Informatics 1: Data & Analysis

Lecture 10: Structuring XML

Ian Stark

School of Informatics
The University of Edinburgh

Friday 15 February 2013
Semester 2 Week 5

Lecture and Tutorial Timing

This is Inf1-DA Lecture 10, in Week 5.

Next week is Innovative Learning Week. All lectures, tutorials, labs and coursework are suspended for the week, and replaced by a series of alternative events organised by different Schools and the University.

After that, starting Monday 25 February, is Week 6. Your next Inf1-DA tutorial is on Monday, Tuesday or Wednesday that week.

Inf1-DA Lecture 11 is on Tuesday 26 February.

There is no Inf1-DA lecture on the following Friday, 1 March.

Inf1-DA Lecture 12 is on Tuesday 5 March. Normal service resumes.

Innovative Learning Week

- Smart Data Hackathon http://data.inf.ed.ac.uk/ilwhack/
- Mobile Apps with SkyScanner
- NonFiSci: Fixing Bad Science on the Big Screen
- Hadoop Hackathon http://events.inf.ed.ac.uk/ilw/hadoop/
- Robotics and Decision Making
- Dare to be Fair? Unconscious bias in how we interact with others.
- UG4 Student Project test lab
- GameJam 2-day game development

http://www.inf.ed.ac.uk/student-services/teaching
Informatics Innovative Learning Week

Lecture Plan

XML

We start with technologies for modelling and querying *semistructured data*.

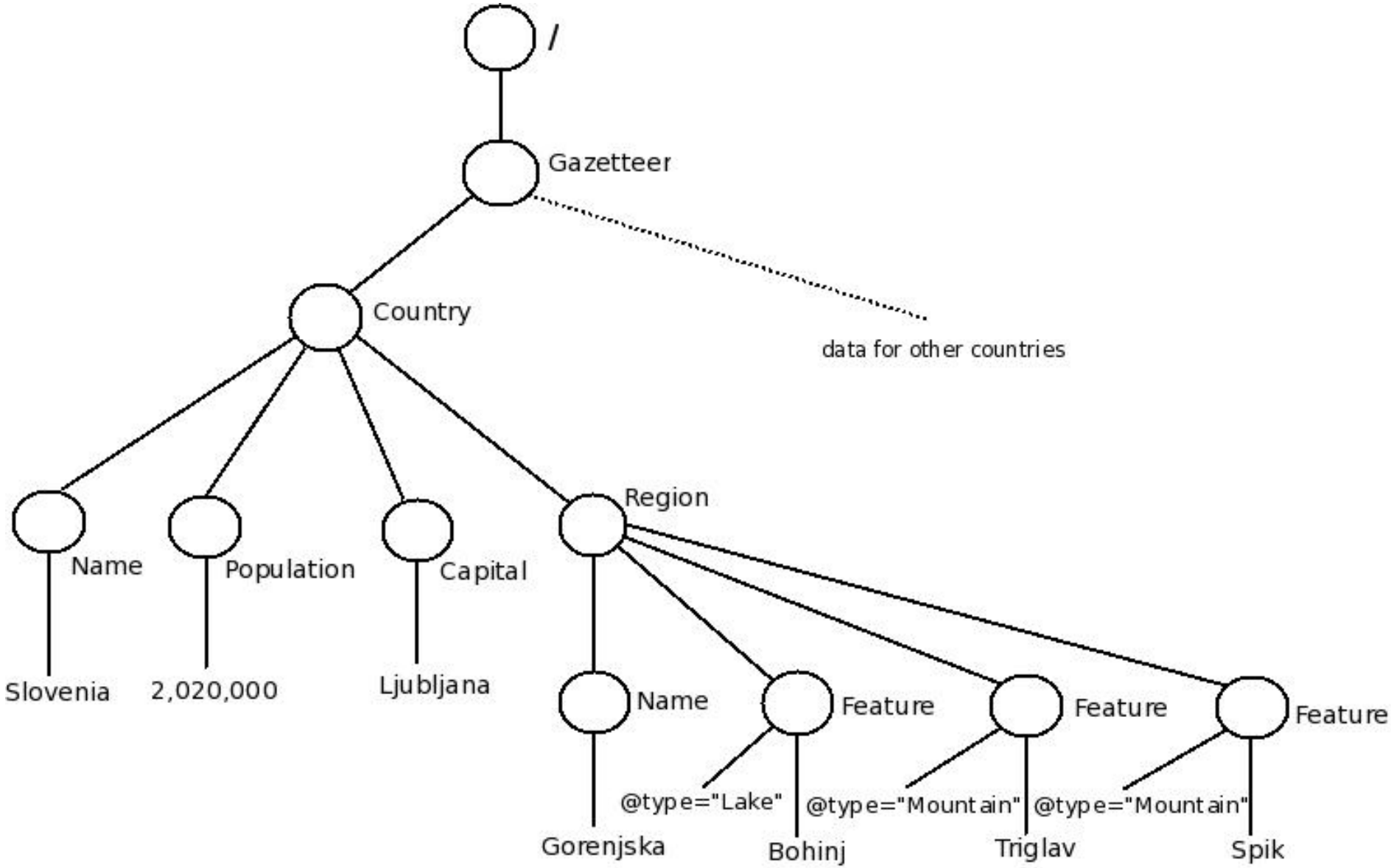
- Semistructured Data: Trees and XML
- Schemas for structuring XML
- Navigating and querying XML with XPath

Corpora

One particular kind of semistructured data is large bodies of written or spoken text: each one a *corpus*, plural *corpora*.

- Corpora: What they are and how to build them
- Applications: corpus analysis and data extraction

Sample Semistructured Data



Sample Semistructured Data in XML

```
<Gazetteer>
    <Country>
        <Name>Slovenia</Name>
        <Population>2,020,000</Population>
        <Capital>Ljubljana</Capital>
        <Region>
            <Name>Gorenjska</Name>
            <Feature type="Lake">Bohinj</Feature>
            <Feature type="Mountain">Triglav</Feature>
            <Feature type="Mountain">Spik</Feature>
        </Region>
    </Country>
    <!-- data for other countries here -->
</Gazetteer>
```

Structuring XML

XML documents are *self-describing*, to a degree:

- The tree structure can always be extracted from textual nesting;
- Elements are always given with their complete name;
- Attributes are all named;
- Everything else is unstructured text.

This is useful as far as it goes, but is fairly rudimentary.

In any given application domain, there may well be a much stricter intended structure which XML documents should follow.

Structuring XML

In any given application domain, there may well be a much stricter intended structure which XML documents should follow.

For example, in the Gazetteer we expect a certain hierarchy:

- The Gazetteer element contains Country elements;
- A Country contains information about its Name, Population and Capital, together with some Region elements.
- A Region includes its Name and zero or more Feature elements.
- A Feature will include a suitable type attribute.

We specify this kind of expected structure with a *schema*.

Schema Languages for XML

In relational databases, a schema specifies the content of a relation.

A schema language for XML is any language for specifying similar kinds of structure in XML documents. There are a number of different schema languages in common use.

Using a formal schema language means:

- Schemas are precise and unambiguous;
- A machine can *validate* whether a document satisfies a certain schema.

If a document X has the format specified by schema S then we say X is *valid* with respect to S.

One document may be valid with respect to several different schemas.

Document Type Definitions

Document Type Definition or *DTD* is a basic schema mechanism for XML.

The DTD schema language is simple, widely used, and has been an integrated feature of XML since its inception.

A DTD includes information about:

- The elements that can appear in a document;
- The attributes of those elements;
- The relationship between different elements such as their order, number, and possible nesting.

We illustrate this by going through a sample DTD for a gazetteer, against which the Slovenian example seen earlier can be validated.

Example DTD

```
<!DOCTYPE Gazetteer [

<!ELEMENT Gazetteer (Country+)>
<!ELEMENT Country (Name,Population,Capital,Region*) >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Population (#PCDATA)>
<!ELEMENT Capital (#PCDATA)>
<!ELEMENT Region (Name,Feature*) >
<!ELEMENT Feature (#PCDATA)>

<!ATTLIST Feature type CDATA #REQUIRED>
]>
```

Dissecting a DTD

Every DTD is a list of declarations.

Dissecting a DTD

Every DTD is a list of declarations.

<!**ELEMENT** Gazetteer (Country+)>

This declares that the Gazetteer element consists of one or more Country elements.

Dissecting a DTD

Every DTD is a list of declarations.

<!**ELEMENT** Gazetteer (Country+)>

This declares that the Gazetteer element consists of one or more Country elements.

<!**ELEMENT** Country (Name,Population,Capital,Region*)>

This declares that a Country element consists of one Name element, followed by one Population element, followed by one Capital element, followed by zero or more Region elements.

Dissecting a DTD

Every DTD is a list of declarations.

<!**ELEMENT** Gazetteer (Country+)>

This declares that the Gazetteer element consists of one or more Country elements.

<!**ELEMENT** Country (Name,Population,Capital,Region*)>

This declares that a Country element consists of one Name element, followed by one Population element, followed by one Capital element, followed by zero or more Region elements.

<!**ELEMENT** Name (#**PCDATA**)>

This declares that the Name element contains text. The keyword #PCDATA stands for "parsed character data".

Dissecting a DTD

<!**ELEMENT** Region (Name,Feature∗)>

This declares that a Region element consists of one Name followed by zero or more Feature elements.

Dissecting a DTD

<!**ELEMENT** Region (Name,Feature∗)>

This declares that a Region element consists of one Name followed by zero or more Feature elements.

<!**ELEMENT** Feature (#**PCDATA**)>

This declares that the Feature element contains just text.

Dissecting a DTD

<!**ELEMENT** Region (Name,Feature∗)>

This declares that a Region element consists of one Name followed by zero or more Feature elements.

<!**ELEMENT** Feature (#**PCDATA**)>

This declares that the Feature element contains just text.

<!**ATTLIST** Feature type **CDATA** #**REQUIRED**>

This declares that the Feature element must have an attribute called type, and that the value of the attribute should be a text string (CDATA stands for "character data").

Why #PCDATA and CDATA? Historical reasons. Please don't ask.
There are precise explanations, but it's hair-splitting.

Element Declarations

An element declaration has this form:

<!**ELEMENT** *elementName* (*contentType*)>

There are four possible content types.

1. EMPTY indicating that the element has no content.
2. ANY meaning that any content is allowed
   (Elements nested within this still need their own declarations).
3. #PCDATA where the element contains text.
4. A regular expression of element names
   (optionally preceded by #PCDATA too).

See the next slide for more on the regular expressions. . .

Element Declarations

An element declaration has this form:

<!**ELEMENT** *elementName* (*contentType*)>

A *mixed contentType* has an optional #PCDATA followed by a regular expression to indicate what content matches this part of the schema.

This regular expression can be of the following.

- A single element name: just that element matches.
- *re1*, *re2* : content matching *re1* followed by more matching *re2*.
- *re** : zero or more pieces of content each matching *re*.
- *re*+ : one or more pieces of content each matching *re*.
- *re*? : content either empty or matching *re*.
- *re1* | *re2* : content matching either *re1* or *re2*.

Attribute Declarations

Attributes of an element are declared separately to the element itself.

<!**ATTLIST** *elementName attName attType attDefault* ...>

This defines attributes for elementName. Multiple attributes can either be defined all together, using the ... here, or in several separate declarations.

Each attribute has three items declared:

- *attName* is the attribute name
- *attType* is a datatype for the value of the attribute.
- *attDefault* indicates whether the attribute is required or optional, and may specify a default value.

Attribute Datatypes and Defaults

Possible *attType* declarations include:

- String: CDATA means that the attribute may take any string value.
- Enumeration: (s1 | s2 | ... | sk) indicates the attribute value must be one of the strings s1, s2, . . . , sk.

Other possibilities are various technical kinds of entities and tokens.

The *attDefault* declaration can be any of:

- #REQUIRED meaning that the attribute must always be given a value in the start tag for that element.
- #IMPLIED meaning that the attribute may be given a value, but it isn't essential.
- Giving a particular string means that is the default for the attribute unless some other value is declared in the element start tag.

Example DTD

```
<!DOCTYPE Gazetteer [

<!ELEMENT Gazetteer (Country+)>
<!ELEMENT Country (Name,Population,Capital,Region*) >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Population (#PCDATA)>
<!ELEMENT Capital (#PCDATA)>
<!ELEMENT Region (Name,Feature*) >
<!ELEMENT Feature (#PCDATA)>

<!ATTLIST Feature type CDATA #REQUIRED>
]>
```

Variation

We might replace the original Feature attribute declaration

```
<!ATTLIST Feature type CDATA #REQUIRED>
```

with an alternative

```
<!ATTLIST Feature type (Mountain|Lake|River) "Mountain">
```

The original Gazetteer would still validate against this, and so would elements like

```
<Feature>Ben Nevis</Feature>
```

which would receive the default type of Mountain.

However, something like

```
<Feature type="Castle">Eilean Donan</Feature>
```

would no longer be valid.

Character Sets

Linking Document and DTD

A Document Type Declaration can appear within an XML document. It indicates what schema should be used to validate the document.

The most common way to connect a document with a DTD is the declaration of an external link:

<!**DOCTYPE** *rootName* SYSTEM "*URI*">

where *rootName* is the name of the root element and *URI* is a *uniform resource identifier* (can be an http:// URL, but there are other possibilities).

It's also possible to include a DTD within the XML document itself

<!**DOCTYPE** *rootName* [*DTD*]>

Here the entire DTD text is placed within the brackets [...] .

Inline DTD

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

<!DOCTYPE Gazetteer [
<!ELEMENT Gazetteer (Country+)>
<!ELEMENT Country (Name,Population,Capital,Region*) >
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Population (#PCDATA)>
<!ELEMENT Capital (#PCDATA)>
<!ELEMENT Region (Name,Feature*) >
<!ELEMENT Feature (#PCDATA)>
<!ATTLIST Feature type CDATA #REQUIRED>
]>

<Gazetteer>
  <!-- Information about countries, regions and features -->
</Gazetteer>
```

DTD Limitations

One of the strengths of the DTD mechanism is its essential simplicity.

However, it is inexpressive in ways that limit its usefulness. For example:

- Elements and attributes cannot be assigned datatypes beyond text and simple enumerations.
- It is impossible to place constraints on data values.
- Constraints apply to only one element at a time, not sets of related elements in the document tree.

These and other issues have led to the development of more powerful XML schema languages, such as XML Schema, Relax NG and Schematron.

However, all of these languages retain the common idea of a schema against which an XML document may be validated.

Publishing Relational Data as XML

XML works well for publishing data online; in particular, it's often used to publish the content of relational database tables.

A key motivation for this is that the simple text format makes the data easily readable and robustly transferable across platforms.

The generality and flexibility of the XML format means that there are many ways to translate relational data into XML.

We illustrate one possible approach using, again, the example data on students taking courses.

Students and Courses

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE University [
<!ELEMENT University (Students,Courses,Takes)>
<!ELEMENT Students (Student*)>
<!ELEMENT Student (mn,name,age,email)>
<!ELEMENT Courses (Course*)>
<!ELEMENT Course (code,name,year)>
<!ELEMENT Taking (Takes)>
<!ELEMENT Takes (mn,name,mark)>
<!ELEMENT mn (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT code (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT mark (#PCDATA)>
]>
```

Students and Courses

```
<University>
  <Students>
    <Student>
      <mn>s0456782</mn><name>John</name>
      <age>18</age><email>john@inf</email>
    </Student>
    ...
  </Students>
  <Courses>
    <Course>
      <code>inf1</code><name>Informatics 1</name><year>1</year>
    </Course>
    <Course>
      <code>math1</code><name>Mathematics 1</name><year>1</year>
    </Course>
  </Courses>
  <Taking>
    <Takes><mn>s0412375</mn><code>inf1</code><mark>80</mark></Takes>
    <Takes><mn>s0378435</mn><code>math1</code><mark>70</mark></Takes>
  </Taking>
</University>
```

Efficiency Concerns

Relational database systems are typically optimised for highly efficient data storage and querying.

In contrast, representing relational data in XML is extremely verbose. As a transport mechanism, though, it is clear and robust. So it can make sense to expand relational database tables into XML for communication: once downloaded, they can be converted back to relational form for a local database system to organise efficient storage and query.

As it happens, ample repetition means that even during transmission XML text compresses well using on-the-fly compression techniques. However, in that compressed form it's not suitable for querying.

There are more recent technologies for compressing XML using knowledge of its structure, in ways that allow efficient querying of the compressed document.