Weeks 7 and 9: XML Data and Query Processing

Semistructured Data, HTML
XML and DTDs
XPath, XQuery

Hypertext

→Most human knowledge exists today in document format (books etc.)

→Need technologies that store, and retrieve such ***unstructured*** data same way as ***structured*** data!

→From text to ***hypertext***: add ***annotations*** (***tags***, ***markups***) in a document, to be used for indexing.

→Old idea (Vannevar Bush,*Atlantic Monthly*, 1945) **http://www.theatlantic.com/doc/194507/bush.**

→Markup languages exist since 1970 -- SGML,

→Great tutorial: **http://www.brics.dk/~amoeller/XML/**

HTML: HyperText Markup Language

- →**Motivation**: Exchange data on the Internet; documents are published by servers and are presented by clients (browsers).
- →HTML was created by Tim Berners-Lee and Robert Caillau at CERN in 1991; they wanted to keep track of experimental data.
- →HTML describes only the **logical structure** of documents:
  - ✓browsers are free to interpret markup tags as they please;
  - ✓the document even makes sense if the tags are ignored.

HTML Data

- →**An HTML document to be displayed on the Web:**

```
<dt>Name: John Doe
    <dd>Id: s111111111
    <dd>Address:   <ul>
       <li>Number: 123</li>
       <li>Street: Main</li>
                   </ul>
</dt>
<dt>Name: Joe Public
     <dd>Id:  s222222222
    …
</dt>
```

HTML does not distinguish between attributes and values

What's Great About HTML?

- Many document formats are bulky: author controls precise layout, formatting details stored with content.
- In comparison, **HTML is light-weight**: author sacrifices control for compactness, only content and logical structure is represented.
- Sizes of documents containing just the text "Hello World!":

PostScript	hello.ps	11,274 bytes
PDF	hello.pdf	4,915 bytes
MS Word	hello.doc	19,456 bytes
HTML	hello.html	44 bytes

From Logical to Physical Structure

- Originally, HTML described logical structure:
  - ✓h2: "this is a header at level 2";
  - ✓em: "this text should be emphasized";
  - ✓ul: "this is a list of items".
- Quickly, users wanted more control:
  - ✓"this header is centered and written in Times-Roman in size 28pt","italicize text";
- The early hack for commercial pages was to make everything a huge image:

HTML	hello.html	44 bytes
GIF	hello.gif	32,700 bytes

- HTML developers kept adding layout tags.

Cascading Style Sheets (CSS)

- → Specify physical properties (layout) of HTML tags; are (usually) written in separate files; can be shared for many HTML documents.
- → There are many advantages:
  - ✓ logical and physical properties may be separated;
  - ✓ document groups can have consistent looks;
  - ✓ the look can easily be changed.
- → A CSS stylesheet works by:
  - ✓ allowing >50 properties to be defined for each tag;
  - ✓ definitions for a tag may depend on its context;
  - ✓ undefined properties are inherited;
  - ✓ normal HTML corresponds to default properties.
- → Using stylesheets, all tags become logical.

Why XML?

- → XML is a standard for data exchange that is taking over the World.
- → All major database products have been retrofitted with facilities to store and construct XML documents.
- → There are already database products that are ***specifically designed*** to work with XML documents rather than relational or object-oriented data.
- → XML is closely related to object-oriented and so-called ***semistructured*** data.

Semistructured Data

- To make the HTML student list (earlier example) suitable for machine consumption on the Web, it should have these characteristics:
  - ✓Be ***object-like***;
  - ✓Be ***schema-less*** — no guarantee it conforms exactly to any schema, but different objects share some commonalities;
  - ✓Be ***self-describing*** — some schema-like information, e.g., attribute names, is part of the data itself.
- Data with these characteristics are referred to as ***semistructured***.

What is Self-Describing Data?

Non-self-describing first (relational, object-oriented):

***Data part*:**

```
(#123, ["Students",
  {["John", s111111111, [123,"Main St"]],
   ["Joe", s222222222, [321, "Pine St"]] } ] )
```

***Schema part*:**

```
PersonList[ ListName: String,
  Contents: [ Name: String,Id: String,
  Address: [Number: Integer, Street: String] ] ]
```

Self-Describing Data

→ Attribute names embedded in the data itself, but are distinguished from values.

→ Doesn't need schema to figure out what is what (but schema might be useful nonetheless)

```
(#12345, [ListName: "Students",
 Contents:{ [Name: "John Doe",
               Id:  "s111111111",
          Address: [Num: 123,
                    Str: "Main St."] ] ,
            [Name: "Joe Public",
               Id:  "s222222222",
          Address:[Num:321,Str:"Pine  St."]  ]
  }   ]  )
```


XML – The De Facto Standard for Semistructured Data

→ XML, the e**X**tensible **M**arkup **L**anguage – suitable for semistructured data and has become a standard:

- ✓ Easy to describe object-like data;
- ✓ Self-describing;
- ✓ Doesn't require a schema – but can use one.

→ We will study:

- ✓ **DTD**s – an early technique for specifying XML schemas;
- ✓ Query and transformation languages – **XPath** and **XQuery**.

Overview of XML

- Like HTML, but any number of different tags can be used (up to the document author) – hence extensible.
- Unlike HTML, no semantics behind the tags:
  - For instance, HTML's <table>…</table> means: render content as table; in XML doesn't mean anything special;
  - Some semantics can be specified using XML Schema (types); some using stylesheets (browser rendering)
- Unlike HTML, XML is intolerant to bugs:
  - Browsers will render buggy HTML pages;
  - *XML processors* are not supposed to process buggy XML documents.

a brief segway

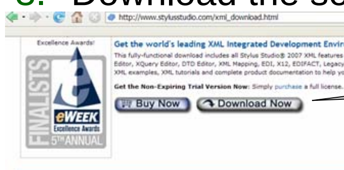
- Stylus Studio is a development tool which can be used to create and query XML.
- Data Direct Technologies has allowed us to use it for the duration of this course.
- Take the time to download Stylus Studio to your PC.
- It is a valuable learning tool and can be used to verify your assignments.
- It is fairly easy to do – the following slides will guide you through the process.

Downloading Stylus Studio to your PC

1. Go to the Stylus Studio Web page at http://www.stylusstudio.com/
2. From there, navigate to the download page.

Click on: Download

3. Download the software



Select: Download Now

4. Install it on your PC.

Running Stylus Studio for the first time

1. Click on the Stylus Studio icon.
2. The registration screen will appear. Fill in the necessary information.
3. You will be required to enter the following registration code:

4. **You are encouraged to download it early and start using XML. Have fun!!**

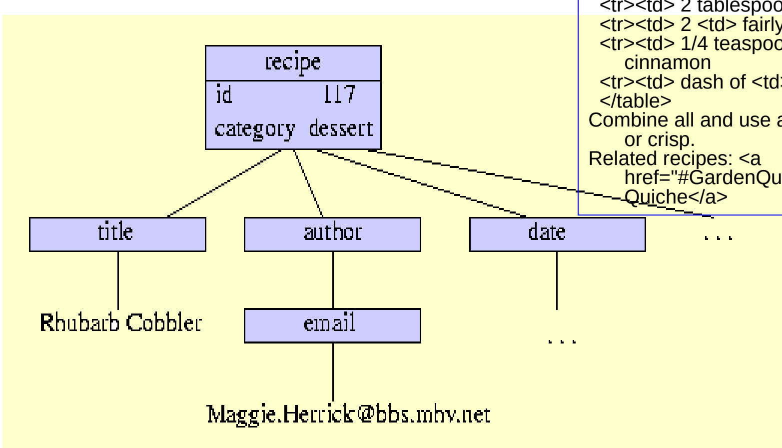
Conceptual View of XML

- An XML document is (isomorphic to) an ordered, labeled tree.
- ***Character data leaf nodes*** contain the actual data (text strings); usually, character data nodes must be non-empty and non-adjacent to other character data nodes
- ***Elements nodes***, are each labeled with
  - a name (often called the ***element type***), and
  - a set of ***attributes*****,** each consisting of a name and a value,

and these nodes can have child nodes

Domain-Specific Markups

<h1>Rhubarb Cobbler</h1>
<h2>Maggie.Herrick@bbs.mhv.net</h2>
<h3>Wed, 14 Jun 95</h3>
Rhubarb Cobbler made with bananas as the main sweetener. It was delicious. Basicly it was
<table>
<tr><td> 2 1/2 cups <td> diced rhubarb
<tr><td> 2 tablespoons <td> sugar
<tr><td> 2 <td> fairly ripe bananas
<tr><td> 1/4 teaspoon <td> cinnamon
<tr><td> dash of <td> nutmeg
</table>
Combine all and use as cobbler, pie, or crisp.
Related recipes: Garden Quiche



An XML Document

```
<?xml  version="1.0" ?>

<PersonList Type="Student"  Date="2002-02-02" >
    <Title  Value="Student List" />
    <Person>
    … … …
    </Person>
    <Person>
    … … …
    </Person>
</PersonList>
```

elements

Root element

→ Elements are nested.

→ Root element contains all others.

More Terminology

```
<Person  Name = "John"  Id = "s111111111">
  John is a nice fellow
   <Address>
      <Number>21</Number>
      <Street>Main St.</Street>
   </Address>
   … … …
</Person>
```

Content of Person

Well-formed XML Documents

- → Must have a ***root element***.
- → Every ***opening tag*** has a matching ***closing tag***.
- → Elements must be ***properly nested***
  - <foo><bar></foo></bar> is a no-no
- → An ***attribute*** name can occur ***at most once*** in an opening tag. If it occurs,
  - ✓ It must have an explicitly specified value (Boolean attrs are not allowed);
  - ✓ The value must be quoted (with " or ').
- → XML processors are not supposed to try and fix ill-formed documents (unlike HTML browsers).

Identifiers and References with Attributes

An attribute can be declared to have type:

- → ***ID***: unique identifier of an element; if attr1 and attr2 are both of type ID, then it is illegal to have <something attr1="*abc*"> … <somethingelse attr2="*abc*"> within the same document
- → ***IDREF***: references a unique element with matching ID attribute; if attr1 has type ID and attr2 has type IDREF then we can have <something attr1="*abc*"> … <somethingelse attr2="*abc*">
- → ***IDREFS*** – a list of references, if attr1 is ID and attr2 is IDREFS, then we can have <something attr1="*abc*"> … <something1 attr1="*cde*">…<something2 attr2="*abc cde*">

Report Document with Cross-Refs

```
<?xml version="1.0"  ?>
<Report  Date="2002-12-12">
   <Students>
      <Student  StudId="s111111111">
         <Name><First>John</First><Last>Doe</Last></Name>
  <Status>U2</Status>
         <CrsTaken  CrsCode="CS308"  Semester="F1997" />
         <CrsTaken  CrsCode="MAT123"  Semester="F1997" />
      </Student>
      <Student  StudId="s666666666">
         <Name><First>Joe</First><Last>Public</Last></Name>
  <Status>U3</Status>
         <CrsTaken  CrsCode="CS308"  Semester="F1994" />
         <CrsTaken  CrsCode="MAT123"  Semester="F1997" />
      </Student>
      <Student  StudId="s987654321">
         <Name><First>Bart</First><Last>Simpson</Last></Name>
  <Status>U4</Status>
         <CrsTaken  CrsCode="CS308"  Semester="F1994" />
      </Student>
   </Students>
   ……  continued … …
```


Report Document (cont'd.)

```
   <Classes>
     <Class>
        <CrsCode>CS308</CrsCode> <Semester>F1994</Semester>
        <ClassRoster  Members="s666666666  s987654321" />
     </Class>
     <Class>
        <CrsCode>CS308</CrsCode> <Semester>F1997</Semester>
        <ClassRoster  Members="s111111111" />
     </Class>
     <Class>
        <CrsCode>MAT123</CrsCode> <Semester>F1997</Semester>
        <ClassRoster  Members="s111111111  s666666666" />
     </Class>
  </Classes>
……  continued … …
```

Report Document cont'd

```
  <Courses>
    <Course CrsCode = "CS308" >
      <CrsName>Market Analysis</CrsName>
    </Course>
    <Course CrsCode = "MAT123" >
      <CrsName>Market Analysis</CrsName>
    </Course>
  </Courses>
</Report>
```


XML Namespaces

- A mechanism to prevent name clashes, like scoping rules.
- Namespace declaration
  - *Namespace* – a symbol, typically a URL;
  - *Prefix* – an abbreviation of the namespace;
  - Actual name (element or attribute) – *prefix*:*name*
  - Declarations/prefixes behave like a begin/end.

Example

```
<item  xmlns =
          http://www.acmeinc.com/jp#supplies
   xmlns:toy ="http://www.acmeinc.com/jp#toys">
        <name>backpack</name>
        <feature>

     <toy:item><toy:name>cyberpet</toy:name>
     </toy:item>
        </feature>
   </item>
```

Default namespace

reserved keyword

toy *namespace*

More Namespaces

→ Scopes of declarations are color-coded:

```
<item xmlns="http://www.foo.org/abc"
            xmlns:cde="http://www.bar.com/cde">
   <name>…</name>
   <feature>
      <cde:item><cde:name>…</cde:name></cde:item>
   </feature>
        <item  xmlns="http://www.foobar.org/"
                 xmlns:cde="http://www.foobar.org/cde" >
               <name>…</name>
               <cde:name>…</cde:name>
        </item>
   </item>
```

New default; overshadows old default

Redeclaration of cde*; overshadows old declaration*

Namespaces (cont'd)

- xmlns="http://foo.com/bar" ***doesn't*** mean there is a document at this URL: using URLs is just a convention; a namespace is just an identifier.
- Namespaces aren't part of XML 1.0, but all XML processors understand this feature now
- A number of prefixes have become "standard" and some XML processors might understand them without any declaration. E.g.,
  - ✓xs for http://www.w3.org/2001/XMLSchema
  - ✓xsl for http://www.w3.org/1999/XSL/Transform
  - ✓Etc.

Document Type Definition (DTD)

- A ***DTD*** is a grammar specification for an XML document – you can think of it as a schema.
- DTDs are optional – don't need to be specified; if specified, a DTD can be part of the document (at the top); or it can be given as a URL
- A document that conforms (i.e., parses) w.r.t. its DTD is said to be ***valid***.
- XML processors are ***not required to check validity***, even if DTD is specified; but they are required to test well-formedness.

Attaching a DTD to a Document

- DTD specified as part of a document:

```
<?xml  version="1.0" ?>
<!DOCTYPE  Report [
    … DTD Report spec …
]>
<Report> … … … </Report>
```

- DTD can also be specified as a standalone thing

```
<?xml  version="1.0" ?>
<!DOCTYPE  Report
  "http://foo.org/Report.dtd">
<Report> … … … </Report>
```


Example DTD Constructs

- <!ELEMENT *elt-name*
  (…*contents*…)/EMPTY/ANY >
- <!ATTLIST *elt-name* *attr-name*
  CDATA/ID/IDREF/IDREFS
  #IMPLIED/#REQUIRED
  >

Element's contents

An attr for elt

Type of attribute

Optional/mandatory

- Can define other things, like macros (called ***entities*** in XML jargon)

DTD Language

→ ***<!DOCTYPE root-element [doctype-declaration...]>*** — determines name of root element and contains document type declarations

→ ***<!ELEMENT element-name content-model>*** — associates a content model to every element

→ Content models:

- ✓ **EMPTY:** no content is allowed
- ✓ **ANY:** any content is allowed
- ✓ **(#PCDATA|element-name|...)*:** "mixed content", arbitrary sequence of character data and listed elements;
- ✓ Deterministic regular expression (cont'd).

DTD Language: Regular Expressions

→ **Deterministic regular expression over element names**: sequence of elements matching the expression

+ choice: (...|...|...)
+ sequence: (...,...,...)
+ optional: ...?
+ zero or more: ...*
+ one or more: ...+

DTD Language: Attributes

- →**<!ATTLIST element-name attr-name attr-type attr-default ...>** — declares which attributes are allowed or required in which elements
- →Attribute types:
  - ✓**CDATA**: any value is allowed (the default)
  - ✓**(value|...):** enumeration of allowed values
  - ✓**ID, IDREF, IDREFS**: ID attribute values must be unique (contain "element identity"), IDREF attribute values must match some ID (reference to an element)
  - ✓ENTITY, ENTITIES, NMTOKEN, NMTOKENS, NOTATION: just forget these...

DTD Language: Attribute Defaults

- →**#REQUIRED**: the attribute must be explicitly provided.
- →**#IMPLIED**: attribute is optional, no default provided.
- →**"value"**: if not explicitly provided, this value is inserted by default.
- →**#FIXED** "value": as above, but only this value is allowed.

DTD Example

```
<!DOCTYPE  Report  [
   <!ELEMENT  Report  (Students, Classes, Courses)>
   <!ELEMENT  Students  (Student*)>
   <!ELEMENT  Classes  (Class*)>
   <!ELEMENT  Courses  (Course*)>
   <!ELEMENT  Student  (Name, Status, CrsTaken*)>
   <!ELEMENT  Name  (First,Last)>
   <!ELEMENT  First  (#PCDATA)>
   … … …
   <!ELEMENT  CrsTaken  EMPTY>
   <!ELEMENT  Class (CrsCode,Semester,ClassRoster)>
   <!ELEMENT  Course  (CrsName)>
   … … …
   <!ATTLIST  Report  Date  CDATA #IMPLIED>
   <!ATTLIST  Student  StudId  ID  #REQUIRED>
   <!ATTLIST  Course  CrsCode  ID  #REQUIRED>
   <!ATTLIST  CrsTaken  CrsCode  IDREF  #REQUIRED>
   <!ATTLIST  ClassRoster  Members  IDREFS  #IMPLIED>
]>
```

Zero or more

Has text content

Empty element, no content

Same attribute in different elements

Another Example

```
<!ELEMENT collection (description,recipe*)>
<!ELEMENT description ANY>
<!ELEMENT recipe (title, ingredient*, preparation, comment?,
   nutrition)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT ingredient (ingredient*,preparation)?>
<!ATTLIST ingredient name CDATA #REQUIRED
                amount CDATA #IMPLIED
                unit CDATA #IMPLIED>
<!ELEMENT preparation (step*)>
<!ELEMENT step (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT nutrition EMPTY>
<!ATTLIST nutrition protein CDATA #REQUIRED
               carbohydrates CDATA #REQUIRED
               fat CDATA #REQUIRED
               calories CDATA #REQUIRED
               alcohol CDATA #IMPLIED>
```

Limitations of DTDs

- → Don't understand namespaces.
- → Very limited assortment of data types (just strings).
- → Very weak wrt consistency constraints (ID/IDREF/ IDREFS only).
- → Can't express unordered contents conveniently.
- → All element names are global: can't have one Name type for people and another for companies, e.g.,

  <!ELEMENT Name (Last, First)>

  <!ELEMENT Name (#PCDATA)>

  can't be in the same DTD

XML Schema

- → Proposed in order to rectify drawbacks of DTDs.
- → Advantages:
  - ✓ Integrated with namespaces;
  - ✓ Many built-in types;
  - ✓ User-defined types;
  - ✓ Has local element names;
  - ✓ Powerful key and referential constraints.
- → Disadvantages: Unwieldy, much more complex than DTDs

XML Query Languages

→**XPath** – core query language. Very limited, a glorified selection operator. Very useful, though: used in XML Schema, XSLT, XQuery, many other XML standards.

→XSLT – a functional document transformation language. Very powerful, *very* complicated.

→**XQuery** – W3C standard. Very powerful, fairly intuitive, SQL-style

→SQL/XML – attempt to marry SQL and XML, part of SQL:2003

Why Query XML?

→Need to extract parts of XML documents.

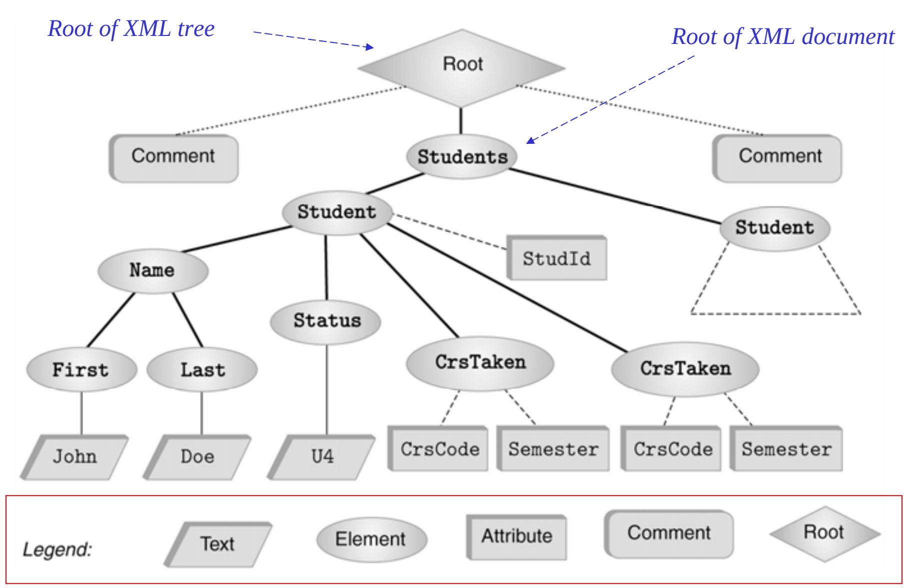
→Need to transform documents into different forms.

→Need to relate – join – parts of the same or different documents.

XPath

- Analogous to path expressions in object-oriented languages (e.g., OQL).
- Extends path expressions with query facility.
- XPath views an XML document as a tree
  - Root of the tree is a ***new*** node, which doesn't correspond to anything in the document
  - Internal nodes are elements;
  - Leaves are either
    - Attributes, Text nodes, Comments;
    - Or other things that we won't discuss (e.g., processing instructions, …)

XPath Document Tree



…and Corresponding Document…

→ A fragment of the report document used earlier:

```
<?xml  version="1.0" ?>
<!-- Some  comment -->
<Students>
    <Student StudId="111111111" >
        <Name><First>John</First><Last>Doe</Last></Name>
        <Status>U2</Status>
        <CrsTaken  CrsCode="CS308"  Semester="F1997" />
        <CrsTaken  CrsCode="MAT123"  Semester="F1997" />
    </Student>
    <Student StudId="987654321" >
        <Name><First>Bart</First><Last>Simpson</Last></Name>
        <Status>U4</Status>
        <CrsTaken  CrsCode="CS308"  Semester="F1994" />
    </Student>
</Students>
<!-- Some other comment -->
```


Terminology

→ ***Parent/child*** nodes, as usual.

→ Child nodes (that are of interest to us) are of types ***text***, ***element***, ***attribute***.

→ ***Ancestor/descendant*** nodes – as usual in trees.

XPath Basics

- →An XPath expression takes a document tree as input and returns a multi-set of nodes of the tree.
- →Expressions that *start* with **/** are ***absolute path expressions***
  - ✓Expression **/** – returns root node of XPath tree;
  - ✓ /Students/Student – returns all Student-elements that are children of Students elements, which in turn must be children of the root;
  - ✓ /Student – returns empty set (no such children at root).
- →The basic idea here is similar to that of directory paths.

More XPath Basics

- →*Current* (or *context* node) – exists during the evaluation of XPath expressions (and in other XML query languages)
- → . – denotes the current node; .. – denotes the parent
  - foo/bar – returns all bar-elements that are children of foo nodes, which in turn are children of the current node;
  - ./foo/bar – same;
  - ../abc/cde – all cde e-children of abc e-children of the *parent* of the current node.
- →Expressions that don't start with / are *relative* (to the current node).

Attributes, Text, etc.

Denotes an attribute

- /Students/Student/@StudentId – returns all StudentId a-children of Student, which are e-children of Students, which are children of the root.
- /Students/Student/Name/Last/text( ) – returns all t-children of Last e-children of …
- XPath provides means to select other document components as well.

Basic Idea and Semantics

- An XPath expression is: locationStep1/locationStep2/…
- *Location step*: Axis::nodeSelector[predicate]
- Navigation *axis*:
  - *child, parent* – have seen;
  - *ancestor, descendant, ancestor-or-self, descendant-or-self* – will see later;
  - some other -- will see later.
- *Node selector*: node name or wildcard; e.g.,
  - ./child::Student (we used ./Student, which is an abbreviation)
  - ./child::* – any e-child (abbreviation: ./*)
- *Predicate*: a selection condition; e.g.,

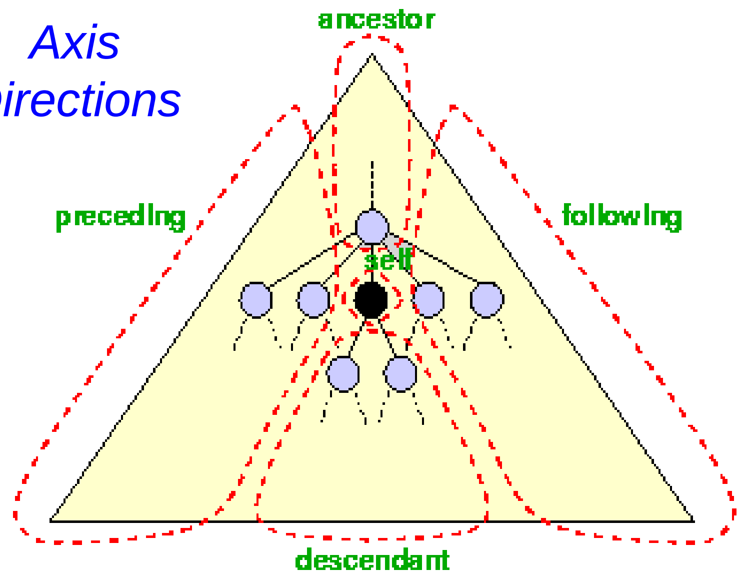
  Students/Student[CourseTaken/@CrsCode = "CSC343"]

This is called *full* (rather than abbreviated) syntax.

Complete Set of Axes

- **Child** — the children of the context node
- **Descendants** — all descendants (children+);
- **Parent** — the parent (empty if at the root)
- **Ancestor** — all ancestors from the parent to the root
- **Following-sibling** — siblings to the right
- **Preceding-sibling** — siblings to the left
- **Following** — all following nodes in the document, excluding descendants
- **Preceding** — all preceding nodes in the document, excluding ancestors
- **Attribute** — the attributes of the context node
- **Namespace** — namespace declarations in context node
- **Self** — the context node itself
- **descendant-or-self** — the union of descendant and self
- **ancestor-or-self** — the union of ancestor and self

Axis Directions



Node Tests

- Testing by node type:
  - text() — chardata node;
  - comment() — comment node;
  - processing-instruction() — processing instruction node;
  - node() — any node (not including attributes and namespace declarations);
- Testing by node name:
  - Name — nodes with that name
  - * — any node

Essential Predicates

- [attribute::name="flour"]: test equality of an attribute
- [attribute::name!="flour"]: test inequality of an attribute
- [attribute::amount='0.5' and attribute::unit='cup']: test two things at once (also or)
- [position()=2]: test position among siblings
- [attribute::amount<'0.5']: a syntax error
- [attribute::amount<'0.5']: a useless test of lexicographical order
- [number(attribute::amount)<number('0.5')]: what you meant to write instead!

An entire location path may be used as a predicate

- [attribute::amount]: the node has an amount attribute
- [descendant::ingredient]: the node has a nested ingredient

XPath Semantics

The meaning of the expression *locStep1/locStep2/…* is the set of all document nodes obtained as follows:

- ✓Find all nodes reachable by locStep1 from the current node;
- ✓For each node *N* in the result, find all nodes reachable from *N* by locStep2; take the union of all these nodes;
- ✓For each node in the result, find all nodes reachable by locStep3, etc.;
- ✓The value of the path expression on a document is the set of all document nodes found after processing the last location step in the expression.

…More Generally…

- →locationStep1/locationStep2/… means:
  - ✓Find all nodes specified by locationStep1
  - ✓For each such node N:
    - Find all nodes specified by locationStep2 using N as the current node
    - Take union
  - ✓For each node returned by locationStep2 do the same using locationStep3, …
- →locationStep = axis::node[predicate]
  - ✓Find all nodes specified by axis::node
  - ✓Select only those that satisfy predicate

More Navigational Primitives

- *Second CrsTaken child of first Student child of Students*:

  /Students/Student[1]/CrsTaken[2]

- *All **last** CourseTaken elements within each Student element*:

  /Students/Student/CrsTaken[last( )]

- *All href attributes in cite elements in the first 5 sections of an article document*:

  child::section[position()<6] / descendant::cite / attribute::href

Wildcards

- Wildcards are useful for unknown document structures.
- The // wildcard descends down any number of levels (including 0):
  - ✓ //CrsTaken – all CrsTaken nodes *under the root*;
  - ✓ Students//@Name – all Name attribute nodes under the elements Students, who are children *under the current node*.
- Note: ./Last and Last are same; but .//Last and //Last are *different.*
- The * wildcard:
  - * – any element: Student/*/text()
  - @* – any attribute: Students//@*

Selection Predicates

- Recall: Location step = Axis::nodeSelector[predicate]
- Predicate:
  - ✓ XPath expression = const | built-in function | XPath expression (equality predicate);
  - ✓ XPath expression (returns false if result is empty);
  - ✓ built-in predicate;
  - ✓ a Boolean combination thereof;
- Axis::nodeSelector[predicate] $\subseteq$ Axis::nodeSelector but contains only the nodes that satisfy predicate.
- Built-in predicates include ones for string matching, set manipulation, etc. Built-in function include large assortment of functions for string manipulation, aggregation, etc.

XPath Queries – Examples

- *Students who have taken CSC343*:

  //Student[CrsTaken/@CrsCode="CSC343"]

- Complex example:

  //Student[Status="U3" and starts-with(.//Last, "A")
  and contains(concat(.//@CrsCode), "ESE")
  and not(.//Last = .//First)]

- Aggregation: sum( ), count( )

  //Student[sum(.//@Grade) div count(.//@Grade) > 3.5]

XPath Queries cont'd

- →Testing whether a subnode exists:
  - ✓//Student[CrsTaken/@Grade] – students who have a grade (for some course)
  - ✓//Student[Name/First or CrsTaken/@Semester or Status/text() = "U4"] – students who have either a first name or have taken a course in some semester or have status U4
- →Union operator, | :
  - ✓//CrsTaken[@Semester="F2001"] | //Class[Semester="F1990"]

union lets us define *heterogeneous* collections of nodes.

XQuery – XML Query Language

- →Integrates XPath with earlier proposed query languages: XQL, XML-QL
- →SQL-style, not functional-style
- →2004: XQuery 1.0

An Example

```
<BOOKS>
  <BOOK YEAR="1999 2003">
    <AUTHOR>Abiteboul</AUTHOR>
    <AUTHOR>Buneman</AUTHOR>
    <AUTHOR>Suciu</AUTHOR>
    <TITLE>Data on the Web</TITLE>
    <REVIEW>A <EM>fine</EM> book.</REVIEW>
  </BOOK>
  <BOOK YEAR="2002">
    <AUTHOR>Buneman</AUTHOR>
    <TITLE>XML in Scotland</TITLE>
    <REVIEW><EM>The <EM>best</EM> ever!</EM></REVIEW>
  </BOOK>
</BOOKS>
```


Some Queries

Titles of all books published before 2000

```
/BOOKS/BOOK[@YEAR < 2000]/TITLE
```

Year and title of all books published before 2000

```
for $book in /BOOKS/BOOK
where $book/@YEAR < 2000
return <BOOK>{ $book/@YEAR, $book/TITLE }</BOOK>
```

Books grouped by author

```
for $author in distinct(/BOOKS/BOOK/AUTHOR) return
  <AUTHOR NAME="{ $author }">{
    /BOOKS/BOOK[AUTHOR = $author]/TITLE
  }</AUTHOR>
```

transcript.xml

```
<Transcripts>
 <Transcript>
  <Student StudId=“111111111”  Name=“John Doe” />
   <CrsTaken  CrsCode=“CS308”  Sem=“F97”  Gr=“B” />
   <CrsTaken  CrsCode=“MAT123”  Sem=“F97”  Gr=“B” />
   <CrsTaken  CrsCode=“EE101”  Sem=“F1997”  Gr=“A” />
   <CrsTaken  CrsCode=“CS305”  Sem=“F1995”  Gr=“A” />
 </Transcript>
 <Transcript>
  <Student StudId=“987654321”  Name=“Bart Simpson” />
   <CrsTaken  CrsCode=“CS305”  Sem=“F1995”  Gr=“C” />
   <CrsTaken  CrsCode=“CS308”  Sem=“F1994”  Gr=“B” />
  </Transcript>
```

… … *cont'd* … …

transcript.xml (cont'd)

```
<Transcript>
  <Student StudId=“123454321”  Name=“Joe Blow” />
   <CrsTaken  CrsCode=“CS315”  Sem=“S97”  Gr=“A” />
   <CrsTaken  CrsCode=“CS305”  Sem=“S96”  Gr=“A” />
   <CrsTaken  CrsCode=“MAT123”  Sem=“S96”  Gr=“C” />
</Transcript>
<Transcript>
 <Student StudId=“023456789”  Name=“Homer Simpson”
  />
  <CrsTaken  CrsCode=“EE101”  Sem=“F1995”  Gr=“B” />
  <CrsTaken  CrsCode=“CS305”  Sem=“S1996”  Gr=“A” />
</Transcript>
</Transcripts>
```

XQuery Basics

- General structure:
  FOR *variable declarations*
  WHERE *condition*
  RETURN *document*

  XQuery expression
- Example:
  (: *students who took MAT123* :)
  FOR $t IN
  doc(“http://uoft.edu/transcript.xml”)//Transcript
  WHERE $t/CrsTaken/@CrsCode = “MAT123”
  RETURN $t/Student

  comment — *See next slide*
- Result:
  <Student StudId=“111111111” Name=“John Doe” />
  <Student StudId=“123454321” Name=“Joe Blow” />

XQuery Basics (cont’d)

- Previous query doesn’t produce a well-formed XML document; the following does:

```
<StudentList>
{
    FOR $t IN doc("transcript.xml")//Transcript
    WHERE $t/CrsTaken/@CrsCode = "MAT123"
    RETURN $t/Student
}
</StudentList>
```

Query inside XML

- FOR binds $t to Transcript elements one by one, filters using WHERE, then places Student-children as *e*-children of StudentList using RETURN.

Doc Restructuring with XQuery

→ *Reconstruct lists of students taking each class using the* Transcript *records*:

```
FOR $c IN distinct(doc("transcript.xml")//CrsTaken)
RETURN
 <ClassRoster CrsCode={$c/@CrsCode}
  Sem={$c/@Sem}>
   { FOR $t IN doc("transcript.xml")//Transcript
     WHERE $t/CrsTaken/[@CrsCode = $c/@CrsCode
      and @Semester = $c/@Sem]
     RETURN $t/Student ORDER BY
      $t/Student/@StudId}
      </ClassRoster>
ORDER BY $c/@CrsCode
```


Document Restructuring (cont'd)

→ *Output elements have the form*:

<ClassRoster CrsCode="CS305" Sem="F1995" > <Student StudId="111111111" Name="John Doe" /> <Student StudId="987654321" Name="Bart Simpson" /> </ClassRoster>

→ *Problem*: the above element ***will be output twice*** – once when $c is bound to

John Doe's

<CrsTaken CrsCode="CS305" Sem="F1995" Grade="A" />

and once when it is bound to

Bart Simpson's

<CrsTaken CrsCode="CS305" Sem="F1995" Grade="C" />

→ *Note*: grades are different – distinct( ) won't eliminate transcript records that refer to same class!

Document Restructuring (cont'd)

- *Solution*: instead of

  FOR $c IN distinct(doc("transcript.xml")//CrsTaken)

  use

  FOR $c IN doc("classes.xml")//Class

  where classes.xml lists course offerings (course code/semester) *explicitly* (no need to extract them from transcript records).

- Then $c is bound to each class exactly once, so each class roster will be output exactly once.

Document on next slide

http://uoft.edu/classes.xml

```
<Classes>
    <Class  CrsCode="CS308"  Semester="F1997" >
        <CrsName>SE</CrsName> <Instructor>Adrian Jones</Instructor>
    </Class>
    <Class  CrsCode="CS305"  Semester="F1995" >
        <CrsName>Databases</CrsName> <Instructor>Mary Doe</Instructor>
    </Class>
    <Class  CrsCode="CS315"  Semester="S1997" >
        <CrsName>TP</CrsName> <Instructor>John Smyth</Instructor>
    </Class>
    <Class  CrsCode="MAR123"  Semester="F1997" >
        <CrsName>Algebra</CrsName> <Instructor>Ann White</Instructor>
    </Class>
</Classes>
```

Document Restructuring (cont'd)

- *More problems*: the previous query will list classes with no students. Reformulation that avoids this:

Test that classes aren't empty

```
FOR $c IN doc("classes.xml")//Class
WHERE doc("transcripts.xml")
          //CrsTaken[@CrsCode = $c/@CrsCode
                    and @Sem = $c/@Sem]
RETURN
<ClassRoster CrsCode = {$c/@CrsCode} Sem= {$c/@Sem}>
  { FOR $t IN doc("transcript.xml")//Transcript
    WHERE $t/CrsTaken[@CrsCode = $c/@CrsCode and
      @Sem = $c/@Sem]
    RETURN $t/Student ORDER BY $t/Student/@StudId
    } </ClassRoster>
  ORDER BY $c/@CrsCode
```


XQuery Semantics

- So far the discussion was informal.
- XQuery ***semantics*** defines what the expected result of a query is.
- Defined analogously to the semantics of SQL.

Evaluate XQuery Queries — Step 1

Produce a list of bindings for variables

- →The FOR clause binds each variable to a *list* of nodes specified by an XQuery expression.
- →The expression can be:
  - ✓An XPath expression;
  - ✓An XQuery query;
  - ✓A function that returns a list of nodes.
- →End result of a FOR clause:
  - ✓Ordered list of tuples of document nodes;
  - ✓Each tuple is a binding for the variables in the FOR clause.

Step 1 — Example

Example (bindings):

- →Let FOR declare $A and $B
- →Bind $A to document nodes {v,w}; $B to {x,y,z}
- →Then FOR clause produces the following list of bindings for $A and $B:
  - ✓$A/v, $B/x
  - ✓$A/v, $B/y
  - ✓$A/v, $B/z
  - ✓$A/w, $B/x
  - ✓$A/w, $B/y
  - ✓$A/w, $B/z

Evaluate Queries — Step 2

- Filter bindings via the WHERE clause -- Use each tuple binding to substitute its components for variables; retain those that satisfy WHERE clause.
- Example:

WHERE $A/CrsTaken/@CrsCode = B/Class/@CrsCode

Binding:

$A/w, where w = <CrsTaken CrsCode="CS308" …/>

$B/x, where x = <Class CrsCode="CS308"… />

Then w/CrsTaken/@CrsCode = x/Class/@CrsCode, so WHERE condition is satisfied & binding retained

Evaluate Queries — Step 3

- Construct result
  - ✓ For each retained tuple of bindings, instantiate the RETURN clause;
  - ✓ This creates a fragment of the output document;
  - ✓ Do this for each retained tuple of bindings in sequence.

User-Defined Functions

- Can define functions, even recursive ones.
- Functions can be called from within an XQuery expression.
- Body of function is an XQuery expression.
- Result of expression is returned; result can be a primitive data type (integer, string), an element, a list of elements, a list of arbitrary document nodes, …

XQuery Functions: Example

- Count the number of *e*-decendants recursively:

Function signature

```
DECLARE FUNCTION  countNodes($e AS
element())  AS  integer {
    RETURN
      IF  empty($e/*)  THEN  0
      ELSE
          sum(FOR  $n  IN  $e/* RETURN
countNodes($n))
            + count($e/*)
            }
```

XQuery expression

Built-in functions sum, count, empty

Grouping and Aggregation

- Does not use separate grouping operator.
  - [OQL does not need one either];
  - Subqueries inside RETURN clause obviate this need.
- Uses built-in aggregate functions count, avg, sum, etc. (some borrowed from XPath).

Aggregation Example

- *Produce a list of students along with the number of courses each student took*:

```
FOR $t IN fn:doc("transcripts.xml")//Transcript,
        $s IN $t/Student
LET  $c := $t/CrsTaken
RETURN
<StudSummary  StudId={$s/@StudId}
Name={$s/@Name}
     TotalCourses = {fn:count(fn:distinct($c))} />
ORDER BY  StudSummary/@TotalCourses
```

- The ***grouping effect*** is achieved because $c is bound to a *new* set of nodes for *each* binding of $t.

Quantification in XQuery

- XQuery supports explicit quantification: SOME (∃) and EVERY (∀).
- Example:

```
FOR $t IN fn:doc("transcript.xml")//Transcript
WHERE SOME $ct IN $t/CrsTaken
          SATISFIES $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

- This is almost equivalent to:

```
FOR $t IN fn:doc("transcript.xml")//Transcript,
      $ct IN $t/CrsTaken
WHERE $ct/@CrsCode = "MAT123"
RETURN $t/Student
```

- ***Not*** quite equivalent, if students can take same course twice!

Implicit Quantification

- In SQL, variables that occur in FROM but not SELECT, are implicitly quantified with ∃. Likewise in XQuery, for variables that occur in FOR, but not RETURN.
- However, XQuery variables are bound to doc nodes:
  - ✓ Two nodes may look textually identical but are still different nodes and thus different variable bindings;
  - ✓ Instantiations of the RETURN expression produced by binding variables to ***different nodes*** are output ***even if these instantiations are textually identical.***
- In SQL a variable can be bound to the same value only once; identical tuples are not output twice (in theory); ***This is why the two queries in the previous slide are not equivalent***

More on Quantification

- *Retrieve all classes (from classes.xml) where each student took MAT123*
- Hard to do in SQL (before SQL-99) because of the lack of explicit quantification.

```
FOR $c IN fn:doc(classes.xml)//Class
LET $g:={(: Transcript records that correspond to class
  $c :)
  FOR $t IN fn:doc("transcript.xml")//Transcript
  WHERE $t/CrsTaken/@Semester = $c/@Semester
        AND $t/CrsTaken/@CrsCode = $c/@CrsCode
  RETURN $t }
  WHERE EVERY $tr IN $g SATISFIES
     NOT fn:empty($tr[CrsTaken/@CrsCode="MAT123"])
  RETURN $c ORDER BY $c/@CrsCode
```