CS3 Database Systems
Handout 1
Introduction and XML

Peter Buneman

21 Sept, 2010

Administrative Stuff

Time & Place:	Tuesdays 11:10am-1pm DHT Fac Room South *
Web site:	`http://homepages.inf.ed.ac.uk/opb/dbs`
Instructors:	Peter Buneman (`opb at inf.ed.ac.uk`) Office 5.15 Informatics Forum Office hours: Wednesdays 1:00 - noon*
Text:	*Database Management Systems* Raghu Ramakrishnan and Johannes Gehrke, McGraw Hill. Currently available from Amazon at £30-40 and maybe less from other places

* Subject to change. Please consult the web site.

Other texts

- Jeffrey D. Ullman and Jennifer Widom, *A First Course in Database Systems*, Prentice Hall, 2nd Edition.
- Ramez A. Elmasri and Shamkant B. Navathe *Fundamentals of Database Systems*, Addison-Wesley, 3rd edition.
- Serge Abiteboul, Richard Hull and Victor Vianu *Foundations of Databases*. Addison-Wesley 1995. For theory heavyweights.

Databases at Edinburgh

- e-Science centre
- Digital Curation Centre
- Strongest DB research group in the Europe
- New DB courses:
  - Applied Databases
  - Advanced Databases
  - Querying and Storing XML
  - Distributed Databases
  - Data Integration and Exchange
- Scottish Database Group email list (seminars)

Important notes

- Please check the web site first!
- There are no tutorials for this course, but I will be available during office hours and will be happy to review material and discuss homeworks. So will the demonstrtator/assistant.
- The homeworks will contain questions like those on the exam. Do them!
- The exam has a simple "answer several short questions" format. A sample will be posted.

What you need in order to take this course

- An understanding of the basic mathematical tools that are used in computer science: basic set theory, graph theory, theory of computation (regular expressions and finite state automata) and first-order logic.
- The ability to pick up and use almost any programming language. In this course you may want to use: Java, SQL, XQuery, XSLT, Python, Perl, PHP, etc.

Students who have completed the first two years of the Informatics honours degree should have acquired these abilities provided they have understood the basic principles of computation and programming languages.

Assessment

- Coursework consists of three assignments for a total of 25%. Each assignment will consist partly of some short questions (like those on the exam) and partly of project work that you will develop during the semester. The assignments, their values and due dates are:
  - Assignment 1: basic relational model, data formats relational model, XML, relational algebra (written answers, 8%); assigned 1 October, due 15 October
  - Assignment 2: SQL programming (10%), assigned 21 Octber, due 4 November
  - Assignment 3: normalization, optimization query/transaction processing, XML (written answers, 7%), assigned 18 November, due 2 December.
- Exam (short questions) 75%

Plagiarism will be refereed externally

Late submissions will be penalised

What the subject is about

- Organization of data
- Efficient retrieval of data
- Reliable storage of data
- Maintaining consistent data
- Sharing data (concurrency)
- Semistructured data and documents (XML)

Not surprisingly all these topics are related.

We won't start with relational databases ...

We'll start with XML. Why?

- Because you are familiar with it (or at least with HTML.)
- Because XML query systems are relatively "lightweight".
- Because it serves as a good introduction for why data organization and efficiency are needed.
- The "busy work" – computer accounts, learning new systems, etc. is better distributed.

We'll start, however, with a brief introduction to databases in general.

What is a Database?

- A *database* (DB) is a large, integrated collection of data.
- A DB models a real-world "enterprise" or collection of knowledge/data.
- A *database management system* (DBMS) is a software package designed to store and manage databases.

Why study databases?

- Everybody needs them, i.e. \$\$\$ (or even £££).
- They are connected to most other areas of computer science:
  - programming languages and software engineering (obviously)
  - algorithms (obviously)
  - logic, discrete math, and theory of comp. (essential for data organization and query languages).
  - "Systems" issues: concurrency, operating systems, file organization and networks.
- There are lots of interesting problems, both in database research and in implementation.
- It is a great area in which systems and theory get combined (relational DBs, transactions, database design, XML processing, distributed data, Google, . . .)

Why not "program" databases when we need them?

For simple and small databases this is often the best solution. Flat files and grep get us a long way.

We run into problems when

- The structure is complicated (more than a simple table)
- The database gets large
- Many people want to use it simultaneously

Example: A personal calendar

Of course, such things are easy to find, but let's consider designing the "database" component from scratch. We might start by building a file with the following structure:

What	When	Who	Where
Lunch	24/10 1pm	Fred	Joe's Diner
CS123	25/10 9am	Dr. Egghead	Room 234
Biking	26/10 9am	Jane	Start at Jane's
Dinner	26/10 6pm	Jane	Cafe le Boeuf
...	...	...	...

This text file is an easy structure to deal with (though it would be nice to have some software for parsing dates etc.) So there's no need for a DBMS.

Problem 1. Data Organization

So far so good. But what about the "who" field? We don't just want a person's name, we want also to keep e-mail addresses, telephone numbers etc. Should we expand the file?

What	When	Who	Who-email	Who-tel	Where
Lunch	24/10 1pm	Fred	fred@abc.com	1234	Joe's Diner
CS123	25/10 9am	Egghead	eggy@boonies.edu	7862	Room 234
Biking	26/10 9am	Jane	janew@xyz.org	4532	Start at Jane's
Dinner	26/10 6pm	Jane	janew@xyz.org	4532	Cafe le Boeuf
...	...	...	...	...	...

But this is unsatisfactory. It appears to be keeping our address book in our calendar and doing so *redundantly*.

So maybe we want to link our calendar to our address book. But how?

Problem 2. Efficiency

Probably a personal address book would never contain more than a few hundred entries, but there are things we'd like to do quickly and efficiently – even with our simple file. Examples:

- "Give me all appointments on 10/28"
- "When am I next meeting Jim?"

We would like to "program" these as quickly as possible.

We would like these programs to be executed efficiently. What would happen if you were maintaining a "corporate" calendar with hundreds of thousands of entries?

Problem 3. Concurrency and Recovery

Suppose other people are allowed access to your calendar and are allowed to modify it? How do we stop two people changing the file at the same time and leaving it in a physical (or logical) mess?

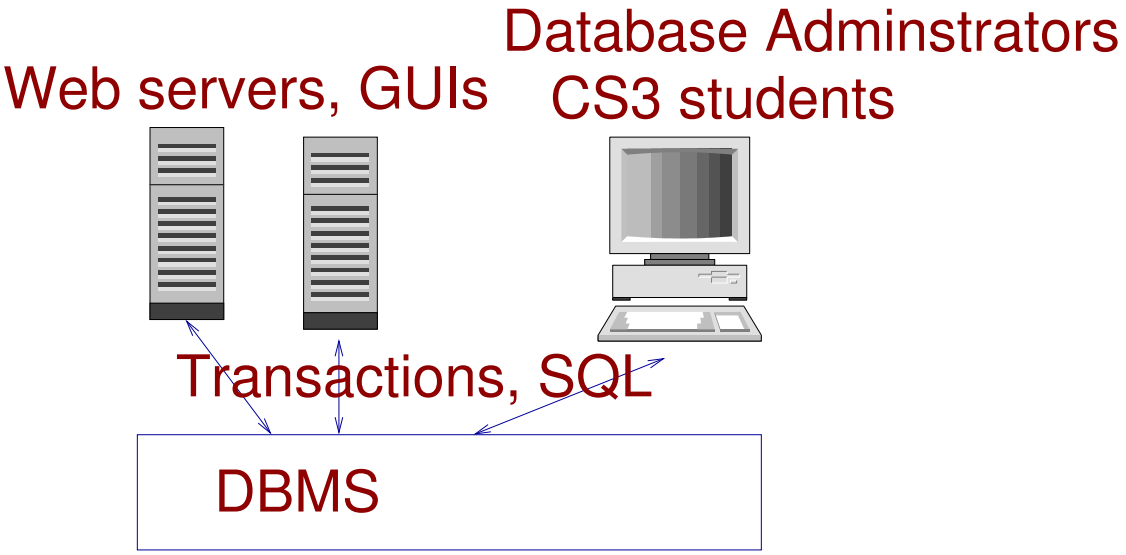
Suppose the system crashes while we are changing the calendar. How do we recover our work?

Example: You schedule a lunch with a friend, and your secretary *simultaneously* schedules lunch with your chairman?

You both see that the time is open, but only one will show up in the calendar. Worse, a "mixture" or corrupted version of the two appointments may appear.

Transactions

- Key concept for concurrency is that of a *transaction* – a sequence of database actions (read or write) that is considered as one indivisible action.
- Key concept for recoverability is that of a *log* – a record of the sequence of actions that changed the database.
- DBMSs are usually constructed with a client/server architecture.



Database architecture – the traditional view

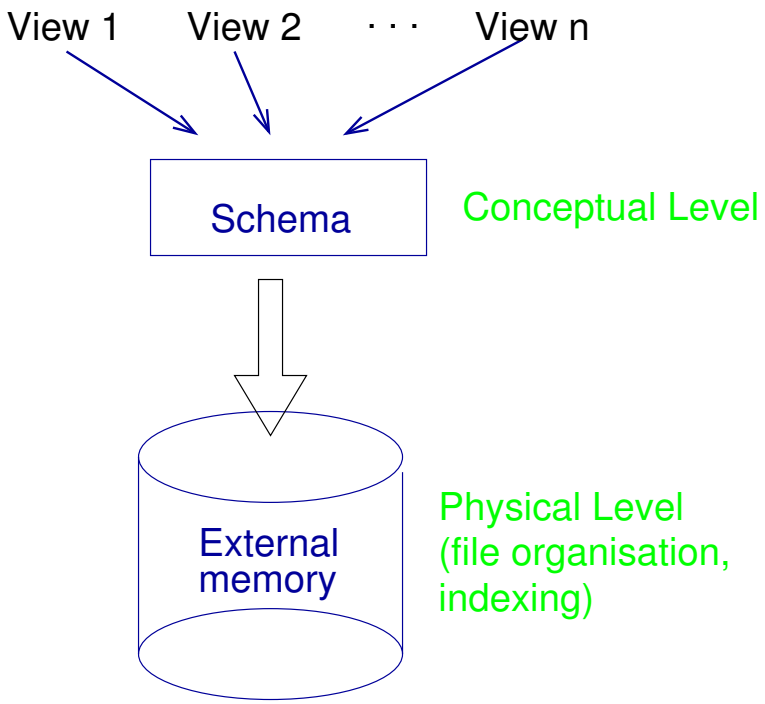
It is common to describe databases in two ways:

- **The logical structure**. What users see. The program or query language interface.
- **The physical structure**. How files are organized. What indexing mechanisms are used.

Further it is traditional to split the "logical" level into two components. The overall database design and the *views* that various users get to see.

This led to the term "three-level architecture"

Three-Level Architecture



Example

A user of a relational database system should be able to use SQL to query the database, e.g.

```
SELECT   When, Where
FROM     Calendar
WHERE    Who = "Bill"
```

without knowing, nor caring about how the precisely how data is stored.

After all, you don't worry much how numbers are stored when you program some arithmetic or use a computer-based calculator. This is really the same principle.

That's the traditional view, but ...

Three-level architecture is never achievable. When databases get big, users still have to worry about efficiency.

There are databases over which we have no control. The Web is a giant, disorganized, database.

There are also well-organized databases on the web, for example,

`http://www.moviedatabase.com`

`http://www.cia.gov/cia/publications/factbook/`

which have a very clean organization, but for which the terminology does not quite apply.

XML – Outline

- Background: documents (SGML/HTML) and databases
- XML Basics
- Programming with XML: SAX and DOM
- XPath and XQuery
- Document Type Descriptors

Some URLs

- XML standard: `http://www.w3.org/TR/REC-xml`
  A caution. Most W3C standards are quite impenetrable. There are a few exceptions to this –some of the XQuery and XML schema documents are readable – but as a rule, looking at the standard is *not the place to start*

- Annotated standard: `http://www.xml.com/axml/axml.html`. Useful if you are consulting the standard, but not the place to start.

- Lots of good stuff at `http://www.oasis-open.org/cover/xml.html`

- Pedestrian tutorials: `http://www.w3schools.com/xml/default.asp` and `http://www.spiderpro.com/bu/buxmlm001.html`

- General articles/standards for XML, XSL, XQuery, etc.: `http://www.w3.org/TR/REC-xml`

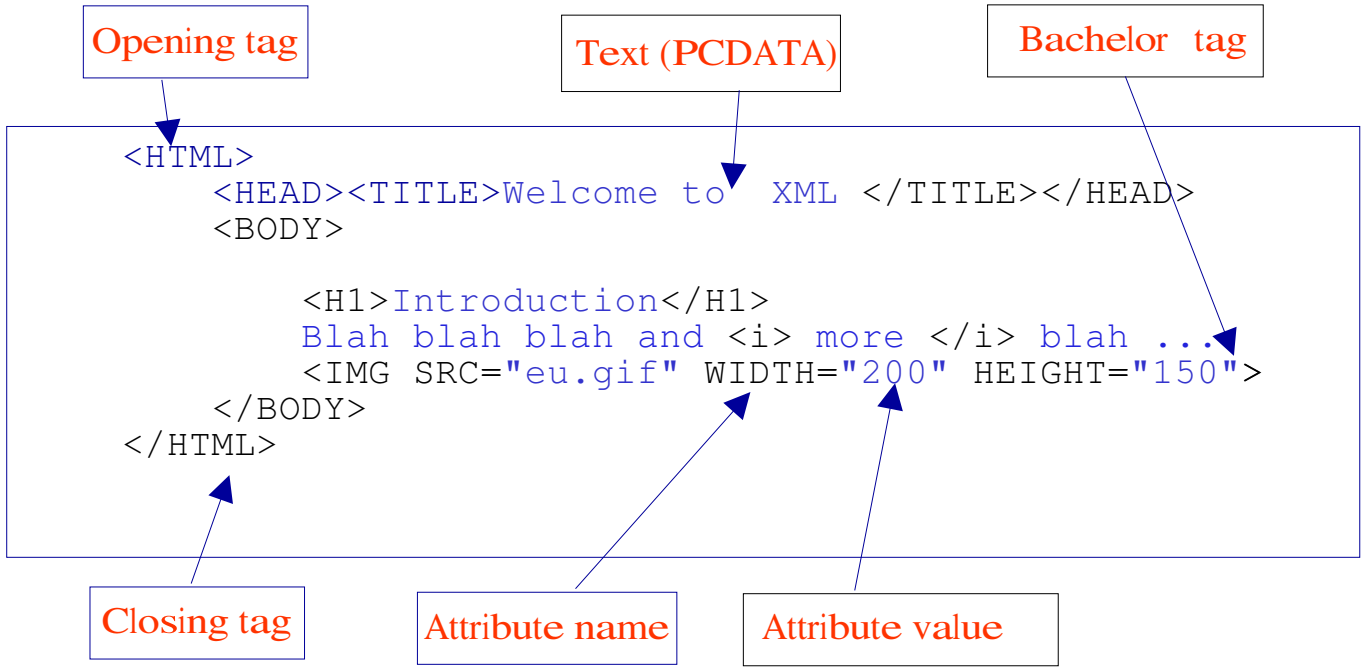
Documents vs. Databases

Documents have structure and contain data. What's the difference?

Documents	Databases
Lots of small documents	Fewer large databases
Usually static	Usually dynamic (lots of updates)
Implicit structure (section, paragraph,...)	Explicit structure (schema)
Structure conveyed by tagging	Structure conveyed by tuples/classes, like types in Java
Human friendly	Machine friendly
Concerns: presentation, editing, character encodings, language.	Concerns: queries, models, transactions, recovery, performance.

Document Formats

HTML is widely used, but there are many others: Tex, LaTex, RTF....



The thin line...

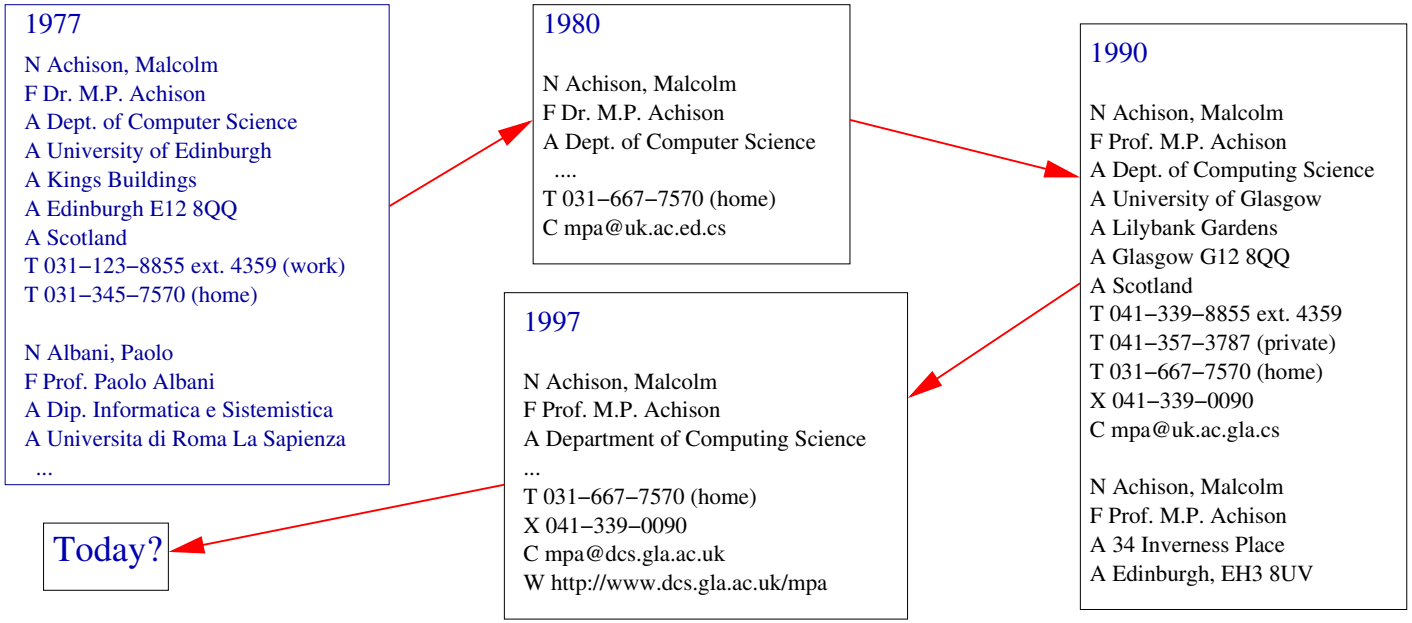
... between document formats and data formats. Much of the world's data – especially scientific data – is held in pre-XML *data formats*.

Files that conform to some data format are sometimes called "flat" files. But their structure is far from flat!

Examples:

- Personal address book
- Configuration files
- Data in specialized formats (e.g. Swissprot)
- Data in generic formats such as ASN.1 (bibliographic data, GenBank)

Data formats: many years of my address book



My Calendar (format of the ical program)

```
Appt [
Start [720]
Length [60]
Uid [horus.cis.upenn.edu_829e850c_17e9_70]
Owner [peter]
Text [16 [Lunch -- Sanjeev]]
Remind [5]
Hilite [always]
Dates [Single 4/12/2001 End
]
]
Appt [
Start [1035]
Length [45]
Uid [horus.cis.upenn.edu_829e850c_17e9_72]
Owner [peter]
Text [7 [Eduardo]]
Remind [5]
...
```

Data formats: Swissprot

```
ID    11SB_CUCMA STANDARD; PRT; 480 AA.
AC    P13744;
DT    01-JAN-1990 (REL. 13, CREATED)
DT    01-JAN-1990 (REL. 13, LAST SEQUENCE UPDATE)
DT    01-NOV-1990 (REL. 16, LAST ANNOTATION UPDATE)
DE    11S GLOBULIN BETA SUBUNIT PRECURSOR.
OS    CUCURBITA MAXIMA (PUMPKIN) (WINTER SQUASH).
OC    EUKARYOTA; PLANTA; EMBRYOPHYTA; ANGIOSPERMAE; DICOTYLEDONEAE;
OC    VIOLALES; CUCURBITACEAE.
RN    [1]
RP    SEQUENCE FROM N.A.
RC    STRAIN=CV. KUROKAWA AMAKURI NANKIN;
RX    MEDLINE; 88166744.
RA    HAYASHI M., MORI H., NISHIMURA M., AKAZAWA T., HARANISHIMURA I.;
RL    EUR. J. BIOCHEM. 172:627-632(1988).
RN    [2]
RP    SEQUENCE OF 22-30 AND 297-302.
RA    OHMIYA M., HARA I., MASTUBARA H.;
RL    PLANT CELL PHYSIOL. 21:157-167(1980).
```

Swissprot – cont

```
CC    -!- FUNCTION: THIS IS A SEED STORAGE PROTEIN.
CC    -!- SUBUNIT: HEXAMER; EACH SUBUNIT IS COMPOSED OF AN ACIDIC AND A
CC        BASIC CHAIN DERIVED FROM A SINGLE PRECURSOR AND LINKED BY A
CC        DISULFIDE BOND.
CC    -!- SIMILARITY: TO OTHER 11S SEED STORAGE PROTEINS (GLOBULINS).
DR    EMBL; M36407; G167492; -.
DR    PIR; S00366; FWPU1B.
DR    PROSITE; PS00305; 11S_SEED_STORAGE; 1.
KW    SEED STORAGE PROTEIN; SIGNAL.
FT    SIGNAL 1 21
FT    CHAIN 22 480 11S GLOBULIN BETA SUBUNIT.
FT    CHAIN 22 296 GAMMA CHAIN (ACIDIC).
FT    CHAIN 297 480 DELTA CHAIN (BASIC).
FT    MOD_RES 22 22 PYRROLIDONE CARBOXYLIC ACID.
FT    DISULFID 124 303 INTERCHAIN (GAMMA-DELTA) (POTENTIAL).
FT    CONFLICT 27 27 S -> E (IN REF. 2).
FT    CONFLICT 30 30 E -> S (IN REF. 2).
SQ    SEQUENCE 480 AA; 54625 MW; D515DD6E CRC32;
      MARSSLFTFL CLAVFINGCL SQIEQQSPWE FQGSEVWQQH RYQSPRACRL ENLRAQDPVR
      RAEAEAIFTE VWDQDNDEFQ CAGVNMIRHT IRPKGLLLPG FSNAPKLIFV AQGFGIRGIA
      IPGCAETYQT DLRRSQSAGS AFKDQHQKIR PFREGDLLVV PAGVSHWMYN RGQSDLVLIV
      ...
```

And if you need futher convincing...

... cd to the `/etc` directory and look at all the "config" files (`.cf`, `.conf`, `.config`, `.cfg`).

These are not huge amounts of data, but having a common data format would at least relieve the need to have as many parsers as files!

The Structure of XML

We start with the basic structure of XML.

- XML consists of tags and text
- Tags come in pairs ⟨date⟩. . . ⟨/date⟩
- They must be properly nested
  - ⟨date⟩...⟨day⟩...⟨/day⟩...⟨/date⟩ — good
  - ⟨date⟩...⟨day⟩...⟨/date⟩...⟨/day⟩ — bad

  (You can't do ⟨i⟩ ...⟨b⟩ ...⟨/i⟩ ...⟨/b⟩ in HTML)

The recent specification of HTML makes it a subset of XML (fixed tag set). Bachelor tags (e.g. ⟨p⟩) are not allowed.

XML text

XML has only one *basic* type – text.

It is bounded by tags e.g.

⟨title⟩The Big Sleep⟨/title⟩ ⟨year⟩1935⟨/year⟩ — 1935 is still text

XML text is called *PCDATA* (for parsed character data). It uses a 16-bit encoding, e.g. \&\#x0152 for the Hebrew letter Mem

Some proposals for XML "types", such as XML-schema, propose a richer set of base types.

XML structure

Nesting tags can be used to express various structures. E.g. A tuple (record) :

```
⟨person⟩
   ⟨name⟩ Malcolm Atchison ⟨/name⟩
   ⟨tel⟩ 0141 898 4321 ⟨/tel⟩
   ⟨email⟩ mp@dcs.gla.ac.sc ⟨/email⟩
⟨/person⟩
```

XML structure (cont.)

We can represent a list by using the same tag repeatedly:

```
⟨addresses⟩
   ⟨person⟩...⟨/person⟩
   ⟨person⟩...⟨/person⟩
   ⟨person⟩...⟨/person⟩
   ...
⟨/addresses⟩
```

Terminology

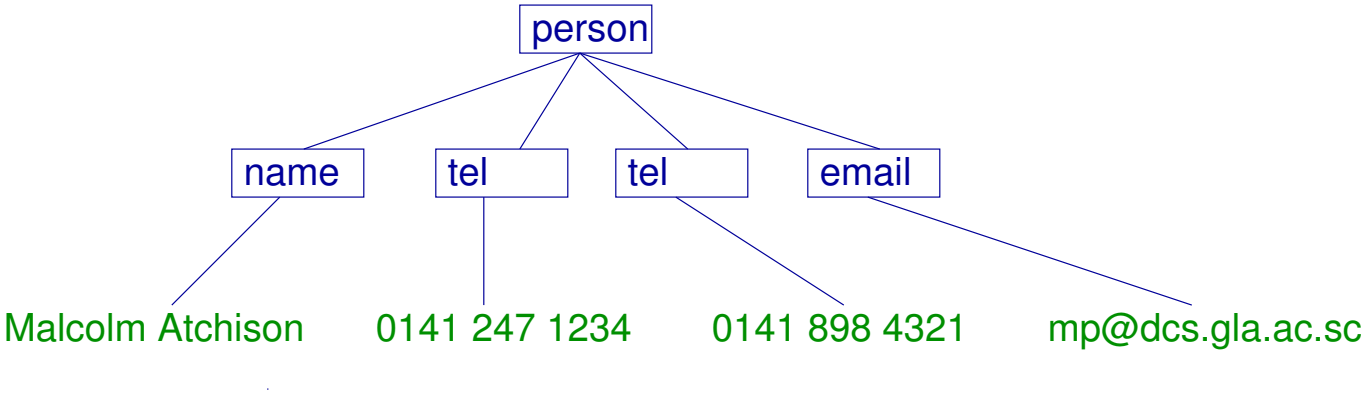
The segment of an XML document between an opening and a corresponding closing tag is called an *element*.

1. ⟨person⟩
2. ⟨name⟩ Malcolm Atchison ⟨/name⟩
3. ⟨tel⟩ 0141 247 1234 ⟨/tel⟩
4. ⟨tel⟩ 0141 898 4321 ⟨/tel⟩
5. ⟨email⟩ mp@dcs.gla.ac.sc ⟨/email⟩
6. ⟨/person⟩

The text fragments ⟨person⟩...⟨/person⟩ (lines 1-6), ⟨name⟩...⟨/name⟩ (line 2), etc. are elements.

The text between two tags is (e.g. lines 2-5) is sometimes called the *contents* of an element.

XML is tree-like



Mixed Content

An element may contain a mixture of text and other elements. This is called *mixed content*

```
⟨airline⟩
    ⟨name⟩ British Airways ⟨/name⟩
    ⟨motto⟩
        World's ⟨dubious⟩favorite⟨/dubious⟩ airline
    ⟨/motto⟩
⟨/airline⟩
```

XML generated from databases and data formats typically do not have mixed content. It is needed for compatibility with HTML.

A Complete XML Document

```
⟨?xml version="1.0"?⟩
⟨person⟩
    ⟨name⟩ Malcolm Atchison ⟨/name⟩
    ⟨tel⟩ 0141 247 1234 ⟨/tel⟩
    ⟨tel⟩ 0141 898 4321 ⟨/tel⟩
    ⟨email⟩ mp@dcs.gla.ac.sc ⟨/email⟩
⟨/person⟩
```

How would we represent "structured" data in XML?

Example:

- Projects have titles, budgets, managers, ...
- Employees have names, employee empids, ages, ...

Employees and projects mixed

```
⟨db⟩
   ⟨project⟩
      ⟨title⟩ Pattern recognition ⟨/title⟩
      ⟨budget⟩ 10000 ⟨/budget⟩
      ⟨manager⟩ Joe ⟨/manager⟩
   ⟨/project⟩
   ⟨employee⟩
      ⟨name⟩ Joe ⟨/name⟩
      ⟨empid⟩ 344556 ⟨/empid⟩
      ⟨age⟩ 34 ⟨/age⟩
   ⟨/employee⟩
   ⟨project⟩...⟨/project⟩
   ⟨project⟩...⟨/project⟩
   ⟨employee⟩...⟨/employee⟩
⟨/db⟩
```

Employees and Projects Grouped

```
⟨db⟩
   ⟨projects⟩
      ⟨project⟩
         ⟨title⟩ Pattern recognition ⟨/title⟩
         ⟨budget⟩ 10000 ⟨/budget⟩
         ⟨manager⟩ Joe ⟨/manager⟩
      ⟨/project⟩
      ⟨project⟩...⟨/project⟩
      ⟨project⟩...⟨/project⟩
   ⟨/projects⟩
   ⟨employees⟩
      ⟨employee⟩...⟨/employee⟩
      ⟨employee⟩...⟨/employee⟩
   ⟨/employees⟩
⟨/db⟩
```

No tags for employees or projects

```
⟨db⟩
      ⟨title⟩ Pattern recognition ⟨/title⟩
      ⟨budget⟩ 10000 ⟨/budget⟩
      ⟨manager⟩ Joe ⟨/manager⟩
      ⟨name⟩ Joe ⟨/name⟩
      ⟨empid⟩ 344556 ⟨/empid⟩
      ⟨age⟩ 34 ⟨/age⟩
      ⟨title⟩...⟨/title⟩
      ⟨budget⟩...⟨/budget⟩
      ⟨manager⟩...⟨/manager⟩
      ⟨name⟩...⟨/name⟩
      ...
⟨/db⟩
```

Here we have to assume more about the tags and their order.

And there is more to be done

- Suppose we want to represent the fact that employees work on projects.
- Suppose we want to constrain the manager of a project to be an employee.
- Suppose we want to guarantee that employee ids are unique.

We need to add more to XML in order to state these constraints.

Attributes

An (opening) tag may contain *attributes*. These are typically used to describe the content of an element

```
⟨entry⟩
      ⟨word language = "en"⟩ cheese ⟨/word⟩
      ⟨word language = "fr"⟩ fromage ⟨/word⟩
      ⟨word language = "ro"⟩ branza ⟨/word⟩
      ⟨meaning⟩ A food made ...⟨/meaning⟩
⟨/entry⟩
```

Attributes (contd)

Another common use for attributes is to express dimension or type

```
⟨picture⟩
    ⟨height dim= "cm"⟩ 2400 ⟨/height⟩
    ⟨width dim= "in"⟩ 96 ⟨/width⟩
    ⟨data  encoding = "gif" compression = "zip"⟩
        M05-.+C$@02!G96YE<FEC ...
    ⟨/data⟩
⟨/picture⟩
```

A document that obeys the *nested tags* rule and does not repeat an attribute within a tag is said to be *well-formed*.

When to use attributes

It's not always clear when to use attributes

```
⟨person id = "123 45 6789"⟩
    ⟨name⟩ F. McNeil ⟨/name⟩
    ⟨email⟩ fmcn@barra.org.sc ⟨/email⟩
⟨/person⟩
```

```
⟨person⟩
    ⟨id⟩ 123 45 6789 ⟨/id⟩
    ⟨name⟩ F. McNeil ⟨/name⟩
    ⟨email⟩ fmcn@barra.org.sc ⟨/email⟩
⟨/person⟩
```

Attributes can only contain text — not XML elements.

How do we program with or query XML?

Consider the equivalent of a really simple database query

"Find the `names` of `employees` whose `age` is `55`"

We need to worry about the following:

- How do we find all the `employee` elements? By traversing the whole document or by looking only in certain parts of it?
- Where are the `age` and `name` elements to be found? Are they children of an `employee` element or do they just occur somewhere underneath?
- Are the `age` and `name` elements unique? If they are not, what does the query mean?
- Do the `age` and `name` elements occur in any particular order?

If we knew the answers to these questions, it would probably be much simple to write a program/query. A DTD provides these answers, so if we know a document conforms to a DTD, we can write simpler and more efficient programs.

However, most PL interfaces and query languages do not require DTDs.

Programming language interfaces. (APIs)

- SAX – Simple API for XML. A parser that does a left-to-right tree walk (or document order traversal) of the document. As it encounters tags and data, it calls user-defined functions to process that data.
  - Good: Simple and efficient. Can work on arbitrarily large documents.
  - Bad: Code attachments can be complicated. They have to "remember" data. What do you do if you don't know the order of `name` and `age` tags?
- Document Object Model (DOM). Each node is represented as a Java (C++, Python, ...) object with methods to retrieve the PCDATA, children, descendants, etc. The chldren are represented (roughly speaking) as an array.
  - Good: Complex programs are simpler. Easier to operate on multiple documents.
  - Bad: Most implementations require the XML to fit into main memory.

Some Sample XML

```
⟨db⟩
 ⟨department⟩
    ⟨dname⟩ manufacturing ⟨/dname⟩
    ⟨tel⟩ 1432 ⟨/tel⟩
    ⟨employee⟩
      ⟨name⟩ Jane Dee ⟨/name⟩ ⟨tel⟩ 6734 ⟨/tel⟩ ⟨sal⟩ 50 ⟨/sal⟩
    ⟨/employee⟩
    ⟨employee⟩
      ⟨name⟩ Mary Smith ⟨/name⟩ ⟨tel⟩ 1432 ⟨/tel⟩ ⟨sal⟩ 45 ⟨/sal⟩
    ⟨/employee⟩
    ⟨employee⟩
      ⟨name⟩ John Brown ⟨/name⟩ ⟨sal⟩ 25 ⟨/sal⟩
    ⟨/employee⟩
  ⟨/department⟩
  ⟨department⟩
    ⟨dname⟩ sales ⟨/dname⟩
    ⟨tel⟩ 3221 ⟨/tel⟩
```

```
    ⟨employee⟩
      ⟨name⟩ Fred Beans ⟨/name⟩ ⟨tel⟩ 3221 ⟨/tel⟩ ⟨sal⟩ 32 ⟨/sal⟩
    ⟨/employee⟩
    ⟨employee⟩
      ⟨name⟩ Kate Smith ⟨/name⟩ ⟨tel⟩ 1432 ⟨/tel⟩ ⟨sal⟩ 42 ⟨/sal⟩
    ⟨/employee⟩
  ⟨/department⟩
  ⟨department⟩
    ⟨dname⟩ research ⟨/dname⟩
    ⟨tel⟩ 7776 ⟨/tel⟩
    ⟨employee⟩
      ⟨name⟩ Sara Lee ⟨/name⟩ ⟨tel⟩ 5554 ⟨/tel⟩
      ⟨tel⟩ 3221 ⟨/tel⟩ ⟨sal⟩ 32 ⟨/sal⟩
    ⟨/employee⟩
    ⟨employee⟩
     ⟨name⟩ Jim Bean ⟨/name⟩ ⟨tel⟩ 1223 ⟨/tel⟩ ⟨sal⟩ 25 ⟨/sal⟩
    ⟨/employee⟩
  ⟨/department⟩
⟨/db⟩
```

A DOM example

Print the names of employees and their telephone numbers.

```
from xml.dom.minidom import parse

source = open("emps.xml", "r")

domtree= parse(source)

for e in domtree.getElementsByTagName("employee"):
    for n in e.getElementsByTagName("name"):
        for c in n.childNodes: print c.data,
    for n in e.getElementsByTagName("tel"):
        for c in n.childNodes: print c.data,
    print "\n",
```

The output ...

Jane Dee 6734
Mary Smith 1432
John Brown
Fred Beans 3221
Kate Smith 1432
Sara Lee 5554 3221
Jim Bean 1223

Note that the data is "ragged".

The preamble

```
from xml.dom.minidom import parse
```

Import the parse function. Python is dynamically typed – the "classes" are generated on the fly.

```
source = open("emps.xml", "r")
```

Usual – open a file in read-only mode.

```
domtree= parse(source)
```

Create the DOM "tree". `domtree` is the root node. We use node methods to navigate the tree

Traversing the tree

```
for e in domtree.getElementsByTagName("employee"):
```

This binds `e` sucessively to all `employee` nodes encountered in a depth-first, left-to-right traversal of the tree.

```
    for n in e.getElementsByTagName("name"):
```

This binds `n` to `name` nodes in a traversal of the subtree of `e`

```
        for c in n.childNodes:  print c.data,
```

Text nodes have their character data in `data`. DOM does not assume that the character data is stored in just one node.

The same thing in SAX?

```
class EmpHandler(xml.sax.handler.ContentHandler):

    def __init__(self):
        self.buffer = ""

    def startElement(self, name, attributes):
        if (name == "tel") or (name == "name"):
            self.buffer = ""


    def endElement(self, name):
        if name == "employee":
            print ""
        elif (name == "name") or (name == "tel"):
            print self.buffer,
            self.buffer = ""

    def characters(self,data):
        self.buffer = self.buffer+data
```

How it works

The class `EmpHandler` inherits from the class `ContentHandler` defined in `xml.sax.handler`. It overwrites the methods of this class so that the code you have written gets called as the sax parser traverses the document. [Classes are partly implemented with smoke and mirrors in Python, but the idea works well.]

The idea is to collect characters whenever we are inside a `name` or `tel` element and print them out when we leave that element.

```
def __init__(self):
    self.buffer = ""
```

The 0-argument constructor that initializes the character buffer.

How it works – continued

```
def startElement(self, name, attributes):
    if (name == "tel") or (name == "name"):
        self.buffer = ""
```

When we enter a `tel` or `name` element, re-initialise the buffer.

```
def characters(self,data):
    self.buffer = self.buffer+data
```

Whenever we encounter character data, append it to the buffer.

How it works – continued

```
def endElement(self, name):
    if name == "employee":
        print ""
    elif (name == "name") or (name == "tel"):
        print self.buffer,
        self.buffer = ""
```

When we leave a `tel` or `name` element print the buffer (and flush it). Print a new-line when we leave an `employee` element

The whole program

```
import xml.sax

class EmpHandler(xml.sax.handler.ContentHandler):
        - - - as above
parser = xml.sax.make_parser()

parser.setContentHandler(EmpHandler())

parser.parse("emps.xml")
```

Create a parser (there may be several ways of doing this); create an instance of the `EmpHandler` class and tell the parser to use it; finally make the parser parse the document.

Does it work?

```
1432   Jane Dee   6734
Mary Smith   1432
John Brown
3221   Fred Beans   3221
Kate Smith   1432
7776   Sara Lee   5554   3221
Jim Bean   1223
```

The problem is that it prints the contents of *every* `tel` element. We have to know when we are "inside" an `employee` element.

The changes needed

We add a flag that is set whenever we are inside a `Employee` element

```
def __init__(self):
    self.buffer = ""
    self.inemp = 0

def startElement(self, name, attributes):
    if name == "employee":
        self.inemp =1
    if (name == "tel") or (name == "name"):
        self.buffer = ""

def endElement(self, name):
    if name == "employee":
        self.inemp = 0
        print ""
    elif ((name == "name") or (name == "tel")) and self.inemp:
```

```
        print self.buffer,
        self.buffer = ""
```

Unfortunately this is still not doing the same as our DOM code. The contents of the `name` and `tel` elements are (with the code above) printed in the order in which they appear in the document. Try change the order of `<name> Jane Dee </name>` and `<tel> 6734 </tel>` in the XML file.

In order to get closer to the DOM code we have to do more buffering.

Further buffering

```
 def __init__(self):
    self.namelist=[]
    self.tellist=[]
    self.buffer = ""

def startElement(self, name, attributes):
    if name == "employee":
self.namelist=[]
self.tellist=[]
    elif (name == "tel") or (name == "name"):
self.buffer = ""

def characters(self,data):
    self.buffer = self.buffer+data
```

```
def endElement(self, name):
    if name == "employee":
        for n in self.namelist:
            print n,
        for n in self.tellist:
            print n,
        print ""
        self.namelist = []
        self.tellist = []
    elif name == "name":
        self.namelist.append(self.buffer)
        self.buffer = ""
    elif name == "tel":
        self.namelist.append(self.buffer)
        self.buffer = ""
```

Style sheets and Query languages

- Style sheets. Intended for "rendering" XML in a presentation format such as HTML. Since HTML is XML, style sheets are query languages. However, they are typically only "tuned" to simple transformations – that is, the structure of the output corresponds to the structure of the input.

  Output of a stylesheet can be something other than XML

  Early style sheets could not "join" data from different sources.
- Query languages. Arguably easier for complex queries – derived from database paradigms. They have a `SELECT ... FROM ... WHERE` (SQL) flavor.

  Query languages have been developed for XML-to XML transformations.

The *big* question: Will we achieve a storage method, evaluation algorithms, and optimization techniques that make query languages work well for large XML "documents"?

XPath and XQuery – reading material

Note: documents on XQuery typically describe XPath too.

- The XQuery specification (impenetrable): `http://www.w3.org/TR/xquery`
- XML Query Use Cases. A set of examples used to "show off" XQuery (or maybe to test implementations). Lots of examples. Much more readable than the standard: `http://www.w3.org/TR/xmlquery-use-cases`
- A nice, straightforward, tutorial: `http://www.brics.dk/~amoeller/XML/querying/`
- An interesting paper showing how XQuery can be typed. Quite readable even if you are not interested in types! `http://homepages.inf.ed.ac.uk/wadler/papers/...` `...` `xquery-tutorial/xquery-tutorial.pdf`

XQuery and XPath

Again, consider the simple database-like query "Find the `names` of `employees` whose `age` is `55`". How do we do this using XQuery? We first discuss XPath.

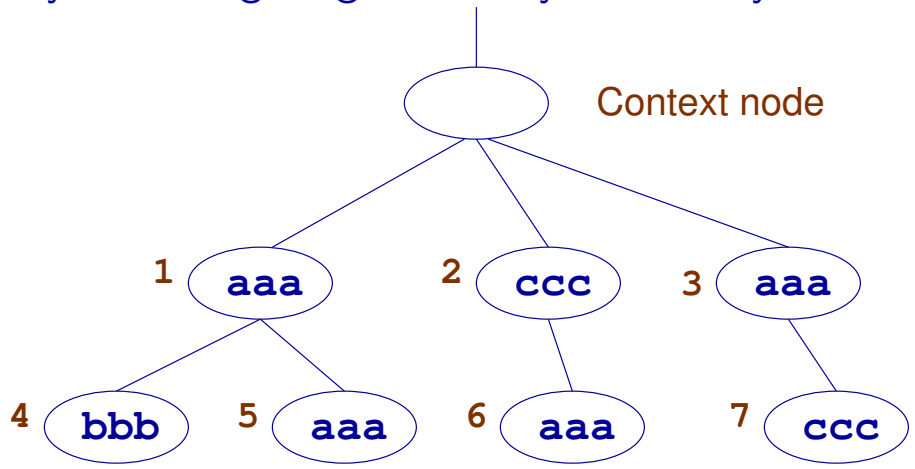
- *XPath* gives us the sets of nodes. In this case it will be a set of `employee` nodes. It binds variables to nodes
- *XQuery* is very like a database query language such as SQL and uses these sets to produce XML (the query result) as output.

Caution! XPath is a relatively complex language. You have the option of expressing certain things, such as selection, conditions either in XPath or elsewhere in the surrounding XQuery.

Caution! XQuery is the current standard query language for XML. It has supplanted other QLs (some nicer than XQuery).

XPath – quick start

Navigation is remarkably like navigating a unix-style directory.



All paths start from some *context node*.

`aaa`	all the child nodes of the context node labeled `aaa` {1,3}
`aaa/bbb`	all the `bbb` children of `aaa` children of the context node {4}
`*/aaa`	all the `aaa` children of *any* child of the context node {5,6}.
`.`	the context node
`/`	the root node

XPath- child axis navigation (cont)

`/doc`	all the `doc` children of the root
`./aaa`	all the `aaa` children of the context node (equivalent to `aaa`)
`text()`	all the text children of the context node
`node()`	all the children of the context node (includes text and attribute nodes)
`..`	parent of the context node
`.//`	the context node and all its descendants
`//`	the root node and all its descendants
`//para`	all the para nodes in the document
`//text()`	all the text nodes in the document
`@font`	the font attribute node of the context node

Predicates

`[2]`	the second child node of the context node
`chapter[5]`	the fifth chapter child of the context node
`[last()]`	the last child node of the context node
`person[tel="12345"]`	the `person` children of the context node that have one or more `tel` children whose string-value is `"1234"` (the string-value is the concatenation of all the text on descendant text nodes)
`person[.//firstname = "Joe"]`	the person children of the context node whose descendants include firstname element with string-value `"Joe"`

From the XPath specification (`$x` is a variable – see later):

> NOTE: If `$x` is bound to a node set then `$x = "foo"` does not mean the same as `not($x != "foo")` .

Unions of Path Expressions

- `employee | consultant` – the union of the `employee` and `consultant` nodes that are children of the context node
- For some reason `person/(employee|consultant)` – as in general regular expressions – is not allowed
- However `person/node()[boolean(employee|consultant)]` is allowed!!

From the XPath specification:

> The boolean function converts its argument to a boolean as follows:
> - a number is true if and only if it is neither positive or negative zero nor NaN
> - a node-set is true if and only if it is non-empty
> - a string is true if and only if its length is non-zero
> - an object of a type other than the four basic types is converted to a boolean in a way that is dependent on that type.

Our Query in XPath

Consider SQL: `SELECT age FROM employee WHERE name = "Joe"`

We can write an XPath expression:

```
//employee[name="Joe"]/age
```

Find all the `employee` nodes under the root. If there is at least one `name` child node whose string-value is `"Joe"`, return the set of all `age` children of the `employee` node.

Or maybe

```
//employee[//name="Joe"]/age
```

Find all the `employee` nodes under the root. If there is at least one `name` *descendant* node whose string-value is `"Joe"`, return the set of all `age` *descendant* nodes of the `employee` node.

Why isn't XPath a proper (database) query language?

It doesn't return XML – just a set of nodes.

It can't do complex queries invoking joins.

We'll turn to XML shortly, but there's a bit more on XPath.

XPath – navigation axes

In Xpath there are several navigation *axes*. The *full* syntax of XPath specifies an axis after the `/`. E.g.,

`ancestor::employee`: all the `employee` nodes *directly above* the context node

`following-sibling::age`: all the `age` nodes that are *siblings* of the context node and to the *right* of it.

`following-sibling::employee/descendant::age`: all the `age` nodes *somewhere below* any `employee` node that is a *sibling* of the context node and to the *right* of it.

`/descendant::name/ancestor::employee`: Same as `//name/ancestor::employee` or `//employee[boolean(.//name)]`

So XPath consists of a series of navigation steps. Each step is of the form: *axis*::*node test*[*predicate list*]

Navigation steps can be concatenated with a `/`

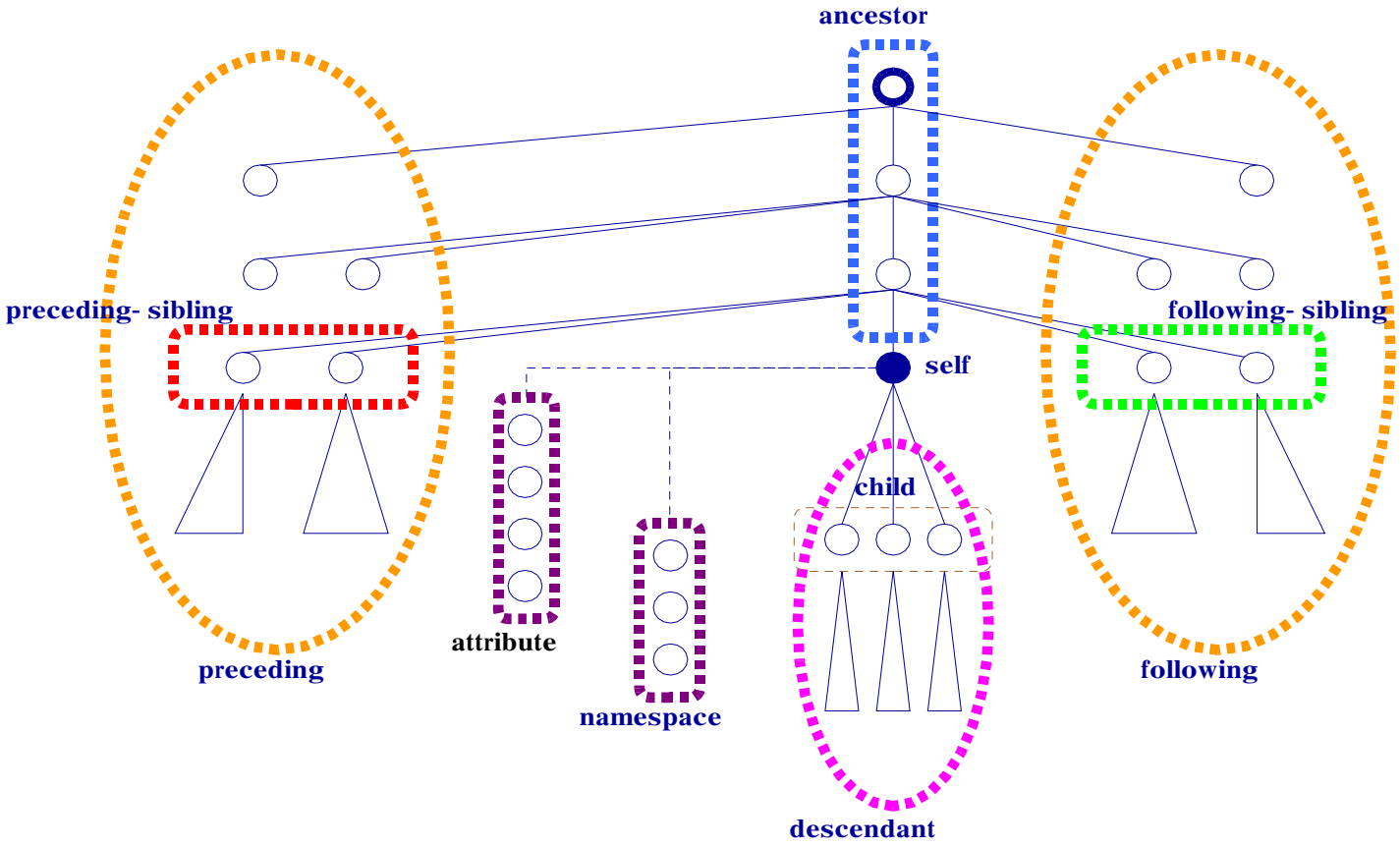
If the path starts with `/` or `//`, start at root. Otherwise start at context node.

The following are abbreviations/shortcuts.

- no axis means child
- `//` means `/descendant-or-self::`

The full list of axes is: `ancestor`, `ancestor-or-self`, `attribute`, `child`, `descendant`, `descendant-or-self`, `following`, `following-sibling`, `namespace`, `parent`, `preceding`, `preceding-sibling`, `self`.

The XPath axes



XQuery

XPath is central to XQuery as well as to style sheets. In addition to XPath, XQuery provides:

- XML "glue" that turns XPath node sets back into XML.
- Variables that communicate between XPath and XQuery.
- Programming structures that allow us to do things like joins, aggregates and more sophisticated conditions than those in XPath.

XQuery is somewhat more complicated than SQL and is best learnt after SQL. Also, there is nothing as clean as the relational algebra on which to base optiisation.

Document Type Descriptors

XML has gained acceptance as a standard for data interchange. There are now hundreds of published DTDs. DTDs are described in the XML standard and in most XML tutorials.

- A Document Type Descriptor (DTD) constrains the structure of an XML document.
- There is some relationship between a DTD and a database schema or a type/class declaration of a program, but it is not close – hence the need for additional "typing" systems, such as XML-Schema.
- A DTD is a syntactic specification. Its connection with any "conceptual" model may be quite remote.

Note: DTDs have been subsumed by XML-Schema, another W3C standard for constraining XML. XML-Schema is verbose and much more complicated than DTDs. To understand it you need DTDs as a starting point in any case!

Example: The Address Book

⟨person⟩	
⟨name⟩ McNeil, John ⟨/name⟩	must exist
⟨greet⟩ Dr. John McNiel ⟨/greet⟩	optional
⟨addr⟩ 1234 Huron Street ⟨/addr⟩	as many address lines as needed
⟨addr⟩ Rome, OH 98765 ⟨/addr⟩	
⟨tel⟩ (321) 786 2543 ⟨/tel⟩	0 or more tel and faxes in any order
⟨fax⟩ (123) 456 7890 ⟨/fax⟩	
⟨tel⟩ (321) 198 7654 ⟨/tel⟩	
⟨email⟩ jm@abc.com ⟨/email⟩	0 or more email addresses
⟨/person⟩	

Specifying the Structure

`name`	to specify a `name` element
`greet?`	to specify an optional (0 or 1) `greet` elements
`name,greet?`	to specify a name followed by an optional `greet`
`addr*`	to specify 0 or more `address` lines
`tel | fax`	a `tel` or a `fax` element
`(tel | fax)*`	0 or more repeats of `tel` or `fax`
`email*`	0 or more email elements

Specifying the structure (cont)

So the whole structure of a person entry is specified by

```
name, greet?, addr*, (tel | fax)*, email*
```

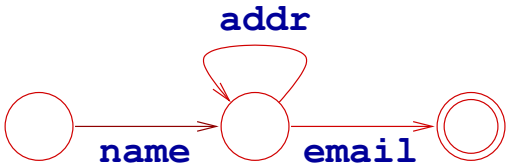
This is a *regular expression* in slightly unusual syntax. Why is it important?

Regular Expressions

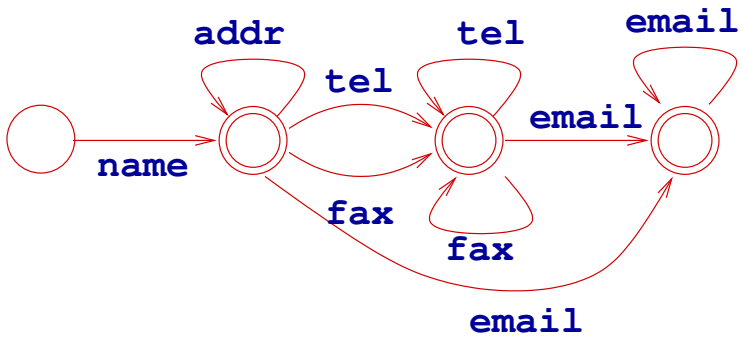
One could imagine more complicated and less complicated specifications for the structure of the content of an XML element.

Regular expressions are "rich enough" (arguable) and easy to parse with a DFA. Examples:

`name,addr*,email`



`name,addr*,(tel|fax)*,email*`



Try adding in `greet?`. The DFA can get large!

A DTD for the address book

```
⟨!DOCTYPE address [
⟨!ELEMENT addressbook (person*)⟩
⟨!ELEMENT person (name, greet?, addr*, (fax|tel)*, email*)⟩
⟨!ELEMENT name (#PCDATA)⟩
⟨!ELEMENT greet (#PCDATA)⟩
⟨!ELEMENT addr (#PCDATA)⟩
⟨!ELEMENT tel (#PCDATA)⟩
⟨!ELEMENT fax (#PCDATA)⟩
⟨!ELEMENT email (#PCDATA)⟩
]⟩
```

Our "database" revisited

Recall:

- Projects have titles, budgets, managers, ...
- Employees have names, employee ids, ages, ...

DTDs for the relational DB

Tuples intermixed

```
⟨!DOCTYPE db [
⟨!ELEMENT  db (project | employee)*⟩
⟨!ELEMENT  project (title, budget, managedBy)⟩
⟨!ELEMENT  employee (name, empid, age)⟩
⟨!ELEMENT  title #PCDATA⟩
...
]⟩
```

Tables grouped:

```
⟨!DOCTYPE db [
⟨!ELEMENT  db (projects,employees)⟩
⟨!ELEMENT  projects (project*)⟩
⟨!ELEMENT  employees (employee*)⟩
⟨!ELEMENT  project (title, budget, manager)⟩
⟨!ELEMENT  employee (name, empid, age)⟩
...
]⟩
```

Tuples unmarked:

```
⟨!DOCTYPE db [
⟨!ELEMENT  db )((name, empid, age)|(title, budget, manager))*)⟩
...
]⟩
```

Recursive DTDs

```
⟨!DOCTYPE  genealogy [
⟨!ELEMENT  genealogy (person*)⟩
⟨!ELEMENT  person (
name,
dateOfBirth,
person,          // mother
person           // father
)⟩
]⟩
```

What is the problem with this?

Another try ...

```
⟨!DOCTYPE genealogy [
⟨!ELEMENT  genealogy (person*)⟩
⟨!ELEMENT  person (
name,
dateOfBirth,
person?,          // mother
person?           // father
)⟩
]⟩
```

What is now the problem with this?

Some things are hard to specify

Each employee element is to contain name, age and empid elements in some order.

```
⟨!ELEMENT employee (
(name, age, empid)
| (age, empid, name)
| (empid, name, age)
...
)⟩
```

Suppose there were many more fields!

This is a fundamental problem in trying to combine XML schemas with simple relational schemas. Research needed!

This is what can happen

⟨!ELEMENT PARTNER (NAME?, ONETIME?, PARTNRID?, PARTNRTYPE?,
SYNCIND?,ACTIVE?, CURRENCY?, DESCRIPTN?, DUNSNUMBER?, GLENTITYS?,
NAME*, PARENTID?, PARTNRIDX?, PARTNRRATG?, PARTNRROLE?, PAYMETHOD?,
TAXEXEMPT?, TAXID?, TERMID?, USERAREA?, ADDRESS*, CONTACT*)⟩

Cited from oagis_segments.dtd (one of the files in Novell Developer Kit
http://developer.novell.com/ndk/indexexe.htm)

⟨PARTNER⟩⟨NAME⟩ Dewey Cheatham ⟨/NAME⟩⟨/PARTNER⟩

Question: Which NAME is it?

Specifying attributes in the DTD

```
⟨!ELEMENT height (#PCDATA)⟩
⟨!ATTLIST height
      dimension CDATA #REQUIRED
      accuracy CDATA #IMPLIED⟩
```

The dimension attribute is required; the accuracy attribute is optional.

CDATA is the "type" the attribute – it means string.

Specifying ID and IDREF attributes

```
⟨!DOCTYPE family [
⟨!ELEMENT  family (person)*⟩
⟨!ELEMENT  person (name)⟩
⟨!ELEMENT  name (#PCDATA)⟩
⟨!ATTLIST person
id       ID     #REQUIRED
mother   IDREF  #IMPLIED
father   IDREF  #IMPLIED
children IDREFS #IMPLIED⟩
]⟩
```

Consistency of ID and IDREF attribute values

- If an attribute is declared as `ID` the associated values must all be *distinct* (no confusion).
- If an attribute is declared as `IDREF` the associated value *must exist* as the value of some `ID` attribute (no "dangling pointers").
- Similarly for all the values of an `IDREFS` attribute
- `ID` and `IDREF` attributes are *not typed*.

Connecting the document with its DTD

- In line:
  ⟨?xml version="1.0"?⟩
  ⟨!DOCTYPE db [⟨!ELEMENT . . . ⟩. . .]⟩
  ⟨db⟩. . .⟨/db⟩
- Another file:
  ⟨!DOCTYPE db SYSTEM "schema.dtd"⟩
- A URL:
  ⟨!DOCTYPE db SYSTEM "http://www.schemaauthority.com/schema.dtd"⟩

Well-formed and Valid Documents

- Well-formed applies to any document (with or without a DTD): proper nesting of tags and unique attributes
- Valid specifies that the document conforms to the DTD: conforms to regular expression grammar, types of attributes correct, and constraints on references satisfied

DTDs v.s Schemas or Types

- By database or programming language standards DTDs are rather weak specifications.
  - Only one base type – PCDATA
  - No useful abstractions e.g., sets
  - IDREFs are untyped. You point to something, but you dont know what!
  - No constraints e.g., child is inverse of parent
  - No methods
  - Tag definitions are global
- On the other hand DB schemas don't allow you to specify the linear structure of documents.

XML Schema, among other things, attempts to capture both worlds. Not clear that it succeeds.

Summary

- XML is a new data format. Its main virtues are widespread acceptance, its ability to represent structured text, and the (important) ability to handle semistructured data (data without a pre-assigned type.)
- DTDs provide some useful syntactic constraints on documents. As schemas they are weak
- How to store large XML documents.
- How to query them efficiently.
- How to map between XML and other representations.
- How to make XML schemas work like database schemas and programming language types. Current APIs and query languages make little or no use of DTDs.

Review

- XML
  - Basic structure and terminology
  - Well-formed documents
- DOM and SAX
- XPath
  - What XPath expressions produce
  - Basic form of navigation.
  - Axes and general navigation.
- DDTs
  - Specifying child order (regular expressions)
  - Specifying attributes
  - Valid documents