Informatics 1B, 2008

School of Informatics, University of Edinburgh

Data and Analysis

Note 6

Semistructured Data and XML

Alex Simpson

Part II — Semistructured Data

XML

Corpora

Background

Relational databases record data in tables conforming to relational schemata. This imposes rigid structure on data

In many situations, it is useful to structure data in a less rigid way; for example:

- when the data needs to be made publicly available in a standard and easily readable data format;
- when we wish to *mark up* (i.e. annotate) existing unstructured data (e.g. text) with additional information (e.g. semantic information);
- when the data possesses a natural hierarchical structure and/or the structure of the data we wish to record varies from item to item.

Extensible Markup Language (XML)

This is a *markup language*, that is it provides a mechanism, based on *elements* (also called *tags*), for annotating (*marking up*) ordinary text with additional information.

It was developed in the mid 1990's from the Standard General Markup Language (SGML) and Hypertext Markup Language (HTML).

XML has a simple text-based format which provides a convenient basis for making data widely available, e.g. over the web. Indeed, XML has become the *de facto* standard for publishing data on the web.

Structured data (stored locally in a relational database) is often translated into XML for web publishing (even though this means losing some of the structure). (Cf. Tutorial 4.)

Semistructured data

The rules governing XML elements impose a loose structure on data, hence the term *semistructured data*.

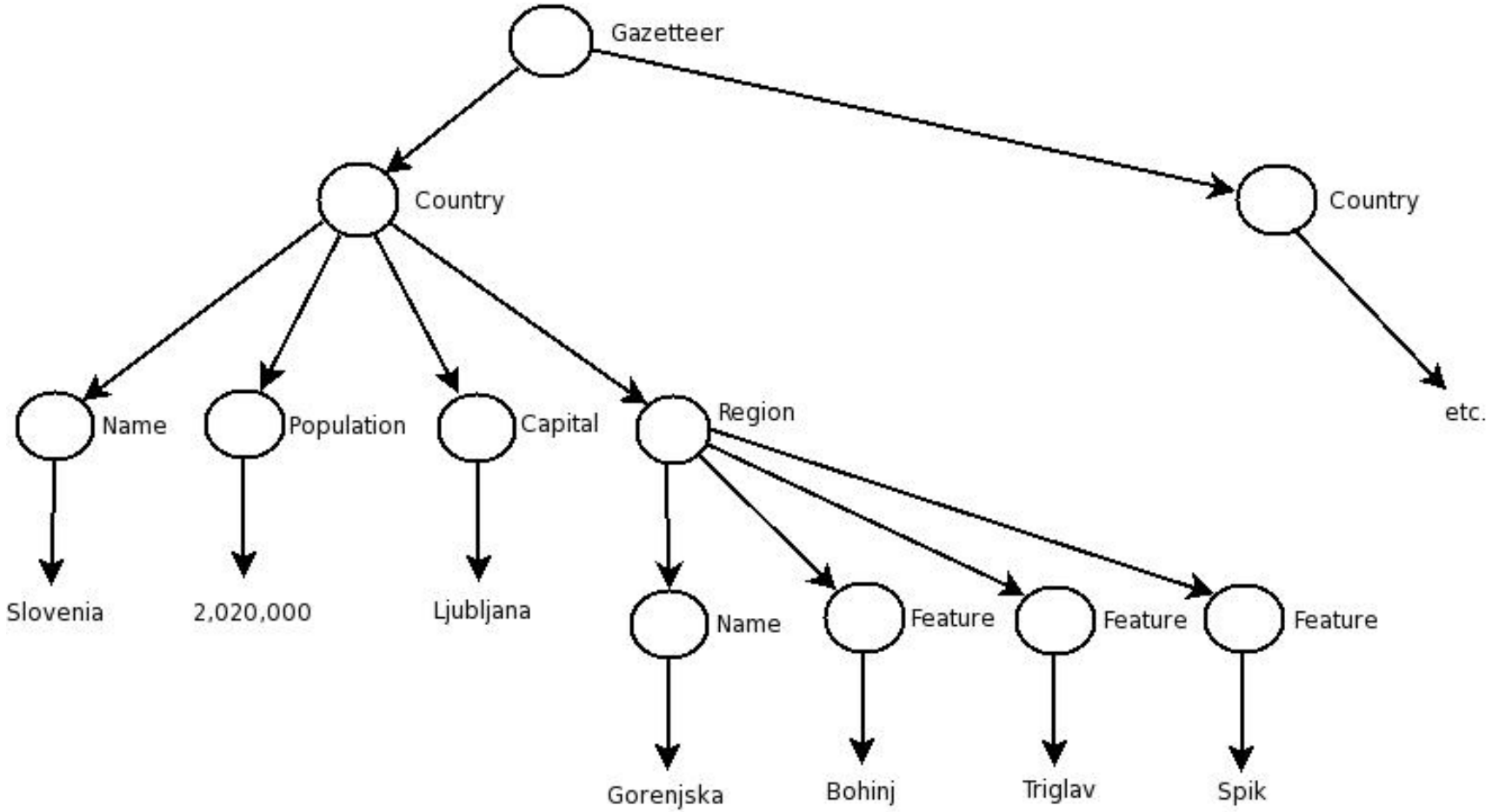
The principal structure that XML imposes is a *tree structure* on elements.

Such a tree structure can be captured (independently of XML) using a tree-based *semistructured data model*.

The next slide illustrates one such model.

The example, a fragment of a gazetter, is chosen because it is one that is naturally accommodated within a hierarchical tree-based structure.

Example of a tree-based semistructured data model



Understanding the tree model

The data is stored at the leaves of the tree.

Each *internal node* of the tree (one that is not a leaf) is given a label that categorises the information that appears in the tree beneath the node.

The meaning of the data at a leaf depends on the labels that appear along the path from the root of the tree (labelled **`Gazetteer`**) to the leaf.

Similarly, the meaning of an internal node depends on the path to the node from the root of the tree. (Note, e.g., that the label **`Name`** is used in two different ways.)

The next slide presents the same structure (with some additional information) in XML format.

```
<Gazetteer>
  <!-- illustrative fragment of data -->
  <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">&#352;pik</Feature>
    </Region>
  </Country>
</Gazetteer>
```

XML Elements

Elements (also called *tags*) are the building blocks of XML documents.

The start of the content of an element **`elm`** is marked with the *start tag* **`<elm>`**, and the end of the content is marked with the *end tag* **`</elm>`**.

In the example, the element **`Gazetteer`** encloses all information in the document. It is the *root element*.

Elements must be *properly nested*. Thus,

```
<Country><Region> ... </Region></Country>
```

is legal, whereas

```
<Country><Region> ... </Country></Region>
```

is illegal.

Elements are case sensitive, so **`Region`** is different from **`REGION`**

Relating XML and the tree model

The existence of a root element together with the proper nesting of elements ensures that every XML document carries a tree structure in a natural way:

- Note that each element of the XML document corresponds to an individual internal node of the tree.

There are some discrepancies between the XML document and tree model:

- In the XML document, elements appear in a certain order (since it is a text document). The tree model is traditionally not assumed to have an ordering on its nodes (though it would be perfectly possible to make such an assumption).
- There is additional information in the XML document (e.g. the value of attributes) that is not present in the associated tree model as we have presented it. (This can be addressed by adapting the tree model.)

Attributes

An element can have descriptive attributes that provide additional information about the element. For example,

```
<Feature type="Mountain">
```

sets the attribute **type** of the element **Feature** to have value **Mountain**.

Note that attribute values are enclosed in (double) quotation marks.

It is possible for one element to have several different attributes, with values defined in sequence within the start tag, e.g.

```
<elm attr1="value1" attr2="value2" attr3="value3">
```

Entity references

These are references to external data, or special characters or text. An entity reference starts with the symbol `&` and ends with the symbol `;`. In the example, the entity reference `&#352;` is a reference to the Unicode character Š which begins the mountain name Špik.

Comments

Comments can be inserted anywhere in an XML document. Comments start with `<!--` and end with `-->`. They can contain arbitrary text apart from the string `--`.

Data content

The data content of the XML document is included as text between the XML tags. Note that all data is text. XML does not have a type structure.

Well-formed documents

An XML document is *well-formed* if it conforms to three guidelines:

- It starts with an XML declaration. The example document does not! A suitable such declaration would be:

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

  This declares the XML version, and states that UTF-8 character encoding is to be used for Unicode. (This is not Inf1B examinable. In Data & Analysis, we are interested in the *content* of a document not in its declaration.)

- It has a root element that contains all other elements.

- All elements are properly nested.

These are minimal requirements on a document. Often there will be other constraints we wish to impose.

Document Type Declarations (DTDs)

A DTD is a set of rules that allows us to specify the desired structure of an XML document. Using a DTD we can specify:

- The elements and entities that can appear in a document.
- What the attributes of the elements are.
- The relationship between different elements including the order of appearance and how they are nested.

An XML document that is structured according to the rules of an associated DTD is called *valid*.

We illustrate DTDs by giving an example DTD for the **`Gazetteer`** XML document of slide 6.8.

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 (Town|Mountain|Lake) #REQUIRED>
]>
```

Understanding the example DTD (1)

A DTD has the form:

```
<!DOCTYPE name [DTDdeclaration]>
```

where:

- ***name*** is the name of the outermost enclosing tag
- ***DTDdeclaration*** contains the rules of the DTD

The DTD declaration has rules for each of the elements that can appear in any XML document matching the DTD.

Understanding the example DTD (2)

The DTD declaration begins with a rule for the outermost element (the *root* element).

In the example:

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

This states that the **`Gazetteer`** element consists of zero or more **`Country`** elements.

Next we have to give rules for the **`Country`** element itself.

Understanding the example DTD (3)

The **Country** element is specified by:

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

This states 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.

The **Name** element is specified by:

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

This states that the **Name** element contains text (i.e. data). The keyword **#PCDATA** abbreviates "parsed character data".

Understanding the example DTD (4)

The **Region** element is specified by:

```
<!ELEMENT Region (Name,Feature+)>
```

This states that a **Region** element consists of: one **Name**, followed by one or more **Feature** elements.

The **Feature** element has two declarations:

```
<!ELEMENT Feature (#PCDATA)>
<!ATTLIST Feature type (Town|Mountain|Lake) #REQUIRED>
```

The first states that it has text content. The second states that it has an attribute **type** that can take one of three values, **Town**, **Mountain** or **Lake**. Moreover, it is required that every **Feature** element must specify a value for the **type** attribute.

Format of element type declaration

An *element type declaration* has the structure:

```
<!ELEMENT name (contentType)>
```

There are five possible content types:

- Another element
- **#PCDATA** indicating text content
- **EMPTY** indicating that the element has no content. (Elements with no content are not required to have an end tag.)
- **ANY** indicating that any content is permitted. (This disables all checking of document structure inside the element.)
- A *regular expression* constructed from the above

Format of regular expressions

Regular expressions were introduced in Informatics 1A Computation and Logic.

DTDs make use of the following format for regular expressions.

- ***exp1*, *exp2***: first ***exp1*** then ***exp2*** in sequence.
- ***exp****: zero or more occurrences of ***exp***.
- ***exp*?**: zero or one occurrences of ***exp***.
- ***exp*+**: one or more occurrences of ***exp***.
- ***exp1*|*exp2***: either ***exp1*** or ***exp2***.

Question: Which of these regular expression forms are used in element type declarations in the example DTD on slide 6.8?

Format of attribute declarations

The attributes of an element are declared separately to the element declaration. The format is:

```
<!ATTLIST elementName (attName attType default)+>
```

This declares a list of at least one attribute for the element ***elementName***.

For each entry in the list:

- ***attName*** is the attribute name
- ***attType*** is a type for the value of the attribute.

  The example, **(Town|Mountain|Lake)** is an *enumerated type* which specifies three possible values (see slide 6.19).

  An alternative is the *string type*, **CDATA**, which allows any string value.

Format of attribute declarations (continued)

- ***default*** allows a default value to be specified, to be assigned in case a start tag for the element does not specify a value for the attribute.

  For example,

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

  specifies that if start the tag **<Feature>** appears, in which no value for the **type** attribute is explicitly stated, then the attribute is given the default value **"Town"**.

  Alternatively, as in the example on slide 6.15, if the keyword **#REQUIRED** is used for ***default*** then every start tag for the element is required to explicitly specify a value for the attribute.

Limitations of DTDs

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

However, it is inexpressive in several ways, and this limits its usefulness. Three particular weaknesses are:

- Elements and attributes cannot be assigned useful types.
- It is impossible to place constraints on data values.
- Elements are always ordered even if this is inappropriate to the application.

These issues and others have been dealt with through the development of more powerful, but more complex, XML format languages, such as XML Schema (which lie beyond the scope of Informatics 1B.)