Informatics 1: Data & Analysis

Lecture 10: Structuring XML

Ian Stark

School of Informatics
The University of Edinburgh

Friday 17 February 2017
Semester 2 Week 5

THE UNIVERSITY *of* EDINBURGH

Festival of Creative Learning !

http://www.festivalofcreativelearning.ed.ac.uk

Edinburgh University Students' Association

Teaching Awards

- Best Feedback
- Best Personal Tutor
- Best Student who Tutors
- Best Research Supervisor
- Best Support Staff
- Best Course
- Best Overall Teacher
- Innovative Assessment

Go to **http://is.gd/eusata** and click on any award to make a nomination.

Homework from Tuesday

Read This

- *XML Essentials* from the W3C
  http://www.w3.org/standards/xml/core
- Sections 2.1–2.5 of Møller and Schwartzbach; distributed by email and available outside the ITO in Forrest Hill.

Do This

- Find an SVG file and open it in a text editor to study its XML content.
- Find a `.docx` file, and look at its XML content.

  This format is in fact a zipped archive of XML files, so you will need to unzip it first. Depending on your platform, this may require renaming the `.docx` extension as `.zip`

Lecture Plan

XML — The Extensible Markup Language

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

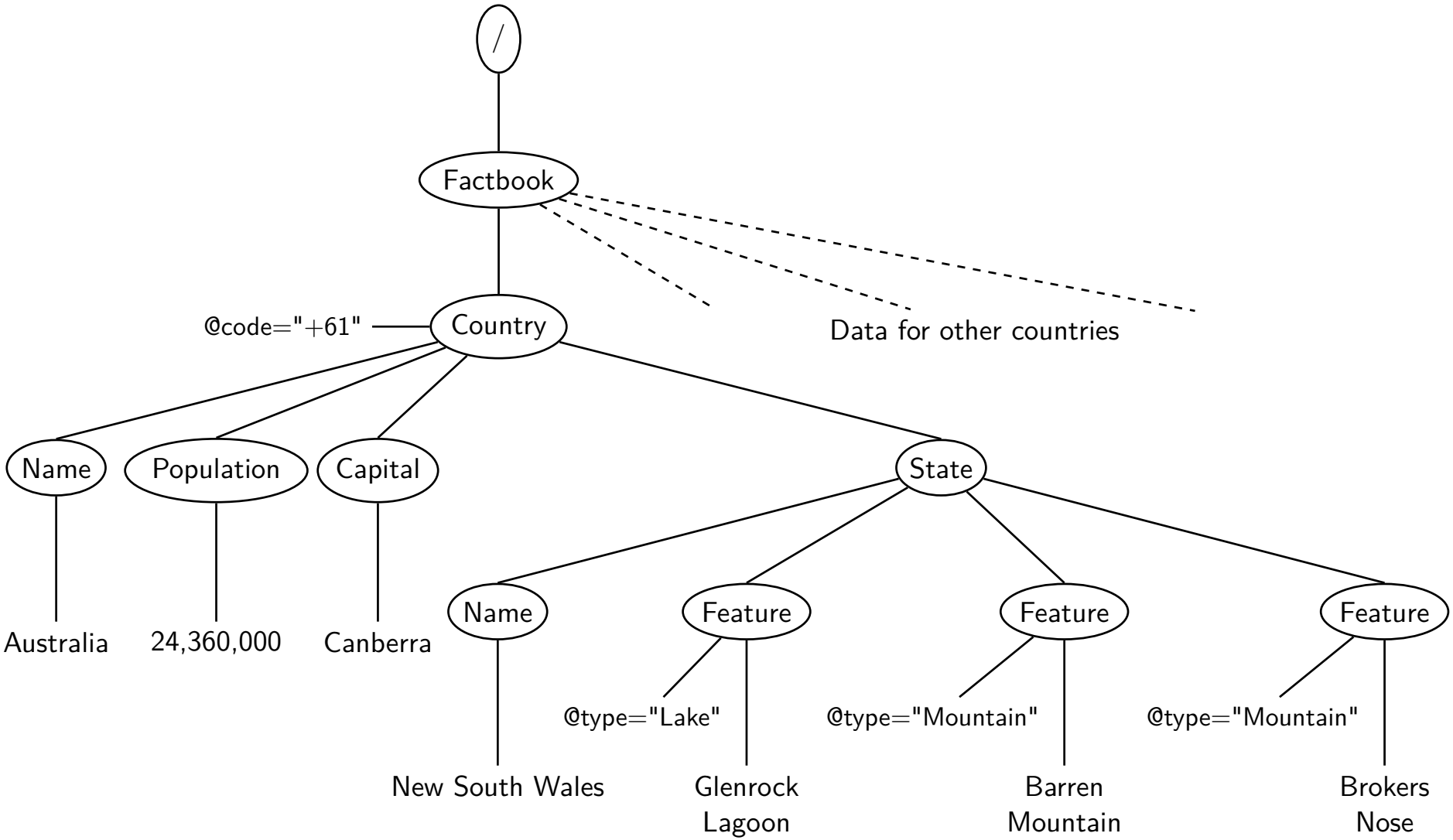
- Semistructured Data: Trees and XML
- Schemas for structuring XML
- Navigating and querying XML with XPath

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

```
<Factbook>
    <Country code="+61">
        <Name>Australia</Name>
        <Population>24,360,000</Population>
        <Capital>Canberra</Capital>
        <State>
            <Name>New South Wales</Name>
            <Feature type="Lake">Glenrock Lagoon</Feature>
            <Feature type="Mountain">Barren Mountain</Feature>
            <Feature type="Mountain">Brokers Nose</Feature>
        </State>
    </Country>
    <!-- data for other countries here -->
</Factbook>
```

Structuring XML

There are a number of basic constraints on XML files, such as proper use of syntax and correctly nesting the tags around elements.

A file satisfying these constraints is a *well-formed XML document*. These are to some extent self-describing:

- 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.

XML tools can manipulate any well-formed document, regardless of the particular elements or attributes used. This is good, but still fairly rudimentary.

In many application domains there is a much stricter structure which XML documents should follow.

Structuring XML

In many application domains there is a much stricter structure which XML documents should follow.

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

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

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

Greek σχημα, with plural "schemata" now almost entirely abandoned in favour of "schemas"

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 or not a document satisfies a certain schema.

If a well-formed XML document D matches the format specified by schema S then we say D is *valid* with respect to S.

One document may be valid with respect to several different schemas; it is also possible to have an XML document that is well-formed but not valid.

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:

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

The following slides run through a sample DTD for a factbook, against which the Australian example seen earlier can be validated.

Example DTD

```
<!ELEMENT Factbook    ( Country+ ) >
<!ELEMENT Country     ( Name, Population, Capital, State* ) >
<!ELEMENT Name        ( #PCDATA ) >
<!ELEMENT Population  ( #PCDATA ) >
<!ELEMENT Capital     ( #PCDATA ) >
<!ELEMENT State       ( Name, Feature* ) >
<!ELEMENT Feature     ( #PCDATA ) >

<!ATTLIST Country code  CDATA #IMPLIED >
<!ATTLIST Feature type  CDATA #REQUIRED >
```

Some think DTD syntax a little ugly

Dissecting a DTD

Every DTD is a list of individual *element declarations* and *attribute declarations*, in any order.

General Form of a DTD

...
<!**ELEMENT** ... >
<!**ELEMENT** ... >
...
<!**ATTLIST** ... ... >
...
<!**ELEMENT** ... >
...

Dissecting a DTD: Declarations (1/2)

Every DTD is a list of individual *element declarations* and *attribute declarations*, in any order.

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

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

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

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 State elements, all in that order.

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

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

Dissecting a DTD: Declarations (2/2)

<!**ELEMENT** State (Name, Feature∗) >

This declares that a State 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** Country code **CDATA** #**IMPLIED** >

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

<!**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.

Historical Reasons

#PCDATA is "parsed character data". The process of *parsing* involves analysing a stream of characters for some local structure. This is appropriate for larger amounts of text, as appears inside an XML element or a text node as a leaf of a tree.

CDATA is "character data". That means a string of characters. This is appropriate for short pieces of text, such as an attribute of an XML element.

Why the difference? Why does one have a hash symbol and the other not?

Historical reasons. Please don't ask for more. There are precise explanations, but it's splitting hairs to explore them.

Element Declarations (1/3)

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. Mixed content where the element contains text, and possibly also child elements.
4. A child declaration using a *regular expression*.

See the next slides for more on mixed content and regular expressions. . .

Element Declarations (2/3)

A mixed content element declaration has one of these forms:

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

<!**ELEMENT** *elementName* (#PCDATA | *child* | *child* | ...)* >

The first of these means that the element can contain arbitrary text as a child node, but no further element nodes.

The second form allows text interspersed with any of the child element nodes named in the declaration, in any order.

The "*" is literal XML syntax, indicating possible repetitions; the ellipsis "..." is not XML syntax, it's there to indicate that the declaration may mention any number of different child elements.

Element Declarations (3/3)

A child declaration uses regular expressions to indicate what element combinations are valid as children of *elementName*.

<!**ELEMENT** *elementName* (*regexp*) >
<!**ELEMENT** *elementName* (*regexp*)? >
<!**ELEMENT** *elementName* (*regexp*)∗ >
<!**ELEMENT** *elementName* (*regexp*)+ >

The *regexp* can be built from any of the following, nested as required:

- A single element name: just that element matches.
- *re1*, *re2* : content matching *re1* followed by more matching *re2*.
- *re1* | *re2* : content matching either *re1* or *re2*.
- Any of these followed by ?, ∗ or + for zero-or-one, zero-or-more, or one-or-more repetitions, respectively.
- Any of these in parentheses (*re*), needed to avoid ambiguity.

Attribute Declarations

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

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

This defines one or more attributes for the named element. Multiple attributes can either be defined all together, using the ... here, or one at a time 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 if optional it may specify a default value.

Attribute Datatypes and Defaults

Possible datatype declarations for attributes include:

- String: **CDATA** means that the attribute may take any string value.
- Enumeration: (s_1 | s_2 | . . . | s_k) indicates the attribute value must be one of the strings s_1, s_2, . . . , s_k.

There are some other more technical possibilities that we won't explore.

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 is optional.
- Giving a particular string means that is the default if none is given in the start tag.

Example DTD

```
<!ELEMENT Factbook    ( Country+ ) >
<!ELEMENT Country     ( Name, Population, Capital, State* ) >
<!ELEMENT Name        ( #PCDATA ) >
<!ELEMENT Population  ( #PCDATA ) >
<!ELEMENT Capital     ( #PCDATA ) >
<!ELEMENT State       ( Name, Feature* ) >
<!ELEMENT Feature     ( #PCDATA ) >

<!ATTLIST Country code  CDATA #IMPLIED >
<!ATTLIST Feature type  CDATA #REQUIRED >
```

Some think DTD syntax a little ugly

Variation

We could replace the original Feature attribute declaration

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

with an alternative

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

This declares a specific list of feature types, and also a default

The original Factbook would still validate against this, and so would this:

```
<Feature>Hawkesbury</Feature>
```

which would (correctly) receive the default type of River.

However, something like

```
<Feature type="Forest">Pilliga</Feature>
```

would be valid under the old declaration — which accepts any text as a feature type — but not under the new one.

Linking Document and DTD

An XML document can use a *Document Type Declaration* to state what DTD (document type definition) schema should be used to validate the document.

The most common way to connect a document with a DTD is by giving an external link:

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

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

It's also possible to include a complete 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 Factbook [

<!ELEMENT Factbook    ( Country+ ) >
<!ELEMENT Country     ( Name, Population, Capital, State* ) >
<!ELEMENT Name        ( #PCDATA ) >
<!ELEMENT Population  ( #PCDATA ) >
<!ELEMENT Capital     ( #PCDATA ) >
<!ELEMENT State       ( Name, Feature* ) >
<!ELEMENT Feature     ( #PCDATA ) >
<!ATTLIST Country code  CDATA #IMPLIED >
<!ATTLIST Feature type  CDATA #REQUIRED >

]>

<Factbook>
  <!-- Information about countries, states and features -->
</Factbook>
```

DTD Limitations

One of the strengths of the DTD mechanism is its 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.
- Element constraints apply to only one element at a time, not to 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 a well-formed 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.

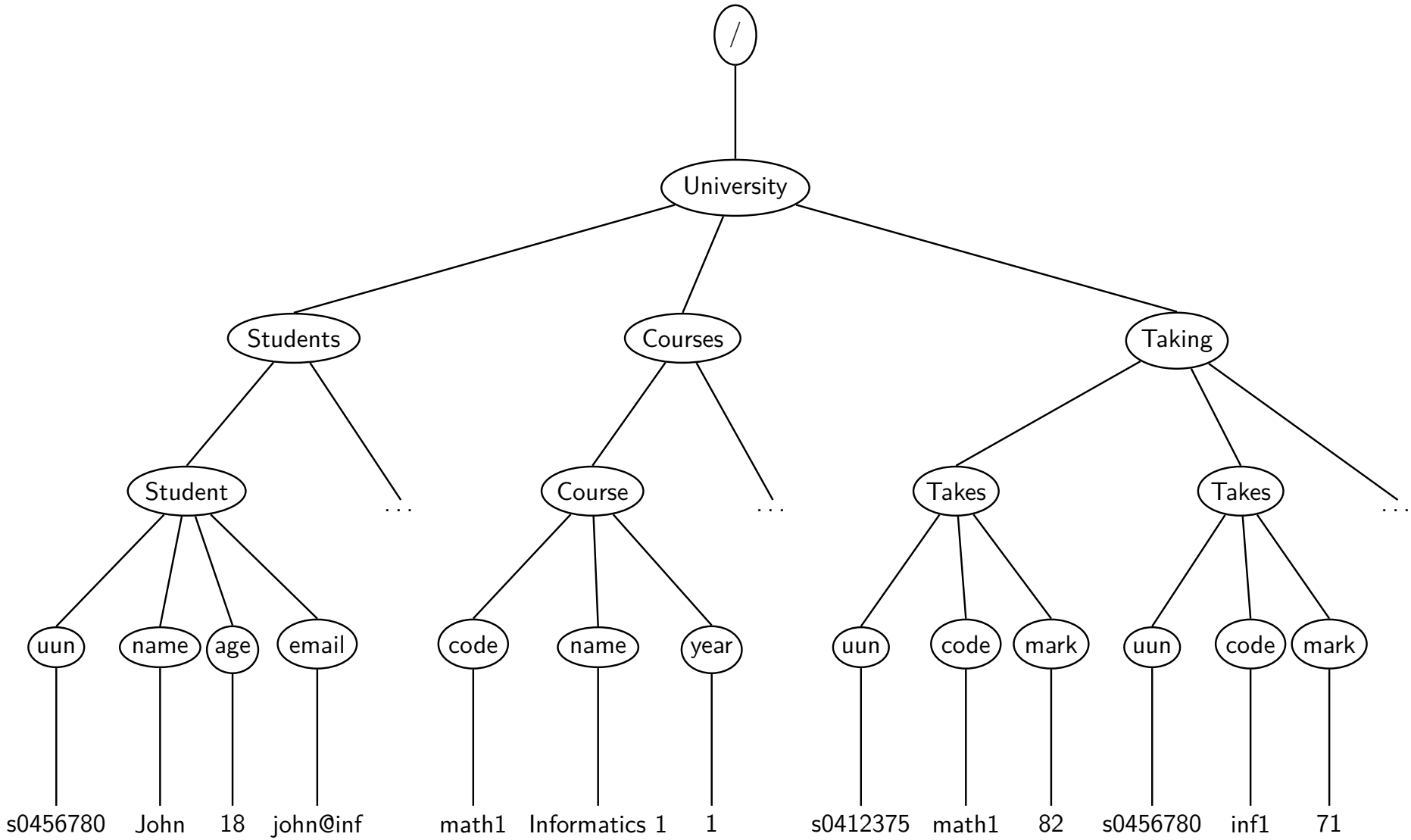
A key motivation for this is that the simple text format makes the data easily readable and robustly transferable across platforms.

Look up *Postel's law*

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

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

Students and Courses — XPath Data Model



Students and Courses — XML Text

```
<University>
  <Students>
    <Student>
      <uun>s0456780</uun><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><uun>s0412375</uun><code>math1</code><mark>82</mark></Takes>
    <Takes><uun>s0456780</uun><code>inf1</code><mark>71</mark></Takes>
    ...
  </Taking>
</University>
```

Students and Courses — XML DTD

```
<!DOCTYPE University [

<!ELEMENT University ( Students, Courses, Taking ) >
<!ELEMENT Students   ( Student )* >
<!ELEMENT Student    ( uun, name, age, email ) >
<!ELEMENT Courses    ( Course )* >
<!ELEMENT Course     ( code, name, year ) >
<!ELEMENT Taking     ( Takes )* >
<!ELEMENT Takes      ( uun, name, mark ) >
<!ELEMENT uun    ( #PCDATA ) >
<!ELEMENT name   ( #PCDATA ) >
<!ELEMENT age    ( #PCDATA ) >
<!ELEMENT email  ( #PCDATA ) >
<!ELEMENT code   ( #PCDATA ) >
<!ELEMENT year   ( #PCDATA ) >
<!ELEMENT mark   ( #PCDATA ) >

]>
```

Efficiency Concerns

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

In contrast, presenting 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.

In fact, as XML contains lots of repetition it will often compress well during transmission 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. These techniques enable true *XML databases*.

Summary

Schemas for XML Documents

XML tools can work with any well-formed XML document. A schema gives a more specialised structure.

If a well-formed XML document D matches the format specified by schema S then document D is valid with respect to S. A document may validate against zero, one, or many different schemas.

The DTD Schema Language

The Document Type Definition language is used to write XML schemas. It states what elements can appear in a document, what attributes and child elements they can have, and in what order.

Linking Document and DTD

An XML document may declare that it meets a specific DTD, either by giving an external link or a complete inline DTD.

Homework

Read This

Find out about Postel's law. This is significant when considering the design of XML, and when writing code to send or receive XML documents.

Do This

Find something in the *Festival of Creative Learning* next week on a topic that interests you. Book and go along to that thing.