# Applied Databases

5. XML

November 15, 2010

## XML – Outline

- Background: documents (SGML/HTML) and databases
- XML Basics
- XPath
- 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`

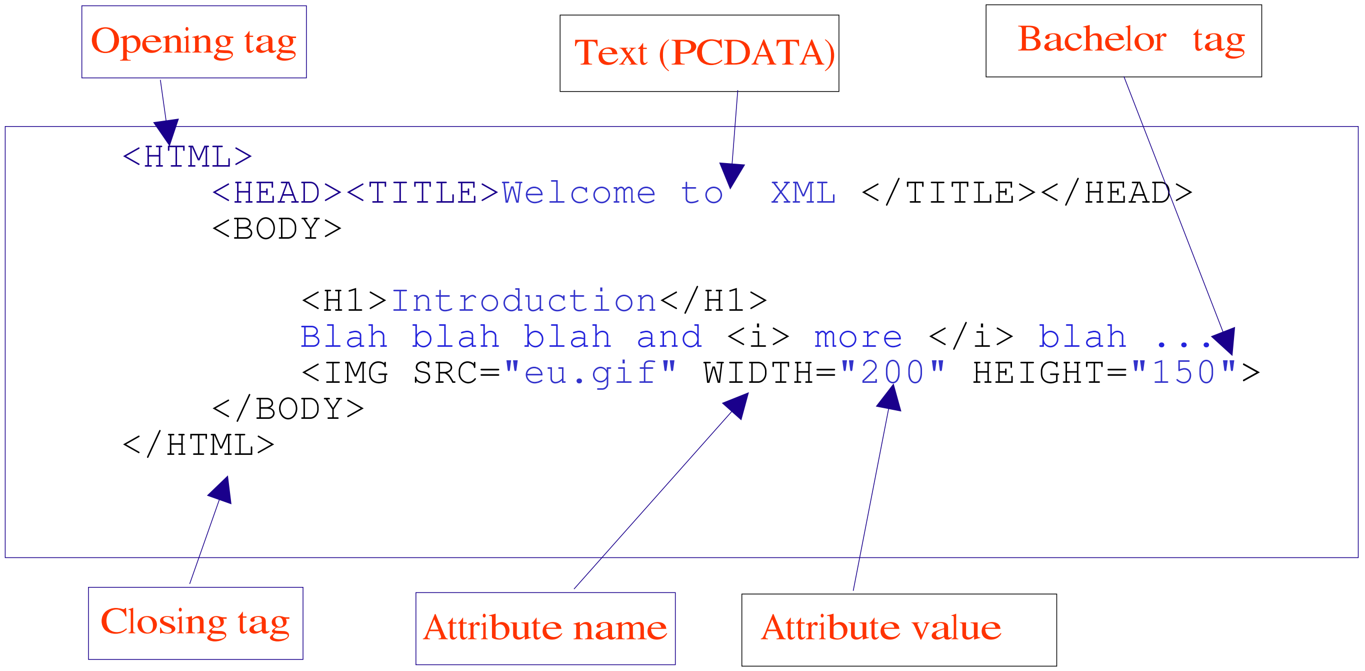
## Documents vs. Databases

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....



```
<HTML>
    <HEAD><TITLE>Welcome to  XML </TITLE></HEAD>
    <BODY>

        <H1>Introduction</H1>
        Blah blah blah and <i> more </i> blah ...
        <IMG SRC="eu.gif" WIDTH="200" HEIGHT="150">
    </BODY>
</HTML>
```

## The thin line...

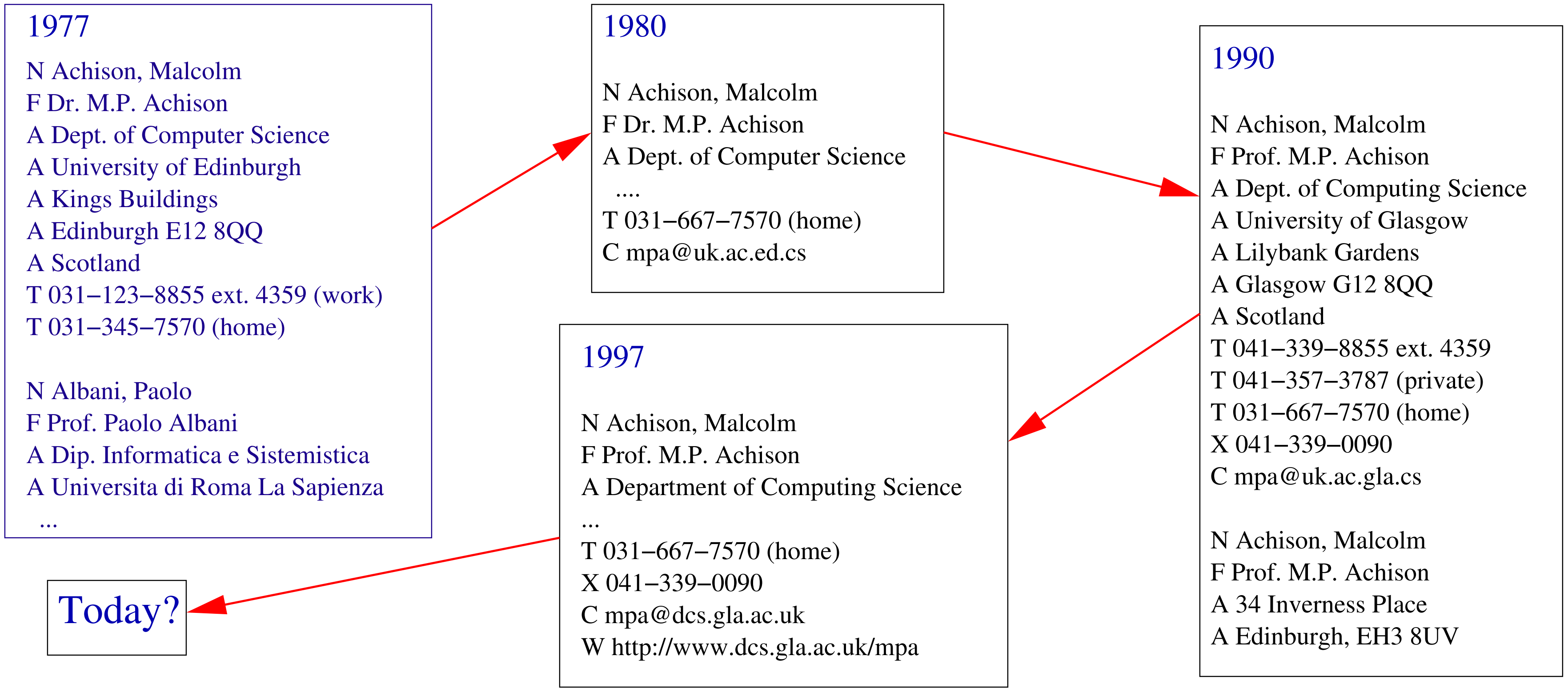
... between document formats and data formats. Much of the world's data – especially scientific data – is held in pre-XML *data formats*.

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: 25 years of my address book



1977

N Achison, Malcolm
F Dr. M.P. Achison
A Dept. of Computer Science
A University of Edinburgh
A Kings Buildings
A Edinburgh E12 8QQ
A Scotland
T 031–123–8855 ext. 4359 (work)
T 031–345–7570 (home)

N Albani, Paolo
F Prof. Paolo Albani
A Dip. Informatica e Sistemistica
A Universita di Roma La Sapienza
...

1980

N Achison, Malcolm
F Dr. M.P. Achison
A Dept. of Computer Science
....
T 031–667–7570 (home)
C mpa@uk.ac.ed.cs

1990

N Achison, Malcolm
F Prof. M.P. Achison
A Dept. of Computing Science
A University of Glasgow
A Lilybank Gardens
A Glasgow G12 8QQ
A Scotland
T 041–339–8855 ext. 4359
T 041–357–3787 (private)
T 031–667–7570 (home)
X 041–339–0090
C mpa@uk.ac.gla.cs

N Achison, Malcolm
F Prof. M.P. Achison
A 34 Inverness Place
A Edinburgh, EH3 8UV

1997

N Achison, Malcolm
F Prof. M.P. Achison
A Department of Computing Science
...
T 031–667–7570 (home)
X 041–339–0090
C mpa@dcs.gla.ac.uk
W http://www.dcs.gla.ac.uk/mpa

Today?

# 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

- 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 spec. 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.

$\langle$`title`$\rangle$`The Big Sleep`$\langle$`/title`$\rangle$     $\langle$`year`$\rangle$1935$\langle$`/year`$\rangle$ — 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

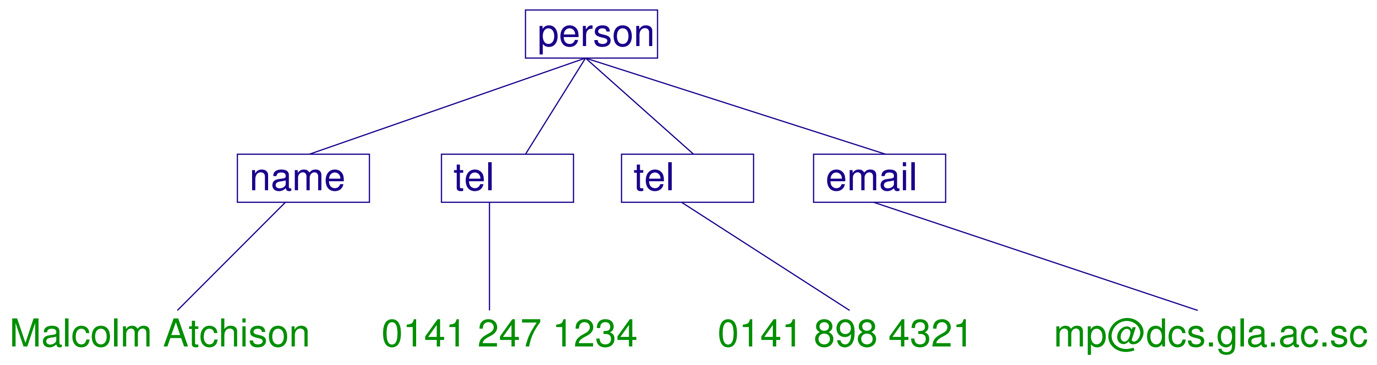
The segment of an XML document between an opening and a corresponding closing tag is called an *element*.

```
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 does 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 intermixed

```
⟨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

Its 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 simpler 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.

## 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. (Early stylesheets couldn’t do joins)
- Query languages. More expressive – derived from database paradigms. They have a `SELECT ... FROM ... WHERE` (SQL) flavor.

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

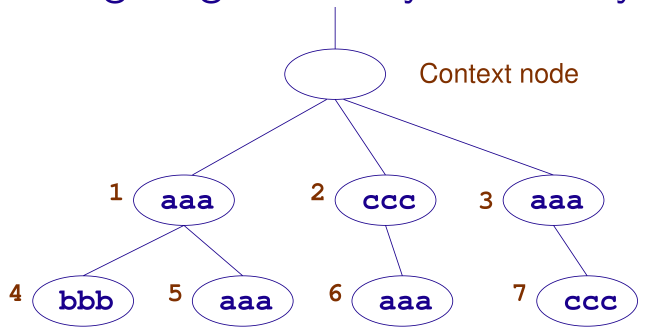
## XPath

Many XML query languages and style sheets need some method of finding nodes in an XML tree. XPath is such a language. It is used, for example in XQuery to generate sets of nodes, which can then be used in XQuery in a similar manner to the way in which relations (sets of tuples) are used in ar relational query language like SQL.

XPath, like lots of things associated with XML, is quite complicated. We shall cover just the basics.

# 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.

## A Query in XPath

Consider: `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 can turn it into XML using XQuery, 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 /

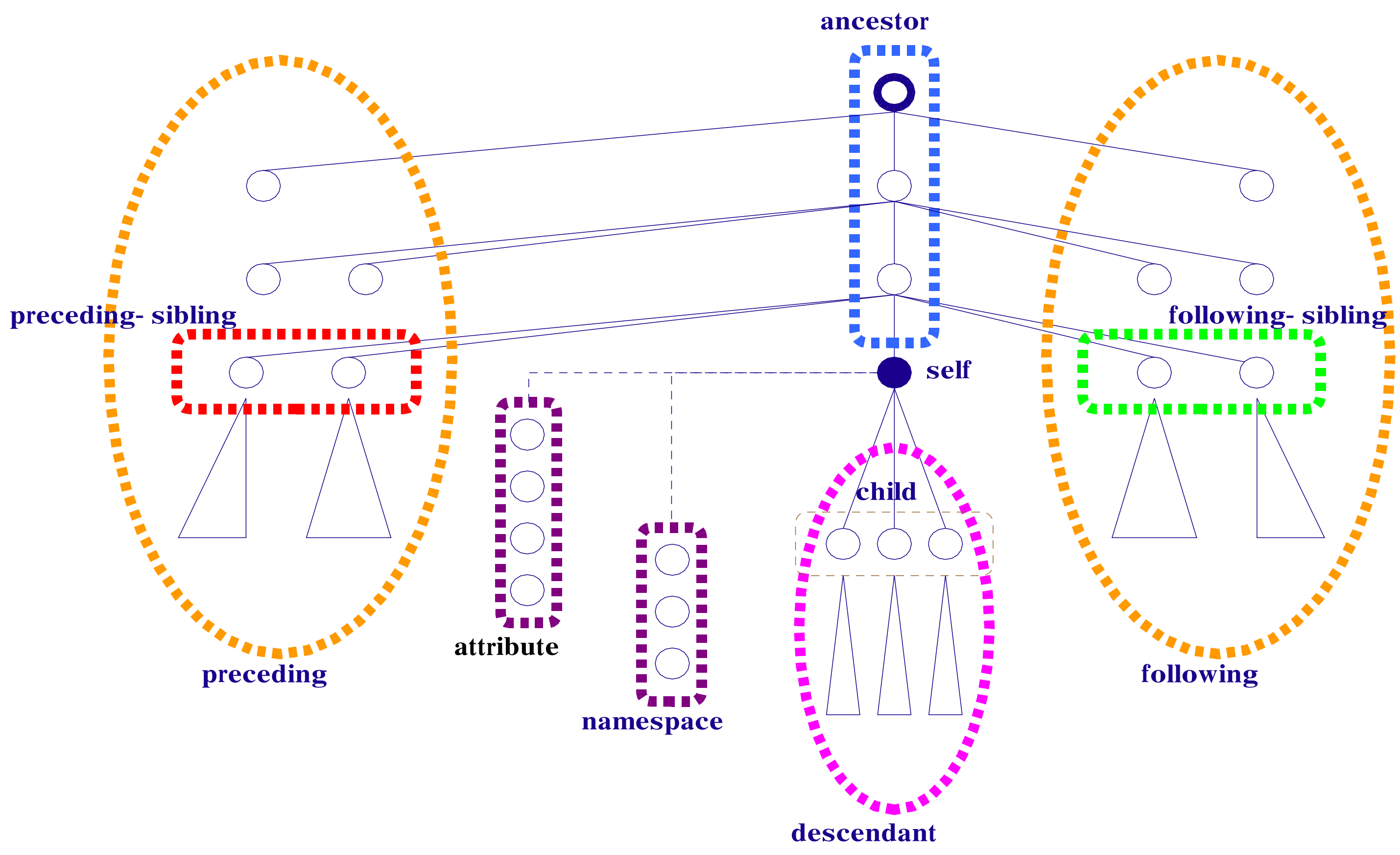
If the path starts with / or //, start at root. Otherwise start at context node.

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



## 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.

## 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*
```

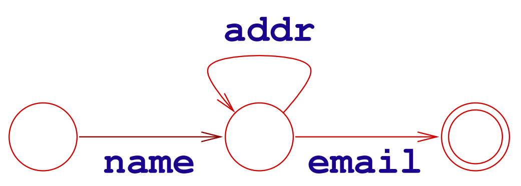
This is a *regular expression* in slightly unusual syntax. Why is it important?

# Regular Expressions

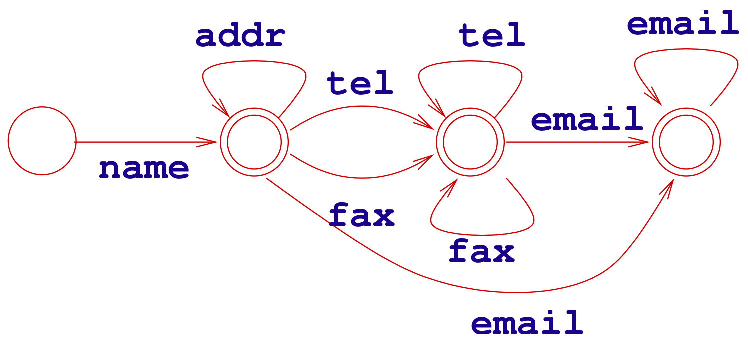
One could imagine more complicated and less complicated specifications for the structure of the content of an XML element.

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⟩ Ben Franklin ⟨/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

IDs and IDREFs act as internal pointers

```
⟨!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 (but recent research is developing query languages that do treat DTDs as types)

## Review

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

## What we haven't covered in XML

Lots, but most notably XQuery – the XML query language of choice, and XML-Schema. Will they replace database technology?

But there's also lots we haven't covered in databases

- Concurrency
- Recovery
- More query optimisation
- Data integration

All this and more in future database/XML courses

**THE END**