```
<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">Špik</Feature>
    </Region>
  </Country>
  <!-- data for other countries here -->
</Gazetteer>
```


Unicode

An XML document is a text document written in *Unicode*.

Unicode is a universal code for "text characters", currently supporting around 100,000 different characters.

The Unicode characters contain the standard 128 ASCII characters, but also many, many other characters in use worldwide, from another 92 scripts.

Each character has an assigned *code point*, which is a number between 0 and 1,114,111 inclusive (hexadecimal 0x0–0x10FFF).

The actual representation of Unicode text in memory or "on the wire" depends on a choice of *encoding* of Unicode character sequences as byte streams. The most common encoding is known as UTF-8; others include UTF-16 and UTF-32.

Well-formed documents

An XML document is one containing text that is *well-formed* according to the XML specification. This requires conformance with several technical guidelines, including:

- It starts with an XML declaration. (Our example gazetteer 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. (Not examinable.)
- It has a root element that contains all other elements.
- All elements are properly nested.

As well as these basic requirements on a document, there may be other constraints on format or content which are useful in particular situations.

Part II — Semistructured Data

XML:

Corpora:

Related reading: §§4.1–4.3 of [XWT]
§7.4.2 of [DMS]

Structuring XML

In a given XML application area, there is often an intended structure that an XML document should possess.

For example, in the **`Gazetteer`** example, we expect the various elements to respect the natural hierarchy:

- the **`Country`** elements are inside **`Gazetteer`**;
- the **`Name`** (of the country), **`Population`**, **`Capital`** and **`Region`** elements are inside **`Country`**;
- and the **`Name`** (of the region) and **`Feature`** elements are inside **`Region`**.

Moreover, the **`Feature`** elements assign a suitable value to the attribute **`type`**.

Schema languages for XML

In relational databases, a *schema* specifies the format of a relation (table).

A *schema language* for XML is a language designed for specifying the format of XML documents.

The use of a schema language has two main advantages over giving an informal specification (cf. the informal and partial specification of the **`Gazeteer`** format on the previous slide):

- It is precise and unambiguous
- It is possible for a machine to check whether an XML document satisfies a given schema specification (*validation*)

If an XML document $\boldsymbol{X}$ has the format specified by a given schema $\boldsymbol{S}$ then we say that $\boldsymbol{X}$ is *valid* with respect to $\boldsymbol{S}$.

Document Type Definitions

The *Document Type Definition (DTD)* mechanism is a basic schema language for XML.

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

DTDs allow the specification of:

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

We illustrate this by giving an example DTD for a gazetteer format, which is satisfied by the XML document on slide II:14.

Example DTD

```
<!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>
```


Understanding the example DTD

`<!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`** abbreviates “parsed character data”.

```
<!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 has text content.

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

This declares that the **Feature** element has an attribute **type**, and that the value of the attribute should be a text string (**CDATA** abbreviates "character data"). Moreover, it is *required* that every **Feature** element in the document must assign a value to the **type** attribute.

General format of element declarations

An *element declaration* has the structure:

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

There are four possible content types:

1. **EMPTY** indicating that the element has no content, i.e. it is an *empty element* as defined on slide II:16.
2. **ANY** indicating that any content is permitted.

   Nevertheless elements that appear within the element content must themselves be declared by corresponding element declarations.
3. **#PCDATA** indicating text content.

   (In fact this is an instance of a more general *mixed content* format, which we shall not consider further.)

4. A *regular expression* of element names.

Regular expressions were introduced in Inf1 Computation and Logic.

DTD's make use of the following format for regular expressions.

- Any element name is a regular expression.

  (The element names are the *alphabet* for the 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***.

General format of attribute declarations

The attributes of an element are declared separately to the element declaration. The general 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.
- ***default*** specifies whether the attribute is required or optional, and may specify a default value for the attribute.

We shall consider only the following attribute types:

- *String type:* **CDATA** means that the attribute may have any text string as its value.
- *Enumerated type:* **(** s_1 **|** s_2 **|...|** s_n **)** means that the attribute must take one of the strings s_1, s_2, ..., s_n as its value.

And the following possibilities regarding default values:

- *Required:* **#REQUIRED** means that the attribute must be explicitly assigned a value in every start tag for the element.
- *Optional:* **#IMPLIED** means it is optional whether a value is assigned to the attribute or not.
- *Default:* A fixed string can be specified as the default value for the attribute to take if no explicit value is given in the element's start tag.

A variation on the example

Consider replacing the attribute declaration in the example DTD with the following declaration.

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

With this new (but not with the original) declaration:

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

would be a valid **Feature** element. The **type** attribute would be given the default (and correct) default value **Mountain**.

The element below is not valid with respect to the new DTD (although it is valid for the original DTD)

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

because **Castle** is not one of the specified values for **type**.

Document type declaration

A *document type declaration* can appear in an XML document between the XML declaration and the root element. It links the XML document to a DTD schema intended to specify the structure of the document.

The usual format of a document type declaration is:

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

where ***rootName*** is the name of the root element, and ***URI*** is the *Uniform Resource Indicator* of the intended DTD.

An alternative (illustrated on the next slide) is to include the DTD within the XML document itself, using an *internal declaration*

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


Example internal document type declaration

```
<?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>...</Gazetteer>
```


Limitations of DTD's

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

However, it is inexpressive in several important ways, and this severely limits its usefulness. For example, three weaknesses are:

- Elements and attributes cannot be assigned useful types.
- It is impossible to place constraints on data values.
- There are restrictions on how character data and elements can be combined (they can only be combined as *mixed content*), and there are also undesirable technical restrictions on the forms of regular expression allowed when declaring the structure of elements.

These issues and others have led to the development of more powerful XML format languages, such as XML Schema or Relax NG (which lie beyond the scope of Data & Analysis.)

Publishing relational data as XML

A common application of XML is as a format for publishing data from relational databases.

The benefit of XML for this is that its simple text format makes the data easily readable and 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 simple approach using example data from previous lectures (cf. slide I:99).


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



```
<UniversityData>
<Students>
  <Student> <mn>s0456782</mn> <name>John</name>
    <age>18</age> <email>john@inf</email> </Student>
  <Student> <mn>s0412375</mn> <name>Mary</name>
    <age>18</age> <email>mary@inf</email> </Student>
  <Student> <mn>s0378435</mn> <name>Helen</name>
    <age>20</age> <email>helen@phys</email> </Student>
  <Student> <mn>s0189034</mn> <name>Peter</name>
    <age>22</age> <email>peter@math</email> </Student>
</Students>
<Courses>
  <C><code>inf1</code><name>Informatics 1</name><year>1</year></C>
  <C><code>math1</code><name>Mathematics 1</name><year>1</year></C>
</Courses>
<Takes>
  <T><mn>s0412375</mn><code>inf1</code><mark>80</mark></T>
  <T><mn>s0378435</mn><code>math1</code><mark>70</mark></T>
</Takes>
</UniversityData>
```

Efficiency

Relational database systems are optimised for storage efficiency.

As we have seen, the XML version of relational data is extremely verbose.

Nevertheless, XML can still be stored efficiently using *data compression* (which can be optimised for XML).

Furthermore, once published XML data has been downloaded, it can be converted back to relational data so it can be stored efficiently in a local database system.

Converting XML to back to relational data has the benefit of enabling the data to be queried ising relational database technology (i.e., SQL).

An interesting alternative is to apply newer technology for directly querying XML.