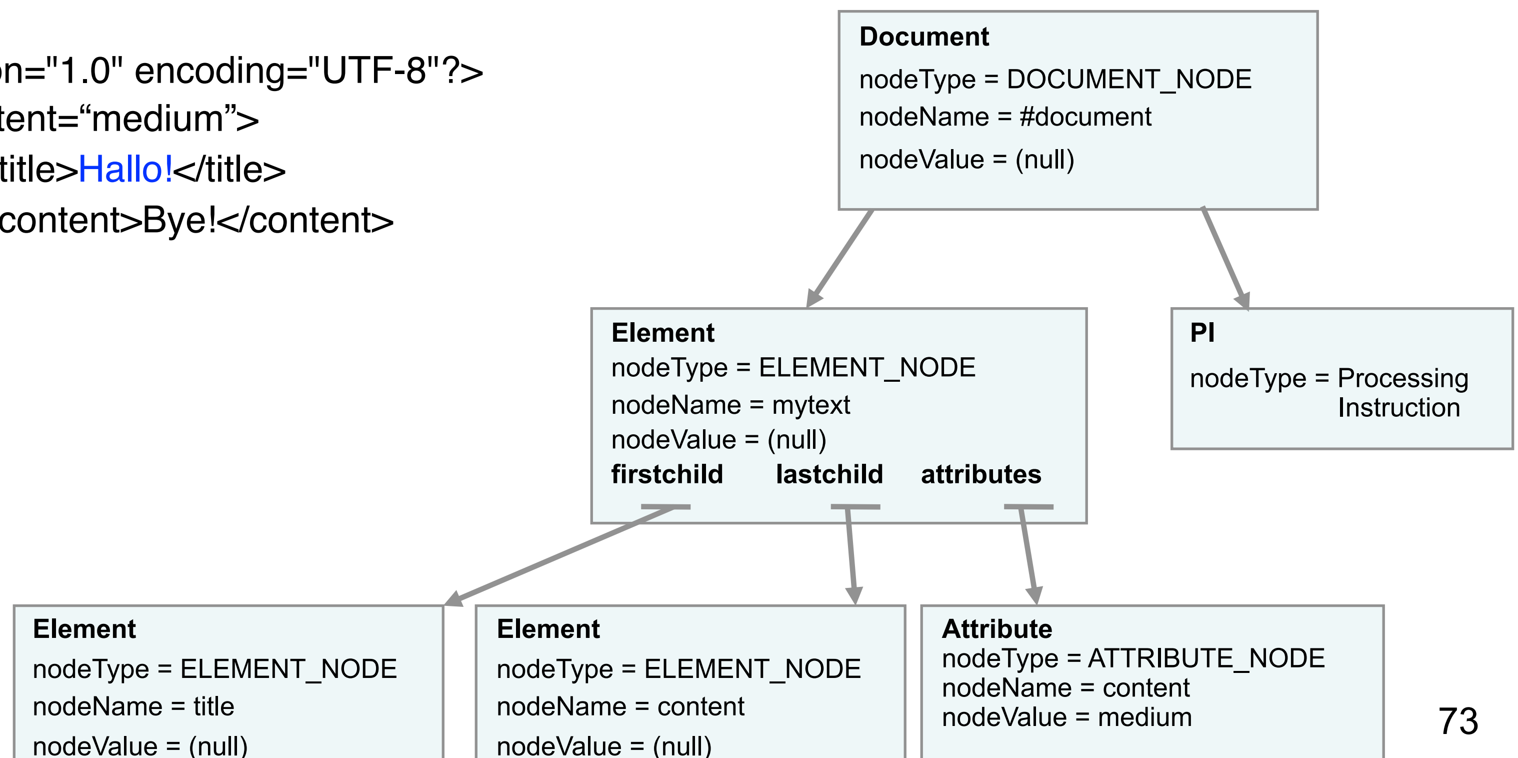
```xml
<?xml version="1.0" encoding="UTF-8"?>
<mytext content="medium">
        <title>Hallo!</title>
        <content>Bye!</content>
</mytext>
```



DOM trees as an InR for XML documents

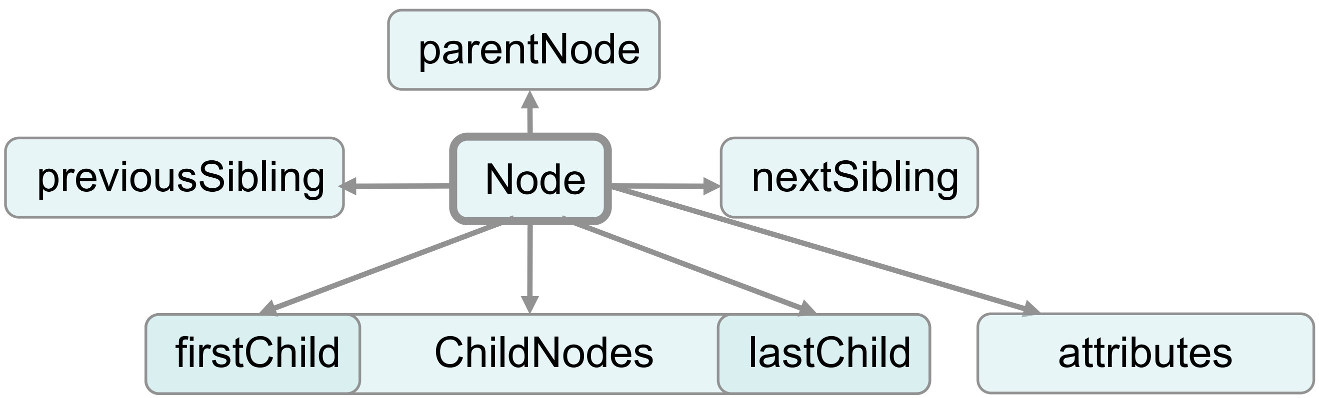
- In general, we have the following correspondence:
  - XML document D → tree t(D)
  - element e in D → node t(e) in t(D)
  - empty element → leaf node
  - root element e in D → **not** root node in t(D), but document node

```
<?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 → tree t(D)
  - element e in D → node t(e) in t(D)
  - empty element → leaf node
  - root element e in D → **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:



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

DOM by example

mydocument.xml:

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

A little Java example:

"if 1st child of **mytext**s is "**Hallo**" return the content of 2nd child"

1. let a parser build the DOM of mydocument.xml

```
factory = DocumentBuilderFactory.newInstance();
myParser = factory.newDocumentBuilder();
parseTree = myParser.parse(”mydocument.xml");
```

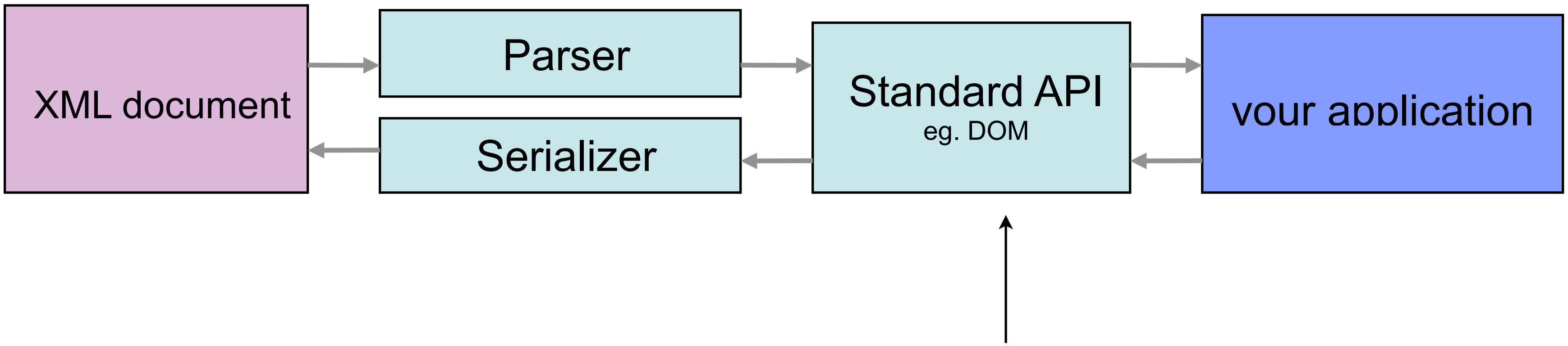
2. Retrieve all "mytext" nodes into a NodeList interface:

```
mytextNodes = parseTree.getElementsByTagName("mytext")
```

3. Navigate and retrieve all contents:

```
for (int i=0; i < mytextNodes.getLength(); i++) {
        actmytextNode = mytextNodes.item(i);
        acttitleNode = actmytextNode.getFirstChild();
        actstring = acttitleNode.getFirstChild().getNodeValue();
        if (actstring.equals("Hallo")) {
                actcontentNode = acttitleNode.getNextSibling();
                returnstring = actcontentNode.getFirstChild().getNodeValue();
        break; } }
```

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...

MANCHESTER 1824

The University of Manchester

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 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?

Bijan,	Parsia,	2.32
Uli,	Sattler,	2.24

```
<csvFile>
   <record>
      <field>Bijan</field>
      <field>Parsia</field>
      <field>2.32</field>
   </record>
   <record>
      <field>Uli</field>
      <field>Sattler</field>
      <field>2.21</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?

```
Name,Surname,Room
Bijan, Parsia, 2.32
Uli,     Sattler, 2.24
```

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

```xml
<csvFile>
   <record>
      <name>Bijan</name>
      <surname>Parsia</surname>
      <room>2.32</room>
   </record>
   <record>
      <record>Uli</name>
      <surname>Sattler</surname>
      <room>2.21</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

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

```
Name,Surname,Room
Bijan,Parsia,2.32
Uli,Sattler,2.24
```

```
<address>
    <name>Bijan</name>
    <surname>Parsia</surname>
    <room>2.32</room>
  </address>
```

- 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?
    - room number ⇒ string
    - height ⇒ number

- Are CSV, ExCSV, XML self-describing?

Self-describing

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

```
Name, Surname, Room
Bijan, Parsia, 2.32
Uli,     Sattler, 2.24
```

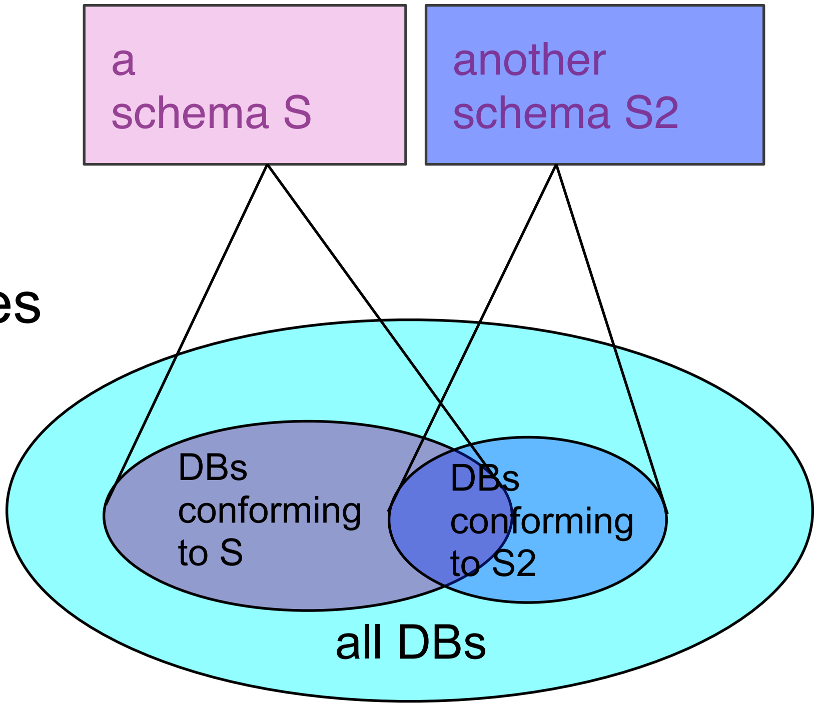
  - what suitable data structure can we extract?
    - CSV, ExCSV: tables, flat records, arrays, lists, etc.
    - XML: labelled, ordered trees of (unbounded) depth!
- Clearly, you could parse *specific* CSV files into trees,
  but you'd need to use *extra-CSV* rules/information for that
- ...in this sense, XML can be said to be more self-describing than ExCSV
  still need to know whether "2.32" is a string or a number?

Schemas!

Schemas: what are they?

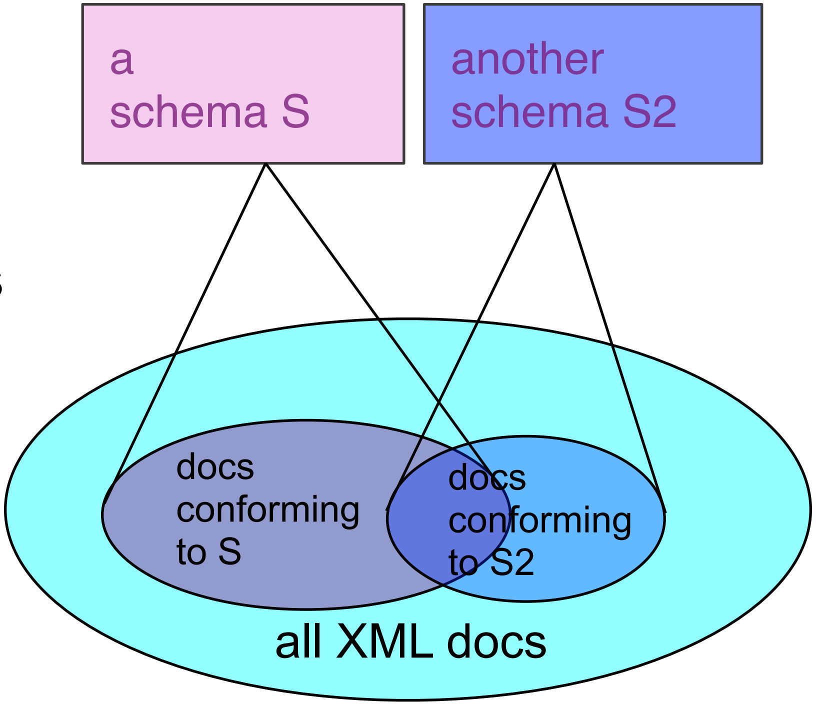
A **schema** is a description

- of **DBs**: describes
  - tables,
  - their names and their attributes
  - keys, keyrefs
  - integrity constraints



Schemas: what are they?

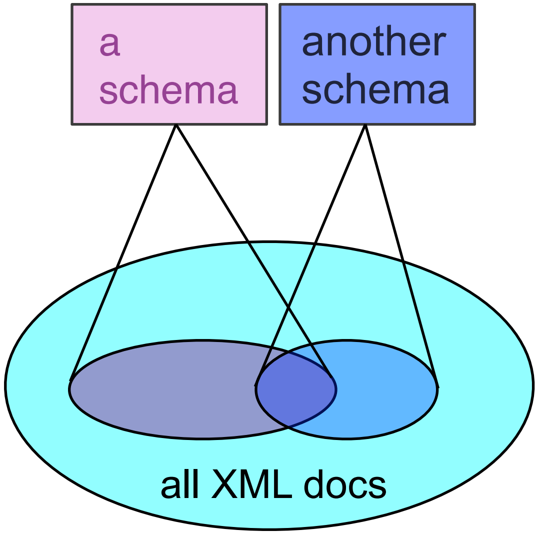
A **schema** is a description



- 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
  - 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-formedness 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 a **format** with
  - 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**

```
<addresses>
   <name>
      <address>Bijan</address>
      <surname>Parsia</surname>
      <room>2.32</room>
   </name>
   <room>
      <room><room>
             Uli</room> </room>
      <room>Sattler</room>
      <room>2.21</room>
   </room>
</addresses>
```

```
<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>
```

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

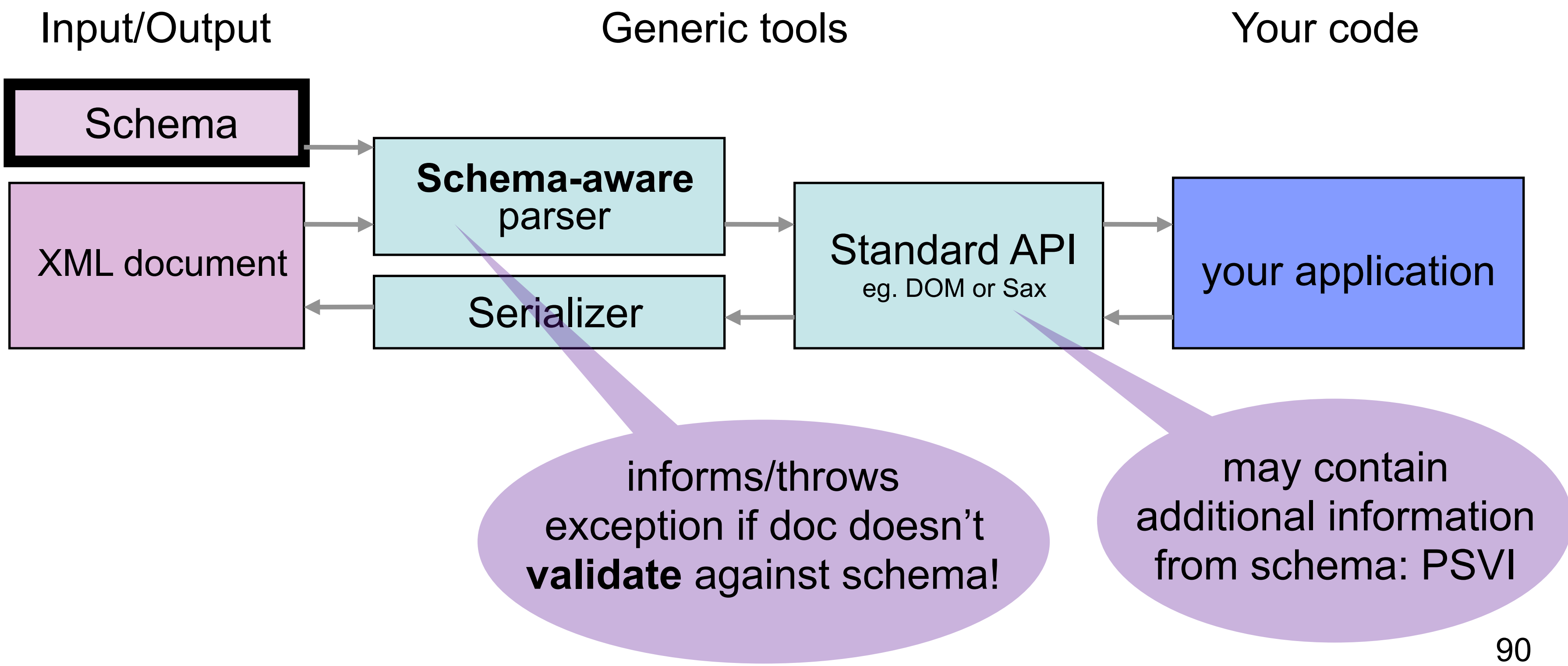
- **Specification**
  - you document/describe/publish your format
  - so that it can be used across multiple implementations
- As **input** for applications
  - applications can do **error-checking** in a **format independent** way
    - checking whether an XML document conforms to a schema can be done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

The University of Manchester

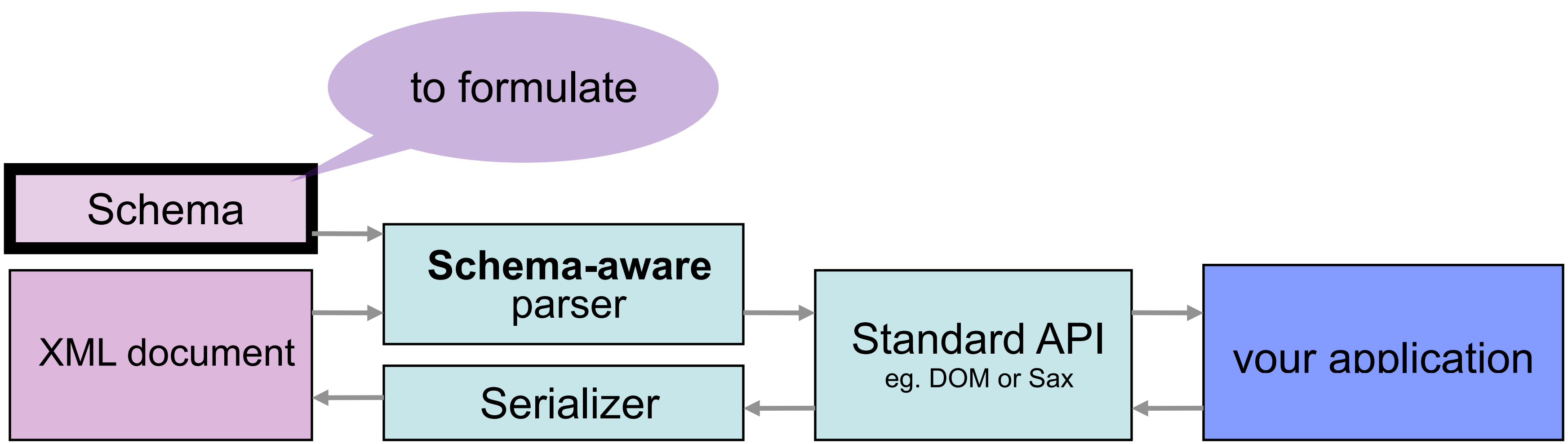
MANCHESTER 1824

Benefits of an (XML) schema

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



RelaxNG,
a very powerful schema language



RelaxNG: a schema language

- RelaxNG was designed to be a **simpler** schema language
- (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 attributes
  - no entity declarations
  - no key/uniqueness constraints
  - minimal datatypes: only “token” and “string” (like DTDs) (but a mechanism to use XSD datatypes)

in case you know about these

- since it is so simple/flexible
  - it’s (claimed/designed 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: another 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

later!

...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

Next Week

- 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:

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

To describe documents like:

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

```
<?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, pfew! (and also into/from other schema languages)

✓Trang can be used from Oxygen

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

```
<grammar
  xmlns="http:...”
  xmlns:a="http:.."
  datatypeLibrary="http:...>
    <start>
       <element name="name">
          <element name="first"><text/></element>
          <element name="last"><text/></element>
       </element>
    </start>
</grammar>
```

RelaxNG - to describe structure:

- 3 kinds of **patterns**, for the 3 "central" nodes:
  - text

```
text
```

  - attribute

```
attribute age { text },
attribute type { text },
```

  - 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

MANCHESTER 1824

The University of Manchester

RelaxNG: ordered groups

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

```
<?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>
   <person>....
</people>
```

```
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’:

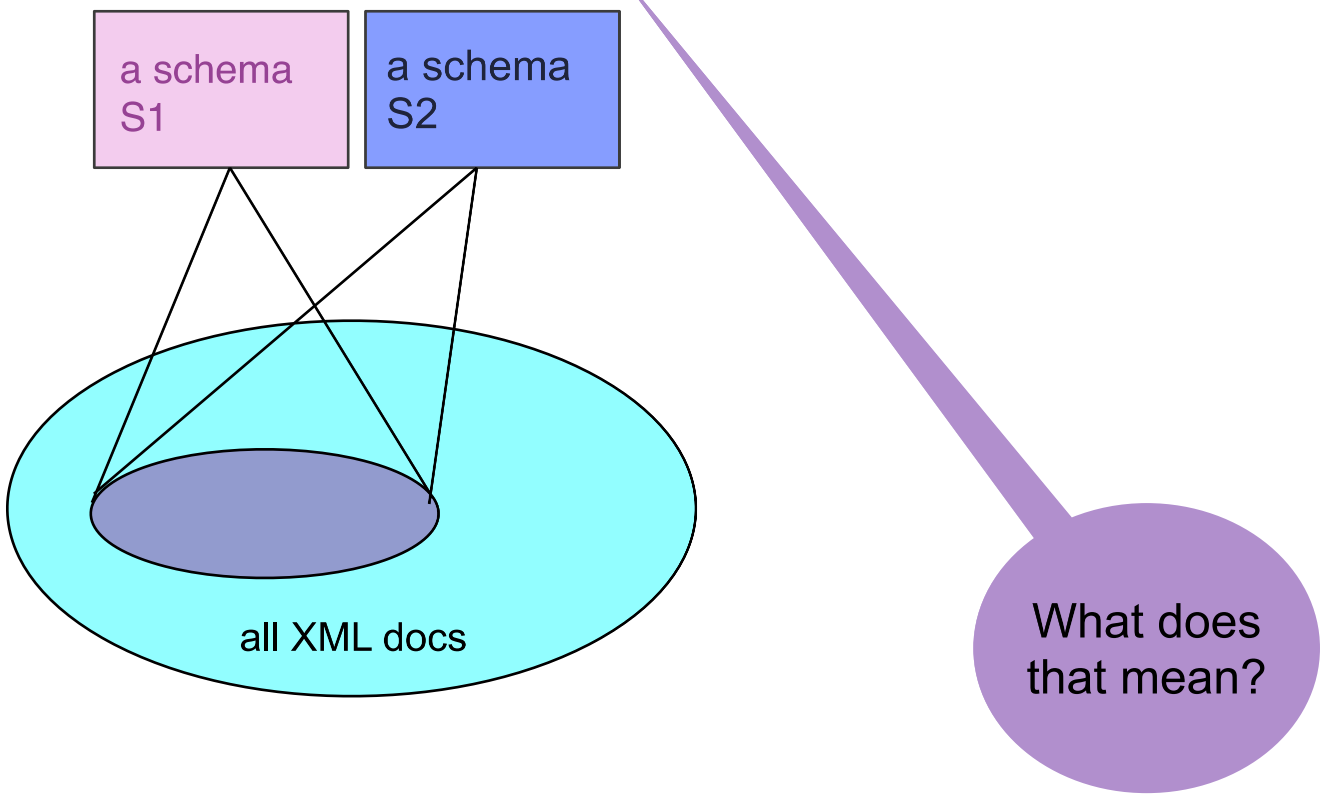
```
grammar { start =  people-description

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

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

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

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

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

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.

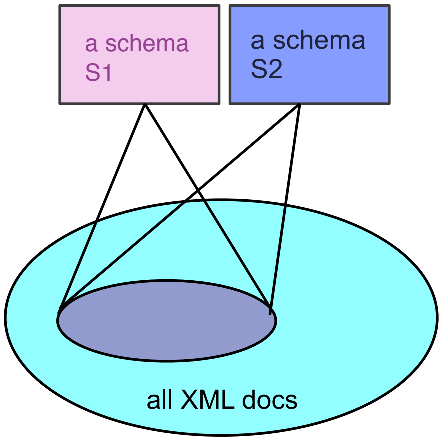
Documents being **valid** wrt schema

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



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

Simple RelaxNG schemas

- Is this schema simple?

```
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  }}
```

- 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?

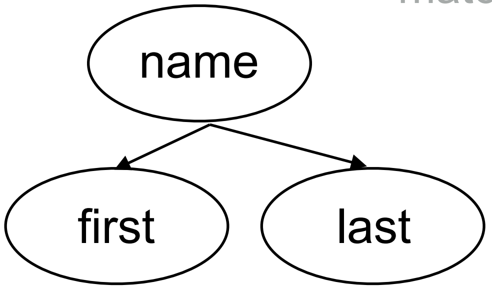
```
grammar { start =  people-description

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  }}
```

- 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”

Documents described by a RelaxNG schema

- An 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 }

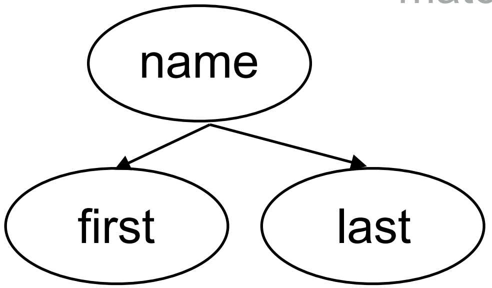


- An XML document *D* ***is valid wrt*** a simple RelaxNG schema *S* if
  - *D*'s root node name is *X* iff *S* contains start = *X*-description
  - each node n in D matches its description,
    I.e., if *D*'s name is *X*, then *S* contains a statement *X*-description = *Y* and *n* matches *Y*.

Documents described by a RelaxNG schema

- An 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 }

- An XML document *D* ***is valid wrt*** a simple RelaxNG schema *S* if
  - *D*'s root node name is *X* iff *S* contains start = *X*-description
  - each node n in D matches its description,
    I.e., if *D*'s name is *X*, then *S* contains a statement *X*-description = *Y* and *n* matches *Y*.

Interlude:
Regular Expressions

Regular Expressions

- a standard concept to describe expressions
- allows us to describe/understand which documents are described here:

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

test-content = (A-content, B-content, C-content)
A-content = element A {text}
B-content = element B {text}
C-content = element C {text}}
```

and here:

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

test-content = (A-content+, B-content?,C-content*)+,(B-content | C-content*)+

A-content = element A {text}
B-content = element B {text}
C-content = element C {text}}
```

Regular expressions

- Given a set of symbols N, the set of **regular expressions** regexp(N) over N is the smallest set containing
  - the empty string ε and all symbols in N and
  - if e1 and e2 ∈ regexp(N), then so are
    - e1,e2 (concatenation)
    - e1|e2 (choice)
    - e1* (repetition)

- Given a regular expression e, a string w **matches** e,
  - if w = ε = e or w = n = e for some n in N, or
  - if w = w1 w2 and e = (e1 , e2) and w1 matches e1 and w2 matches e2 , or
  - if e = (e1 | e2) and w matches e1 or w matches e2
  - if w = ε and e = e1*
  - if w = w1 w2... wn and e = e1* and each wi matches e1

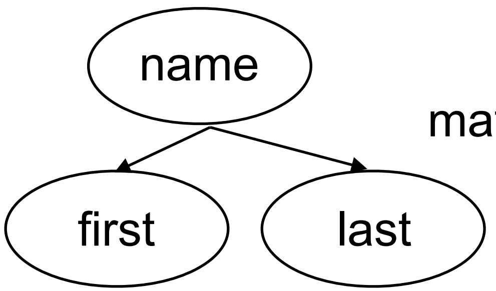
Regular expressions

- Hence we can use
  - e+ as abbreviation for (e,e*)
  - e? as abbreviation for (e|ε)

Let's test our understanding via some Kahoot quiz: go to kahoot.it

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 }

- An XML document *D* ***is valid wrt*** a simple RelaxNG schema *S* if
  - – *D*'s root node name is *X* iff *S* contains 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

Which of these is these is valid wrt

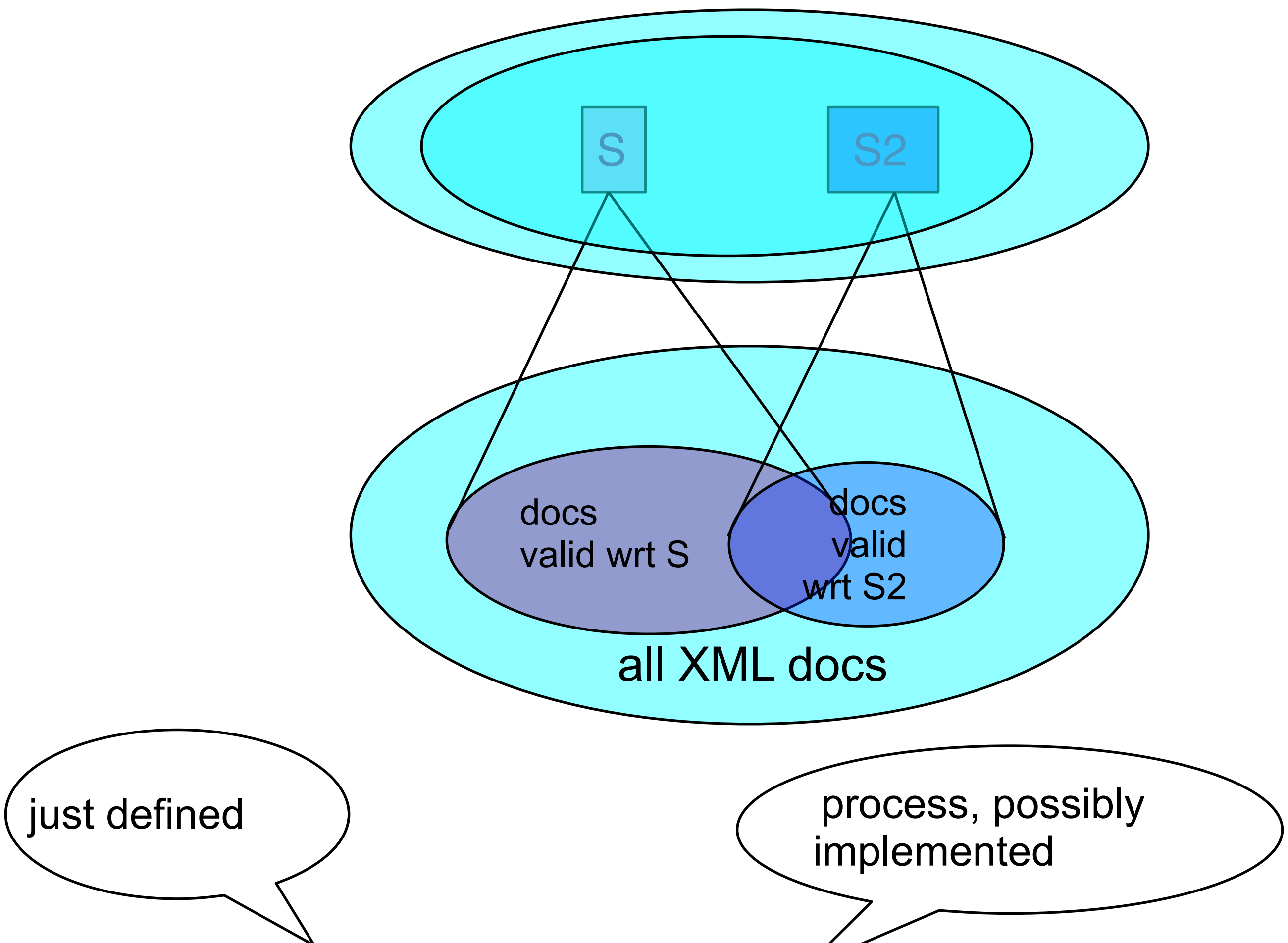
```
<?xml version="1.0" encoding="UTF-8"?>
 <name>
         <first>Harry</first>
         <last>Potter</last>
</name>
```

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

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

```
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

```
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  }}
```

```
<?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"/>
      <ref name="name-element"/>
      <oneOrMore>
        <ref name="address-element"/>
      </oneOrMore>
      <zeroOrMore>
        <ref name="project-element"/>
      </zeroOrMore>
    </element>
  </define>

  <define name="name-element">
    <element name="name">
      <element name="first">
        <text/>
      </element>
      <optional>
        <element name="middle">
          <text/>
        </element>
      </optional>
      <element name="last">
        <text/>
      </element>
    </element>
  </define>

  <define name="address-element">
    <element name="address">
      <text/>
    </element>
  </define>

  <define name="project-element">
    <element name="project">
      <attribute name="type"/>
      <attribute name="id"/>
      <text/>
    </element>
  </define>
</grammar>
```

RelaxNG: ordered groups

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

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

person-content = HR-stuff,
                 contact-stuff

HR-stuff = attribute age { text },
           project-content

contact-stuff = attribute phone { text },
                element name {name-content},
                element address { text }

name-content =  element first { text },
                element middle { text }?,
                element last { text }
project-content = element project {
                attribute type { text },
                    attribute id {text},
                    text  }+}
```

```
<?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>
   <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):

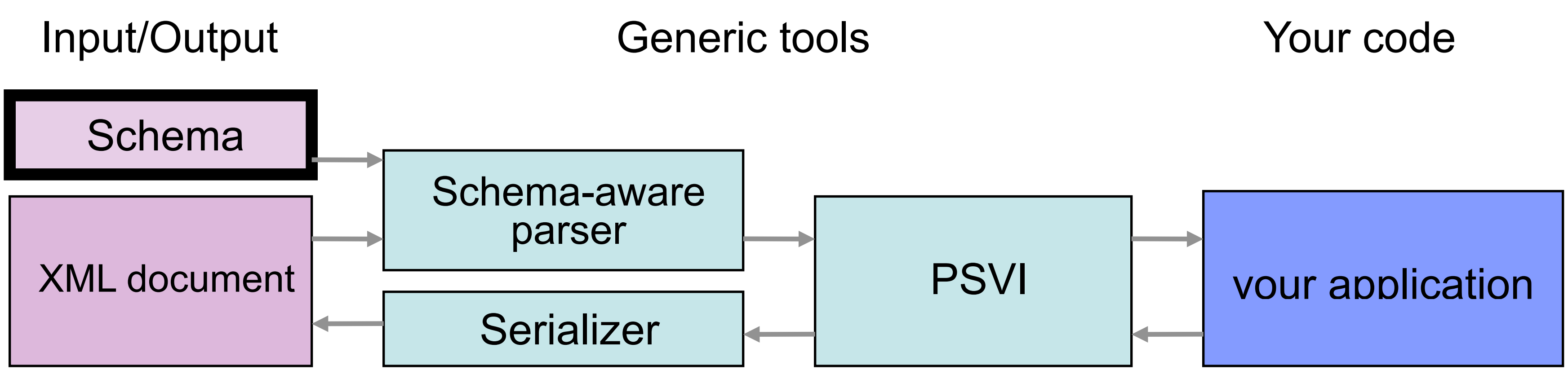
```
element person {
        attribute age { text},
        attribute phone { text},
        name-element ,
        address-element+ ,
        project-element*}
```

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 CW1),
    - no need to be changed when schema changes
    - automatically!



Validity of XML documents w.r.t. RelaxNG

- Try <oXygen/>
  - for your coursework
  - to write XML documents and RelaxNG schemas
  - it automatically checks
    - whether your document is well-formed and
    - whether your document conforms to your schema!

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) and
  in regular expressions:

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

for 4+5

```
<plus>
   <choice>
       <star>A</star>
       <star>B </star>
   </choice>
</plus>
```

for (A*|B*)+

- We have an **element name conflict**!
- How do we distinguish plus[arithmetic] and plus[reg-exp]?
  - semantically?
  - in a combined document?

Uniquing 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!

Uniquing the names (2)

- We can use a separator or other convention

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

```
<regex:plus>
   <choice>
     ...
   </choice>
</regex:plus>
```

- No name clash now
  - The “meaningful” part of the name is **clear**
  - The disambiguator is **clear**
    - But we can get **clashes**!
    - Need a **registry** to coordinate?

MANCHESTER 1824

The University of Manchester

Uniquing the names (3)

- Use URIs for disambiguation

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

```
<http://bjp.org/regex/:plus>
    <choice>
      ...
    </choice>
</http://bjp.org/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 not well formed!

Uniquing the names (4)

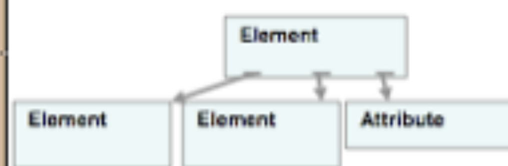
- Combine the (2) and (3)!

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

```
<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!

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		

parsing

serializing

Anatomy & Terminology of Namespaces

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

- **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

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

```
<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?
  - Default namespaces and attributes interact weirdly...

We don’t need a prefix

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

```
<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?

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

  - Default namespaces and attributes interact weirdly...

We don't need a prefix

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

```
<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?

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

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

Let’s test our understanding...

```xml
<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: DOM (and Java)

General concepts:

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

Next: Coursework Old & New

- Review of Week 1 coursework
  - in particular your conceptual model M1

- Quiz
- Short essay
- M2: extend a RelaxNG schema
  - use <OxyGen> for this or some other tools
  - test your schema, share tests
- CW2:
  - use DOM to parse XML document with arithmetic expression, compute value of arithmetic expression after validating it against RelaxNG schema
  - test your program, share tests